.NET’in ABC’si

.Net ile Programcılığa Giriş

KISIM I: PROGRAMCILIĞA GİRİŞ 9

BÖLÜM 0: BİLGİSAYAR VE YAZILIM 10

Bilgisayarlar ve Programlama Dilleri 10
Programlama Dilleri Seviyeleri 10
Sayı Sistemleri 10
Program Çalışma Süreci 11
Problem Çözme Yaklaşımları 12
Akış Şeması (FlowChart) Nedir? 13
Sözde Kod (PseudoCode) Nedir? 15
Programlama Süreci 16

KISIM II: C# 18

BÖLÜM 0: .NET PLATFORMU 19

.NET'in Getirdiği Çözümler 19
.NET'in Yapıtaşları 20
Ortak Çalışma Zamanı (Common Language Runtime -CLR-) 20
Ortak Tip Sistemi (Common Type System -CTS-) 20
Ortak Dil Spesifikasyonları (Common Language Spesification -CLS-) 21
Temel Sınıf Kütüphanesi (Base Class Library) 21
C# : Geçmişi Olmayan Bir Dil 21
.NET'de Assembly Kavramı 22
Assembly Nedir? 22
Metadata ve Manifesto 23
Tek Dosyalı (Single-File) ve Çok Dosyalı (Multiple-File) Assembly 24
Private Assembly ve Shared Assembly 24
Obfuscator Kavramı 25
Ortak Ara Dilin (Common Intermediate Language) Rolü 25
CIL'in Yararları 27
CIL'in Platforma Özel Koda Derlenmesi 27
.NET Framework Kurulumu 28

BÖLÜM 1: MERHABA DÜNYA 31

Bir C# Programının Yapısı 31
Sınıf (Class) 31
Main Metodu 32
Using Direktifi ve System İsim Alanı (Namespace) 33
Basit Giriş/Çıkış (Input/Output) İşlemleri 35
Console Sınıfı 35
Konsol Çıktısını Formatlama 36
Nümerik Formatlama 37
Uygulamalarda Yorum Satırı 39
C# Komut Satırı Derleyicisi ile Çalışmak 40
Visual Studio 2005 ile Çalışmak 44
Visual Studio 2005 Kullanarak Uygulamaların Derlenmesi 44
Visual Studio 2005 Kullanarak Uygulamaların Çalıştırılması 44
Derleme Zamanı Hataları 44
Çalışma Zamanı Hataları 44
Visual Studio 2005 Hata Ayıklayıcısı Yardımıyla Uygulamanın İzlenmesi 44

BÖLÜM 2: DEĞİŞKEN KAVRAMI 46

Değişkenler 46
Değişken Nedir? Neden İhtiyaç Duyarız? 46
Veri Tipi (Data Types) 46

Değişken Tanımlama ... 47
Değişkenlere Başlangıç Değeri Verme ... 47
Önceden Tanımlı Veri Tipleri ... 48
Ortak Tip Sistemi Türleri: Değer Tipi – Referans Tipi ... 50
Değer Tipleri (Value Types) ... 51
Referans Tipleri (Reference Types) ... 51
Değer Tiplerini Anlamak ... 51
Referans Tiplerini Anlamak ... 53
Değer Tiplerini ve Referans Tiplerini Karşılaştırma ... 59
Değer ve Referans Tipleri Hiyerarşisi ... 60
Kullanıcıdan Alınan Değerler ve Parse() Metodu Kullanımı ... 60
Değişken Kapsama Alanı (Variable Scope) ... 64
Sınıf veya Yapı Üyesi Olarak Değişkenler (Fields) ... 67
Referans Değişkenlerinin Boş (Null) Değer Alması ... 68
Doğru Veri Tipine Karar Vermek ... 68
Değişken İsimlendirme Kural ve Önerileri ... 69
Kaçış Karakterleri (Escape Sequences) ... 70
Tip Dönüşümleri ... 71
Bilinçsiz Dönüşüm (Implicit Conversion) ... 71
Bilinçli Dönüşüm (Explicit Conversion) ... 73
Operatörler ... 76
Aritmetik Operatörler ... 76
Birleşik Atamalar (Compound Assignment) ... 80
Arttırma – Azaltma Operatörleri ... 80
İlişkisel Operatörler ... 82
Koşul Operatörleri ... 83
Eşitlik Operatörleri ... 83

BÖLÜM 3: KULLANICI TANIMLI TİPLER ... 85

Numaralandırıcı (Enumaration) ... 85
Yapı (Struct) ... 89

BÖLÜM 4: KONTROL YAPILARI, DÖNGÜLER, İSTİSNALAR ... 93

Kontrol Deyimleri ... 93
If Kontrolü ... 93
Switch Kontrolü ... 96
Döngü Deyimleri ... 97
While Döngüsü ... 97
Do-While Döngüsü ... 98
For Döngüsü ... 100
Atlama (Jump) Deyimleri ... 103
break Anahtar Kelimesi ... 103
continue Anahtar Kelimesi ... 104
goto Anahtar Kelimesi ... 105
Çalışma Zamanı Hatalarının Yönetimi (Exception Management) ... 107
Kodu Dene (try) ve Hataları Yakala (catch) ... 107

BÖLÜM 5: DİZİLER ve KOLEKSİYONLARA GİRİŞ ... 114

Dizi Nedir? Neden İhtiyaç Duyarız? ... 114
Dizileri Kullanmak ... 114
Dizi Değişkenleri Tanımlamak ... 114
Dizi Örnekleri Oluşturmak ... 114
Dizilere Başlangıç Değerlerini Vermek ... 115
Her Bir Dizi Elemanına Erişmek ... 116
Bir Dizinin Eleman Sayısını Elde Etmek ... 116
Bir Dizi İçerisindeki Bütün Elemanları Elde Etmek ... 117
Dizi Elemanlarını foreach Döngüsü ile Elde Etmek ... 117

Koleksiyonlara Giriş 119
BÖLÜM 6: METOTLAR........ 120
Metot Nedir? Neden İhtiyaç Duyarız? 120
Metot Oluşturma Söz Dizimi 120
Metot Nasıl Yazılır ve Kullanılır ?........ 121
return Anahtar Kelimesi........ 123
return ile Bir Değer Döndürmek........ 124
Parametre Alan Metotlar........ 126
Metot İçerisinde Kullanılan Değişkenler........ 129
Metotlara Parametre Aktarma Yöntemleri 129
Değer Yolu ile 129
Referans Yolu ile 130
Output Yolu ile........ 130
params Anahtar Kelimesinin Kullanımı 131
BÖLÜM 7: PROGRAMLAMA YAKLAŞIMLARI ve SINIF KAVRAMINA GİRİŞ 133
Programlamaya Prosedürel Yaklaşım 133
Prosedürel Programlama Yaklaşımında Sınırlamalar 133
Programlamaya Nesne Yönelimli Yaklaşım (Object Oriented Approach) 134
Avantajları 134
.NET'de Sınıf(Class) ve Nesne(Object) Kavramları........ 134
Sınıf 135
Nesne........ 135
C#'da Sınıf Nasıl Tanımlanır ve Nesne Nasıl Oluşturulur? 135
KISIM SONU SORULARI: 138
KISIM III: SQL SERVER 2005 İLE VERİTABANI PROGRAMCILIĞINA GİRİŞ........ 139
BÖLÜM 0: VERİ TABANINA GİRİŞ........ 140
Veri Tabanı Nedir? 140
Veri Tabanı Yönetim Sistemleri 141
İlişkisel Veri Tabanı Yönetim Sistemleri 141
Tablolar (Tables) 142
Anahtarlar (Keys)........ 144
İndeksler (Indexes) 145
BÖLÜM 1: T-SQL GİRİŞ........ 146
SQL (Yapısal Sorgulama Dili)........ 146
T-SQL (Transact SQL) 146
T-SQL'de Veri Tipleri 146
Metin Veri Tipleri 146
Sayısal Veri Tipleri........ 147
Tarihsel Veri Tipleri........ 147
Diğer Veri Tipleri 147
T-SQL İfade Tipleri........ 148
Veri Tanımlama Dili (Data Definition Language - DDL) 148
Veri Kontrol Dili (Data Control Language - DCL)........ 151
Veri İşleme Dili (Data Manipulation Language - DML)........ 152
T-SQL Sorgulama Araçları 153
SQL Server 2000 Query Analyzer........ 154
SQL Server Management Studio Üzerinde Sorgularla Çalışmak........ 154
BÖLÜM 2: TEMEL VERİ İŞLEMLERİ........ 158
Veri Sorgulama (Select)........ 158
Where İfadesinin Kullanımı 159

Karşılaştırma Operatörleri 160
Mantıksal Operatörler 160
Arama İşlemleri 162
Sıralama İşlemleri (Order By) 163
Veri Tekrarlarını Önlemek (Distinct) 164
Alan İsimlerini Değiştirme (Alias) 165
Literal Kullanımı 166
SQL Server'da Sorguların Çalıştırılması 166
Sorgulamalarda Performans İçin İpuçları 167
Veriyi Gruplamak 168
Belirli Sayıdaki İlk Veriyi Seçmek 168
Gruplama Fonksiyonları (Aggregate Functions) 169
Alan Adına Göre Verileri Gruplamak (Group By) 170
Gruplanan Verilere Şart Eklemek (Having) 171
Gruplanmamış Veriler İçerisinde Gruplama Fonksiyonları Kullanma (Compute) 172
Farklı Tablolardan Veri Getirmek 173
Birden Fazla Tabloyu Birleştirmek 173
Tablolara Temsili İsimler (Alias) Verme 174
Join İfadeleri ile Tabloları Birleştirme 175
İç İçe Sorgular (Subquery) 179
Exists ve Not Exists İfadelerinin Kullanımı 180
Veriyi Güncellemek 181
T-SQL'de Transactionlar 181
Veri Ekleme (Insert) 182
Veri Silme (Delete) 183
Veri Güncelleme (Update) 184

KISIM SONU SORULARI: 186

KISIM IV: ADO.NET 187

BÖLÜM 0: VERİ ve VERİYE ERİŞİM TEKNOLOJİLERİ 188

Veriye Erişim Teknolojileri 188
ODBC (Open Database Connectivity) 188
DAO (Data Access Objects) 188
RDO (Remote Data Objects) 189
OLE DB (Object Linking and Embedding DataBase) 189
ADO (ActiveX Data Objects) 189
ADO.NET 189
Veriye Erişim Yöntemleri 190
ADO.NET Mimarisi 190
System.Data 192
System.Data.SqlClient 192
System.Data.OleDB 192
System.Data.Odbc 192
System.Data.Oracle 192

BÖLÜM 1: SQL SERVER .NET VERİ SAĞLAYICISI (DATA PROVIDER) ORTAK TİPLERİ 195

SqlConnection 195
Örnek Uygulama 198
SqlCommand 199

BÖLÜM 2: BAĞLANTILI (CONNECTED) MODEL 202

SqlDataReader 202
Örnek Uygulama 204

BÖLÜM 3: BAĞLANTISIZ (DISCONNECTED) MODEL 210

SqlDataAdapter 210
DataSet........ 212
Örnek Uygulama........ 214

BÖLÜM 4: PROJE 216

KISIM SONU SORULARI: 222

KISIM V: WINDOWS UYGULAMALARINA GİRİŞ 223

BÖLÜM 0: WINDOWS UYGULAMALARI 224

Windows Uygulamaları Geliştirmek 224
Bir Windows Uygulaması Oluşturmak........ 224
Windows Formlarıyla Çalışmak 227
Form Özellikleri 233
Name 233
Text 233
AcceptButton 234
CancelButton 234
ControlBox 234
FormBorderStyle 235
Opacity 235

BÖLÜM 1: WİNDOWS KONTROLLERİ 236

TextBox 236
MultipleLine, WordWrap ve ScroolBars Özelliği 236
PasswordChar ve MaxLength Özelliği 237
Read-Only Özelliği 238
Button 238
Label 239
CheckBox 240
RadioButton 241
ComboBox 242
DropDownStyle Özelliği 243
Items Özelliği 244
SelectedIndex ve SelectedItem Özellikleri 244
MaxDropDownItems Özelliği........ 246
Sorted Özelliği 246
ComboBox Kontrolüne Veritabanından Veri Eklemek 247
DataGridView 254
Kontroller İçin Ortak Özellikler 258
Text Özelliği 259
Location Özelliği 260
Font Özelliği 260
BackColor Özelliği........ 262
Visible Özelliği........ 263
Cursor Özelliği 264
Dock Özelliği........ 264
RightToLeft Özelliği........ 265

BÖLÜM 2: PROJE 267

KISIM SONU SORULARI: 280

KISIM VI: ASP.NET ile WEB UYGULAMALARI GELİŞTİRMEK 281

BÖLÜM 0: WEB UYGULAMALARINA GİRİŞ 282

Temel Web Teknolojileri........ 282
Web Uygulamalarının Gelişimi 282
XML (eXtensible Markup Language)........ 283

JavaScript 283
HTML (HyperText Markup Language) 283
HTML Formları 285
Web Sayfalarını Programlama 287
HTTP (HyperText Transfer Protocol) 290
IIS (Internet Information Services) 290
IIS'te Sanal Klasörler 291

BÖLÜM 1: MERHABA ASP.NET 296

ASP.NET Mimarisi ve Temelleri 296
ASP.NET Çalışma Modeli 296
Web Sitesi Oluşturma Yolları 296
ASP.NET Dosya Tipleri 298
ASP.NET Klasör Tipleri 298
Visual Studio 2005'te Web Projesi Oluşturma 299
İlk ASP.Net Web Sayfası: Merhaba ASP.Net 300
Web Sayfalarında Event (Olay) Kullanımı 302
Web Sayfalarının Olay Tabanlı Yaşam Döngüsü 303
PostBack Kavramı 306

BÖLÜM 2: WEB KONTROLLERİ 311

Sunucu Kontrolleri (Server Controls) 311
Standart Kontroller (Standard Controls) 311
Veri Kontrolleri (Data Controls) 315
Site içi Dolaşım Kontrolleri (Navigation Controls) 315
Web Sayfalarına Kontrol Ekleme 316
HTML Sunucu Kontrolleri 316
HTML Tablolar 318

BÖLÜM 3: ASP.NET İLE TEMEL VERİ İŞLEMLERİ 321

Veri Kaynakları (Data Sources) 321
SqlDataSource 321
AccessDataSource 325
XmlDataSource 327
Veri (Data) Kontrolleri 327
Repeater 328
DataList 329
GridView 330
DetailsView 333

BÖLÜM 4: DURUM YÖNETİMİ (STATE MANAGEMENT) 336

Oturum Nesnesi ile Durum Yönetimi (Session) 336
Session Nesnesine Veri Eklemek 336
Session Nesnesinden Veri Okumak 337
Uygulama Nesnesi ile Durum Yönetimi (Application) 338

KISIM SONU SORULARI: 342

Tablo 1: 1 byte’dan sonra ölçüler 1024 ve katları şeklinde büyür. 11
Tablo 2: Console sınıfının bazı üyeleri 35
Tablo 3: .NET formatlama stringleri ve anlamları 38
Tablo 4: C# derleyisinin çıktı tabanlı seçenekleri 41
Tablo 5: Önceden tanımlı veri tipleri 48
Tablo 6: Değer tiplerinin karakteristik özellikleri 59
Tablo 7: Referans tiplerinin karakteristik özellikleri 60
Tablo 8: Önceden tanımlı ve kullanıcı tanımlı değer ve referans tipleri listesi 60
Tablo 9: Alan (field)’ların varsayılan değerleri 67
Tablo 10: Genişleyen dönüşümleri mümkün kılan tipler 72
Tablo 11: Daralan dönüşümlerde kullanılabilecek tipler 76
Tablo 12: Matematikte VE ile VEYA operatörlerinin kullanımı 83
Tablo 13: Bolumler ve Ogrenciler tabloları ve içerdiği bilgiler 143
Tablo 14: Sql metin veri tipleri 146
Tablo 15: Sql sayısal veri tipleri 147
Tablo 16: Sql tarihsel veri tipleri 147
Tablo 17: Sql diğer veri tipleri 147
Tablo 18: Sql like için arama karakterleri 162
Tablo 19: Like için örnek sorgu ifadeleri 163
Tablo 20: Sql aritmetik işlem operatörleri 166
Tablo 21: Join seçenekleri 175
Tablo 23: SqlCommand tipinin genel üyeleri 200
Tablo 24: SqlDataReader için genel üyeler 203

KISIM I: PROGRAMCILIĞA GİRİŞ

YAZAR: EMRAH USLU

BÖLÜM 0: BİLGİSAYAR VE YAZILIM

Bilgisayarlar ve Programlama Dilleri

Bilgisayar, mantıksal kararlar alarak aritmetik işlemler yapan bir makinedir. Tabi bu işlemleri çok büyük bir hızla gerçekleştirir. Hiçbir hataya yer vermeden çok sayıda işlemi yapıp, sonuçlarını hafızada saklar. Bilgisayarlar türünden ve tipinden bağımsız olarak iki kısımda incelenir: Donanım (hardware) ve yazılım (software).

Donanım, bilgisayarın fiziksel parçalarına verilen isimdir. Örnek olarak monitör, klavye, harddisk verilebilir.

Yazılım, bilgisayar üzerinde çalışan her türlü programa verilen genel addır. Bir bilgisayar, üzerinde herhangi bir yazılım kurulu olmadan hiçbir özelliğini kullanamaz.

Biz insanlar, iletişim kurmak için yirmi dokuz harf ve 10 rakamdan oluşan bir sistem kullanırız. Tüm sözlü iletişimimiz bu karakterlerin çeşitli varyasyonları ile gerçekleşir. Kurulan cümleler ne kadar uzun olursa olsun, hep bu küme içerisindeki karakterlerden faydalanılır. Bilgisayarların alfabesi ise sadece iki elemandan oluşur: **0** ve **1**. Her türlü işlem için bu iki karakterden oluşan alfabe kullanılır. Sadece bir ve sıfırların farklı formatlarda bir araya getirilmesi ile bütün bu teknolojiler oluşturulur. İşte bu 0 ve 1'leri bir araya getirmek, programcının işidir. Ancak tabi ki 0 ve 1'leri kullanarak değil. Uygulama geliştirici, bir programlama dili ile (C++, Java, C# vb.) kendi anlayacağı biçimde program yazar. Bu yazılanlar, özel programlar yardımıyla bilgisayarın anlayabileceği dile, 0 ve 1 rakamlarına yani **makine dili**ne çevrilir.

Programlama Dilleri Seviyeleri

Program dilleri seviyelerine göre aşağıdaki gibi sınıflandırılabilir:

- Düşük seviyeli diller (Makine dili, assembly dili)
- Orta seviyeli diller (C, C#)
- Yüksek Seviyeli Diller (Visual Basic, Pascal)

Seviye kavramı, bir programlama dilinin makine diline olan yakınlığını temsil eder. Bir dil makine diline ne kadar yakınsa o kadar düşük seviyeli bir dildir; makine dilinden ne kadar uzaksa o kadar yüksek seviyeli bir dildir. Makine diline yakın programlar, bilgisayar üzerinde kullanıcıya daha çok hakimiyet imkanı sunarlar (Bellek işlemleri vb.) Örneğin C++ dili kullanarak bir işletim sisteminin belli bir bölümü yazılabilirken, Visual Basic ile bu neredeyse imkansızdır; çünkü işletim sistemi yazılırken donanım seviyesinde makine ile konuşulması gerekir; bu da ancak düşük seviyeli diller ile mümkün olmaktadır. Bunun yanında orta ölçekli grafik ara yüzü olan bir veri tabanı uygulaması Visual Basic kullanarak 3-4 saatte geliştirilirken, bunu C++ ile yapmak günler alabilir. O yüzden bir dilin seviyesi, diğerinden daha iyi bir dil olduğunu göstermez. Her zaman, projenin ihtiyaçlarına cevap verebilen dil, optimum dildir.

Sayı Sistemleri

Genelde matematik işlemlerinde ve günlük hayatta 10'luk sayı sistemi kullanılır. Bu sayı sistemi 0,1,2,3,4,5,6,7,8,9 rakamlarından oluşur. Bilgisayar ise sadece 0 ve 1 ile çalışır. Dolayısıyla bilgisayarlar 2'lik sayı sistemini kullanırlar. Bu sisteme aynı zamanda **binary** sayı sistemi adı verilir.

Programlama dillerinde kullanılan tüm karakterlerin sayısal bir karşılığı vardır. Hangi karakterin hangi sayıya karşılık geleceği, uluslararası standartlarla belirlenmiştir. ASCII standartlarına göre 256 adet karakter vardır ve 0'dan 255'e kadar numaralandırılmıştır. Örneğin 'f' karakterinin ASCII karşılığı 102 iken 'F' karakterininki 70, '6' rakamının 54'tür. Herhangi bir karakter, makine diline çevrileceği zaman, o karakterin ASCII karşılığı

kullanılır. Buna göre 't' karakterinin makine dilindeki karşılığı, ASCII değeri 116'nın ikilik sistemdeki karşılığı 01110100'dır. Bu ifadedeki bir ve sıfırların her biri **bit** olarak adlandırılır. Bu durumda 01110100 ifadesi 8 bitten oluşmaktadır. Bu bir boyut ölçüsüdür. 8 bit'in bir araya gelmesiyle bir **byte** meydana gelir. Öyleyse 8 bit = 1 byte'dır. ASCII listesindeki her karakter 8 bit(1 byte) ile ifade edilir. Bu durumda alfabemizdeki bütün harfler ve rakamların her biri ikilik sayı sisteminde 8 bitlerle ifade edilir.

Boyut birimleri bit ve byte'dan sonra aşağıdaki gibi katlanarak büyür.

Değer	**Karşılığı**
8 bit	1 byte
1024 byte	1 kilobyte (1 KB)
1024 kilobyte	1 megabyte (1 MB)
1024 megabyte	1 gigabyte (1 GB)
1024 gigabyte	1 terabyte (1 TB)

Tablo 1: 1 byte'dan sonra ölçüler 1024 ve katları şeklinde büyür.

Program Çalışma Süreci

Bir bilgisayar, eğer komutları anlayabilirse bir komut kümesini çalıştırabilir ve gereğini yerine getirebilir. Aynı zamanda bir bilgisayar sadece binary (ikilik) sistemdeki komutları, yani 0 ve 1'leri anlayabilir. Bilgisayarların kullandığı bu dile makine dili adı verilirken, bu dilde her fiziksel işlem için 0 ve 1'lerin aralarında oluşturduğu ayrı bir komut vardır. Bir bilgisayarın anlayabileceği bu komutlar seti (makine dili) bütün makineler için ortak değildir. Makineden makineye değişkenlik gösterir.

Bilgisayarlarda hesaplama motoru olarak **mikro işlemciler (microprocessors)** vardır. CPU (Central Processing Unit –Merkezi İşlem Birimi-) olarak da bilinirler. Bir mikro işlemci, kullanıcı tarafından sağlanan komut kümelerini çalıştırır. Mikro işlemci, komutlara göre kendi aritmetik mantığını kullanarak aşağıdaki üç temel eylemi gerçekleştirir:

- Toplama, çıkarma, çarpma, bölme gibi matematiksel işlemleri gerçekleştir.
- Belli bir bellek bölgesinden başka bir yere veri taşır.
- Koşulları değerlendirir ve değerlendirmenin sonucuna bağlı olarak doğru kod bloğunu çalıştırır.

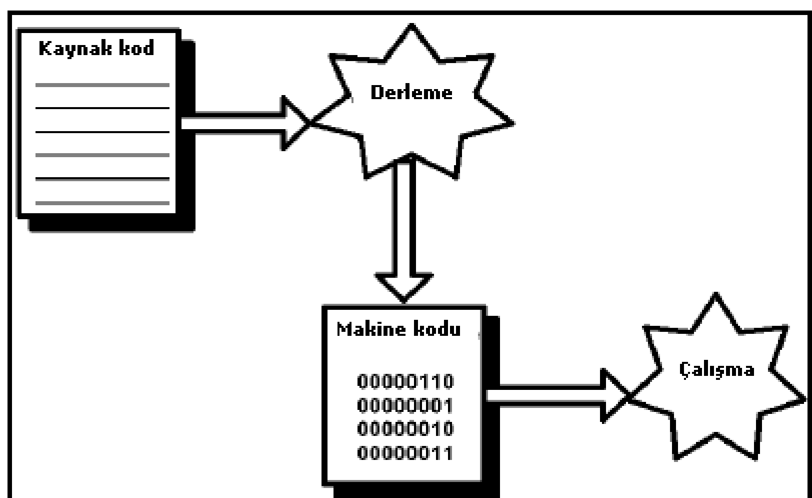
0 ve 1 ler şeklindeki komutları hatırlamak neredeyse imkansız olduğu için, programcılar, uygulama geliştirirken makine dilini kullanmazlar; daha yüksek seviyeli programlama dilleri kullanırlar. Daha yüksek seviyeli programlama dilleri basit ve insan dillerine uygun şekilde komut kümelerine sahiptir.

Bütün programla dillerinin, hepsi belli özel anlamlara sahip kelimeler kümesi olan kendi sözcük hazineleri vardır. Bir programlama dilindeki kelime hazinesinde yer alan kelimelere **anahtar kelime (keyword)** denir. Programlama dilinin dilbilgisine **söz dizimi (syntax)** adı verilir. Birçok farklı programlama dili kullanılarak uygulama geliştirilebilir. Bazı programlama dilleri güncelken bazıları güncelliklerini yitirir. Yazılacak programın karmaşıklığına bağlı olarak programcı, uygulama geliştireceği programlama dilini seçer. Bugünkü popüler programa dilleri C#, Java, Visual Basic, C, C++'dır. Bunlardan C ya da C++ karmaşık dillerdir ve işletim sistemi veya donanım sürücüsü yazılımları geliştirmek için kullanılabilirler.

Editör, program yazmak için kullanılan bir araçtır. Bir editör, basit bir metin düzenleme uygulaması olabilir. Örnek olarak **notepad** verilebilir ya da daha karmaşık bir program da kod yazmak için editör olabilir. Buna örnek olarak ise Visual Studio 2005 gösterilebilir.

Derleyici, uygulamanın yazıldığı programlama dili komutlarını, makine dili komut kümelerine dönüştürebileceği gibi bazı özelleşmiş derleyiciler uygulamanın yazıldığı programlama dilini bir ara dile çevirir. Daha sonra bu ara dili makine koduna çeviren başka derleyiciler de olabilir. Bu şekilde yapılan dönüştürme sayesinde yüksek seviyeli bir

dil ile yazılmış komutların makinenin anlayacağı şekle gelmesi sağlanır. **O yüzden geliştirilen uygulama çalıştırılmadan önce mutlaka derlenmelidir.**



Şekil 1 – Bir programın temel çalışma prensibi

Problem Çözme Yaklaşımları

Bir problem çözmek için bir dizi çözüm adımı takip edilir. Bu adımlar dizisine **algoritma** denir. Bir algoritmayı temsil etmenin iki yolu vardır:

- Akış Şemaları (Flowcharts)
- Sözde kod (Pseudocode)

Bir program yazılmaya başlanmadan önce problemi çözmek için bir dizi mantıksal adım kullanarak bir prosedür yazılır. Bu adımlar;

1) **Programın giriş (input) ve çıkış (output) verileri belirlenir.**
   - Bir problemi çözmek amacıyla çözüm mantığı geliştirmek için öncelikle gerekli çıktılar belirlenir. Belirlenen çıktılara göre de gerekli girdiler belirlenir.

2) **Çözüm için süreçte yapılması gereken işlemler belirlenir.**
   - Girdilerden çıktıları elde edebilmek için gerçekleştirilmesi gereken işlerin belirlenmesi gerekir. Bunu yapabilmek için problem, görevlere bölünür ve **Girdi – Süreç – Çıktı** döngüsüne göre bu görevler sıraya konur. Belirlenen herhangi bir koşullu görev, koşulun sonucuna göre çalıştırılır.

3) **Çözüm mantığı uygulanır.**
   - Girdi ve çıktı gereksinimleri belirlenip süreç hazırlandıktan sonra problem çözüm mantığı bir algoritma ile temsil edilir.

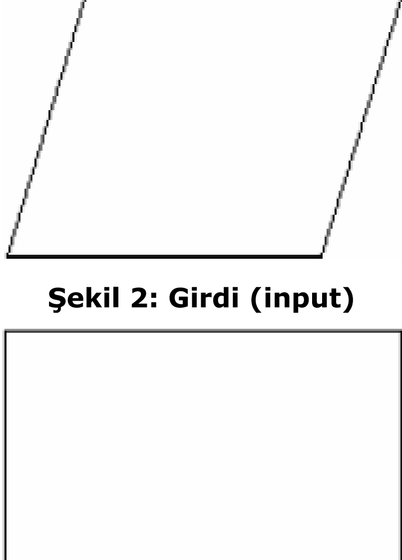
4) **Çözüm mantığı doğrulanır.**
   - Program çözüm mantığının doğruluğundan emin olmak için örnek girdilerle mantığın doğruluğu kontrol edilir. Kodda bazı hatalar tespit edilebilir, gerçekten program yazılmaya başlanmadan önce bunlar düzeltilebilir. Ayrıca bu kontroller yapılırken hem geçerli hem de geçersiz veriler kullanılmalıdır ki uygulamanın her şartta çalıştığı görülsün.

Akış Şeması (FlowChart) Nedir?

Algoritmadaki adımları temsil etmek için akış şemaları kullanılabilir. Bir **akış şeması (flowchart),** algoritmanın grafiksel temsilidir. Semboller kümesinden oluşur. Her sembol özel bir çeşit aktiviteyi temsil eder.

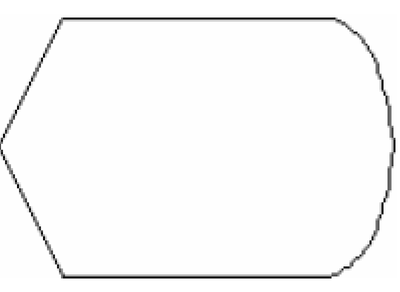
Tipik bir problem çözümleme, giriş verisi (input) kabul etme, alınan girişi işleme ve çıktıyı göstermeyi gerektirir. Bu süreçte programcı bazı kararlar verir. Öyleyse programcı şu aktiviteler için sembollere ihtiyaç duyar: giriş (input) kabul etmek, giriş verisini işlemek (processing), çıktıyı göstermek (output) ve kararlar vermek (decision).

Aşağıda en sık kullanılan akış şeması sembolleri yer almaktadır.

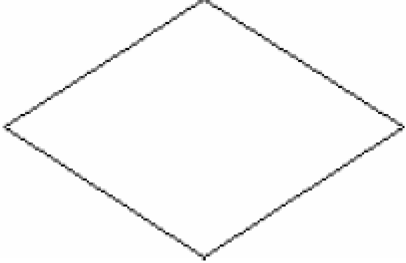


Şekil 2: Girdi (input)

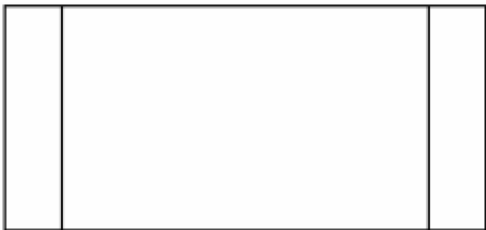
Şekil 3: Giriş verisini işlemek (processing)



Şekil 4: Çıktı (output)

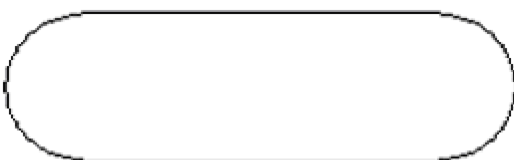


Şekil 5: Karar (decision)

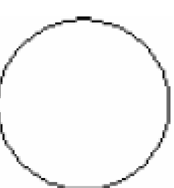


Şekil 6: Prosedür (Procedure)

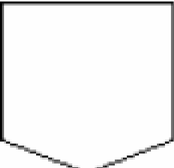
Şekil 7: Akış şemasının adımlarını bağlayan ve sırasını belirleyen akış çizgileri



Şekil 8: Akış şemasının başlangıç ya da bitişini işaret eder (Terminator)

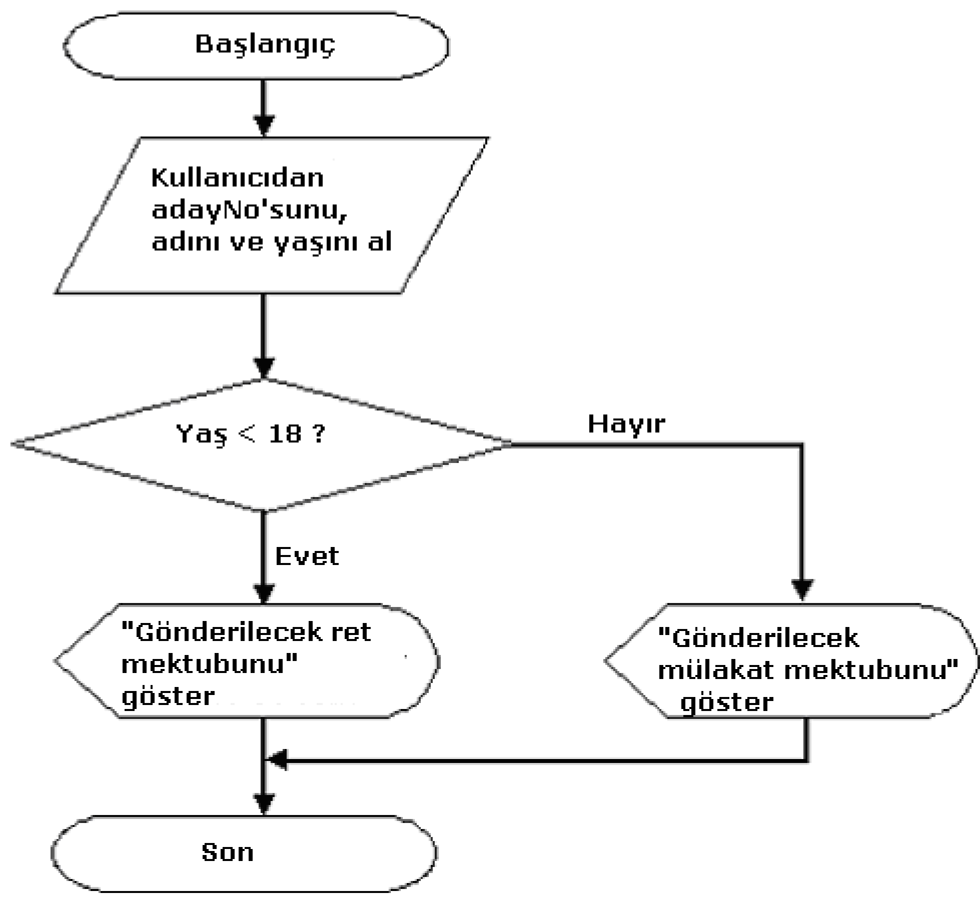


Şekil 9: Akış şemasındaki bir adımı aynı sayfadaki başka bir adıma bağlayan sayfa bağlantısı (on-page connector)



Şekil 10: Akış şemasındaki bir adımı başka sayfadaki diğer bir adıma bağlayan sayfa bağlantısı (off-page connector)

Bu akış şeması sembollerini kullanarak şöyle bir senaryo çerçevesinde bir örnek yapılabilir: Yeni bir pozisyona ilanla eleman arayan bir şirkette çalışıyorsunuz. İlana birçok kişi cevap verdi. İlana başvuranlar arasında 18 yaşın altındakilere red mektubu göndermek ise sizin sorumluluğunuzda. Aynı zamanda uygun adaylarla görüşmek için bir görüşme mektubu da göndermek gerekiyor. Bu senaryoyu programlama ortamına taşımadan önce akış şeması ile algoritmasını çıkarılmak istenirse aşağıdaki gibi olur:



Şekil 11: Basit bir akış şeması

Sözde Kod (PseudoCode) Nedir?

Sözde kod (**pseudocode**), algoritmaların daha kolay anlaşılabilir bir şekilde ifade edilebilmelerini sağlayan bir dildir. Sözde kod programcıya uygulama geliştirme aşamasında belli bir programlama diline ait komutları yazarken bir şema, taslak sağlar.

Sözde kod, ayrıntılı ve okunabilirdir. Sözde kod aşamasında hataları tespit edip çözüme kavuşturmak kolaydır. Sözde kod, doğrulanıp kabul edildikten sonra sanal kod komutları yüksek seviyeli bir programlama diline dönüştürülebilir.

Aşağıda sözde kod yazmak için gereken bazı anahtar kelime ve semboller yer almaktadır:

- **//:** Bu sembol, bir satırın kod değil kod ile ilgili yorum satırı olduğunu işaret etmek için kullanılır. Bir yorum satırı, sanal kod hakkında ekstra bilgi sağlar. Örneğin aşağıdaki yorum satırı, kendisini takip eden sanal kodun iki tane sayının toplamını bulan adımı içereceğini vurgular.
  //İki sayının toplamını bulan kod

- **begin–end:** Bu anahtar kelimeler, bir kod bloğunu işaretlemek için kullanılır. Sözde kodun ilk deyimi her zaman **begin** ile başlar. Son deyim ise her zaman **end** olur.

- **accept:** Kullanıcıdan giriş almak için kullanılır. Örneğin kullanıcıdan adı alındığında aşağıdaki sanal kod kullanılabilir:
  accept cAdi

- **display:** Kullanıcıya çıktı göstermek için kullanılır. Örneğin "Netron'a hoşgeldiniz" mesajını monitörde göstermek için aşağıdaki sanal kod kullanılır:

display “Netron’a hoşgeldiniz”

- **if–else:** Bu anahtar kelimeler, koşulları kontrol etmek ve kararlar vermek için kullanılır.

Aşağıda aldığı iki sayının toplamını bulup kullanıcıya gösteren basit bir uygulamanın sanal kodu yer almaktadır. Kullanılan girişleri (input) temsil eden değerlerin başındaki n, sayısal veriyi temsil eder. Burada bulunmamasına rağmen c ise karakter veriyi temsil eder.

```
begin
numeric nSayi1
numeric nSayi2
numeric nToplam
accept nSayi1
accept nSayi2
nToplam = nSayi1 + nSayi2
display nToplam
end
```

Programlama Süreci

Gerçek hayat uygulamalarında programlama süreci 6 adımda incelenebilir :

1. Problemin tanımlanması (Definition)
2. Gerekli analizlerin yapılması (Analysis)
3. Programın tasarlanması (Design)
4. Programın kodlanması (Coding)
5. Programın değerlendirilmesi ve test edilmesi (Debugging - Testing)
6. Gerekli dokümantasyonun yapılması (Documentation)

Bu adımların nasıl uygulanacağını detaylı bir şekilde belirten yöntemler vardır. Bir yazılım projesine başlamadan önce bu yöntemlerden biri belirlenir ve bu altı adım seçilen yönteme göre takip edilir. Projenin büyüklüğüne bağlı olarak bu altı adımın kaçar gün ya da ay süreceği değişir.

1. Problemin Tanımlanması (Definition)

Yazılan programın amacına ulaşması için, problem doğru bir şekilde tanımlanmalıdır. Eğer problem doğru tanımlanmazsa, işin sonunda çok güzel çalışan ama istenileni yapmayan bir program çıkabilir. Problemin tanımı yapılmadan önce; programcı kendisinden istenileni hiçbir şüphe ve yanlış anlaşılmaya yer vermeyecek şekilde anlamalıdır. Bunun için gerekli tüm sorular sorulmalı ve anlamlı cevaplar alınmalıdır.

2. Gerekli Analizlerin Yapılması (Analysis)

Çözülmesi istenen problem tanımlandıktan sonra, program için gerekli analizler yapılmalıdır. Bu analizin en önemli kısımlarından biri, programı kullanacak olan son kullanıcıların analizidir. Bu analizler tasarım aşamasından önce tamamlanmalı ve program buna göre tasarlanmalıdır. Analizin önemli parçalarından bir diğeri de maliyet analizidir. Bu analiz, projenin başarıyla tamamlanması için gerekli tüm harcamaların önceden planlanması ve projenin bu plana uygun şekilde yönlendirilmesidir. Yanlış maliyet analizi, bir projenin yarıda kalmasına bile sebep olabilir.

3. Programın Tasarlanması (Design)

Gerekli analizler tamamlandıktan sonra artık program tasarlanmaya başlanabilir. Programın tasarlanmasından kastedilen, programı yazmak için kullanılacak olan algoritmanın belirlenmesidir. Algoritma belirlemek dışında programın parçalar halinde nasıl yazılacağı ve programın çalışması sırasında beklenmedik durumlarda yapılacak işlemler de tasarım aşamasında belirlenir.

4. Programın Kodlanması

Kodlama adımında, tasarım adımında belirlenen esaslar uygulanır. Kodların yazıldığı yer bu bölümdür. Tasarım aşamasında yapılan plana ve hazırlanan algoritmaya göre program kodlanır. Bu kitap, özellikle programlama sürecinin 4. adımı üzerinde yoğunlaşmaktadır.

5. Programın değerlendirilmesi ve test edilmesi (Debugging - Testing)

Kodlama kısmı bittikten sonra program çalıştırılır. İlk defa yazılıp çalıştırılan programda hatalar çıkabilir. Programdaki hataları sistemli bir şekilde giderebilmek, hatasız kod yazmaktan daha önemlidir.[1] Hatta çoğu zaman kod yazmaya ayrılan süreden daha fazlası hata ayıklamak için harcanır. Bir programın bir-iki sefer düzgün çalışması her seferinde ve her şartta doğru çalışacağı anlamına gelmez. O yüzden bu süreç için özel test programlarından da faydalanılabilir. Programın çalışırken karşılaşabileceği bütün durumlar düşünülüp oluşabilecek hatalar belirlenerek bu aşamada sistemli bir şekilde giderilir.

6. Gerekli dokümantasyonun yapılması (Documentation)

Dokümantasyon, program hakkındaki detaylı bilgilendirmenin yapıldığı kısımdır. Bu kısımda program hakkında genel bilgiler, programın ne amaç için yazıldığı, programın kullandığı veriler ve verilerin formatları, programın algoritması, çalışma biçimi, test amaçlı kullanılan veriler ve son olarak programın nasıl kullanılacağı belirtilir. Bu işlem programcılar tarafından yapılabileceği gibi programcıların dışında bir çalışan tarafından da yapılabilir. Bu aşama önemlidir; çünkü eğer bir program için dokümantasyon hazırlanmazsa, ileri bir tarihte o programla yeniden uğraşmak durumunda kaldığımızda çok az şey hatırlanacaktır. Herhangi bir doküman da olmadığı için küçük değişiklikler yapmak bile zor olabilir. Ayrıca programı güncelleyecek olan kişinin geliştirme sürecinde yer almayan birisi olma ihtimali de var. Bu durumda dokümantasyonun önemi daha da çok ortaya çıkar. Dokümantasyon hazırlamak ilk başta zaman kaybı gibi görünse de, programlamanın olmazsa olmaz bir bölümüdür.

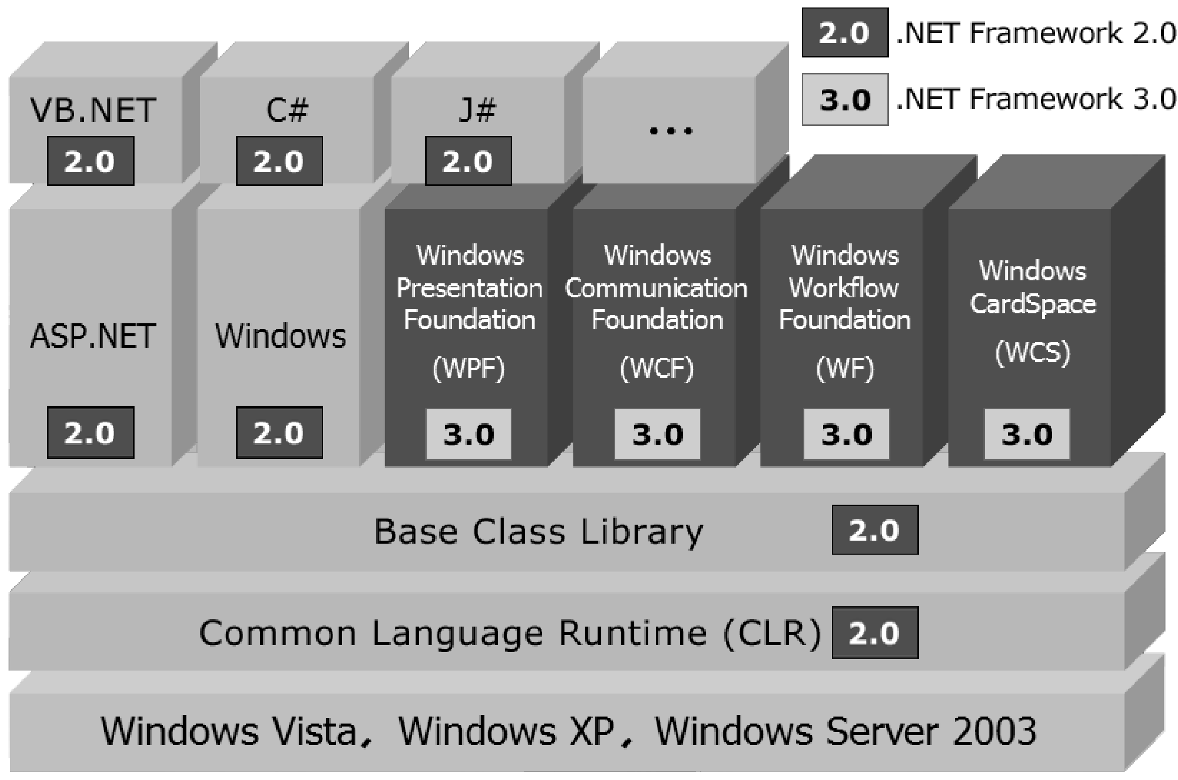
(1) Hatasız kod yazmak ile ilişkili olarak; "Code Complete –Second Edition-, Steve McConell,Microsoft Press" kitabı referans alınabilir.

KISIM II: C#

YAZAR: EMRAH USLU

BÖLÜM 0: .NET PLATFORMU

.NET platformu, Microsoft tarafından geliştirilmiş ve platformdan bağımsız bir şekilde uygulama geliştirilmesini sağlayan bir ortamdır. Sağladığı çoklu dil desteği sayesinde programcıların tek bir dile bağımlı kalmadan (hatta farklı dilleri bir arada kullanmasını sağlayarak) değişik tipte uygulamalar geliştirmelerine olanak sağlar. Masaüstü (Windows, konsol), web, mobil, web servisi, windows servisi, remoting söz konusu uygulama çeşitlerinden bazılarıdır. Microsoft, .NET Framework platformunun 1.0, 1.1, 2.0 sürümlerinin ardından, kitabın yazıldığı tarihlerde 3.0 versiyonunu yayınlamıştır. Framework 3.0, WCF (Windows Communication Foundation), WPF (Windows Presentation Foundation), WF (WorkFlow Foundation) ve CardSpace gibi köklü değişiklikleri içermektedir. Aşağıdaki şekil 2.0 ve şu andaki 3.0 mimarisinin genel hatlarını içermektedir.



.NET'in Getirdiği Çözümler

- **<u>Varolan kodlarla tam çalışabilirlik desteği:</u>** Varolan COM binary'leri[2] ile yeni .NET binary'leri bir arada uyumlu olarak çalışabilirler, ayrıca tam tersi de geçerlidir. Aynı zamanda .NET kodundan C-tabanlı kütüphanelerin çağrılmasına izin verilir.

(2) Geniş bilgi için "RCW (Runtime Callable Wrapper)" ve "CCW (Com Callable Wrapper)" konuları araştırılabilir.

- **<u>Tüm .NET dilleri tarafından paylaşılan ortak bir çalışma zamanı:</u>** .NET ortamında program geliştirirken (kullanılan dilden ve uygulama tipinden -web,masaüstü…- bağımsız olmak üzere) çalışma zamanı prensiplerini belirleyen ve temellerini sağlayan Ortak Çalışma Zamanı (Common Language Runtime), daha önce uygulama geliştiricinin düşünmek zorunda olduğu birçok işin üstesinden gelir (Bellek yönetimi (Memory management), tip güvenliği (Type safety), istisna yönetimi (Exception handling) vb…).
- **<u>Çoklu dil desteği:</u>** Microsoft radikal bir karar alarak CLR ile uyumlu her .NET dilinin kullanılmasına olanak sağlıyor. Visual Studio 2005 ile gelen yazılım geliştirme kitinde C#, VB.NET, J#.NET ve C++.NET kullanarak program geliştirilebiliyor. Öte yandan .NET ortamına entegrasyonu tamamlanmış 50'den fazla programlama diliyle de uygulama geliştirilebilir. (Örnek: Delphi.NET, Perl for .NET…)
- **<u>Tüm .NET dilleri tarafından paylaşılan ortak temel sınıf kütüphanesi:</u>** Artık karmaşık API çağrıları sona erdi. .NET ile birlikte uygulama geliştiricinin hizmetine sunulan 3500'den fazla sınıftan oluşan zengin kütüphane, daha hızlı program geliştirme imkanı ve bütün .NET dilleri tarafından kullanılan tutarlı bir nesne modeli sunuyor.
- **<u>Programlama modelinden bağımsız uygulama geliştirme ortamı:</u>** Tek bir uygulama geliştirme ortamı (Visual Studio 2005) kullanarak ASP.NET, masaüstü form (windows), mobil, web servisi ve remoting uygulamaları geliştirilebilir.
- **<u>Basitleştirilmiş masaüstü uygulama geliştirme ve yayınlama modeli:</u>** .NET ortamında geliştirilen bir masaüstü uygulaması, herhangi bir windows işletim sisteminin kurulu olduğu makinede çalıştırılabilir, gereken tek şey .NET Framework'ünün kurulu olmasıdır. (Linux/Unix işletim sistemleri üzerinde de uygulama geliştirmek için MONO projesi[3] halen devam etmektedir.) Ayrıca sistem kayıt defterine (registry) yazılmasına gerek yoktur. Bunun yanında .NET aynı makinede bir .dll'in farklı versiyonları ile çalışılmasına izin verdiği için ".dll cehennemi" (".dll hell") adı verilen durum oluşmamaktadır.

.NET'in Yapıtaşları

.NET'in sağladığı bazı avantajlar incelendi, şimdi de bu avantajları hayata geçirmek için gereken (birbirleriyle bağlantılı) üç yapıtaşı incelenecek: **CLR**, **CTS** ve **CLS**. Yazılım geliştiricinin bakış açısıyla .NET yeni bir çalışma zamanı ve çok yönlü bir temel sınıf kütüphanesi olarak görülebilir.

Ortak Çalışma Zamanı (Common Language Runtime -CLR-)

Çalışma zamanı ortamı, **Common Language Runtime (ortak çalışma zamanı)** olarak adlandırılır ve CLR kısaltmasıyla anılır. CLR'ın birincil rolü .NET tiplerinin yerini öğrenmek, bu tipleri kendi ortamına yüklemek ve yönetmektir. CLR ayrıca bellek yönetimi ve tip güvenlik kontrollerini yerine getirmek gibi birçok alt seviye ayrıntıdan da sorumludur.

Ortak Tip Sistemi (Common Type System -CTS-)

.NET platformunun bir diğer yapıtaşı **Common Type System (ortak tip sitemi)**'dir ve kısaca CTS olarak anılır. CTS spesifikasyonları, çalışma zamanı tarafından desteklenen bütün veri tipleri ve programlama yapılarını tanımlar, bu yapıların birbirleriyle nasıl etkileşeceklerini açıkça belirtir ve .NET metadata formatında nasıl temsil edileceklerinin ayrıntılarını belirler (metadata hakkında ayrıntılı bilgiye bir sonraki bölümde erişilebilir). Böylece .NET destekli tiplerin, aynı veri tiplerini kullanabilmesi sağlanabilmektedir.

(3) Detaylı bilgi için http://www.mono-project.com sitesi ziyaret edilebilir.

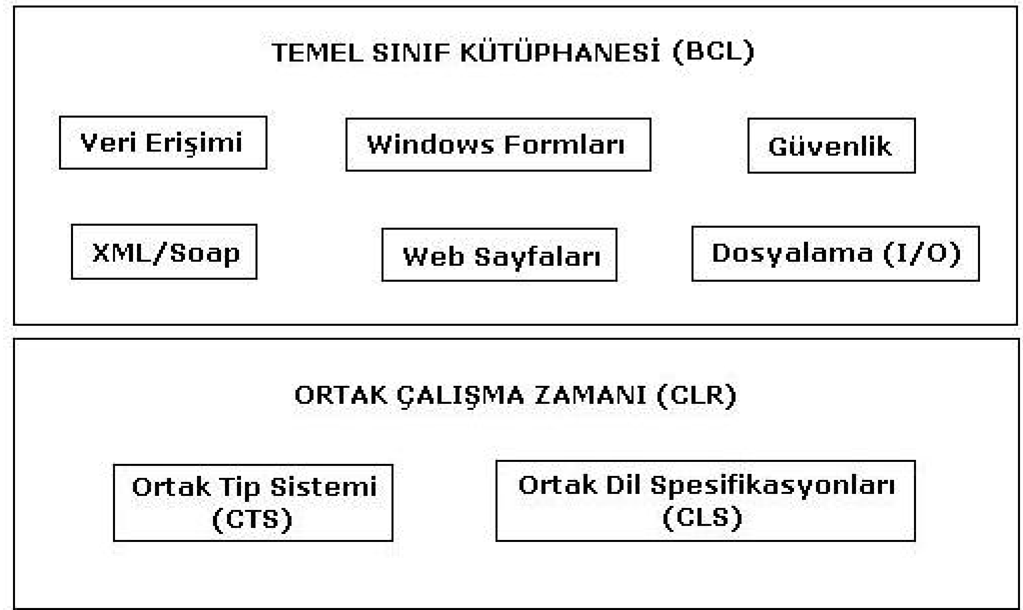
Ortak Dil Spesifikasyonları (Common Language Spesification -CLS-)

.NET tabanlı bir programlama dili, ortak tip sistemi CTS tarafından tanımlanan her bir özelliği desteklemeyebilir. **Common Language Specification (ortak dil spesifikasyonu),** tüm .NET dillerinin orta noktada buluşabileceği ortak tip ve programlama altyapısını tanımlayan yönerge listesidir. Dolayısıyla CLS uyumlu özelliklere sahip bir .NET tipi geliştirilirse, bu tipi diğer bütün .NET dillerinin kullanabileceği garanti edilmiş olur. Tersine CLS sınırlarının dışında bir veri tipi ya da programlama yapısı oluşturulursa, bu kod kütüphanesi ile bütün .NET dillerinin sağlıklı bir şekilde etkileşebileceği garanti edilemez.

Temel Sınıf Kütüphanesi (Base Class Library)

CLR ve CTS/CLS spesifikasyonlarına ek olarak .NET platformu, tüm .NET programlama dillerinin kullanabileceği **Base Class Library'i (temel sınıf kütüphanesi)** sunar. Her biri belli bir görevi yerine getirmekle sorumlu olan sınıflardan oluşan bu kütüphane hem temel işler (thread -kanal-, dosya giriş çıkış,grafiksel görünüm…) için kullanılacak tipleri içerir hem de gerçek hayat uygulamalarının ihtiyaç duyacağı birçok servise destek sağlar. Örneğin temel sınıf kütüphanesinin bize sağladığı tipler veritabanı erişimi, xml etkileşimleri, programatik güvenlik konularını ele almayı ve web,masaüstü ya da konsol tabanlı kullanıcı ara yüzleri geliştirmeyi çok kolaylaştırır.

.NET platformunun yazılım geliştiricilere sunduğu bu koleksiyonun güzel yanlarından biri de kullanılmasının son derece kolay olmasıdır. Kullanılan isimler o kadar açıklayıcıdır ki temel ingilizce bilgisine sahip birisi için sezgisel olarak hangi üyenin kullanacağının bulunması çok fazla zaman almamaktadır.



Şekil 12: .NET Framework temel bileşenleri : CLR,CTS,CLS ve BCL

C# : Geçmişi Olmayan Bir Dil

.NET'in önceki teknolojilere göre ne kadar radikal bir proje olduğunun göstergesi olarak Microsoft, bu yeni platforma özel yeni bir programlama dili geliştirdi. Bu dil modern nesne yönelimli dizayn metodolojisi üzerine kuruldu ve Microsoft C#'ı geliştirirken yıllardır nesne yönelimli prensiplere sahip benzer dillerden elde ettiği tecrübelerden faydalandı. Sonuç olarak ortaya sözdizimi son derece temiz, öğrenmesi ve yazması kolay, ayrıca güçlü ve esnek bir dil çıktı.

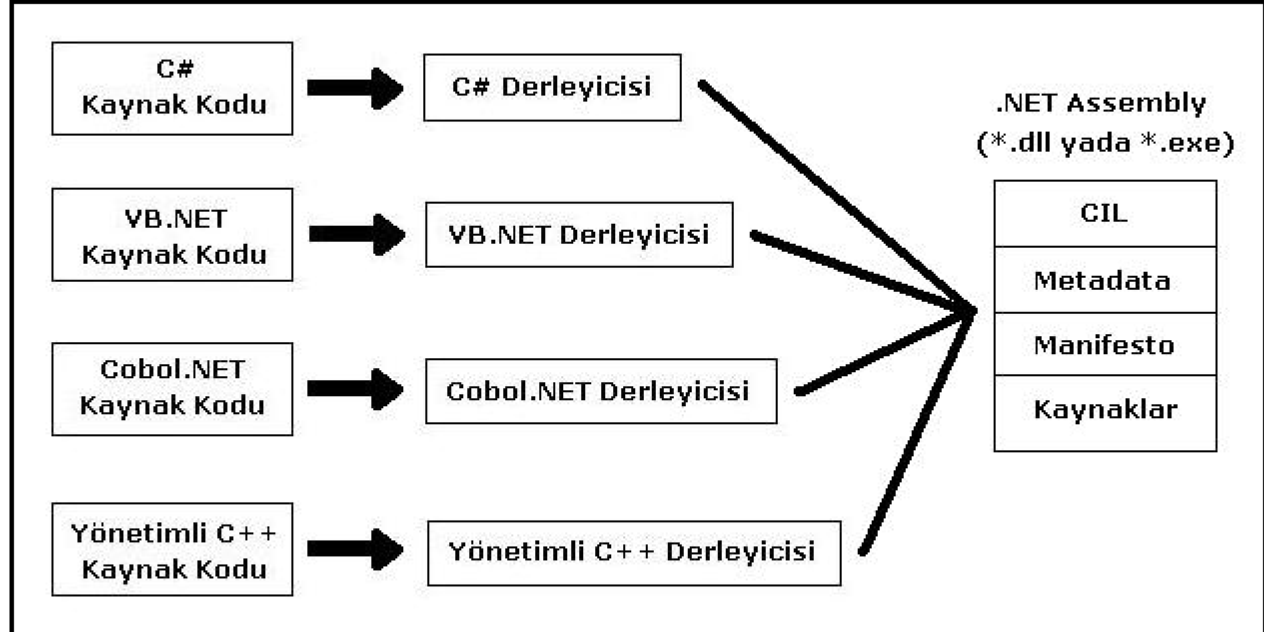
C#'ın Microsoft.NET platformu ile gelmesi ile ilgili olarak anlaşılması gereken önemli noktalardan biri, sadece .NET çalışma zamanında çalışacak kod üretmesidir. C# hiçbir zaman COM ya da Win32 API uygulaması geliştirirken kullanılamaz. Teknik bir ifadeyle .NET çalışma zamanında işlenecek kodu tanımlayan terim **managed code** (yönetimli kod) ; yönetimli kod içeren binary birimin .NET dünyasındaki terim karşılığı ise **assembly** dir. Aksine doğrudan .NET çalışma zamanı tarafından işlenmeyen kod ise **unmanaged code** olarak adlandırılır. Örneğin C,C++ gibi diller yardımıyla geliştirilebilirler.

.NET *.dll'leri, istenirse COM nesneleri haline getirilip yönetimli (managed) kod tarafından, yönetimsiz (unmanaged) kod tarafına geçirilebilir. (CCW) Bu sürecin tam tersi de geçirilebilir. (RCW)

.NET'de Assembly Kavramı

Assembly Nedir?

Hangi .NET dilinin kullanıldığından bağımsız olarak .NET binary'leri bir dosya uzantısı alırlar (*.dll ya da *.exe). Burada dikkati çekilmesi gereken nokta bu binary'lerin işletim sistemine özel komutlar içermemesidir. Bunun yerine .NET binary'leri, platformdan bağımsız **Common Intermediate Language(CIL)** adındaki ara dili içerirler.



Şekil 13: Tüm .NET derleyicileri kaynak kodu assembly içerisinde CIL'e derler.

IL kısaltması ile ilgili bir noktaya dikkat çekmek gerekiyor. .NET'in geliştirilme aşamasında IL'in resmi adı "Microsoft intermediate language" (MSIL) iken final sürümüyle birlikte bu terim "Common intermediate language" (CIL) olarak değiştirildi. Dolayısıyla kaynaklarda geçen MSIL ve CIL aynı kavramı işaret etmektedir.

.NET ortamındaki bir programlama dilinin derleyicisi kullanılarak bir .dll ya da .exe oluşturulduğunda bu bir **assembly** içerisine koyulur. Daha önce bahsedildiği gibi assembly, CIL kodu içerir ve bu kod, ihtiyaç duyulana kadar platforma özel bilgilere derlenmez. Burada ihtiyaç duyulan nokta ile kastedilen, .NET çalışma zamanı tarafından kullanılmak üzere başvurulan IL kod bloğudur (Metot çağrısı gibi)

Şunu da eklemek gerekir ki binary dosya (.exe ya da .dll uzantılı) ile assembly tektir ve aynı kavramlardır; alt seviye bir programlama dili olan Assembly ile sadece isim

benzerliği vardır. Ayrıca çalıştırılabilir kod (.exe) ve kütüphane kodu (.dll) aynı assembly yapısına sahiptir ve terim olarak her ikisi de assembly olarak çağrılır. Aralarındaki tek gerçek fark çalıştırılabilir assembly (.exe) program için ana giriş noktası (Main metodu) içerirken, kütüphane assembly'si (.dll) bunu içermez.

Metadata ve Manifesto

Assembly, CIL dışında bir de **metadata** içerir. **Tip metadata**, binary içerisindeki her bir tipin ayırt edici özelliklerini tanımlar. Daha aydınlatıcı olması açısından şöyle bir örnek verelim: Mevcut evimizden başka bir eve taşınırken eşyaları kutulara koyarız. Eğer kutuların üzerine tek tek içerisinde neler olduğunu bir kağıda yazıp yapıştırırsak yeni evimizde açarken aradığımızı kolayca buluruz. Metadata, kutunun üzerindeki bu listedir ve assembly içerisindeki .dll ya da .exe'nin sahip olduğu tüm tipler hakkında bilgi içerir. Örneğin X adında bir sınıf varsa, tip metadası X sınıfının türediği sınıf, varsa hangi arayüzleri uyguladığı gibi ayrıntıları taşırken, X tipinin her bir üyesinin tüm tanımlamalarını da içerir.

Bir assembly'de aynı zamanda kendisini tanımlayan bilgileri içeren **assembly metadata** bölümü bulunur. Assembly adı, versiyonu, kültür bilgisi, kısa bir açıklama, başka assembly'lere olan referanslar gibi bilgilerin tamamına assembly metadata denir ve **manifesto** adıyla assembly içerisinde yer alır. Bir assembly'nin assembly metadatası taşımasının arkasındaki gerçek, içerisindeki kodu çağıran uygulama ya da diğer assembly'lerin o assembly'yi nasıl kullanacaklarını öğrenmek için registry'ye ya da başka bir veri kaynağına başvurmalarına gerek kalmamasıdır[4].

Öyleyse bir assembly içerisindeki kaynak kodun CIL karşılığı, tip metadatası, manifesto bilgisi ve kaynaklar yer alır.

CIL C# Kaynak Kodunun derlenmesi sonucu oluşan Intermadiate Language komutları
METADATA Bu assembly içerisindeki üyelerin listesi
MANIFESTO Bu assembly'nin kendisine erişimini sağlayan ve kendisini tanımlayan bilgileri
KAYNAKLAR Assembly'deki üyelerin kullandığı yönetimsel olmayan kütüphaneler, dosyalar vb.

Şekil 14: Dört bileşeni ile bir .NET assembly'si

(4) Assembly'ların kendi içeriklerini bilmeleri, çalışma zamanında bu bilgilerin elde edilip kullanılabileceği anlamına gelir ki, bu plug-in tabanlı programlama içerisinde önem arz eden bir konudur ve reflection (yansıma) olarak isimlendirilir. Hatta Visual Studio 2005 uygulama geliştirme ortamının intelli-sense özelliği bu kavramdan faydalanarak çalışır.

Tek Dosyalı (Single-File) ve Çok Dosyalı (Multiple-File) Assembly

.NET ortamında geliştirdiğimiz uygulamalar varsayılan olarak tek bir assembly'den oluşurlar. Bir assembly tek bir .dll ya da tek bir .exe modülünden oluşuyorsa ona **Single-File Assembly** denir. Bu şekildeki tek-dosya assembly'ler gerekli bütün ortak ara dil yönergeleri (CIL),metadata ve manifesto bilgilerini benzersiz,tek bir pakette tutar. **Multi-File Assembly'**ler ise birden fazla assembly'nin bir araya gelmesiyle oluşur ve bu durumda her birine **modül** adı verilir. Çok-dosyalı assembly oluşturulurken modüllerden bir tanesi (Birincil modül –primary module-) manifesto bilgisini içerir. Diğer modüller ise, modül seviyesinde kendi manifesto,CIL ve metadata bilgilerini taşımaya devam ederler. Bu bilgilerden anlaşılacağı gibi birincil modül manifestosu, diğer modüllere ne zaman başvurulacağı, modüller içerisindeki tipler ve diğer dosyalar (kaynaklar) gibi bilgileri sunar. Ayrıca tüm assembly'ler farklı .NET dilleri kullanılarak yazılmış parçalar olabilir ve çok dosyalı assembly mimarisi ile bu uygulama parçaları tek bir assembly altında birleştirilerek kullanılabilir.

Private Assembly ve Shared Assembly

.NET ortamında assembly'ler private (özel) ve shared (paylaşılan) olmak üzere ikiye ayrılırlar:

Private (özel) assembly'ler çalıştırılabilmeleri için, geliştirilen uygulama ile birlikte aynı klasörde yer almalıdır. Bu yolla assembly'nin sadece o uygulama ile birlikte çalışması amaçlanır. Varsayılan olarak oluşturulan assembly'ler private'dır. Çalışma zamanı, private assembly'lerin diğer uygulamalar tarafından kullanılmayacağını garanti eder; çünkü bir uygulama, private assembly'yi sadece çalıştırılabilir dosyanın (.exe) çağrıldığı dizin ya da bir alt dizini ile aynı dizinden çağırabilir.

Aynı makinede çalışan birden çok .NET uygulamasının ortak bir dizinde bulunması senaryosunu düşündüğümüzde dahi bir uygulamanın yanlışlıkla başka bir uygulamanın kullandığı private assembly'lerin üzerinde modifiye yapması ya da kendi assembly'si yerine bir diğerini çağırması gibi riskler söz konusu değildir. Private assembly'lerin sadece kendisini kullanan uygulama tarafından erişilmesi çok fazla esneklik sağlar. Örneğin bir uygulama başka bir uygulamanın kullandığı assembly'nin yeni versiyonunu kullanırken diğer uygulama eski versiyonla çalışmaya devam edebilir.

Private assembly'nin bu özelliği ile dağıtılması da çok kolaydır. Dizin sisteminde uygun klasöre uygun dosya ya da dosyaların yerleştirilmesi yeterlidir (Yapılması gereken registry girişi vb. ayrıntılar yoktur).

Shared (paylaşılan) assembly'ler ile aynı makinedeki bütün uygulamaların kullanabilmesi için ortak bir kütüphane oluşturmak amaçlanır. Öyleyse aynı bilgisayardaki shared assembly'lere diğer uygulamalar erişebileceği için aşağıdaki risklere karşı daha dikkatli davranılmalıdır:

- *İsim çakışmaları:* X uygulamasının kullandığı shared assembly'deki tiplerin isimleri, Y uygulamasının kullandığı başka bir shared assembly'dekilerle aynı olabilir. Client tarafındaki kodlar teorik olarak aynı anda her iki assembly'ye de erişebildiği için,bu büyük bir problem olabilir.
- *Versiyon problemi:* Bir assembly'nin farklı bir versiyonu üzerine kaydedilebilir ve böylece yeni versiyon, varolan bazı kodlarla uyuşmazlık yaşayabilir.

Kullanıma açık assembly'ler dosya sisteminde özel bir alt-dizine yerleştirilir, bu dizin **Global Assembly Cache**'dir ve assembly'nin özel olarak bu cache'e atılması gereklidir. Erişim adına oluşabilecek yukarıda belirtilen riskleri önlemek için shared assembly'lere özel şifre kriptografisine sahip bir isim verilir. Bu isim **strong name** olarak bilinir, o

assembly'nin tekliğini garanti eder ve assembly'yi referanse etmek isteyen uygulamalar tarafından bilinmesi gerekir.

Global Assembly Cache'in Windows işletim sistemli makinelardaki varsayılan yolu şudur: **C:\WINDOWS\assembly**

Obfuscator Kavramı

.NET mimarisi kullanılarak geliştirilen assembly'lerin CIL içerikleri ILDASM (Intermadite Language Disassembler) aracı ile görülebilir, pek çok faydalı bilgiye erişilebilir. Bu uygulama geliştirici için avantaj olsa da binary dosyalara ulaşan birisi reverse – engineering işlemleri ile CIL kodunu herhangi bir .NET diline çevirebileceği için dezavantaj da oluşturabilir. Bu işi yapan araçlardan birini de Türkiye'den bir yazılım geliştirici yazmıştır. Kötü niyetli kullanıcılar uygulamadaki varsa güvenlik sorunlarını bularak bunlardan faydalanabilir ya da kullanılan orijinal fikirleri çalabilir. Bu noktada reverse-engineering ile okunduğunda kaynak kodun anlamsız olarak görüntülenmesini hedef alan **obfuscation** metodu geliştirilmiştir. Bu işi yapan araca da **obfuscator** denilmektedir.

Obfuscation, uygulama kodlarının çalışmasını etkilemeden kaynak kodları saklamaktır ve kodlara değil .NET assembly'lerine uygulanan bir metottur. Bu metot içerisinde kullanılan tekniklerden bazıları şunlardır: Metadata içerisindeki adları anlamsız olarak yeniden isimlendirmek, çalışma zamanının kullanmadığı metadata bilgilerini silmek, string şifreleme, kontrol deyimlerinin karıştırılması vb… Visual Studio 2005 içerisinde bu iş için Dotfuscator Community Edition aracı bulunmaktadır[5].

Ortak Ara Dilin (Common Intermediate Language) Rolü

.NET assembly'leri hakkında elde edilen bilgiler ışığında platformun ortak ara dilinin (CIL) rolü biraz daha ayrıntılı incelenebilir. CIL, herhangi bir platform-özel direktif setinin yerine geçmiştir. Seçilen .NET tabanlı dilin hangisi olduğundan bağımsız, ilgili derleyici CIL direktifleri üretir. Örnek olarak aşağıdaki C# kodu basit bir hesap makinesini modelliyor. Şimdilik sözdizimi ile çok ilgilenmeye gerek yok; ancak HesapMak sınıfının içerisindeki Topla() metodunun formatına dikkat:

```
//Hesap.cs
using System;
namespace HesapMakinesiOrnek
{
    //C# hesap makinesi
    public class HesapMak
    {
        public int Topla(int x, int y)
        {
            return x + y;
        }
    }
    //Bu sınıf, programın giriş noktasını içerir.
    class HesapMakUygulamasi
    {
        static void Main(string[] args)
        {
            HesapMak hesap = new HesapMak();
            int cevap = hesap.Topla(23, 41);
            Console.WriteLine("23 + 41 = {0}", cevap);
            Console.ReadLine();
        }
    }
}
```

(5) Obfuscate işleminin dezavantajları da vardır. Debug işlemlerinin zorlaşması, performans kaybı, reflection API'si ile ters düşme bu dezavantajlar arasında sayılabilir.

C# derleyicisi (csc.exe) ile bu kaynak kod dosyası derlendiğinde;

- CIL direktifleri
- Manifest
- HesapMak ile HesapMakUygulamasi sınıflarının her ayrıntısını tanımlayan metadata bilgisi

içeren tek dosyalı (single-file) bir *.exe assembly elde ederiz. Bu assembly ildasm.exe ile açılırsa Topla() metodunun CIL kullanılarak aşağıdaki gibi temsil edildiği görülür. (Visual Studio 2005 Tools altındaki Visual Studio 2005 Command Prompt'a ildasm.exe yazılarak bu araç açılabilir ve ilgili .dll ya da .exe'nin yolu gösterilir) :

```
.method public hidebysig instance int32  Topla(int32 x,
                                               int32 y) cil managed
{
  // Code size       9 (0x9)
  .maxstack  2
  .locals init ([0] int32 CS$1$0000)
  IL_0000:  nop
  IL_0001:  ldarg.1
  IL_0002:  ldarg.2
  IL_0003:  add
  IL_0004:  stloc.0
  IL_0005:  br.s       IL_0007
  IL_0007:  ldloc.0
  IL_0008:  ret
} // end of method HesapMak::Topla
```

Bu metot için üretilen CIL kodundan pek bir şey anlaşılmaması çok önemli değildir. Burada üzerinde durulması gereken nokta şu ki; C# derleyicisi işletim sistemine özel kodlar değil, CIL üretir.

Tekrarlamakta fayda var, bu özellik bütün .NET tabanlı derleyiciler için geçerlidir. Bunu gösterebilmek için aynı uygulamanın C# yerine Visual Basic .NET (VB.NET) kullanılarak yazıldığını varsayalım :

```
'Hesap.cs
Imports System

Namespace HesapMakinesiOrnek
    'Bir VB.NET 'Modülü'sadece static üyeler içeren bir sınıftır.
    Module HesapMakUygulamasi
        Sub Main()
            Dim hesap As New HesapMak
            Dim sonuc As Integer
            sonuc = hesap.Topla(23, 41)
            Console.WriteLine("23 + 41 = {0}", sonuc)
            Console.ReadLine()
        End Sub
    End Module

    //C# hesap makinesi
    Class HesapMak
        Public Function Topla(ByVal x As Integer, ByVal y As Integer) As
Integer
            Return x + y
        End Function
    End Class
End Namespace
```

Eğer Topla() metodunun CIL direktiflerine bakacak olursak C# kaynak kodu için üretilenlerin VB.NET için üretilenlerle aynı olduğunu görürüz.

```
.method public instance int32  Topla(int32 x,
                                     int32 y) cil managed
{
  // Code size       9 (0x9)
  .maxstack  2
  .locals init ([0] int32 Topla)
  IL_0000:  nop
  IL_0001:  ldarg.1
  IL_0002:  ldarg.2
  IL_0003:  add.ovf
  IL_0004:  stloc.0
  IL_0005:  br.s       IL_0007
  IL_0007:  ldloc.0
  IL_0008:  ret
} // end of method HesapMak::Topla
```

CIL'in Yararları

Bu noktada kaynak kodun platforma özel komut setine (makine diline) değil de CIL'e derlenmesinden tam olarak ne elde edildiği merak edilebilir. Bunun cevabı öncelikle dil entegrasyonudur. Az önce de gördüğümüz gibi bütün .NET tabanlı derleyiciler neredeyse aynı CIL kodunu üretiyorlar. Dolayısıyla bütün diller iyi-tanımlanmış bu binary ortamda aralarında anlaşabileceklerdir. Terimler şu an için yabancı gelebilir; ancak aşağıdaki örnekler verilebilir

- Bir dilde yazılmış sınıf, başka dilde yazılmış bir sınıfın üyelerini kalıtım yoluyla kullanabilir.
- Bir sınıf hangi dilde geliştirildiğinden bağımsız olarak başka bir sınıf örneğini içerebilir.
- Nesneler ya da nesne referansları metotlar arasında parametre olarak geçirilebilir.
- Farklı dillerde yazılmış metotları çağırırken hata ayıklayıcı ile metot çağrımları arasında gezilebilir; yani farklı dillerdeki kaynak kodları arasında adım adım ilerlenebilir.

Ayrıca CIL'in platformdan bağımsız olmasından yola çıkarak .NET Framework'ün de platformdan bağımsız olduğunu söyleyebiliriz. (Tek bir kod bloğunun sayısız işletim sistemi üzerinde çalışabilmesi) Bu platform-bağımsızlık Windows işletim sistemlerinde geçerli olmakla beraber bir-çok Windows olmayan platformda deneme aşamasında da olsa uygulamaları mevcuttur. (Mono ve Portable.NET projeleri) Özet olarak CIL'in getirdiklerine bakılınca işin en güzel yanı .NET'in yazılım geliştiriciye hangi dil ile bunu yapmak istediği seçeneğini sunmasıdır.

CIL'in Platforma Özel Koda Derlenmesi

Herhangi bir dilde yazılmış uygulamanın bir bilgisayarda çalışması için mutlaka bilgisayarın anlayacağı komutlara dönüştürülmesi gereklidir. .NET assembly'leri platforma özel kod yerine CIL içerdiğinden dolayı kullanılmadan önce makine koduna derlenmelidirler. CIL'i uygulamanın çalıştığı makine için anlamlı talimatlara dönüştüren derleyiciye **just-in-time compiler (tam zamanında derleyici)** adı verilir ve aynı zamanda **jitter** olarak da anılır. .NET çalışma zamanı ortamı, her biri üzerinde çalıştığı işletim sistemi için optimize edilmiş, her CPU için bir JIT derleyicisi kullanır ve bu optimizasyonu otomatik yapar.

Örneğin cep bilgisayarı gibi küçük akıllı cihazlarda yayınlanmak üzere bir .NET uygulaması geliştiriliyorsa, düşük bellek ortamında çalışması için uygun jitter kullanılır. Diğer yandan geliştirilen assembly bir yedek sunucuda dağıtılacaksa (bellek çoğu zaman sorun olmayacaktır) jitter yüksek bellekli ortamda işlevini yerine getirmesi için optimize

edilecektir. Bu yolla yazılım geliştiriciler, etkin bir şekilde "tam-zamanında-derlenecek" ve farklı mimariler kullanılarak çalıştırılacak kod blokları yazabilirler.

Bütün bunlara ek olarak kullanılan jitter ile CIL komutları uygun makine kodlarına derlendiğinde, sonuçlar uygun bir yolla hedef işletim sisteminin belleğinde saklanacaktır. Buna göre *DosyayaYaz()* isimli metoda bir çağrı yapılırsa, ilk çağrıda CIL komutları platforma özel kodlara derlenecek ve sonraki kullanımlar için hafızada tutulacaktır. Dolayısıyla DosyayaYaz() metodunun ikinci çağrılışında CIL'i yeniden derlemeye gerek kalmayacaktır.

.NET Framework Kurulumu

.NET platformu üzerinde uygulama geliştirebilmek için bilgisayarımızda bazı yazılımların yüklü olması gerekmektedir. Bunlar;

.NET platformunda uygulam geliştirebilmek için gerekli yazılımları **http://msdn2.microsoft.com/en-us/netframework/aa731542.aspx** adresinden indirebilirsiniz.

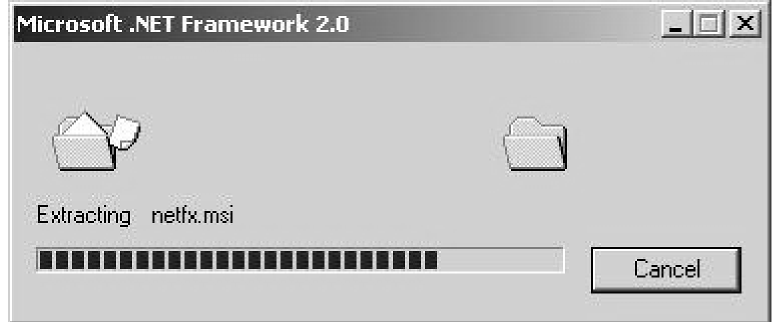
- **.NET Framework 2.0 Redistributable Paketi**: 22.4 Mb. büyüklüğündeki bu paket .NET 2.0 uygulamalarının bilgisayarınızda çalışması için gerekli olan herşeyi kurar. Amacınız sadece .NET uygulamalarını çalıştırmak ise .NET Framework 2.0 Redistributable paketini bilgisayarınıza kurmanız yeterlidir.
- **.NET Framework 2.0 SDK(Software Developement Kit)**: 354 Mb. büyüklüğündeki bu paket .NET platformu üzerinde uygulama geliştirmek için gerekli olan araçları, dökümanları ve örnekleri içermektedir. Uygulama geliştirebilmek için, bu yazılımların da bilgisayarınızda kurulu olması gerekmektedir.

.NET Framework 2.0 Redistributable paketi **Windows XP Service Pack-2** ve **Windows Vista** işletim sistemleri ile birlikte otomatik olarak sisteme yüklenmektedir. Bu işletim sistemlerinden birinde çalışıyorsanız, bu yazılımı kurmanıza gerek yoktur.

.NET Framework 2.0 Redistributable Paketinin Kurulumu

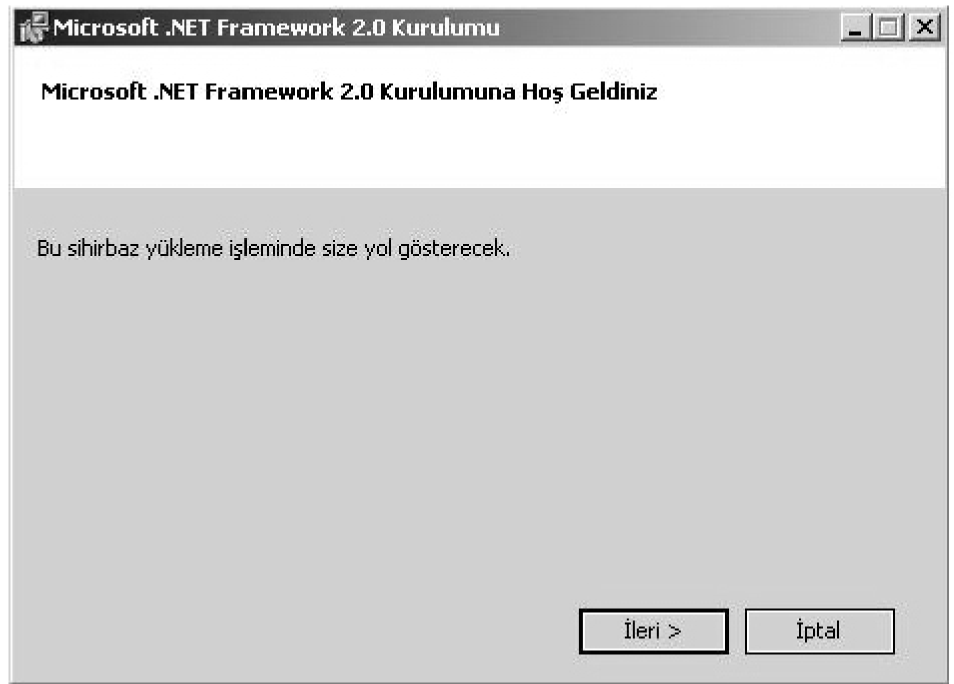
Yukarıda belirtilen adresten indirdiğiniz .NET Framework 2.0 Redistributable paketinin kurulumunu şu şekilde yapılmaktadır:

Dosyaya çift tıklayarak kuruluma başlayınız. Öncelikle paketteki dosyalar açılacaktır.



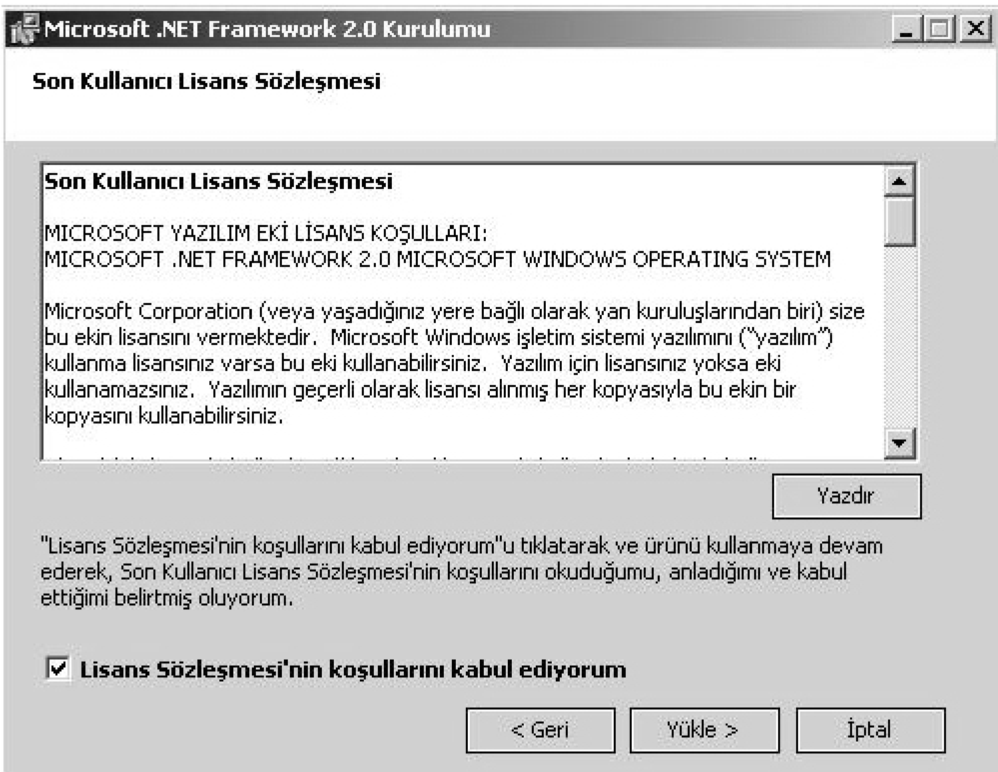
Şekil 15: Paketteki dosyaların açılması

Dosyalar açıldıktan sonra **Şekil 15**'deki gibi bir kurulum sihirbazı ile karşılaşacaksınız. Buradan **İleri** seçeneğine tıklayarak kuruluma devam ediniz.



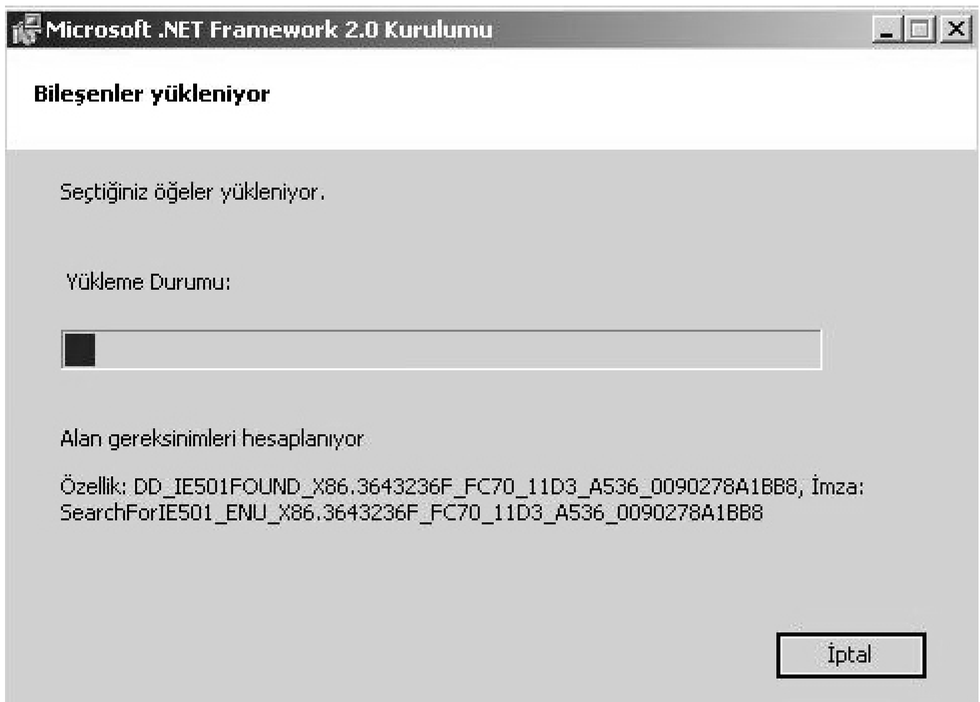
Şekil 16 – Kurulum sihirbazı

Son Kullanıcı Lisans Sözleşmesini okuyup, kabul ediyorsak **"Lisans Sözleşmesi'nin koşullarını kabul ediyorum"** seçeneğini işaretleyin. Bunu seçenek seçilmezse kurulum devam etmeyecektir. **Yükle** seçeneğine tıklayarak bir sonraki ekrana geçiniz.



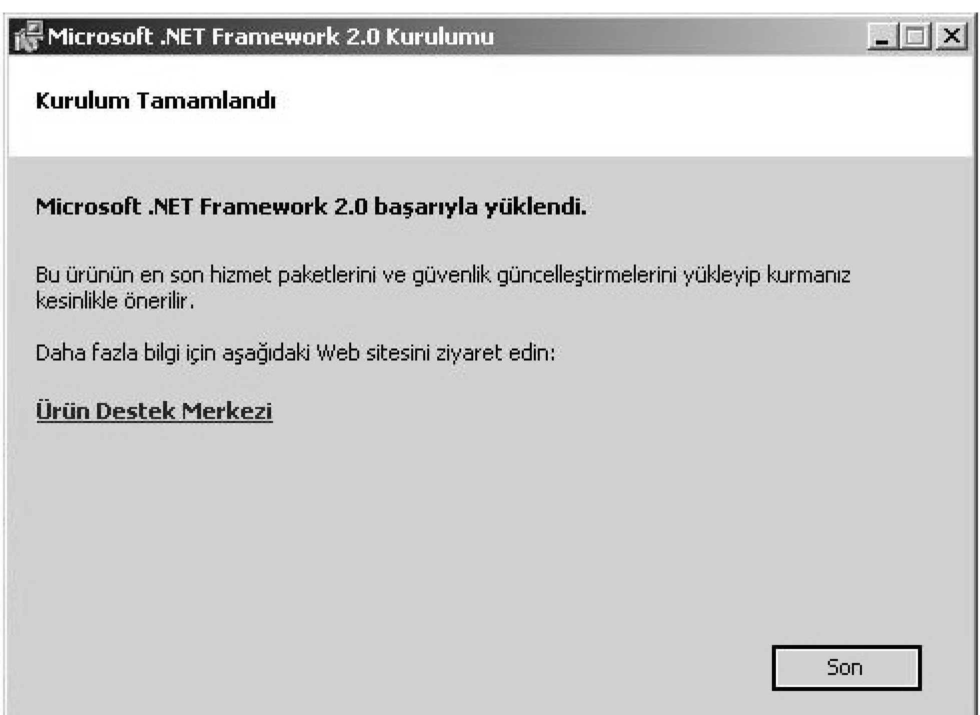
Şekil 17: Son Kullanıcı Lisans Sözleşmesi

Böylece kurulum başlayacaktır. Kurulum sistemin durumuna ve hızına göre 5-10 dakika arasında sürebilir.



Şekil 18: Bileşenler kuruluyor

Kurulum tamamlandığında karşınıza **Şekil 19**'taki gibi bir ekran gelecektir.



Şekil 19: Kurulum tamamlandı

Böylece bilgisayarınıza .NET Framework 2.0 Redistributable paketi kurulmuş olacak ve bundan sonra .NET uygulamaları çalıştırılabilecektir.

BÖLÜM 1: MERHABA DÜNYA

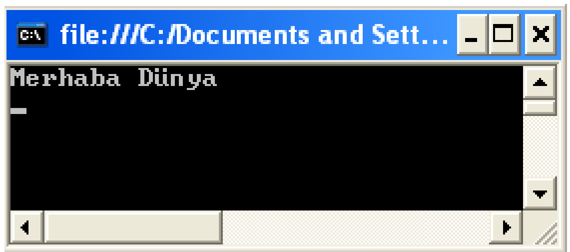
Bu bölüm çalışan basit bir örneği analiz ederek bir C# programının temel yapısını öğretmeyi hedeflemektedir. Bu bölümün içeriğinde ayrıca bazı temel giriş/çıkış operasyonlarını gerçekleştirmek için **Console** sınıfının nasıl kullanılacağı da yer almaktadır.

Bir C# Programının Yapısı

Yeni bir programlama dili öğrenirken bir çok insanın yazdığı ilk program kaçınılmaz "Merhaba Dünya" dır. Aşağıdaki örnek kod bir C# programının gerekli bütün elemanlarını içerir:

```
//C# dosyaları, *.cs dosya uzantısı ile sona ererler.
using System;

class HelloClass
{
    public static void Main(string[] args)
    {
        Console.WriteLine("Merhaba Dünya");
        Console.ReadLine();
    }
}
```

Bu kod bloğu csc.exe ile komut satırından (csc.exe'nin kullanımı ilereleyen bölümlerde anlatılmaktadır) ya da Visual Studio 2005 ile derlendiğinde çıktı aşağıdaki gibi olur:



Şekil 20: İlk uygulama

Sınıf (Class)

C# bütün programlama mantığının bir tip tanımlaması içerisinde olacağını söyler. Geliştirilen bir uygulama bir ya da daha fazla sınıf(class), yapı(struct) ve diğer tiplerden oluşan bir koleksiyon olur. (.NET dünyasında tip, şu kümenin üyelerinden birini anmak için kullanılan bir terimdir {Sınıf (class), yapı (structure), numaralandırıcı (enumaration), arayüz (interface), temsilci (delegate)}). Bu tiplerin arasında en temel olanı sınıftır. Sonraki bölümlerde ayrıntılı olarak görülecek olan sınıf; veri, bu verilerle ilişkide bulunabilen metotlar ve diğer üyeleri ile belli bir programlama görevini yerine getirmekle sorumludur.

"Merhaba Dünya" uygulamasındaki kodlara bakılacak olursa HelloClass adında tek bir sınıf olduğu görülür. Bu sınıf programa *class* anahtar kelimesi ile tanıtılırken sınıf adının ardından süslü parantez - { - gelir. Süslü parantez arasındaki her şey ilgili sınıfın parçasıdır.

Her bir C# sınıfı birer .cs dosyasına yazılabildiği gibi, bir .cs dosyasında birden fazla sınıf tanımlaması yapılabilir. Ayrıca C# 2.0 öncesinde bir sınıfı birden fazla fiziki dosyaya

parçalayacak şekilde tasarlamak mümkün değilken, C# 2.0 ile birlikte class anahtar kelimesinin önüne **partial** anahtar kelimesini koyarak bu mümkün olmaktadır. Aynı zamanda tüm sınıf tanımlamaları için .cs uzantılı dosya adı ile sınıf adı aynı olmak zorunda değildir; ancak genellikle değiştirilmez.

Main Metodu

Her uygulamanın bir başlangıç noktası olmalıdır. Bir C# uygulaması çalıştırıldığı zaman çalışmaya **Main** adındaki metottan başlar, kontrol bu metodun sonuna geldiğinde ise uygulama sonlanır. (Ya da varsa metot içerisinde **return** ifadesinin görüldüğü yerde uygulama sonlanır)

	C#, büyük-küçük harf duyarlı (case-sensitive) bir dil olduğu için "Main" ile "main", (Benzer bir örnek vermek gerekirse "ReadLine" ile "Readline") aynı değildir. Dolayısıyla başlangıç metodu her zaman ilk harfi büyük, geri kalan harfleri küçük "**Main**" olacak şekilde kodlanmalıdır.

Bir C# uygulamasında birden fazla sınıf olabilir, ancak sadece bir tane giriş noktası olmalıdır. Aynı uygulamada her birinde Main olan birden çok sınıf da yer alabilir; ancak sadece bir tane Main çalıştırılacaktır. O yüzden uygulama derlendiğinde hangisinin kullanılacağı belirtilmelidir.

Main'in niteleyicileri de önemlidir. Koda bakılacak olursa, Main metodunun tanımlamasındaki niteleyiciler "public", "static" ve "void" olarak belirlenmiştir. Bu anahtar kelimeler sonraki modüllerde daha ayrıntılı olarak incelenecek; o zamana kadar "public" üyelerin diğer tipler ve üyeler tarafından erişilebilir, static üyelerin sınıf düzeyinde üyeler olduğunu ve "sınıfAdi.uyeAdi" şeklinde çağrılarak kullanılabileceğini, "void" metotların ise çalışması sona erdiğinde ortama bir değer döndürmediğini bilmek yeterlidir.

Main() metodunun erişim belirleyicisi "public" yerine "private" da olabilir; bu şekilde diğer assembly'lerin, yazdığımız uygulamanın giriş noktasını çağıramaması sağlanmış olur.(Visual Studio 2005, bir programın Main() metodunu otomatik olarak "private" tanımlar) Ayrıca dönüş tipi de "void" yerine "int" olabilir. Bu, uygulama geliştiriciye Main() metodunun başarıyla sonlanıp sonlanmadığını öğrenmek için geriye sayısal değer döndürme fırsatı sağlar. Ancak Main() metodu her zaman "static" olmak zorundadır; yoksa derleyici tarafından uygulama için uygun bir giriş noktası bulunamaz.

	Hatta isim alanları kendi içerisinde başka isim alanları da içerebilir. Örneğin System.Runtime.Serialization.Formatters.Binary gibi…

HelloWorld_Uygulamasi.HelloClass | Main(string[] args)

```
using System;

namespace HelloWorld_Uygulamasi
{
    class HelloClass
    {
        public void Main(string[] args)
        {
            Console.WriteLine("Merhaba Dünya");
            Console.ReadLine();
        }
    }
```

Error List

1 Error | 0 Warnings | 0 Messages

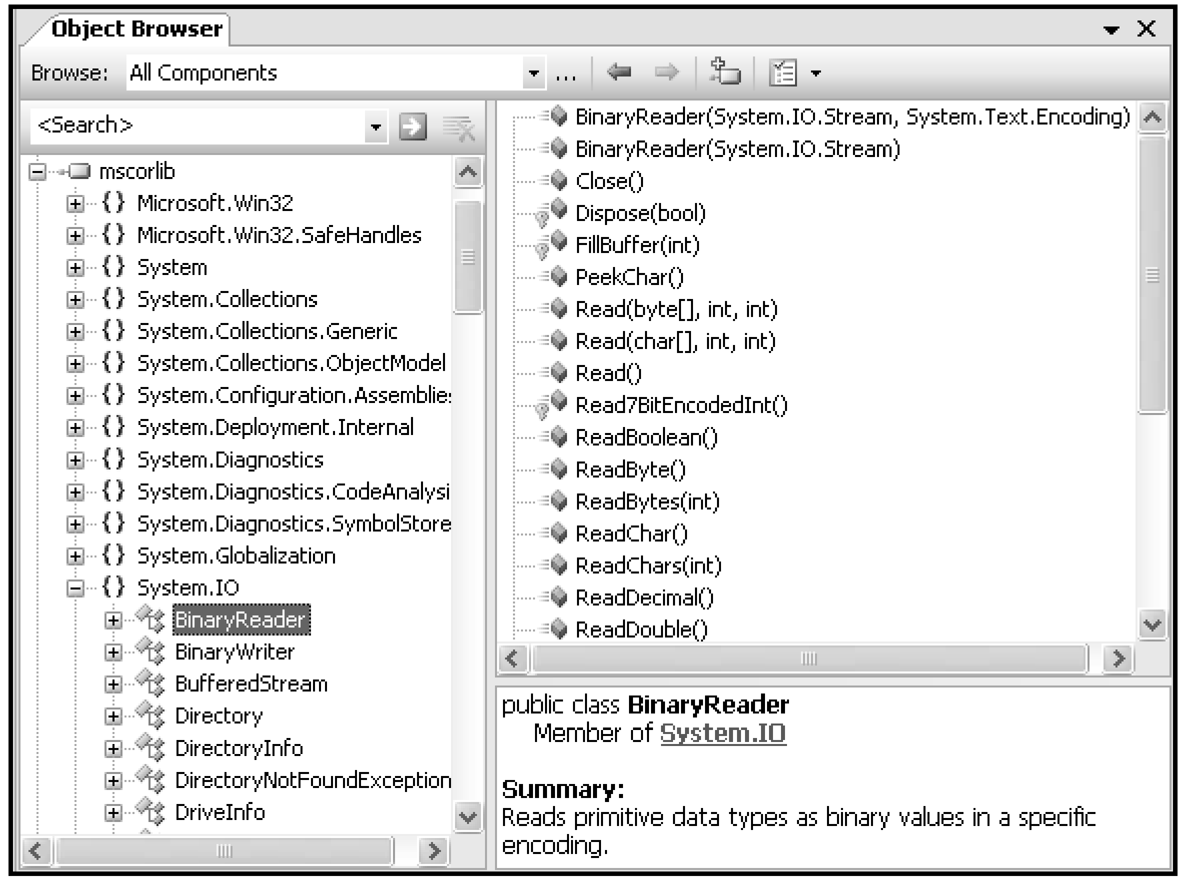
Description

Program 'C:\Documents and Settings\Administrator\My Documents\Visual Studio 2008\Projects\HelloWorld_Uygulamasi\obj\Debug\HelloWorld_Uygulamasi.exe' does not contain a static 'Main' method suitable for an entry point

Şekil 21: Main() metodu static olmalıdır.

Using Direktifi ve System İsim Alanı (Namespace)

C#, Microsoft.NET Framework'ün bir parçası olarak temel ve faydalı programlama işlerini gerçekleştirmek için yazılım geliştiricinin hizmetine birçok kullanışlı sınıf sunmuştur. Bu sınıflar isim alanları (namespaces) altında organize edilmişlerdir. Bir namespace, benzer amaca hizmet eden sınıflardan oluşan bir kümedir. Aynı zamanda bir isim alanı, başka isim alanlarını da içerebilir. Mesela System.IO isim alanı dosya giriş/çıkış ile ilgili tipleri, System.Data isim alanı temel veritabanı işlemleri ile ilgili tipleri içerir. Ayrıca vurgulamak da fayda var ki; tek bir assembly (mscorlib.dll gibi) istenilen sayıda isim alanı içerebilir, her bir isim alanında da istenilen sayıda tip yer alabilir. Bu, Visual Studio 2005 içerisindeki View menüsünden erişilebilecek olan Object Browser yardımıyla da görülebilir:



Şekil 22: Tek bir assembly, birden fazla isim alanına sahip olabilir.

Bütün bu isim alanları arasında en temel olanı "System" dir. Bütün isim alanları "System" altında toplanmıştır. Herhangi bir c# uygulaması geliştirirken mutlaka bu isim alanına atıfta bulunulur; çünkü kullanılan hemen hemen bütün tipler bu isim alanı altında ve bu isim alanının altındaki diğer isim alanlarının altında yer alır.

Yeniden belirtmek de fayda var ki isim alanları; ilişkili tipleri organize etmek ve uygulama geliştiricinin anlamasını, bulmasını kolaylaştırmak için uygun bir yoldan başka bir şey değildir, tek başlarına bir anlamları yoktur. Örneğin Console sınıfı, System isim alanı altındadır, bu sınıf WriteLine() gibi birçok üye sunar. WriteLine() metoduna erişmek için aşağıdaki kod yazılabilir:

```
System.Console.WriteLine("Merhaba Dünya");
```

Bu şekilde bir tipin ya da üyenin tam adını kullanmak çok kullanışlı değil; çünkü uygulamada satır sayısı arttıkça tekrar tekrar yazılan isim alanı sayısı artar. Bunun yerine uygulamanın en üst satırlarına ilk sınıf tanımlamasından önce **using** direktifi ile bir isim alanı belirtilebilir. **using** direktifi ile kod yazılırken hangi isim alanı altında olduğu belirtilmeyen tipler için kullanılacak isim alanı belirtilebilir. Ayrıca kaynak kod dosyasında birden fazla using direktifi yer alabilir.

```
using System;
. . .
Console.WriteLine("Merhaba Dünya");
```

Artık System isim alanı altındaki herhangi bir tipi ve üyelerini ya da diğer isim alanlarını kullanırken başlarına "System" yazmaya gerek yoktur.

Basit Giriş/Çıkış (Input/Output) İşlemleri

Console Sınıfı

Kitabın şu ana kadarki örneklerinde **System.Console** sınıfı sık sık kullanıldı, kullanılmaya da devam edilecek. Konsol kullanıcı arayüzü, Windows kullanıcı arayüzü kadar çekici değilken örnekleri konsol arayüzü ile kısıtlamak bize grafik arayüzünün kompleksliği ile uğraşmak yerine öğrenmeye çalıştığımız C# temellerine odaklanma fırsatı sunuyor.

Adından anlaşılabileceği gibi bu sınıf konsol uygulamalarında kullanıcıdan veri alma, kullanıcıya veri gösterme, oluşan hata durumlarında gerekli manipülasyonları gerçekleştirmeyi üstlenir. .NET 2.0 ile birlikte bu sınıfa katılan yeni üyelerle fonksiyonelliği arttırılmıştır. Aşağıdaki üyeler örnek olarak verilebilir…

Üye	Tanımı
BackgroundColor **ForegroundColor**	Ekran çıktısının arka plan ve yazı rengini değiştirmeye yarar.
WindowHeight **WindowWidth** **WindowTop** **WindowLeft**	Konsol ekran boyutunu değiştirmeye yarar.
BufferHeight **BufferWidth**	Konsol ekranında yazı yazılabilecek alanı belirler.
Clear()	Konsoldaki yazı alanını temizleyen metot.
Title	Çalışan konsol penceresinin başlığını belirler.

Tablo 2: Console sınıfının bazı üyeleri

Tablodakilere ek olarak **Console** sınıfı, ekrandan yapılan giriş ve çıkışları yakalayan üyeler tanımlar. Bu üyelere sınıf adı üzerinden erişiriz ve daha sonra görüleceği üzere bu üyelere static üyeler denir. **WriteLine()**, ekrana metin olarak yazı yazılmasını sağlar ve imleci bir alt satırda beklemeye zorlar, böylece arkasından yazılacak metin bir alt satırdan başlar. **Write()** metodunun tek farkı ise imleci ekrana yazılan yazının hemen sonunda bekletmesidir, böylece arkasından yazılacak metin aynı satırdan devam edecektir. **ReadLine()**, ekrana yazılan yazıyı metin olarak satır sonuna kadar okumayı sağlar, **Read()** ise ekrana yazılan metnin sadece ilk karakterini okur ve bunu karakterin ASCII karakter setindeki sayısal karşılığı olarak döndürür (Daha sonra görülecek dönüştürme yöntemleri ile bunun karakter karşılığı elde edilebilir), bir döngü içerisinde okunduğunda ve okuyacak karakter kalmadığında ise -1 döndürür. ReadLine() ve Read() metodu, ekrandaki yazıları kullanıcı klavyesinden ENTER tuşuna basılınca okur; yani uygulama, kaynak kodunda bu iki metoda rastlarsa ENTER tuşlanana kadar bekleyecektir.

Console sınıfı ile basit I/O işlemlerini örneklemek için kullanıcıdan veri alıp bu verileri yeniden ekrana yazan aşağıdaki gibi bir Main() metodu yazılabilir:

```
...

    public static void Main(string[] args)
    {
        //Ekrana yazı yazıyoruz...
        Console.Write("İsminizi giriniz : ");
        //Kullanıcı ENTER'a basınca o ana kadar yazılan yazı okunur ve bir
```

```
değişkene atılır.
        string giris = Console.ReadLine();
        //Değişken üzerindeki bilgiyi ekrana yazıyoruz...
        Console.WriteLine("Merhaba {0}",giris);

        Console.Write("Yaşınız kaç? : ");
        giris = Console.ReadLine();
        Console.WriteLine("Demek {0} yaşındasınız...",giris);
    }
...
```

```
Hello World
İsminizi giriniz : Emrah
Merhaba Emrah
Yaşınız kaç? : 25
Demek 25 yaşındasınız...
```

Şekil 23: Basit I/O işlemleri

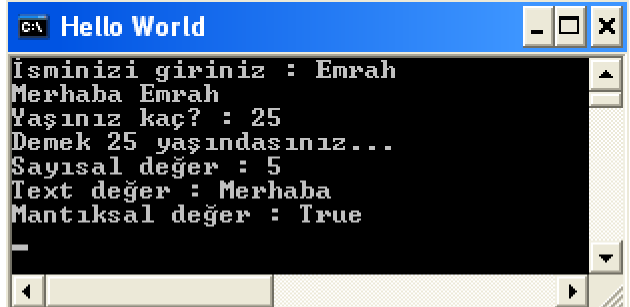
Konsol Çıktısını Formatlama

Console sınıfının WriteLine() ve ReadLine() metotlarından genel kullanım amaçlarının yanı sıra şu iki genel amaç için de faydalanılır : *Console.WriteLine()* metodu parametre almadan ekranda bir satır atlamak için kullanılırken; Console.ReadLine(), konsol penceresinin ekranda hemen görünüp kaybolmasına çözüm olarak ENTER tuşlanana kadar beklemesini sağlar.

	Uygulama Visual Studio 2005'de F5 ile ya da Debug → Start Debugging ile başlatıldığında konsol penceresi ekranda durmaz. Ctrl + F5 ile ya da Debug → Start Without Debugging ile başlatıldığında ise ekranda sabit durur. Buradaki temel fark uygulamanın debug modda başlatılıp başlatılmaması tercihidir.

Şu ana kadarki örneklerde WriteLine() içerisine yazılan birçok {0}, {1} kullanımlarını, text alanın ardından da değerlerinin verildiği görüldü. .NET, string formatlamanın yeni bir stilini geliştirdi, buna "yer tutucu" (PlaceHolder) denir.

```
    public static void Main(string[] args)
    {
      ...

        int sayisal = 5;
        string text = "Merhaba";
        bool mantiksal = true;

        // Bir textin içerisinde yazılan '\n'yeni bir satır ekler.
        Console.WriteLine("Sayısal değer : {0}\nText değer : {1}\n
        Mantıksal değer : {2}", sayisal, text, mantiksal);

        Console.ReadLine();
    }
```



Şekil 24: String ifade içerisinde birden çok "yer tutucu" kullanımı

WriteLine() içerisine çift tırnak ("...") arasında yazılan her şey normalde ekranda görünür. Bunun istisnai durumlarından biri kullanılan süslü parantezlerdir. Bu süslü parantezler ekran çıktısında görünmez; çünkü onlar başka değerler için yer tutarlar. Yer tutmak istediğimiz her değer için bir süslü parantez içerisine '0'dan başlamak kaydıyla indeks numaraları verilir. Bu indekslerin yerine ekranda görünecek değerler ise çift tırnakların arkasından verilir `("...",degerler)`. Eğer birden fazla yer tutucu kullanılmışsa değerler, virgülle ayrılarak içerdeki indeks sırasına göre yazılır.

Aynı zamanda tek bir yer tutucunun, tek bir string veri ile birden fazla yerde kullanılması da mümkündür. Örneğin "asla, asla vazgeçemem senden asla" şarkı sözlerini koda dökmek istersek;

```
//Tarkan söylüyor...
Console.WriteLine("{0}, {0} vazgeçemem senden çok {0}","asla");
```

Yer tutucu indeksine eklenecek bir parametre ile alan genişliği belirlenebilir ve ilgili değerin bu alanda sağa ya da sola dayalı olarak yazdırılması sağlanabilir.

```
Console.WriteLine("\"10 birimlik alanda sola dayalı :{0,-10}\"",23);
Console.WriteLine("\"10 birimlik alanda sağa dayalı :{0,10}\"",23);
```

Bir string verinin içerisinde kullanılan '\'i şareti, kendisinden sonra gelen karakterin özel anlamını iptal edip metin olarak ekrana yazdırır. Mesela '\{', ekrana '{' yazılmasını, '\\' ekrana '\' yazılmasını sağlar. Buna alternatif olarak string verinin arasına yazıldığı çift tırnağın önüne '@' işareti koyularak verinin kelimesi kelimesine ekrana yazılması sağlanabilir. Örneğin @"c:\\Documents and Settings\Administrator" olarak kodlamak, ekrandaki çıktının "c:\\Documents and Settings\Administrator" olmasını sağlar.

Nümerik Formatlama

Nümerik verilerin nasıl formatlanıp görüntüleneceğini belirtmek için "format stringi" kullanılır. Tam söz dizimi (syntax) **{N,M,:FormatString}** iken burada N yer tutucunun indeks numaralandırıcısını, M alan genişliği ve önüne koyulan '+' – '-' işaretiyle yazının hangi tarafa dayalı yazılacağını, FormatString ise nümerik verinin nasıl gösterilmesi gerektiğini belirtir. Bütün format seçenekleri aşağıdadır:

Format Karakteri	Açıklaması
C ya da c	**Nümerik** veriyi lokal para sembolünü kullanarak para olarak gösterir. Format string harfini "Currency" kelimesinin baş harfinden alır.
D ya da d	**Bu** format sadece tam sayılar için desteklenir. Hemen arkasına aldığı sayı ile (d5 gibi) ekranda gösterilmesi gereken minimum basamak sayısı belirtilebilir ve eğer sayının basamak sayısı, belirtilen minimum basamak sayısından fazlaysa sayının başına

	‘0’ eklenerek stringe çevrilir. Format string harfini “Decimal” kelimesinin baş harfinden alır.
E ya da e	**Sayıyı** üstel notasyon kullanarak göstermek için kullanılır. Sayıyı “d.dddd…E+ddd” formatında stringe çevirir. (d’ ler birer rakamı temsil ediyor) Burada ilk basamak her zaman tam sayı olur ve format stringinin hemen arkasına belirtilebilecek bir sayı ile ondalık kısmında kaç basamak bulunacağı belirtilebilir. Belirtilmezse varsayılan olarak bu değer 6’dır. Üstel ifade ise her zaman minimum 3 basamaktan oluşur; dolayısıyla eğer ihtiyaç duyulursa e ya da E (string format’ına yazılacak harfin büyük mü küçük mü olduğuna bağlı olarak değişiyor)’den sonraki rakam “0” larla 3 basamağa tamamlanır… Format string harfini “Exponential” kelimesinin baş harfinden alır.
F ya da f	**Sayı**, ondalık olsun ya da olmasın varsayılan olarak 2 basamak ondalıkla stringe çevrilir. Eğer tam sayı ise sonuna ondalık olarak sıfırlar eklenir. Eğer ondalık sayı ise ve ondalık kısmı ikiden fazla ise varsayılan değer iki olduğu için virgülden sonraki değerlerinin sadece ilk ikisi alınarak formatlama yapılır… Format string’inin hemen arkasından verilecek bir değer ile sayının ondalık kısmında bulunması istenilen basamak sayısı belirlenebilir (f4 gbi). Format string harfini “Fixed” kelimesinin baş harfinden alır.
G ya da g	**Sayıyı** hemen arkasından verilebilen değere ve basamak sayısına bağlı olarak ya aynen yazılır ya da ‘e’ string formatıyla stringe çevrilir. Eğer format string ile birlikte bir değer verilmezse sayı olduğu gibi alınır. Eğer verilirse; değer, stringe çevrilecek sayının istenilen tam ve ondalık basamak sayıları toplamını işaret eder. Değer, sayının tam kısmındaki basamak sayısından az ise formatlama üstel olarak yapılır, diğer durumlarda sayı olduğu gibi çevrilir. Format string harfini “General” kelimesinin baş harfinden alır.
N ya da n	**Bu** format stringi ile soldan başlayarak her 3 basamakta bir araya bin ayıracı koyulması sağlanır. Hemen arkasından verilebilecek değer ile istenilen ondalık basamak sayısı verilebilir. Bu değer varsayılan olarak 2’dir. Format string harfini “Number” kelimesinin baş harfinden alır.
P ya da p	**Sayı** 100 ile çarpılarak yüzde işareti ile sunulur. Hemen arkasından verilebilecek değer, yüzde değerinden sonra gelmesi istenilen ondalık basamak sayısıdır. Bu değer varsayılan olarak 2’dir. Format string harfini “Percent” kelimesinin baş harfinden alır.
X ya da x	**Sayı** hexadecimal (onaltılı sayı sistemi) formatta stringe çevrilir. Sadece tam sayılar için desteklenir. Format string harfini “Hexadecimal” kelimesindeki x harfinden alır.

Tablo 3: .NET formatlama stringleri ve anlamları

Aşağıda nümerik formatlama ile ilgili bazı örneklere yer verilmiştir:

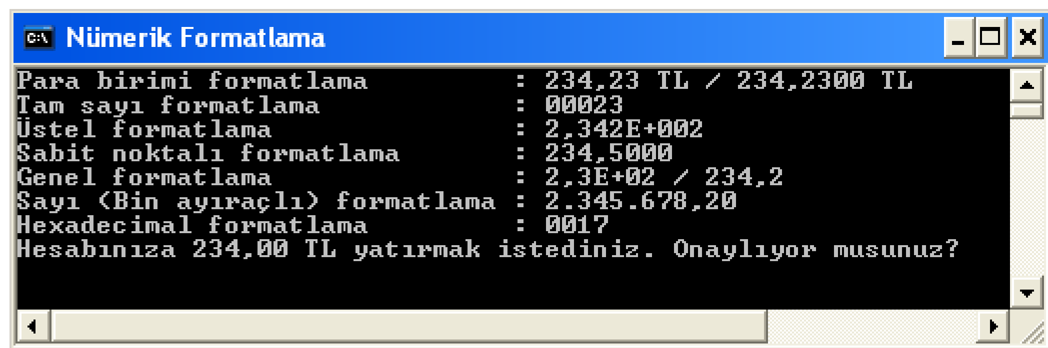
```
public static void Main(string[] args)
   {
        Console.Title = "Nümerik Formatlama";
        Console.WriteLine("Para birimi formatlama: {0:C} / {0:C4}",
234.23);
        Console.WriteLine("Tam sayı formatlama : {0:D5}",23);
        Console.WriteLine("Üstel formatlama : {0:E3}",234.2);
        Console.WriteLine("Sabit noktalı formatlama : {0:f4}",234.5);
        Console.WriteLine("Genel formatlama : {0:G2} {0:G4}",234.23);
        Console.WriteLine("Sayı(Bin ayraçı) formatlama :
{0:N}",2345678.2);
        Console.WriteLine("Hexadecimal formatlama : {0:X4}",23);
        Console.ReadLine();
```

```
    }
```

.NET formatlama karakterlerinin kullanımı sadece konsol uygulamaları ile sınırlı değil. **String.Format()** metodu ile bütün format stringleri herhangi bir uygulama tipinde (Masaüstü uygulamaları, ASP.NET, XML Web servisleri…) kullanılabilir.

```
public static void Main(string[] args)
    {
    ...
        //Yeni bir string veri oluşturmak için string.Format() metodu 
kullanılıyor.
        string formatStr;
        formatStr = string.Format("Hesabınıza {0:C} yatırmak istediniz. 
Onaylıyor musunuz?",234);
        Console.WriteLine(formatStr);

        Console.ReadLine();

    }
```

Bu kodların çalıştırılmasıyla elde edilen ekran çıktısı ise aşağıdadır:



Şekil 25: Nümerik formatlama örnekleri

Uygulamalarda Yorum Satırı

Uygulamalar için dokümantasyon sağlamak önemlidir. Yazılan kod ile ilgili yeterli bilgi sağlamak, geliştirilme sürecinde hiç yer almamış veya sonradan katılmış bir programcının uygulamayı anlayıp takip edebilir seviyeye gelmesi sürecini fark edilir düzeyde etkiler. İyi yorumlar, sadece kodlara bakılarak anlaşılması kolay olmayacak bilgiyi sağlarlar ve orada verilmeye çalışılan cevap "NE" değil "NEDEN" sorusu olmalıdır.

C#, uygulama kodlarına yorum eklemek için şu yolları sunar: Tek yorum satırı, çoklu yorum satırları ve XML dokümantasyonu. Tek satırlık yorumlar çift bölü işareti (//) ile eklenebilir. Uygulama çalıştırıldığı zaman bu karakterlerin arkasından gelen yazılar satır sonuna kadar ihmal edilir.

```
//Uyelik için kullanıcıdan ismi alınır
Console.Write("Adınızı giriniz : ");
string ad = Console.ReadLine();
```

Eğer aynı zamanda birden fazla satır yorumlanmak istenirse ya her satıra çift bölü işareti ('//') koyulur ya da blok yorumlama işaretinden faydalanılır. Blok yorum, '/*' işareti ile başlayıp '*/' işaretini görene kadar devam eder.

```
/*Aşağıdaki işlemin sonucu
  x değişkenine atanıp kullanılır*/
  int x = (…);
```

C# Komut Satırı Derleyicisi ile Çalışmak

C# uygulaması çalıştırılmadan önce **derlenmelidir.** Bu iş için Visual Studio 2005 gibi uygulama geliştirme ortamı kullanılabileceği gibi C# uygulamaları için **csc.exe** ile de .NET assembly'leri oluşturabilir (csc, C-Sharp Compiler'ı simgeliyor). Bu derleme aracı .NET Framework 2.0 SDK'nın (Software Development Kit –Yazılım Geliştirme Kiti-) içerisinde yer almaktadır. Büyük ölçekli bir uygulama hiçbir zaman komut satırından konsol penceresi kullanarak derlenmez; ancak *.cs uzantılı dosyanın nasıl derleneceğinin temellerini bilmek bir uygulama geliştirici açısından önemlidir. Ayrıca böyle bir sürece ihtiyaç duyulması için birkaç sebep göstermek gerekirse;

- Uygulamanın bulunduğu ortamda Visual Studio 2005 bulunmayabilr.
- C# bilgimizi biraz daha derinleştirmek isteyebiliriz. Böylece otomatik derleme aracı yazma hakkında temel bilgimiz artmış olur.
- Bir uygulamayı derlemek için grafik bir arabirim kullanılsa da eninde sonunda csc.exe'ye başvurulacaktır. Dolayısıyla burada arka planda neler olup bittiği ile ilgili biraz daha ayrıntılı bilgi elde etmek istenebilir. (Örneğin kendi Visual Studio IDE'mizi yazmak istediğimizde…)

C# komut-satırı derleyicisi kullanılmaya başlanmadan önce çalışılan makinenin csc.exe'yi tanıdığından emin olunması gerekiyor. Eğer makine doğru bir şekilde konfigüre edilmediyse C# dosyalarını derlemek için her seferinde csc.exe'nin bulunduğu dizini belirtmek gerekecek. Makinemize *.cs uzantılı dosylarımızı her hangi bir dizinden derleme yeteneği kazandırmak için (Windows XP işletim sistemi üzerinde gerçekleştirilen) aşağıdaki adımları takip etmek gerekir:

- Bilgisayarıma (MyComputer) sağ tıklanır ve Seçenekler (Properties) penceresi açılır.
- Gelişmiş (Advenced) tabı açılır ve oradan Ortam değişkenleri (Environment Variable) butonu tıklanır.
- "Sistem Değişkenleri" bölümünden "Path" değişkenine çift tıklanır ve listenin en sonuna csc.exe'nin içerisinde bulunduğu klasörün fiziki yolu yazılır. ("Path" değişkenindeki herbir değer noktalı virgül ile ayrılmıştır)

```
C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727
```

Tabiki sizlerin adresi kullandığınız Framework'ün versiyonu ve .NET Framework 2.0 SDK'sının dizininizdeki yerine bağlı olarak değişebilir. "Path" değişkeni güncellendikten sonra test etmek amacıyla açık olan bütün konsol pencereleri kapatılıp yeni bir tane açılır ve komut penceresindeyken

```
csc ?/  ya da  csc -?
```

yazılır. Eğer işler yolunda gitmişse C# derleyicisinin sunduğu seçenekler listesinin elde edilmiş olması gerekmektedir.

csc.exe'yi incelemeye başlamadan önce aşağıdaki "Path" değişkenini de sistem değişkenleri listesine ekleyelim:

```
C:\Program Files\Microsoft Visual Studio 8\SDK\v2.0\Bin
```

Bu klasörde uygulama geliştirme sürecinde kullanılabilecek birçok komut satırı derleyicisi yer alır. Bu iki değişkenin eklenmesiyle artık herhangi bir .NET aracını herhangi bir komut penceresinden çalıştırmak mümkündür.

Uygulama geliştirme bilgisayarı csc.exe'yi tanıdığına göre C# komut satırı derleyicisi ve notepad kullanarak DemoUyg.exe adında basit bir tek dosyalı assembly geliştirilebilir. Bir notepad açarak aşağıdakiler yazılır:

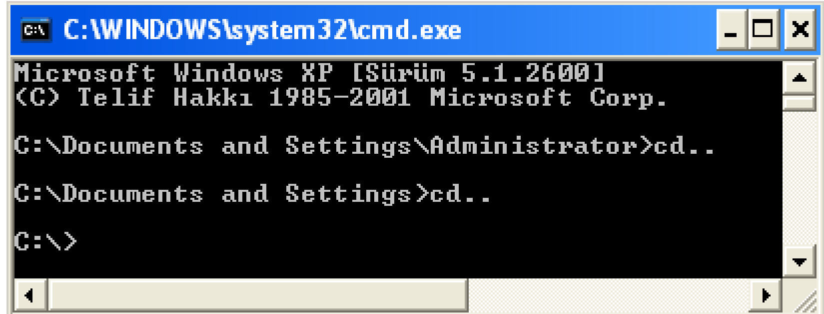
```
//Basit bir C# uygulaması
using System;

class DemoUyg
{
    public static void Main(string[] args)
    {
        Console.WriteLine("Deneme 1,2");
        Console.ReadLine();
    }
}
```

Dosya uygun bir dizine (Mesela c:\Ornekler klasörüne) DemoUyg.cs olarak kaydededilir. C# derleyicisinin seçenekleri incelenecek olursa dikkat edilecek ilk nokta; oluşturulacak assembly'nin adı ve tipinin nasıl belirtileceğidir (Örneğin Uygulamam.exe adında bir konsol uygulaması, Robotik.dll adında bir kod kütüphanesi, WinUygulamam.exe adında bir masaüstü uygulaması vb...) Her olasılık, csc.exe'ye komut satırı parametresi olarak geçirilen özel bir işaret ile temsil edilir.

Seçenek	Tanımı
/out	Oluşturulan assembly'ye isim vermek için kullanılır. Varsayılan olarak oluşan assembly'nin adı, başlangıç input'u olan *.cs dosya adıdır.
/target.exe	Çalıştırılabilir konsol uygulaması geliştirmek için bu seçenek kullanılabilir. Varsayılan çıktı tipi budur.
/target : library	Bu seçenek tek dosyalı bir *.dll assembly'si inşa edilir.
/target : module	Bu seçenek bir **module** oluşturur. Her bir modül, çok dosyalı bir assembly'nin elemanıdır.
/target : winexe	Target:exe ile masaüstü uygulamaları geliştirilebilirken, bu seçenek arka planda konsol penceresinin kapatılması gerektiğini garanti eder.

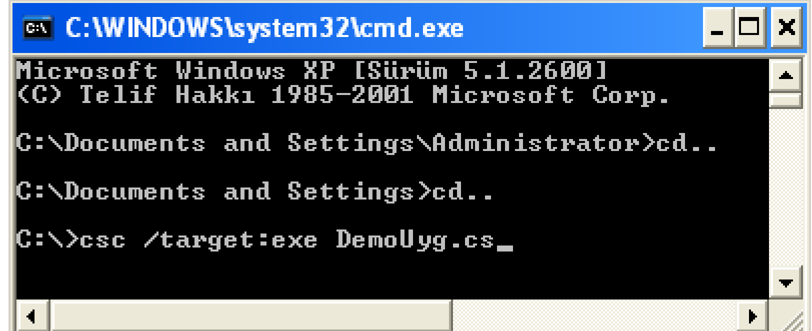
Tablo 4: C# derleyisinin çıktı tabanlı seçenekleri

DemoUyg.cs'i DemoUyg.exe ismindeki çalıştırılabilir bir uygulamaya derlemek için komut penceresinden *.cs dosyanın dizinine geçilir:



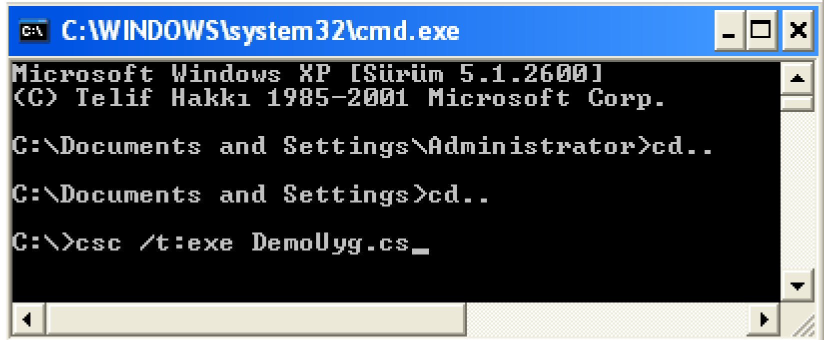
Şekil 26: Derlenecek dosyanın dizinine inilir.

ve aşağıdaki komut satırı girilip ENTER'a basılır:



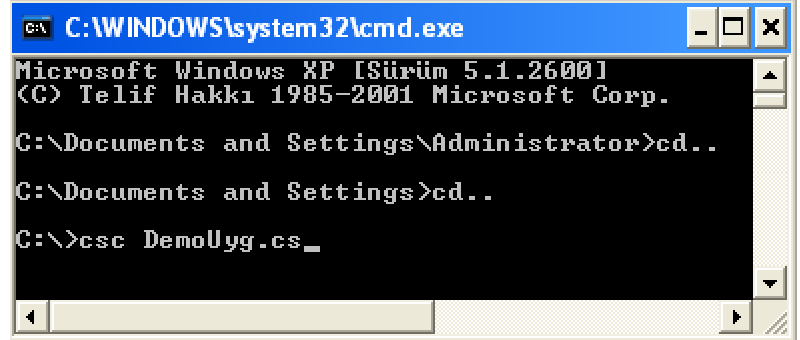
Şekil 27: DemoUyg.cs, aynı isimli çalıştırılabilir (.exe) dosyaya derleniyor.

Burada bilinçli olarak /out kullanılmadı. O yüzden programın giriş noktasını içeren (Main() metodu) DemoUyg sınıfından üretilen çalıştırılabilir dosya *.cs dosyasıyla aynı isimde (DemoUyg.exe olarak) ilgili dizinde oluşturulur. Bu arada birçok C# işareti, kısaltmaları tanır. Burada da /target yerine /t kullanılabilir:



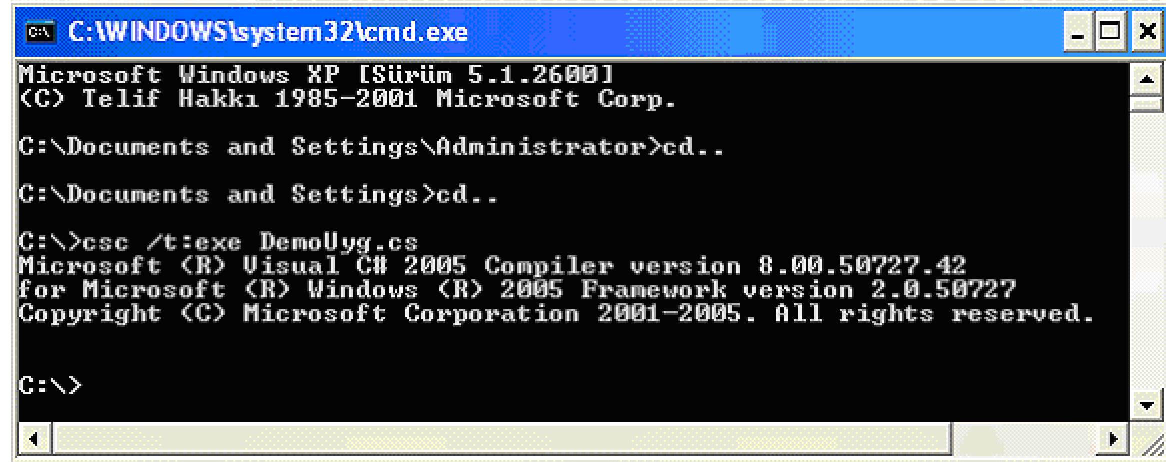
Şekil 28: Derleme /target:exe yerine kısaca /t:exe şeklinde de yapılabilir.

Aynı zamanda bu işaret (t:exe) C# derleyicisinin varsayılan dosya çıktı tipidir; dolayısıyla DemoUyg.cs şu şekilde de derlenebilir.



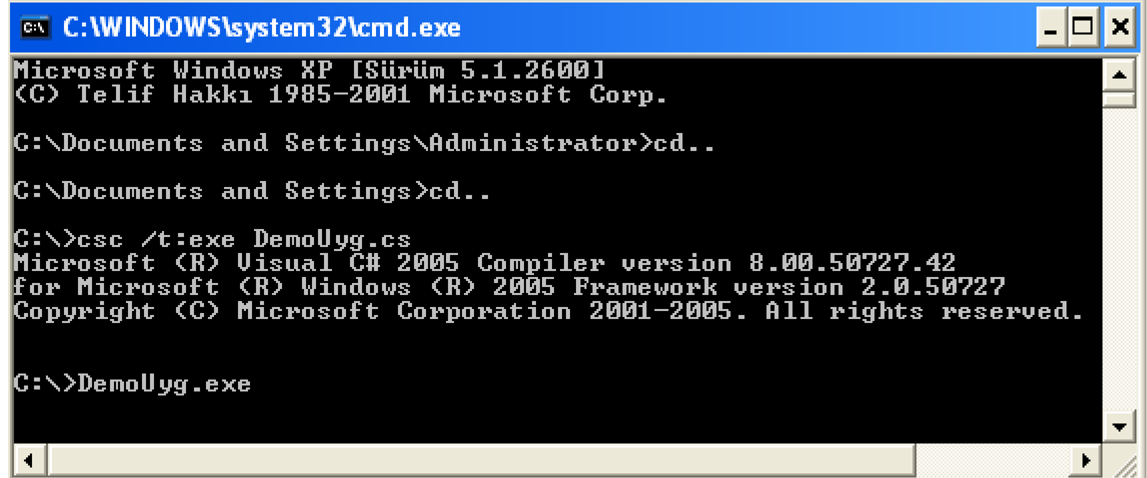
Şekil 29: Varsayılan /target:exe olduğu için hiçbir dosya çıktı tipi belirtilmediğinde target:exe olarak işlem yapılır.

Artık üretilen DemoUyg.exe komut satırından ya da üretildiği dizinden çalıştırılabilir. Ekran çıktıları aşağıdaki gibidir:



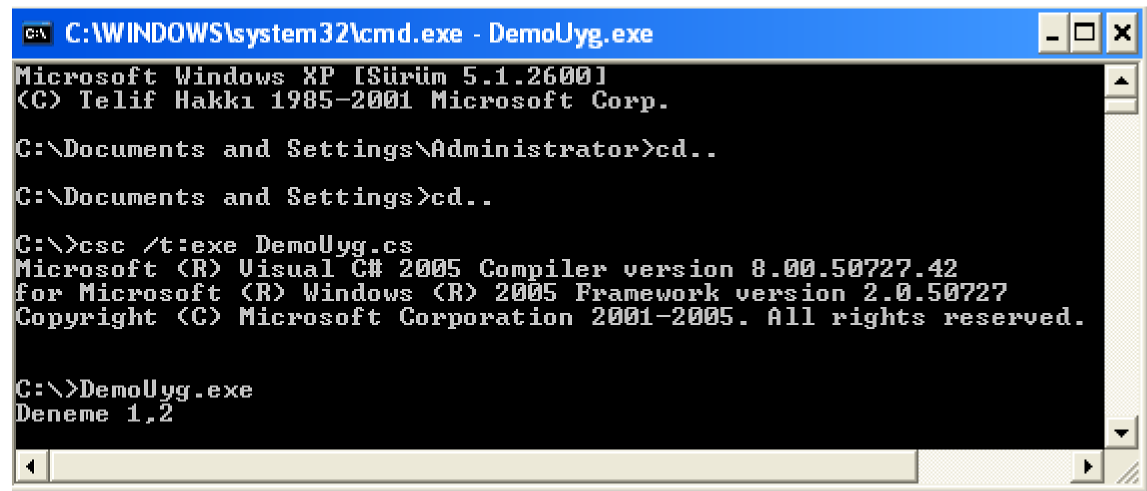
Şekil 30: Sayfa başarılı bir şekilde derlenip programın çalıştırılabilir assembly'si (DemoUyg.exe) oluşturulur...

İster 'c' dizini altındaki dosyaya gidip, ister komut-satırından .exe dosyasının adı yazılıp ENTER'a basılarak çalıştırılabilir.



Şekil 31: Derleme sonucunda oluşan .exe dosyasının çalıştırılması

Ve son çıktı aşağıdaki gibi olur:



Şekil 32: Çalışan programın ekran çıktısı

*.cs dosyasında bir değişiklik yapılmadığı sürece uygulama, üretilen bu *.exe dosyasından çalıştırılabilir. *.cs dosyasında bir değişiklik yapıldığında ise uygulama, güncel hali ile çalışmaya devam etsin diye yeniden derlenmelidir.

Visual Studio 2005 ile Çalışmak

Visual Studio 2005 Kullanarak Uygulamaların Derlenmesi

Visual Studio 2005 kullanarak uygulamaların derlenmesi için C# uygulamasını içeren proje açılarak **Build** menüsünden **Build Solution** tıklanır.

Visual Studio 2005 Kullanarak Uygulamaların Çalıştırılması

Visual Studio 2005 kullanarak uygulamaların derlenmesi için **Debug** menüsünden **Start Debugging** (ya da Ctrl F5) ya da **Start Without Debugging** (ya da F5) tıklanır. Konsol uygulaması geliştirilirken Ctrl + F5 ile çalıştırılması, uygulama sonlandığında kullanıcı herhangi bir tuşa basana kadar pencerenin açık kalmasını sağlar. Bu iki çalıştırma yolunun farkı; ilkinde uygulama debug (hata ayıklama) modunda çalıştırılırken ikincisinde debug mod kapalı olarak çalıştırılmasıdır.

Derleme Zamanı Hataları

Hem komut satırından hem de Visual Studio 2005 kullanarak uygulama derlendiğinde csc.exe, sözdizimi ya da semantik (anlam) bir hata yakalarsa bunu raporlar. Eğer komut satırından derlenmişse ekrana yazdırılan mesaj ile hatanın dosyadaki satır sayısı ve karakter pozisyonu bildirilir. Eğer derleme için Visual Studio 2005 kullanılmışsa Hata Listesi (ErrorList) penceresi bütün hatalı satırları işaret edip hata mesajını bildirir; hataya çift tıklanmasıyla da uygulama ortamı bizi ilgili satıra götürecektir. Uygulama derlendiğinde artık hata alınmıyorsa derleme zamanı hatalarından arınmışız demektir; uygulamanın .exe'si çalıştırılabilirdir.

	Uygulama derlenmeden çalıştırılırsa, otomatik olarak önce derlenir, ardından hata yoksa çalıştırılır.

Çalışma Zamanı Hataları

Uygulama çalıştırılabilir dosyasının (*.exe) sorunsuz olarak üretilmesinin ardından çalıştırılır ve eğer çalışma sırasında bir hata oluşursa buna **çalışma zamanı hatası (exception)** denir. Uygulama geliştirici olarak çalışma zamanında oluşacak hatalara karşı kod yazmamışsak bütün programlama ortamları için uygulama ekranında hatayla ilgili bilgilerin yer aldığı bir ekranla karşılaşılır. Uygulamadan faydalanan kullanıcının böyle bir ekranla karşılaşması istenmeyeceği için (en azından anlamlı bir hata mesajı ile karşılaşmasını isteyeceğimiz) bu hataları ele alan kodların yazılmış olması gerekir. (Bu konu "Çalışma zamanı hatalarını ele alma" başlığı altında ileriki konularda detaylı olarak incelenecektir)

Visual Studio 2005 Hata Ayıklayıcısı Yardımıyla Uygulamanın İzlenmesi

Uygulamaya kesme noktaları (break point) koymak ve kullanılan değişkenlerin çalışma zamanı değerlerini incelemek,takip etmek için Visual Studio 2005 Hata Ayıklayıcısı'ndan (Visual Studio 2005 Debugger) faydalanılabilir.

Eğer bir uygulama satır satır ilerlenerek çalıştırılmak istenirse **Debug** menüsünden **StepOver** kliklenerek çalıştırılır ve yine aynı şekilde uygulama sonlanana kadar

ilerlenebilir; satırlar arası ilerlerken uygulamanın nerelere dallanıp çalıştığı takip edilebilir ve kullanılan değişkenlerin değerleri incelenebilir.

Uygulamaya kesme noktaları koymak için kod yazdığımız C# dosyasında (*.cs) herhangi bir satıra sağ tıklayıp **BreakPoint** seçeneğinden **Insert BreakPoint** tıklanır. Aynı zamanda sol kenar boşluğuna tıklanarak da koyulabilir. Kaldırmak için yine aynı satır üzerinden **BreakPoint** seçeneğinden **Delete BreakPoint** tıklanır. Aynı zamanda sol kenar boşluğunda çıkmış olan işarete tekrar tıklanarak da kaldırılabilir. Uygulama debug modda çalıştırıldığı zaman (F5 ile) herhangi bir satır ya da satırlara koyulan kesme noktasına kadar çalışıp o satırda bekleyecektir. Kullanılan değişkenlerin o anki değerlerinin incelenme ihtiyacı duyulduğunda bu faydalı olacaktır. Daha sonra uygulama ister adım adım (F10 ya da F11 ile) ister normal şekilde (F5) çalıştırılmaya devam edilir.

	F10 (Step Over) ile adım adım hata ayıklama yapılırken eğer kullanılan bir metot vb… varsa onların içerisine girilmezken F11 (Step Into) kullanıldığında her birinin içerisine girilir.

BÖLÜM 2: DEĞİŞKEN KAVRAMI

Değişkenler

Değişken Nedir? Neden İhtiyaç Duyarız?

Bütün uygulamalar veriyi hemen hemen aynı yollarla işlerler. Genel olarak bir programcının ilk öğrenmesi gereken, uygulamalarda veriyi nerede saklayıp nasıl kullanacağıdır. Uygulama çalışırken geçici olarak veriyi saklama ihtiyacı duyarsak değişken üzerinde saklarız. Değişkeni kullanmadan önce tanımlamak gereklidir; değişken tanımlandığı zaman uygulama belleğinde bu değişkenin tutulması için bir yer ayrılır. Bu tanımlama sırasında daha sonra değişkene erişebilmek için bir ad verilir ve veri tipi belirlenir. Tanımlandıktan sonra artık değişkene taşıyacağı değer verilebilir.
Bir değişken tanımlanırken aşağıdaki söz dizimi kullanılır:

VeriTipi degiskenAdi;

Öyleyse burada tartışılması gereken sıradaki konu, **veri tipi** kavramının ne olduğu ve çeşitlerinin neler olduğudur.

Veri Tipi (Data Types)

Veri tipi, bir değişken için bellekte ayrılacak bölgenin hangi içerikteki ve formattaki verileri saklayacağını bildirmek için kullanılır. Örneğin bir kitabın sayfa sayısı, uygulamanın ilerleyen aşamalarında kullanmak amacıyla saklanmak istenirse tanımlanacak değişken için bellekte ayrılacak alana sadece sayısal verilerin girebileceğini belirtmek gerekir. Bu alanın sayısal veriler için ayrılmasına ek olarak, tam sayı mı ondalık sayı mı olacağı, sayısal aralıkları gibi kriterleri belirtilerek sınırları daha da kesinleştirilmelidir. Kitabın adı için ayrılacak bellek alanı metin tabanlı veriler için olacaktır. Bu yolla kitabın sayfa sayısı için bellekte açılan alana ondalık sayı ya da text tabanlı bir değer girilmesinin; kitabın adı için ise sayısal bir değer girilmesinin önüne geçilmiş olup değerin istenilen içerik ve formatta tutulacağı garanti edilmiş olur. İşte bunları garanti eden kavram **veri tipidir.**
Şimdi de değişkenler için belirlenebilecek veri tipi çeşitlerini inceleyelim: .NET dünyasında 5 ana tip olduğundan bahsedilmişti. Bunlar tekrar sayılacak olursa;

- Sınıf (class)
- Yapı (struct)
- Numaralandırıcı (enumaration)
- Arayüz (interface)
- Temsilci (delegate)

olduğu görülür. Bir değişken tanımlarken belirtilmesi gereken veri tipi, bu 5 ana tipten birinde oluşturulmuş olur. Yanlış anlaşılmasın; sayılan bu 5 ana tipte değişken oluşturulmaz; bu ana tiplerde oluşturulan veri tipleri kullanılır. Örnek vermek gerekirse C#'da tam sayı olarak veri saklamak istendiğinde veri tipi olarak, bir çok programlama dilinde var olan **int** kullanılabilir. **int,** .NET Framework'u tarafından uygulama geliştiricilerin temel programlama görevlerinde kullanmaları için önceden oluşturulan bir veri tipidir. Bellekte kaplayacağı alan bellidir: 4 byte (32 bit). Özelliği, tam sayı datalarını saklayabilmesi ve bunu yaklaşık olarak -2 milyar,+2 milyar aralığındaki sayılar için yapabilmesidir. Ayrıca **int**, bir yapıdır (struct). **int** tipinde oluşturulacak bir değişkenin alabileceği minimum, maksimum değerleri ve daha sonra görülecek tip dönüşümlerde kullanılan üyeleri **int**'in bir **struct** olmasının getirileridir. Önceden tanımlı veri tiplerine bir

başka örnek olarak metin tabanlı verilerin saklanabileceği "**string**" verilebilir. İstenilen uzunlukta karakterlerden oluşan bu veri tipinin taşıyabileceği veri uzunluğu sınırsızdır. (Ya da uygulama belleğiyle sınırlıdır) Bütün dillerin alfabelerindeki karakterleri, rakamları, matematiksel işlemlerde kullanılan simgeleri metin olarak karakter dizisi şeklinde veri saklayabilen bu tipin bellekte ne kadar yer kaplayacağı int veri tipi gibi önceden belli değildir; çalışma zamanında belli olur. **String** ile ilgili verilebilecek son bilgi de bir **sınıf (class)** olduğudur. Yine birçok programlama dilinde herhangi bir formda yer alan mantıksal veri tipi C#'da **bool** olarak yer almaktadır. Taşıyabileceği değerler iki tanedir: "True" ve "False". Ondalıklı sayıları saklamak için **double** veri tipi, tek bir karakter tutmak için **char** veri tipi vardır. Bir de bütün tiplerin babası sayılabilecek **object** tipi vardır ki taşıyabileceği değerlerde sınır yoktur; bütün tiplerin taşıyabileceği değerleri üzerinde tutabilir. Örneğin **object** tipinden oluşturulmuş bir değişken 5 (sayısal tip), "bilgisayar" (string tip), 's' (char tipi), ahmet isminde Personel tipinde kullanıcı tanımlı vb. değişken değerleri taşıyabilir. **Object** tipini daha çok duyacağınıza emin olabilirsiniz.

Değişken Tanımlama

Değişken tanımlamak için takip edilmesi gereken sözdiziminin;

veriTipi degiskenAdi;

olduğunu görmüştük. Bunu C# veri tipleriyle örneklemek gerekirse;

```
int sayac;
```

ya da

```
string adres;
```

Değişken tanımlamaları ile ilgili örnekleri test ederken Main() metodu içerisinde yazıldığı varsayılır.

Değişken tanımlaması bir satırda tek bir değişken için yapılabilirken, aynı tip için olmak kaydıyla birden fazla değişken için de yapılabilir.

```
double basariOrani, katilimYuzdesi;
```

C#'da bir değişken tanımlandığında saklayacağı veri için veri tipinin boyutu kadar bellekte yer ayrılır (bazı tiplerin bellekte ne kadar yer kaplayacağı önceden belliyken –int, bool gibi- bazı tiplerin ki çalışma zamanında belli olur –string gibi-) ve o bellek alanına daha sonra erişim için bir isim verilir (değişken adı); ancak bu bellek alanının içi henüz boştur. Bu yüzden;

```
int sayac;
Console.WriteLine("Sayacın şu anki değeri : {0}",sayac);
```

şeklinde yazılacak bir kod derlenmeyecektir; çünkü henüz o değişken için başlangıç değeri verilmemiştir.

Değişkenlere Başlangıç Değeri Verme

Bir değişkene değer vermek için atama operatörü kullanılır. Atama operatörü C#'da '=' dir. Öyleyse yeni tanımlanmış bir değişkene başlangıç değeri aşağıdaki gibi verilebilir:

```
 int sayac;
sayac = 1;
```

Değişkenin başlangıç değeri aynı zamanda değişken tanımlandığı anda da verilebilir:

```
int sayac = 1;
string isim = "Keskin C";
```

Eğer aynı satırda birden fazla değişken tanımlanıp başlangıç değerleri de verilmek istenirse aşağıdaki yol takip edilmelidir:

```
double basariOrani = 88.8, katilimYuzdesi = 52.4;
```

Önceden Tanımlı Veri Tipleri

Şimdi de .NET Framework'ün uygulama geliştiricilere sunduğu önceden tanımlı veri tiplerinin tam listesini ve bazı özelliklerini inceleyelim:

C# Kısaltma	System Tipi	Aralığı-Kapsamı	Kullanımı ve Bellek Alanı
sbyte	System.SByte	- 128' den 127	İşaretli 8-bit tam sayı
byte	System.Byte	0'dan 255	İşaretsiz 8-bit tam sayı
short	System.Int16	- 32.768'den 32.767	İşaretli 16-bit tam sayı
ushort	System.UInt16	0'dan 65535	İşaretsiz 16-bit tam sayı
int	System.Int32	-2.147.483.648'den 2.148.483.647	İşaretli 32-bit tam sayı
uint	System.UInt32	0'dan 4.294.967.295	İşaretsiz 32-bit tam sayı
long	System.Int64	–9,223,372,036,854,775,808'den - 9,223,372,036,854,775,807	İşaretli 64-bit tam sayı
ulong	System.UInt64	0'dan 18.446.744.073.709.551.615	İşaretsiz 64-bit tam sayı
float	System.Single	3.4. x $10e^{-38}$ 'den 3.4 x $10e^{38}$	32-bit ondalıklı sayı
double	System.Double	5 x $10e^{-308}$ 'den 1.7x $10e^{308}$	64-bit ondalıklı sayı
decimal	System.Decimal	$10e^{-28}$ 'den 7.9 x $10e^{28}$	128-bit işaretli ondalıklı sayı
bool	System.Bool	true ya da false	Doğru ya da yanlışlığı temsil eder
char	System.Char	U0000'den Uffff	16-bit unicode tek karakter
string	System.String	Uygulama belleği ile sınırlı	Unicode karakter kümesi temsil eder
object	System.Object	Herhangi bir tip tutulabilir.	.NET'deki bütün tiplerin temel sınıfı.

Tablo 5: Önceden tanımlı veri tipleri

Bu veri tipleri ile ilgili aşağıdaki bilgiler önemlidir:

- C#, negatif değer almayan veri tiplerini destekler. Bu veri tipleri her ne kadar ortak dil spesifikasyonları dışında olsa da kullanışlı oldukları kesindir. Örnek vermek gerekirse kullanıcı yaşları bir değişken içerisinde tutulmak istenildiğinde

bu değişkenin muhtemel değerleri arasında negatif sayıların olmayacağı, işaretsiz tiplerden en uygun olanıyla sağlanabilir.

Ortak dil spesifikasyonları (CLS –Common Language Spesification-), .NET tabanlı dillerin aralarında anlaşabilmeleri için desteklemeleri gereken minimum ortak tip listesi (CTS –Common Type System-) kuralları olarak özetlenebilir. Yukarıdaki bilgiden çıkarılacak sonuç şu olmalıdır : C#'da geliştirilmiş bir kod kütüphanesi, (.NET'in sunduğu özgürlük çerçevesinde) başka bir programlama dili ile geliştirilen bir uygulamada kullanılmak istendiğinde C#'da olup diğer dilde olmayan özellikler sorun teşkil eder, bunun en güzel örneği de işaretsiz tipler olarak verilebilir. İşte diller arası etkileşimde bu sorunun dert açmaması için ortak dil spesifikasyonları listesi mevcuttur ve bu liste bütün dillerin desteklediği özelliklerin bir listesidir. Dolayısıyla başka diller ile geliştirilen uygulamalarda kullanmak için sınıf kütüphaneleri geliştirirken ortak dil spesifikasyonları çerçevesinde bunu gerçekleştirmek gerekmektedir.

- Veri tipleri listesindeki onbeş tipin sekizi **tamsayı** (sbyte, short, int, long, byte, ushort, uint, ulong), üçü **ondalık** sayı (float, double, decimal), ikisi karakter tabanlı (char,string),biri **mantıksal** (bool), biri de **bütün** seçenekleri kapsayan (object) verileri tutabilecek tiplerdir.
- Ondalık tipler arasında **double**, varsayılandır. Yani bir değişkene ondalık değere sahip veri atanırsa eşitliğin sağ tarafındaki verinin tipi her zaman double olarak algılanır. Bu yüzden float ve decimal veri tiplerindeki değişkenlerin değerlerinde eğer ondalık kısım varsa verinin sonuna bitişik olarak veri tipini belirten bir son-ek koyulmalıdır. Bu sayede değerin double değil diğer ondalık veri saklayabilen tiplerden biri olduğu anlaşılabilmektedir. Bu son ekler float için 'f' ya da 'F' iken; decimal için 'm' ya da 'M' dir.

```
double ondalikDouble = 23.56;       //Double için son-eke gerek yok.

decimal tamSayiDecimal = 34;        //Ondalık kısmı olmayan decimal ve
                                      float için de sorun yok.

float ondalikFloat = 23.45F;        //Ondalık kısmı olan float ve decimal
                                    //için sonekler koyulmalı; yoksa derleme
decimal ondalikDecimal = 67.89M;   //zamanı hatası alınır.
```

- C#, değişkenlerin uygun veri tipleri ile birlikte saklanması sayesinde **tip güvenli bir dildir**. Aynı bellek bölgesi için yanlışlıkla farklı tiplerde veriler tutulmasına müsaade edilmez.
- **Object** tipinde oluşturulan bir değişkene istenilen tipte veri atanabilir. Bu sınıf, .NET FrameWork içerisinde tanımlanmış bütün tiplerin ve kendi tanımlayacağımız tiplerin temelidir.
- Dikkat edilirse tabloda tipler için iki tane kolon var: Biri "**System tipleri**", diğeri de **"C# kısaltmaları"** bir başka deyişle "C# takma adları (alias)". Yazılan kodlar csc.exe derleyicisi ile ortak bir ara dile (CIL -Common Intermediate Language-) çevrilir ve ortak çalışma zamanının (CLR –Common Language RunTime-) ele aldığı kod CIL kodu olur; bu noktadan sonra C# kodunun bir işlevi kalmaz. Bu senaryo diğer bütün diller için de tabi ki geçerlidir. Öyleyse herhangi bir dilin kaynak kodu kendi derleyicisi ile CIL'e derlenir ve o noktada dile özel hiçbir bilgi kalmaz. **Süreç, ortak çalışma zamanı (CLR) kontrolünde ortak tip sistemi (CTS) kuralları çerçevesinde ortak ara dilde (CIL) yürür**. Yani belli bir noktadan sonra her şey standarttır. (Bu yüzden .NET'de performansı diller belirlemez.) İşte kendi oyun alanımızda tipler için int, bool, float gibi anahtar kelimeler kullanırken, kontrol CLR'a geçince hepsinin adı diğer bütün diller düzeyinde ortak olacaktır. Örneğin C#'ta tamsayı değişkenler "int" anahtar sözcüğü ile tanımlanabilirken, VB.NET "Integer" kelimesini kullanır. Her iki dilin derleyicileri kaynak kodlarını derleyip CIL'e çevrilince tipler System.Int32 olur. Bu standartları belirleyip takip eden

mekanizmanın adı ortak tip sistemidir (CTS –Common Type System-). Demek ki değişkenlerin tipleri işaretlenirken ortak tip sisteminin desteklediği adlarıyla işaretlenir, değişkenin bellekteki değeri güncelleneceği zaman tip güvenlik kontrolü System isim alanı altındaki orijinal adlarıyla yapılır. Bunun sağlamasını görmenin en basit yollarından biri Visual Studio 2005 kullanarak uygulama geliştirirken imleci tanımlanacak bir değişkenin üzerine getirip bilgiyi okumak olacaktır.

Start Page | Program.cs*

DemoUyg

```
//Basit bir C# uygulaması
using System;

class DemoUyg
{
    public static void Main(string[] args)
    {
        float oran1 = 12.9f;
        struct System.Single
        Represents a single-precision floating point number.
    }
}
```

Şekil 33: Bir değişken CLR tarafından System isim alanındaki eşdeğer tipi ile de kullanılabilir.

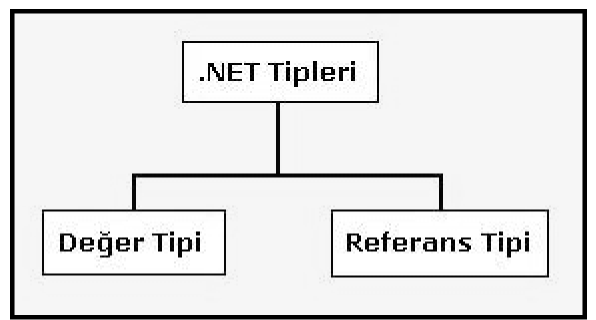
Şunu da belirtmekde fayda var ki; bir değişken tanımlamasında veri tipi olarak C# takma adı float ya da ortak tip sistemindeki karşılığı System.Single belirtilebilir.

```
float oran = 12.9f;
ya da
Single oran2 = 12.9f; //(Uygulamanın en üstünde "using System;"
                        olduğu varsayılmıştır)
```

- Listedeki tiplerden string ve object **sınıf**, geri kalan 13 tip **struct**'tır.
- .NET Framework'ünde Tablo 5'dekilere ek olarak birçok yapı, sınıf vb. tipler bulunmaktadır. Bu listedekiler hem temel programlama görevlerini gerçekleştirmek için en sık başvurulan tipler, hem de C# anahtar kelimesi olan tiplerdir. Örneğin System.DateTime struct'ı, tarih bilgilerini tutmak için sık sık kullanılan bir tip olmasına rağmen C# kısaltması yoktur.
- Beş ana tipte oluşturulan veri tiplerine ek olarak uygulama geliştirici ihtiyaç duyduğu yerde kendi tiplerini de yazabilir; ki zaten bir .NET uygulaması, daha önce yazılmış olan tipler ve onları kullanarak programcının yazdığı tiplerin etkileşmeleri ile oluşan bütündür. İlerleyen konularda özel numaralandırıcı (enumaration), yapı (struct) ve sınıfların (class) nasıl yazılacağı incelenecekken; temsilci (delegate) ve arayüz (interface) geliştirmek ise bu kitabın kapsamı dışında kalmaktadır.

Ortak Tip Sistemi Türleri: Değer Tipi – Referans Tipi

.NET platformunda çalışma zamanının tipleri tanımlarken, kullanırken ve yönetirken takip ettiği kuralları tanımlayan model olan "Ortak Tip Sistemi", tipleri iki grupta ele alır. Bu gruplar, ilgili tiplerde oluşturulacak değişkenlerin bellekte tutulacak yerleri, orada tutulma süreleri, atamalarda birbirlerine karşı davranışları bakımından farklılık gösterir.



Şekil 34: CTS, tipleri temel olarak iki grupta inceler.

Değer Tipleri (Value Types)

Değer tipleri, şu ana kadar adlarını duyduğunuz .NET'deki 5 ana tipten **struct** olarak oluşturulmuş önceden tanımlı tipler (tam sayı tipleri, ondalık sayı tipleri,karakter, mantıksal ve tarih veri tipi), temel sınıf kütüphanesinin içerisinde yer alan birçok **enum** ve bizim kendi oluşturacağımız struct ve enum tiplerini içerir. Değer tipli bir değişkenin en temel özelliği, **veriyi kendi üzerinde taşımasıdır**.

Referans Tipleri (Reference Types)

Referans tipleri, .NET'deki 5 ana tipten **sınıf(class)** olarak oluşturulmuş önceden tanımlı tipler, (string,object) temel sınıf kütüphanesinde yer alan sayısız sınıf, **temsilci (delegate)** ve arayüzler (**interface)** ile kendi oluşturacağımız sınıf, temsilci ve arayüz tiplerini içerir. Referans tipli bir değişkenin en temel özelliği, sakladığı veriyi doğrudan kendi üzerinde tutmamasıdır; bunun yerine veri belleğin başka bir bölgesinde dururken değişken, verinin bellek üzerindeki adresini tutar. Yani referans tipli değişken, **sadece bir adrestir**; belleğin farklı bölgesindeki verisi için bir **referanstır**.

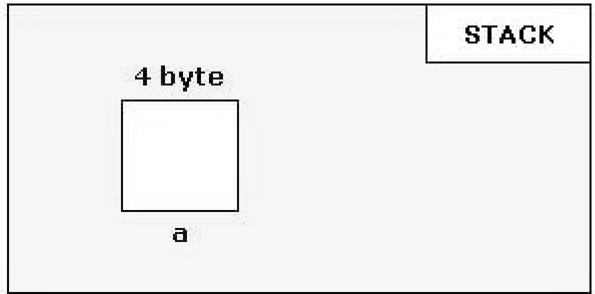
Değer Tiplerini Anlamak

int, değer tiplidir; int tipinde oluşturulmuş değişken ise değer tipli bir değişkendir. (int ve değer tipli diğer önceden tanımlı tipler, struct ana tipinde oluşturulmuş tiplerdir) String ve Object hariç Tablo 5'teki bütün tipler değer tiplidir. Ayrıca kendi struct ve enum'larımızı oluşturduğumuzda, onlar da değer tipli olur.

Değer tipli bir değişken tanımlandığında belleğin **stack** (Uygulamanın çalıştığı makinenin geçici belleğinde –Ram'inde- belli bir bölge) adı verilen bölgesinde ilgili tipte değişkeni tutmak için yeterli bellek alanı ayrılır; bu alan önceden tanımlı tüm değer tipleri için bellidir. Değişkene değer ataması yapılan bir ifade ile birlikte değer, ayrılan bellek alanına yerleştirilir. Aynı tipte ikinci bir değişken tanımlanıp, başlangıç değeri ilk değişkenin değerinden verilirse, ikinci değişken ilk değişken ile aynı veriyi tutacaktır. Her iki değişken atama sonucu aynı veriyi tutsa da bu, ilgili değerin iki kopyası olduğu gerçeğini değiştirmez: Biri ilk değişkenin üzerinde, diğeri ikinci değişkenin üzerinde. Dolayısıyla bu değişkenlerden birinde yapılacak değişiklik diğerinin taşıdığı değeri etkilemeyecektir. Bu anlatılanlar kod ve grafik üzerinde aşağıdaki gibi incelenebilir:

<u>1.Adım:</u>

```
int a;
```

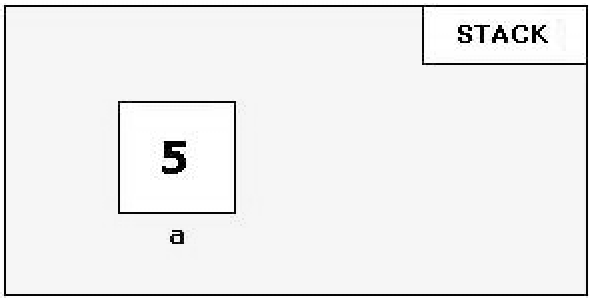


Şekil 35: Değer tipli a değişkeni tanımlanır.

- a adında int tipinde bir değişken tanımlanır. Bu satır ile birlikte belleğin stack bölgesinde 4 byte'lık yer ayrılır. Bu bellek bölgesine daha sonra erişim için değişkenin adı kullanılacaktır.

2.Adım:

```
a = 5;
```

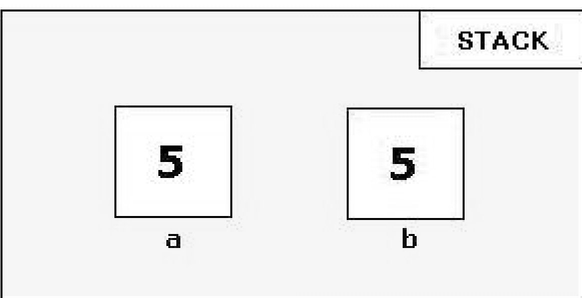


Şekil 36: a değişkene başlangıç değeri verilir.

- Bu satır ile değişkene başlangıç değer ataması yapılmış olur. '5' değeri, a değişkeni için ayrılan bellek alanına yerleştirilir.

3.Adım:

```
int b = a;
```



Şekil 37: Yeni tanımlanan b değişkeni, başlangıç değerini a değişkeninin değerinden alır.

- Burada b adında int tipinde ikinci bir değişken oluşturuluyor ve başlangıç değeri oluşturulduğu anda a değişkeninden veriliyor. Sırasıyla önce belleğin

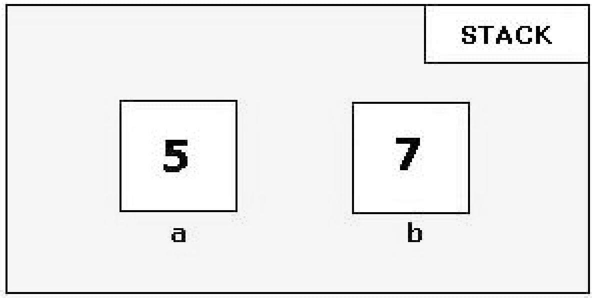
stack bölgesinde b değişkeni için 4 byte'lık yer ayrılır, bellek bölgesine değişkeni adı (b) verilir ve sonra ilk değeri a değişkeninin değerinden elde edilip 5 olarak yerleştirilir. Bu iki değişkenin değeri her ne kadar aynı (5) olsa da şu anda iki farklı kopyası bulunmaktadır. Biri a değişkeninin üzerinde, diğeri de b değişkeninin. İki değişkenin mevcut değerlerini ekrana yazdıralım :

```
Console.WriteLine("a : {0} / b : {1}",a,b);
```

Şekil 38: b değişkeni oluşturulduktan sonra değişkenlerin değerleri

4.Adım:

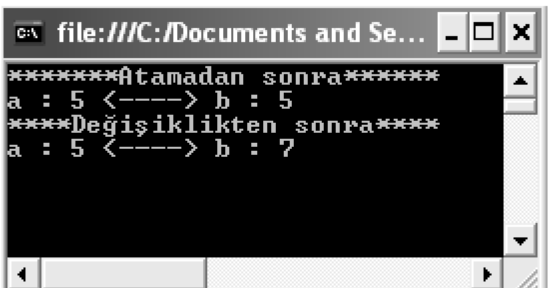
```
b = 7;
```



Şekil 39: b değişkenini değeri değiştirildikten sonra her ikisinin de durumları

- Bu değişkenlerden birinde yapılacak değişikliğin diğer değişkenin değerini etkilemediği şekilden ve ekran çıktısından anlaşılıyor:

```
Console.WriteLine("a : {0} <----> b : {1}", a, b);
```



Şekil 40: b değişkeninin değeri değiştirildikten sonra değişkenlerin değerleri

Referans Tiplerini Anlamak

String, değer tiplidir; string tipinde oluşturulmuş bir değişken ise referans tipli bir değişkendir. Tablo 5'deki önceden tanımlı tiplerden String ve Object birer sınıf oldukları için referans tiplidirler. Ayrıca kendi class, delegate ve interface'lerimizi yazdığımız zaman onlar da referans tipli olur.

Burada amaç sınıf kullanımını göstermek olmamakla birlikte referans tiplerin çalışma mekanizmasını açıklamak için basit bir sınıf yazacağız. **Sınıf (class)**, gerçek hayatta var olan varlıkları uygulama üzerinde modellemek için durum ve davranışlar tanımlayabilen, bunları bünyesinde barındırabildiği değişken verileri, metotlar ve diğer üyeleri ile gerçekleştiren bir tiptir.

Referans tipli bir değişken oluşturuldğunda bellekte iki ayrı bölgede yer işgal edilir. Önce referans için stack bölgesinde 4 byte'lık bir yer açılır; burada esas verinin bellekteki adresini tutacak olan referans değişkeni yer alacaktır. Referans tiplerin değer tiplerinden ayrılan önemli özelliklerinden biri de oluşturulurken "new" anahtar kelimesinin kullanılmasıdır. Yani referans tipli bir değişken oluşturulurken;

VeriTipi degiskenAdi = new VeriTipi();

söz dizimi kullanılır. En sondaki parantezler, sınıflara özgü bir özellik olup **Yapıcı Metot (Constructor)** olarak adlandırılıran üyeyi temsil eder. Sınıflar öğrenilirken ayrıntılarıyla incelenmesi gereken bir üyedir.

Referans oluşturulduğunda henüz hangi veriyi işaret edeceğini bilmez. **new** anahtar kelimesinin görülmesiyle birlikte gerçek veriyi içeren nesne de bellekteki yerini alır. Belleğin bu bölgesine **heap** adı verilir. Burada referans tipli değişkenlerin gerçek verisi yer almaktadır, değer tipli değişkenler asla burada kendilerine yer edinemezler. Değinilmesi gereken bir diğer bilgi de referans tipli değişken verilerinin bellekte ne kadar yer kaplayacakları önceden bilinemez; çalışma anında belli olur. Değer tipleri anlatılırken takip edilen grafikli örneğin bir benzerini burada da yer almaktadır:

1.Adım:

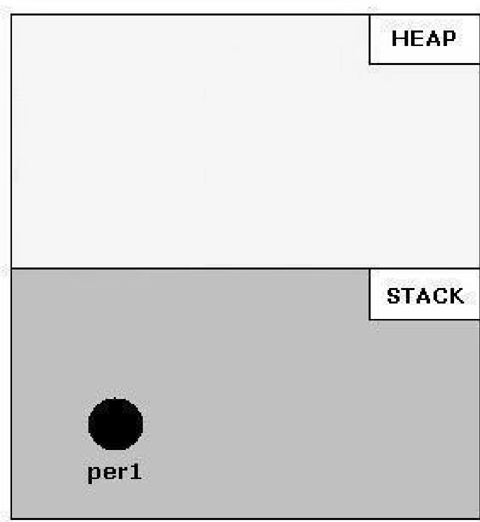
- Öncelikle referans tiplerini temsil edecek çok basit bir **sınıf** yazılır. Bu sınıf gerçek hayatta çalışan birini modelliyor olsun. Bu nedenle **Personel** isimli bir sınıf yazmak tercih edilebilir. Bu sınıf, maas adında bir tane üyeye sahip olsun. Personel sınıfını kullanan kişi ya da başka bir tipe, Personel sınıfının bütün üyelerini gösterip göstermemek seçeneği uygulama geliştiricinin elindedir; yani sınıf üyelerine erişim seviyeleri vardır. Burada o seviyelerden en yukarda olanı **public** erişim belirleyicisi kullanılmaktadır; böylece Personel sınıfını kullanan kimsenin maas üyesine erişebileceği garanti edilmiş olur.

```
using System;

namespace CTStipleri
{
    class Personel
    {
        public int maas;
    }


    class ReferansTipleri
    {
        public static void Main(string[] args)
        {

        }
    }
}
```

Şekil 41: İsim alanı (namespace) düzeyinde yazılan basit bir Personel sınıfı

2.Adım:

```
Personel per1;
```

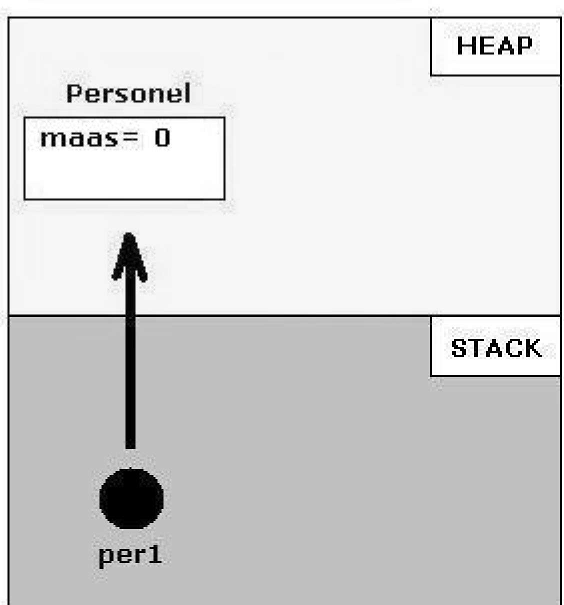


Şekil 42: Personel tipinden bir değişken oluşturulur.

- Main() içerisinde Personel sınıfından bir değişken oluşturulur. Bu ifade ile belleğin stack bölgesinde 4 byte yer kaplayan ve esas verilerin adresini tutacak olan referans oluşturulur. Per1 referansının henüz çalışma zamanında işaret edeceği herhangi bir nesne yok. Öyleyse devam edelim.

3.Adım:

```
per1 = new Personel();
```



Şekil 43: "new" anahtar kelimesi ile birlikte, verileri içeren nesne bir tane üyesiyle belleğin heap bölgesine yerleşir.

- Referans tiplerinin "new" anahtar kelimesi ile oluşturulması gerekir. Bu anahtar kelimenin yaptığı iş, ilgili tipi üyeleri ile birlikte belleğin **heap** bölgesine yerleştirmektir. Aynı anda per1 referansı neyi işaret edeceğini öğrenir ve artık heap'deki Personel nesnesinin adresini tutmaya başlar. Burada referans tipleri

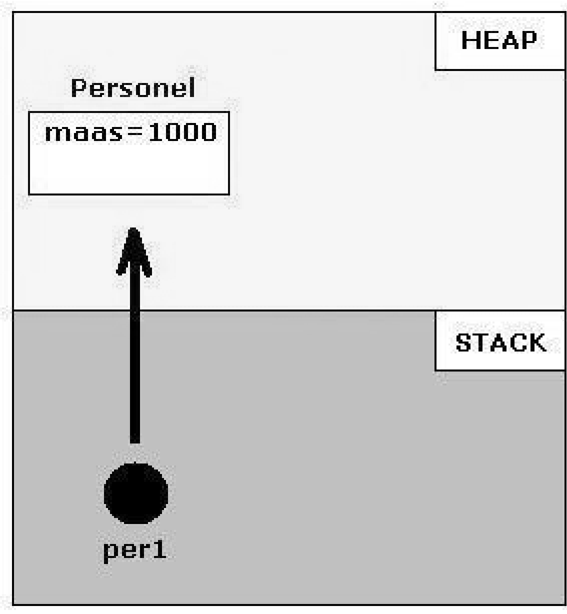
ile ilgili altı çizilmesi gereken bir konu da, sınıf üyelerinden değişken olanların **varsayılan başlangıç değerleri** almasıdır. Bir metot içerisinde kullanılan değişkenlerde kural; (Şu ana kadarki örneklerde hep **Main() metodundan** faydalanıldı) eğer değişkenin başlangıç değeri verilmemişse kullanılamazdır. Bir sınıf içerisinde, o sınıfın üyesi olarak temsil edilen bir değişkene ise başlangıç değeri verilmek zorunda değildir; eğer verilmezse o sınıf çalışma zamanında oluşturulduğunda varsayılan değerleri yerleştirilir. Bu varsayılan değerler; tam sayılar için '0'; ondalık sayılar için '0.0'; karakter veri için boşluk karakteri; mantıksal veri için false; referans tipler için ise (string, object, vb.) değeri yok anlamında 'null' dır. Dolayısıyla bu sınıfın durumunu temsil eden veri olarak yer alan maas değişkeni int tipinde olduğu için varsayılan olarak '0' başlangıç değerini otomatik olarak alır. Eğer istenirse sınıf içerisinde bu değişkenlerin hangi ilk değerle oluşturulacağı belirlenebilir:

```
class Personel
{
      public int maas = 750;
}
```

Bu durumda Şekil 43'deki per1 referansının işaret ettiği Pesonel nesnesinin maas üyesi varsayılan değeri '0' ile değil belirlenen başlangıç değeri '750' ile oluşturulur.

4.Adım:

```
per1.maas = 1000;
```

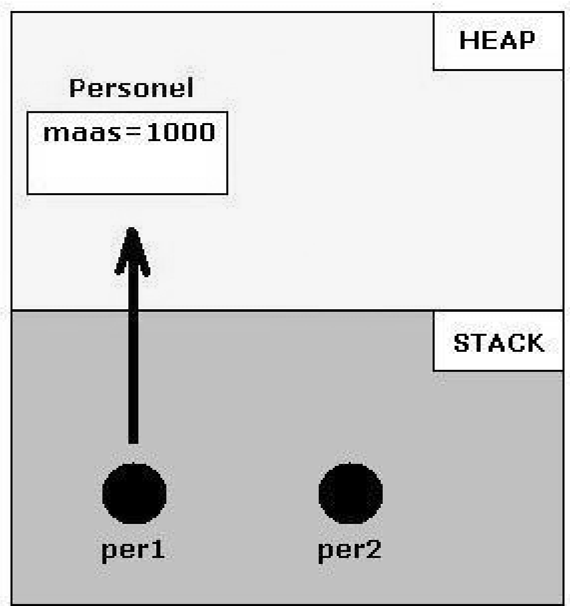


Şekil 44: per1 referansının işaret ettiği Personel örneğinin maas verisi değiştirilir…

- per1 referansı ile erişilebilen maas üyesine yeni değeri atanır.

5.Adım:

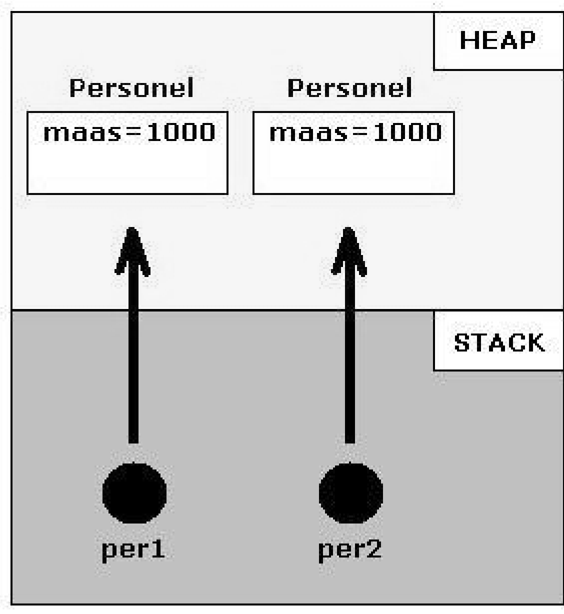
```
Personel per2;
```



Şekil 45: Personel tipinden per2 adında yeni bir referans oluşturulur.

- Personel tipinden per2 adında yeni bir değişken oluşturulur. Bu değişken şu anda neyi işaret edeceğini bilmez. İkinci bir referans değişkeni için de **new** anahtar kelimesi kullanıldığında belleğin heap bölgesinde yeni bir veri alanı açılır. Bu senaryo per2 değişkeni için de geçerlidir. Bu durumda heap bölgesine per2'nin işaret edeceği yeni bir Personel örneği yerleşir. Şöyle ki;
-

```
per2 = new Personel();
per2.maas = 1000;
```

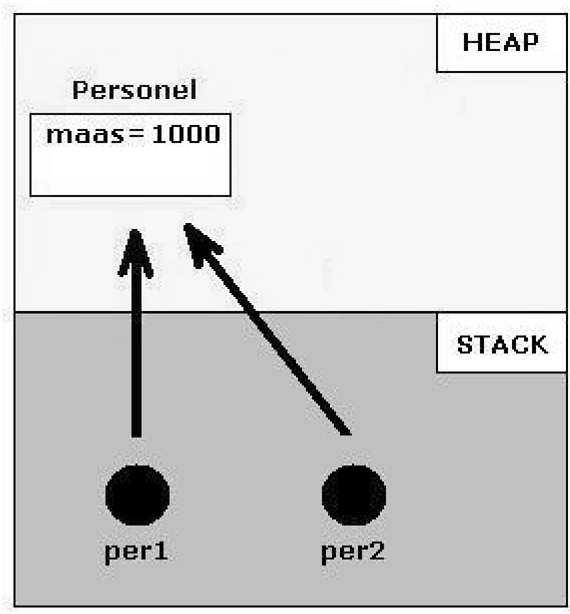


Şekil 46: per2 referansı için de "new" ile yeni bir Personel örneği oluşturulur.

6.Adım:

Şekil 3.13 için yazılan kodların değil de aşağıdaki kod satırının çalıştığını varsayalım:

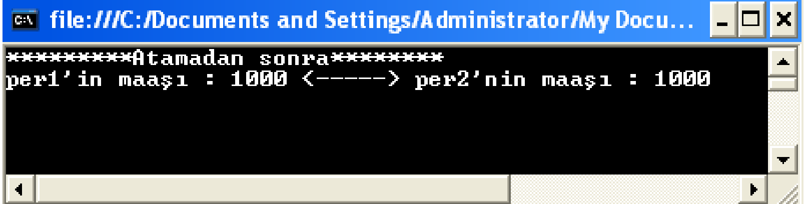
```
per2 = per1;
```



Şekil 47: per1 değişkeninin tuttuğu adres ile per2 referansının tuttuğu adres eşitlenir. Dolayısıyla işaret ettikleri Personel örneği aynı olur.

- Burada yeni oluşturulan per2 referansı, per1 referansı ile eşitlenir. Değer tipli değişkenlerde her değişken kendi değerini üzerinde taşıdığı için böyle bir atamada değerinin bir kopyası oluşturulur. Burada ise gelişmeler oldukça farklıdır. '**per2 = per1'** ataması ile belleğin heap bölgesinde yeni bir nesne oluşturulmaz ve aslında adresler eşitlenir. per1, per2'ye tuttuğu bellek bölgesinin adresini vermiş olur. Bu kod satırıyla, per1 referansının işaret ettiği Personel örneğini artık per2 referansı da işaret edecektir; çünkü per2 değişkeninin elinde de aynı adres vardır. Her iki değişken üzerinden elde edilen maaş bilgileri ekranda gösterilebilir:

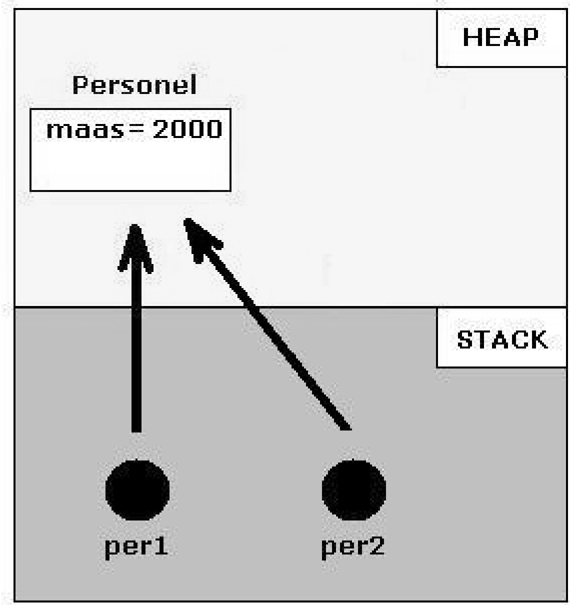
```
Console.WriteLine("per1'in maaşı : {0} <----> per2'nin maaşı :
{1}",per1.maas,per2.maas);
```



Şekil 48: per2, per1'e atandıktan sonra her iki referans üzerinden elde edilen maaşbilgileri

7.Adım:

```
per2.maas = 2000;
```



Şekil 49: Referanslardan biri üzerinden yapılacak değişiklikten diğer referans da etkilenmiş olur; çünkü ikisi de aynı Personel örneğini işaret ederler.

- Mevcut her iki referans değişkeni de heap'de aynı bellek bölgesini işaret ettiği için birinde yapılacak değişiklik otomatik olarak diğerini de etkiler. Çünkü her iki değişkenin de maas üyesinde yapılacak bir değişiklikte gidip güncelleme yapacağı yer aynıdır. İlgili kod ve ekran çıktıları aşağıdaki gibi olur:

```
Console.WriteLine("per1'in maaşı : {0} <----> per2'nin maaşı : {1}",
per1.maas, per2.maas);
```

```
file:///C:/Documents and Settings/Administrator/My Docu...
**********Atamadan sonra*********
per1'in maaşı : 1000 <-----> per2'nin maaşı : 1000
**********Değişiklikten sonra*********
per1'in maaşı : 2000 <----> per2'nin maaşı : 2000
_
```

Şekil 50: per2 referansı üzerinden yapılan değişiklikten sonra maaş bilgileri

Değer Tiplerini ve Referans Tiplerini Karşılaştırma

Değer Tipleri
* Verilerini doğrudan kendi üzerlerinde saklarlar. * Her bir değişken, verisinin bir kopyasını taşır. * Bir değişken üzerinde yapılan değişiklik diğerini etkilemez * Veri, belleğin stack bölgesinde tutulur. * Tanımlamak için veri tipi ve değişken adı yeterlidir. * Null (boş) değer alamazlar.(.NET 2.0 ile birlikte boş değer alabilen değer tipleri ayrıca tanımlanabilir.)

Tablo 6: Değer tiplerinin karakteristik özellikleri

Referans Tipleri
* Verilerinin adreslerini (referanslarını) saklarlar * İki referans değişkeni aynı veriyi işaret edebilir. * Bir değişken üzerinde yapılan değişiklik diğerini etkileyebilir. * Veri, belleğin heap bölgesinde tutulurken; verinin heap bölgesindeki adresini tutan değişken stack bölgesinde tutulur. * Değer tiplerine ek olarak "new" anahtar kelimesi ile oluşturulurlar. * Boş (null) değer alabilirler. Referans değişkeninin 'null' değer alması, işaret edeceği bir nesnenin olmaması anlamına gelir.

Tablo 7: Referans tiplerinin karakteristik özellikleri

Değer ve Referans Tipleri Hiyerarşisi

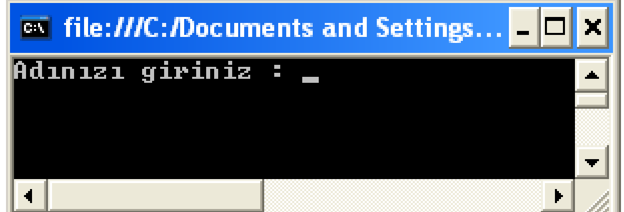
	Değer Tipleri (Value Types)	**Referans Tipleri (Reference Types)**
Kullanıcı Tanımlı Tipler (User-Defined Types)	enum struct	class delegate interface
Önceden Tanımlı Tipler (Built-in Types)	sbyte byte short ushort int uint long ulong float double decimal char bool	string object

Tablo 8: Önceden tanımlı ve kullanıcı tanımlı değer ve referans tipleri listesi

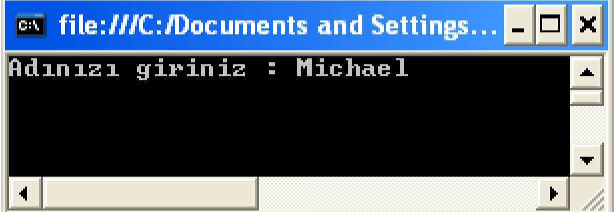
Kullanıcıdan Alınan Değerler ve Parse() Metodu Kullanımı

Kullanıcıdan bir değer alındığında (Şimdilik konsol ekranından, daha sonrası için masaüstü ve web uygulamalarında textBox gibi bir kontrolden olabilir) bu değerin istenilen değişkene doğru tipte atıldığının garanti edilmesi gerekir. Console.ReadLine() ile okunan veriler daha önce görüldüğü gibi her halükarda text tabanlı olacaktır ve doğrudan bir değişkene atılmak istenirse bu değişkenin tipi (eğer ekstra efor harcanmazsa) sadece string olabilir :

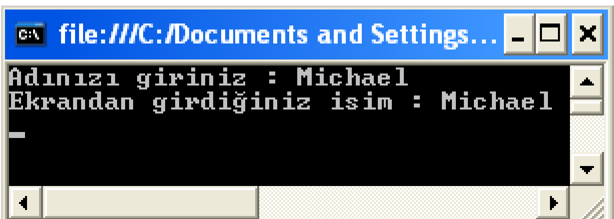
```
Console.Write("Adınızı giriniz : ");
string isim;
isim = Console.ReadLine();

Console.WriteLine("Ekrandan girdiğiniz isim : {0}",isim);
```



Şekil 51: Console.ReadLine() olduğu için kullanıcıdan ENTER beklenir.



Şekil 52: Kullanıcı girişini yapar.



Şekil 53: Ardından ENTER'a basılır ve kodun devamı çalışır.

Kullanıcıdan sayısal bir değer alınırsa, bu değer string tipinde bir değişken üzerinde taşınabilir ve matematiksel bir işlemde kullanılamaz.

```
Console.Write("Ürün fiyatını giriniz : ");
string fiyat;
fiyat = Console.ReadLine();
```

Bu kodun devamında aşağıdaki gibi bir matematiksel işlem gerçekleştirilemez.

```
int zamliFiyat = fiyat + 15; //Derleme-zamanı hatası !!!
```

Çünkü toplama gibi bir matematiksel işleme sokulmaya çalışılan değişkenin değeri '15' olmasına rağmen veri tipi herhangi bir tamsayı ya da ondalık sayı değil bu sayının text tabanlı temsilini yapan 'string' dir. Burada yapılması gereken; '15' verisini doğru tipte ele alabilmektir.

Tam bu soruna yönelik bir çözüm vardır: **Parse() metodu.** Bu metot, doğru tipte ele alınması gereken önceden-tanımlı tipler üzerinden çağrılabilen (yukarıdaki senaryoda int üzerinden) bir üyedir. Bütün önceden tanımlı tiplerin Parse() metotları vardır. Kullanımını incelemek üzere aşağıdaki kod incelenebilir:

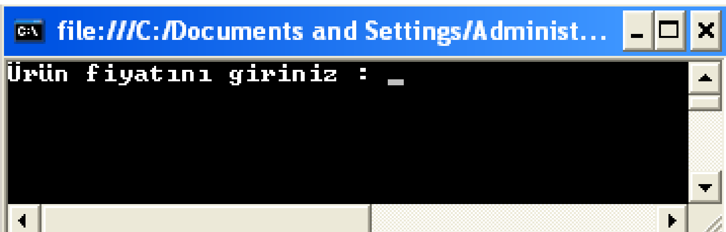
```
using System;

namespace Degiskenler
{
    class ParseMetodu
    {
        public static void Main(string[] args)
        {
            Console.Write("Ürün fiyatını giriniz : ");
            string fiyat;
            fiyat = Console.ReadLine();

            int fiyatInt = int.Parse(
                                  ▲1 of 4▼ int int.Parse (string s)
                                  s: A string containing a number to convert.
        }
    }
}
```

Şekil 54: Parse() metodunun kullanımı

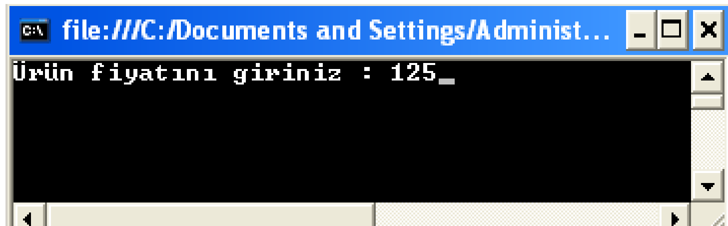
Öyleyse artık kullanıcıdan alınan fiyat bilgisi bir tamsayı değişkeni üzerine alınıp, matematiksel işlemlerde kullanılabilir formata getirilebilir:

```
Console.Write("Ürün fiyatını giriniz : ");
string fiyat;
fiyat = Console.ReadLine();

int fiyatInt = int.Parse(fiyat);
int zamliFiyat = fiyatInt + 15;
Console.WriteLine("Eski fiyat :{0}, yeni fiyat :{1}",fiyatInt,zamliFiyat);
```

Bu kodun ekran çıktısı aşağıdaki gibidir:



Şekil 55: Kullanıcıdan fiyat bilgisi istenir.



Şekil 56: – Kullanıcının veriyi girmesiyle değer string değişken üzerine alınır.

```
file:///C:/Documents and Settings/Administ...
Ürün fiyatını giriniz : 125
Eski fiyat :125, yeni fiyat :140
```

Şekil 57: Kullanıcıdan alınan string değer, parse edilerek int tipte değişkene alınır ve matematiksel işlem sonucu zamlı fiyat hesaplanıp kullanıcıya gösterilir.

Parse() metodu bütün önceden tanımlı tipler üzerinden çağrılabilir; hangi tip üzerinden çağrılmışsa verinin string temsilini, çağrıldığı tipe çevirir. Böylece verinin kendine özgü özelliklerinden faydalanılması sağlanmış olur.

Console sınıfının Read() ve ReadLine() metotları sadece string verileri okurken; Write() ve WriteLine() metotları da sadece string verileri ekranda gösterebilir. Peki o zaman nasıl oluyor da

```
Console.WriteLine("Eski fiyat :{0}, yeni fiyat :{1}",fiyatInt,zamliFiyat);
```

satırı ile ekrana 'int' tipinde iki değişkenin değeri yazdırılabiliyor? Bu sorunun cevabı Console.WriteLine() metodunun, ekrana yazılması için kendisine verilen string olmayan verileri otomatik olarak stringe çevirip verilerin string temsillerini ekrana yazdırmasında gizlidir. Dolayısıyla ekrana yazdırılmak istenen tiplerin string olmayanları her seferinde kontrol edilir ve Parse() işleminin tersi bir davranışla ekranda tamamen string tipte veriler yer alır. Burası için küçük de olsa performansa katkı sağlamak için string olmayan veriler ekrana yazdırılmak istendiğinde, ToString() metodu çağrılırsa arka tarafta yapılan kontrol ve dönüşümlerden kurtulmuş olunur. Bütün önceden tanımlı değer tipli değişken verileri üzerinden ToString() metotları çağrılabilir ve verilerin string temsilleri elde edilebilir, gerekirse string değişken üzerine alınabilir. Öyleyse önceki örnek için WriteLine() metodunun kullanıldığı ikinci satır şu şekilde yeniden düzenlenebilir :

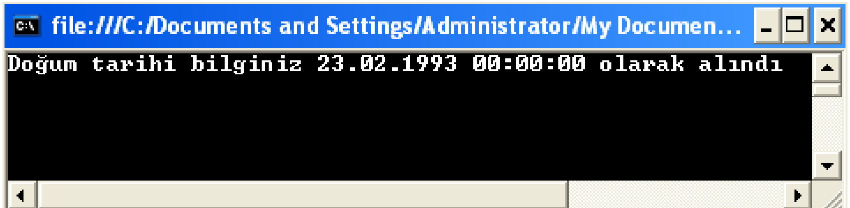
```
Console.WriteLine("Eski fiyat : {0}, yeni fiyat :
{1}",fiyatInt.ToString(), zamliFiyat.ToString());
```

Bu bölümde bahsi geçen Parse() metodunun bütün önceden tanımlı tipler üzerinden çağrılabilmesi ve benzeri şekilde ToString() metodunun bütün tiplerde oluşturulan değişkenler üzerinden çağrılabilmesi, C#'ın nesne yönelimli bir dil olmasının getirilerindendir.

Parse() metodunun kullanımına bir örnek daha verelim. Bu sefer string formatında bulunan doğum tarihi bilgisi System.DateTime tipine Parse() edilsin.

```
//Doğum tarihi formatı varsayılan olarak çalışılan bilgisayarın lokal
//ayarlarından elde edilir. Buradaki format ay/gün/yıl'dır.
DateTime dogumTarihi = DateTime.Parse("02.23.2003");
Console.WriteLine("Doğum tarihi bilginiz {0} olarak alındı",dogumTarihi);
```

Bu Parse() işleminin çıktısı aşağıdaki gibi olur :



Şekil 58: DateTime.Parse() metodunun çalışması ile tarih bilgisinin System.DateTime tipindeki bir değişken ile yeniden kullanılması

DateTime tipindeki **dogumtarihi** değişkeni ekrana yazdırılırken Console sınıfı yine bizim için ToString() metodunu çağırır ve değişkenin değeri ekrana string temsili ile yazdırılır. Bu şekliyle değişkenin değerinde doğum tarihi bilgisi ile birlikte saat,dakika,saniye bilgileri de gelir. Eğer gelmesi istenmiyorsa ya da biraz daha ayrıntılı şekilde gelmesi isteniyorsa bunların belirtileceği yer, DateTime tipindeki değişken üzerinden çağrılabilecek ToLong... ve ToShort… kelimeleri ile başlayan metotlardır. Örneğin saatin tarih bilgisi ile birlikte gelmesi istenmiyorsa ToShortDateString() metodundan aşağıdaki şekilde faydalanılabilir:

```
Console.WriteLine("Doğum tarihi bilginiz {0} olarak alındı", dogumTarihi.ToShortDateString());
```

Tarih bilgisi DateTime struct'ın kullanımı için yazılırken ayraç olarak **.(nokta)**, **-(tire)** ve **/(bölü)** kullanılabilir.

Değişken Kapsama Alanı (Variable Scope)

Öncelike şu ayrımı yapmak gerekir. Bir metot içerisinde tanımlanmış (Örneğin Main() metodu) değişken **lokal değişken (local variable)** adını alırken; bir sınıfın ya da bir yapının üyesi olarak tanımlanan değişken **alan (field)** adını alır. Şu ana kadar ki bütün örnekler Main() metodu içerisinde test edildiği için kullanılan değişkenler lokal değişkenlerdir. Ayrıca daha sonra görülecek döngü ve kontrol deyimleri, özellik (property),indeksleyici (indexer) gibi sınıf ya da yapı üyelerinin içerisinde tanımlanan değişkenler de lokal değişkenlerdir.

Lokal değişkenlerin tanımlanmaları ile birlikte belleğin stack bölgesine yerleştiğini artık biliyoruz. Peki, bu değişkenler ne kadar süre orada kalıyorlar. Acaba uygulama kapanana kadar mı? Yoksa bizim ihtiyacımız kalmayıncaya kadar mı? Aslında ikisi de değil. **Lokal değişkenler, tanımlandıkları süslü parantezler arasında erişilebilirdirler.** Evet, kulağa ilginç geliyor ama tam açıklaması böyle. Bu değişkenler tanımlandıkları bloğa özeldirler ve programda kontrol, bu bloğun sonuna geldiğinde blok içerisinde belleğe çıkarılmış olan bütün lokal değişkenler otomatik olarak bellekten düşer. Özetlemek gerekirse lokal değişkenler sadece tanımlandıkları blok içerisinde kullanılabilirler.

```
...
{
  int a = 5;              //a değişkeni bellekte yerini alır.
}                         // a değişkeni, bellekten düşer.
Console.WriteLine(a);     //Dizayn-zamanı hatası !!!
...
```

Yukarıdaki kod bloğunda a değişkeni tanımlanıp başlangıç değeri verilir ve blok sonuna kadar kullanılır. Tanımlandığı blok sonunda ise bellekten otomatik olarak düşer; sonrasında a değişkeni kapsama alanı dışında kaldığı için, kullanılmaya çalışıldığında henüz dizayn zamanında buna izin verilmez.

Burada yalnız başına kullanılan blok, anlamsız gibi gözükse de aslında lokal değişkenlerin, içinde kullanılabileceği metot ya da daha sonra öğrenilecek deyimler ve diğer üyeleri temsil etmektedir.

```
...
{
  int b;    //b değişkeni belleğe çıkar.
  b = 23;
}           //b değişkeni bellekten düşer.
{
  int b;    //b adında bir değişken belleğe yeniden çıkarılır.
  b = 1;
}           //Yeni b değişkeni bellekten düşer.
```

```
...
```

Yukarıdaki kod bloğunda yapılanlar tamamen legaldir. Burada birbirinden bağımsız iki tane blok yer almaktadır. İlk blok içerisinde bellekte 4 byte'lık bir yer ayrılır ve buraya 23 değeri atılarak blok içerisinde kullanılır. Blok sonuna gelinince bellekteki 4 byte'lık alan tamamen silinir; dolayısıyla hem bellek içerisinde saklanan değer hem de bellek alanına erişmek için verilen isim (değişken adı b) yok edilmiş olur. Artık uygulamanın ilerleyen satırlarında yer alacak yeni bir kod bloğunda 'b' değişken isminin kullanılmaması için herhangi bir sebep yoktur.

```
...
{
  int b;
  b = 23;
      {
         int b; //Derleme-zamanı hatası.
      }
}
...
```

Bu blokta önce 'b' isimli değişken bellekte yerini alır ve başlangıç değeri verilir. Ardından b değişkeninin kapsama alanında bulunduğu bloğun içerisinde yeni bir blok açılır ve burada belleğe yeniden 'b' adında bir değişken çıkarılmaya çalışılır. Ancak bellek bölgesinde aynı anda aynı isimli iki değişkenin yer almasına izin verilmez; çünkü değişken değerlerine erişim, bellek bölgelerine verilen değişken isimleri ile sağlanır ve eğer aynı isimli birden fazla değişkene izin verilseydi işler karışırdı. Dolayısıyla yeni bir değişken ihtiyacı varsa 'b' değil de farklı isimde bir değişken adıyla bu sağlanabilir:

```
...
{
  int b;
  b = 23;
      {
        int c;       //Sorun yok.
        c = 23;
      }
}
...
```

İçerdeki blokta b değişkeni hala erişilebilir olduğu için aynı isimli bir değişken tanımlanamamasına rağmen bu blokta 'b' değişkeninden faydalanılabilir:

```
...
{
  int b;
  b = 23;
```

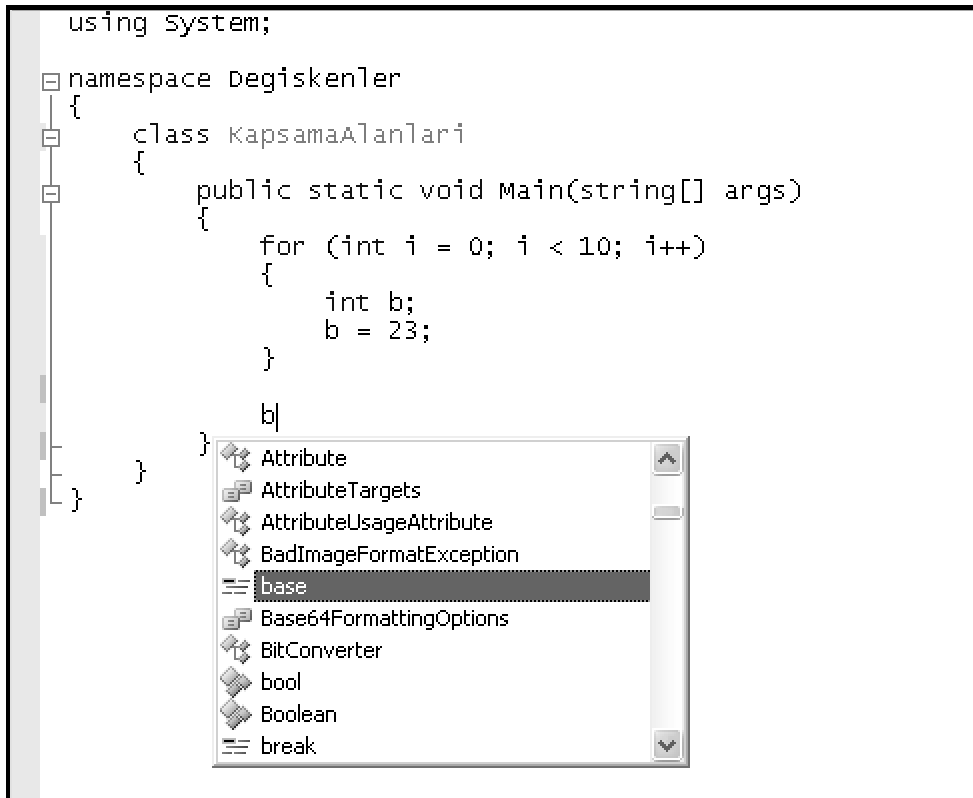
```
        {
          Console.WriteLine(b);      //Geçerli bir kullanım.
        }
}
...
```

Şimdi de değişken kapsama alanı kombinasyonları için son örneği inceleyelim:

```
...
{
        {
          int b;
          b = 23;
        }
        int b = 42; //Derleme zamanı hatası !!!
}
...
```

Burada içerdeki blokta tanımlanan ‘b’ değişkeni aynı zamanda dışardaki bloğun da kapsama alanındadır. O yüzden içerdeki bloğun sonuna gelindiğinde ‘b’ hala bellektedir; dolayısıyla aynı isimli değişken oluşturulmasına izin verilmeyeceği için derleme zamanında bununla ilgili hata alınır.

Ayrıca beklenilenin aksine içerdeki bloktan çıkıp ‘b’ kullanılmak istendiğinde de kullanılamayacaktır.



Şekil 59: İçerdeki blokta tanımlan bir değişkene blok sonunda, dışarıdaki bloğun kapsama alanı dahilinde olsa da erişilemez.

'b' değişkeni yukarıda görüldüğü gibi içerdeki blokta kullanılıp bloğun dışarısında erişilmeye çalışıldığı zaman erişilemez; bunun sebebi 'b' aslında halen bellekte olmasına rağmen tanımlandığı içsel bloğun dışında erişilip yönetilememesidir.(debug edilerek bu görülebilir –'b' değişkeninin ilk tanımlandığı satıra bir kesme işareti <breakpoint> koyup uygulama F5 ile çalıştırılır ve ardından F11 ile satır satır ilerlenerek Locals penceresinden <Menülerden Debug→Windows→Locals ile açılabilir> - değişkenin durumu daha sağlıklı bir şekilde gözlemlenebilir. Ya da kesme işareti koymak yerine uygulama doğrudan F11 ile başlatılarak da aynı işlemler yapılabilir)

Yukarıdaki kod örneğinde kullanılan 'for' deyimi ilerleyen kısımlarda **Kontrol Yapıları, Döngüler, İstisnalar** başlığı altında ele alınacaktır.

Sınıf veya Yapı Üyesi Olarak Değişkenler (Fields)

Lokal değişkenlerin erişim alanlarını artık biraz daha yakından tanıyoruz. Peki ya alanlar? Yani sınıf ya da yapının üyesi olan bir değişken. Burada yanlış anlaşılmaya mahal vermemek için şu noktaya dikkat edilmelidir; lokal değişken ve alan birbirinden farklı değişken tiplerini temsil etmek için kullanılan kavramlar değildirler. 'int' tipinde bir değişken yeri geldiğinde lokal değişken olur; başka bir 'int' değişkeni ise yeri geldiğinde alan olarak kullanılabilir. Eğer değişken metot içerisinde oluşturulmuşsa **lokal değişken (local variable)**; eğer bir sınıf ya da yapı içerisinde oluşturulmuşsa **alan** (field) adını alır. Bir alan, sınıf açılış ve kapanış süslü parantezleri arasında kapsama alanındadır; bu kural yine değişmez. Ayrıca sınıfın içerisindeki bütün üyeler bu alana erişebilir ve kullanabilirler. Bir lokal değişkenin kullanılabilmesi için değişkenin mutlaka başlangıç değerine sahip olması gereklidir, yoksa kullanılamaz. Alanlarda ise kural, eğer bir değişkene başlangıç değeri verilmemişse o değişken tipinin varsayılan değeri geçerli değer olarak kullanılır. Peki nedir bu varsayılan değerler:

Veri Tipi	Varsayılan Değeri
Bool	False
byte	0
char	'\0' (Boş karakter)
decimal	0.0M
double	0.0D
float	0.0F
int	0
long	0L
sbyte	0
short	0
uint	0
ulong	0
ushort	0
string	Null
object	Null

Tablo 9: Alan (field)'ların varsayılan değerleri

Dolayısıyla bütün alanlar, bağlı bulunduğu sınıf gerçek hayata geçirildiği zaman başlangıç değerleri ya bizim vermemizle; ya da verilmezse tablodaki varsayılan değerlerle oluşturulurlar. Bu konunun pekişmesi için 'Referans Tiplerini Anlamak' konusundaki Şekil 3.9 ve Şekil 3.10'u yeniden incelenmesi ve yazıların yeniden okunması faydalı olur.

Referans Değişkenlerinin Boş (Null) Değer Alması

Varsayılan değerlerde dikkati çeken bir nokta, referans tipli string ve object'in varsayılan değerlerinin 'null' olmasıdır. Öncelikle şunu belirtmekte fayda var; sadece referans tipli değişkenler 'null' değer alabilir. Değer tipli değişkenler ise 'null' değer alamazlar; ancak .NET 2.0 ile gelen Boş değer alabilen tipler (Nullable types) ile bu da mümkün olmaktadır. Bir referans değişkeninin görevi, belleğin 'heap' bölgesinde yer alan gerçek veriyi işaret etmektir. İşte bir referans değişkeni sadece tanımlandığında ya da bir alan olarak oluşturulduğunda varsayılan olarak 'null' değerini alır; çünkü henüz belleğin 'heap' bölgesinde işaret edeceği herhangi bir veri yoktur. 'new' anahtar kelimesi ile bu değişkene işaret edeceği verinin bellekteki adresi atanır. Bu noktadan sonra değişkene yapılacak her başvuruda değişken görevini yerine getirir ve esas veriyi (bir sınıf,temsilci ya da arayüz) işaret eder. Belli bir noktadan sonra 'heap' deki verilere ihtiyaç kalmayabilir. Bu durumda referans değişkenine **null** değer atanması yeterli olur.

```
...
per1 = null;
...
```

Doğru Veri Tipine Karar Vermek

Her değişken bellekte kendisine ayrılan byte'ların içerisinde bir değer tutar. Bu değer bellek bölgesinden okunup kullanılır; aynı zamanda başka bir değişkene değer olarak verilebilir. Bellek bölgesinin tipi, orada tutulan değerin tipidir. Birçok değer, birden fazla tiple ifade edilebilir. Mesela '4' değeri sayısal olarak kullanılmak istenirse bu değer 'int' tipiyle temsil edilebileceği gibi byte, sbyte, short, ushort, uint, long, ulong, float, double,decimal tipleriyle de temsil edilebilir. Sadece matematiksel işlemlerde kullanılmayacak ve örneğin kullanıcıya gösterilmek için değerden faydalanılacaksa char ve string tipleri ile de tutulabilir. Burada akla şu soru gelebilir: İstediğimiz değeri bu kadar tip içerisinden hangisinde tutacağız? Buna karar vermek için oluşturulacak değişkenin alacağı değerler yelpazesine bakmakta fayda vardır.

Eğer bir değişken içerisinde haftanın günlerinin sayısal karşılıkları ya da kullanıcıların yaşları tutulacaksa bu değişkenin alacağı değerler kümesi hem pozitif olacaktır hem de 0 ile 130 aralığından daha geniş olmayacaktır. Değişkenin hem negatif değer almayacağını garanti etmek hem de bu değerleri bellekte minimum yer kaplayacak şekilde tutmak için burada karar kılınması gereken tip 'byte' dır.

Eğer bir şirketin kar-zarar durumu temsil edilmek isteniyorsa burada hem negatif hem pozitif sayılara aynı zamanda ondalıklı sayılara ve yüksek basamak değerlerine ihtiyaç duyulacaktır. Dolayısıyla doğru veri tipi decimal olacaktır.

Bir web sitesindeki kayıtlı kullanıcı sayısı bir değişken üzerinde temsil edilmek isteniyorsa bu değer yine negatif değer olmayacaktır. Uygulamanın hedef kitlesine göre bellekte minimum yer kaplayacak olan işaretsiz tam sayı veri tipinde karar kılınmalıdır : ushort, uint veya ulong…

Metin tabanlı tipler söz konusu olduğunda sadece alfabedeki harfleri tutma ihtiyacı varsa tek karaktere izin veren char tipinden; diğer durumlar için ise string tipinden faydalanılabilir. Eğer hiçbiri tek başına bizim işimizi görmüyorsa o zaman yapılacak şey kendi tipimizi yazmak olacaktır (numaralandırıcı (enum), yapı (struct) , sınıf (class) …).

Genel olarak saklamak istenilen veriler için doğru veri tipine karar verirken göz önüne alınması gerekenler şu şekilde özetlenebilir:

- **Değişkenin alacağı değer kümesinin genişliği.** Buna bağlı olarak da veri tiplerinden bu genişliği minimumda sağlayana karar vermek. Daha geniş aralığa sahip olup ihtiyaçtan daha fazlasını sağlayacak veri tipinden kaçınmak gerekir. Bu da değişkenin güvenliğine yani değişkene bizim kontrolümüz dışında istemediğimiz değerlerin verilmesini engellemeye destek verir.

- **Bellekte minimum yer kaplamak.** Bellekte 1 byte yer kaplayan sbyte ile üstesinden gelinebilecek bir iş için 4 byte yer kaplayan int veri tipini kullanmak bellek israfından başka bir şey olmayacaktır. Bu, küçük çaplı uygulamalarda çok fazla farkedilmese de uygulamanın çapı büyüdükçe performansı negatif yönde etkiler.

Değişken İsimlendirme Kural ve Önerileri

Bir değişken tanımlanırken aşağıdaki kural ve önerileri dikkate almak gereklidir :

<u>Kurallar:</u>

- Değişken adı ya bir harf ya da alt çizgi karakteri ile başlayabilir. Sayı ile başlayamaz.

```
double oran;          //geçerli bir değişken tanımlama

double 2nciOran       //derleme-zamanı hatası !!!
```

- İlk karakterin ardından gelen diğer karakterler arasında artık sayı kullanılabilir. Diğer seçenekler yine harf ve alt çizgidir.

```
double oran2;         //geçerli bir değişken tanımlama

double oran_2;        //geçerli bir değişken tanımlama

double oran/2;        //derleme-zamanı hatası !!!

double oran*2;        //derleme-zamanı hatası !!!
```

- C#'da anahtar kelime olarak kullanılan kelimeler değişken adı olamazlar.

```
double public;        //derleme-zamanı hatası !!!

char int;             //derleme-zamanı hatası !!!

int void;             //derleme-zamanı hatası !!!
```

<u>Öneriler:</u>

- Bütün harfleri büyük olan değişken adlarından kaçınılmalıdır.

```
uint SAYAC;                 //tercih edilmemeli...
```

- Daha sonra kodu okuyacak olan kişi için hatta uzun süre sonra okuduğumuzda kendimiz için, verilen değişken adının amacına uygun anlamlı bir isimde olmasına dikkat edilmelidir. Özellikle kısaltma şeklinde verilen değişken isimlerinde bu kısaltmaların anlaşılabilir olmaları önemlidir. Örneğin birinin soyadını saklamak için;

```
string sa = "Aksoy";        //tercih edilmemeli...

string soyad = "Aksoy";     //daha uygun
```

- C#, küçük-büyük harf duyarlı bir dildir. Dolayısıyla iki farklı iş için bir değişkenin sadece baş harfini küçültüp ya da büyültüp yeni bir değişken adı olarak kullanmaktan kaçınılmalıdır. (Bu teknik, daha sonra öğrenilecek olan alan-özellik arasındaki ilişkide kullanılacaktır)

```
bool dogruMu;
bool DogruMu;              //kaçınılmalıdır.
```

- Diğer tavsiyeler ise bütün değişken isimlerinin küçük harfle başlaması ve eğer isim iki kelime ya da daha fazla kelimeden oluşacaksa ikinci kelime ve sonraki kelimenin baş harflerinin büyük yapılmasıdır. Bu tavsiyeler tamamen belli bir kod standardı oluşturmak için vardır. Bir takım içerisinde çalışıldığında herkesin bu ve bunun gibi var olan diğer standartlara uyarak kod yazması, yazılan kodun okunurluğunu arttırır. Yani kodu yazarken kendimizi düşünmüyorsak başkalarını düşünerek bu standartlara uymak yerinde bir davranış olur.

```
char harf;                //baş harfler küçük tercih edilmeli.

char basHarf;             //ikinci kelimenin baş harfi büyük

sbyte disKapininBoyu;     //ilk kelimeden sonraki her kelimenin baş
                            harfi büyük
```

Literatürde bu isimlendirme şekline **camelCasing** (camel isimlendirme) adı verilir[6].

Kaçış Karakterleri (Escape Sequences)

Metin tabanlı veriler ekrana çıktı olarak yazılırken (Gerek konsol ekranında, gerekse windows uygulamalarında) bazı karakterler bunun nasıl yapılacağı ile ilgili bilgiler içerir. Örneğin alt satıra geçmek, **tab** aralığı verdirmek vb. Bu karakterlere **kaçış karakterleri (escape characters)** adı verilir. Bütün kaçış karakterleri ters bölü işareti ile başlar ve ardından belli bir karakter alırlar. Ters bölü işareti, kendisinden sonra gelen karakterin özel anlamını gizlemekle görevlidir. Örneğin 'Bilinçli Dönüşüm' konusunda Şekil 3.32 ile Şekil 3.33 arasındaki kod bloğunda Console.WriteLine() ile ekrana bir yazı yazdırılırken string ifadenin içerisinde kullanılan '\n' kaçış karakterlerine bir örnektir. En sık kullanılanlarına ve anlamlarına bakalım:

(6)İsimlendirme kuralları ile ilişkili olarak genel kaynak : Juval Lowy – C# Coding Standards -

- \' String ifadenin içerisinde tek tırnak görünmesini sağlar.
- \" String ifadenin içerisinde çift tırnak görünmesini sağlar.
- \\ String ifadenin içerisinde ters bölü işareti görünmesini sağlar. Dosya yolu belirtirken faydalı olabilir.
- \a Sistem bip sesini tetikler.
- \n Bir satır atlatmak için kullanılır.
- \r Enter tuşunu temsil eder.
- \t Tab tuşunu temsil eder. String ifadenin içerisinde bir tab boyu (varsayılan olarak 8 karakter) aralık bırakır.

Aşağıda kaçış karakterlerinin örnek kullanımı ile ilgili kodlar yer almaktadır.

```
string s = "Herkese \tmerhabalar \t, hoş geldiniz...";
Console.WriteLine(s);

Console.WriteLine("Herkes \"Merhaba Dünya\"yı sever.");
Console.WriteLine("c:\\Dosyalarim\\Uygulamam");
Console.WriteLine("Sonunda\n\tbitti");
```

```
Kaçış karakterleri ile çalışmak
Herkese           merhabalar       , hoş geldiniz...
Herkes "Merhaba Dünya"yı sever.
c:\Dosyalarim\Uygulamam
Sonunda
       bitti
```

Şekil 60: Kaçış karakterleri örnekleri

Kaçış karakterlerine ek olarak string ifadenin başına koyularak kaçış sekansının özel anlamını gizleyen '@' işareti de kullanılabilir, özellikle dizin yolu belirtirken faydalı olabilir.

```
Console.WriteLine(@"c:\Dosyalarim\Uygulamalarim\Karakterler.cs");
```

Aynı zamanda '@' işareti bir string ifadenin içerisindeki beyaz boşlukların korunmasını ve ekranda o şekilde görüntülenmesini sağlar.

```
Console.WriteLine(@"Bu
                       çok
                           ama
                               çok
                    uzun
                        bir
                            yazı...");
```

Ve son olarak '@' işareti ile çift tırnak iki defa yazılarak ekranda gösterilebilir:

```
Console.WriteLine(@"Herkese ""selam"" söyledi...");
```

Tip Dönüşümleri

C#'da iki farklı tip 'tür dönüşümü' vardır :

1. Bilinçsiz Dönüşüm (Implicit Conversion)
2. Bilinçli   Dönüşüm (Explicit Conversion)

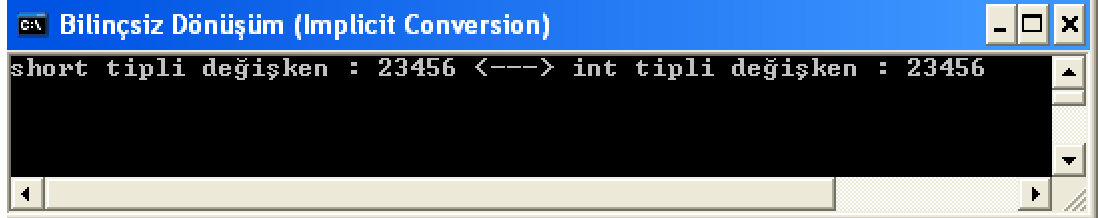
Bilinçsiz Dönüşüm (Implicit Conversion)

'Doğru Veri Tipine Karar Vermek' konusunda geçtiği üzere '4' gibi bir değer, işaretli işaretsiz bütün tam sayı ve ondalık sayı veri tiplerinde saklanabilir. Bu bilgiden yola çıkarak tip dönüşümü, yeni bir tipte değer oluşturur; bu değer eski tipteki değişkenin değeri ile eşit olacaktır. Bu dönüşümde değer, yeni veri tipi ile daha geniş bir yelpazeye sahip oluyorsa; dönüşüm sessizce gerçekleşir. Buna **Implicit Conversion** (Bilinçsiz Dönüşüm) denir. Örneğin önceden short tipinde oluşturulmuş bir değişken int tipinde bir değişken üzerine atanmak isteniyorsa bu genişleyen dönüşümdür ve bilinçsiz dönüşüm (implicit conversion)'e örnektir; çünkü short tipinde oluşturulmuş bir değişkenin alabileceği değerler kümesi tam sayı olarak -32.768 ile 32.767 arasında iken dönüştürüldüğü yeni tipte değişkenin alabileceği değerler kümesi -2.147.483.648 ile 2.147.483.647 arasındadır. Yani eski veri tipi short'da alması muhtemel bütün değerler, yeni veri tipi int'in alabileceği değerler kümesinin içerisinde kalmaktadır.

```
short ilkDegisken = 23456;
int sonrakiDegisken = ilkDegisken; //sonrakiDegisken'in değeri 23456
Console.WriteLine("short tipli değişken : {0} <---> int tipli değişken :
```

```
{1}",ilkDegisken,sonrakiDegisken);
```

Bu dönüşüm, ekstra kod yazmaya gerek kalmadan her zaman gerçekleşir ve dönüşüm sırasında hiçbir zaman veri kaybı yaşanmaz, yani eski değişkenin değeri neyse yeni değişkenin değeri de aynı olur.



Şekil 61: Bilinçsiz dönüşümde (Implicit Conversion) veri kaybı yaşanmaz.

Yukarıdaki örnekte olduğu gibi genişleyen bütün tip dönüşümleri sessizce gerçekleşir. Aşağıdaki tablo genişleyen dönüşümlerin hangi tipler arasında gerçekleneceğini listelemektedir:

Tip	Veri kaybı yaşanmadan dönüştürülebilecek tipler
byte	ushort, short, uint, int, ulong, long, float, double, decimal
sbyte	short, int, long, float, double, decimal
short	int, long, float, double, decimal
ushort	uint, int, ulong, long, float, double, decimal
char	ushort, uint, int, ulong, long, float, double, decimal
int	long, double, decimal
uint	long, double, decimal
long	decimal
ulong	decimal
Float	double

Tablo 10: Genişleyen dönüşümleri mümkün kılan tipler

Bir sayısal tipin alabileceği minimum ve maksimum değerleri kod yoluyla öğrenmek için ilgili tipin MinValue ve MaxValue özellikleri kullanılabilir. Özelliklerin (properties) çalışma mekanizması daha sonra öğrenilecek bir konu olup şimdilik sadece yapı ve sınıfların bir üyesi olduğunun bilinmesi yeterlidir.

```
...
public static void Main(string[] args)
{
      Console.WriteLine("int tipinin taşıyabileceği minimum değer : {0} ",
int.MinValue);
      Console.WriteLine("int tipinin taşıyabileceği maximum değer : {0} ",
int.MaxValue);
      Console.ReadLine();
}

...
```

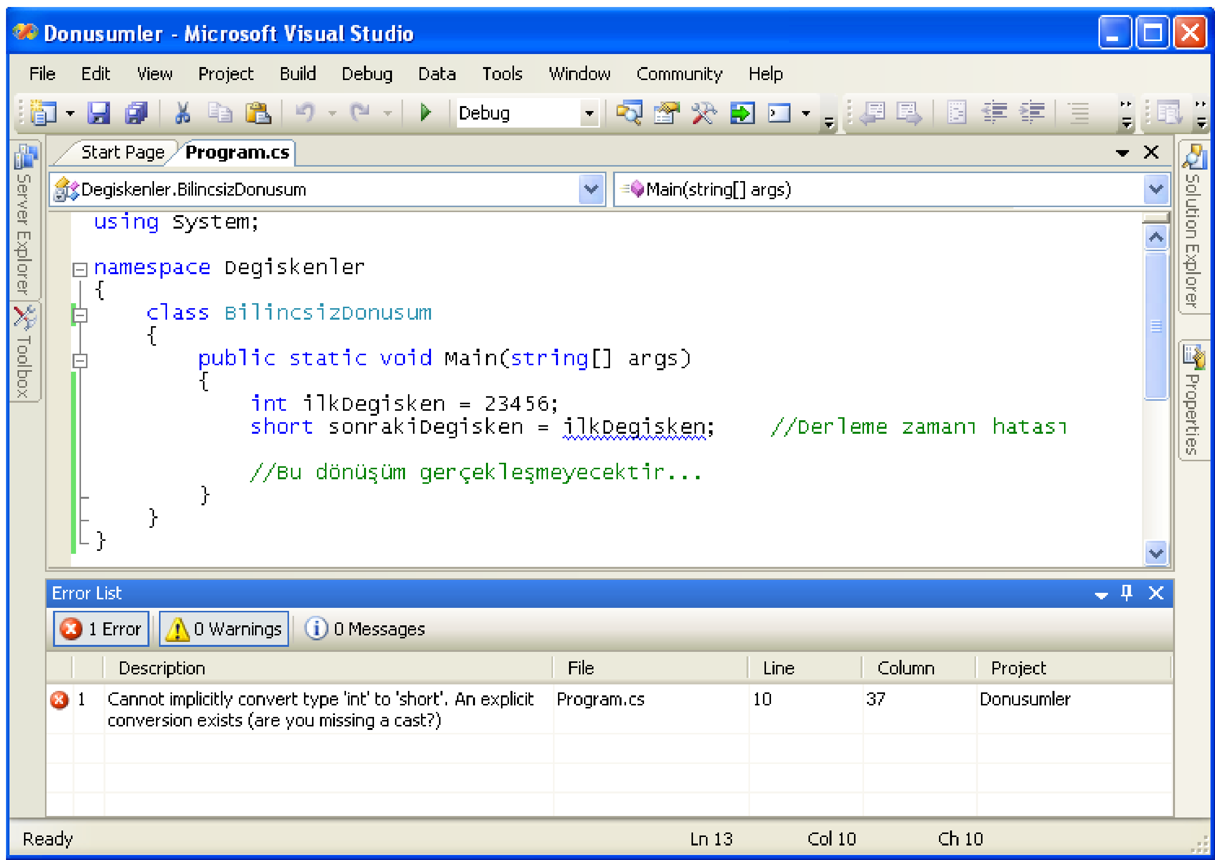
Sayısal değerlerin yelpazelerinin elde edilmesi

```
int tipinin taşıyabileceği minimum değer : -2147483648
int tipinin taşıyabileceği maximum değer : 2147483647
```

Şekil 62: Sayısal tiplerin taşıyabilecekleri minimum ve maksimum değerlerin elde edilmesi

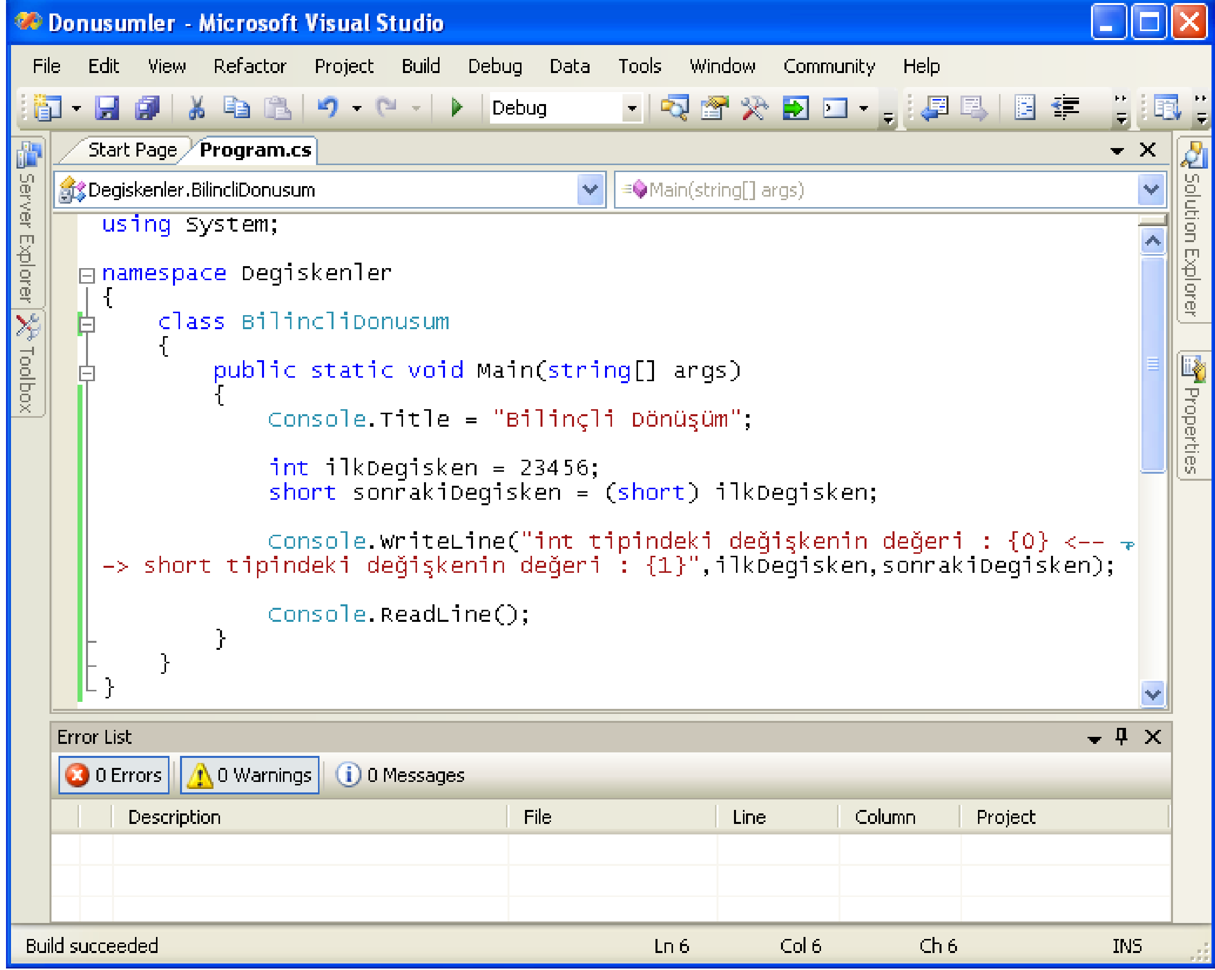
Bilinçli Dönüşüm (Explicit Conversion)

Yapılan tip dönüşümünde değer, yeni veri tipi ile daha geniş bir yelpazeye sahip oluyorsa; dönüşümün sessizce gerçekleşeceğini ve buna 'Implicit Conversion' denileceğini biliyoruz. Bir değişken üzerindeki değer, yeni veri tipi ile daha dar aralığa sahip oluyorsa; dönüşüm sessizce gerçekleşmez; çünkü eski veri tipindeki değişkenin değeri, yenisinin taşıyabileceği değerler kümesi dışında kalabilir. Dönüşüm sırasında olmasa bile böyle bir ihtimal olduğu için C#'da buna izin verilmez. (Bu durum, diğer diller arasında farklı şekillerde ele alınır) Örnek olarak bilinçsiz dönüşümdeki senaryonun tersi ele alınabilir:



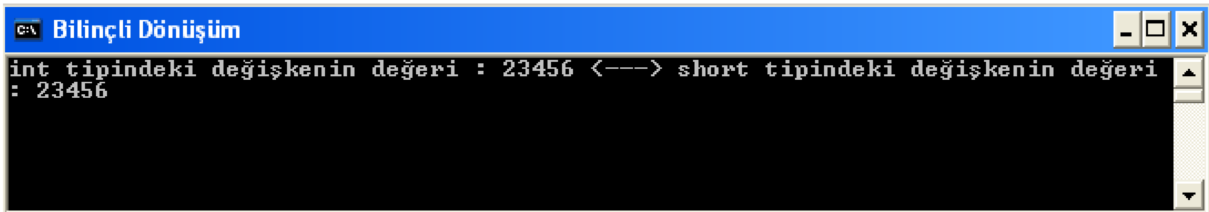
Şekil 63: 'int' tipindeki değişkenin değeri daha dar değerler yelpazesine sahip 'short' tipinde bir değişkenin üzerine sessizce alınamaz.

Burada int tipindeki **ilkDegisken** adlı değişkenin değeri (23456), short tipinin taşıyabileceği bir değer olmasına rağmen **sonrakiDegisken** adlı değişkenini üzerine atanmaya çalışılırken derleyici buna izin vermemektedir. **Yapılması gereken, bunun bilinçli olarak yapıldığını derleyiciye söylemektir.** Bunu yapmanın yolu, cast operatörünü kullanmaktır:



Şekil 64: 'int' tipindeki değişkenin değeri daha dar değerler yelpazesine sahip 'short' tipinde bir değişkenin üzerine cast operatörü ile alınabilir.

Bu kod bloğu sonucunda dönüşüm gerçekleşir ve uygulama çalışır. Bu şekilde cast operatörü kullanılarak yapılan dönüşüme **Explicit Conversion** (Bilinçli Dönüşüm) denir. Ayrıca bu dönüşüm, dönüştürmede rolü olan tiplerin taşıyabilecekleri değer aralıklarına göre, daralan bir dönüşümdür.



Şekil 65: 'int' tipindeki değişkenin değeri daha dar değerler yelpazesine sahip sahip 'short' tipinde bir değişken üzerine alındı.

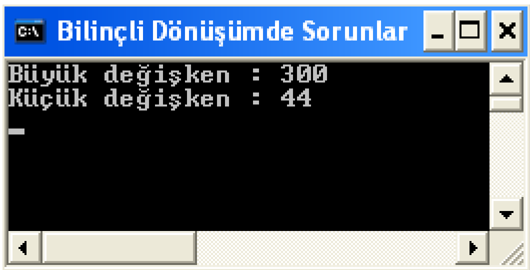
Cast operatörü, dönüştürme yapılan satırda eşitliğin sağ tarafına parantez içerisinde eklenir ve burada dönüştürme yapılacak tip belirtilir. Bu şekilde derleyiciye **'Hey derleyici, ben ne yaptığımı biliyorum'** demiş olunur. Derleyici, bu dönüşüm sonrasında yeni tipteki değişkenin, eski tipteki değişkenin değerini taşıyamama ihtimali olmasına rağmen kendisine verilen garanti ile bu dönüşümün yapılmasına izin verir.

```
short sonrakiDegisken = (short) ilkDegisken;
```

Peki ya int tipindeki değişkenin değeri, gerçekten derleyicinin şüphelenmesini haklı çıkaracak şekilde short tipinin taşıyabileceği bir değer değilse? Bu durumda gerçekleşecekleri yine kod üzerinden analiz edelim:

```
...

//sbyte tipinin taşıyabileceği değerler aralığı : -128 ile 127
short buyukDegisken = 300;
sbyte kucukDegisken = (sbyte) buyukDegisken;

Console.WriteLine("Büyük değişken : {0}\nKüçük değişken :
{1}",buyukDegisken,kucukDegisken);

...
```

Yukarıdaki kod başarılı bir şekilde derlenir. (Visual Studio 2005 kullanılıyorsa Ctrl + Shift + B ya da Build menüsünden Build Solution ile) Çünkü derleyiciye cast operatörü ile bu işin bilinçli olarak yapıldığı bildirilmiş olur. Ancak dönüştürülmeye çalışılan sbyte tipindeki değişkene verilen değere bakılırsa, bu değerin sbyte tipinin taşıyabileceği aralık dışında olduğu hemen görülür. Bunun kontrolü derleme zamanında yapılmadığı için kodun derlenmesinde bir sorun oluşmaz. Zaten bu kontrol derleme zamanında yapılamadığı için derleyiciye bunun geçerli bir dönüşüm olduğunu cast operatörü ile belirtme ihtiyacı duyulmaktadır. Öyleyse uygulamanın nasıl çalıştığına bakalım:



Şekil 66: Bilinçli dönüşüm yaparken tipler arasındaki değer uyuşmazlığı

Sonuç oldukça ilginçtir; dönüşüm sırasında değer kaybı bir yana anlamsız bir değerle karşılaşıldı. O zaman şu soru sorulmalı: "Tamam, derleme zamanında önceki değişkenin sahip olduğu değer kontrol edilemediği için cast operatörü kullanıldı; peki çalışma zamanında bunun kontrolü niye yapılmadı?" Cevap ise şu: "Eğer bizim kontrolümüz dışında ilk değişkenin değeri, dönüştürme yapılan değişkenin taşıyabileceği değerler yelpazesinin dışında kalacaksa, bu durumda çalışma zamanının tanımlanamayan ve güvenli olmayan değer getirmesi yerine en azından çalışma zamanı hatası üretmesi için **'checked'** anahtar kelimesi kullanılmalı" (Çalışma zamanı hatalarına istisna (exception) denir ve böyle durumlarda kullanıcıya uygun bir hata mesajı göstermek gerekir. İleriki bölümlerde bu konu yer almaktadır). 'checked' anahtar kelimesi, bu senaryodaki gibi bir tipin taşıyabileceği değer taşması durumunu kontrol edip çalışma zamanında böyle bir durumla karşılaşıldığında çalışma zamanı hatası oluşmasını sağlar. Öyleyse kodu aşağıdaki gibi güncelleyelim:

```
...

short buyukDegisken = 300;
sbyte kucukDegisken;
checked
{
      kucukDegisken = (sbyte)buyukDegisken;
}

Console.WriteLine("Büyük değişken : {0}\nKüçük değişken : {1}",
buyukDegisken, kucukDegisken);

...
```

Bu durumda istenildiği gibi çalışma zamanında hata oluşacaktır. Daha sonraki bölümlerde öğrenilecek olan "Çalışma Zamanı Hatalarını Ele Alma" konusunda böyle bir durumda kullanıcıya nasıl uygun bir hata mesajı gösterileceği öğrenilecektir.

```
    {
        public static void Main(string[] args)
        {
            Console.Title = "Bilinçli Dönüşümde Sorunlar";

            //sbyte tipinin taşıyabileceği değerler aralığı : -128 ile 127
            short buyukDegisken = 300;
            sbyte kucukDegisken;
            checked
            {
                kucukDegisken = (sbyte)buyukDegisken;
            }

            Console.WriteLine("Büyük değişken : {0}\nKüçük değişken : {1}",buyukDegisken,kucukDegisken);

            Console.ReadLine();
        }
    }
}
```

Bilinçli Dönüşümde Sorunlar

OverflowException was unhandled

Arithmetic operation resulted in an overflow.

Troubleshooting tips:
Make sure you are not dividing by zero.
Get general help for this exception.

Search for more Help Online...

Actions:
View Detail...
Copy exception detail to the clipboard

Şekil 67: Bilinçli dönüşümde 'checked' anahtar kelimesi sayesinde yanlış değer ataması sonucunda çalışma zamanı hatası alma

Aşağıdaki tablo daralan tip dönüşümlerinde değer uyuşmazlığı gibi bir durumda 'checked' blokları kullanılmışsa, çalışma zamanı hatası oluşturacak tipleri listelemektedir.

Tip	Dönüştürülebilecek tipler
byte	sbyte
sbyte	byte, ushort, uint, ulong
short	byte, sbyte, ushort
ushort	byte, sbyte, short
İnt	byte, sbyte, short, ushort, uint
uint	byte, sbyte, short, ushort, int
long	byte, sbyte, short, ushort, int, uint, ulong
ulong	byte, sbyte, short, ushort, int, uint, long
decimal	byte, sbyte, short, ushort, int, uint, long, ulong
float	byte, sbyte, short, ushort, int, uint, long, ulong
double	byte, sbyte, short, ushort, int, uint, long, ulong

Tablo 11: Daralan dönüşümlerde kullanılabilecek tipler

Operatörler

Aritmetik Operatörler

C#, basit aritmetik işlemlerin yapılmasını sağlayan toplama (+), çıkarma (-), çarpma (*), bölme (/) operatörlerinin yanında bir de kalan (%) operatörünü

desteklemektedir. Kalan operatörünün işlevi, bir sayının başka bir sayıya bölümünden kalan sayıyı vermektir. Operatörler sol ve sağına aldıkları değerler üzerinde işlem yaparlar ve sonucunda yeni bir değer üretirler. Operatörlerin, üzerinde işlem yaptığı bu değerlere **operand** adı verilir.

Aşağıdaki örnekte **toplamSaat** değişkeninin değeri, alınan bir eğitimin bir gündeki ders saati ile toplam eğitim gün sayısı çarpılarak elde edilmektedir.

```
...

int toplamSaat;
toplamSaat = 5 * 60;
Console.WriteLine("Eğitim toplam saati : {0}",toplamSaat);

...
```

Şekil 68: Aritetik operatör kullanımına örnek

Bu işlemde 5 ve 60 işlemin operandları iken, '*' aritmetik operatör, '=' ise atama operatörüdür.

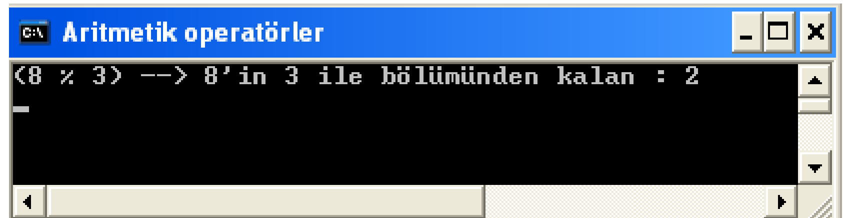
İşlem, değişkenler üzerinden de gerçekleştirilebilir:

```
...

int toplamSaat;
int gunlukDersSaati = 5;
int toplamGun = 60;
toplamGun = gunlukDersSaati * toplamGun;
Console.WriteLine("Eğitim toplam saati : {0}",toplamGun);

...
```

Çıktı yukarıdaki ile aynı olur. Aşağıdaki kod, kalan operatörünün işlevini ele almaktadır.

```
...

int kalan;
kalan = 8 % 3;
Console.WriteLine("(8 % 3) --> 8'in 3 ile bölümünden kalan : {0}",kalan);

...
```

Yukarıdaki kodun çıktısı aşağıdaki gibi olacaktır :



Şekil 69: Kalan operatörünün kullanımına örnek

Kalan operatörü, solundaki operandı sağındaki operanda bölüp, bölme işlemi sonucunda kalan değeri verir. Bu örnekte de 8 sayısının 3 ile bölümünden kalan değerini vermiştir.

Değişkenler üzerinden aritmetik işlemler yapılıken dikkat edilmesi gereken bir nokta vardır. Aşağıdaki kodu inceleyelim:

```
...

short a = 23;
short b = 24;
short c = a + b;        //Derleme zamanı hatası

...
```

Tamamen geçerli görünen bir kod olmasına rağmen derlenmeyecektir. Sebebi ise iki tane short tipli değişkenin toplamının int değeri aralığına girebileceğidir. Dolayısıyla büyük türü, küçük türe bilinçsiz olarak dönüştürme durumu söz konusudur ki; buna zaten izin verilmemektedir. Bu nedenle bilinçli şekilde tip dönüşümü yapılmalıdır.

```
...

short a = 23;
short b = 24;
short c = (short) a + b;      //Yine derleme zamanı hatası

...
```

Yukarıdaki kod yine derlenmez; cast operatörü sadece 'a' değişkeni için çalışır ve bir şey değişmez. Yine iki tane short tipli değişken toplanıp int tipinde bir sonuç elde edilir. a + b kodu eğer parantez içerisine alınırsa cast operatörünün, bu iki değişkenin toplamından elde edilen değere uygulanması gerektiği sağlıklı bir şekilde derleyiciye iletilmiş olur.

```
...

short a = 23;
short b = 24;
short c = (short) (a + b);

...
```

Burada tabiki elde edilen sonucun short tipinin taşıyamayacağı bir değer olma olasılığı mevcut. Dolayısıyla bu tarz bir işlem yaparken 'checked' anahtar kelimesine ihtiyaç duyulabilir.

- Bu tip uyuşmazlığı durumu sbyte,byte,ushort,short ve int tiplerinin birbirleriyle ve aralarında yaptıkları işlemlerde geçerlidir. Sayılan tiplerde değişkenlerin birbirleriyle ve aralarında yaptıkları işlemlerde elde edilen değerin tipi int olacaktır.
- **uint** tipli bir değişken ile sbyte,byte,ushort,short tipli değişkenlerden herhangi biri arasında yapılan aritmetik işlemde oluşan sonuç uint tipinde olur.
- **uint** tipli bir değişken ile int tipli değişken arasında yapılan aritmetik işlemde oluşan sonuç long tipinde olur.
- **long** tipli bir değişken ile ulong hariç diğer bütün tam sayı tiplerindeki değişkenler arasında yapılan aritmetik işlemde oluşan sonuç long tipinde olur.
- **long** tipli bir değişken ile ulong tipli değişken aritmetik işleme sokulamaz.
- **float** tipli bir değişken ile bütün tam sayı tipli ve kendi tipinde değişkenler ile yapılacak aritmetik işlemlerde oluşan sonuç float tipinde olur.
- **double** tipli bir değişken ile bütün tam sayı tipli değişkenler,float ve kendi tipindeki bir değişken arasında yapılacak işlemler sonucunda oluşacak sonuç double tipinde olur.

- **decimal** tipli bir değişken ile ondalıklı sayıları tutan double ve float tipli değişkenler arasında aritmetik işlem yapılamaz.
- **decimal** tipli bir değişken ile bütün tam sayı tipli değişkenler arasında yapılacak işlemler sonucunda oluşan sonuç decimal tipinde olur.
- Değişken kullanmadan aritmetik işleme sokulan iki sayı eğer tam sayı ise bu değerlerin tipleri derleyici tarafından otomatik olarak int olarak algılanır ve işlem sonuçları ona göre üretilir. Örneğin;

```
    Console.WriteLine(5 / 2);
```

gibi bir işlemin sonucu 2.5 değil 2 olacaktır; çünkü 5 ve 2 değerleri int olarak hesaplanacak ve sonuçları da int olacaktır. Dolayısıyla int tipli bir değer 2.5 sonucunun ondalık kısmını taşıyamayacağı için sonuç 2 olarak üretilecektir. Eğer işlem;

```
    Console.WriteLine(5.0 / 2.0);
```

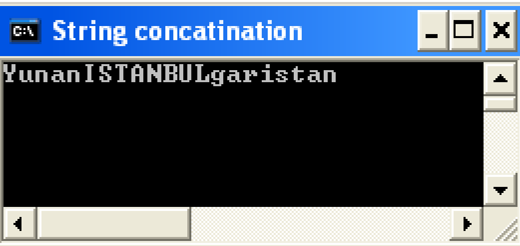
olarak yapılsaydı bu sefer ondalık sayıların varsayılan tipi devreye girecektir; bu tip ise 'double' dır. Bölme işleminin operandlarının tipleri double olacağı için sonuç double tipinde olacak ve 2.5 olarak elde edilecektir. Peki ya bir operand tam sayı, diğeri ondalık sayı ise? Derleyici böyle bir işlemle karşılaşınca sonucun varsayılan tipini double olarak belirler ve;

```
    Console.WriteLine(5 / 2.0);
```

sonucu 2.5 olarak üretir.

Yukarıda verilen bilgilerden de anlaşılacağı üzere bütün aritmetik operatörler bütün veri tipleri ile kullanılamamaktadır. Bunlara ek olarak aritmetik operatörler string ve bool tipleri üzerinde de kulanılamaz. Ancak bir istisna vardır, o da artı (+) operatörünün string tipindeki değerler için kullanıldığında farklı bir anlama gelmesidir:

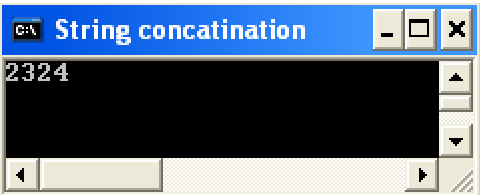
```
...

string deger1 = "YunanISTAN";
string deger2 = "BULgaristan";
Console.WriteLine(deger1 + deger2);

...
```



Şekil 70: Artı operatörünün string tipi ile kullanımına örnek

Artı operatörü string tipindeki değerlerle kullanıldığında sağındaki ve solundaki değerleri text olarak birleştirme özelliğine sahiptir. Buna **string concatenation (birleştirme)** denir. Dolayısıyla aşağıdaki gibi string iki sayının toplamı tam sayı olarak iki sayının toplanmasındaki sonucu vermeyecektir.

```
Console.WriteLine("23" + "24");
```

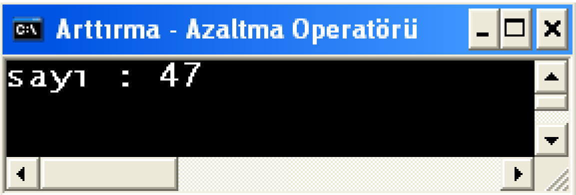


Şekil 71: String tipindeki iki sayının artı operatörü ile kullanımı

Birleşik Atamalar (Compound Assignment)

Bir değişkenin değerinin, üzerine bir sayı eklenerek değiştirildiği durumlarda yazacağımız kod aşağıdaki gibi olacaktır:

```
...
int sayi = 23;
sayi = sayi + 24;   //değişkenin değeri 24 arttırılıyor.
Console.WriteLine("sayı : {0}",sayi);

...
```



Şekil 72: Bir değişkenin değerini bir sayı ile basit aritmetik işleme sokma

C#, bu atamayı yapmanın daha kolay bir yolunu önermektedir:

```
...
int sayi = 23;
sayi += 24;             //sayi = sayi + 24 ile eşdeğer
Console.WriteLine("sayı : {0}",sayi);

...
```

Bu birleşik atamalar, bütün aritmetik operatörler için geçerlidir:

```
sayi += 2;
sayi -= 2;
sayi *= 2;
sayi /= 2;
sayi %= 2;
```

Arttırma – Azaltma Operatörleri

Bir değişkenin değeri 1 artırılmak istenirse izlenecek yol muhtemelen şu olacaktır:

```
...
int sayi = 23;
sayi = sayi + 1;        //Değişkenin değeri 1 arttırılır.
Console.WriteLine("sayı değişkeninin son değeri : {0}",sayi);

...
```

ya da

```
...

int sayi = 23;
sayi += 1;         //Değişkenin değeri 1 arttırılır.
Console.WriteLine("sayı değişkeninin son değeri : {0}",sayi);

...
```

Arttırma - Azaltma Operatörü

```
sayı değişkeninin son değeri : 24
```

Şekil 73: Bir sayısal tipli değişkenin değerini 1 arttırmak

Bir değişkenin değerini 1 arttırmanın C#'da daha kolay bir yolu vardır: Artırım operatörü bu işi yapmak için özelleşmiştir:

```
sayi++;
```

Aynı şekilde bir değişkenin değerini 1 azaltmak için de kullanılabilir:

```
Sayi--;
```

Arttırma ve azaltma operatörleri değişkene son ek olarak gelebildiği gibi ön ek olarak da gelebilir. Böyle olduğunda davranışı biraz farklı olacaktır:

```
...

int sayi = 23;
int baskaSayi = sayi++;
Console.WriteLine("sayı : {0} <---> baskaSayi : {1}",sayi,baskaSayi);

...
```

Sayi değişkeninin değeri 1 arttırılarak 24 yapılır; ama önemli olan nokta **baskaSayi** değişkenine, arttırılmış değerini mi geçiridiği yoksa arttırılmadan önceki değerini mi geçirdiğidir? İşte burada operatörün ön ek mi son ek mi olduğu önemlidir; çünkü atama operatörü ile arttırım operatörü arasında bir öncelik sırası mevcuttur. Bu sıra, arttırım operatörü son ek olduğunda atamanın; ön ek olduğunda ise arttırımındır. Dolayısıyla yukarıdaki kod örneğinde arttırım operatörü son ek olduğu için öncelik 'atama operatörü'nün olur ve önce **sayi** değişkeninin değeri **baskaSayi** değişkenine 23 olarak atanır; daha sonra ise 'sayi' değişkeninin değeri arttirilir. Ekranda çıktısı aşağıdadır:

Arttırma - Azaltma Operatörü

```
sayı : 24 <---> baskaSayi : 23
```

Şekil 74: Arttırma operatörünün son-ek olarak kullanıldığı durum.

Tam tersi, yani arttırım operatörünün ön-ek olarak kullanıldığı durumda ise;

```
...

int sayi = 23;
int baskaSayi = ++sayi;
Console.WriteLine("sayı : {0} <---> baskaSayi : {1}",sayi,baskaSayi);

...
```

Önce **sayi** değişkeninin değeri 1 arttırılıp 24 yapılır ve ardından 24 değeri **baskaSayi** isimli değişkene atanır; dolayısıyla her iki değişken de aynı değer sahip olur.

```
Arttırma - Azaltma Operatörü
sayı : 24 <---> baskaSayi : 24
```

Şekil 75: Arttırım operatörünün ön-ek olarak kullanıldığı durum.

İlişkisel Operatörler

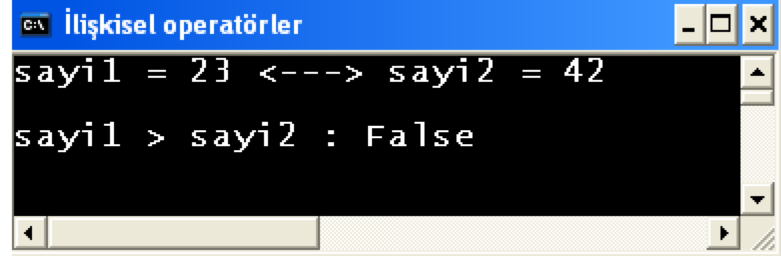
C#'da ilişkisel operatörler şunlardır:

- < Küçüktür
- > Büyüktür
- <= Küçük eşittir.
- >= Büyük eşittir.

İlişkisel operatörler iki değeri karşılaştırmada kullanılırlar ve ortaya mantıksal bir sonuç çıkarırlar; bu sonuç **bool** değişkeni ile temsil edilebilir ya da daha sonra görülecek kontrol ve döngü deyimlerinde yoğun olarak kullanılabilir.

```
...

int sayi1 = 23;
int sayi2 = 42;
bool buyukMu = sayi1 > sayi2;
Console.WriteLine("sayi1 = 23 <---> sayi2 = 42\n");
Console.WriteLine("sayi1 > sayi2 : {0}",buyukMu);

...
```

Yukarıdaki kod bloğunda **sayi1** değişkeninin değeri **sayi2** değişkeninin değerinden daha küçüktür ve dolayısıyla **sayi1 > sayi2** gibi bir önerme yanlış olur. Ekran çıktısı da **dogruMu** mantıksal sonuç değişkeninin değerinin **false** çıkmasıyla bunu destekler.



Şekil 76: İlişkisel operatörler

Koşul Operatörleri

C#'da mantıksal ifadelerin içerisinden birinin ya da her ikisinin birden doğruluğu kontrol edilmek istenirse koşul operatörlerinden faydalanılır. Koşul operatörleri;

- && (Ve operatörü)
- || (Veya operatörü)

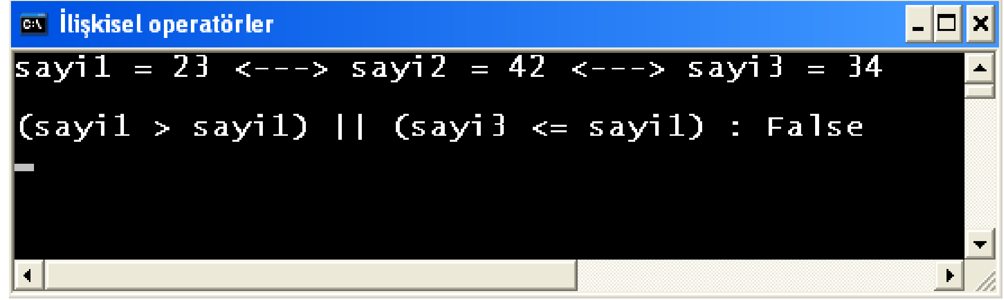
VE (&& - AND)	VEYA (|| - OR)
0 && 0 = 0	0 || 0 = 0
0 && 1 = 0	0 || 1 = 1
1 && 0 = 0	1 || 0 = 1
1 && 1 = 1	1 || 1 = 1

Tablo 12: Matematikte VE ile VEYA operatörlerinin kullanımı

Bu operatörlerin en yoğun şekilde kullanıldığı yer kontrol deyimleridir. (if kontrolü gibi). Ayrıca koşul operatörleri, mantıksal sonuç değişkeni ile de kullanılabilir; zaten ilişkisel, koşul ve eşitlik operatörlerinin geçtiği her yerde bool bir sonuç ortaya çıkar.

```
...

int sayi1 = 23;
int sayi2 = 42;
int sayi3 = 34;
bool dogruMu = (sayi1 > sayi2) || (sayi3 <= sayi1) ;
Console.WriteLine("sayi1 = 23 <---> sayi2 = 42 <---> sayi3 = 34\n");
Console.WriteLine("(sayi1 > sayi1) || (sayi3 <= sayi1) : {0}", dogruMu);

...
```

Burada geçen **dogruMu** adlı değişkenin değeri bir dizi mantıksal ifadenin sonucunda hesaplanmaktadır. **|| (veya)** operatörünün sağındaki ya da solundaki ifadelerden bir tanesinin mantıksal (bool) olarak **doğru** (true) sonuç üretmesi, **dogruMu** değişkenin **doğru** (true) değer alması için yeterlidir; çünkü **|| (veya)** operatörü, operandlarından bir tanesinin doğru dönmesi ile toplamda **doğru** bool sonucunu üretir. Ancak burada ayrı ayrı bakıldığında hem **sayi1** değişkeni **sayi2** değişkeninden büyük değil; hem de **sayi3** değişkeni **sayi1**'den küçük ya da eşit değil. **|| (veya)** operatörünün her iki tarafındaki ifade de **yanlış** (false) sonuç ürettiği için **dogruMu** değişkeni de **yanlış** (false) değer alır.



Şekil 77: Koşul operatörleri

Eşitlik Operatörleri

Bu başlık altında yer alan iki operatör şunlardır:

- == (Eşittir operatörü)
- != (Eşit değil operatörü)

Bu operatörler, sağındaki ve solundaki operandların aynı değerlere sahip olup olmadıklarını kontrol eder. Aynı değere sahip iki değer **==** operatörü ile karşılaştırılırsa sonuc bool olarak doğru (true) olacaktır. Aynı iki değer, **!=** operatörü ile karşılaştırıldıklarında ise sonuç yanlış (false) olur. C# dışındaki birçok dilde atama ile eşitlik kontrolü aynı operatörle yapılır. C#'da atamalar **=** operatörü ile yapılır. Bu iki operatörün ayrılması ile karıştırılmaları önlenmiş olup, çalışma zamanında oluşacak istek dışı atamaların ve istenildiğinde gerçekleşmeyen atamaların önüne geçilmiş olmaktadır.

```
...

int stok = 25;
int ihtiyac = 50;

Console.WriteLine("stok = {0}; ihtiyaç = {1}\n",stok,ihtiyac);

bool esitMi = (stok == ihtiyac);
Console.WriteLine("stok == ihtiyaç : {0}",esitMi);

bool esitDegilMi = (stok != ihtiyac);
Console.WriteLine("stok != ihtiyaç : {0}", esitDegilMi);

...
```

Yukarıdaki kod bloğunda her iki operatörün örnek kullanımına örnek yer almaktadır. Bu operatörlerin en sık kullanıldığı yer, koşul ve ilişkisel operatörlerde olduğu gibi kontrol ve döngü deyimleridir.

Operatörlerin Kullanıldığı Deyimlerde Parantezler

Yukarıdaki kodda dikkat edilmesi gereken bir nokta da eşitlik operatörlerinin kullanıldığı yerin parantez içerisine alınmasıdır. Bu zorunlu yapılması gereken bir şey değildir. Parantezler kullanılmasa da burada uygulama sağlıklı bir şekilde çalışır. Kullanılmalarının amacı ise atama operatörü ile eşitlik operatörlerinin öncelik sınırlarını kesin olarak çizmektir. Daha uzun deyimler kullanıldığında parantezlerle hem olası hataların önüne geçilmiş olur; hem de kodun okunurluğu arttırılmış olur. Bu parantezler sadece eşitlik operatörleri ile değil diğer bütün operatörler ile kullanılmalıdır.

BÖLÜM 3: KULLANICI TANIMLI TİPLER

Önceden tanımlı tipler belli bir programlama görevini yerine getirmemizde yeterli olmuyorsa, kendi tiplerimizi yazma yoluna gideriz. Kendi tiplerimizi yazmamıza olanak sağlayan iki tane değer tipi vardır. Bunlar numaralandırıcı (enumaration) ve yapı (struct) tipleridir. Nasıl ki .NET Framework'ü geliştirenler, uygulama geliştiriciler için int, bool, char gibi önceden tanımlı yapılar (struct'lar) ve henüz hiç görmediğimiz ancak framework'de sayısız miktarda var olan önceden tanımlı numaralandırıcılar (enum'lar) oluşturmuşlarsa; biz de kendi numaralandırıcı ve yapılarımızı yazabiliriz.

Numaralandırıcı (Enumaration)

Numaralandırıcı, .NET dünyasındaki beş ana tipten biridir. Uygulamada kullanılacak bir değişkenin alacağı değerler, belli bir küme içerisinden seçilmeye uygunsa ve bu değerler önceden tanımlı tipler içerisinde yer almıyorsa numaralandırıcılar bu noktada faydalı olur. Bir yıldaki mevsimlerin, programlama ortamında modellenmek istendiğini düşünelim: int tipinde bir değişken oluşturup 0, 1, 2 ve 3 değerleri sırasıyla ilkbahar, yaz, sonbahar, kış mevsimlerini temsil edecek şekilde bu değişkene atanır. Bu algoritma ile çalışılabilir; ancak bu çok da etkili bir yol değildir. Örneğin '0' değerinin gerçekten ilkbaharı temsil edeceği açık değildir. Ayrıca mevsimler için oluşturulacak değişkene 0, 1, 2 ve 3 dışında bir değer verilmesi engellenemez. C#, böyle bir senaryo için daha iyi bir çözüm önerir: **Numaralandırıcı**.

Bir numaralandırıcı tipi oluşturmak için **enum** anahtar kelimesini kullanmak gerekir. Tip tanımlaması içerisine, ihtiyaç duyulan değerler listesi yazılır; aynı zamanda varsayılan olarak bu listedeki değerlere 0'dan başlayıp 1'er artarak sayısal karşılıkları da otomatik olarak verilir. İhtiyaç durumunda sayısal karşılıkların istenildiği gibi düzenlenmesi esnekliği de uygulama geliştiriciye sunulmuş durumdadır. Bu senaryo koda dökülmek istendiğinde öncelikle numaralandırıcı tanımlaması nerede yapılacak sorusuna cevap verilmelidir. Numaralandırıcı tanımlamaları, bir isim alanı altında, bir sınıf veya bir yapı tanımlaması içerisinde olabilir. Bir sınıf veya yapı içerisinde tanımlanırsa tip içerisinde tip tanımlanmış olacak ki şu an o kadar derine girmeye gerek yoktur. Biz örneğimiz için enum tanımlamasını isim alanı düzeyinde gerçekleştirelim. Bir numaralandırıcı olarak oluşturulan tipin adı **Mevsim** olsun. Mevsim tipinde oluşturulacak bir değişkenin alabileceği değerler virgüllerle ayrılacak şekilde süslü parantez bloğu içerisinde listelenir.

```
using System;

namespace Degiskenler
{
    enum Mevsim
    {
        ilkbahar,
        yaz,
        sonbahar,
        kış
    }

    class Numaralandirici
    {
        public static void Main(string[] args)
        {
            Console.Title = "Numaralandırıcı tipi ile çalışmak";
        }
    }
}
```

Şekil 78: Mevsim adında bir numaralandırıcı oluşturmak

Mevsim tipinin oluşturulması ile birlikte listedeki her değer (arka tarafta) otomatik olarak **int** tipinde sayısal karşılıklarını alır. Bu karşılıklar varsayılan olarak 0'dan başlar ve birer birer artar. Dolayısıyla şu an her bir mevsim elemanının değeri;

ilkbahar = 0;
yaz = 1;
sonbahar = 2;
kış = 3;

olarak verilmiştir. Bu değerler, eğer bu örnekte olduğu gibi küçük sayısal karşılıkları ile kullanılacaklarsa bellekte int için 4 byte yer kaplamalarına gerek yoktur. Burada uygulama geliştiriciye bir enum tipinin sayısal karşılıklarının tutulacağı tam sayı tipini belirleme esnekliği tanınmıştır. Sayısal karşılıklar eğer int yerine sbyte tipinde tutulmak istenirse tanımlama şu şekilde yapılmalıdır:

```
enum Mevsim : sbyte
{
      ilkbahar,         // = 0 değerine sahiptir.
      yaz,              // = 1 değerine sahiptir.
      sonbahar,         // = 2 değerine sahiptir.
      kış               // = 3 değerine sahiptir.
}
```

Her bir mevsim değeri için yine aynı sayısal karşılıklar tutulur ancak bu durumda bellek yönetimi biraz daha düşünülmüş olur. Eğer ihtiyaç çerçevesinde numaralandırıcı listesinin sayısal karşılıkları düzenlenmek istenirse aşağıdaki gibi gerçekleştirilebilir:

```
enum Mevsim : sbyte
{
      ilkbahar = 1,
      yaz,
      sonbahar,
      kış
}
```

Listedeki değerlerden sadece **ilkbaharın** sayısal karşılığı değiştirilmiş gibi gözükse de diğer üyeler de bundan etkilenmişlerdir. Şöyle ki; değişiklik yapılan bir sayısal karşılığın ardından sıradaki üyelerin sayısal karşılıkları ardışık değerler alırlar. Yani ilkbaharın 1 değerini almasıyla yaz, sonbahar ve kış sırasıyla 2,3,4 değerlerini alır.

```
enum Mevsim : sbyte
{
      ilkbahar = 1,
      yaz,              //2
      sonbahar,         //3
      kış               //4
}
```

Sayısal karşılıklar konusu için eklenecek son şey, bu karşılıkların illa ki sıralı olmak zorunda olmamasıdır. Aşağıdaki tanımlama tamamen geçerlidir:

```
enum Mevsim : sbyte
{
      ilkbahar = 12,
      yaz = 9,
      sonbahar = 6,
      kış = 15
}
```

Bir numaralandırıcı oluşturma aşaması görüldüğüne göre artık bir enum sabitinin nasıl kullanılacağı incelenebilir. Mevsim artık, int, bool vb. yapıları gibi kullanılabilen bir veri

tipidir. Ayrıca Mevsim, enum sabiti olmasından dolayı değer tiplidir. Dolayısıyla Mevsim tipinden tanımanacak bir değişken artık kullanılabilirdir. Burada şu hususa dikkat edilmesi gereklidir: **Mevsim tipinde oluşturulacak bir değişkenin değeri sadece Mevsim tipi üzerinden elde edilebilir**. Bir numaralandırıcının oluşturulmasının temel sebebi de bu durumdur.

```
Mevsim lokalDegisken = Mevsim.sonbahar;
```

Tanımlama yapılırken atama operatörü yazılıp bir boşluk bırakıldığında eğer o ana kadar kodda bir hata yoksa oluşturulan **lokalDegisken** adlı değişkenin alabileceği değerler intelli-sense özelliği sayesinde elde edilebilir. Bunun sonucunda ekran görüntüsünde olduğu gibi enum sabitinin değerleri sıralanır. Bu sıralama, tipi oluştururken geliştiricinin belirttiği sırada değil, alfabetik sırada olur.

```
using System;

namespace Degiskenler
{
    enum Mevsim : sbyte
    {
        ilkbahar = 12,
        yaz = 9,
        sonbahar = 6,
        kış = 15
    }

    class Numaralandirici
    {
        public static void Main(string[] args)
        {
            Console.Title = "Numaralandırıcı tipi ile çalışmak";

            Mevsim lokalDegisken = |
        }
    }
}
```

Math
mbox
MemberAccessException
MethodAccessException
Mevsim
Microsoft
MidpointRounding
MissingFieldException
MissingMemberException
MissingMethodException

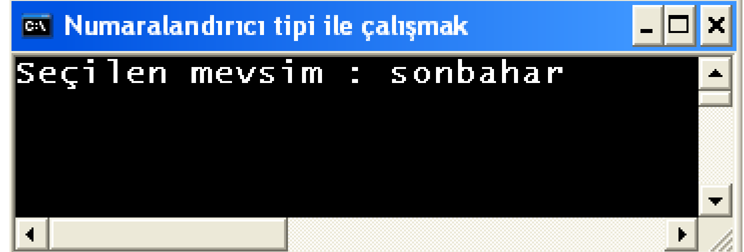
enum Degiskenler.Mevsim

Şekil 79: Bir enum tipine başlangıç değeri verilmesi

Numaralandırıcılar değer tipli olduğu için Mevsim tipinde oluşturulan değişken belleğin **stack** bölgesinde oluşturulur. Diğer değer tipleri için geçerli kuralların hepsi numaralandırıcılar için de geçerlidir. Örneğin Mevsim tipindeki değişken, bir metot içerisinde oluşturulduğunda lokal değişken olabilirken, bir sınıfın üyesi olarak oluşturulduğunda alan (field) adını alır ve bu ayrıma ait özellikleri taşır.

```
...
Mevsim lokalDegisken = Mevsim.sonbahar;
Console.WriteLine("Seçilen mevsim : {0}",lokalDegisken);

...
```

Yukarıdaki örnek ile ekrana **lokalDegisken** değeri yazdırıldığında değer, **Mevsim.sonbahar** değil, sadece **sonbahar** olarak elde edilir.



Şekil 80: Bir enum tipinde değişkenin değerinin elde edilmesi

Enum tipinde bir değişkenin sayısal karşılığının elde edilmesi mümkündür. Mevsim tipindeki değerin, sayısal karşılığın tutulduğu tipe dönüştürülmesi yeterlidir.

```
...

Mevsim lokalDegisken = Mevsim.sonbahar;
Console.WriteLine("Seçilen mevsimin sayısal karşılığı: {0}",
(sbyte)lokalDegisken);

...
```

Mevsim tipinin üyeleri için sayısal karşılıklarında yapılan son düzenlemenin ardından yukarıdaki kodun çıktısı aşağıdaki gibi olur.

```
Numaralandırıcı tipi ile çalışmak
Seçilen mevsimin sayısal karşılığı : 6
```

Şekil 81: Bir enum tipinde değişkenin sayısal karşılığının elde edilmesi

Yukarıda yapılan dönüştürmenin tam tersi de geçerlidir. Örneğin kullanıcıdan alınan sayısal değerin hangi mevsime karşılk geldiği bu yolla bulunabilir:

```
...

Console.WriteLine("ilkbahar = 12 ; yaz = 9 ; sonbahar = 6 ; kış = 15");
Console.Write("Favori mevsiminizi seçiniz : ");
int secilenSayi = int.Parse(Console.ReadLine());
Mevsim favoriMevsim = (Mevsim)secilenSayi;
Console.WriteLine("Favori mevsiminiz : {0}. Teşekkürler.",favoriMevsim);

...
```

```
Numaralandırıcı tipi ile çalışmak
ilkbahar = 12 ; yaz = 9 ; sonbahar = 6 ; kış = 15
Favori mevsiminizi seçiniz : _
```

Şekil 82: Bir enum tipinde sayısal karşılıklardan değerlerin elde edilmesi.

Kullanıcının yukarıdaki satırda verilen dört sayısal değerden birini girdiği varsayılmaktadır. '6' değerinin girildiği durumu ele alalım;

```
ilkbahar = 12 ; yaz = 9 ; sonbahar = 6 ; kış = 15
Favori mevsiminizi seçiniz : 6
Favori mevsiminiz : sonbahar. Teşekkürler.
```

Şekil 83: Bir enum tipinde sayısal karşılıklardan değerlerin elde edilmesi.

Görüldüğü gibi enum tipi ile yapılabilecekler oldukça zengin bir içeriğe sahiptir. Enum sabitlerinin kullanımına örnek olarak; herhangi bir ülkenin telefon alan kodları, posta kodları, haftanın günleri, hatta görsel uygulamalarda kullanılan ızgara (GridView,DataGridView) kontrollerinin başlıkları vb. verilebilir.

Yapı (Struct)

Yapılar, değer tipli davranışı benimserler. Bellekten düşmesi, tanımlandığı kapsama alanına bağlıdır. Bu şekilde bellekte nesneleri temsil etmek için kullanılabilir. Henüz sınıfları (class) tam anlamıyla bilmiyoruz; ancak bir yapı (struct), sınıfın taşıdığı neredeyse bütün özellikleri bünyesinde barındırır. Sınıflar daha kuvvetli, kullanışlı programlama objeleri olduğu için genelde yapıya göre tercih edilirler. Peki o zaman yapıları nerede tercih edebiliriz? **Yapılardan, farklı tipteki az sayıda değişkeni bir arada taşıyıp kullanmak için faydalanılabilir**. Ayrıca önceden tanımlı birçok yapı olduğunu (int, long, char, bool vb.) ve sık sık bunları kullandığımızı unutmayalım. Yapılar numaralandırıcılar gibi isim alanı düzeyinde, içsel tip olarak başka bir yapı ya da sınıfın içerisinde oluşturulabilirler. İsim alanı düzeyinde oluşturulan **Kitap** yapısı için aşağıdaki kodu inceleyelim.

```
using System;

namespace Degiskenler
{

    struct Kitap
    {
        public string kitapAdi;
        public int sayfaSayisi;
    }

    class Yapi
    {
        public static void Main(string[] args)
        {
            Console.Title = "Kendi yapı'mızı (struct) yazmak";

            Console.ReadLine();
        }
    }
}
```

Şekil 84: Kitap adında bir yapı oluşturmak.

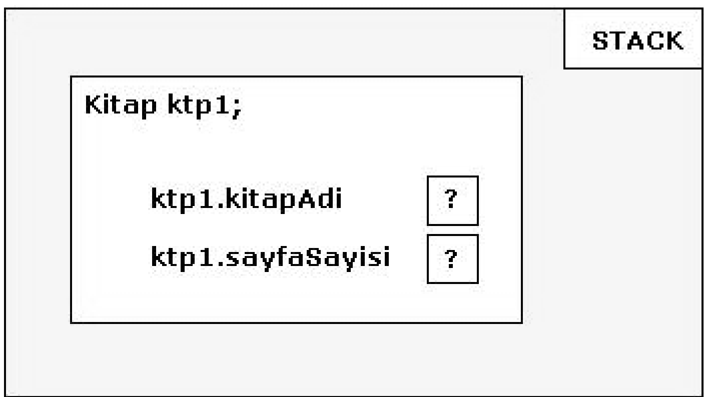
Bir tip başka bir tipin içerisinde tasarlandığında **dahili tip (inner type)** adını alır.

Bu kod Kitap adında, içerisinde iki tane alan (field) üyesi olan (kitapAdi ve sayfaSayisi) yeni bir tip oluşturmaktadır.

Bu yapıyı kullanmak için normal bir değer tipi gibi ondan değişken oluşturulur ve içerisindeki alanlara değerleri verilir.

```
Kitap ktp1;                   //Değişken tanımlanır.
```

Değişken ilk oluşturulduğunda belleğin stack bölgesinde Kitap tipinde bir nesne yerini alır. Ancak içerisindeki üyeler değer tiplerinin kararteristik özelliği gereği henüz değerlerini alamadıkları için kullanılamazlar.

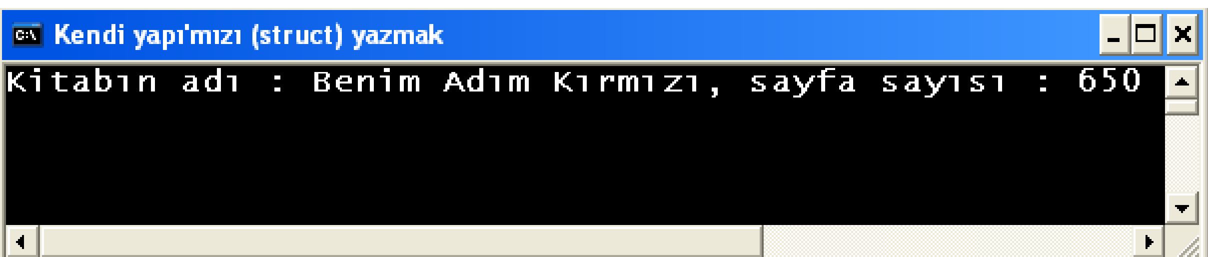


Şekil 85: Kitap tipinde oluşturulan değişken üzerinde henüz değerleri verilmemiş alanlar (fields) yer alır.

```
ktp1.kitapAdi = "Benim Adım Kırmızı";     //Alanlarına değerleri verilir.
ktp1.sayfaSayisi = 650;
```

Değer atamaları yapıldıktan sonra alanların bellekteki bölgelerine değerleri yerleştirilir ve böylece ktp1'in üyeleri kullanılabilir duruma gelir.

```
Console.WriteLine("Kitabın adı : {0}, sayfa sayısı : {1}", ktp1.kitapAdi,
ktp1.sayfaSayisi);
```



Şekil 86: Kitap adındaki yapıdan değişken tanımlanır ve değerleri verilip kullanılır.

Yapılar arasında yapılan atamalar, her zaman için bellekte farklı alanların açılmasına neden olur.

```
...
Kitap ktp2;
```

```
ktp2.kitapAdi = "Kar";
ktp2.sayfaSayisi = 350;
Console.WriteLine("Kitabın adı : {0}, sayfa sayısı : {1}", ktp2.kitapAdi,
ktp2.sayfaSayisi);

...
```

```
Kendi yapı'mızı (struct) yazmak
Kitabın adı : Benim Adım Kırmızı, sayfa sayısı : 650
Kitabın adı : Kar, sayfa sayısı : 350
```

Şekil 87: Kitap tipinde istenilen sayıda değişken oluşturulabilir.

Bir yapıdan, yeni bir yapı elde edilmeye çalışıldığında (özellikle atama yolu ile) referans tiplerinde meydana gelen aynı veri bölgesini adresleme durumu gerçekleşmez. Yeni oluşturulan Kitap değişkeninin değerleri için belleğin stack bölgesinde yeni bir kopya oluşturulur:

```
...

Kitap ktpKopya = ktp2;
Console.WriteLine("Kitabın adı: {0}, sayfa sayısı: {1}",
ktpKopya.kitapAdi, ktpKopya.sayfaSayisi);

...
```

```
Kendi yapı'mızı (struct) yazmak
Kitabın adı : Benim Adım Kırmızı, sayfa sayısı : 650
Kitabın adı : Kar, sayfa sayısı : 350
Kitabın adı : Kar, sayfa sayısı : 350
```

Şekil 88: Kitap tipli bir değişkenden başlangıç değerini alan Kitap tipli yeni değişken için bellekte aynı değerlerle yeni bir kopya oluşturulur.

Bir yapı, değer tipi özelliklerine aykırı olarak 'new' anahtar kelimesi ile oluşturulabilir. Ancak bu, yapının belleğin heap bölgesinde yer alacağı anlamına gelmez. Bu, yapı tanımlanırken üyelerine ilk değerlerinin verilebilmesi için kullanılır.

```
Kitap ktp = new Kitap();
```

Yukarıdaki tanımlama ile Kitap tipinde oluşturulan **ktp** değişkeninin taşıdığı **kitapAdi** ve **sayfaSayisi** değişkenlerinin başlangıç değerleri verilmiş olur. 'new' anahtar kelimesi kullanılmadan tanımlanan bir yapı için aşağıdaki kod derleme zamanı hatası üretecekken, new operatörü sayesinde artık **kitapAdi** için null (boş), **sayfaSayisi** için '0' başlangıç değerleri ile yapı tanımlanır.

```
Console.WriteLine("Kitabın adı : {0}, sayfa sayısı : {1}", ktp.kitapAdi,
ktp.sayfaSayisi);
```

```
Kendi yapı'mızı (struct) yazmak
Kitabın adı : Benim Adım Kırmızı, sayfa sayı
Kitabın adı : Kar, sayfa sayısı : 350
Kitabın adı : Kar, sayfa sayısı : 350
Kitabın adı : , sayfa sayısı : 0
```

Şekil 89: 'new' anahtar kelimesi ile tanımlanan yapı

Yapıların ne zaman kullanılacağı ile ilişkili olarak Microsot'un geliştiricelere bir önerisi vardır. Buna göre, eğer bir yapı içerisinde kullanılan alanların toplam boyutu **16 byte**'ın üzerindeyse sınıfların tercih edilmesi tavsiye edilir. Bu sebebi kavrayabilmek için, çalışma zamanında bu tarz tiplerden **n** adedine ihtiyaç duyulabileceğini düşünmekte fayda vardır.

BÖLÜM 4: KONTROL YAPILARI, DÖNGÜLER, İSTİSNALAR

Kontrol Deyimleri

If Kontrolü

Mantıksal (boolean) bir ifadenin sonucuna bağlı olarak iki farklı kod bloğundan sadece birini çalıştırmayı seçmek için **if** deyiminden faydalanılabilir. Uygulamayı belli koşullara bağlı olarak dallandırmak için kullanılan **if** deyiminde kontrol her zaman ilişkisel, koşul ve eşitlik operatörleri ile sağlanır. Genel söz dizimi aşağıdaki gibidir :

```
if(mantıksal ifade)
{
     //Yapılacak iş 1
}
else
{
     //Yapılacak iş 2
}
```

Burada eğer mantıksal ifade **true** değer döndürürse "Yapılacak iş 1" in bulunduğu ilk kod bloğu çalışır; aksine **false** değer döndürürse "Yapılacak iş 2" nin bulunduğu kod bloğu çalışır.

Ayrıca **else** bloğu her seferinde kullanılmak zorunda değildir; bu durumda eğer mantıksal ifadenin döndürdüğü değer **false** olursa hiçbir şey yapılmaz ve kod aşağıya doğru akmaya devam eder.

```
...

int not1 = 70;
int not2 = 80;
int ortNot = (not1 + not2) / 2;

if (ortNot < 50)
{
     Console.WriteLine("Öğrenci bu dersten kaldı...");
}
else
}
     Console.WriteLine("Öğrenci bu dersten geçti...");
}

...
```

not1 ve **not2** değişkenlerinin ortalaması şeklinde hesaplanan **ortNot** değişkeninin değerine bağlı olarak uygulamada iki farklı kod bloğundan biri çalışır. Hangi kod bloğunun çalışacağına karar verilen yer if(...) deyiminin içerisindeki mantıksal ifadedir. ortNot, 75 olarak hesaplanmış olarak if kontrolüne gelir; burada (75 < 50) mantıksal çözümlemesi yapılır ve sonuç 'yanlış (false)' olarak üretilir. Bu noktada if, kontrolün ilk bloktan mı yoksa ikinci bloktan mı devam edeceğine karar verir ve sonuç yanlış olduğu için "else" bloğuna yönlendirir.

if ve else anahtar kelimelerinin ardından çalışacak olan kod blokları eğer tek satırdan oluşuyorsa süslü parantezler içerisine almaya gerek yoktur; ancak birden fazla satır olduğu anda parantezler koyulmalıdır, yoksa if'den sonra koyulmayan parantezler için derleme zamanı hatası oluşurken (eğer arkasından

if geliyorsa), else'den sonra koyulmayan süslü parantezlerin bedeli ilk satırdan sonraki kodların if/else kontrolünden bağımsız olarak her halükarda çalışması olur. Bu gibi hatalara yer vermemek için en doğru olanı bir satır da olsa if/else bloklarını parantezler içerisinde yazmaktır. Benzer durum, ileriki konuda görülecek olan for döngüsü için de geçerlidir.

Yukarıdaki örneğin çıktısı şöyledir:

Şekil 90: Basit bir if kontrolü

if bloğu kullanımına örnek olarak aşağıdaki kodları inceleyelim:

```
...
if ((7 % 2) == 0)
{
      Console.WriteLine("7, tek sayıdır...");
}
else
{
      Console.WriteLine("7, çift sayıdır...");
}

...
```

7'nin 2 ile bölümünden kalan 1'dir; dolayısıyla if deyimi **false** değer döndürür ve kontrol else bloğundan akışına devam eder. Bu aşamalar debug edilerek çok daha sağlıklı bir şekilde görülebilir; bu, konunun anlaşılmasını da kolaylaştırır.

Operatör öncelikleri

if bloğu içerisinde herhangi bir mantıksal ifade olması yeterlidir. Bu birden fazla ifadenin birleşimi de olabilir, çok basit bir kontrol de olabilir. Örnekte if bloğu içerisinde 7'nin 2 ile bölümünden kalan sayının 0'a eşit olup olmadığı kontrol edilmektedir. (7 % 2) ifadesi parantez içerisine alınmasa da kod sağlıklı bir şekilde çalışır; çünkü aritmetik operatörlerin eşitlik operatörüne karşı önceliği vardır. Ancak tavsiye edilen şudur ki; **hiçbir zaman işi önceliklere bırakma, kendi önceliklerini parantezlerle belirle**. Örnekte de bu tavsiyeye uyulmuştur.

Boolean ifadelerin sayısal değerlere (0 – 1 gibi) dönüştürülmesi ya da sayısal değerlerden elde edilmesine C# destek vermemektedir.

Hiçbir zaman işi önceliklere bırakma, kendi önceliklerini parantezlerle belirle.

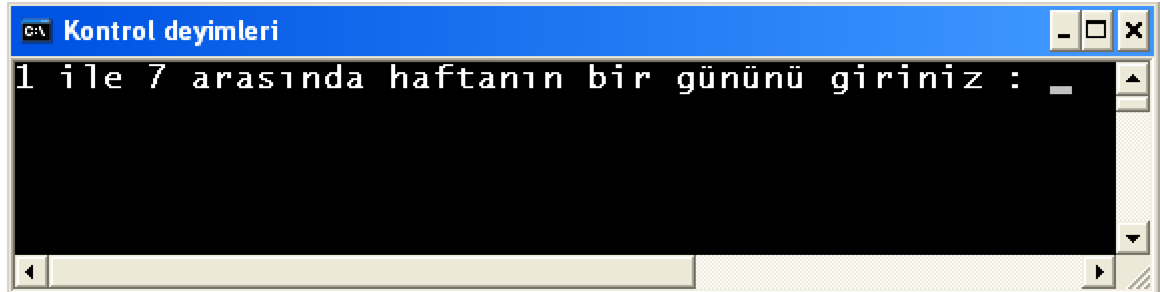
if – else if İfadeleri

if deyimleri iç içe kullanıldığında, mantıksal deyimler birbirlerine zincirlenir ve bir tanesi **true** değer döndürünceye kadar birbiri arkasına çalıştırılır. Örneğin haftanın günlerini ele alalım. Kullanıcıdan alınacak haftanın gün sayısı ile günün ne olduğunu ekrana yazdıracağımızı düşünelim:

```
...

string gunAdi;
```

```
Console.Write("1 ile 7 arasında haftanın bir gününü giriniz : ");
int gun = int.Parse(Console.ReadLine());
if (gun == 1)
      gunAdi = "Pazartesi";
else if (gun == 2)
      gunAdi = "Salı";
else if (gun == 3)
      gunAdi = "Çarşamba";
else if (gun == 4)
      gunAdi = "Perşembe";
else if (gun == 5)
      gunAdi = "Cuma";
else if (gun == 6)
      gunAdi = "Cumartesi";
else if (gun == 7)
      gunAdi = "Pazar";
else
      gunAdi = "Doğru gün sayısı girilmedi";

Console.WriteLine(gunAdi);

...
```

Burada birinci if'den itibaren kullanıcının varsayılan olarak hatasız girdiği sayı ile karşılaştırma yapılarak ilerlenir. Bu kadar if, else if kontrolleri ile birbirine bağlı bir şekilde çalışmaktadır. Bu yüzden herhangi bir if ya da else/if'den birinde doğru değer tutturulmuşsa kontrol bu else/if bloğunun sonuna gider, diğerlerini boşuna kontrol etmez; çünkü oradaki dizayn, bir tane doğrunun olması üzerine kuruludur.



Şekil 91: else/if yapısı

```
Kontrol deyimleri
1 ile 7 arasında haftanın bir gününü giriniz : 5
Cuma
```

Şekil 92: else/ if blokları doğru bir şekilde çalışıyor.

if bloklarında koşul operatörleri de (&&, ||) kullanılabilir. Özellikler birden fazla koşulu bir if içerisinde test etmek istediğimizde bu yola başvurabiliriz. Aşağıdaki yüzde kontrolü buna iyi bir örnektir.

```
int yuzde = 90;

if ((yuzde < 0) && (yuzde > 100))
{
      Console.WriteLine("Yüzde {0}",yuzde);
}
```

```
else
{
      Console.WriteLine("Geçersiz yüzde...");
}
```

Switch Kontrolü

if – else ve/veya **if – else if** deyimleri ile yapılabilen kontrollerin, bazı durumlarda **switch** deyimi ile yapılması tercih edilebilir. Özellikle if – else if deyimlerinin sayısı arttıkça switch deyimi tercih edilirse, hem daha etkili, hem de daha okunaklı kod yazılmış olunur. Dikkat edilmesi gereken nokta, switch bloklarının sadece sabit değerler üzerinden koşul denetimi yapıyor olmasıdır. Bununla birlikte switch bloklarında karşılaştırma ve eşitlik operatörlerinden faydalanılamaz. Genel söz dizimi aşağıdaki gibidir:

```
...

int kararDegiskeni = 1;
switch (kararDegiskeni)
{
    case 1:
    Console.WriteLine("Karar 1");
      break;
    case 2:
    Console.WriteLine("Karar 2");
      break;
    default:
    Console.WriteLine("Varsayılan Karar");
      break;
}

...
```

- Burada kodun hareketini kontrol eden **kararDegiskenidir**; onun değeri switch bloğu içerisindeki **case** anahtar kelimesinin önündeki değererle karşılaştırılır.
- Eğer uyan bir tanesine rastlarsa bir alt satıra geçer ve kodu çalıştırır; uymazsa otomatik olarak bir sonraki **case** anahtar kelimesini kontrol etmeye geçer.
- Eşleşen hiçbir **case** bulunamazsa default anahtar kelimesinin altındaki kodlar çalışır.
- ‘case’ lerin altında herhangi bir süslü paranteze ihtiyaç duyulmamaktadır; çünkü eşleşen bir **case** in kodları aşağıya doğru akar ve **break** anahtar kelimesini görene kadar çalışır. **break**, kontrolü switch deyiminin sonuna götürür. Böylece şartlardan sadece bir tanesinin çalışması garanti altına alınır.
- Switch bloğunda **default** bulunmak zorunda değildir; eğer varsa **default** anahtar kelimesinin de sonuna **break** koyulmalıdır, yoksa en son case deyiminde **break** olmalıdır. Zaten unutulursa derleyici, derleme zamanı hatası vererek geliştiriciyi uyaracaktır.

Aşağıdaki örnekte kullanıcıdan sayısal bir değer alınıp, bu değere bağlı olarak kullanıcıya basit cevap veren bir uygulama yer almaktadır. Kullanıcıdan istenilen değer dışında bir sayısal değer girişi yapılması durumunda if kontrolündeki else bloğunun benzeri işlevselliğe sahip default bloğu çalışır.

```
...

Console.WriteLine("1 [C#], 2 [VB]");
Console.Write("Lütfen dil seçiminizi yapınız: ");
string dilSecimi = Console.ReadLine();
int n = int.Parse(dilSecimi);
switch (n)
{
   case 1:
      Console.WriteLine("İyi tercih. C#, iyi bir dil...");
```

```
      break;
   case 2:
      Console.WriteLine("VB .NET: İşin biraz zor ama imkansız değil...");
      break;
   default:
      Console.WriteLine("Peki... Sana iyi şanslar...");
      break;
}

...
```

switch deyimi kullanılarak kontrol yapılabilecek değişken (Yukarıdaki örnekte **dilSecimi** değişkeni) tipleri şunlar olabilir: Bütün tam sayı tipler, metinsel tipler, bool tipi ve her hangi bir numaralandırıcı (enum) tipi.

Döngü Deyimleri

While Döngüsü

Mantıksal bir ifade doğru olduğu sürece belli bir kod bloğu, tekrar tekrar çalıştırılmak istendiğinde başvurulacak döngü deyimlerinden biridir.

Döngülerin çalışma mekanizmasını daha kolay kavrayabilmek için aşağıdaki kod parçasını içeren uygulamanın çalışma zamanında debug edilip adım adım ilerlenilmesi önerilir.

```
…

int a = 1;
while (a <= 6)
{
      Console.WriteLine("a'nın şu anki değeri : {0}",a);
      a++;
}

…
```

Önce a değişkeni tanımlanır ve 1 değerini alır. Sonra while deyiminin içerisi kontrol edilir, bu deyimin içi doğru olduğu sürece while bloğu (süslü parantezler arası) çalıştırılır; dolayısıyla deyim içerisinde kontrol edilen değişkenin değeri, her dönülüşünde deyim içerisindeki şartı bozmaya yönelik olarak verilmelidir. Yani deyim içerisinde kontrol edilen **a<=6** iken döngü bloğu içerisinde **a**'nın değeri her seferinde azaltılırsa bu döngüden çıkılmaz, çıkılsa da sağlıklı olarak uygulamaya geri dönülemez demektir. Bu yüzden **a** değişkeninin değeri her seferinde 1 arttırılır. İlk şart sağlanıp ekrana gerekli bilgilerin yazılmasının ardından önce değişkenin değeri arttırılır. Ardından döngü deyiminin kontrolü yeniden yapılır ve '2' değeri için de doğru olduğundan yine bloğa girilip kod çalıştırılır. Döngü değişkeninin değeri tekrar arttırır ve bu döngü **a**, 6 değerini alana kadar devam eder. Hatta 6 değeri de şartı sağladığı için döngü içerisindeki kod bloğu bir kere daha çalışır; ancak sonraki artışta a değişkeni 7 değerini aldığı için koşul **false** değer döndürür ve döngü sonlanır. Kontrol, döngü bloğunun sonundan devam eder. Çıktı aşağıdaki gibi olur:

```
Döngü Deyimleri
a'nın şu anki değeri : 1
a'nın şu anki değeri : 2
a'nın şu anki değeri : 3
a'nın şu anki değeri : 4
a'nın şu anki değeri : 5
a'nın şu anki değeri : 6
```

Şekil 93: While döngüsü örneği

Burada dikkat edilmesi gereken nokta; while döngüsünde 3 ayrı çok önemli parametrenin tamamen birbirinden ayrı yerlerde işleniyor olmalarıdır.

- Döngü değişkeninin tanımlandığı ve ilk değerinin verildiği yer, döngü bloğunun dışındadır.
- Döngünün ne kadar süre ile devam edip etmeyeceğine karar verilen yer, döngü deyiminin içerisindedir.
- Döngü değişkeninin arttırım değeri ise döngü bloğunun içerisinde bulunmaktadır.

Özellikle arttırım ifadesinin unutulmadan ve doğru olarak verilmesine dikkat edilmelidir. Bu koordinasyon iyi kurulduğu sürece while döngüsünden ihtiyaç duyulan yerlerde maksimum fayda sağlanır.

While döngüsü için yazılmış aşağıdaki örnekte kullanıcıdan istenen iki kelimeden biri girilmediği sürece döngü içerisinde kalınmaktadır. Kullanıcıdan alınan değerin döngüyü sonlandıracak değer olabileceği; ancak büyük harfli olması ihtimali göz önünde bulundurularak ToLower() metodu ile her durumda küçük harfli temsili elde edilir. While döngüsünde döngü deyimi içerisindeki şart, kod bloğu çalışmadan önce kontrol edildiği için değişkene bir de döngünün devamını sağlayacak bir başangıç değeri verilir.

```
...

string kullaniciOK = "no";

while (kullaniciOK.ToLower() != "yes")
{
      Console.Write("Tamam mı? [yes] [no]: ");
      kullaniciOK = Console.ReadLine();
      Console.WriteLine("*****While döngüsündesin*****");
}
Console.WriteLine("-----While döngüsünden çıktın-----");

...
```

Burada döngünün devamını kontrol eden şey ekrandan “yes” dışında herhangi bir şey girilmesidir. Çünkü döngü deyiminin içerisinde **‘eşit olmadığı sürece’** kontrolü yapılmaktadır.

```
kullaniciOK.ToLower() != "yes"
```

Do-While Döngüsü

Do-While döngüsü tamamen while döngüsü gibi çalışır; aralarındaki tek fark while döngüsünde döngü deyimi içerisindeki kontrol, döngüye girilmeden önce yapılırken; do-while döngüsünde bu kontrol ilk döngü bloğundan çıkıldıktan sonra yapılmaya başlanır.

Yani döngü bloğu içersinde yer alan kod, kayıtsız-şartsız ilk sefer için çalıştırılır. Öyleyse while döngüsünde verilen 2. örnek aşağıdaki gibi düzenlenebilir:

```
...

string kullaniciOK = "";

do
{
      Console.WriteLine("*****While döngüsündesin*****");
      Console.Write("Tamam mı? [yes] [no]: ");
      kullaniciOK = Console.ReadLine();

} while (kullaniciOK.ToLower() != "yes");

Console.WriteLine("-----While döngüsünden çıktın-----");

...
```

Öncelikle bir do-while döngüsünün yapısını inceleyelim. Döngü bloğunun içerisi, ilk sefer için kontrol edilmeden çalışacağından while anahtar kelimesi ve döngü deyimi, kod bloğunun altında yer alır ve bittiğine dair bir de noktalı virgül bekler. Kod bloğunun başına da **do** anahtar kelimesi gelir. Kod incelenecek olursa; en önemli değişiklik, blok ilk çalıştırılmada kontrole maruz kalmadığı için **kullaniciOK** değişkenine herhangi bir değer verilmemesidir. İkinci değişiklik ise, bütün şartlar altında döngü bloğundaki kod çalıştırıldığı için (döngü deyimi içerisindeki şart sağlanmasa da)

```
Console.WriteLine("*****While döngüsündesin*****");
```

yazısı döngü bloğunun başına alınmasıdır.

Do-while döngüsü kullanımına örnek olarak kullanıcı adı ve şifreyi kullanıcıdan minimum bir defa isteyen bir algoritma düşünülebilir. Aşağıdaki kodu inceleyelim:

```
...

string kullaniciAdi = "";
string sifre = "";
do
{
      Console.Write("Kullanıcı adı giriniz : ");
      kullaniciAdi = Console.ReadLine();
      Console.Write("Şifre giriniz : ");
      sifre = Console.ReadLine();

} while ((kullaniciAdi != "emrah") || (sifre != "1234"));
Console.WriteLine("Siteye login oldunuz...");

...
```

do bloğu herhangi bir kontrol yapılmadan ilk sefer için çalışır ve istenildiği gibi kullanıcıdan kullaniciAdi ve sifre bilgileri alınıp ikinci sefer kontrol ettirilebilir. Kullanıcı adı ya da şifreden bir tanesinin bile yanlış olması durumunda login olmayı engellemek için kontrol

```
while ((kullaniciAdi != "emrah") || (sifre != "1234"));
```

şeklinde yapılmıştır. Bu deyim **eğer kullanıcı adı "emrah" değilse ya da sifre="1234" değilse döngüye devam edilsin** olarak yorumlanabilir.

For Döngüsü

For döngüsü, while döngüsü ile benzer fonksiyonelliğe sahiptir. Döngü değişkeninin tanımlanması, değişkenin döngü içerisinde güncellenmesi ve döngü koşul deyiminin bildirilmesi for'da tek bir yerde yapılır. Genel söz dizimi aşağıda yer almaktadır:

```
for (başlangıç değeri; mantıksal ifade ; güncelleme kontrol değişkeni)
{
      //kod bloğu;
}
```

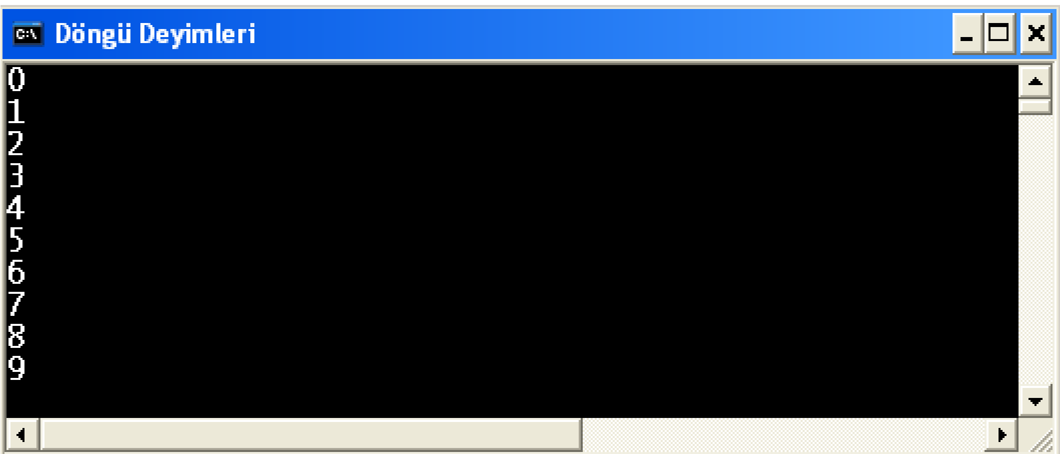
For döngüsü kullanıldığında bu 3 işlemden herhangi birini ihmal etmek neredeyse imkansız olur. Aşağıdaki örneği inceleyelim:

```
for (int i = 0; i != 10; i++)
{
      Console.WriteLine(i);
}
```

1. Baştaki i değişkeni, sadece döngünün başında bir defa oluşturulur.
2. Eğer ortadaki mantıksal (boolean) ifade doğru ise döngü bloğu çalıştırılır.
3. Sondaki kontrol değişkeni güncellenir ve mantıksal ifade yeniden hesaplanır.
4. Eğer koşul hala doğruysa döngü bloğu yeniden çalıştırılır.
5. Kontrol değişkeni güncellenir ve mantıksal ifade yeniden hesaplanır.
6. Ortadaki koşul (mantıksal ifade) **false** dönene kadar bu süreç devam eder.

Burada dikkat edilmesi gereken bazı noktalar vardır:

- Değişken oluşturma ve başlanıç değeri verme sadece ilk döngüde gerçekleşir.
- Döngü bloğu her zaman konrol değişkeninin (i) güncellenmesinden önce çalışır.
- Güncelleme her zaman mantıksal koşul ifadesinden önce çalışır.



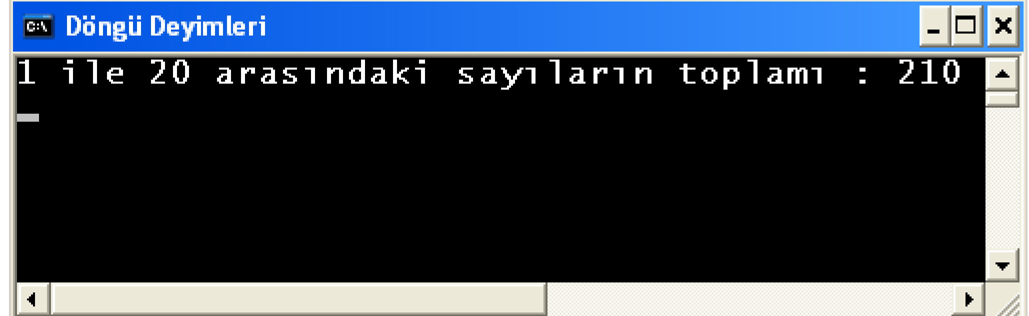
Şekil 94: For döngüsü örneği

For döngüsü kullanarak 1 ile 20 arasındaki bütün tam sayıların toplamını bulan kod aşağıda yer almaktadır:

```
...

int toplam = 0;
for (int i = 0; i <= 20; i++)
{
      toplam += i;
}
Console.WriteLine("1 ile 20 arasındaki değerlerin toplamı : {0}",toplam);
```

```
...
```

Bu kod çalıştığında döngü dışında tanımlanan ve başlangıç değeri '0' olan **toplam** değişkeninin üzerine her döngü turunda i değişkeninin güncel değeri eklenerek 1'den 20'ye kadar olan sayıların kümülatif toplamı elde edilir.



Şekil 95: For döngüsü kullanarak 1 ile 20 arasındaki sayıların toplamı

For döngüsünde aralarına noktalı virgül koyularak belirtilen üç bileşenden herhangi biri,ikisi ya da üçü birden belirtilmeyebilir veya uygun başka bir yerde belirtilebilir. İhtiyaç varsa çoklu başlangıç değeri ve güncelleme parçası kullanılabilir. Ayrıca döngü içerisinde oluşturulan değişkenlerle ilgili farkında olunması gereken bazı önemli noktalar vardır:

- Eğer mantıksal ifade yazılmazsa, varsayılan olarak dönecek değer **true** olacaktır. Dolayısıyla aşağıdaki for döngüsü sonsuza kadar çalışır:

```
...
for (int i = 0; ; i++)
{
   Console.WriteLine("Biri beni durdursun...");
}
...
```

Yukarıdaki kod test edilmiştir, gerçekten sonsuza kadar çalışıyor !

Bir diğer sonsuz döngü çeşidi de aşağıda yer almaktadır:

```
for( ; ; )
{

}
```

Bu tarz sonsuz döngülere çok sık ihtiyaç duyulmamakla birlikte özellikle giriş-çıkış (I/O) işlemlerinde, şifreleme algoritmalarında ve örneğin kullanıcı doğru değer girene kadar devam etmesi gereken durumlarda vb. kullanılabilir.

- Eğer değişken oluşturma ve başlangıç değerini verme ile değişkeni güncelleme parçaları for deyiminin içerisine yazılmazsa; ancak doğru satırlara yerleştirilirse ortaya çıkan manzara 'while' döngüsüne benzemektedir:

```
...
int i = 1;

for ( ; i != 11 ;)
{
   Console.WriteLine(i);
   i++;
```

```
    }

    ...
```

- For döngüsü içerisinde birden fazla değişken tanımlanıp yönetilebilir. Çoklu değişken tanımlama, başlangıç değerlerini verme ve bu değişkenleri güncelleme ifadeleri (ortadaki mantıksal ifade her zaman tek olmalıdır) noktalı virgülle ayrılan parça içerisinde virgülle ayrılır. Aşağıdaki örnek bunu modellemektedir:

```
    ...

    for (int i = 0, j = 20 ; i != j; i += 2, j -= 2)
    {
         Console.WriteLine(i + " <---> " + j);
    }

    ...
```

Döngü Değişkenleri

For döngüsü kullanırken oluşturulan değişkenler, sadece döngü bloğu içerisinde geçerli değişkenlerdir. Yani döngü deyimi içerisindeki mantıksal (boolean) ifadenin **false** döndürmesinin ardından program kontrolü döngü sonuna gelir ve değişken bellekten düşer. Dolayısıyla döngü sonunda değişken, kullanılmaya çalışılırsa derleyici bizi uyarır:

```
...

for (int i = 0; i < 10; i++)
{
     // Hesaplamalar...

}// i değişkeni burada bellekten düşer.

Console.WriteLine(i); //Derleme zamanı hatası

...
```

Yukarıdaki özelliğe bakarak şu ifadenin rahatlıkla söylenebilmesi gereklidir: Ard arda gelen döngüler içerisinde aynı isimli değişken adları kullanılabilir; çünkü her değişken, farklı kapsama alanlarına sahiptir:

```
...

for (int i = 0; i <= 10; i++)
{
     Console.WriteLine(i);
}

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

for (int i = 10; i >= 0; i-- )
{
     Console.WriteLine(i);
}

...
```

Son olarak for döngüsünde önceliklerin daha iyi anlaşılması adına aşağıdaki kodu analiz etmekte fayda vardır: (Ancak döngülerin bu şekilde kullanılması tavsiye edilmez)

```
…

int i = 0;
Console.WriteLine("*****Döngü başladı\n");
```

```
for(Console.WriteLine("Ben sadece ilk sefer çalışırım"); i < 6;
Console.WriteLine("Ben döngü deyimi doğru olduğu sürece çalışırım"))
{
     Console.WriteLine(i);
     i++;
}

Console.WriteLine("\n*****Döngü sona erdi...");

...
```

Bu döngünün çıktısı aşağıdaki gibi olacaktır :

```
Döngü Deyimleri
*****Döngü başladı

Ben sadece ilk sefer çalışırım
0
Ben döngü deyimi doğru olduğu sürece çalışırım
1
Ben döngü deyimi doğru olduğu sürece çalışırım
2
Ben döngü deyimi doğru olduğu sürece çalışırım
3
Ben döngü deyimi doğru olduğu sürece çalışırım
4
Ben döngü deyimi doğru olduğu sürece çalışırım
5
Ben döngü deyimi doğru olduğu sürece çalışırım

*****Döngü sona erdi...
```

Şekil 96: For döngüsünde döngü deyimi parçalarının çalışma sırası.

Atlama (Jump) Deyimleri

Atlama deyimleri (jump statements) kullanılarak, program kontrolü hemen transfer edilerek akışta dallanma(lar) sağlanabilir. Atlama deyimleri şunlardır:

- break
- continue
- goto
- return
- throw

Bu sıçrama deyimleri arasından **return**, ileriki konularda görülecek olan metotlarda kullanılır ve bir metotun çalışmasını sonlandırır; kontrolü metot bloğunun sonuna götürür.
throw ise, program akışı sırasında oluşan alışılmışın dışında bir durumu (çalışma zamanı hatası -exception-) işaret etmek için kullanılır. **throw** anahtar kelimesinin kullanımına bir sonraki konuda değinilecek.

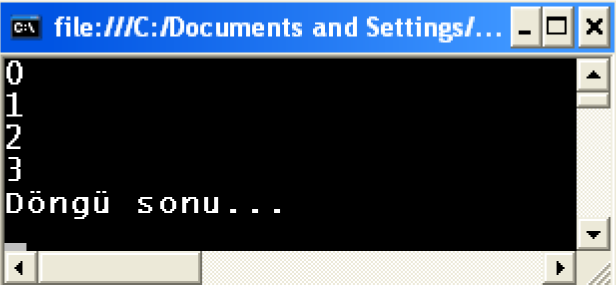
break Anahtar Kelimesi

break anahtar kelimesine daha önce switch kontrolünde değinilmişti. Orada yaptığı iş, kontrolü switch bloğunun sonuna transfer etmek ve akışın oradan devam etmesini

sağlamaktır. Bu anahtar kelime, bir döngü içerisinde kullanıldığında, yine switch' deki kullanımına benzer bir davranış gösterir ve kontrolü bir daha dönmemek üzere döngünün sonuna götürür. Bir başka deyişle döngü içerisinden bir koşul nedeniyle çıkılmak istendiğinde kullanılabilir. Aşağıdaki örnek bunu modellemektedir:

```
...
for (int i = 0; i < 10; i++)
{
      if (i == 4)
      {
         break;
      }
      Console.WriteLine(i);
} //break çalışınca kontrolü buraya transfer eder.
Console.WriteLine("Döngü sonu...");
...
```

Döngü bloğu 0,1,2 ve 3 için sağlıklı bir şekilde çalışır. Her dört değer için de yapılan if kontrolleri sonucunda **false** değer döner ve **break** komutunu çalıştırmadan geçerler. Döngü değişkeni (i) 4 olunca if kontrolü **true** döneceği için **break** deyimi çalışır ve kontrol, for döngü bloğunun sonuna geçer. Dolayısıyla 4 için değeri ekrana yazdırılamaz; sonraki sayılar için ise döngü çalışmaz.



Şekil 97: 'break' anahtar kelimesinin çalışma mekanizması

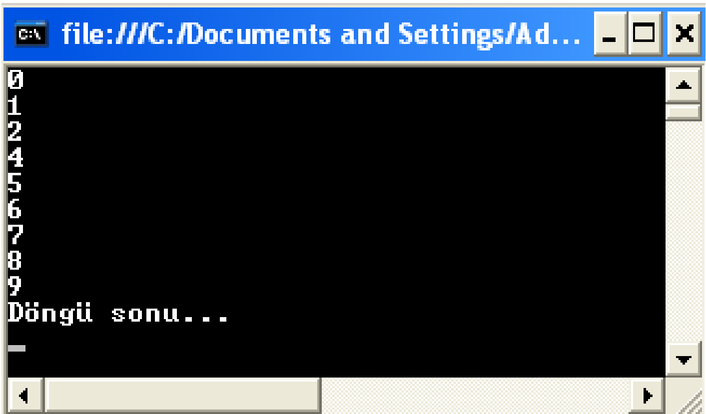
Eğer iç içe döngü yapısı kullanılıyorsa; break, kontrolü kendisini çevreleyen en yakın döngü bloğunun dışına transfer eder. Yani kontrol dışardaki döngüden çalışmaya devam eder.

continue Anahtar Kelimesi

continue anahtar kelimesi kontrolü kendisini çevreleyen en yakın döngünün başına transfer eder. Görüldüğü noktadan sonrasını o tur için çalıştırmaz, yani döngüyü bir turluğuna erteler; ancak döngü bir sonraki iterasyondan normal çalışmasına devam eder. **break** için yazılan kodu şimdi de **continue** için test edelim:

```
...
for (int i = 0; i < 10; i++)
{
      if (i == 3)
      {
            continue; //aşağıdaki kod, 3 değeri için çalışmaz; ancak
                      //4'den itibaren normal çalışmasına döner.
      }
      Console.WriteLine(i);
}
```

```
Console.WriteLine("Döngü sonu...");
...
```



Şekil 98: 'continue' anahtar kelimesinin çalışma mekanizması

Yukarıdaki şekilden görüldüğü gibi döngü sadece 3 değeri için çalışmamış; ancak daha sonra normal çalışmasına devam etmiştir.

break ve **continue** deyimleri sadece for döngüsünde değil; while, do-while ve diziler konusunda incelenecek olan foreach döngülerinde de kullanılabilir.

goto Anahtar Kelimesi

Bu anahtar kelime, kontrolü etiketlenmiş bir noktaya transfer eder.

```
...

Console.Write("Bir sayı giriniz : ");
int a = int.Parse(Console.ReadLine());
if (a % 2 == 0)
{
   goto cift;
}
Console.WriteLine("Girilen sayı tek");
goto Son;
cift:
Console.WriteLine("Girilen sayı çift");
Son:

...
```

Yukarıdaki kod, doğru bir şekilde kullanıcıdan alınan sayının tek mi çift mi olduğu kontrolünü yapar; ancak buradaki kullanımı ve diğer kullanılabileceği yerler düşünüldüğünde oldukça karışık bir düzen oluşturduğu ve kodu anlaşılması güç hale getirdiğini söylemek yanlış olmaz. O yüzden **goto** kullanımı için sadece şu iki senaryoyu tercih etmek gerekir:

1. switch-case kontrolünde, akışı belli bir case'e ya da default'a yönlendirmek söz konusu olduğunda.
2. İç içe geçmiş for döngülerinde **break** ya da **continue** ile sadece bir kademe dışarı çıkılabilirken tamamen dışarı çıkılmak istendiğinde kontrolü en dıştaki döngünün sonuna yönlendirmek söz konusu olduğunda.

Bu iki senaryo için **goto** kullanımı anlamlı iken bunlar dışında kullanmak spagetti kod (Anlaşılması zor, iç içe geçmiş kodları tasvir etmek için kullanılır) oluşturur.

```
...

Console.WriteLine("Kahve boyutu: 1=Küçük 2=Orta 3=Büyük");
Console.Write("Lütfen seçiminizi yapınız : ");
string secim = Console.ReadLine();
int n = int.Parse(secim);
int fiyat = 0;
switch (n)
{
      case 1:
        fiyat += 25;    //Fiyatı : 25
        break;
      case 2:
        fiyat += 25;    //Fiyatı = 25 + 25 = 50
        goto case 1;
      case 3:
        fiyat += 25;    //Fiyatı 25 + 25 + 25 = 75
        goto case 2;
      default:
        Console.WriteLine("Yanlış seçim");
        break;
}

if (fiyat != 0)
{
      Console.WriteLine("Lütfen makineye {0} kuruş atınız.", fiyat);
}

Console.WriteLine("Teşekkürler...");

...
```

Burada doğru hesaplamayı yapmak için case'ler arasında 'goto' lar ile paslaşmalar yer almaktadır.

- Eğer kullanıcı küçük boy kahve istiyorsa 1 girer. 'n' değişkeninin değerine göre uygulamayı yönlendirecek olan switch kontrolünün içerisinde case 1:'den sonraki kod çalışır ve başlangıç değeri 0 olan fiyat değişkeninin değeri 25 olarak değiştirilir. Ardından break, kontrolü switch bloğunun sonuna götürür ve ekrana fiyatla ilgili yazılar yazılır.
- Eğer kullanıcı orta boy kahve istiyorsa 2 girer. Switch kontrolünün içerisinde case 2:'den sonraki kod çalışır ve fiyat değişkeninin değeri yine 25 yapılır; ancak kod dikkatle incelenirse kontrol bu noktada break ile switch bloğu sonuna değil 'goto' ile case 1'e transfer olmaktadır ve yukarı maddedeki süreç tekrar eder. Dolayısıyla fiyat 25 + 25 = 50 kuruş olarak elde edilip kulanıcıya bildirilir.
- Eğer kullanıcı büyük boy kahve istiyorsa 3 girer. Bu sefer önce case 3'ün içerisindeki kod çalışıp fiyat değişkenini 25 yapıp kontrolü case 2'ye teslim eder; o fiyatı 50'ye tamamlar ve son olarak case 2 kontrolü case 1'e devreder ve nihai fiyat 75 kuruş olarak hesaplanır..
- Bunun avantajı, eğer küçük seçimin case'indeki fiyat artışında bir değişiklik yapılırsa bu artışın diğer 2 boya otomatik olarak yansımasıdır.

Çalışma Zamanı Hatalarının Yönetimi (Exception Management)

Nasıl ki gerçek hayatta araba sürerken lastik patlaması, televizyon izlerken tv tüpü bitmesi (tamam, artık birçok tv sıvı gazla çalışıyor olabilir!) ya da taşınabilir elektronik bir alet ile müzik dinlerken pil bitmesi gibi beklenmedik olaylar gerçekleşiyorsa yazılan uygulamanın kullanıcıları da beklenmedik davranışlarda bulunabilir. Program geliştirirken derleme zamanı hatalarını zaten oluştuğu anda çözerek yola devam ederiz; ancak uygulama çalışırken de her hangi bir aşamasında hata meydana gelebilir. Buradaki soru şudur: Çalışma anında oluşan hatayı nasıl algılarız ve ardından nasıl uygun bir çözüm üretiriz? Eğer sağlıklı bir C# programı geliştirmek istiyorsak **istisna**lardan **(exception)** faydalanmak durumundayız. İstisna, çalışma zamanında oluşan hatalara verilen genel isimdir.

Kodu Dene (try) ve Hataları Yakala (catch)

CLR, .NET uygulamalarının çalışma zamanında üretecekleri istisnaları yönetebilecek bir mekanizmaya sahiptir. Ancak bu mekanizmanın doğru bir şekilde ele alınabilmesi uygulama geliştiricinin kod içerisinde yapacağı düzenlemelere bağlıdır. Nitekim uygulamaların çalışma zamanında herhangi bir hata nedeniyle (özellikle CLR tarafından tespit edilebilen) istem dışı sonlandırılması istenmez. **try / catch** blokları bu istisna yönetim mekanizmasının uygulama geliştirici tarafından ele alınan parçasıdır. İstisna kontrolüne sahip kod yazmak için yapılması gerekenler şunlardır:

1. Kod **try** bloğu içerisine yazılır. Kod çalıştığı zaman, **try** bloğundaki bütün satırlar sırasıyla çalıştırılır. Eğer bu blok içerisindeki satırlardan herhangi biri hata oluşturmazsa bütün satırlar sağlıklı bir şekilde çalışmaya devam ederler. Eğer bir hata oluşmuş ise kontrol, try bloğundaki hatanın oluştuğu satırdan catch bloğuna geçer.
2. **try** bloğundaki kodların herhangi bir satırında hata oluşursa; çalışma zamanı ortama hata fırlatır. Bu fırlatılan hata **istisna (exception)** dır. Çok sayıda istisna tipi bulunmaktadır. Fırlatılan istisnanın tipi, ilgili satırda yapılmak istenen ama başarılamayan işle ilgili olur. Artık try bloğundaki hata oluşan satırın altındaki kodların çalışma ihtimali yoktur; çünkü çalışma zamanı, istisnayı ortama fırlattıktan sonra bir **catch** bloğu arar. Eğer uygun bir **catch** bloğu bulunursa kontrol otomatik olarak oraya transfer olur. **catch** blokları belli bir istisna tipini ele almak üzere dizayn edilmişlerdir. Bu şekilde çalışma zamanında oluşan farklı tipteki hataları farklı şekilde ele alma şansımız olmaktadır. Burada hatayı ele almaktan çıkarılacak anlam, genelde kullanıcıya hatayla ilgili anlamlı bir hata mesajı göstermektir. Bu hata mesajı çok ayrıntılı olmamalıdır; yani kullanıcıya önemli veriler gösterilmemelidir. Aynı zamanda çok da genel olmamalıdır. Oluşan hatanın sebebini yalın bir dille anlatmak, en uygun yoldur.

CLR, ortama framework içerisinde önceden tanımlanmış istisna sınıflarına ait nesne örnekleri fırlatabileceği gibi geliştirici tarafından tasarlanmış istisna nesnelerini de fırlatabilir.

Yukarıda iki maddede özetlenen durum aşağıdaki örnekte incelenebilir. Bu örnekte kullanıcıdan alınan değer int tipine dönüştürülerek aritmetik bir işlemde kullanılmak istenmektedir. Kullanıcıdan alınan değer Console ekranının özelliği gereği string olur. Bu değerin int tipine sağlıklı bir şekilde dönüştürülmesi gerekir. Bunun için string ifade, int tipine **Parse()** edilir. Ancak Parse() işleminin hatasız çalışması için ekrandan alınan değerin sayısal değere dönüştürülebilir olması gerekmektedir. Eğer olmazsa dönüştürme satırında bir hata meydana gelir. Bu hatanın oluşmasının dizayn zamanında önüne

geçilme şansı yoktur; çünkü görüldüğü gibi hatanın kaynağı kullanıcıdan alınan giriştir. Zaten genelde çalışma zamanı hatalarının kaynağı kullanıcıdan alınan bu tarz girişler olur.

- Öncelikle kodu hata yönetimi olmadan Main() metodu içerisine yazalım:

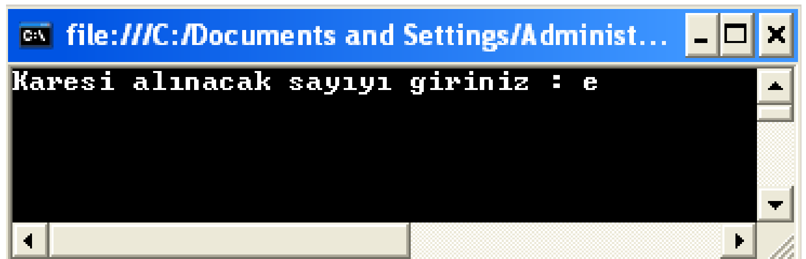
```
...

Console.Write("Karesi alınacak sayıyı giriniz : ");
int kok = int.Parse(Console.ReadLine());
int sonuc = kok * kok;
Console.WriteLine("{0} sayısının karesi = {1}",kok,sonuc);

...
```

- Burada yapılan işlem, kullanıcıdan bir sayı istenip sonuç olarak bu sayının karesininin hesaplanması ve kullanıcıya gösterilmesidir. Uygulamayı test edelim.

```
file:///C:/Documents and Settings/Administra...
Karesi alınacak sayıyı giriniz : 4
4 sayısının karesi = 16
```

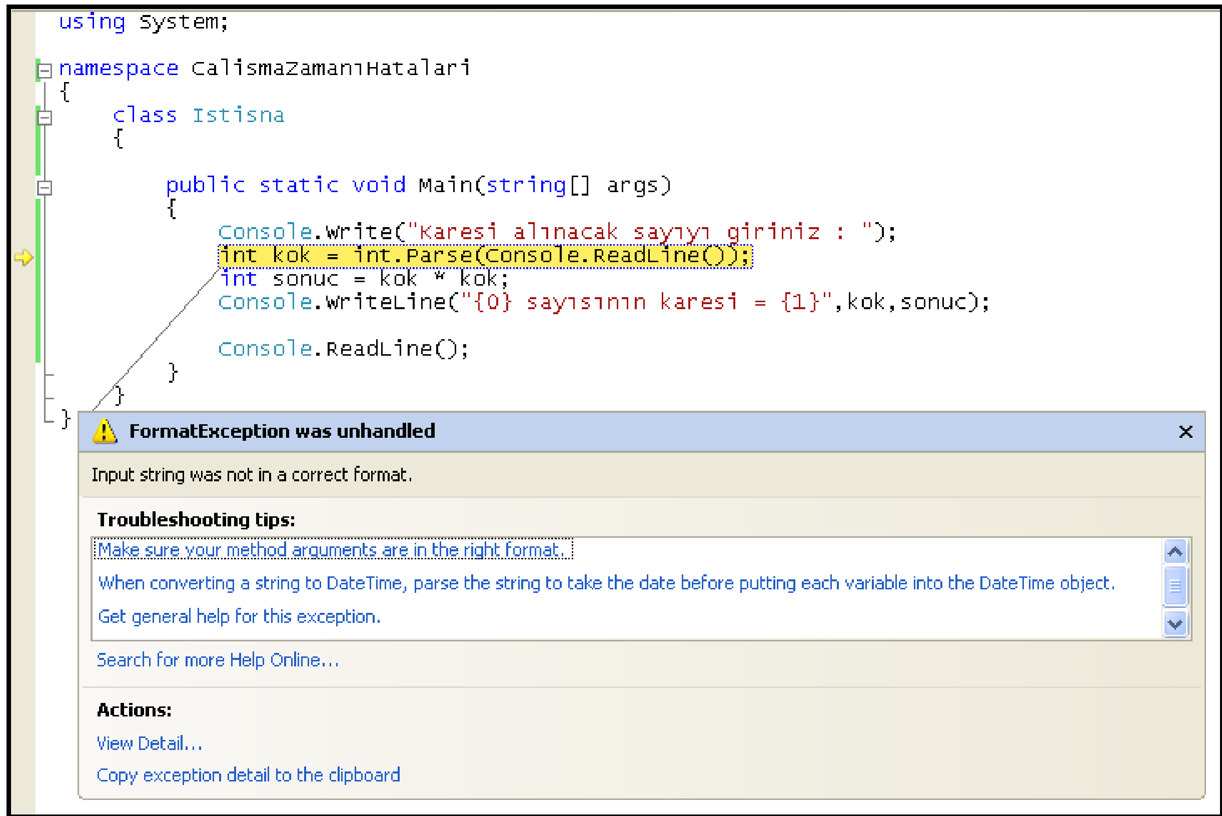
Şekil 99: Uygulama doğru bir şekilde çalışıyor.

int tipine dönüştürülebilecek bir sayı girildiğine, bu sayının string temsili, int tipine Parse() edilebilir ve uygulama sorunsuz çalışır. Şimdi de dönüştürme yapılamayacak bir giriş yapıldığını varsayalım. Bu, gerçek hayatta da karşılaşılabilecek bir durumdur. Kullanıcıdan alınan verilerin her zaman için hatalı olma riski bulunmaktadır. Dolayısıyla bu verileri kontrolden geçirerek işlemek, olası çalışma zamanı hatalarına karşı da gerekli denetim mekanizmalarını uygulamak düşünülebilir. (Verinin doğruluğunu test ederek kullanmak, çalışma zamanı muhtemel hatalarını ele almak vb.)



Şekil 100: Uygulamada hata oluşturacak bir giriş yapılıyor

- Kullanıcı ENTER'a bastığında karşılaşacağı ekran aşağıdadır:



Şekil 101: Çalışma zamanı hatası oluşması ile CLR tarafından ortama fırlatılan istisna

- Kullanıcıdan alınan veri **e** olunca, Parse() metodu int tipine dönüştürecek doğru veriyi bulamadığı için o satırda çalışma zamanı hatası oluşur ve CLR tarafından ortama bir istisna (exception) fırlatılır.

Yukarıdaki gibi bir ekran, eğer uygulama debug modda çalıştırıldığında karşımıza çıkar, yani Debug → Start Debugging ile ya da F5 ile uygulama çalıştırıldığında oluşur.

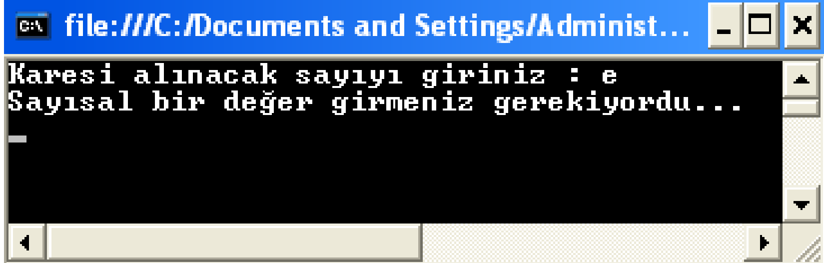
- Ortama fırlatılan bu istisnanın tipi FormatException'dır. Bu ve bunun gibi birçok istisna tipi bulunmaktadır. (Hatanın oluştuğu anı yakalayabilmek için uygulamayı F11 ile başlatıp yine F11'le satır satır ilerlemek faydalı olur). Tabiki bir kullanıcının böyle bir ekranla karşılaşmasını istemeyiz. Yapılması gereken, kullanıcıya doğru formatta veri girişi yapmadığına dair bir mesaj vermektir. Bu işlemin yapılacağı yer **catch** bloklarıdır. Yukarıdaki kodda CLR herhangi bir catch bloğu bulamadığı için bu ekranla karşılaşıldı. **catch** bloğunda ele alınması gereken istisna tipini de biliyoruz : FormatException (Hata penceresinin sol üst köşesinde yazmaktadır. Hemen altında da bu hatayla ilişki kısa bir mesaj yer almaktadır)

```
...

try
{
	Console.Write("Karesi alınacak sayıyı giriniz : ");
	int kok = int.Parse(Console.ReadLine());
	int sonuc = kok * kok;
	Console.WriteLine("{0} sayısının karesi = {1}", kok, sonuc);
}
catch (FormatException)
{
	Console.WriteLine("Sayısal bir değer girmeniz gerekiyordu...");
}
```

```
...
```

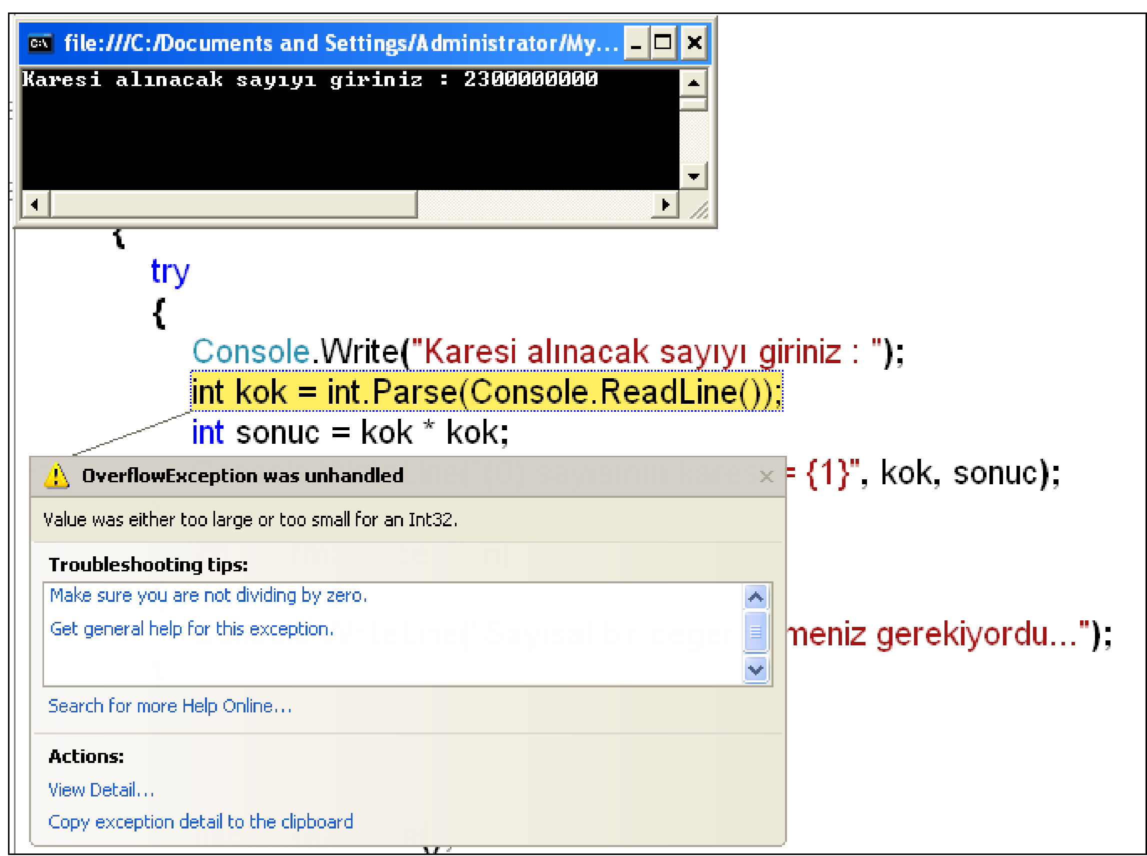
- Çalışma zamanında hata oluşturması muhtemel kod, **try** bloğuna alınır. Hemen altına da doğru **catch** bloğu koyularak kontrolün, hata oluşan satırdan doğrudan buraya transfer olması sağlanır ve ardından kullanıcıya mesaj gösterilir.

FormatException aslında bir sınıftır. Bütün istisnalar, .NET sınıf kütüphanesinde ayrı birer sınıf ile temsil edilir.



Şekil 102: İstisna yönetimi yapılmış bir uygulama

- Uygulamalarda tek bir istisna tipiyle karşılaşılmayacaktır. Örneğin yukarıdaki kodda doğru formatta veri girişi yapılmadığı için **FormatException** tipinde bir istisna alınır. Peki eğer kullanıcı doğru formatta veri girer; ancak int tipinin sınırları dışına çıkarsa ne olur? Görelim :



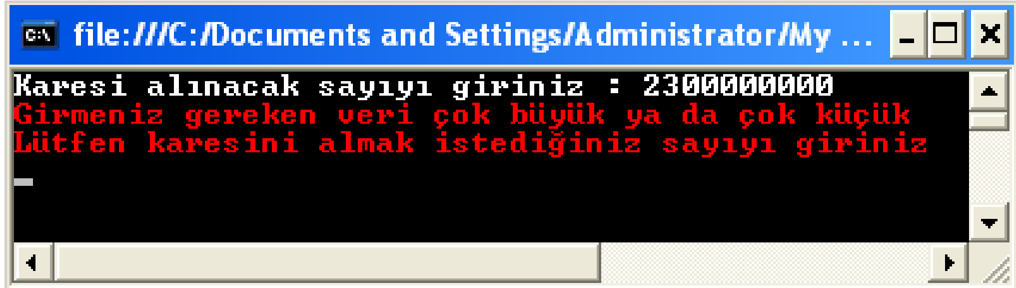
Şekil 103: Tek bir catch bloğu, özel bir istisna tipini yakalamak için özelleşmiştir.

- Bahsetmiş olduğumuz hata ele alınmadığı için yine istenilmeyen bir ekranla karşılaşılmıştır. İstisna yönetimi yaparken, birden fazla catch bloğu yazılabilir.

Öyleyse koda OverflowException tipindeki istisnayı ele alan bir catch bloğu daha eklenebilir.

```
...

try
{
      Console.Write("Karesi alınacak sayıyı giriniz : ");
      int kok = int.Parse(Console.ReadLine());
      int sonuc = kok * kok;
      Console.WriteLine("{0} sayısının karesi = {1}", kok, sonuc);
}
catch (FormatException)
{
      Console.WriteLine("Sayısal bir değer girmeniz gerekiyordu...");
}
catch (OverflowException)
{
      Console.ForegroundColor = ConsoleColor.Red;
      Console.WriteLine("Girmeniz gereken veri çok büyük ya da çok
      küçük\nLütfen karesini almak istediğiniz sayıyı giriniz");
}

...
```

- Uygulama çalıştırılıp yine int tipinin sınırları dışında bir giriş yapılırsa kullanıcı bu sefer de o hataya özel mesajla karşılaşır. Gerçek hayat uygulamalarında da en uygun davranış bu olmalıdır. Yani, oluşabilecek muhtemel çalışma zamanı hataları test edilip, kod try bloğuna alındıktan sonra herbir hata için ayrı catch bloğu kullanılmalıdır:



Şekil 104: Çoklu catch blokları ile istisna yönetimi

- Ancak ne kadar test edilse de hala geliştirici tarafından tespit edilemeyen çalışma zamanı hataları oluşabilir. Bu noktada başka bir ihtiyaç devreye girer. Bütün istisna sınıflarını tek bir catch bloğunda üzerine çekebilecek bir istisna tipi olsa fena olmazdı. Bu tip **Exception** tipidir. Gerek test edilen ve ele alınan catch bloklarının en sonunda emniyet sübabı olarak kullanılabilir (Genel istisna ele alımı diğer catch bloklarından sonra yapılmalıdır); gerekse tek başına bir catch bloğu olarak kullanılıp bütün hatalarda kullanıcıya aynı hata mesajı gösterilebilir. İlk yol tercih edilmelidir; çünkü oluşan her hatanın tek bir catch bloğu ile ele alınabilmesi her ne kadar kolay olsa da hepsi için aynı hata mesajı çok anlamlı olmaz. Bunun yerine test edilebilen çalışma zamanı hataları, kendi tiplerinde catch bloklarında ele alınıp, en sona da bu genel istisna sınıfı eklenir.

```
...

try
{
      Console.Write("Karesi alınacak sayıyı giriniz : ");
      int kok = int.Parse(Console.ReadLine());
      int sonuc = kok * kok;
```

```
        Console.WriteLine("{0} sayısının karesi = {1}", kok, sonuc);
    }
    catch (FormatException)
    {
        Console.WriteLine("Sayısal bir değer girmeniz gerekiyordu...");
    }
    catch (OverflowException)
    {
        Console.ForegroundColor = ConsoleColor.Red;
        Console.WriteLine("Girmeniz gereken veri çok büyük ya da çok
        küçük\nLütfen karesini almak istediğiniz sayıyı giriniz");
    }
    catch (Exception)
    {
        Console.WriteLine("Bir hata oluştu. Lütfen tekrar deneyiniz...");
    }

    ...
```

- Eğer kontrol genel istisna tipinin ele alındığı, **FormatException** ve **OverflowException** dışındaki bütün çalışma zamanı hataları oluşması durumunda girilecek **Exception** tipinin catch bloğuna girerse, kullanıcıya genel bir hata mesajı verilmelidir.

- İstisna yönetimi ile ilgili şimdilik söylenebilecek son şey; her istisna tipinin kendine ait hata mesajlarını kullanabilmesidir. Her ne kadar kendi hata mesajlarımızı yazmak en doğru yöntem olsa da bu hata mesajlarından da faydalanılabilir. Bunun için yapılması gereken, catch blokları içerisinde bildirilen istisna tipinden bir değişken oluşturmak ve catch bloğu içerisinde aşağıda görüldüğü şekilde **Message** özelliğini kullanmaktır. **Message** özelliği, bütün istisna sınıfları için ortak bir özelliktir.

```
    ...

    try
    {
        Console.Write("Karesi alınacak sayıyı giriniz : ");
        int kok = int.Parse(Console.ReadLine());
        int sonuc = kok * kok;
        Console.WriteLine("{0} sayısının karesi = {1}", kok, sonuc);
    }
    catch (FormatException hata)
    {
        Console.WriteLine(hata.Message);
    }
    catch (OverflowException hata)
    {
        Console.ForegroundColor = ConsoleColor.Red;
        Console.WriteLine(hata.Message);

    }
    catch (Exception hata)
    {
        Console.WriteLine(hata.Message);
    }

    ...
```

Herbir catch bloğunda kullanılan **hata** değişkeni, kendi catch bloğu içerisinde geçerlidir. Dolayısıyla her bloğun sonunda hata isimli değişken bellekten düşürülmekte ve bir sonraki blokta yeniden belleğe sorunsuzca çıkarılmaktadır.

- İstisnalar çalışma zamanında CLR tarafından fırlatılır; geliştiriciler ise catch blokları yazarak CLR'ın fırlattığı istisnaları ele alır. Aynı zamanda istisna fırlatma işini

uygulama geliştirici de yapabilir. Özellikle bazı durumlarda programcı tarafından geliştirilen tiplerin kullanımı sırasında, gerekli koşulların sağlanmaması halinde ortama **catch** blokları tarafından ele alınabilecek istisna nesneleri fırlatılmak istenebilir. Bu durum, **throw** anahtar kelimesi kullanılarak gerek framework içerisindeki önceden tanımlı, gerek programcı tarafından tasarlanmış istisna tipleri için gerçekleştirilebilir.

```
throw new Exception();
```

- Yukarıdaki örnek incelenecek olursa kullanıcıdan alınan değer kendisi ile çarpılıp yine **int** tipli bir değişken üzerinden gösterilmektedir. Bu sonuç değişkeninin **int** tipinin sınırları dahilinde kalması için kullanıcının girmesi gereken maksimum sayı 46340, minimum sayı ise -46340'dır. Eğer -46340'dan küçük ya da 46340'dan büyük bir sayı girilirse yine **OverflowException** oluşur ama bu sefer **Parse()** işleminin yapıldığı satırda değil, çarpma işlemini yapıldığı satırda oluşur.

```
int sonuc = kok * kok;
```

- Dolayısıyla girilen sayıyı başarılı bir şekilde **int** tipte değişkene **Parse()** ettikten sonra belirtilen aralıklar için kodun boşuna çalışmasını engellemek adına bu duruma uygun olan istisna tipini ortama biz fırlatırız. Ayrıca kendi yazabileceğimiz istisna tiplerini de **throw** anahtar kelimesi ile fırlatabiliriz.

```
...

try
{
     Console.Write("Karesi alınacak sayıyı giriniz : ");
     int kok = int.Parse(Console.ReadLine());
     if ((kok > 46340) || (kok < -46340))
     {
          throw new OverflowException();
     }
     int sonuc = kok * kok;
     Console.WriteLine("{0} sayısının karesi = {1}", kok, sonuc);
}

...
```

BÖLÜM 5: DİZİLER ve KOLEKSİYONLARA GİRİŞ

Dizi Nedir? Neden İhtiyaç Duyarız?

Diziler aynı tipte değerlerin bir arada taşınmasını sağlayan referans tipli programlama öğeleridir. Diziler, elemanlarına ismiyle değil indeks sıralarıyla erişim sunar. Örneğin 30 öğrencinin notlarını tutmak istediğimizi düşünelim. Tip olarak **byte** işimizi görecektir (0 – 255 aralığında değer tutabilir). Bu programlama görevini yerine getirebilmek için byte tipinde 30 tane değişkene ihtiyaç olur. Ancak bu, takip etmesi oldukça güç durumlara yol açabilir. Bunun yerine byte tipinde 30 elemanlı bir dizi oluşturmak ve öğrencilerin notlarını bu dizide tutmak daha etkin bir çözümdür. Ayrıca program içerisinde 30 kişinin notunu bir yerden başka bir yere taşımak da daha kolay olacaktır. Diziye atılan her bir eleman otomatik olarak 0'dan başlayıp 1'er 1'er artan bir indeksleme sistemi ile kaydedilir. Daha sonra dizi elemanlarına erişilmek istendiğinde bu indeksler ile erişilir. Öğrencileri sınıf listesindeki sıralarına göre diziye atıp sonra da bu sıraya bakılarak, istendiğinde notlarına erişilebilir. Diziler referans tiplidirler. Dolayısıyla diziye atılan veriler belleğin **heap** bölgesinde tutulurken, diziye erişim için oluşturulan değişken belleğin **stack** bölgesinde tutulur.

Dizileri Kullanmak

Dizi Değişkenleri Tanımlamak

Bir dizi içerisinde tek tipten değerler tutabileceği için tanımlamasında bu tip belirtilmelidir. Dizi değişkeni tanımlamasını normal değişken tanımlamasından ayıran şey tip adının ardından koyulan köşeli parantezlerdir. Örneğin **sayilarim** adında int tipinde bir dizi tanımlaması aşağıdaki gibi yapılır:

```
int[] sayilarim;
```

Burada tanımlanan tip, sadece önceden tanımlı tipler olmak zorunda değildir. Temel sınıf kütüphanesindeki herhangi bir tipte olabileceği gibi kendi yazdığımız tipleri içeren bir dizi de tanımlanabilir.

Dizi değişkenlerine isim verirken çoğul adlar verilirse bu değişkenin bir diziyi işaret ettiği rahatlıkla anlaşılabilir.

Dizi Örnekleri Oluşturmak

İçerisinde tutulacak elemanların tipi ne olursa olsun diziler, referans tiplidirler. Yani belleğin 'stack' bölgesindeki dizi değişkeni, belleğin 'heap' bölgesindeki bir dizi örneğinin adresini tutup işaret edecek demektir; dizi değişkeni 'stack' da kendi üzerindeki verileri saklamaz.

Belleğin 'heap' bölgesinde veriler için yer ayrılması, referans tipli değişkenlerin 'new' anahtar kelimesi ile oluşturulmaları ile gerçekleşir. Diziler de referans tipli olduğu için bu kural üzerinden çalışırlar.

Dizi değişkenini tanımlarken dizinin bellekte kaç tane eleman saklayacağı bilgisini vermezken, verileri belleğe çıkardığımız yer olan 'new' anahtar kelimesi ile oluşturulduklarında bu bilgiyi vermek zorundayız. Diziler oluşturulurken, kaç elemanlı oldukları söylenmek zorundadır. Bellekte, bu bilgiye göre yer açılır ve ilerleyen

aşamalarda bu alana belirtilen miktardan fazla eleman girilmeye çalışıldığında boyut dinamik olarak artmaz.

Hatırlanacağı gibi referans tipleri oluşturulduklarında, üyeleri kendi tiplerinin varsayılan değerlerini alırlar. Daha sonra bu değerler, ihtiyaçlar doğrultusunda değiştirilir. Öyleyse 10 elemanlı bir bool dizisi 'new' ile oluşturulduğunda bu dizinin 10 elemanının değeri de doğrudan 'false' (bool tipinin varsayılan değeri) yapılır. Yukarıda değişkeni tanımlanan dizi, şimdi de gerçekten oluşturulur:

```
sayilarim = new int[4];
```

Bu şekilde 4 elamanlı int tipli bir dizi oluşturulur. Burada eleman sayısı sabit olmak zorunda değil. Tek kural var; o da bir dizi oluşturulduğunda kaç elamanlı olduğunu bir şekilde bilmelidir. O yüzden bu dizi örneğinin belleğe çıkarılışının birden fazla yolu kullanılabilir. Örneğin dizinin eleman sayısı çalışma zamanında belirlenebilir:

```
...

Console.Write("Lütfen öğrenci sayınızı giriniz : ");
int boyut = int.Parse(Console.ReadLine());
sayilarim = new int[boyut];
Console.WriteLine("{0} elemanlı diziniz oluşturuldu...",boyut);

...
```

Aynı zamanda çok boyutlu diziler de oluşturulabilir. Çok boyutlu diziler bu kitabın kapsamı dışında olsa da nasıl oluşturulacakları aşağıda yer almaktadır:

```
int[,] tablo = new int[3,4];
```

Dizilere Başlangıç Değerlerini Vermek

Bir dizi oluşturulduğu zaman dizinin içi, oluşturulduğu tipin varsayılan değerleriyle doldurulur. Dizinin bu davranışı değiştirilebilir ve istenilen değerlerle belleğe çıkarılması sağlanabilir. Bunu sağlamak için, dizi oluşturulma söz diziminin sonunda, belirtilen tipte ve belirtilen eleman sayısı kadar değerin çevresinde süslü parantezler, aralarında virgül olacak şekilde yazılmasıyla sağlanır. Örnek olarak yukarıdaki **sayilarim** adlı dizi 4,9,7 ve 3 değelerini içerecek şekilde aşağıdaki kod parçasında olduğu gibi oluşturulabilir.

```
sayilarim = new int[4] { 4,9,7,3};
```

Burada dizi boyutunun 4 elemanlı olacağı bildirildiği için süslü parantezler içerisine 4 elemandan az ya da fazla değer girilmesi durumunda derleme zamanı hatası alınır. Birkaç veri tipi için bu söz diziminin kullanılışı aşağıda yer almaktadır:

```
string[] sehirler = new string[] { "Antalya","İstanbul","Ankara" };
double[] oranlar = new double[3] { 0.32 , 0.45 , 0.56 };
char[] alfabe = new char[3] { 'a','b','c'};
```

Dizi oluşturulurken eğer başlangıç değerleri veriliyorsa eleman sayısı belirtilmek zorunda değildir. Derleyici başlangıç değerlerinin sayısından, dizinin eleman sayısını hesaplar.

```
string[] sehirler = new string[] { "Antalya","İstanbul","Ankara" };
```

Dizi oluşturulurken eğer başlangıç değerleri veriliyorsa **new** anahtar kelimesi ihmal edilebilir. Derleyici başlangıç değerlerinin sayısından dizinin eleman sayısını otomatik olarak hesaplar.

```
string[] sehirler = { "Antalya","İstanbul","Ankara" };
```

Her Bir Dizi Elemanına Erişmek

İstenilen dizi elemanına erişmek için dizi üzerinden ilgili elemanın indeks sırası verilir. Örneğin **sayilarim** adiyla oluşturulup değerleri verilen dizinin son elemanına erişmeye çalıştığımızı düşünelim. Burada dikkat edilmesi gereken nokta; **C#'da dizi mantığındaki herşeyin indeks olarak 0 değeri ile başlamasıdır**. Tıpkı ekrana bir şey yazdırırken kullanılan yer tutucularda (placeholder) olduğu gibi ({0}). O yüzden aşağıdaki dizinin

- 0. indeksli elemanı 4
- 1.indeksli elemanı 9
- 2.indeksli elemanı 7
- 3.indeksli elemanı ise 3

'tür.

```
int[] sayilarim = new int[4] { 4,9,7,3};
int sayi2 = sayilarim[1];
Console.WriteLine("Dizinin {0}. indeksli elemanı : {1}",1,sayi2);
```

Bir dizinin içeriği, yine aynı yolla değiştirilebilir. Dizinin ikinci indeksli elemanı olan 7 değerini aşağıdaki kod parçasında görüldüğü gibi 10 olarak değiştirebiliriz.

```
sayilarim[2] = 10;
// ya da
int baskaSayi = 10;
sayilarim[2] = baskaSayi;
```

Bir diziye değerleri verilmeden sadece eleman sayısı söylenerek (varsayılan değerleri ile) oluşturulduktan sonra dizi elemanları tek tek verilebilir.

```
int[] sayilarim = new int[4];
sayilarim[0] = 3;
sayilarim[1] = 5;
sayilarim[2] = 7;
sayilarim[3] = 9;
```

Dizinin elemanlarına erişimde 0'dan başlayan indeksler kullanıldığı için dizinin son indeksi, <eleman sayısı -1> olarak ele alınabilir. Eğer yanlışlıkla üç elemanlı bir dizinin üçüncü indeksli elemanına erişilmeye çalışılırsa **IndexOutOfRange** istisnası ile karşılaşılır.

```
try
{
      int[] sayilarim = new int[4] { 4, 9, 7, 3 };
      Console.WriteLine(sayilarim[4]);
}
catch(IndexOutOfRangeException)
{
      Console.WriteLine("Diziye erişim için kullanacağınız indeksler 0 ile
      3 arasında olmalıdır.");
}
```

Bir Dizinin Eleman Sayısını Elde Etmek

Bir dizinin eleman sayısı, dizi örneği üzerinden çağrılan **Length** özelliği ile elde edilebilir.

```
Console.WriteLine("sayilarim dizisinin eleman sayısı : {0}",
sayilarim.Length);
```

Bir Dizi İçerisindeki Bütün Elemanları Elde Etmek

Bir dizideki eleman sayısı dizinin **Length** özelliği ile elde edilebildiğine göre, **for** döngüsü kullanılarak dizinin 0'ncı indeksi ile <eleman sayısı -1>'nci indeksi arasındaki bütün değerler elde edilebilir.

```
int[] sayilarim = new int[4] { 3,5,7,9 };
for (int i = 0; i < sayilarim.Length; i++)
{
      Console.WriteLine("{0}.eleman = {1}",i + 1,sayilarim[i]);
}
```

Yukarıdaki döngüde ekrana her bir elemanın dizideki sırası (indeks + 1) ve değeri yazdırılmaktadır. Döngü değişkeni önce sıfır değerini alır ve ekrana sıradaki eleman indeksi olarak 1'i; sonra da dizideki sıfırıncı indeksli elemanın değeri olan 3'ü yazdırır. Burada dikkat edilmesi gereken noktalardan biri; döngü değişkeninin başlangıç değeri olarak 1'le değil 0'la başlamasıdır. Diğer nokta ise döngünün mantıksal koşul deyiminin, döngü değişkenini dizinin boyutu için çalıştırmamasıdır. (Bu durum, koşul deyiminin i <= sayilarim.Length yerine i < sayilarim.Length olarak yazılmasından kaynaklanır). Döngü bu şekilde dört değer için de çalışır ve dizideki bütün elemanların değerleri elde edilmiş olur.

Dizi Elemanlarını foreach Döngüsü ile Elde Etmek

C#, dizilerin içerisinde dönüp değerlerini elde etmek için bir yol daha önerir: **foreach** döngüsü.

```
foreach (int gecici in sayilarim)
{
      Console.WriteLine(gecici);
}
```

foreach döngüsünde tanımlanan döngü değişkeni otomatik olarak dizideki her elemanı sırayla elde eder. Döngünün ilk iterasyonunda (ilk turunda) belleğe çıkarılan değişkenin tipi, dizinin tipinde olmalıdır. foreach, arka tarafta kendi algoritması ile her iterasyonda dizinin bir elemanını okuyup bellekteki geçeci değişkene atar, ardından döngü bloğu içerisinde değişkenin o anki değeri okunur. **foreach**, dizideki değerleri okuma işini 0. indeksten başlayıp <eleman sayısı - 1>'nci indekse kadar sırayla yapar. **for** döngüsünde düşünülmesi gereken ayrıntılar **foreach**'de olmadığı için dizinin bütün elemanlarını okumada tercih edilir. Ancak yine de bir dizinin elemanları elde edilmek istendiğinde for döngüsü kullanılmak zorunda kalınabilir:

- **foreach** deyimi her zaman bütün diziyi döner. Eğer sadece dizinin belli bir bölümü elde edilmek isteniyorsa (mesela yarısı) ya da belli değerler atlanmak isteniyorsa (Mesela her 3 elemandan biri) bunu for kullanarak yapmak çok daha kolaydır.
- **foreach** deyimi her zaman dizinin 0.indeksinden <eleman sayısı – 1>.'nci indeksine doğru çalışır. Dolayısıyla eğer dizinin değerleri tersten okunmak istenirse for döngüsünü kullanmak daha kolaydır.
- Eğer döngü bloğu içerisinde sadece dizideki elemanın değeri değil indeks sayısı da istenirse, bu desteği sağlamak için for döngüsü kullanılmak durumundadır. (Ya da suni bir değişken oluşturulup arttırım işlemleri yapılabilir.)
- Eğer dizi elemanlarının değerleri bir döngü içerisinde güncellenmek istenirse, for döngüsü kullanılmak zorundadır; çünkü foreach deyimindeki döngü değişkeni, dizideki her bir elemanın sadece okunabilir kopyasını elde eder, bu yüzden de dizi elemanlarını değeri değiştirilemez.

Foreach döngüleri, sadece ileri ve yalnız okunabilir bir öteleme hareketine izin verdiklerinden, bazı durumlarda for döngülerine göre daha performanslı çalışabilirler.

Aşağıda örnek bir uygulama yer bulunmaktadır. Öncelikle kullanıcıdan kaç öğrenci için not gireceği bilgisi alınır. Alınan bu bilgi ile bir dizi oluşturulur ve dizinin dinamik oluşan eleman sayısı kadar kullanıcıdan değer alınıp bunlar dizinin elemanları olarak sırayla kaydedilir. Ardından dizinin boyutu ve elemanları toplu olarak kullanıcıya gösterilir:

```
...

try
{
//Kullanıcıdan oluşturulacak dizinin eleman sayısı alınır...
      Console.Write("Lütfen öğrenci sayınızı giriniz : ");
      byte elemanSayisi = byte.Parse(Console.ReadLine());
      byte[] notlar = new byte[elemanSayisi];
//Kullanıcıdan alınıp döngü içerisinde diziye girilen değerleri tutacak
//olan değişken tanımlanır.
      byte alinanEleman;


//Döngü içerisinde kullanıcıdan sırayla notlar alınıp bir değişken
//aracılığıyla diziye verilir.

      for (byte i = 0; i < notlar.Length; i++)
      {
         Console.Write("Lütfen {0}. öğrencinin notunu girip ENTER'a
basınız : ",i + 1);
         alinanEleman = byte.Parse(Console.ReadLine());
         notlar[i] = alinanEleman;
      }
      Console.WriteLine("Veri girişi tamamlandı. Devam etmek için ENTER'a
      basınız...");
      Console.ReadLine();
//Kullanıcıdan girişleri aldıktan sonra verileri görmek isteyip
//istemediği sorulur.
      etiket:
      Console.Write("Girilen notları görmek ister misiniz? [E] [H] : ");
      string devamMi = Console.ReadLine().ToLower();
//Kullanıcı eğer [E] yazarsa, ona öğrenci sayısını ve notlarını
gösterilir.
//Kullanıcı eğer [H] yazarsa, program sonlandırılır.
//Kullanıcı eğer [E] ve [H] dışında bir değer girerse, tercihi tekrar
//sorulur...
      switch (devamMi)
      {
         case "e":
            Console.WriteLine("\n{0} öğrencinin notları sırasıyla
            aşağıdadır:\n",notlar.Length);
            foreach (byte gecici in notlar)
            {
                  Console.Write(gecici + "  ");
            }
            break;

         case "h":
            break;

         default:
            goto etiket;
      }
}
catch (Exception hata)
{
      Console.WriteLine(hata.Message);
}
```

Koleksiyonlara Giriş

Diziler çok yararlı programlama nesneleridir. Fakat bazı sınırlamaları vardır. Bunlardan birincisi dizilerin sadece tek tipten değerler kabul edebilmesidir. Ancak bu sorun yine dizilerle çözümlenebilir. Biliyorsunuz ki object tipinden bir değişken .NET dünyasındaki bütün tipleri üzerinde taşıyabilmektedir. Öyleyse object tipinden oluşturulan bir dizi ile tip sınırlaması sorunu aşılabilir ve tek bir dizi içerisinde istenilen tipte değerler taşınabilir:

```
object[] karmaDizi = new object[] { 'A',1,true,"Emrah",0.45 };
```

Dizilerle ilgili ikinci sıkıntı ise dizi boyutudur. Bir dizinin boyutu, tanımlama sırasında verildikten sonra **Array** sınıfının **Resize** adlı metodu ile değiştirilebilir; ancak diziye her eleman eklenmesinde ya da çıkarılışında bu üyenin çağrılması gerekmektedir. Bu da sıkıntıcı verici bir durumdur.

C#, birden fazla değeri saklamak için bir programlama öğesi daha sunar: **Koleksiyonlar (Collections). System.Collections** isim alanı altında yer alan birçok koleksiyon sınıfı vardır. Koleksiyon sınıfları, her halükarda object tipinden değer alan, yani her türlü tipte değer kabul eden ve herhangi bir boyut sınırlaması olmayan diziler gibidir.

Bir koleksiyon sınıfı oluşturulurken referans tipli olduğu için **new** anahtar kelimesi kullanılır. Herhangi bir eleman sayısı bildirimi yapılmaz; dahası oluşturulduktan ve elemanları verildikten sonra istenildiği zaman herhangi bir eleman koleksiyondan çıkarılabilir veya yeni bir eleman eklenebilir; boyut dinamik olarak değişir. Ayrıca bütün bunlar tip sınırlaması olmadan yapılır. En sık kullanılan koleksiyon tiplerinden biri **ArrayList**'tir. Bir koleksiyonun oluşturuluşu, değer atanması, değerlerine erişilmesi ve bütün değerlerinin elde edilmesi aşağıdaki kod parçasında olduğu gibi gerçekleştirilebilir:

```
using System;
using System.Collections;

namespace Koleksiyonlar
{
    class ArrayListKullanimi
    {
        public static void Main(string[] args)
        {
            //Koleksiyon sınıfı oluşturulur..
            ArrayList koleksiyonum = new ArrayList();
            //Koleksiyona object tipinden elemanlar eklenir.
            koleksiyonum.Add(1);
            koleksiyonum.Add("Emrah");
            koleksiyonum.Add(25);
            koleksiyonum.Add(new DateTime(1982,2,2));
            //Koleksiyon elemanlarına, dönüşüm (cast) yaparak erişilir.
Çünkü koleksiyon içerisinde object tipi ile tutulurlar.
            string adi = (string) koleksiyonum[1];
            Console.WriteLine("******Koleksiyondaki 1.indeksli elemana
erişildi******\n" + adi);
            //Koleksiyondaki elemanların tümüne erişilir : herhangi bir
döngü türü ile olabilir.
            Console.WriteLine("******Koleksiyondaki tüm elemanlara
erişildi******");
            foreach (object gecici in koleksiyonum)
            {
                Console.WriteLine(gecici);
            }

            Console.ReadLine();
        }
    }
}
```

BÖLÜM 6: METOTLAR

Metot Nedir? Neden İhtiyaç Duyarız?

Metotlar, bir kod bloğunun tamamına isim verip, uygulamanın başka yerlerinde, o isimle çağırıp yeniden kullanılmasına izin veren programlama öğeleridir. Her metodun bir ismi ve bir kod bloğu vardır. Metodun adı, amacına uygun bir isim olmalıdır. Örneğin, bir sınıftaki öğrencilerin başarı oranının hesaplamasını yapan bir metoda **BasariOraniHesapla** gibi bir isim verilebilir. Verilen adla çağrıldığında çalışacak kodlar süslü parantez ({ }) bloğu arasında yazılır. Bazı metotlara kod bloğu içerisinde kullanması için girdi olarak veri aktarılabilir. Ayrıca metot, içeride çalışan kodların sonucunda bilgi döndürebilir. Çoğunlukla bir metot geriye bilgi döndürdüğünde, bu aynı zamanda metodun çalışmasının sonu olur.

- Metotlar çok temel ve güçlü nesnelerdir. Kodun yeniden kullanılabilirliğini sağlarlar. Ayrıca tek bir yerden değişiklik yaparak, kullanıldığı her yerde akış mantığını değiştirebilirler.
- Metotlar, sadece bir sınıf veya bir yapı içerisinde oluşturulabilirler. Bağımsız olarak bir isim alanı altına, sınıf veya yapı dışındaki bir tipin içerisine metot yazılamaz.

Metot Oluşturma Söz Dizimi

Bir C# metodu yazarken takip edilmesi gereken söz dizimi şu olmalıdır:

```
erisimBelirleyicisi  niteleyici  donusTipi  MetotAdi  (parametre listesi)
                     (varsa)                          (varsa)
{

    //Metot çağrıldığında çalışacak kodlar buraya yazılır.

}
```

Bir C# metodu çağrılırken kullanılacak söz dizimi ise şudur.

```
MetotAdi(varsa parametre değerleri);
```

erisimBelirleyicisi*,* metotların dışında bir konudur. Bir metodun sınıf veya yapı üyesi olmasının sonucu belirtilmesi gereken bir anahtar kelimedir. C# dilinde beş erişim belireyicisi vardır ve bu erişim belirleyiciler birer anahtar kelime ile temsil edilir. Burada erişim belirleyicilerin sadece ikisinden bahsedilecektir: Private ve Public.

Private erişim belirleyicisi metodun sadece yazıldığı sınıf veya yapı içerisinden çağırılıp kullanılabileceği; **public** erişim belirleyicisi ise metodun her yerden çağırılıp kullanılabileceği (kendi yazdığımız başka yapı ve sınıfların içerisinden) anlamına gelir. Metot yazılırken erişim belirleyicisi belirtilmezse metodun erişim belirleyicisinin private olduğu varsayılır.

donusTipi*,* bir .NET tip adıdır ve metodun çalışması sonucu, kendisini çağırıp kullanana göndereceği verinin tipini belirler. Bu tip int, string gibi önceden tanımlı tipler olabileceği gibi kendi yazdığımız her hangi bir tip de olabilir. Eğer geriye değer döndürmeyen bir metot yazılıyorsa bu kısma **void** kelimesi yazılmalıdır. Mesela bir metot, kod bloğu içerisinde bazı işler yaptıktan sonra sadece ekrana bir şeyler yazdırıyorsa, değer döndürmüyor demektir ve dönüş tipi **void** olarak tanımlanır. Başka bir metot, içeride yaptığı bir hesaplama sonucunu çağırıldığı ortama aktarıyorsa aktardığı bilginin veri tipi neyse, dönüş tipi de o olacaktır (Metot içerisinde iki int tipli değişkenin toplamı, metodu kullanan ortama **int** dönüş tipiyle bir değer gönderebilir.)

niteleyici, ileri seviye programlamada kullanılan **static**, **abstract**, **sealed**, **virtual** kelimelerinden birinin yerini tutar. (Bu anahtar kelimelerin kullanıldığı konular, kitabın kapsamı dışında kaldığı için **static** hariç bu anahtar kelimeler anlatılmayacaktır.)

MetotAdi, metot çağrılırken kullanılacak olan isimdir. Bu isim verilirken takip edilecek kural ve öneriler değişkenlerinki ile benzerken sadece öneriler kısmında adın baş harfinin büyük olması, kod yazım standartlarına destek vermek adına tavsiye edilir (Örneğin değişkenler için girisSayisi iken, metotlar için GirisSayisiniHesapla() olur).

parametre listesi, opsiyoneldir; yani kullanılabilir de kullanılmayabilir de. Ancak her iki durum için de metot adından sonra açılış ve kapanış parantezleri bulunmak zorundadır. Parametreler, metodun çalışması sırasında kullanılabilecek değerlerdir. Metodun yazım aşamasında sanki varmış gibi kabul edilerek kullanılırlar. Gerçek değerleri, metodun çağrıldığı yerde verilir. Bu sayede son derece esnek uygulama geliştirme modeli sunarlar. Kullanılacak parametreler, metodun yazım aşamasında metot adından sonra açılan parantezin ardından sırayla tip adı ve değişken adı olacak şekilde tanımlanır. Varsa diğer parametreler, aralarında virgül olacak şekilde aynı kural takip edilerek sıralanır.

Metot bloğu, metot çağrıldığında çalışacak olan kodların yazılacağı yerdir ve süslü parantezlerle sınırlanır ({ }).

Metot Nasıl Yazılır ve Kullanılır ?

Aşağıda basit bir metot örneği yer almaktadır:

```
using System;

namespace Metotlar
{
    class MetotKullanimi
    {
        public static void MesajYazdir()
        {
            Console.WriteLine("Merhaba C# metotları...");
        }

        public static void Main(string[] args)
        {
            MesajYazdir();

            Console.ReadLine();
        }
    }
}
```

Şekil 105: Basit bir metot örneği

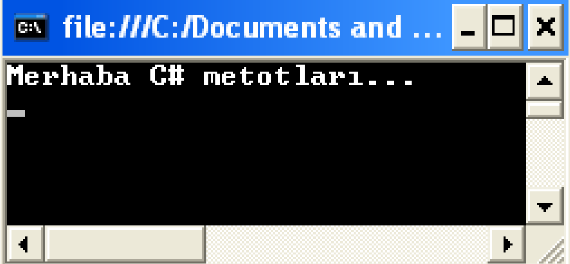
Yukarıda, geriye değer döndürmeyen ve çağrıldığında sadece ekrana bir şeyler yazdıran **MesajYazdir** adında bir metot tanımlanmaktadır. Geriye değer döndürmediği için dönüş tipi olarak void anahtar kelimesi belirtilmiştir.

Bir metot sadece sınıf ya da yapı içerisine yazılabilir. Bu metoda baktığımızda **MetotKullanimi** sınıfının içerisinde yazıldığı ve dolayısıyla bu sınıfın bir üyesi olduğu görülür.

Ayrıca metodun, **static** niteleyicisi ile yazıldığı görülmektedir. Main metodunun bulunduğu sınıf içerisine bir metot yazılırsa, doğrudan metot adıyla erişim için static niteleyicisinin kullanılması gerekmektedir. Bu kuralı test etmek için, static kelimesini kaldırıp uygulama yeniden çalıştırılabilir. Bu durumda derleme zamanı hatası alınır. Bu kelimenin sürece kattığı gerçek anlam, sınıf kavramı hakkıyla öğrenildiğinden daha kolay anlaşılır.

MesajYazdir isimli metodun, Main() içerisinde çağrılması sonucu oluşan ekran çıktısı aşağıdadır. (Şu ana kadarki bütün örnekler Main() metodu içerisinde yazılmıştır. Bunun

sebebi Main() 'in uygulamanın başlangıç noktası olmasıdır. O yüzden metodun çağrıldığı yer yine Main()'in içidir).



Şekil 106: Metot örneğinin çalışması

Metotların içerisinde yazılacak kodun satır sınırı gibi bir kavram yoktur. İstenilen programlama görevi, bir metot içerisinde halledilebilir. Ancak satır sayısı çok fazla ise yapılan işleri farklı metotlara bölmek daha faydalı olabilir. Yazılan herhangi bir kod bloğunu, uygulamanın birçok yerinde kullanma ihtiyacı varsa, bu kodlar her ihtiyaç olduğunda tekrar tekrar yazılmak yerine bir defa metot içerisine yazılır; daha sonra nerede isteniyorsa metodun adıyla kullanılır. Bu, aynı zamanda program kodunun daha rahat okunabilmesine, optimize edilebilmesine ve bakımının kolayca yapılabilmesine olanak sağlar. Örneğin uygulamanın çeşitli yerlerinde güncel tarih ve saat kullanıcıya gösterilmek istenirse ilgili kodu aşağıdaki gibi her yere yazmak tercih edilmez.

```
static void Main(string[] args)
{
	Console.WriteLine(DateTime.Now);

	//...

	Console.WriteLine(DateTime.Now);

	//...

	Console.WriteLine(DateTime.Now);

	Console.Read();
}
```

Bunun yerine güncel tarih ve saati gösteren kod bir metot içerisine yazılır ve istenilen yerde metot çağrılır. (Kodun Tekrar Kullanılabilirliği – Code Reusability)

```
...

static void TarihGoster()
{
	Console.WriteLine(DateTime.Now);
}

public static void Main(string[] args)
{
	TarihGoster();

	//...

	TarihGoster();

	//...

	TarihGoster();

	Console.Read();
}
```

```
...
```

Metot içerisindeki kod biraz daha uzun olduğunda tabiki daha anlamlı olacaktır.

Bu yaklaşımın bir kazancı daha vardır: Kodda bir değişiklik yapmak istenildiğinde eğer metot kullanılmamışsa; bu değişikliğin kodun yazıldığı her yerde ayrı ayrı yapılması gerekir. Ancak metot kullanılmışsa; değişiklik sadece bir yerde yapılır ve metodun kullanıldığı her nokta bu değişiklikten otomatik olarak etkilenir. (Kodun bakım kolaylığı – Code Maintability)

Buraya kadar kendi metotlarımızı nasıl yazıp kullanılabileceği incelendi. Aynı zamanda kitabın bu bölümüne gelene kadar faydalanılan metotlar var. Örneğin Main() metodu incelendiği zaman; programın başlangıç noktası olma özelliğini bir metot olarak taşıdığı, normal bir metotta olması gereken herşeye sahip olduğu görülür. Başka bir örnek olarak metodların bir sınıf(class) ya da yapı(struct) üyesi olabileceği bilgisinden yola çıkarak; WriteLine() ve ReadLine() metotları göz önüne alınabilir. Console sınıfının üyesi olan bu metotlar ve diğer önceden tanımlı sınıf ve yapılar içerisinde yer alan pek çok metot, .NET platformu için geliştirilmiş olup, programcıların ihtiyacı olan fonksiyonellikleri karşılamak üzere tasarlanmışlardır.

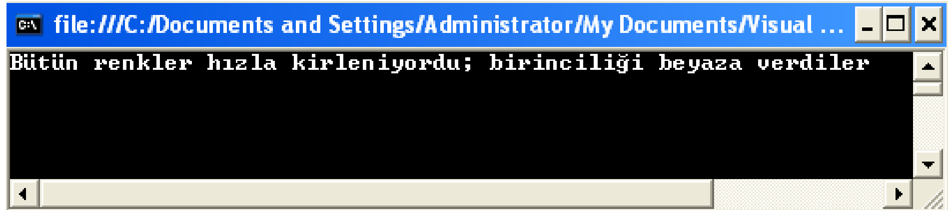
.Net Framework 2.0 içerisinde yaklaşık olarak 75.000' in üzerinde metot vardır.

return Anahtar Kelimesi

Bir metot içerisindeki kodların çalışması varsayılan olarak kapanış süslü parantezine gelindiğinde sonlanır ve kontrol metodun çağırıldığı yere geri döner. Eğer metodun son deyimi beklenmeden sonlanması istenirse **return** anahtar kelimesi kullanılmalıdır.

```
...
static void GununSozunuVer()
{
      Console.WriteLine("Bütün renkler hızla kirleniyordu; birinciliği
beyaza verdiler");
      return;
      Console.WriteLine("Sözün sahibi : Özdemir Asaf");
}

public static void Main(string[] args)
{
      GununSozunuVer();

      Console.ReadLine(); //Uygulama F5 ile çalıştırıldığında konsol
                          //ekranınının hemen kapanmaması için yazılır.
}
...
```

GununSozunuVer, Main() metodu içerisinde çağrılıp çalıştırılmasıyla elde edilen çıktıda; return anahtar kelimesinden sonra ekrana yazdırılmak istenilen **"Sözün sahibi : Özdemir Asaf"** kısmı yer almaz; çünkü return kelimesinin görülmesiyle birlikte metodun çalışması sonlanır.

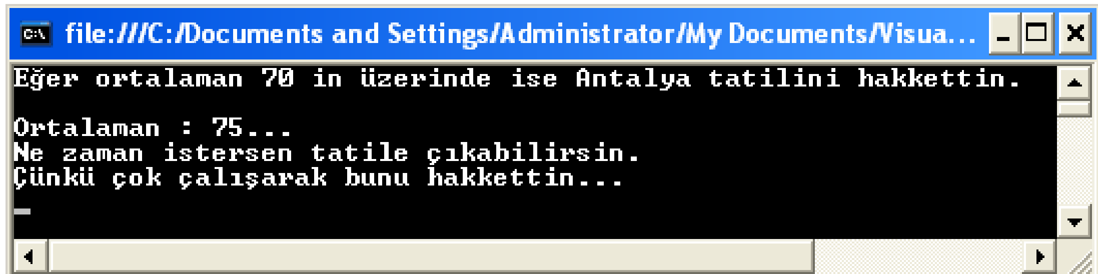


Şekil 107: return anahtar kelimesinin kullanımı

Bu şekilde bir kullanımın çok da anlamı yoktur; çünkü çalışmayacağını bile bile neden return anahtar kelimesinden sonra kod yazalım ki diye düşünülebilir. O yüzden return anahtar kelimesini, if ya da switch gibi koşullu deyimin bir parçası olarak kullanmak daha kullanışlı olacaktır. Yani belli bir koşul sağlanırsa metodun sonlanması beklenmeden çıkılmak istendiğinde **return** anahtar kelimesinden faydalanılabilir.

```
static void OdulVer()
{
      Console.WriteLine("Eğer ortalaman 70 in üzerinde ise Antalya
      tatilini hakkettin.\n");
      int ortalama = 75;
      if (ortalama > 70)
      {
            Console.WriteLine("Ortalaman : {0}...\nNe zaman istersen
            tatile çıkabilirsin.\nÇünkü çok çalışarak bunu hakkettin...",
            ortalama);
            return;
      }

      Console.WriteLine("Ortalaman : {0}...\nMaalesef tatil fırsatını
      kaçırdın.\nÇok çalışmak lazım, çook...",ortalama);
}

public static void Main(string[] args)
{
      OdulVer();

      Console.ReadLine();
}
```

Yukarıdaki kod bloğunda ortalama değişkeninin alacağı değer 70'in üzerinde ise ekrana bir şeyler yazılacak ve ardından return deyimi çalışacağı için metodun çalışması sonlanacaktır. Return deyiminden sonra yazılan kodlar çalışmaz. Eğer ortalama değişkeninin değeri 70 ya da 70'den küçük ise kontrol, return deyiminin bulunduğu if bloğuna girmeyeceği için return'den sonraki kodlar çalışır. Bu kod, ortalama değişkeninin değeri değiştirilerek test edilebilir.



Şekil 108: return anahtar kelimesi ile metodu, kapanış süslü parantezine gelmeden sonlandırmak

return ile Bir Değer Döndürmek

Şu ana kadar öğrenilenler sonucu **return** deyimini metodun sonunda kullanmanın çok da anlamlı olmayacağı söylenebilir; çünkü zaten metodun sonuna gelindiğinde çalışması sonlanır. Ancak metodun çalışması sonucu kendisini çağıran kullanıcıya bir değer göndermesi durumunda, return deyiminin metodun sonunda kullanılması daha anlamlı olur. Metodun çağırıldığı yerde kullanılmak üzere, metot içerisinde yapılan işlerin sonucunda oluşan bir değere ihtiyaç duyulabilir ve bu değeri ele alıp başka işlemlerin yapılması istenebilir. İşte bu değer metodun sonunda return ifadesinden hemen sonra yazarak gönderilebilir. Örneğin metot içerisinde tanımlanan ve değeri verilen iki değişkenin toplamının hesapladığını düşünelim. Yapılan işlemlerin arkasından return deyimi ile sonuc değişkeninin değeri, metodun kullanıldığı yere döndürülebilir. Burada

dikkat edilmesi gereken bir nokta vardır. Hatırlarsanız bu konunun başında **metot söz dizimi** anlatılırken, **donusTipi** kavramından bahsedilmiştir ve şu ana kadar yazılan metotlarda sadece konsol ekranına birşeyler yazdırıldığı için sadece **void** anahtar kelimesi kullanılmıştır. Az önce bahsedilen senaryo için ise dönüş tipi void değil; return ile döndürülecek değere uygun bir veri tipi olmalıdır:

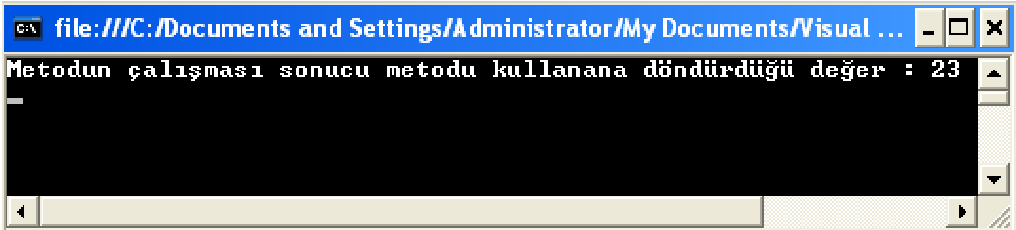
```
static int Hesapla()
{
       int a = 2;
       int b = 21;
       int sonuc = a + b;

       return sonuc;
}
```

Metot, yukarıda belirtildiği gibi iki değişkenin değerini toplayıp üçüncü bir değişkene atar ve kendisini çağıranın bu değeri elde edebilmesi için **sonuc** isimli değişkeni döndürür. Metot dışarıya bir değer döndürdüğü ve **sonuc** isimli değişkenin tipi int olduğu için metodun dönüş değeri artık void olamaz.

Bir metodun dönüş tipini belirlerken değişkenlerde yer alan büyük tür, küçük tür kuralları geçerlidir.

Şimdi de bu metodun Main() içerisinde çağrılıp, dönüş değerinin ele alınışını görelim:

```
public static void Main(string[] args)
{
       int metottanGelenDeger = Hesapla();
       Console.WriteLine("Metodun çalışması sonucu metodu kullanana
       döndürdüğü değer : {0}",metottanGelenDeger);

       Console.ReadLine();
}
```



Şekil 109: return anahtar kelimesi ile dışarıya değer döndürmek

Dönüş değeri **void** iken kullanıldığı şekilde sadece metodu çağırmak tamamen geçerli bir davranış olacakken;

```
public static void Main(string[] args)
{
       Hesapla();

       Console.ReadLine();
}
```

bu şekildeki bir kullanımın çok da bir anlamı olmaz; çünkü metodu yazarken amacımız, metodun çağırıldağı yere yapılan işlemin sonucunu döndürmektir. Sadece metodu çağırmakla, içerde yapılan işlemin sonucunu elde edemeyiz. (En azından şimdilik.) O yüzden yapılması gereken metodun geri dönüş değerinin veri tipi neyse (burada int) o tipte bir değişken oluşturmak ve değişkenin başlangıç değerini metottan almaktır. Yukardaki kodda bu değişken, **metottanGelenDeger** dir. Değişken ilk değer atamasının sağ tarafında bir metot bulunması ilginç gelebilir. Ancak geri dönüş değeri olan bir

metodun kullanıcı için anlamı, döndürdüğü değerdir ve bu değer doğru tipte bir değişken üzerinde taşınabilir. Visual Studio 2005 uygulama geliştirme ortamı da (IDE –Integrated Development Environment-) buna destek verir ve bir metodu kullanmak isteyen kullanıcının o metodu nasıl ele alacağı konusunda yardımcı olmak için metodun geri dönüş değerini gösterir.

```
using System;

namespace Metotlar
{
    class MetotKullanimi
    {
        static int Hesapla()
        {
            int a = 2;
            int b = 21;
            int sonuc = a + b;

            return sonuc;
        }

        public static void Main(string[] args)
        {
            Hesapla();
            int MetotKullanimi.Hesapla()
            Console.ReadLine();
        }

    }
}
```

Şekil 110: .NET uygulama geliştirme ortamının metodun geri dönüş tipini kullanıcıya göstermesi

Metodun çağrıldığı yerde adı yazılırken veya yukarıdaki gibi yazıldıktan sonra imleç ile üzerine gelindiğinde en başta metodun geri dönüş tipini görebiliriz. Buna göre eğer **void** ise sadece metodu çağırırız; eğer geri dönüş tipi herhangi bir veri tipi ise o tipte bir değişken üzerine alarak geri dönüş değerinden faydalanabiliriz. Bir metodun geri döndürebileceği veri tipi önceden tanımlı tiplerden biri olabileceği gibi, kendi yazdığımız bir tip de olabilir.

Parametre Alan Metotlar

Parametreler, metot içerisinde yapılan işlerde kullanılmak için dışardan alınan değerlerdir. Metot oluşturulurken metot adından sonra gelen parantezlerin içerisinde bildirilir. Buraya kadarki örneklerde parametre kullanılmamıştır.

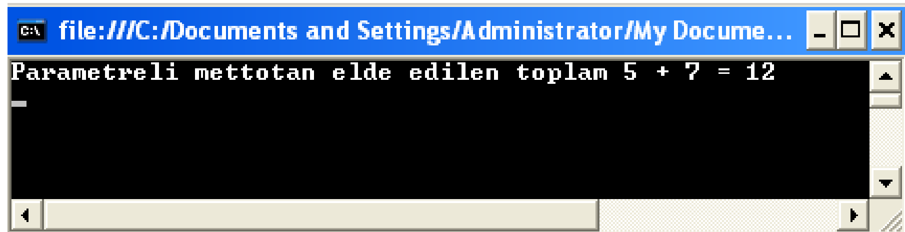
Her parametre, tipi ve değişken adı ile metoda bildirilir. Bu parametreler metot içerisinde lokal değişken gibi kullanılır. (Ancak değer vermek söz konusu olamaz) Metoda birden fazla parametre atanacağı zaman, parametrelerin virgülle ayırarak bildirilmesi gerekir.

```
static int Hesapla(int a,int b)
{
      int sonuc = a + b;
      return sonuc;    //Doğrudan “a + b” de return edilebilirdi.
}
```

Yukarıdaki metot, int tipinde iki tane parametre alır. Bu parametreler, a ve b adlarıyla metot içerisinde istenildiği gibi kullanılabilir. Örnekte bu iki değişken bir aritmetik işleme tabi tutulup başka bir değişkene ilk değer olarak atanmakta ve bu yeni tanımlanan değişken, metodun dönüş değeri olmaktadır. Bu metodun kullanılışı aşağıdadır:

```
public static void Main(string[] args)
{
      int toplam = Hesapla(5, 7);
      //Artık metodun dönüş değeri toplam isimli değişken üzerinde
      Console.WriteLine("Parametreli metottan elde edilen toplam 5 + 7 = "
+ toplam);
}
```

Hesapla() metodu yazılırken parantez içerisinde bildirilen iki tane int tipli değişken, kod bloğu içerisinde sanki varmış gibi kullanılır. Bu yapılırken değerleri ne olursa olsun metodun çalışacağı varsayılır ve toplamları metodun geri dönüş değeri olarak başka bir değişkene atanır. Metot çağrıldığında ise o iki değişkenin alması istenen değerler, metodu çağıran tarafından verilir. Bu değerler, metodun kod bloğu içerisinde a ve b değişkenlerinin kullanıldığı her yere yazılır ve gerekli hesaplamalar yapılıp geri dönüş değeri hesaplanır. Parametrelerin değerleri, metodun her çağrılışında değişik olabilir. Hesapla() metodu da her seferinde kendisine verilen farklı değerlerin toplamını geri döndürür. Bu metodun çıktısı aşağıdadır:



Şekil 111: Parametre alan bir metodun çalıştırılması

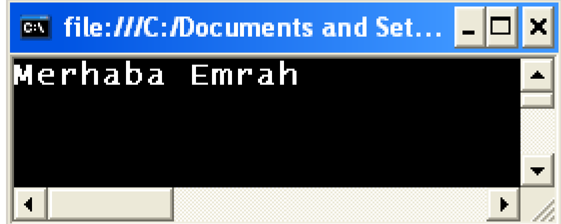
Parametre alan metotların bir dönüş değeri olması zorunlu değildir.

```
static void SelamVer(string ad)
{
      Console.WriteLine("Merhaba {0}",ad);
}
```

Yukarıdaki metot string tipinde bir parametre alır. Kod bloğu içerisinde parametrenin değerinin başına bir yazı getirilerek ekrana yazdırılır. Bu metodun kullanılışı aşağıdaki gibidir:

```
public static void Main(string[] args)
{
      SelamVer("Emrah");

      Console.ReadLine();
}
```

Bu metodun çalışmasıyla elde edilen çıktı aşağıdadır:

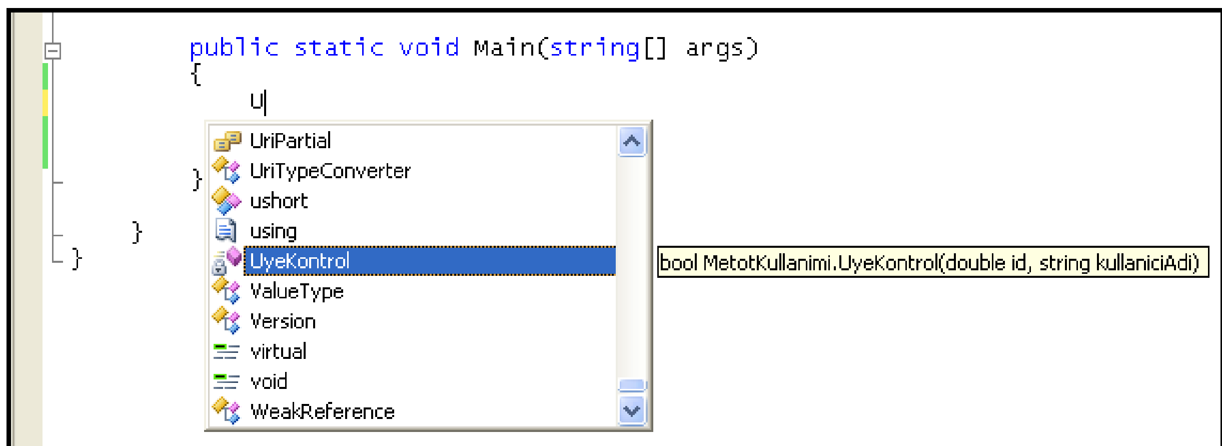


Şekil 112: Dönüş değeri olmayan parametrik bir metot

Aşağıdaki metot, bu konunun son örneğidir.

```
static bool UyeKontrol(double id, string kullaniciAdi)
{
      if ((id == 12345678) && (kullaniciAdi == "emrhsl"))
      {
            return true;
      }


      else
      {
            return false
      }
}
```

Bu metot, biri double diğeri string tipinde iki tane parametre alır. Bu parametreler ile bir kullanıcının web sitemizin kayıtlı bir üyesi olup olmadığının kontrol edildiğini varsayalım. Bu örnek siteye, sadece id'si **12345678** ve kullaniciAdi **emrhsl** olan kullanıcı kabul edilsin ve aşağıdaki şekilde bir metot çağrısı yapılıp kullanılabildiğini düşünelim. Metodu çağırırken metodun kaç tane ve hangi tipte parametre beklediği de intelli-sense özelliği sayesinde görülebilir:



Şekil 113: Metot çağrılışında istenen parametrelerin IDE tarafından gösterilişi

Bu metodun geri dönüş tipi bool'dur. Alınan parametrelerin her ikisi de metot içerisinde kontrol edilen değerler ile aynı ise true; farklı ise false değer dönecektir. Öyleyse metot çağrılıp, parametreleri verildikten sonra geri dönüş değeri bool tipte bir değişken üzerine alınıp kontrol edilmelidir.

```
public static void Main(string[] args)
{
      bool uyeMi = UyeKontrol(87654321, "lshrme");

      if (uyeMi == true)
      {
            Console.WriteLine("Login oldunuz...");
      }
      else
      {
            Console.WriteLine("Id ya da kullanıc adı yanlış");
      }

      Console.ReadLine();
}
```

Metot, parametreleri verilip çağrılır ve geri dönüş değeri bir değişken üzerine alınır. Verilen parametre değerleri metot içerisine yerleştirilip gerekli koşullar kontrol edilir. Koşullar sağlanmadığı için metottan geriye false döner. Main() içerisinde yapılan ise bu değişkenin değerine bağlı olarak kullanıcıya bir mesaj göstermektir.

Metot İçerisinde Kullanılan Değişkenler

Gerek metoda alınan parametreler, gerekse metot içerisinde oluşturulan değişkenler; metot, çalışma zamanında çağrıldığı zaman belleğe alınırlar ve metodun çalışması sona erdiğinden bellekten düşerler. Bu değişkenler sadece tanımlandıkları metot içerisinde kullanılabilirler. Dolayısıyla uygulamanın başka herhangi bir yerinden bu değişkenlere erişmek mümkün değildir. Bu değişkenler, metot her çağrılışında yeniden oluşturulur ve yok edilirler; dolayısıyla bu değişkenler üzerindeki değerler metodun bir sonraki çağrılışında elde edilemezler. Ayrıca metot içerisinde kullanılan bir değişken adı, başka bir metot içerisinde de kullanılabilir; çünkü her bir değişken kendi bloğuna özeldir.

Metotlara Parametre Aktarma Yöntemleri

Değer Yolu ile

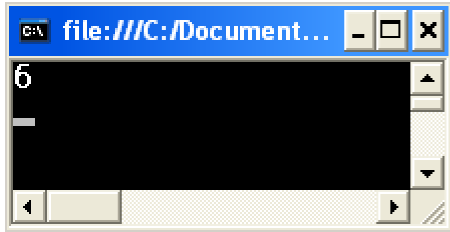
Metot parametreleri metoda aktarılırken varsayılan olarak değer yolu ile geçirilirler. Yani, metot çağrıldığında parametre olarak geçirilen bir değişkenin değerinin kendisi değil kopyasının oluşturulup metoda özel olarak kullanılması ve çalışması bitince de yok edilmesi söz konusudur. Bunu gözlemlemenin en güzel yolu aşağıdaki türde bir metottur.

```
static void BirEkle(int x)
{
      x++;    //x'in değerini 1 arttırır.
}
```

BirEkle() metodu yardımıyla parametre olarak vereceğimiz bir değerin gerçekten kendi değerinin mi artırılacağını yoksa bu artışın sadece metot içerisinde kalıp sonra yok mu edileceği incelenbilir.

```
public static void Main(string[] args)
{
      int k = 6;
      BirEkle(k);
      Console.WriteLine(k);

      Console.ReadLine();
}
```

Bu metodun çalıştırılmasıyla elde edilen çıktı aşağıdadır:



Şekil 114: Parametrenin, metoda değer yolu ile geçirilişi

Main() içerisinde tanımlanan k isimli değişkene başlangıç değeri olarak 6 verilir ve ardından BirEkle() metoduna parametre olarak gönderilir. Bu aşamada k değişkeninin değerinin bir kopyası (x) bellekte metot için oluşturulur. k değişkeninin orjinal değeri ise belleğin ayrı bir bölgesinde hala yerini ve değerini korumaktadır. Metot için oluşturulan bellek bölgesine, k değişkeninden elde edilen 6 değeri kopyalanır ve metot içerisinde

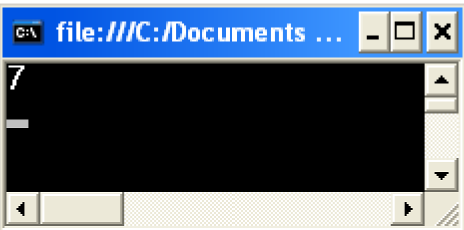
değeri bir arttırılıp 7 yapılır. Metot sonlandığında ise bu değişken bellekten düşer. k değişkeni ise hala ilk tanımlandığı gibi 6 sayısal değerine sahiptir.

Referans Yolu ile

Metot çağrıldığında parametre olarak geçirilen bir değişkenin değerinin kendisi kullanılır. Değer yolundaki gibi bellekte yeni bir alan oluşturulmaz. Bir başka deyişle metoda, ilgili parametrenin bellekteki referansı aktarılmaktadır. Az önceki örnek referans yolu ile parametre geçirmeyi temsil etmek için kullanılabilir. Bir parametreyi metoda referans yolu ile geçirmek için hem metodun yazılışında hem de çağrılışında ilgili parametrenin ve değerin önüne **ref** anahtar kelimesi koyulur.

```
static void BirEkle(ref int x)
{
      x++;    //x'in değerini 1 arttırır.
}

public static void Main(string[] args)
{
      int k = 6;
      BirEkle(ref k);
      Console.WriteLine(k);

      Console.ReadLine();
}
```

Main() içerisinde tanımlanan ve başlangıç değeri 6 olarak verilen k değişkeni, bellekteki adresi ile birlikte metoda parametre olarak geçirilir. Bu, parametrenin başına getirilen **ref** anahtar kelimeleri ile sağlanır. Dolayısıyla kod bloğu içerisinde yapılan artış bu bellek bölgesini etkiler ve metot çağrısının ardından sorgulanan k'nın değeri 7 olarak elde edilir:



Şekil 115: Parametrenin, metoda referans yolu ile geçirilişi

Output Yolu ile

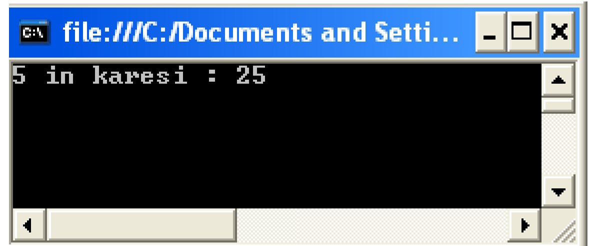
Metoda parametreyi output yoluyla geçirmek, kavramsal olarak referans yolu ile geçirmekle benzerdir. **ref** anahtar kelimesi yerine hem metot yazılışına hem de metot çağrımına ilgili parametrelerin başına **out** anahtar kelimesi getirilmesi yeterlidir. **out** parametresinin tek farkı metoda **parametre olarak geçirilen değişkenin başlangıç değerinin verilmesine gerek olmamasıdır. Aksine ref anahtar kelimesi kullanıldığında ilgili değişken metoda aktarılmadan önce mutlaka ilk değer almalıdır.**

```
static void KaresiniVer(int a,out int sonuc)
{
      sonuc = a * a;
}

public static void Main(string[] args)
{
```

```
        int kok = 5;
        int karesi;
        KaresiniVer(5, out karesi);
        Console.WriteLine("5 in karesi : {0}",karesi);

        Console.ReadLine();
}
```

Burada yazılan KaresiniVer() metoduna iki tane parametre kabul edilir. Biri karesi alınacak değer, diğeri ise hesaplanan karenin değerini dışarıya (metodu çağırana) göndermek için kullanılacak out parametresidir. Kod bloğunda **sonuc** isimli out parametresine **a** parametresinin karesi hesaplanıp atanır ve bırakılır. Dikkat edilirse metot geriye hiçbir değer döndürmez gibi görünse de aslında **out** parametresi sayesinde içerde hesaplanan bu parametrenin değeri dışarıya çıkarılabilir.

Bu bilgiler ışığında Main() metodu içerisinde KaresiniVer() metodunun çağrılmasıyla 5 değeri verilen **kok** değişkeni ile başlangıç değeri verilmeyen ve out olarak işaretlenmiş **sonuc** değişkeni bu metoda parametre olarak geçirilir. Burada gözlemlenmesi gereken, metoda parametre olarak geçirilirken out parametresinin bellekte yeri ayrılı olmasına rağmen henüz değerinin olmamasıdır. Metot içerisinde bu bellek alanına 5 değerinin karesi 25 atanır ve bu değerin metot dışında da erişilebilir olduğu ekrana yazdırılıp görülür.



Şekil 116: Parametrenin, metoda out yolu ile geçirilişi

Referans(ref) ve Output(out) yoluyla metotlara değer aktarma sayesinde, bir metottan geriye birden fazla değer döndürülebilir.

params Anahtar Kelimesinin Kullanımı

Bazı durumlarda metot yazılırken metodu çağıranın kaç tane parametre geçireceği bilinmez. Şu ana kadar öğrendiklerimiz çerçevesinde bunu halletmenin bir yolu yoktur. Ancak metotlara herhangi bir tipten bir dizi aktarılması halinde bu sorun kısmen de olsa aşılabilir. Bu iş için C#, metoda parametre olarak bir dizi alınmasını ve dizinin önüne **params** anahtar kelimesinin getilmesinin yeterli olacağını söyler:

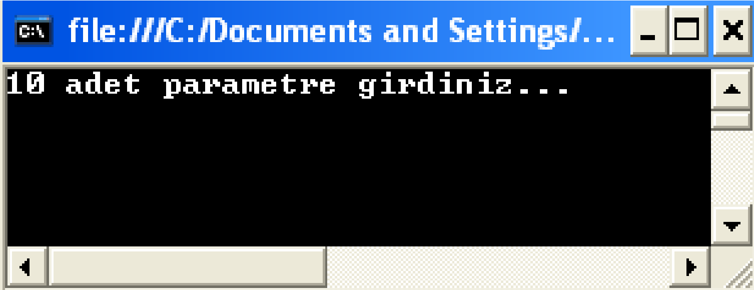
```
static void ParametreSayisiniVer(params int[] dizim)
{
        Console.WriteLine("{0} adet parametre girdiniz...",dizim.Length);
}
```

Yukarıdaki metodun çağırıldığı yerde, int tipinde istenildiği sayıda parametre aktarımı gerçekleştirilebilir. Parametreler verilip uygulama çalıştırıldığında C#, parametre değerlerinden oluşan **dizim** adındaki diziyi oluşturur. Bu sayede metot içerisinde bu dizi ele alınabilir. **ParametreSayisiniVer()** metodu içerisinde girilen parametrelerin sayısı kullanıcıya söylenir.

```
public static void Main(string[] args)
{
        ParametreSayisiniVer (3, 4, 5, 6, 2, 5, 45, 4, 3, 2);

        Console.ReadLine();
```

```
}
```

Bu metodun çağrılmasıyla elde edilen çıktı aşağıdadır:



Şekil 117: Parametreler, metoda geçirilirken params anahtar kelimesinin Kullanılışı

Siz de params anahtar kelimesi kullanarak kullanıcının belirlediği sayıda tam sayı tipinde parametre alıp bunların toplamlarını ve ortalamalarını bulan bir metot yazmaya çalışabilirsiniz.

BÖLÜM 7: PROGRAMLAMA YAKLAŞIMLARI ve SINIF KAVRAMINA GİRİŞ

Programlamaya Prosedürel Yaklaşım

Problemlerin çözülmesi için programlar yazılır, uygulamalar geliştirilir. Programlamaya **prosedürel yaklaşım**da bir problem küçük parçalara bölünür ve her bir problem ayrı ayrı çözülür. Daha sonra bütün problemi çözmek için bu küçük çözümler birleştirilir.

Her bir küçük problem, bir görev olarak ele alınır. Bu görevi gerçekleştirmek için **prosedür** adı verilen kod blokları yazılır. Bir prosedür başka bir prosedürden erişilebilir. Prosedürler birleştirilip bir **program** oluşturulabilir. Daha sonra bütün programlar birbirlerine entegre edilip yazılım uygulaması ortaya çıkar.

Tipik bir prosedürel program bir satır kod ile başlar. Ardından kontrol deyimleri, operatörler, iş mantığının uygulanması ile ilerlenir ve son satıra gelindiğinde program sonlanır.

Fortran ve C, prosedürel yaklaşımı esas alan programlama dilleridir. Aşağıda bir çalışanın mesleği tabanlı prim hesaplaması yapan sözde kod (pseudocode) yer almaktadır.

```
begin
character cAdi
character cMeslegi
numeric nPrim
Procedure PrimHesapla
begin
if cMeslegi = “Mudur”
begin
nPrim = 1000
end
if cMeslegi = “Analist”
begin
nPrim = 500
end
display nPrim
end
accept cAdi
accept cMeslegi
Call PrimHesapla
End
```

Burada veriler çalışanın adı, mesleği ve prim miktarlarıdır. Çalışanın primini hesaplayan iş mantığını **PrimHesapla** prosedürü içerir. Program satır satır çalışır ve prim miktarı hesaplanır.

Prosedürel Programlama Yaklaşımında Sınırlamalar

Prosedürel yaklaşımda prosedürler, bir uygulamadan diğerine yeniden kullanılamaz. Çünkü kod parçaları birbirlerine bağlıdır ve belli bir seriyi takip ederler. Örneğin PrimHesapla() prosedürü başka bir uygulamada kullanılmak istenirse, ikinci uygulamada yeniden yazılması gerekmektedir.

Prosedürel yaklaşım kullanılarak geliştirilen büyük bir uygulama kolayca modifiye edilemez. Yazılan bütün kodlar birbirlerine sıkı sıkı bağlıdır ve bir yerde yapılan değişikliğin, uygulamanın bir çok yerine etkilerini de ele almak gerekir.

Oysa bir yazılım uygulamasının dinamik ihtiyaçlara cevap verebilmesi, değişiklikleri sessizce yansıtabilmesi gerekmektedir. Prosedürel yaklaşımda bu tarz değişiklikler yüklü

miktarda yeniden kod yazmayı gerektirir. Bu da uygulamanın bakım maliyetlerini (hem zaman, hem ekonomik olarak) arttırır.

Programlamaya prosedürel yaklaşımın getirdiği yukarıda bahsedilen sınırlamalardan dolayı, yazılım geliştirme mühendisleri daha iyi bir yol aramaya başlamışlardır.

Programlamaya Nesne Yönelimli Yaklaşım (Object Oriented Approach)

Nesne yönelimli yaklaşım, programcılara daha dengeli yazılım uygulamaları geliştirmeleri için yardım eder; çünkü çok çeşitli boyut ve karmaşıklıktaki problemleri kolayca çözme yolları sunar. Daha hızlı ve daha etkili kod yazılmasını sağlar. Nesne yönelimli olmayan bir dille, yüzbin satırlık kodu yönetmek kabusa dönüşebilecekken; bu kodun nesne yönelimli olarak yazıldığında yönetilmesi çok daha kolay olur; çünkü kendi içerisinde yönetilmesi kolay onlarca nesneye bölünecektir. En önemli nokta; nesneye yönelik programlama, tekrar kullanılabilir kodlar yazılmasını sağlar. Nesneye yönelimin altındaki temel düşünce budur.

Prosedürel yaklaşımda örnek verilen programı yeniden ele alalım. Nesne yönelimli yaklaşım kullanıldığında **çalışanın** her bir karakteristiği (adı,mesleği gibi) tek bir kod bloğunda tanımlanır. Sonra çalışan tanımlamasından başka bir kod bloğunda **PrimHesapla()** metodunun iş mantığı kodlanır. Bu şekilde veriler (çalışan bilgileri) ile iş mantığı (prim hesaplaması) tamamen birbirinden ayrı olarak ele alınmış olur.

Burada mantık, geliştirilen programı mümkün olduğu kadar küçük parçalara bölerek, her parçayı bir türde **nesne** haline getirmek ve daha sonra bu nesneleri gerekli yerlerde çağırarak kullanmaktır. Nesneleri yazmak için önce bir **şablon** (kalıp) oluşturulur. Daha sonra bu şablondan istenilen sayıda nesne çıkarılır. Programlama dilinde bu şablonlara **class**, nesnelere de **object** adı verilir.

Avantajları

İstenilen şablon ve dolayısıyla metotlar, bir uygulamadan başka bir uygulamaya taşınıp yeniden kullanılabilirler. Örneğin çalışanların mesleklerine göre yaz tatil planlamalarını ele alan bir uygulamada, yukarıdaki programda yer alan **çalışan** tanımlaması yeniden kullanılabilir. Çalışan bilgilerini kullanarak yaz tatil planlamalarının iş mantığını içeren yeni bir metot oluşturulabilir.

Çok büyük çaplı uygulamalar dahi kolaylıkla güncellenebilir; çünkü veri ve iş mantığı hem aralarında hem de kendi içlerinde ayrı ve birbirlerinden bağımsız olarak ele alınırlar. Örneğin çalışanların mesleklerini tanımlayan kod bloğuna dokunmadan prim miktarını belirleyen kodda değişiklik yapılabilir.

Günümüzün popüler programlama dilleri ve işletim sistemleri nesne yönelimli programlamayı kullanır ve desteklerler. Programlama dillerine örnek olarak C++, Java ve burada öğrenilen C# verilebilir.

Gelişen teknolojiler, çözümlenen gerçek hayat vakaları yeni mimarilerin oluşmasınada ön ayak olmaktadır. Öyleki günümüzde sadece nesne yönelimli değil, bileşen yönelimli yaklaşım (Component Oriented Approach) ve çok daha popüler hale gelmiş servis yönelimli yaklaşım (Service Oriented Approach) modelleri de vardır.

.NET’de Sınıf(Class) ve Nesne(Object) Kavramları

Nesne yönelimli bir uygulamada sınıf (class) ve nesne (object) kavramları bazen karışıklığa neden olabilir; dolayısıyla bunun çizgilerini kesin olarak koymak gerekir. Bir sınıf, nesnenin özelliklerini ve davranışlarını tanımlar. Nesne ise sınıfın aktif bir örneğidir.

Sınıf

Sınıf (class), veriler içeren (değişkenler ile) ve hem bu verileri işleyen hem de iş mantığı fonksiyonelliğini sağlayan (metotlar ve başka üyeler ile) veri tipidir.

Sınıflar, gerçek hayat objelerini programlama ortamına aktarmak için kullanılan proje taslağı ya da tasarısı olarak düşünülebilir (Aynı zamanda şema,şablon da denebilir). Bu taslak, verinin nasıl saklanacağını, işleneceğini ve erişilebileceğini tanımlar. Sınıf örneği olarak bir **araba** gösterilebilir. Arabanın rengi, kapı sayısı, modeli gibi özellikleri vardır. Ayrıca bir arabada kapıları kilitlemek, kapıları açmak, pencereleri açmak gibi işlemler (davranışlar) gerçekleştirilebilir. Burada arabanın özellikleri uygun tiplerde değişkenler ile; davranışları ise metotlar ile temsil edilebilir.

Nesne

Nesneler, sınıfların gerçek hayata geçirilmiş örnekleridir. Gerçek hayattan kasıt aslında programın çalıştığı ortamdır. Tek başına taslak (sınıf), bir ya da birden fazla nesnenin oluşturulması için kaynak olarak kullanılır.

Bir sınıf, kod yazma aşamasında tanımlanır ve her zaman var olur; ancak nesne çalışma zamanında oluşturulur ve uygulama çalıştığı sürece kullanılabilir. Her ihtiyaç olduğunda istenilen sayıda oluşturulabilir.

Örnek olarak bir bina mimarisini ele alalım. Programlama ortamında binanın taslağı sınıf olarak düşünülürse; gerçek hayatta bu taslağa bakılarak binanın inşaatının yapılması ise nesneyi oluşturmaktır. Söz konusu taslağa bakılarak istenilen sayıda bina yapılabilir; her bir bina birbirinden farklı özelliklere sahip olabilir. **Yani bir sınıftan istenilen sayıda nesne oluşturulup kullanılabilir; her nesnenin özellikleri ve davranışı farklı olabilir.**

C#'da Sınıf Nasıl Tanımlanır ve Nesne Nasıl Oluşturulur?

Bir sınıf tanımlanırken aşağıdaki kurallar geçerlidir.

- Bir sınıf tanımlamak için sınıf adından önce **class** anahtar kelimesi kullanılır.
- Kodlar C# dilindeki diğer birçok programlama öğesinde olduğu gibi süslü parantezler arasına yazılır.
- Sınıf adı isimlendirme kuralları, metot isimlendirme kuralları ile aynıdır.
- Sınıflar genellikle bir isim alanı altına yazılırlar; ancak başka bir sınıf ya da yapının içerisine yazıldığı durumlar da olabilir. (Dahili tip – Inner Type)

Aşağıdaki sınıf bir alan(field) ve bir metottan oluşmaktadır:

```
class Daire
{
      public double AlanHesapla()
      {
            return 3.14 * yaricap * yaricap;
      }

      public double yaricap;
}
```

Sınıf kod bloğu içerisinde (daha önce öğrenilen) bir metot (AlanHesapla) ve (yine daha önce öğrenilen) bir alan (yaricap) yer almaktadır. Dikkat edilirse burada yeni bir söz dizimiyle karşılaşılmamıştır. Şu ana kadar geliştirilen uygulamaların Main() metotları zaten bir sınıf içerisinde yer almaktadır.

Daire sınıfını kullanmak, daha önce karşılaşılan veri tiplerini kullanmakla tamamen benzerdir. Sınıflar referans tipli olduğu kullanıcı tanımlı bu yeni veri tipi, new anahtar kelimesi ile oluşturulur.

```
Daire d1 = new Daire();
```

Main() metodu içerisinde önce Daire tipinde bir değişken tanımlanır (d1); ardından da **new** operatörü ile nesne oluşturulur ve aynı zamanda bu nesne içerisindeki verilerin geçerli başlangıç değerleri atanır (Bu örnekte yaricap alanının(field) başlangıç değeri 0.0 olur; çünkü veri tipi double'dır.). d1 değişkeni, bellekteki isimsiz Daire nesnesi için bir referanstır, bu nesnenin bellekteki adresini tutar ve kendine yapılan her başvuruda bu nesneyi ve üyelerini işaret eder.

d1 değişkeni belleğin stack bölgesinde, onun işaret ettiği isimsiz Daire nesnesi ise belleğin heap bölgesinde yer alır **(Bakınız: Referans Tiplerini Anlamak).**

Yazılan Daire sınıfı, hayata geçirilecek daire nesneleri için bir şablon niteliğindedir; bu şablon tektir. Daire sınıfından **new** anahtar kelimesi ile istenilen sayıda nesne oluşturulup her seferinde yaricap üyesi farklı verilerek (varsayılan olarak 0.0 değerini alır), farklı boyutlarda daire nesneleri elde edilebilir. Bu da nesne yönelimli uygulama geliştirmenin gücüdür.

```
using System;

namespace Siniflar
{

    class Daire
    {
        public double AlanHesapla()
        {
            return 3.14 * yaricap* yaricap;
        }

        public double yaricap;
    }

    class SinifKavrami
    {
        public static void Main(string[] args)
        {
            Daire d1 = new Daire();
            d1.yaricap = 8;

            Daire d2 = new Daire();
            d2.yaricap = 5;


            Console.ReadLine();
        }
    }
}
```

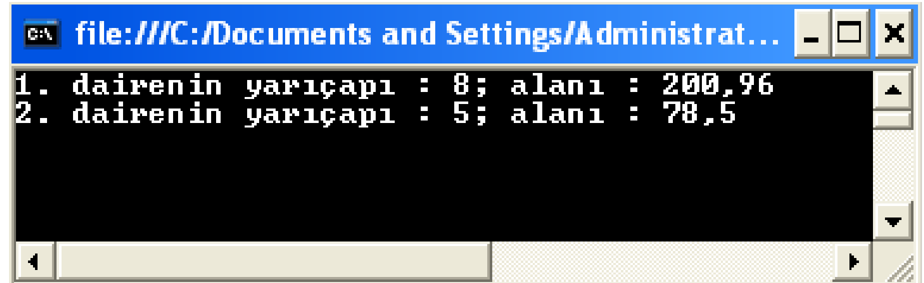
Şekil 118: Sınıf ve Nesne Kullanımı

Sınıf içerisinde hem alana hem de metoda erişim belirleyicisi olarak public verilmesinin sebebi, bu üyeleri sınıf dışarısından erişime açmaktır. **yaricap** alanının(field) erişim belirleyicisi **public** değil de **private** yapılsaydı (ya da hiç yazılmasaydı varsayılan olarak yine private değerini alırdı), Main() içerisinde Daire sınıfından bir nesne örneği oluşturulduktan sonra referans değişkeni d1 üzerinden 'yaricap' değişkenine erişilemezdi.
Bu durumda şöyle bir soru akla gelebilir: Erişim belirleyicisi **public** olmayan ve dolayısıyla dışarıya kullanıma açılmayan bir üye sınıf içerisinde neden kullanılır? Böyle bir sorunun cevabı şu olabilir: Erişim belirleyicisi **private** olan bir üye sadece sınıf içerisinden erişilebilirdir; dolayısıyla sınıf içerisinde gerçekleştirilen herhangi bir iş mantığında yardımcı üye olarak kullanılabilir.

Daire nesne örneklerinin biri 8, diğeri 5 birim yarıçapındadır. Dolayısıyla elimizde aynı şablondan çıkmış 2 farklı daire nesne örneği bulunmaktadır. Yine aynı şablondan her iki nesne örneği için de aynı metot çağrılır ve her nesnenin davranışının farklılık gösterdiği kanıtlanır:

```
public static void Main(string[] args)
{
      Daire d1 = new Daire();
      d1.yaricap = 8;
      double alan1 = d1.AlanHesapla();
      Console.WriteLine("1. dairenin yarıçapı : {0}; alanı : {1}", 
d1.yaricap, alan1);

      Daire d2 = new Daire();
      d2.yaricap = 5;
      double alan2 = d2.AlanHesapla();
      Console.WriteLine("2. dairenin yarıçapı : {0}; alanı : {1}", 
d2.yaricap, alan2);
      Console.ReadLine();  //Ekranı dondurur.
}
```

İki ayrı nesne örneği için iki ayrı **AlanHesapla()** metodu çalıştırıldığında ikisinin de sonuçları farklı çıkar; çünkü dairelerin yarıçapları farklıdır. AlanHesapla() metotu tip döndüren bir metot olduğu için, çağrıldığında double tipli bir değişken üzerinde ele alınabilir. Bu kodun çalışmasıyla elde edilen çıktı aşağıdadır:



Şekil 119: Daire sınıfından oluşturulan nesne örnekleri üzerinden metot çağrımı.

Burada Daire sınıfı, Main() metodu içerisinde bulunan sınıfın isim alanına(namespace) yazılır. Tasarlanan bir sınıfın, başka uygulamalarda kullanılması ya da başkalarıyla paylaşılması isteniyorsa yapılması gereken şey; (Visual Studio 2005 ile çalışılıyorsa) **File → New → Project → Class Library** sekmesi takip edilir. Açılan sınıf tanımlamasına istenilen sınıfın kodları yazılır; ardından derleme işlemi ile sadece kütüphane kodları içeren bir assembly(*.dll) oluşur. Bu assembly içerisinde bilindiği gibi **CIL** komutları yer alır ve başka .Net tabanlı uygulamalarda kullanılabilir. Bir sınıf kütüphanesini (*.dll) kullanacak uygulamada **Solution Explorer**'da **Reference**'a sağ tıklanarak **Add Reference...** sekmesinden ilgili sınıf kütüphanesinin (.dll) yeri işaret edilmelidir. Bu işlemin ardından referans edilen assembly ve içersindeki tipler, sahip oldukları erişim belirleyecilerinin verdiği yetkiler dahilinde ilgili uygulama içerisinde kullanılabilirler.

KISIM SONU SORULARI:

1) Lokal değişkenler, başlangıç değerleri verilmeden kullanılamazlar. Ancak bu kuralın bir tane istisnası vardır. Bu istisna nedir?

2) .NET tabanlı dillerde tip tanımlarken, değişken kullanırken ve yönetilirken uyulması gereken ortak kurallar listesi vardır. Eğer bir uygulama bu kurallar takip edilerek geliştirilirse, geliştirildiği dil dışındaki diğer dillerde de sorunsuz şekilde kullanılacağı garanti edilmiş olur? Bu ortak kurallar listesinin adı nedir?

3) Ortak tip sisteminin, tipleri değer ve referans tipli olarak iki grup altında toplamasının sebebi ne olabilir?

4) Bir öğretmensiniz ve n tane öğrenciniz var. n öğrencinizin ilk sınavdan aldıkları notları bir programa input olarak alıp ortalamalarını hesaplayan bir uygulama yazınız.
Bunu yaparken şu süreci takip ediniz:

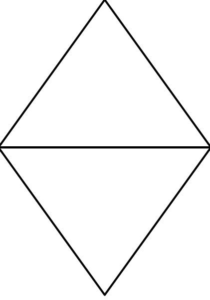
- Program başında kaç öğrencinin notunun hesaplanacağı kullanıcıdan alınmalıdır.
- Öğrenci sayısı kadar not kullanıcıdan alınmalıdır (Bu işlem için dizi kullanılabilir).
- Son olarak, genel ortalama bulunmalıdır.

5) Sınıf (class) ve yapı (struct) hemen hemen benzer amaçlara hizmet eden, yetenekleri birbirine benzer iki .NET tipidir. Peki aralarındaki farklar nelerdir? Araştırınız...

6) Bir dizinin elemanlarını tersten yazan kodları yazınız.

7) Elemanları rastgele üretilen iki matrisin toplamını bulan örnek bir konsol uygulaması yazınız.

8) Ekranda "*" karakterlerini kullanarak aşağıdaki şekli oluşturan örnek uygulamayı yazınız.



9) Cast işlemi ile Convert ya da Parse işlemi arasındaki farklar nelerdir? Araştırınız.

10) Bir öğrencinin birinci sınav (vize1), ikinci sınav (vize2) ve final notlarını kullanıcıdan alan, finalden 50' nin altında not alması durumunda KALDI mesajı veren ve harf olarak da F yazan, eğer finalden 50'nin üzerinde not almışsa; ortalamayı, Final'in %60'ını vizelerin % 20'lerini alarak hesaplayan, eğer ortalama 50'nin üzerinde ise; GEÇTİ mesajı veren ve

50-60 : E
60-70 : D
70-80 : C
80-90 : B
90-100 : A

ortalamayı yukarıdaki aralıklara göre isimlendiren örnek uygulamayı yazınız.

KISIM III: SQL SERVER 2005 İLE VERİTABANI PROGRAMCILIĞINA GİRİŞ

YAZAR: UĞUR UMUTLUOĞLU

BÖLÜM 0: VERİ TABANINA GİRİŞ

Veri Tabanı Nedir?

Veri tabanı (**Database**) kavramına girmeden önce veriyi kısaca tanımlamak önemlidir. **Veri**, (**data**) bir veya birden fazla bilgiden oluşan bir kümedir. İsim, yaş, telefon numarası, bir toplama işleminin sonucu ya da bir sınıfın yaş ortalaması birer veridir. Bir veri tabanı yapısı içerisinde tutulan bilgilere veri denilmektedir.

Veri tabanı temel olarak farklı tipteki verileri saklamamızı ve kullanmamızı sağlayan depolama ortamıdır. Bu ortam içerisinde verileri saklayabilir, onlara kolay bir şekilde ulaşabilir ve gerektiğinde bu verilerin üzerinde değişiklikler yapabiliriz. Veri tabanı veriler arasında bütünlük, düzen, hızlı erişim, bakım kolaylığı ve benzeri fonksiyonelliklerin sağlanmasında önemli bir rol üstlenir.

Bakkal ve marketlerde veresiye satış yapıldığında müşterinin aldığı ürünlerin adı, miktarı, fiyatı gibi bilgiler bir veresiye defteri içerisinde müşteriye ait bir sayfada saklanır. Bu şekilde bakkalın sahibi, gerektiğinde ilgili müşterinin sayfasını açıp hangi ürünlerden ne kadar miktarda aldığını ve borç toplamının ne kadar olduğunu hesaplayabilir. Fakat bu durum birçok açıdan kötü sonuçlara yol açabilir. Verilerin defterde tutulması düzensizliklere ve karışıklıklara yol açabilir. Yine kâğıttan yapılmış bir defterin başına yırtılması, ıslanması, kaybolması gibi bakkal sahibi açısından tüm hesaplarının kaybolmasına dahi sebep olabilecek bir takım kazaların gelmesi ciddi bir risktir. O zaman gelin biz bu defterdeki verileri bilgisayar ortamına aktaralım. Bilgisayarımızda veresiye adında bir klasör oluşturup içerisine Notepad gibi bir metin editörü ile her müşterimiz için bir metin dosyası açalım ve dosyamızın adına daha sonra kolay bir şekilde ulaşabilmek için bir numara verelim. (Örneğin Ahmet isimli müşterimiz için 302 numarasını verelim) Bu metin dosyası içerisinde defterde tuttuğumuz verileri daha düzenli bir şekilde tutabiliriz. Yine bu bilgileri düzenli bir şekilde kaydederek, yedekleyerek başımıza gelebilecek veri kayıplarını en aza indirebiliriz.

İkinci senaryomuz, yani verileri bilgisayarda saklamamız ilk senaryodaki veresiye defterine göre elbette ki daha esnek, hızlı ve sağlıklı bir yol olacaktır. Fakat bir metin dosyası üzerinde girilen verilere müdahale etme şansımız pek olmayacaktır. Örneğin ücret bilgisinin yazılacağı alana *40*, *40,5* veya *kırk* gibi farklı tipte veriler girilebilir ya da ücret kısmı boş bırakılabilir. Bu şekilde kaydedilen bir dosya daha sonra bazı karışıklıklara yol açabilecektir. Peki, bu bilgiler üzerinde yeni veri eklemek, istediğimiz kıstaslara göre veri seçmek, var olan veriyi değiştirmek veya veriler üzerinde işlemler yapmak istediğimizde çok iyi sonuçlara ulaşabilir miyiz? Elbette ki bu tip işlemleri metin dosyası üzerinde yapmamız biraz olsun işimizi kolaylaştıracaktır. Fakat *"Daha iyi bir yol olabilir mi?"* diye düşünecek olursak cevabı kesinlikle *"Evet"* olacaktır.

Veri tabanı, verilerimizi düzenli bir şekilde depolamamızı sağlar. Depolanan verilere erişimi, veriler üzerinde işlemler yapmayı çok hızlı, sağlıklı ve verimli bir hale getirir. Bir veri tabanı sistemi üzerinde istediğimiz verileri depolayabilir, depolanan verileri çağırırken bazı şartlar belirtebilir, verileri değiştirebilir ve silebiliriz. İlerleyen konularda öğreneceğimiz T-SQL dili ile aşağıdaki örnekleri çok basit bir şekilde gerçekleştirebileceğiz:

- Ahmet Demir isimli müşterinin borç bilgisine 1,5 YTL fiyatıyla 2 adet margarin ekle.
- Ahmet Demir isimli müşterinin satın aldığı tüm ürünlerin listesini getir.
- Fatma Altuntaş isimli müşterinin 1 Ocak 2007 ile 10 Şubat 2007 tarihleri arasında satın aldığı tüm ürünleri getir.
- Murat Şahin isimli müşterinin toplam borç bilgisini getir.
- Tüm müşterilerin toplam borç bilgisini getir.

Bu örnekleri genişletmemiz ve arttırmamız elbette mümkün. Fakat şimdilik bu örnekler bir veri tabanı sistemi üzerinde ne gibi işlemleri yapabileceğimiz açısından temel örnekleri teşkil edecektir.

Günümüzde veritabanları hemen hemen her alanda sıklıkla kullanılmaktadır. Bu alanlara örnek verecek olursak;

- Kişisel adres defterleri
- Kütüphane sistemleri
- Ödeme ve borç sistemleri
- Ürün satış ve sipariş sistemleri
- Banka sistemleri
- Okul sistemleri
- Hastane sistemleri

gibi birçok alanda gerekli bilgiler veritabanlarında tutulmaktadır. Örneğin bir kütüphaneden ödünç aldığımız kitaplarla ilgili olarak, kitap adı, alış tarihi, geri veriş tarihi gibi bilgiler veri tabanındaki bir tabloda tutulmaktadır. Kütüphanedeki görevli istediği zaman kayıtlı bir kullanıcı ile ilgili bilgilere ulaşabilmektedir.

Veri Tabanı Yönetim Sistemleri

Veri Tabanı Yönetim Sistemleri (**DataBase Management System - DBMS**), veri tabanında tutulacak olan verilerin uyacağı standartları, bu verilere nasıl erişilebileceğini ve verilerin disk üzerinde nasıl tutulacağını belirleyen sistemlerdir. Bu sistemler aracılığıyla verilerin bütünlüğü ve güvenliği sağlanmaktadır. Programların veya sistemlerin veri tabanı içerisindeki verilere kolay ve hızlı bir şekilde erişebilmesini ve verileri yedekleyebilmesini sağlamaktadır. Bu sistemler, birden fazla veri tabanı üzerinde işlemler yapabilecek şekilde tasarlanmıştır. Yine olabilecek felaket senaryolarına (disaster case) karşısında, veri tabanının sorunsuzca işleyebilmesi için tedbirler almaktadır.

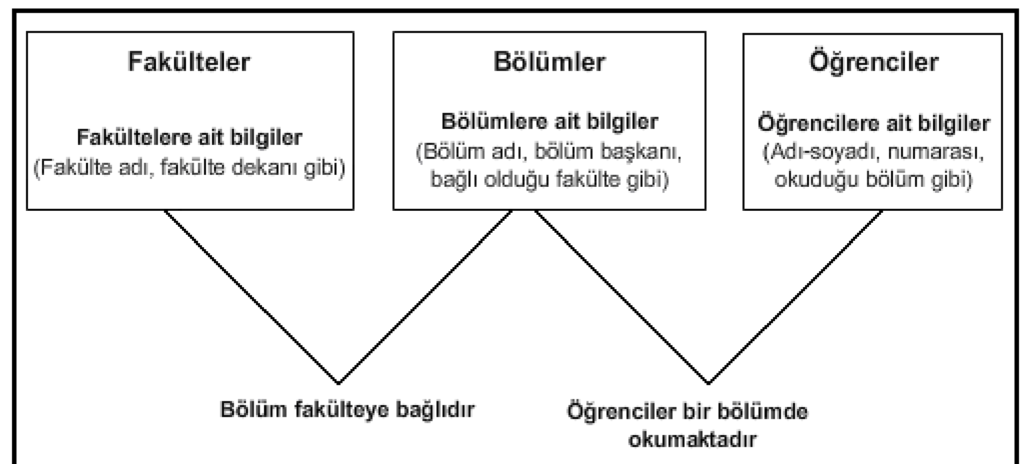
İlişkisel Veri Tabanı Yönetim Sistemleri

Gelişen bilgi teknolojileri ile birlikte veri tabanı sistemlerinde de birçok yenilikler olmuştur. İlk veri tabanı sistemlerinde veriler gelişigüzel bir şekilde depolandığı için veriler üzerinde değişiklikler yapmak zor bir işlemdi. Kullanıcıların veriler üzerinde değişiklik yapabilmesi için veri tabanının yapısını iyi bilmesi gerekiyordu. Yine bu sistemler tam olarak veri bütünlüğünü ve güvenliğini sağlayamıyordu. Günümüzde en yaygın olarak kullanılan veri tabanı mimarisi olan ilişkisel veritabanlarında ise veriler tablolar, satırlar ve alanlar halinde tutulmaktadır. Bu sayede veriler arasında ilişkilendirmeler yapılabilmekte, verilerin kullanımı ve yönetimi daha verimli hale getirilmektedir. İlişkisel veri tabanı mimarisini kullanan sistemlere ise **İlişkisel Veri Tabanı Yönetim Sistemi (Relational Database Management System - RDBMS)** adı verilmektedir. Günümüzde en sık kullanılan ilişkisel veri tabanı yönetim sistemleri arasında SQL Server 2005, SQL Server 2000, Oracle ve Sybase SQL Server'ı gösterebiliriz.

Bir veri tabanını doğru bir biçimde tasarlayabilmek için varlıklar arasındaki ilişkileri (entity relationship) iyi şekilde kavramak gerekmektedir. Verileri gerçek hayat ile ilişkilendirip, aralarındaki bağlantıları oluşturup, veri tabanının mantıksal şablonu çıkarılmalıdır. SQL Server 2005 gibi bir RDBMS ile hazırlanan veri tabanında yer alacak veriler, tablolar ve alanlar (sütunlar) içerisinde tutulur. Tablolara kaydedilecek olan verilerin kendilerine ait uygun veri tipleri belirlenip, girilecek olan değerler için bazı sınırlandırmalar getirilebilir. Yine veriler arasında ilişkilendirmeler yapıp, veri tabanına ilişkisel bir yapı kazandırılabilir. **SQL (Structered Query Language**) dili ile sorgular oluşturarak RDBMS üzerinden veri tabanı ile iletişim kurulabilir.

Veri tabanı tasarlanırken, verilerin gerçek dünyada aralarındaki ilişkileri göz önüne aldığımızda, verileri ilişkilendirmemiz daha da kolaylaşır. Örneğin bir üniversite için temel olarak fakülteleri, bölümleri ve öğrencileri ele aldığımızda *bölüm ile fakülte* arasında ve *öğrenci ile bölüm* arasında bazı ilişkiler olacaktır. Alt birimden üst birime doğru bir sıralama yaptığımızda *öğrenci-bölüm-fakülte* gibi bir ilişkilendirme ortaya çıkmaktadır.

Böyle bir senaryoda, veri tabanı tasarlanırken öğrenci, bölüm ve fakülte arasındaki ilişkiler ele alınarak tablolar tasarlanabilir ve gerekli ilişkilendirmeler yapılabilir. Aşağıdaki şekilde öğrenci, bölüm ve fakültenin işlevleri ve aralarındaki ilişkilerin şekle dökülmüş hali bulunmaktadır.



Şekil 120: Fakülte, bölüm ve öğrencilerin şekil üzerinde tanımlanması ve ilişkilendirilmesi

İlişkisel bir veri tabanında üç temel eleman bulunmaktadır.

- **Tablolar (Tables)**
- **Anahtarlar (Keys)**
- **İndeksler (Indexes)**

Tablolar (Tables)

Tablolar verilerin saklanmasını sağlayan, **alanlar (sütun-column)** ve **satırlardan (row)** oluşan birimlerdir. Tablo, ilişkisel veritabanlarında temel veri depolama nesnesidir ve bilgiler tablolarda saklanmaktadır.

Tablo içerisinde her satır bir kaydı temsil etmektedir. Alanlar ise kayıtlara ait özellikleri taşımaktadırlar. Örneğin bir öğrenciye ait kayıtta okul numarası, öğrenci adı ve öğrenci soyadı bilgilerini temsil eden "128445", "Yakup", "Şeref" gibi bilgiler tutulabilmektedir. Burada öğrenci numarası, öğrenci adı ve soyadı birer alanı, bu üç verinin tamamı ise bir satırı, yani bir kaydı oluşturmaktadır. (Aşağıdaki şekilde bir tablo üzerinde satır ve alan gösterilmiştir.) Bir tabloda *en az bir tane alan* bulunması gerekmektedir. Tabloda satırlar olabileceği gibi hiçbir satır da olmayabilir, yani tablomuzda hiç veri taşımayacağımız durumlar olabilir. Tablo içerisinde kayıtlara ait bilgileri tutacak olan alanların hangi veri tipinde veri saklayabileceği de belirlenebilmektedir. Yukarıdaki öğrenci örneğindeki kayıtlarda, öğrenci adı ve soyadı bilgileri metinsel bir ifade, öğrenci numarası da rakamsal bir ifade içersin gibi…

Table - Ogrenciler

OgrenciId	OgrenciAd	OgrenciSoyad
128356	Kemal	Dağlıoğlu
128445	Yakup	Şeref
128539	Refika	Köseler

Şekil 121: Bir tablo üzerindeki satırlar ve alanlar

Şekil 121'de 1 numara ile belirtilmiş kısım satır, 2 numara ile belirtilmiş kısım ise alan olarak adlandırılır.

1. **Satır (Row)**: Tabloda bulunan bir kayıt.
2. **Alan (Sütun-Column)**: Bir kayda ait özellik. Kayıtlı öğrencinin adı.

Tablolar tasarlanırken içerecekleri bilgi türlerine göre gruplandırılmalıdır. Daha önce kısaca ele aldığımız örneği düşünecek olursak, bir üniversitenin veri tabanı fakülteler, bölümler ve öğrenciler gibi tablolardan oluşabilir. Bölüm ve öğrenci bilgileri birbirinden farklı olabileceği için, ayrı tablolarda tutulması hem veri bütünlüğü açısından, hem de veriye erişmenin daha kolay ve hızlı olması açısından önemlidir. İsterseniz şimdi örnek olarak bölümler ve öğrenciler tablosu oluşturarak, bu tabloların yapılarını ve içeriklerini inceleyelim.

Bolumler Tablosu

BolumId	BolumAd
101	Bilgisayar Mühendisliği
102	Bilgisayar Öğretmenliği
106	Çevre Mühendisliği
114	Fizik

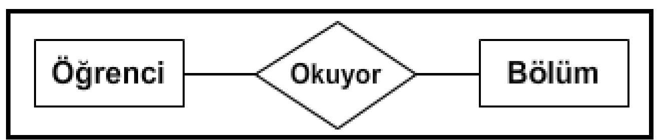
Ogrenciler Tablosu

OgrenciId	OgrenciAd	OgrenciSoyad	OgrenciBolumId
128356	Kemal	Dağlıoğlu	101
128445	Yakup	Şeref	114
128539	Refika	Köseler	102
128488	Oğuzhan	Alaşehir	102
129112	Elif	Çakır	101

Tablo 13: Bolumler ve Ogrenciler tabloları ve içerdiği bilgiler

Bolumler tablosu 2 alan (sütun) ve 4 tane satırdan (kayıt), **Ogrenciler** tablosu ise 4 alan ve 5 satırdan oluşmaktadır. Bu şekilde birbirinden farklı içerikleri olan bölümleri ve öğrencileri farklı iki tabloda depolamış olduk. Yine **Ogrenciler** adlı tablodaki **OgrenciBolumId** alanına dikkat edecek olursak; öğrencinin hangi bölümde bulunduğunun bilgisini **Bolumler** tablosundaki **BolumId** ile ilişkilendirmiş olduk. (Buradaki 101, 102 gibi ifadeler aslında Bolumler tablosundaki Bilgisayar Mühendisliği, Bilgisayar Öğretmenliği gibi bölümleri temsil etmektedir.) Bölümler tablosunda BolumId ve BolumAd dışında, bölüm adresi, bölüm başkanı, bölümün bağlı olduğu fakülte gibi bilgileri de tuttuğumuzu varsayarsak, bu bilgileri öğrencilerin bilgileri ile aynı tabloda tutmak hem gereksiz yere verilerin tekrarlanmasına, hem de veri karışıklığına sebep olacaktı.

İlişki diyagramlarında bir tablo ile diğer tablo arasındaki ilişki aşağıdaki şekilde olduğu gibi belirtilir.



Şekil 122: Bir öğrenci ve bir bölüm arasındaki ilişkinin şekille gösterimi

Tablo ve alan isimlendirilirken dikkat etmemiz gereken bazı hususlar vardır.

- MS SQL Server 2005'ten önceki versiyonlarda aynı veri tabanı içerisinde aynı isme sahip sadece bir tablo bulunabilirken, MS SQL Server 2005 ile gelen şema (schema) kavramı sayesinde bu durum aşılabilmektedir. Bununla birlikte, bir tablo içerisinde aynı isme sahip sadece bir alan bulunabilir.

- Tablo ve alan isimleri içerisinde rakam veya harfler dışındaki karakterlerin kullanılmaması tavsiye edilmektedir. (*)
- İsimlerin rakam ile başlamaması tavsiye edilmektedir. (*)

(*) Tavsiye edilmeyen durumlarda kullanılacak isimlerin ancak **[ ]** işaretleri arasına alınarak kullanılmasına izin verilmektedir.

Anahtarlar (Keys)

Bir kayıt içerisinde farklılıkları ve nitelikleri gösteren belirleyicilere **anahtarlar (keys)** denir. Farklı içeriklere sahip olacak verileri farklı tablolarda depolayarak yapabileceğimiz birçok işi kolaylaştırabiliyorduk. Benzer şekilde, tablodaki kayıtları da birbirinden ayırt edebilmek için tablo içindeki alanlara belirli anahtarlar atayarak birçok işlemi kolaylaştırabilmekteyiz. Bir tablo içerisinde bulunabilecek anahtarlar, **birincil anahtar** (primary key), **tekil anahtar** (unique key), **referans anahtar** (foreign key) ve **birleşik anahtar**dır. (composite key)

Birincil anahtar (Primary key): Bir tablo içerisindeki satırları birbirinden ayırt eder. Birincil anahtar olan bir veri aynı tablo içerisinde tekrarlanamaz. Yine bu alandaki veri boş bırakılamaz, yani *NULL* değeri alamaz. Tek bir alan birincil anahtar olabileceği gibi bazı tablolarda birden fazla alanın birleşmesiyle birincil anahtar oluşabilir. (Bu aslında az sonra göreceğimiz birleşik anahtardır)

Tekil anahtar (Unique key): Tablonun tekil anahtar olarak tanımlanmış bir alanına aynı değer sadece bir kez girilebilir. Birincil anahtardan farklı olarak, tabloda bu alana ait sadece bir kayıt NULL değeri alabilir. Birincil anahtar aynı zamanda tek anahtar olarak sayılabilir fakat tek anahtarlar birincil anahtar değildirler.

Referans anahtar (Foreign key): Tablodaki bir veriyi başka tablodaki bir veri ile ilişkilendirir. İki tablo arasında yapılan bu ilişkilendirme ile referans anahtar olarak tanımlanmış alana sadece ilişkilendirdiği tablonun alanındaki veriler eklenebilir.

Birleşik anahtar (Composite Key): Birden fazla alanın birleştirilmesiyle birincil anahtar görevini üstlenecek tanımlamalar yapılabilir. Bunlar birleşik anahtar olarak adlandırılır.

Ogrenciler Tablosu

OgrenciId	OgrenciAd	OgrenciSoyad	OgrenciBolumId	EPosta
128356	Kemal	Dağlıoğlu	101	e128356@metu.edu.tr
128445	Yakup	Şeref	114	yakup@netron.com.tr
128539	Refika	Köseler	102	e128539@metu.edu.tr
128488	Oğuzhan	Alaşehir	102	NULL
129112	Elif	Çakır	101	e129112@metu.edu.tr

Primary key — Foreign key — Unique key

Şekil 123: Ogrenciler tablosu üzerinde bulunan anahtarlar

OgrenciId her öğrenci için tek ve belirleyici bir unsur olacağı için Ogrenciler tablosunda *birincil anahtar* olarak belirlenmiştir. Yine **EPosta** her öğrenci için tek olacağı için bu alan tablo için *tekil anahtar* olacaktır. Daha önceki örneğimizden hatırlayacağınız gibi **OgrenciBolumId** Bolumler tablosunun BolumId alanı ile ilişkilendirilmişti. Ogrenciler tablosunun bu alanına sadece Bolumler tablosundaki değerleri alabilmektedir. Başka bir tablodaki anahtar ile ilişkilendirildiği için tablonun *referans anahtarı* olmuştur.

İndeksler (Indexes)

Tablolardaki verilere daha hızlı ulaşmak için hazırlanan yapılara indeks(index) denir. Bir kütüphanedeki kitapların raflara rastgele dizildiğini düşünün. Bir kitap arandığı zaman tüm kitaplara teker teker bakılması gerekir. Eğer şanssız iseniz belki de baktığınız son kitap, aradığınız kitap olabilir. Bir de bu kitapları raflara belirli bir sıraya göre dizildiğini ya da kitapların hangi rafta, kaçıncı sırada olduğunu belirten bir liste hazırlandığını varsayalım. Örneğin, kitaplar adlarına göre alfabetik sırayla dizildiğinde bir kitabı aradığımız zaman rafları ve kitapları tek tek gezmek zorunda kalmayız. Aradığımız kitaba alfabetik sırasını takip ederek çok kolay bir şekilde ulaşabiliriz. Bir veri tabanında indekslerin yaptığı işlev de tam olarak budur. Yine önceki konularda ele aldığımız Ogrenciler tablosunda öğrencileri isimlerine ya da bölümlerine göre sıralayacak indeksler oluşturup daha sonra bu verilere daha hızlı bir şekilde ulaşılması sağlanabilir. Böylece bir sorgulama işleminde, aranan kayıt indeks içerisindeki yeri bilineceği için otomatik olarak bulunur ve kayıtlar arasında tek tek gezilmek zorunda kalınmaz.

BÖLÜM 1: T-SQL GİRİŞ

SQL (Yapısal Sorgulama Dili)

SQL, yani **Structered Query Language (Yapısal Sorgulama Dili)** tüm ilişkisel veritabanlarında standart olarak kullanılan bir dildir. SQL veri tabanı ile kullanıcı arasındaki iletişimi sağlayan dildir. Kullanıcılar bu dil aracılığıyla hazırladıkları sorguları oluşturarak veri tabanında depolanan veriler üzerinde bütün işlemleri yapabilmektedir. SQL dilinin standartları ANSI (American National Standart Institute) ve ISO (International Standarts Organization) tarafından sağlanmakla birlikte, günümüzde en yaygın olarak ANSI standartları kullanılmaktadır.

T-SQL (Transact SQL)

T-SQL, **Transact SQL** adı verilen bir SQL dilidir. SQL dilinin **Microsoft SQL Server** üzerinde kullanılan sürümüdür. Daha iyi performans sağlaması için SQL dili üzerine eklentiler ve fonksiyonellikler eklenerek oluşturulmuştur. Veri tabanından bağımsız olarak, bir programlama dili aracılığıyla kullanıcıdan gelen T-SQL sorgularının sonuçları ilişkisel veri tabanı yönetim sistemi (RDBMS) tarafından oluşturularak kullanıcıya gönderilir. Bu şekilde kullanıcı, veri tabanı ile birebir uğraşmaksızın, sadece sorgular yazarak veri tabanı üzerinde işlemler yapabilir, veri tabanından gelen sonuçları program veya bir web sayfası üzerinde görüntüleyebilir.

T-SQL'de Veri Tipleri

Tablo oluştururken tablo içerisindeki her alanın hangi tipte veri taşıyabileceğini belirleyen bazı veri tipleri bulunmaktadır. T-SQL'de bulunan temel veri tipleri şunlardır:

Metin Veri Tipleri

Tip	Değer Aralığı
char(n)	ASCII türünden ve sabit boyutta veri saklar. En fazla 8000 karakter tutulabilir. (n) alabileceği en fazla karakteri belirler.
nchar(n)	Unicode türünden ve sabit boyutta veri saklar. En fazla 4000 karakter tutulabilir.
varchar(n)	ASCII türünden ve değişir uzunlukta veri saklar. En fazla 8000 karakter tutulabilir.
nvarchar(n)	Unicode türünden ve değişir uzunlukta veri saklar. En fazla 4000 karakter tutulabilir.
varchar(MAX)	varchar veri tipi ile aynı özelliklere sahiptir. Fakat 2 GB'a kadar veri tutabilmektedir.
nvarchar(MAX)	nvarchar veri tipi ile aynı özelliklere sahiptir. Fakat 2 GB'a kadar veri tutabilmektedir.
text	ASCII türünden metin saklamak için kullanılır. 2 GB'a kadar sınırı vardır.
ntext	Unicode türünden metin saklamak için kullanılır. 2 GB'a kadar sınırı vardır.

Tablo 14: Sql metin veri tipleri

Sayısal Veri Tipleri

Tip	Değer Aralığı
int	Yaklaşık -2 milyar ile +2 milyar arasındaki tamsayı değerlerini tutar.
bigint	Yaklaşık -2^{63} ile $+2^{63}$ arasındaki tamsayı değerleri tutar.
smallint	Yaklaşık -32 bin ile +32 bin arasındaki tamsayı değerlerini tutar.
tinyint	0-255 arasındaki tamsayı değerlerini tutar.
float(n)	Kayan noktalı sayı değerlerini tutar. -1.79e+308 ile 1.79E+308 arasında değer tutabilir. n, 1 ile 53 arasında değer alabilir. 1 ile 24 arasında olduğunda 7 haneye kadar hassasiyet ve 4 byte yer ayrılması söz konusudur. 25 ile 53 aralığı için ise 15 haneye kadar hassasiyet ve 8 byte yer ayrılması söz konusudur. Varsayılan olarak n değeri 53'tür.
real	-3.40e+38 ile 3.40e+38 arasında değerler alabilir. 7 haneye kadar hassasiyet sunar ve 4 byte yer kaplar. Bu veri tipi float(24)'ün karşılığıdır. Eğer 7 haneye kadar hassasiyet gerekiyorsa real tipi varsayılan float tipi yerine tercih edilebilir.
money	Yaklaşık -922 milyar ile +922 milyar arasındaki değerleri tutar. Bu tip genelde parasal değerlerin tutulacağı alanlarda kullanılır.

Tablo 15: Sql sayısal veri tipleri

Tarihsel Veri Tipleri

Tip	Değer Aralığı
datetime	01.01.1753 ile 31.12.9999 arasındaki tarihleri tutar.
smalldatetime	01.01.1900 ile 06.06.2079 arasındaki tarihleri tutar.

Tablo 16: Sql tarihsel veri tipleri

Diğer Veri Tipleri

Tip	Değer Aralığı
bit	Boolean değerler tutmak için kullanılan veri tipidir. Sadece 1 veya 0 değerlerini alabilir. 1 True, 0 False değerlerini temsil eder.
image	Resim dosyalarının veri tabanında tutulması için kullanılan veri tipidir. 2 GB'a kadar resim dosyası tutabilmektedir.
xml	XML dosyalarını ve XML kodlarını saklayabilen veri tipidir. 2 GB'a kadar veri taşıyabilir.
binary(n)	Sabit uzunluktaki binary veriyi tutmak için kullanılır. Maksimum uzunluğu 8000 byte' tır. Varsayılan uzunluğu ise 1 byte' tır.
varbinary(n)	Değişken uzunlukta binary veriyi tutmak için kullanılır. Maksimum uzunluğu 8000 byte' tır. Varsayılan uzunluğu ise 1 byte' tır.
varbinary(MAX)	Maksimum 2 GB binary veriyi tutabilen ve Sql Server 2005 ile birlikte gelen veri tipidir.

Tablo 17: Sql diğer veri tipleri

Microsoft, Sql Server'ın 2005 sürümüyle birlikte, image, text ve ntext veri tipleri yerine varbinary(MAX), nvarchar(MAX), varchar(MAX) türlerinin kullanılmasını tavsiye etmektedir.

T-SQL İfade Tipleri

T-SQL içerisinde 3 tane ifade tipi bulunmaktadır.

- **Veri Tanımlama Dili (Data Definition Language)**
- **Veri Kontrol Dili (Data Control Language)**
- **Veri İşleme Dili (Data Manipulation Language)**

Veri Tanımlama Dili (Data Definition Language - DDL)

Veri tabanında nesneler oluşturmak için gerekli olan ifadeleri sağlamaktadır. Bu ifadeler veri tabanı veya tablo gibi yeni bir nesnenin oluşturulması, var olan nesne üzerinde değişiklikler yapılması ve nesnenin yok edilmesi için kullanılır. Üzerinde işlem yapılan nesnenin ne gibi özellikleri ve alanları olacağı bu ifadeler içerisinde belirlenir.

T-SQL dilinde 3 tane veri tanımlama ifadesi bulunmaktadır. Bunlar:

- **CREATE**
- **ALTER**
- **DROP**

ifadeleridir. CREATE nesne oluşturmak, ALTER var olan bir nesne üzerinde değişiklikler yapmak, DROP ise varolan bir nesneyi kaldırmak için kullanılmaktadır.

CREATE

Veri tabanı üzerinde nesne oluşturmak ya da tanımlamak için kullanılan komuttur. Oluşturulacak nesnenin özelliklerine göre farklı parametreler alabilmektedir. Genel kullanımı şu şekilde olmaktadır:

```
CREATE NESNETİPİ Nesne Adı (Nesneye ait gerekli tanımlamalar)
```

Aşağıda CREATE ifadesi ile ilgili örnek kullanımlar bulunmaktadır.

```
CREATE DATABASE Universite
ON
(
      NAME = Universite,
      FILENAME = 'C:\universite.mdf',
      SIZE = 4mb,
      MAXSIZE = 10mb,
      FILEGROWTH = 1mb
)
```

Yukarıdaki ifadede **Universite** adında yeni bir veri tabanı oluşturuyoruz. **ON ( )** kısmında oluşturulan veri tabanının özellikleri belirleniyor. **NAME** değeri ile oluşan birincil veri tabanı dosyasının adını, **FILENAME** değeri ile veri tabanı dosyasının nereye ve ne isimle yazılacağını, **SIZE** değeri ile dosyanın diskte ne kadar yer kaplayacağını, **MAXSIZE** ile dosyanın diskte en fazla ne kadar yer kaplayabileceğini, **FILEGROWTH** ile dosyanın ne kadarlık boyutlarla artacağını belirlemektedir.

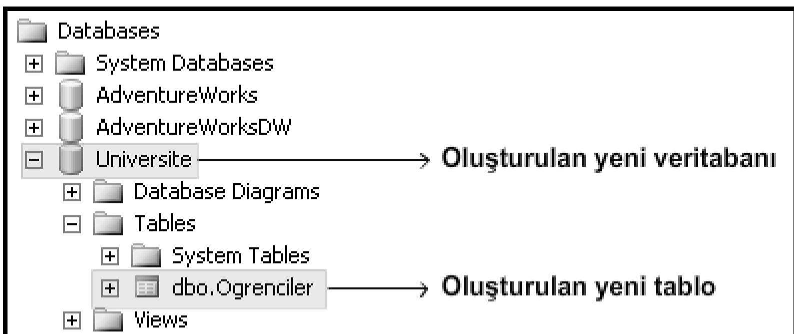
Sadece **CREATE DATABASE Universite** ifadesini kullanarak ta Universite isimli bir veri tabanı oluşturabiliriz. Bu durumda oluşturacağımız veri tabanı sistem tarafından atanmış olan varsayılan ayarlamalara göre oluşturulacaktır.

Universite adında yeni bir veri tabanı oluşturduktan sonra bu veri tabanı içerisinde aşağıdaki sorgular aracılığıyla tablolar oluşturabiliriz.

```
CREATE TABLE Ogrenciler
(
   OgrenciId INTEGER NOT NULL,
   OgrenciAdSoyad VARCHAR(25),
   OgrenciDogumTarih SMALLDATETIME,
   OgrenciBolumId INTEGER
)
```

Yukarıdaki ifadede **Ogrenciler** adında yeni bir tablo oluşturuluyor. Tablonun içeriği **( )** işaretlerinin arasındaki kısımda belirleniyor. Her alan tanımlaması arasında ise **,** (virgül) işareti bulunmaktadır. Bu tabloda şu alanlar bulunuyor:

- **OgrenciId**: INTEGER tipinden veri alabilen ve NULL değer almayan(boş bırakılamayan) bir alan.
- **OgrenciAdSoyad**: VARCHAR tipinden veri alabilen ve en fazla 25 tane karakter girilebilen bir alan.
- **OgrenciDogumTarih:** SMALLDATETIME tipinden veri alabilen bir alan.
- **OgrenciBolumId**: INTEGER tipinden veri alabilen bir alan.



Şekil 124: Çalıştırılan sorgular sonucunda Universite adında bir veri tabanı ve bu veri tabanının içerisinde Ogrenciler isimli bir tablo oluşturulur

```
CREATE TABLE Bolumler
(
   BolumId INTEGER,
   BolumAd VARCHAR(40),
   CONSTRAINT PK_Bolumler PRIMARY KEY (BolumId)
)
```

Yukarıdaki ifadede ise Bolumler adında yeni bir tablo oluşturuluyor. Bu tabloda şu alanlar bulunuyor:

- **BolumId**: INTEGER tipinden veri alan bir alandır. Bu alan sorgunun **PRIMARY KEY (BolumId)** ifadesi ile Bolumler tablosunun birincil anahtarı(primary key) olarak ayarlanmaktadır. Birincil anahtar olan bir alan tanımlanırken NOT NULL ifadesinin belirtilmesine gerek yoktur. Bu alan otomatik olarak NULL değer almayacak şekilde kaydedilecektir.
- **BolumAd**: VARCHAR tipinden veri alabilen ve en fazla 40 tane karakter girilebilen bir alandır.

İfade içerisinde yer alan **CONSTRAINT PK_Bolumler** ifadesi ile tabloya PK_Bolumler ismine sahip bir kısıtlama eklenir ve bu kısıtlama tablonun birincil anahtarı olan BolumId'ye bağlanır. Böylece **CREATE** komutunu kullanarak **Universite** adında bir veri

tabanı oluşturulur. Yine CREATE komutu aracılığıyla Universite adlı veri tabanı içerisine **Ogrenciler** ve **Bolumler** adında iki tablo eklenmiş olur.

T-SQL sorgu cümlelerinde, yazacağımız komut kelimelerini büyük harfle yazmak, (CREATE, TABLE, ALTER, INTEGER, VARCHAR gibi) tanımlamaları ise Pascal isimlendirme kurallarına göre yazmak (Örneğin OgrenciId, OgrenciAdSoyad, UretimTarihi vb. şeklinde) cümlelerin okunabilir olması açısından tavsiye edilen bir kullanım biçimidir.

ALTER

Varolan bir nesne üzerinde değişiklikler yapmak için kullanılır. CREATE komutunda olduğu gibi değiştireceği nesneye göre farklı parametreler alabilmektedir. Genel kullanımı aşağıdaki gibidir.

```
ALTER NESNETIPI Nesne Adı Yapılacak Değişiklik
```

Aşağıdaki örneklerde ALTER ifadesi kullanılarak varolan bir tablo üzerinde nasıl değişiklikler yapılabileceği gösterilmiştir.

```
ALTER TABLE Ogrenciler
ALTER COLUMN OgrenciAdSoyad VARCHAR(30) NOT NULL
```

Ogrenciler tablosundaki **OgrenciAdSoyad** alanına artık VARCHAR tipinden en fazla 30 karakter girilebilir. NOT NULL ifadesi ile birlikte bu alanın boş bırakılması engellenir. Burada **ALTER TABLE** ifadesiyle Ogrenciler tablosu, **ALTER COLUMN** ifadesiyle ise OgrenciAdSoyad alanı değiştirilmektedir. (Tablo içerisindeki bir alanı NOT NULL olarak değiştirebilmek için tablo içerisinde bu alandaki kayıtların boş olmaması gerekecektir.)

```
ALTER TABLE Ogrenciler
ADD OgrenciEPosta VARCHAR(30)
```

Ogrenciler tablosuna **ADD** ifadesi ile **OgrenciEPosta** adında bir alan eklenir. Bu alana VARCHAR tipinden en fazla 30 karakterlik veri girilebilir.

```
ALTER TABLE Ogrenciler
ADD CONSTRAINT PK_Ogrenciler PRIMARY KEY (OgrenciId)
```

Ogrenciler tablosuna **ADD** ifadesi ile birincil anahtar eklenmekte ve **OgrenciId** alanı, tablonun birincil anahtarı (primary key) olarak ayarlanmaktadır. **CONSTRAINT PK_Ogrenciler** ifadesi ile tabloya PK_Ogrenciler ismine sahip bir kısıtlama eklenmekte ve bu kısıtlama tablonun birincil anahtarına bağlanmaktadır. Yapılan değişikliklerden sonra Ogrenciler tablosunun yapısı aşağıdaki gibidir.

Column Name	Data Type	Allow Nulls
OgrenciId	int	☐
OgrenciAdSoyad	varchar(30)	☑
OgrenciDogumTarih	smalldatetime	☑
OgrenciBolumId	int	☑
OgrenciEPosta	varchar(30)	☑

Şekil 125: Yapılan değişikliklerden sonra tablonun son hali

DROP

Veri tabanındaki herhangi bir nesneyi silmek için kullanılır. Silinen nesne ile ilgili olarak içerisinde tuttuğu tüm bilgiler de silinmektedir. Örneğin bir tablo silindiğinde içerisindeki tüm bilgiler de veri tabanından silinecektir. Kullanımında dikkat edilmesi gereken bir sorgu ifadesidir. Genel kullanımı aşağıdaki gibidir.

```
DROP NESNETIPI Nesne Adı
```

Aşağıdaki örneklerde DROP ifadesi kullanılarak varolan tabloların nasıl silinebileceği gösterilmiştir.

Üzerinde çalışmış olduğunuz veri tabanında herhangi bir veri kaybı yaşamamak için, önce geçici olarak işlevi olmayan bir veri tabanı ve tablo oluşturup, bu nesneleri DROP sorgusu ile nasıl silebileceğimizi görelim.

```
CREATE DATABASE TestVeritabani
GO

USE TestVeritabani
CREATE TABLE TestTablo
(
   Isim VARCHAR(10)
)
```

Yukarıdaki ifade ile **TestVeritabani** adında bir veri tabanı oluşturulur. **GO** komutu öncelikle **CREATE DATABASE TestVeritabani** sorgusunun çalışmasını sağlar. Daha sonra **USE TestVeritabani** ifadesi ile oluşturulan veri tabanı üzerine **TestTablo** adında bir tablo oluşturulur. Şimdi DROP komutu ile tablomuzu ve veri tabanımızı silelim.

GO ifadesi bir sorguyu çalıştır anlamında kullanılmaktadır. Yukarıdaki sorguda iki farklı sorgu cümlesi bulunmaktadır ve buradaki ikinci cümlenin çalışabilmesi için öncelikle TestVeritabani isimli bir veri tabanının oluşturulması gerekmektedir. GO ifadesi önce ilk sorgunun çalışmasını sağlamaktadır. Daha sonraki sorguda kullanılan **USE TestVeritabani** ifadesi ise, sıradaki sorgunun TestVeritabani isimli veri tabanı üzerinde çalışmasını sağlayacaktır.

```
DROP TABLE TestTable
```

DROP TABLE ifadesi ile **TestTablo** isimli tablo silinmektedir.

```
DROP DATABASE TestVeritabani
```

DROP DATABASE ifadesi ile **TestVeritabani** isimli veri tabanı silinmektedir.

Veri Kontrol Dili (Data Control Language - DCL)

T-SQL'de veri kontrol ifadeleri veri tabanı üzerindeki kullanıcılara ve rollere hak vermek için kullanılmaktadır. T-SQL'de 3 tane veri kontrol komutu bulunmaktadır. Bunlar:

GRANT

Kullanıcıya veri tabanına erişebilmesini veya T-SQL ifadeleri çalıştırabilmesini sağlayacak yetkileri verir.

DENY

Kullanıcının belirli bir alana erişimini veya belirli T-SQL ifadelerini çalıştırmasını engellemek amacıyla kullanılır.

REVOKE

Daha önceden GRANT veya DENY ile verilmiş yetki ve engelleri kaldırmak için kullanılır.

Veri kontrol ifadelerini çalıştırabilmek için veri tabanına bağlı olan kullanıcının **sysadmin**, **dbcreator**, **db_owner** veya **db_securityadmin** rollerinden birine sahip olması gerekmektedir.

Veri İşleme Dili (Data Manipulation Language - DML)

Veri işleme ifadeleri, veri tabanı üzerinde depolanan veriler üzerinde yapılması gereken işlemler için kullanılan ifadelerdir. Veri seçme, veri ekleme, veri güncelleme ve veri silme gibi işlemlerin yapılmasını sağlarlar. Toplam 4 tane veri işleme ifadesi bulunmaktadır.

- **SELECT**
- **INSERT**
- **UPDATE**
- **DELETE**

İsimlerinden de anlaşılacağı gibi, SELECT tablolardan veri seçme işlemini, INSERT tablolara veri ekleme işlemini, UPDATE varolan verileri değiştirme, güncelleştirme işlemini, DELETE ise varolan verilerin silinmesi işlemini gerçekleştirmektedir.

SELECT

Bir tablo içerisindeki verilerin tamamını veya belirli şartlara göre bir kısmını filtreleyerek seçme işlemlerini gerçekleştirir. Genel kullanım şekli aşağıdaki gibidir.

```
SELECT SeçilecekAlan1, SeçilecekAlan2, …. FROM TabloAdı
```

Örnek Kullanım: Ogrenciler tablosundaki tüm kayıtların OgrenciId ve OgrenciAdSoyad bilgileri aşağıdaki sql ifadesi ile elde edilebilir.

```
SELECT OgrenciId, OgrenciAdSoyad FROM Ogrenciler
```

OgrenciId	OgrenciAdSoyad
128539	Kemal Dağlıoğlu
128445	Yakup Şeref
128539	Refika Köseler
128488	Oğuzhan Alaşehir

Şekil 126: Sorgu sonucunda seçilen öğrencilerin numaraları ve ad-soyadları

INSERT

Bir tablo içerisine yeni bir veri eklemek için kullanılır. Genel kullanımı aşağıdaki gibidir.

```
INSERT INTO TabloAdı (VeriEklenecekAlan1, VeriEklenecekAlan2, ...)
VALUES (EklenecekDeğer1, EklenecekDeğer2, ...)
```

Örnek Kullanım: Ogrenciler tablosuna öğrenci numarası 129302 olan 101 numaralı bölümden Muhammet Turşak isimli öğrenciyi aşağıdaki sql ifadesi yardımıyla eklenebilir.

```
INSERT INTO Ogrenciler (OgrenciId, OgrenciAdSoyad, OgrenciBolumId)
VALUES (129302, 'Muhammet Turşak', 101)
```

UPDATE

Bir tablo içerisinde bulunan verilerin değiştirilmesi için kullanılır. Genel kullanımı aşağıdaki gibidir.

```
UPDATE TabloAdı Set GuncellenecekAlan1 = YeniVeri1,
GuncellenecekAlan2 = YeniVeri2 WHERE Koşul veya koşullar
```

Örnek Kullanım: Ogrenciler tablosunda ogrenci numarası 129302 olan öğrencinin bölüm kodunu 102 olarak değiştirmek için aşağıdaki sql ifadesinden yararlanılabilir.

```
UPDATE Ogrenciler SET OgrenciBolumId = 102
WHERE OgrenciId = 129302
```

DELETE

Tablo içerisinde bulunan bir kaydı veya kayıtları silmek için kullanılır. Genel kullanım şekli aşağıdaki gibidir.

```
DELETE FROM TabloAdı WHERE Koşul veya koşullar
```

Örnek Kullanım: Ogrenciler tablosundaki 129302 numarasına sahip öğrencinin kaydını silmek için aşağıdaki sql ifadesinden yararlanılabilir.

```
DELETE FROM Ogrenciler WHERE OgrenciId = 129302
```

T-SQL Sorgulama Araçları

Veri tabanı üzerindeki işlemleri programlama dilleri aracılığıyla yapabildiğimiz gibi SQL Server 2000 ve SQL Server 2005 üzerinde bulunan bazı araçlar yardımıyla da gerçekleştirebiliriz. Bu sorgulama araçlarından bazıları şunlardır:

- **SQL Server 2000 Query Analyzer**
- **SQL Server Management Studio**

SQL Server 2000 Query Analyzer

Query Analyzer, veri tabanı üzerinde T-SQL sorgularını çalıştırmak, sonuçlarını görmek, sorguları analiz etmek ve veri tabanı üzerinde bazı ayarlamaları yapmak için kullanabileceğimiz bir araçtır. Query Analyzer;

- Veri tabanı ve tablolar üzerinde yapılabilecek tüm sorgu işlemlerinin gerçekleştirebileceği bir metin editörü sunmaktadır. Bu metin editörü yazılan kodları renklendirilerek sorguların daha anlaşılır olmasını sağlar.
- Çalışan sorguların sonuçlarının görülmesini sağlar.
- Hazırlanan sorguların test edilmesini sağlar. Sorguda bir hata olması durumunda hatanın neden olduğunu ve nereden kaynaklandığını belirtir.
- Bağlı olunan veri tabanı üzerindeki tabloların ve diğer nesnelerin görüntülenmesini sağlayan bir araç sunar. (Object Browser)
- Birçok SQL sorgusunun hazır olarak bulunduğu şablonlar (template) içermektedir. Bu şablonlar yardımıyla kolay bir şekilde sorgular hazırlanabilmektedir.

Query Analyzer, SQL Server 2000 ile birlikte gelen bir araçtır. SQL Server 2000 üzerinde olduğu gibi, SQL Server 2005 veritabanları üzerinde de sorgular çalıştırabilmemizi sağlar. Bu kitabımızdaki konular boyunca SQL Server 2005 veri tabanı üzerinde çalışacağımız için, **Microsoft SQL Server 2005** ile birlikte gelmiş ve yapısında birçok yeniliği barındıran **SQL Server Management Studio** aracını inceleyeceğiz.

SQL Server Management Studio Üzerinde Sorgularla Çalışmak

SQL Server Management Studio, Microsoft SQL Server 2005 ile birlikte gelmiş olan yeni bir sorgulama ve yönetim aracıdır. Management Studio, SQL Server veritabanlarına erişme, veritabanları üzerinde ayarlama işlemlerini gerçekleştirme, yönetim ve veri tabanı üzerinde sorgular çalıştırma gibi işlemleri yapabilmektedir. Yine Management Studio üzerinde SQL Server projeleri oluşturup, çalışmaların Visual Studio 2005 ortamında olduğu gibi proje şeklinde kaydedip, daha sonra bu projeler üzerinde geliştirme işlemleri yapılabilmesi sağlanmaktadır.

Management Studio, SQL Server 2000 ile birlikte gelen Query Analyzer, Enterprise Manager ve Analysis Manager gibi araçların birleştirilmiş ve daha da geliştirilmiş halidir.

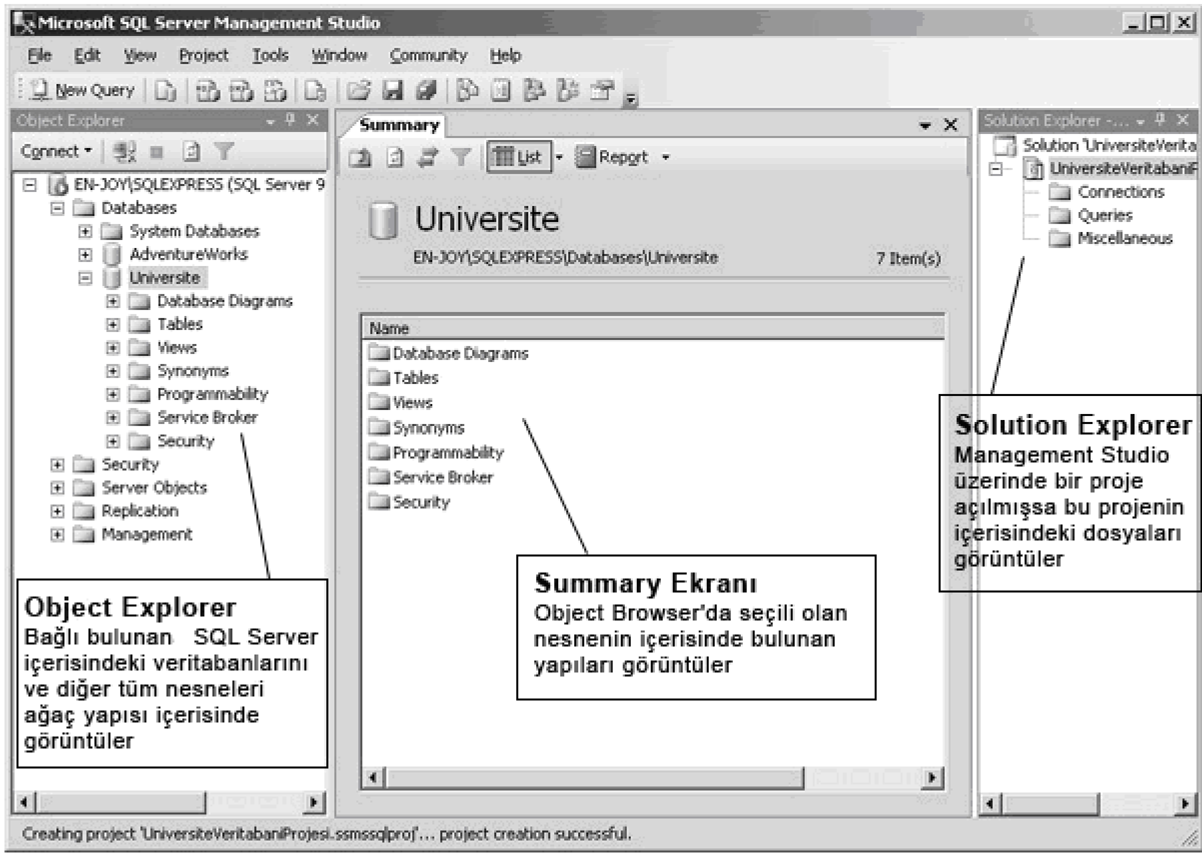
SQL Server Management Studio, Microsoft tarafından SMO (SQL Server Management Object) kütüphanesinden yararlanılarak **Visual Studio 2005** ile geliştirilmiştir.

SQL Server Management Studio'yu açmak için Türkçe işletim sistemlerinde **Başlat > Programlar > Microsoft SQL Server 2005** menüsünden **SQL Server Management Studio** programını çalıştırmamız gerekmektedir.

Management Studio açıldığında karşımıza ilk olarak bir SQL Server sunucusuna ulaşmamız için bir bağlantı penceresi gelecektir. Bu pencereden **Server Type**, **Server Name** ve **Authentication** seçeneklerinden bağlanacağımız sunucunun özelliklerine göre uygun seçenekleri girerek bağlantı yapabiliriz. Kendi bilgisayarımızdaki SQL Server sunucusuna bağlanmak istediğimizde, Server Name kısmında bilgisayarımızda kayıtlı olan SQL Server'ın ismini (instance name) yazmamız gerekir. Uzak bir bilgisayar üzerindeki SQL Server sunucusuna bağlanmak için ise Server Name kısmına bağlanılacak bilgisayarın IP numarası veya uzak sunucudaki SQL Server'ın ismi yazılır. Windows üzerinde o an giriş yapmış olan kullanıcının hesabı ile bağlantı yapmak için Windows Authentication seçeneğini seçmemiz yeterli olacaktır. Bağlanılacak SQL Server üzerindeki bir kullanıcı ile giriş yapmak için ise SQL Server Authentication'ı seçip sunucuya bağlanma yetkisi olan kullanıcı adı ve şifreyi girmemiz gerekecektir. **Connect** butonuna tıkladığımızda girilen bilgiler doğru ise SQL Server sunucusuna bağlanılacaktır.

Şekil 127: SQL Server Management Studio bağlantı ekranı

Management Studio ile bir SQL Server sunucusuna bağlandığımızda karşımıza gelen arayüzde **Object Explorer**, **Solution Explorer** ve **Summary** pencereleri açılmaktadır. (Bu pencerelerden herhangi biri açık değilse, **View** menüsünden kapalı olan pencereler açılabilir.)



Şekil 128: – SQL Server Management Studio genel görünümü

Object Explorer: Bağlı bulunan SQL Server sunucusu üzerinde bulunan tüm nesneleri (veritabanları, tablolar, kullanıcılar vb.) ağaç yapısı biçiminde görüntülememizi ve bu yapıların içerisinde gezinmemizi sağlar. Yine Object Explorer üzerinden yeni veri tabanı oluşturma, tablo ekleme, varolan nesneleri silme gibi işlemler yapılabilmektedir.

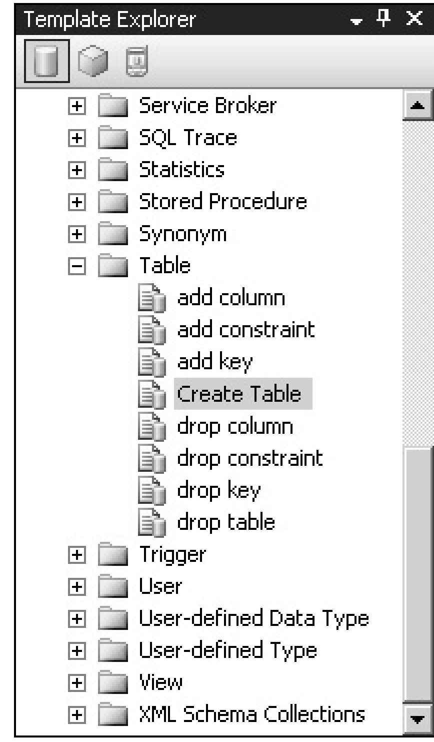
Görüntülenen nesneler üzerinde filtreleme işlemleri yapılabilmektedir. Management Studio ile birden fazla SQL Server sunucusuna bağlanabilir, bağlı bulunduğumuz tüm sunucuları Object Explorer penceresi içerisinde görüntüleyebilir ve üzerlerinde işlemler yapabiliriz. Object Explorer üzerinde yapabileceğimiz bazı temel işlemler şunlardır:

- Yeni bir SQL Server sunucusuna bağlanmak için Object Explorer penceresinin sol üst köşesinde yer alan **Connect** butonu aracılığı ile aynı anda birden fazla SQL Server sunucusuna bağlanılabilir.
- Bağlı olduğunuz SQL Server'da sorgu çalıştırmak için sunucu adının üzerine sağ tıklayıp **New Query** seçeneği seçilir.
- Bir veri tabanı üzerinde sorgu çalıştırmak için o veri tabanının üzerine sağ tıklayıp **New Query** seçeneği seçilir.
- Veri tabanına yeni bir tablo eklemek için veri tabanı içindeki **Tables** kısmında sağ tıklayarak **New Table** seçeneği seçilir.
- Bir tablonun yapısını incelemek ve değişiklikler yapmak için tablo üzerine sağ tıklayıp **Modify** seçeneği seçilir.
- Bir tablo içerisindeki kayıtları görmek için tablo üzerine sağ tıklayıp **Open Table** seçeneğini kullanmak gerekir.

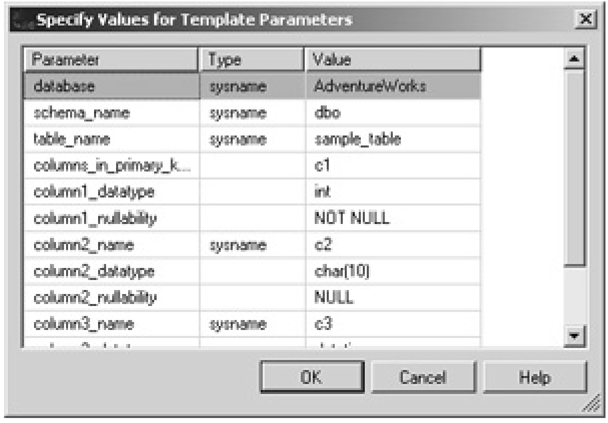
Solution Explorer: SQL Server Management Studio, Microsoft Visual Studio 2005 IDE'sinde olduğu gibi yaptığınız çalışmaları, proje (project) veya çözüm (solution) dosyaları olarak kaydetmenizi sağlar. Hazırlanan projede kod script dosyaları ve bağlantı bilgileri tutulur. Yeni bir proje açmak için **File > New** menüsünden **SQL Server Script** seçilir. Management Studio proje için oluşturduğu dosyaları Solution Explorer içerisinde görüntüler. Daha sonra proje için gerekli olan SQL kod dosyaları ve bağlantı bilgileri bu pencere içerisinden oluşturabilir ve yönetilebilir. Daha önceden hazırlanan bir SQL Server Scripts projesi, **File > Open > Project/Solution** seçeneğinden proje adı seçilerek açılabilir.

Summary Ekranı: Bu ekranda ise Object Explorer içerisinde o an için seçilmiş olan nesnenin içeriği görüntülenmektedir. Pencerenin kendi içerisinden bir üstteki veya bir alttaki yapılara ulaşılabilir. Veri tabanı içerisinde gezinme işlemlerini daha kolay hale getirmek için bu pencere kullanılabilir.

Template Explorer: SQL Server'ın içerisinde gelen ve bazı işlemleri basit bir şekilde yapmamızı sağlayan **sorgu şablonları(template)** bulunmaktadır. Bu şablonlar içerisinde birçok işlevi yerine getiren SQL sorgu cümleleri bulunmaktadır. Template Explorer içerisindeki sorgu şablonları açılıp gerekli değişiklikler yapılarak sorgular çalıştırılabilir ve hızlı bir şekilde işlemler gerçekleştirilebilir. Şablon içerisinde değiştirilmesi gereken parametreler, **Query** menüsünden **Specify Values for Template Parameters** seçeneği ile çıkan penceredeki gerekli parametreler girilerek düzenlenebilir.



Şekil 129: Template Explorer içerisinde bulunan hazır SQL sorgu şablonları

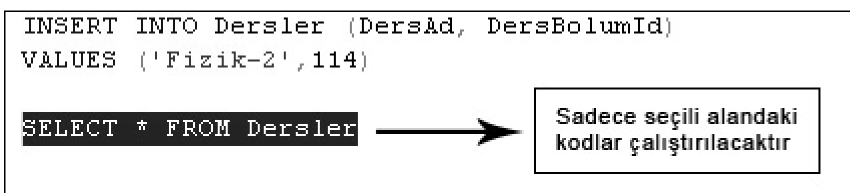


Şekil 130: Specify Values for Template Parameters ekranından template içerisindeki parametreler kolay bir şekilde değiştirebilir

Management Studio'da SQL Komutları ile Çalışmak

Management Studio içerisinde SQL komutları çalıştırmak için komutların çalıştırılacağı veri tabanı üzerine sağ tıklayıp New Query seçeneğini seçmek gerekmektedir. Açılan pencerede, yazılan SQL cümleleri test edilip, çalıştırılabilir. Yazılan SQL sorgularını test etmek için **Query** menüsünden **Parse** seçeneği seçilebilir veya kısayol olarak **CTRL+F5** tuş kombinasyonu kullanılabilir. Test etme işlemi sadece yazılan kodların doğru olup olmadığını kontrol etmek için yapılan bir işlemdir. Test edilen kodlar çalıştırılmaz, yani veri tabanı veya tablolar üzerinde herhangi bir işlem yapılmaz. Test işlemi sonucunda eğer yazılan SQL ifadelerinde bir hata varsa, alt kısımda açılan bir pencerede hatanın neden kaynaklandığı ve kodun neresinde oluştuğu belirtilmektedir. Hazırlanan SQL ifadelerini çalıştırmak için ise **Query** menüsünden **Execute** seçeneği seçilebilir veya kısayol tuşu olarak **F5** kullanılabilir. Test işleminden farklı olarak çalıştırma işleminde veri tabanı hazırlanan sorgu sonuçlarından etkilenecektir.

Sorgu çalıştırma ekranında birden fazla SQL ifadesi yazılabilir ve bunlardan sadece istenilen kısım çalıştırılabilir. Çalıştırılmak istenen SQL kodları seçilerek Execute (F5) işlemi yapılırsa, sadece seçili alandaki kodlar çalıştırılacaktır.



Şekil 131: Query penceresi içerisinde sadece fare ile seçilmiş sorgular çalıştırılır

BÖLÜM 2: TEMEL VERİ İŞLEMLERİ

Veri Sorgulama (Select)

Tablolarda bulunan verilerin sorgulanacağı tüm işlemlerde **SELECT** ifadesi kullanılır. Veri tabanı üzerinde kullanılan SELECT ifadelerinin performanslı ve verimli bir şekilde çalışabilmesi için SELECT ifadesinin etkili şekilde nasıl kullanılabileceğini bilmekte fayda vardır.

Veri sorgulama işlemlerini yaparken SELECT ifadesi tüm seçeneklerinde FROM ile birlikte kullanılmaktadır. SELECT'ten sonra seçilecek olan alanlar sırasıyla yazılır. Bu ifade FROM ile birleştirilir ve FROM'dan sonra da verilerin çekileceği tablo ismi yazılır. Bir tablo içerisindeki tüm alanlar SELECT ifadesi ile seçilmek istenirse alan isimlerini tek tek yazmak yerine * işareti kullanılabilir. Aşağıda SELECT ifadesinin farklı kullanım şekillerine örnekler bulunmaktadır.

```
SELECT * FROM Ogrenciler
```

Bu ifade ile Ogrenciler tablosundaki bütün veriler tüm alan bilgileri ile birlikte elde edilmektedir. Sorgu cümlesi çalıştırıldığında aşağıdaki ekran görüntüsünde yer alan sonuçlar elde edilir.

OgrenciId	OgrenciAdSoyad	OgrenciDogumTarih	OgrenciBolumId	OgrenciEPosta	OgrenciSehir
128356	Kemal Dağlıoğlu	1982-12-20 00:00:00	101	NULL	Kayseri
128445	Yakup Şeref	1983-05-05 00:00:00	114	NULL	Nevşehir
128539	Refika Köseler	1983-06-26 00:00:00	102	refika@netron.com.tr	Ankara
128488	Oğuzhan Alaşehir	1982-06-13 00:00:00	102	NULL	İstanbul
129112	Elif Çakır	1983-02-14 00:00:00	101	e129112@metu.e...	İstanbul

Şekil 132: Sorgu sonucunda seçilen veriler

```
SELECT OgrenciId, OgrenciAdSoyad FROM Ogrenciler
```

Bu ifade ile Ogrenciler tablosundaki tüm verilerin sadece OgrenciId ve OgrenciSoyad bilgileri getirilmektedir. Sorgu cümlesi çalıştırıldığında, aşağıdaki gibi bir sonuç elde edilir.

OgrenciId	OgrenciAdSoyad
128356	Kemal Dağlıoğlu
128445	Yakup Şeref
128539	Refika Köseler
128488	Oğuzhan Alaşehir
129112	Elif Çakır

Şekil 133: Sorgu sonucunda seçilen veriler

Veri seçme işlemlerinde sadece gerekli olan alanların seçilmesi performans açısından oldukça önemlidir. Örneğin SELECT * FROM ile tablodaki tüm alanları seçmek yerine * ifadesinin yerine sadece gereken alanların seçilmesi, sorguların daha hızlı çalışmasını sağlayacaktır.

Tablolarda seçme, güncelleme ve silme işlemlerini yaparken, kısıtlama amacıyla Where ifadesi kullanılmaktadır. WHERE komutu ile sadece istenilen şartlara uyan verilere ulaşılabileceği için istenmeyen bilgilerle uğraşılmasına gerek kalmaz. Sonuç itibariyle, sadece istenen veri kümesi üzerinde çalışılarak daha performanslı bir çalışma planı

(Execution Plan) hazırlanmış olur. WHERE ifadesini karşılaştırma ve mantıksal operatörleri ele alarak daha etkin bir şekilde kullanabiliriz.

Aşağıdaki örnek senaryolarda Where kullanımına örnekler verilmiştir.

Bu kısma kadar Universite adında bir veri tabanı üzerinde çalışılmıştır. Bundan sonraki kısımlarda yer alan bazı örneklerin daha anlaşılır olması için, **Microsoft SQL Server 2005** ile birlikte gelen **AdventureWorks** isimli veri tabanı üzerinde çalışılacaktır.

Örnek: Person.Address tablosundan City değeri Seattle olan kayıtların AddressLine1, AddressLine2 ve PostalCode bilgileri aşağıdaki ifade yardımıyla seçilebilir.

```
SELECT AddressLine1,AddressLine2,PostalCode
FROM Person.Address WHERE City='Seattle'
```

Örnek: Purchasing.Vendor tablosundan ModifiedDate bilgisi 25 Şubat 2002 sonraki tarihlerde kayıtlı olan tüm kayıtlar seçilmektedir.

```
SELECT * FROM Purchasing.Vendor
WHERE ModifiedDate > '20022502'
```

Örnek: Production.Product tablosundan ListPrice değeri 120'ye eşit ve büyük olan ürünlerin Name, ProductNumber ve ListPrice bilgileri aşağıdaki sorgu ifadesi yardımıyla seçilebilir.

```
SELECT Name, ProductNumber, ListPrice
FROM Production.Product WHERE ListPrice >= 120
```

Örnek: Production.Product tablosundan ListPrice değeri 100'den küçük olan ve Class alanı NULL olmayan ürünlerin Name, ProductNumber, ListPrice, Class bilgileri aşağıdaki sorgu ifadesi yardımıyla seçilebilir.

```
SELECT Name, ProductNumber, ListPrice, Class
FROM Production.Product WHERE ListPrice < 100 AND Class IS NOT NULL
```

Örnek: Production.Product tablosundan ListPrice alanı 100 ile 200 arasında olan ürünlerin tüm bilgileri aşağıdaki sorgu ifadesi yardımıyla seçilebilir.

```
SELECT * FROM Production.Product
WHERE ListPrice BETWEEN 100 AND 200
```

WHERE ifadesi ile kullanılan eşitliklerin bazılarında aranan verinin **tırnak işaretleri** (**' '**) arasına alındığına dikkat edilmelidir. Sayısal alanlarda tırnak işareti kullanılmasına gerek olmazken, metinsel değer taşıyan alanlarda(char, varchar, text gibi) ve tarihsel değerleri taşıyan alanlarda (datetime, smalldatetime) şart içerisinde kullanılan değer tek tırnak işaretleri içerisine alınmalıdır. Nitekim, tırnak işaretinin kullanılmaması durumunda hatalar oluşacak ve sorgu ifadeleri çalışmayacaktır.

Where İfadesinin Kullanımı

WHERE ifadesi sorgulara belirli koşul veya koşullar getirerek kısıtlamalar yapmak amacıyla kullanılır. Örneğin bir tabloda veri çekme işlemi yaparken sadece belirli şartlara uyan verilerin getirilmesi isteniyorsa, WHERE ifadesi ile birlikte bir koşul belirtilmesi

gerekmektedir. Sorgu sonucunda sadece verilen koşula veya koşullara uyan veriler getirilecektir.

WHERE ifadesi SELECT, UPDATE ve DELETE ifadeleri ile birlikte kullanılabilir. WHERE ifadesinin genel kullanımı aşağıdaki gibidir.

```
[SELECT, UPDATE veya DELETE ifadelerinden biri] WHERE Koşul veya koşullar
```

Aşağıdaki örneklerde WHERE ifadesinin SELECT, UPDATE ve DELETE ifadeleri ile birlikte kullanarak sorgulamalara nasıl kısıtlamalar getirebileceği görülmektedir.

```
SELECT OgrenciAdSoyad, OgrenciSoyad FROM Ogrenciler
WHERE OgrenciBolumId = 102
```

Yukarıdaki sorgu ifadesinde, **Ogrenciler** tablosundan sadece 102 koduna ait bölümde kayıtlı olan öğrencilerin **OgrenciAdSoyad** ve **OgrenciEPosta** alanındaki bilgileri getirilmektedir. Sorgudaki **WHERE OgrenciBolumId = 102** ifadesi sadece **OgrenciBolumId** alanı **102** olan verilerin getirilmesini sağlamaktadır.

```
UPDATE Ogrenciler SET OgrenciEPosta = 'refika@netron.com.tr'
WHERE OgrenciId = 128539
```

Ogrenciler tablosundaki öğrenci numarası 128539 olan öğrencinin OgrenciEPosta alanı, yani e-posta adresi refika@netron.com.tr olarak güncellenmektedir.

```
DELETE FROM Bolumler WHERE BolumId=122
```

Bolumler tablosunda BolumId bilgisi 122 olan kayıt tablodan silinmektedir.

Karşılaştırma Operatörleri

WHERE ifadesi ile hazırlanan kısıtlamalarda karşılaştırma operatörleri kullanılmaktadır. Aşağıdaki tabloda karşılaştırma operatörleri bulunmaktadır.

Operatör	Anlamı
=	Eşit ise
<	Küçük ise
>	Büyük ise
<=	Küçük veya eşit ise
>=	Büyük veya eşit ise
<>	Eşit değil ise

Karşılaştırma operatörleri WHERE ifadesinden sonra gelen alan adı ile şart içerisinde kullanılacak değerin arasında kullanılır. Örnek kullanımı şu şekildedir.

```
WHERE Alan Adı [Karşılaştırma Operatörü] Değer
```

Mantıksal Operatörler

WHERE ifadesi ile birlikte birden fazla koşul yazılacağı durumlarda mantıksal operatörler kullanılmaktadır. Mantıksal operatörler koşullar arasında bağlama yaparak oluşacak mantıksal(logical) sonuca göre işlemler yapılmasını sağlar. Aşağıdaki tabloda WHERE ifadesi ile birlikte kullanılan mantıksal operatörler bulunmaktadır.

Operatör	Anlamı
AND	Her iki koşulun da sonucu DOĞRU(TRUE) ise ifade doğrulanır.
OR	Koşullardan sadece biri DOĞRU(TRUE) ise ifade doğrulanır.
NOT	Koşul YANLIŞ(FALSE) dönerse ifade doğrulanır.

AND Operatörünün Kullanımı

AND işlemi iki veya daha fazla koşul arasında kullanılabilir. Tüm koşulların doğru olması durumunda true sonucu döndürerek WHERE ifadesindeki durumun gerçekleşmesi sağlanmış olur.

```
SELECT * FROM Ogrenciler
WHERE OgrenciBolumId = 101 AND OgrenciSehir = 'Ankara'
```

Yukarıdaki ifadede Ogrenciler tablosunda OgrenciBolumId değeri 101 ve OgrenciSehir bilgisi Ankara koşullarına uyan öğrencilerin bütün bilgileri seçilmektedir. Yani, seçilen kayıtlarda hem OgrenciBolumId alanı 101 değerine, hem de OgrenciSehir bilgisi Ankara eşit olacaktır. İfade çalıştırıldığında aşağıdaki gibi bir sonuç elde edilecektir.

OgrenciId	OgrenciAdSoyad	OgrenciDogumTarih	OgrenciBolumId	OgrenciEPosta	OgrenciSehir
129116	Oğuz Özkavukçu	1983-07-12 00:00:...	101	e129116@metu.e...	Ankara
129303	Mustafa Uysal	1982-04-04 00:00:...	101	mustafauysal@ho...	Ankara
129455	Fatih Şahin	1983-11-12 00:00:...	101	e129445@metu.e...	Ankara

Şekil 134: Sorgu sonucunda seçilen veriler

OR Operatörünün Kullanımı

OR işlemi iki veya daha fazla koşul arasında kullanılabilir. Koşullardan herhangi birisinin doğru olması durumunda doğru bilgisini döndürerek WHERE ifadesindeki durumun gerçekleşmesi sağlanır.

```
SELECT * FROM Ogrenciler
WHERE OgrenciBolumId=101 OR OgrenciSehir='İstanbul'
```

Yukarıdaki ifadede Ogrenciler tablosundan, bulunduğu şehir Istanbul olan veya OgrenciBolumId değeri 101 olan öğrencilere ait bilgiler çekilmektedir. İfade çalıştırıldığında aşağıdaki gibi bir sonuç elde edilecektir.

OgrenciId	OgrenciAdSoyad	OgrenciDogumTarih	OgrenciBolumId	OgrenciEPosta	OgrenciSehir
128356	Kemal Dağlıoğlu	1982-12-20 00:00:0...	101	NULL	Kayseri
128488	Oğuzhan Alaşehir	1982-06-13 00:00:0...	102	NULL	İstanbul
129112	Elif Çakır	1983-02-14 00:00:0...	101	e129112@metu.ed...	İstanbul
129116	Oğuz Özkavukçu	1983-07-12 00:00:0...	101	e129116@metu.ed...	Kayseri
129303	Mustafa Uysal	1982-04-04 00:00:0...	101	mustafauysal@hotm...	Ankara
129455	Fatih Şahin	1983-11-12 00:00:0...	101	e129445@metu.ed...	Ankara
129498	Esma Umutluoğlu	1983-12-19 00:00:0...	102	e129498@metu.ed...	İstanbul

Şekil 135: Sorgu sonucunda seçilen veriler

NOT Operatörünün Kullanımı

NOT operatörü, yapılan işlemlerin tersini kontrol etme amacıyla **LIKE, IN, BETWEEN** gibi kelimelerle birlikte kullanılır. Aşağıdaki sorgu cümleleri NOT operatörünün kullanımlarına birer örnektir.

```
SELECT * FROM Bolumler WHERE BolumId NOT LIKE 101
```

Bu ifadede **Bolumler** tablosundaki **BolumId** bilgisi **101** olmayan tüm kayıtlar seçilmektedir. **WHERE NOT LIKE 101** ifadesi, getirilecek kayıtlarda BolumId bilgisi 101 olmayan kayıtları getir anlamına gelmektedir.

```
SELECT * FROM Ogrenciler
WHERE OgrenciSehir NOT IN ('İstanbul', 'Ankara')
```

Bu ifadede ise Ogrenciler tablosundan OgrenciSehir alanı İstanbul ve Ankara olmayan kayıtlar getirilmektedir. **WHERE OgrenciSehir NOT IN ('İstanbul', 'Ankara')** ifadesine göre, parantez içerisinde verilen şehirler dışındaki kayıtlar seçilecektir.

WHERE ifadeleriyle birlikte NOT operatörü kullanmak sorguyu yavaşlatacağı için zorunlu olmadıkça kullanılmaması tavsiye edilmektedir.

Arama İşlemleri

Veri sorgulama işlemlerinde bazı durumlarda bir alan içerisinde arama yapılarak veri seçilmek istenilebilir. Bir alan içerisinde arama işlemlerinin yapılması için WHERE ifadesi ile birlikte = operatörü yerine LIKE anahtar kelimesi kullanılmaktadır. Örneğin bir tabloda kişilerin yaşadıkları yer bilgisi ile ilgili olarak sadece ev adresleri tutulmakta ise, İstanbul şehrinde yaşayanları bulabilmek için adres alanı içerisinde İstanbul kelimesi geçen kayıtlar bulunmaya çalışılabilir. İşte böyle bir durumda LIKE kelimesi tam olarak istenilen işi yapacaktır. LIKE, metinsel alanlar(char, nchar, varchar, nvarchar, binary, varbinary) içerisinde arama yapmak için kullanılabilir. (SQL Server arama yapılan veri tipi eğer metine dönüştürülebilen bir tip ise bu veri tipine sahip bir alanda da LIKE ile arama yapabilir. Örneğin INT, DATETIME gibi alanlarda da LIKE ile arama yapılabilir) Kullanım açısından diğer bağlaçlardan farklı olarak aldığı parametreler içerisinde bazı karakterler kullanılarak çeşitli aramalar yapılabilir. Bu karakterler ile ilgili bilgiler aşağıdaki tabloda yer almaktadır.

Operatör	Kullanım amacı
%	Arama metni içerisinde bir veya birden fazla karakter anlamına gelir. Yani metin içerisinde % işaretinin yerine herhangi bir karakter veya karakter topluluğu gelebilir.
_	% işareti ile aynı işleve sahiptir. % işaretinden farklı olarak arama metni içerisinde sadece bir karakteri temsil etmektedir.
[]	Parantezler içerisinde belirtilen harfler arasındaki kayıtları getirir. Genellikle % ile birlikte kullanılır.
[^]	Parantezler içerisindeki karakterin geçmediği kayıtları getirir.

Tablo 18: Sql like için arama karakterleri

Kullanımı	Anlamı
WHERE AlanAdı LIKE 'ev%'	ev kelimesi ile başlayan kayıtlar. Örneğin; eve, evde, eve giderken, evli…
WHERE AlanAdı LIKE '%Ankara%'	İçerisinde Ankara kelimesi geçen kayıtlar
WHERE AlanAdı LIKE '_en'	3 karakterden oluşan, son 2 karakteri en ve ilk karakteri herhangi bir karakter olan kayıtlar. Örneğin; sen, ben, fen…
WHERE AlanAdı LIKE '[A-K]%'	A ile K arasındaki tüm harflerle (A ve K dahil) başlayan tüm kayıtlar. Örneğin; Ahmet, Burak, Esra, Kemal…
WHERE AlanAdı LIKE '[^A-T]%'	A ile T arasındaki harfler dışındaki harfler ile başlayan tüm kayıtlar. Örneğin; Uğur, Yakup, Zehra...

Tablo 19: Like için örnek sorgu ifadeleri

Aşağıdaki örnek sorgu cümlesinde, Production.Product tablosunda Name alanı içerisinde blue ile biten kayıtlar çekilmektedir.

```
SELECT * FROM Production.Product WHERE Name LIKE '%blue'
```

Aşağıdaki örnek sorgu cümlesinde, Production.Product tablosunda Class alanı K ile M harfi arasindaki harflerle başlayan kayıtlar seçilmektedir.

```
SELECT * FROM Production.Product WHERE Name LIKE '[K-M]%'
```

Sıralama İşlemleri (Order By)

Bazı durumlarda tablolardan seçtiğimiz verilerin belirli bir sıra ve düzen içerisinde olması istenebilir. Örneğin seçilen verilerin son kayıt tarihlerine göre veya herhangi bir alan içerisindeki alfabetik sıraya göre dizilerek ele alınması çoğu zaman istenilen bir durumdur. Tabloda bulunan bir alana göre sıralama işlemi yapılmak istendiğinde SELECT ifadesi ile birlikte **ORDER BY** kullanarak seçilen kayıtların bir veya birden fazla alana göre sıralanması sağlanabilir. ORDER BY ifadesinin örnek kullanımı şu şekildedir:

```
SELECT AlanAdı1, AlanAdı2 FROM TabloAdı ORDER BY SiralanacakAlanAdı
```

Bu ifade ile seçilen veriler SiralanacakAlanAdı alanına göre sıralanacaktır. ORDER BY artan veya azalan sıralama işlemleri için kullanılabilir. ORDER BY sözcükleri sıralama tipini belirlemek için ek takı olarak **ASC** (Ascending-artan) veya **DESC** (Descending-azalan) alabilir. Takı almadan kullanıldığında (yukarıdaki örnekte olduğu gibi) ASC takısı aldığı varsayılır. Alan adından sonra **ASC** ek takısı aldığı durumlarda veya herhangi bir ek takı almadığı durumlarda normal sıralama işlemi yapılacaktır. Normal sıralama işlemleri aşağıdaki biçimlerde ele alınmaktadır.

- Metinsel alanlarda A harfinden Z harfine doğru yapılır.
- Rakamsal alanlarda küçükten büyüğe doğru yapılır.
- Tarihsel alanlarda ise eski tarihten yeni tarihe doğru yapılır.

Bazı durumlarda sıralama işleminin bu durumların tersi olarak yapılması istenebilir. (Rakamsal bir alanın değerlerinin büyükten küçüğe doğru sıralanması gibi) Azalan sıralama işlemleri için ORDER BY ile birlikte **DESC** kelimesi kullanılmaktadır. ASC ve DESC kelimelerinin kullanımları şu şekildedir:

```
SELECT AlanAdı1, AlanAdı2 FROM TabloAdı ORDER BY SiralanacakAlanAdı ASC
SELECT AlanAdı1, AlanAdı2 FROM TabloAdı ORDER BY SiralanacakAlanAdı DESC
```

ORDER BY ile bir alana göre sıralama işlemleri yapılabildiği gibi birden fazla alana göre de sıralama yapılabilmektedir. **ORDER BY SiralanacakAlanAdı1, SiralanacakAlanAdı2** şeklinde bir kullanım ile seçilen veriler birden fazla alan içerisinde sıralanabilir. Böyle bir durumda söz konusu sorgu, seçilen verileri önce **SiralanacakAlanAdı1** alanına göre sıralayacaktır. Daha sonra sıralanan kayıtlardan sadece **SiralanacakAlanAdı1** değerleri birbirine eşit olan kayıtların içerisinde **SiralanacakAlanAdı2** alanına göre sıralama yapılacaktır.

```
SELECT OgrenciAdSoyad, OgrenciBolumId, OgrenciSehir FROM Ogrenciler
ORDER BY OgrenciBolumId, OgrenciSehir
```

Aşağıdaki ekran görüntüsünde, yukarıdaki sorgu cümlesinin sonucunda seçilen kayıtlar yer almaktadır. OgrenciBolumId bilgisi eşit olan kayıtların kendi aralarında OgrenciSehir alanına göre sıralandığına dikkat edilmelidir.

OgrenciAdSoyad	OgrenciBolumId	OgrenciSehir
Mustafa Uysal	101	Ankara
Elif Çakır	101	İstanbul
Oğuz Özkavukçu	101	Kayseri
Kemal Dağlıoğlu	101	Kayseri
Refika Köseler	102	Ankara
Muhammet Turşak	102	Ankara
Oğuzhan Alaşehir	102	İstanbul
Tayfun Akçay	114	İzmir
Yakup Şeref	114	Nevşehir

Şekil 136: Sorgu sonucunda seçilen veriler

Veri Tekrarlarını Önlemek (Distinct)

Yapılan arama işlemlerinde seçilen veri içerisinde aynı kayıttan birden fazla olabilmektedir. Böyle bir durumda birbirini tekrar eden kayıtların sadece birininin kullanılması istenebilir. Örneğin Ogrenciler tablosunda öğrencilerin yaşadıkları şehirlerin bir listesi elde edilmek istenebilir. **SELECT OgrenciSehir FROM Ogrenciler** şeklinde bir SQL sorgusu çalıştırıldığında tabloda kayıtlı olan tüm öğrencilerin şehirleri listelenecektir. Fakat seçilen kayıtların içerisinde aynı şehirlerin birden fazla sayıda tekrarlanması gibi istenilmeyen bir durumla karşılaşılacaktır. Böyle bir sorgu içerisinde **DISTINCT** kelimesi kullanılarak tekrar eden kayıtların teke indirilip her kayıttan bir tane seçilmesi sağlanabilir. DISTINCT kelimesinin örnek kullanımı şu şekildedir.

```
SELECT DISTINCT AlanAdı FROM TabloAdı
```

Az önceki senaryoda verilen örnek DISTINCT kelimesi olmadan ve DISTINCT kelimesi ile çalıştırılarak incelendiğinde aşağıdaki sonuçlar ile karşılaşılacaktır.

SELECT OgrenciSehir FROM Ogrenciler	SELECT DISTINCT OgrenciSehir FROM Ogrenciler

OgrenciSehir
Kayseri
Nevşehir
Ankara
İstanbul
İstanbul
Kayseri
Ankara
Ankara
İzmir

OgrenciSehir
Ankara
İstanbul
İzmir
Kayseri
Nevşehir

Şekil 137: DISTINCT ile yapılan sorgularda tekrarlanan iller tek sayıya inmiştir

Alan İsimlerini Değiştirme (Alias)

Tablolarda alanları isimlendirirken bazı kısıtlamalar olduğu belirtilmişti. Kelimeler arasında boşluk bırakmak ve Türkçe karakter kullanmak gibi durumlarda problemler çıkabileceği için isimlendirmeler yaparken kelimeleri birleştirerek kullanmak en güvenilir yoldur. Fakat bazı durumlarda sorgu sonucunda getirilen kayıtların alan isimlerinin değiştirilmesi istenebilir. Örneğin **OgrenciAdSoyad** şeklinde olan bir alan **Öğrenci Adı ve Soyadı** şeklinde değiştirmek istenebilir. Bu tip durumlarda SELECT ifadesi ile birlikte AS kelimesini kullanılabilir. **AS** kelimesi sorgu cümlelerinde getirilen alan isimlerinin sonuca farklı şekilde yansıtılmasını sağlar. Bir başka deyişle çekilen alanlara takma isimler (alias) verilebilmesini sağlar. Genel kullanımı şu şekildedir:

```
SELECT AlanAdı1 AS 'Yeni Alan İsmi-1', AlanAdı2 AS 'Yeni Alan İsmi-2'
FROM TabloAdı
```

Yukarıdaki kullanımdan da anlaşılacağı gibi ismi değiştirilecek alan adından sonra AS 'Yeni Alan İsmi' şeklindeki kullanım ile getirilecek veri kümesindeki alan isimleri değiştirilebilir. AS ifadesi bir veya birden fazla alanda kullanılabilir. Aşağıda AS kelimesinin örnek kullanımı yer almaktadır.

```
SELECT OgrenciId AS 'Öğrenci Numarası',
OgrenciAdSoyad AS 'Öğrenci Adı ve Soyadı', OgrenciSehir FROM Ogrenciler
```

Öğrenci Numarası	Öğrenci Adı ve Soyadı	OgrenciSehir
128356	Kemal Dağlıoğlu	Kayseri
128445	Yakup Şeref	Nevşehir
128539	Refika Köseler	Ankara
128488	Oğuzhan Alaşehir	İstanbul

Şekil 138: Sorgu sonucunda seçilen veriler

Sorgunun çalışmasıyla elde edilecek sonuç kümesindeki alan isimlerine dikkat edilirse; OgrenciId ve OgrenciAdSoyad alanları AS ile değiştirildiği için farklı, OgrenciSehir ise AS ile değiştirilmediği için aynı şekilde görüntülenecektir.

AS ifadesinin kullanımında tablonun yapısındaki alan ismine müdahalede bulunulmamaktadır. Yani tablodaki alan ismi değiştirilmemekte, sadece getirilen sonuç kümesi içerisindeki alan adı değiştirilmektedir.

Literal Kullanımı

SELECT ifadeleri ile sorgu sonucunda alan isimleri değiştirilebileceği gibi getirilen kayıtların içerikleri de değiştirilebilir. Bazı durumlarda verilere bilgi eklenmesi veya verilerin matematiksel işlemlere tabi tutulması gibi ihtiyaçlar olabilir. Bu tip gereksinimlerde, SELECT ifadeleri içerisinde matematiksel işlemlerde kullanılan operatörlerden faydalanılabilir.

İşaret	Kullanım Amacı	Örnek	
+	Toplama	AlanAdı1 + AlanAdı2	'Eklenecek Metin' + AlanAdı
-	Çıkarma	AlanAdı1 - AlanAdı2	
*	Çarpma	AlanAdı1 * AlanAdı2	
/	Bölme	AlanAdı1 / AlanAdı2	
%	Mod alma	AlanAdı1 % AlanAdı2	

Tablo 20: Sql aritmetik işlem operatörleri

Tablodaki tüm operatörler matematiksel işlemler için kullanılmaktadır. Yani kullanılacak alanların rakamsal değer taşıması gerekmektedir. Bununla birlikte, toplama operatörü (+) metinsel ifadeleri birleştirmek için de kullanılabilir. Bu operatörlerin SELECT ifadesi içerisindeki kullanımlarına aşağıdaki SQL cümlecikleri örnek verilebilir.

```
SELECT AlanAdı1 + AlanAdı2 FROM TabloAdı
SELECT AlanAdı1 * 250 FROM TabloAdı
SELECT (AlanAdı1 – AlanAdı2) * 30 FROM TabloAdı
SELECT 'Eklenecek Metin' + AlanAdı + 'Eklenecek Metin' FROM TabloAdı
```

Aşağıda, matematiksel operatörler kullanılarak seçilen kayıtların değiştirilmesi işleminin nasıl yapılacağına dair örnek bir SQL cümlesi yer almaktadır.

```
SELECT 'Zamlanan Ürün Ismi: ' + Name, ListPrice * 1.20
FROM Production.Product WHERE ListPrice < 70
```

Zamlanan Ürün Ismi: HL Road Frame - Red, 62	1717.800000
Zamlanan Ürün Ismi: HL Road Frame - Red, 44	1717.800000
Zamlanan Ürün Ismi: HL Road Frame - Red, 48	1717.800000
Zamlanan Ürün Ismi: HL Road Frame - Red, 52	1717.800000
Zamlanan Ürün Ismi: HL Road Frame - Red, 56	1717.800000
Zamlanan Ürün Ismi: HL Mountain Frame - Silver, 42	1637.400000
Zamlanan Ürün Ismi: HL Mountain Frame - Silver, 44	1637.400000

Şekil 139: Sorgu sonucunda seçilen veriler

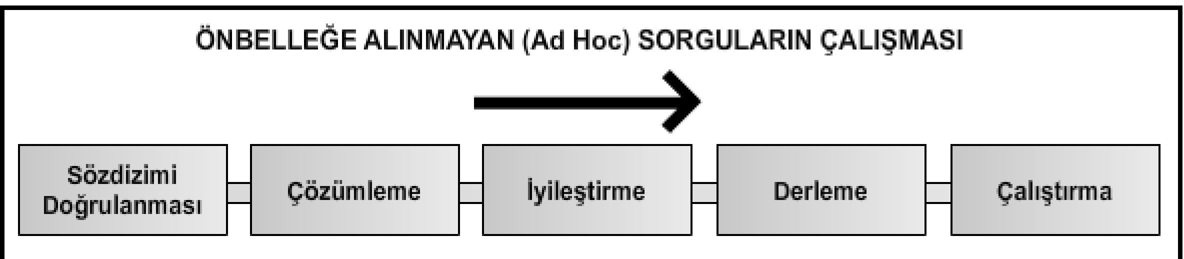
Sorgu çalıştırıldığında Name alanındaki ürün isimlerinin önüne "Zamlanan Ürün Ismi:" şeklindeki bir metin eklenir. Aynı zamanda ListPrice alanındaki veriler 1.20 ile çarpımlarının sonucunda %20 zamlı olarak elde edilirler.

SQL Server'da Sorguların Çalıştırılması

SQL Server üzerinde bütün sorgular, çalıştırılmadan önce aynı işlemlerden geçerler ve bu işlem adımlarının bazıları, bu iş için ayrılmış özel bir alanda depolanabilir. Bu işlem aynı sorguların bir daha çalıştırılması durumunda performans artışı sağlar.

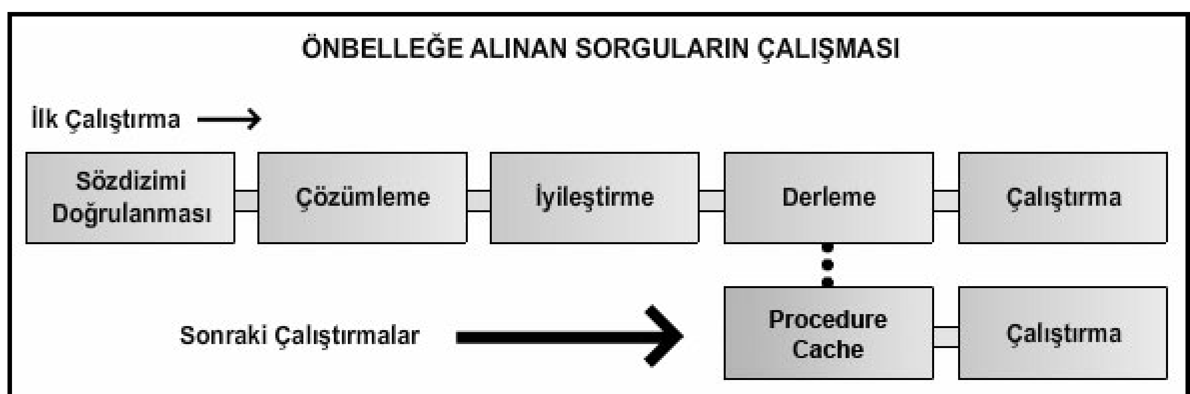
Bir SQL cümlesinin ilk çalıştırılmasında şu adımlar izlenir:

- **Sözdimi Doğrulaması(Parse):** Yazılan SQL sorgularında yazımsal hata olup olmadığının kontrol edilmesi.
- **Çözümleme(Resolve):** Yazılan nesnelerin doğrulunun kontrol edilmesi ve kodu çalıştıran kullanıcının yeterli yetkiye sahip olup olmadığının tespit edilmesi.
- **İyileştirme(Optimize):** İndekslere göre gerekli organizasyonlar ve iyileştirmelerin yapılması ve Where koşulunda belirtilen ifadelerin hangisinin önce çalışacağına karar verilmesi.
- **Derleme(Compile):** Yazılan kodun çalıştırılabilir hale getirilmesi.
- **Çalıştırma(Execute):** Derlenen kodun çalıştırılması ve sonuçların alınması.



Şekil 140: SQL Server'da önbelleğe alınmayan sorguların çalışırken izlediği yollar

İlk çalıştırma sonucunda önbelleğe alınan sorgular, sözdizimi doğrulaması, çözümleme ve iyileştirme aşamalarından geçmeden, daha önceden hazırlanmış çalışma planı (execution plan) procedure cache adı verilen kısımdan okunacağı için daha hızlı çalışacaktır. Aşağıda önbelleğe alınan bir sorgunun çalışma akışı gösterilmektedir.



Şekil 141: SQL Server'da önbelleğe alınan sorguların çalışırken izlediği yollar

Sorgulamalarda Performans İçin İpuçları

SQL Server üzerinde çalışırken performans kazanımları sağlayacak sorgularla çalışmak oldukça önemli bir hedeftir. Bu özellikle büyük veritabanları için önem arz eden bir konudur. Sorgu içerisinde kullanılan bazı yapılar, sorguları yavaşlatmakta ve sonuçların daha geç elde edilmesine sebep olmaktadır. Bunun en büyük nedeni sorguların çalışma sürelerinin uzaması ve SQL Server çalışma ortamının daha çok yorulmasıdır. Bu tip durumlardan kaçınmak için sorgular içerisinde dikkat edilecek bazı önemli hususlar şunlardır.

- **WHERE** ifadeleri içerisinde **NOT** kelimesini içeren koşullar sorguları yavaşlatacağı için zorunlu olmadığı durumlarda NOT kelimesi kullanılmamalıdır. (NOT IN, NOT BETWEEN, IS NOT NULL gibi ifadeler)
- **LIKE** kullanmadan da istenilen verilere ulaşılabilecek durumlarda WHERE ifadesi LIKE ile kullanılmamalıdır. Nitekim LIKE operatörü, eşitlik ifadelerine göre daha detaylı aramalar yapacağı için sorgular daha yavaş çalışacaktır.
- **ORDER BY** kelimeleri ile verileri sıralama işlemi sorguları yavaşlatacağı için sıralanmış verilere ihtiyaç olunmayan durumlarda ORDER BY kullanılmamalıdır.
- Veri seçme işlemlerinde aranılan şart tam olarak istenilen veriyi getirmelidir. İhtiyaç olmadığı halde genişletilen şartlar, seçilecek olan verilerin sayısını gereksiz yere arttıracağı için sorguların da yavaş çalışmasına sebep olacaktır.

Yine SQL sorguları ile çalışırken hem performans açısından, hem de çalışma zamanında SQL ortamını yormamak açısından aşağıda önerilen iyileştirmeler yapılabilir.

- Tablo, alan ve diğer nesnelere daha anlaşılır, kolay ve akılda kalıcı isimler verilmesi sorguların hazırlanmasında kolaylık sağlayacaktır.
- Alanlar tanımlanırken sadece gerektiği kadar yer tanımlaması yapılmalıdır. Örneğin ad-soyad veya e-posta bilgisinin tutulacağı bir alanı VARCHAR tipinden en fazla 40 karakter alacak şekilde tanımlamak yeterli olacaktır. Böyle bir alan için 200 karakterlik yer ayırmak kayıtların artması durumunda veri tabanını gereksiz yere büyütecektir.
- Sorgu sonuçlarında okunabilirliği arttırmak için AS anahtar kelimesinin kullanılması tercih edilebilir.

Veriyi Gruplamak

Tablolardan elde edilen veriler bazı durumlarda gruplanmak zorunda kalınabilir. Gruplanan veriler üzerinde grup bazında bazı bilgilere ulaşılıp, bu gruplar üzerinde bazı işlemler ve hesaplamalar yapılabilir. Örneğin müşterilerin kayıtlı olduğu bir tabloda şehre göre gruplandırma yapılarak, şehir bazında müşteri sayıları elde edilebilir.

SQL dili ile tablolardan seçilen veriler alan bazında gruplanabilir ve hatta belirli sayıda satır da veri tabanından grup halinde alınabilir.

Belirli Sayıdaki İlk Veriyi Seçmek

Tablolardan çekilen verilerin tamamına her zaman ihtiyaç duyulmayabilir. Örneğin belirli bir şartı sağlayan verilerin sadece istenilen kısmı alınmak istenebilir. Bu şekilde sorguların daha hızlı çalışması ve sadece ihtiyaç duyulan kayıtların elde edilerek gereksiz verilerle uğraşılmaması sağlanabilir. SQL dilinde bu işlem **TOP n** kelimesi ile yapılabilir. Buradaki **n** seçilecek kayıt sayısını temsil etmektedir. Belirli bir sayıda kayıt seçmek yerine, seçilecek toplam kaydın belirli bir oranının seçilmesi istenildiği durumlarda **PERCENT** ifadesi kullanılmaktadır. **TOP n PERCENT** şeklindeki kullanım sorgu sonucunda oluşacak toplam kaydın yüzde n kadarını getir anlamına gelecektir. TOP n ve TOP n PERCENT ifadelerinin genel kullanımı şu şekildedir:

```
SELECT TOP n AlanAdı1, AlanAdı2,... FROM TabloAdı
SELECT TOP n PERCENT AlanAdı1, AlanAdı2,... FROM TabloAdı
```

İlk ifadede (TOP n) n sayıda kayıt getirilirken, ikinci durumda ise tablodaki kayıtların % n kadarı getirilmektedir.

TOP ifadesi, **ORDER BY** ifadesi ile sıralanan kayıtların içerisinden belirli miktarda kayıt getirilmesi amacıyla sıklıkla kullanılmaktadır. Aşağıdaki örnekte ORDER BY ile sıralanan kayıtların içerisinden TOP ifadesi kullanılarak belirli sayıda kayıt getirilmiştir.

```
SELECT TOP 5 * FROM Production.Product ORDER BY ListPrice DESC
```

Sorgu sonucunda Production.Product tablosundan ListPrice bilgisi en yüksek olan ilk 5 kayıta ait tüm bilgiler getirilecektir. (* yerine alan adları yazılarak sadece istenilen alanlar da getirilebilir.)

Yukarıdaki örnekte şu soru ile karşılaşılabilir: *"Eğer sorgu sonucunda son sırada gelen kayıt ile aynı ListPrice değerine sahip başka kayıtlar da var ise bu kayıtlar da görüntülenir mi?".* TOP n ifadesi kullanıldığında son kayıtla aynı değeri taşıyan birçok kayıt varsa bile sadece n tane değer getirilecektir. (TOP n PERCENT içinde aynı durum geçerlidir) Yani yukarıdaki örnek için her durumda dönecek kayıt sayısı 5 olacaktır. Fakat **TOP n** ifadesi ile birlikte **WITH TIES** kelimelerinin kullanılması durumunda eğer son kayıt ile aynı değere sahip başka kayıtlar varsa, bu kayıtlar da sorgu sonucunda getirilecektir. **TOP n WITH TIES** ifadesinin örnek kullanımı aşağıdaki gibidir.

```
SELECT TOP 5 WITH TIES Name, ProductNumber, ListPrice
FROM Production.Product ORDER BY ListPrice DESC
```

Bu sorgu sonucunda eğer 5. sırada getirilen kaydın ListPrice değerine sahip başka kayıtlar var ise bu kayıtlar da sonuç kümesinde yer alacaktır.

WITH TIES ifadesi sadece ORDER BY ifadesi içeren sorgu cümlelerinde kullanılabilir.

Gruplama Fonksiyonları (Aggregate Functions)

T-SQL içerisinde tanımlı olan ve bazı görevleri yerine getiren yapılara fonksiyon denilir. SQL'de fonksiyonlar, C# vb. dillerdeki amaçlara benzer yapılardır. Fonksiyonlar, tekrarlı işlemlerin tek bir noktada toplanması ve yönetilebilirliğinin sağlanması gibi gereksinimlere cevap vermekle birlikte, geliştirme zamanında da daha çok ölçeklenebilirlik sağlar. SQL içerisinde önceden tanımlanmış ve farklı amaçlara (matematiksel, metinsel vb...) hizmet eden pek çok hazır fonksiyon bulunmaktadır. Söz konusu yardımcı metodlar içerisinde gruplama işlemlerinde sıklıkla kullanacağımız bazı fonksiyonlar da bulunmaktadır. Bu fonksiyonlar bir tablodaki tüm kayıtlar için veya gruplanan sonuç kümeleri için sıkça kullanılabilir. Tüm fonksiyonlar tek bir değer üretmektedir.

T-SQL'in yapısında bulunan temel gruplama fonksiyonları şunlardır:

- MIN(): **MIN(AlanAdı)** şeklinde kullanılmaktadır. Parametre olarak aldığı alan içerisindeki en küçük değeri bulur. Alan rakamsal veri taşıyorsa en küçük değeri, metinsel bir değer taşıyorsa alfabetik olarak en başta olan değeri bulur.

- MAX():**MAX(AlanAdı)** şeklinde kullanılmaktadır. MIN() fonksiyonunun tam tersine, parametre olarak aldığı alan içerisindeki en büyük değeri bulur.

- SUM(): **SUM(AlanAdı)** şeklinde kullanılmaktadır. Parametre olarak aldığı alana ait kayıtlı verilerin toplamını hesaplar. SUM() fonksiyonunun parametre olarak alacağı alan rakamsal bir değer taşımak zorundadır.

- AVG(): **AVG(AlanAdı)** şeklinde kullanılmaktadır. Parametre olarak aldığı alana ait kayıtlı verilerin toplamının aritmetik ortalamasını hesaplar.

- COUNT(): **COUNT(*)** ve **COUNT(AlanAdı)** şeklinde kullanılmaktadır. COUNT(*) ifadesinde tablodaki NULL değerler dahil tüm kayıtları sayar. COUNT(AlanAdı) şeklindeki bir kullanımda ise AlanAdı isimli alanda NULL değeri almamış olan tüm kayıtları sayar.

Bu fonksiyonların SELECT ifadesi ile örnek kullanımları aşağıdaki gibidir.

```
SELECT MAX(ListPrice) FROM Production.Product WHERE Class = 'M'
```

Production.Product tablosunda Class alanı M değerine eşit olan kayıtlar içerisindeki en yüksek ListPrice değerini getirir.

```
SELECT COUNT(*) FROM Production.Product
```

COUNT(*) ifadesi kullanıldığı için Production.Product tablosundaki toplam kayıt sayısını getirir.

```
SELECT COUNT(Class) FROM Production.Product
```

COUNT, Class isimli alan ile birlikte kullanıldığı için Production.Product tablosundaki kayıtlardan Class alanında NULL değer taşımayanların toplam sayısını getirir.

```
SELECT SUM(ListPrice), AVG(ListPrice)
FROM Production.Product WHERE Color = 'Red'
```

Production.Product tablosunda Color değeri Red olan tüm alanların ListPrice değerlerinin toplamını ve ListPrice değerlerinin ortalamasını getirir.

Alan Adına Göre Verileri Gruplamak (Group By)

SQL dilinde belirli sayıdaki veri gruplanıp sorgu sonucunda getirilebileceği gibi, tablodaki belirli alanlara göre gruplamalar yapılıp bu gruplar üzerinde işlemler yapılabilmektedir. Veri seçme işlemi yapılırken bazı durumlarda verileri gruplamak ve gruplanan veriler içerisinde bazı hesaplamalar ve matematiksel işlemler yapılması istenebilir. Örneğin müşterilerin kayıtlı olduğu bir tablodan müşterilerin yaşadıkları şehirlere göre gruplama yapılabilir, her şehirdeki toplam müşteri sayısı bulunabilir (COUNT fonksiyonu ile) veya bir şehirde yaşayan müşterilerimizin ortalama yaşı bulunabilir (AVG fonksiyonu ile). Gruplama fonksiyonu içeren bir SELECT sorgu cümlesinde **GROUP BY** ifadesini kullanarak alan ismine göre gruplama yapılabilir ve her grup içerisinde kullanılan gruplama fonksiyonuna göre bilgiler elde edilebilir. GROUP BY ifadesinin genel kullanımı şu şekildedir.

```
SELECT GruplanacakAlanAdı, GruplamaFonksiyonu1(AlanAdı1) FROM TabloAdı
GROUP BY GruplanacakAlanAdı
```

GROUP BY kullanıldığında, SELECT ifadesiyle birlikte sadece GROUP BY ile gruplanan alan adı ve gruplama fonksiyonlarından alınacak sonuçlar çekilebilir. SELECT ile birlikte birden fazla gruplama fonksiyonu kullanılabilir.

Bir tablodaki verileri gruplama fonksiyonu kullanmadan, bir alana göre görsel olarak gruplamak istersek kullanacağımız ifade GROUP BY yerine ORDER BY olmalıdır. GROUP BY sadece gruplama fonksiyonlarının olacağı SELECT sorgularında kullanılabilmektedir.

Aşağıdaki örneklerde GROUP BY ile veri gruplama işlemleri gösterilmiştir.

Örnek: Aşağıdaki sorgu cümlesi ile, Production.Product tablosundaki ürünlerin her renge ait ortalama ListPrice ve StandartCost değerleri getirilmektedir.

```
SELECT Color, AVG(ListPrice), AVG(StandardCost)
FROM Production.Product
GROUP BY Color
```

Örnek: Aşağıdaki sorgu cümlesi ile, Person.Address tablosunda her şehirde yaşayan kişilerin toplam sayıları bulunarak, veriler şehir adına göre alfabetik sırada elde edilmektedir.

```
SELECT City, COUNT(AddressID) AS Sayi FROM Person.Address
GROUP BY City
ORDER BY City
```

	City	Sayi
1	Abingdon	1
2	Albany	4
3	Alexandria	2
4	Alhambra	1
5	Alpine	1
6	Altadena	2
7	Altamonte Springs	1
8	Anacortes	3
9	Arlington	1
10	Ascheim	1
11	Atlanta	2
12	Auburn	1

Şekil 142: Sorgu sonucunda seçilen veriler

GROUP BY ile birden fazla alana göre gruplama yapılabilmektedir. GROUP BY AlanAdı1, AlanAdı2 şeklindeki bir kullanımda veriler önce AlanAdı1'e göre sonra da AlanAdı1'in içerisinde aynı değerlere sahip kayıtlar AlanAdı2'ye göre tekrardan gruplanır.

ÖRNEK: Ogrenciler tablosunda öğrencilerin okudukları bölümler içerisinde, yaşadıkları şehirlere göre rakamsal dağılımları görüntülenmek istensin. Ayrıca getirilecek olan kayıtların bölüm bilgilerine göre sıralanması da istensin.

```
SELECT OgrenciBolumId, OgrenciSehir, COUNT(*) FROM Ogrenciler
GROUP BY OgrenciBolumId, OgrenciSehir
ORDER BY OgrenciBolumId
```

Gruplanan Verilere Şart Eklemek (Having)

Veri gruplama işlemlerine bazı durumlarda sınırlamalar getirmek istenebilir. GROUP BY kullanıldığı durumlarda SELECT ile birlikte **WHERE** ifadesi kullanılarak gruplanacak verilere alan bazında sınırlamalar getirilebilir. Gruplama fonksiyonları sonucunda elde edilen değere göre filtreleme işlemi yapılmak istenildiğinde **HAVING** ifadesi kullanılmalıdır. Böyle bir amaç için SQL cümlemize HAVING COUNT(AlanAdı)>10 gibi bir ifade ilave edilmelidir. HAVING sadece gruplama fonksiyonlarını içeren şart ifadelerini

kullanabilir. **WHERE** ve **HAVING** ifadelerinin GROUP BY içerisindeki kullanımları şu şekildedir:

```
SELECT GruplanacakAlanAdı, GruplamaFonksiyonu1(AlanAdı1)
FROM TabloAdı
WHERE Şart veya Şartlar
GROUP BY GruplanacakAlanAdı
HAVING GrupFonksiyonu Şartı veya Şartları
ORDER BY SıralanacakAlanAdı
```

Bu kullanımda ifadelerin sıralamaları yukarıdaki gibi olmalıdır. Yani:

- WHERE ifadesi her zaman GROUP BY'dan önce kullanılmalıdır.
- HAVING ifadesi her zaman GROUP BY'dan sonra kullanılmalıdır.
- ORDER BY her zaman GROUP BY'dan sonra kullanılmalıdır.

Örnek: Production.Product tablosunda kırmızı veya mavi renkli ürünlerin Class bilgisine göre gruplanması, en az 5 kaydı olan Class kayıtlarının listelenmesi ve sonuçların Class adına göre sıralanması istendiğinde aşağıdaki sorgu cümlesi kullanılabilir.

```
SELECT Class, COUNT(ProductId) FROM Production.Product
WHERE Color = 'Red' OR Color = 'Blue'
GROUP BY Class
HAVING COUNT(ProductId) > 5
ORDER BY Class
```

Gruplanmamış Veriler İçerisinde Gruplama Fonksiyonları Kullanma (Compute)

Sadece veri seçme işlemlerinin yapıldığı bir sorgu içerisinde aynı zamanda gruplama fonksiyonları ile üretilen bir veriyi de alabilmek için **COMPUTE** ve **COMPUTE BY** ifadeleri kullanılabilir. Bu ifadeler seçilen verileri içeren sonuç kümesinden ayrı olarak bir de gruplama fonksiyonunun ürettiği sonucu içeren bir sonuç kümesi daha içermektedir. COMPUTE ifadesi tüm SELECT cümleleri ile birlikte kullanılabilirken, COMPUTE BY ifadesi **ORDER BY** içeren SELECT cümlelerinde kullanılabilmektedir. COMPUTE ve COMPUTE BY ifadelerinin örnek kullanımları aşağıdaki gibidir.

```
SELECT OgrenciId, OgrenciAdSoyad,OgrenciEPosta FROM Ogrenciler
COMPUTE COUNT(OgrenciEPosta)
```

Sorgu sonucunda seçilen veriler ayrı bir sonuç kümesi, COMPUTE ile getirilen veri ise ayrı bir sonuç kümesi olarak oluşturuldu.

OgrenciId	OgrenciAdSoyad	OgrenciEPosta
128356	Kemal Dağlıoğlu	NULL
128445	Yakup Şeref	NULL
128539	Refika Köseler	refika@netron.com.tr
128488	Oğuzhan Alaşehir	NULL
129112	Elif Çakır	e129112@metu.edu.tr
129116	Oğuz Özkavukçu	e129116@metu.edu.tr
129302	Muhammet Turşak	mtursak@hotmail.com
129303	Mustafa Uysal	mustafauysal@hotmail.com
129419	Tayfun Akçay	e129419@metu.edu.tr

cnt
6

COMPUTE ile çalıştırılan COUNT fonksiyonunun oluşturduğu veri ayrı bir sonuç kümesi olarak kayıtlara eklendi

Şekil 143: COMPUTE ifadesi seçilen sonuç kümesi ile birlikte gruplama fonksiyonunun ürettiği sonucu ayrı bir sonuç kümesi olarak getirmektedir.

```
SELECT OgrenciId, OgrenciBolumId, OgrenciSehir FROM Ogrenciler
ORDER BY OgrenciBolumId
COMPUTE COUNT(OgrenciId) BY OgrenciBolumId
```

Bu sorgu sonucunda ise Ogrenciler tablosundaki öğrencilerin bazı bilgileri seçilerek bölümlerine göre sıralanmaktadır. Sorgu sonucunda getirilen her bölüm için ikişer sonuç kümesi oluşur. İlk sonuç kümesini öğrencilerin seçilen bilgileri oluştururken, ikinci sonuç kümesini de o bölümde kaç kişi olduğunu vermektedir.

Farklı Tablolardan Veri Getirmek

İlişkisel veritabanlarında bir veri tabanı oluşturulurken, veri tekrarını azaltmak ve verileri daha verimli şekilde depolamak için kayıtlar farklı tablolarda toplanır. Bazı durumlarda birden fazla tabloda taşınan bilgilere ihtiyaç olabilmektedir. Bu durumlarda farklı iki tabloyu ifade içerisinde birleştirme işlemini (JOIN) gerçekleştirmek gerekir.

Şu ana kadar anlatılan SQL ifadeleri, sadece bir tablodan çeşitli şekillerde verilerin alınmasını sağlamaktadır. Birden fazla tabloyu birleştirmek, bu tablolardaki veriler üzerinde işlemler yapmak için T-SQL içerisinde çeşitli yapılar bulunmaktadır. Bu konu içerisinde birden fazla tablonun nasıl birleştirilebileceği ve bu tablolar üzerinde nasıl işlemler yapabileceği ele alınacaktır.

Birden Fazla Tabloyu Birleştirmek

Tablo birleştirme işlemi birden fazla tabloyu birleştirerek çalıştırılacak sorgunun bir sonuç kümesi olarak oluşmasını sağlar. Aşağıdaki örnek sorguda iki farklı tablonun basit bir şekilde nasıl birleştirilebileceği görülmektedir.

```
SELECT AlanAdı1, AlanAdı2 FROM Tablo1, Tablo2
```

AlanAdı1'in Tablo1'e, AlanAdı2'nin de Tablo2'ye ait alanlar olduğu düşünülebilir. Bu şekilde hazırlanacak bir sorgu cümlesiyle, iki farklı tablo birleştirebilir ve seçilen kayıtlar bir sonuç kümesi içerisinde toplanabilir.

Birleştirilen iki tabloda, tablolar arasında herhangi bir bağlantı ifadesi kullanılmazsa getirilecek sonuç her iki tablodan dönen verilerin kartezyen çarpımı olacaktır. (Yukarıdaki ifade herhangi iki tablo üzerinde denenecek olursa, seçilen verilerin iki ayrı tablodan dönen verilerin birbirleriyle eşleşmiş şekilde yeni bir sonuç kümesi oluşturduğu görülebilir.) Kayıtların bu şekilde eşlenmesi istenilen bir durum değildir ve bu nedenle ilişkilendirilen tablolar arasında çoğu zaman şartlar ve ilişkilendirmeler kullanılır.

Tabloların birleştirilmesi çeşitli ilişkilendirmeler ile gerçekleştirilebilir. Bu ilişkilendirmelerden en basit olanı **WHERE** ifadesinin kullanıldığı cümlelerdir. İki farklı tablodaki alanlar aynı tablo içerisindeymiş gibi WHERE ifadesi ile karşılaştırılabilir ve bu

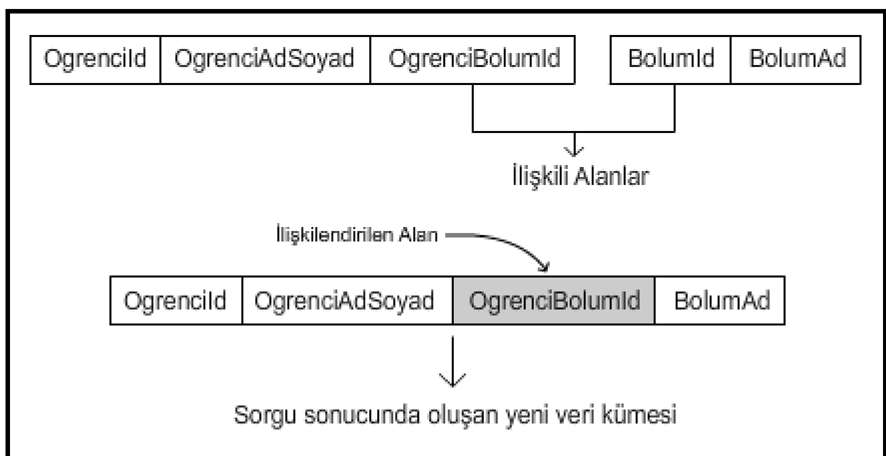
karşılaştırma sonucunda doğru sonuç elde edilen satırlardaki kayıtlar seçilebilir. WHERE ifadesiyle karşılaştırılan alanlar arasında foreign key (referans anahtar) ve primary key (birincil anahtar) ilişkisi olması, daha sağlıklı sonuçlar alınmasını sağlayacaktır. Aşağıda iki tablo arasında WHERE ifadesi kullanılarak nasıl ilişkilendirme yapılabileceği görülmektedir.

```
SELECT Ogrenciler.OgrenciAdSoyad, Bolumler.BolumAd
FROM Ogrenciler, Bolumler
WHERE Ogrenciler.OgrenciBolumId = Bolumler.BolumId
```

OgrenciAdSoyad	BolumAd
Kemal Dağlıoğlu	Bilgisayar Mühendisliği
Yakup Şeref	Fizik
Refika Köseler	Bilgisayar Öğretmenliği
Oğuzhan Alaşehir	Bilgisayar Öğretmenliği
Elif Çakır	Bilgisayar Mühendisliği
Oğuz Özkavukçu	Bilgisayar Mühendisliği
Muhammet Turşak	Bilgisayar Öğretmenliği
Mustafa Uysal	Bilgisayar Mühendisliği
Tayfun Akçay	Fizik

Şekil 144: Sorgu sonucunda seçilen veriler

Örnek sorguda Ogrenciler ve Bolumler tablosu birleştirilerek öğrencilerin isimleri ve kayıtlı oldukları bölümleri listelenmektedir. **WHERE Ogrenciler.OgrenciBolumId = Bolumler.BolumId** ifadesi ile Ogrenciler tablosundaki bölüm numarası ile Bolumler tablosundaki bölüm numarasının eşit olduğu yerler tek bir satırmış gibi ele alınarak değerlendirilir ve bir sonuç kümesi oluşturulur.



Şekil 145: İki tablodaki ilişkili alanların birleşmesinin şekil ile ifade edilmesi

Tablolara Temsili İsimler (Alias) Verme

Birden fazla tabloda birleştirme işlemlerinde bazı durumlarda karışıklıklar olabilmektedir. Örneğin iki farklı tabloda aynı alan isminin bulunması durumunda hangi alan üzerinde işlem yapılacağı karmaşaya sebep olacaktır. Böyle bir durumda **TabloAdı.AlanAdı** şeklindeki bir kullanım ile karışıklığa sebep vermeden işlemler yapılabilir. Fakat sorgu içerisinde birden çok alan yazılması gibi bir durumda sürekli alanların başına tablo adını yazmak bazen zahmetli bir iş olabilmektedir. Özellikle tabloların içerisinde bulunduğu şemalarda(Schema) devreye girdiğinde sorguların

okunması zorlaşacaktır. Bu tip durumlarda tablolara temsili olarak takma isimler (alias) verilerek sorgular oluşturabilir. Takma isim kullanmak karışıklıkları engellediği gibi yazılan sorguların okunabilirliğini de arttırmaktadır. Alias kullanımı **TabloAdı AS TakmaIsim** şeklinde olmaktadır. Tablo adı ile birlikte kullanılan **AS** ifadesinden sonra yazılan yeni isim sorgu içerisinde artık tabloyu temsil edecektir. Örneğin, *Production.Product* isimli tabloya, **Production.Product AS pr** şeklinde pr ismi alias verilirse, artık sorgu içerisinde **pr.Name** şeklinde bu tablonun alanları ifade edilebilir. Yukarıdaki örnekte kullanılan sorgunun alias kullanarak nasıl daha sade hale getirilebileceği, aşağıdaki örnek sorgu cümlesinde görülmektedir.

```
SELECT o.OgrenciAdSoyad, b.BolumAd
FROM Ogrenciler AS o, Bolumler AS b
WHERE o.OgrenciBolumId = b.BolumId
```

Join İfadeleri ile Tabloları Birleştirme

Farklı tabloları birleştirmek için kullanılan bir diğer yolda sorgu içerisinde **JOIN** ifadelerinin kullanılmasıdır. JOIN ifadelerinin bazı uygulamaları WHERE ile yapılan birleştirme işlemine benzese de daha farklı ve karmaşık kullanımları da bulunmaktadır. Tablo birleştirmede kullanılan JOIN ifadeleri aşağıdaki tabloda kısaca anlatılmıştır.

JOIN Türü	Açıklama
INNER JOIN	Birleştirilen iki tablodaki verilerden sadece aynı olanların getirilmesini sağlar. (JOIN ifadesi de INNER JOIN ile aynı işi yapar.)
OUTER JOIN	Birleştirilen iki tablodan verilerin bir tabloda olması durumunda getirilmesini sağlar. LEFT JOIN, RIGHT JOIN ve FULL JOIN ifadeleri ile getirilecek verilerin hangi tabloda olacağı seçimi yapılabilir.
CROSS JOIN	Birleştirilen tablolardan seçilen veriler arasındaki tüm kombinasyonları getirir.

Tablo 21: Join seçenekleri

INNER JOIN ile Tabloları Birleştirme

INNER JOIN, bağlanan tablolarda ortak olan alanları kontrol ederek her iki tabloda da eşleşen kayıtların getirilmesini sağlar. INNER JOIN ifadesinin kullanım şekli şöyledir:

```
SELECT Seçilecek Alan
FROM Tablo1 INNER JOIN Tablo2 ON İlişkilendirme Şartı
```

INNER JOIN birleştirilecek olan tabloların arasında yer alır. Bu şekilde sorgu cümlesine 2 tabloyu birleştirileceği söylenir. Bu iki tablonun ilişkilendirileceği alanlar arasında nasıl bir şart arandığını ise ON ifadesi belirler. ON ifadesinin kullanımı WHERE ifadesindeki gibidir.

Bir işyerinin veri tabanında, **Calisanlar** isimli tabloda çalışanların ismi ve görev kodu, **Gorevler** tablosunda da görev kodu ile görevlerin isimleri tutulsun. Buradaki amaç, iki tabloyu birleştirip çalışanlardan sadece görevleri tanımlı olanlarını bulmak ve çalışan isimleri ile görevlerini bir sonuç kümesinde toplamak olabilir. Böyle bir durumda iki tabloyu, ilişkili olan alanları aracılığıyla birleştirip her iki tarafta da var olan kayıtların getirilmesi istenilebilir.

Calisanlar
Isim
GorevId

Gorevler
GorevId
GorevIsim

Calisanlar tablosundaki **GorevId** ile **Gorevler** tablosundaki **GorevId** birbiri ile ilişkili alanlardır.

Isim	GorevId
Ahmet	15
Mehmet	16
Serpil	14
Murat	12
Sema	14

GorevId	GorevIsim
11	Genel Müdür
12	Proje Yöneticisi
13	Yazılım Uzmanı
14	Grafiker
15	Programcı

İki tablo birleştirilip GorevId alanları eşleştirilirse aşağıdaki gibi bir sonuç ortaya çıkar. Calisanlar tablosundaki GorevId bilgisi, Gorevler tablosundaki GorevId bilgilerinden birine uymayan bir satırda GorevIsim kısmında NULL bir değer gelecektir.

Ahmet	15	Programcı
Mehmet	*16*	*NULL*
Serpil	14	Grafiker
Murat	12	Proje Yöneticisi
Sema	14	Grafiker

Yukarıda anlatılan senaryo, INNER JOIN ifadesi kullanılarak şu şekilde bir sorgu cümlesine çevrilebilir.

```
SELECT c.Isim, g.GorevIsim
FROM Calisanlar AS c INNER JOIN Gorevler AS g
ON c.GorevId = g.GorevId
```

Sorgu çalıştırıldığında oluşacak sonuç kümesi şu şekilde oluşacaktır. Mehmet isimli çalışanın görevinin, Gorevler tablosunda karşılığı olmadığı için bu kayıt sonuç kümesine eklenmeyecektir.

Isim	GorevIsim
Ahmet	Programcı
Serpil	Grafiker
Murat	Proje Yöneticisi
Sema	Grafiker

Şekil 146: Sorgu sonucunda oluşacak sonuç kümesi

INNER JOIN ile birlikte WHERE, ORDER BY gibi ifadeler kullanılarak getirilecek olan kayıtlar daha da özelleştirilebilir. Yukarıda yapılan işlemde getirilecek kayıtlara, GorevId alanı 12 değerinden büyük olan kayıtlar getirilsin şeklinde bir kısıtlama WHERE ifadesi kullanılarak aşağıdaki örnekte olduğu gibi yapılabilir.

```
SELECT c.Isim, g.GorevIsim
FROM Calisanlar AS c INNER JOIN Gorevler AS g
ON c.GorevId = g.GorevId
WHERE c.GorevId > 12
```

Bu tip sorgularda **INNER JOIN** ifadesi yerine sadece **JOIN** de kullanılabilmektedir. Bu iki ifade kullanım ve çalışma bakımından aynı işleve sahip olmakla birlikte T-SQL'de genellikle INNER JOIN kullanımı tercih edilir. Bu daha çok JOIN işleminin çeşidini açık bir şekilde (explicitly) belirtmek amacıyla yapılır.

OUTER JOIN ile Tabloları Birleştirme

OUTER JOIN ifadeleri, INNER JOIN'den farklı olarak tabloların iki tarafında da bir eşleşme olmasına gerek kalmaksızın herhangi bir tablodaki tüm satırları getirmeye yarar. OUTER JOIN'in üç farklı kullanım şekli bulunmaktadır: **LEFT OUTER JOIN**, **RIGHT OUTER JOIN** ve **FULL JOIN**.

- LEFT JOIN ile birleşen tablolardan sorgu içerisinde LEFT JOIN'in sol kısımda kalan tablonun tüm kayıtları,
- RIGHT JOIN ile birleşen tablolardan sorgu içerisinde RIGHT JOIN'in sağında kalan tablonun tüm kayıtları,
- FULL JOIN ile birleşen tablolardan her ikisindeki tüm kayıtlar getirilir.

İki tablodaki kayıtların eşlenerek oluşturulan sonuç kümesinde diğer tabloda karşılığı olmayan kısım NULL olarak getirilir.

"**OUTER JOIN**" ifadeleri sorgu içerisinde kullanılmaz. OUTER JOIN ifadelerinde yukarıda bahsedilen **LEFT OUTER JOIN**, **RIGHT OUTER JOIN** ve **FULL JOIN** kelimeleri kullanılır.

OUTER JOIN ifadelerinin genel kullanımı şu şekildedir.

```
SELECT Seçilecek Alanlar
FROM Tablo1 [ LEFT JOIN, RIGHT JOIN veya FULL JOIN ] Tablo2
ON İlişkilendirme Şartı
```

Yukarıda INNER JOIN için yazılan örnek OUTER JOIN ifadeleri ile kullanıldığında hem INNER JOIN ve OUTER JOIN arasındaki fark, hem de LEFT OUTER JOIN, RIGHT OUTER JOIN ve FULL JOIN ifadeleri arasındaki farklılıklar analiz edilebilir.

```
SELECT c.Isim,
g.GorevIsim
FROM Calisanlar AS c
LEFT OUTER JOIN
Gorevler AS g
ON c.GorevId=g.GorevId
```

Isim	GorevIsim
Ahmet	Programcı
Mehmet	NULL
Serpil	Grafiker
Murat	Proje Yöneticisi
Sema	Grafiker

LEFT OUTER JOIN ifadesinin sol kısmındaki Calisanlar tablosundan tüm kayıtlar gelmektedir. Sağ tablodan eşlenemeyen verilere NULL değeri atanır.

```
SELECT c.Isim,
g.GorevIsim
FROM Calisanlar AS c
RIGHT OUTER JOIN
Gorevler AS g
ON c.GorevId=g.GorevId
```

Isim	GorevIsim
NULL	Genel Müdür
Murat	Proje Yöneticisi
NULL	Yazılım Uzmanı
Serpil	Grafiker
Sema	Grafiker
Ahmet	Programcı

RIGHT OUTER JOIN ifadesinin sağ kısmındaki Gorevler tablosundan tüm kayıtlar gelmektedir. Sol tablodan eşlenemeyen verilere NULL değeri atanır.

```
SELECT c.Isim,
g.GorevIsim
FROM Calisanlar AS c
FULL JOIN Gorevler AS g
ON c.GorevId=g.GorevId
```

Isim	GorevIsim
Ahmet	Programcı
Mehmet	NULL
Serpil	Grafiker
Murat	Proje Yöneticisi
Sema	Grafiker
NULL	Genel Müdür
NULL	Yazılım Uzmanı

FULL JOIN ifadesi ile tablolar arasında kayıt eşleşmesi dikkate alınmadan tüm kayıtlar gelmektedir. Diğer tablodan eşlenemeyen verilere NULL değeri atanır.

Şekil 147: OUTER JOIN ifadelerinin ürettiği farklı sonuçlar

Üç farklı OUTER JOIN işleminin sonucundan da anlaşılacağı gibi, INNER JOIN ifadeleri sadece eşleşen verileri getirirken, OUTER JOIN ifadeleri eşleşme olmasa bile sadece bir tablodaki veya her iki tablodaki tüm verileri sonuç kümesine eklemektedir.

CROSS JOIN ile Tabloları Birleştirme

Tablolar arasında yapılan birleşmelerde, seçilen tüm verilerin birbiriyle kartezyen çarpımı şeklinde eşleşmesini sağlayan ifadelerdir. Veritabanlarında pek kullanılmayan bir yöntemdir. Seçilen verilerin çok sayıda olması durumunda tüm verilerin kendi aralarındaki tüm kombinasyonları sonuç kümesine ekleneceği için çok sayıda satırdan oluşabilir. Genel kullanımı şu şekildedir.

```
SELECT Seçilecek Alanlar FROM Tablo1 CROSS JOIN Tablo2
```

Aşağıdaki sorgu cümlesiyle, Calisanlar ve Gorevler tablosundan CROSS JOIN ifadesi ile çalışanların ve görev isimlerinin eşleştirilmesinin nasıl bir sonuç oluşturacağı gösterilmektedir.

```
SELECT c.Isim, g.GorevIsim
FROM Calisanlar AS c CROSS JOIN Gorevler AS g
```

Isim	GorevIsim
Ahmet	Genel Müdür
Mehmet	Genel Müdür
Serpil	Genel Müdür
Murat	Genel Müdür
Sema	Genel Müdür
Ahmet	Proje Yöneticisi
Mehmet	Proje Yöneticisi
Serpil	Proje Yöneticisi
Murat	Proje Yöneticisi
Sema	Proje Yöneticisi
Ahmet	Yazılım Uzmanı
Mehmet	Yazılım Uzmanı
Serpil	Yazılım Uzmanı

Isim	GorevIsim
Murat	Yazılım Uzmanı
Sema	Yazılım Uzmanı
Ahmet	Grafiker
Mehmet	Grafiker
Serpil	Grafiker
Murat	Grafiker
Sema	Grafiker
Ahmet	Programcı
Mehmet	Programcı
Serpil	Programcı
Murat	Programcı
Sema	Programcı

Şekil 148: Sorgu sonucunda seçilen veriler

Görüleceği gibi her iki tablodan da seçilen tüm veriler birbiriyle eşlenmiş ve sonuç kümesinde toplam 25 kayıt oluşmuştur. (Calisanlar tablosundaki 5 kayıt x Gorevler tablosundaki 5 kayıt = Toplam 25 kayıt) Eğer tablolardaki veriler daha fazla olsaydı sonuç binlerce, on binlerce satırdan oluşabilirdi. Bu nedenle CROSS JOIN veritabanlarında tercih edilmeyen bir ifade türüdür. Bu kullanıma şöyle bir senaryo örnek olarak verilebilir; ürünlere ait renk, model ve beden tablolarının olduğu kabul edilsin. Belli bir modeldeki ürünün, ilgili renk veya bedende üretilip üretilmediğine bakılmaksızın olası tüm alternatiflerin elde edilmek istendiği bir durumda CROSS JOIN kullanımı tercih edilebilir.

İç İçe Sorgular (Subquery)

Veri tabanı üzerinde bazı işlemler yapılırken çok karmaşık ve uzun sorgular kullanmak zorunda kalınailir. Yine bu işlemler yapılırken farklı tablolardan veya durumlardan alınacak sonuçlara göre bir sorgular çalıştırılması gerekebilir. İç içe sorgular kullanarak bu tip işlemlerin aşama aşama yapılması sağlanabilir. Bir sorgulama işlemi, diğer sorgulama işleminin sonucunu kullanabilmektedir. İç içe sorgularda, asıl sorguya sonuç getiren iç sorgu bazen tek değer üretebilirken, bazı durumlarda da birden fazla değer üretebilmektedir.

Asıl sorgu, bir alt sorgunun getirdiği tek sonuç ile işlem yapacaksa, genelde MAX, MIN, AVG gibi gruplama fonksiyonları veya tek değer döndüren SELECT ifadeleri kullanılmaktadır. WHERE ifadesinden sonraki şartda bir alt sorgudan gelen sonuç kullanılarak karşılaştırma yapılır. Tek değer döndüren bir sorgunun SELECT ifadesi içerisinde örnek kullanımı aşağıda gösterilmiştir.

```
SELECT * FROM Tablo
WHERE Alan > (SELECT AVG(Alan) FROM Tablo)
```

Aşağıdaki sorgu Production.Product tablosundaki ListPrice değeri en yüksek olan ürünü veya ürünleri getirir.

```
SELECT Name, Color, ListPrice FROM Production.Product
WHERE ListPrice = (SELECT MAX(ListPrice) FROM Production.Product)
```

İç kısımdaki sorgu Production.Product tablosundaki en yüksek ListPrice değerini getirecektir. Önce iç kısımdaki sorgu çalıştırılıp tablodaki en yüksek ListPrice değeri

üretilir. Daha sonra bu değer WHERE ifadesindeki şart içerisinde kullanılarak sorgu çalıştırılır. Böylece tablo içerisindeki en yüksek fiyata sahip ürün veya ürünler sorgu sonucunda getirilir.

```
SELECT Name, Color, ListPrice FROM Production.Product
WHERE ListPrice = (SELECT MAX(ListPrice) FROM Production.Product)
                          ↓
                  3578,27 değeri getirildi
                    Sorgu şu hale geldi
                          ↓
SELECT Name, Color, ListPrice FROM Production.Product
WHERE ListPrice = 3578,27
```

Şekil 149: Max gruplama fonksiyonu için örnek kullanım

Bir sorgu içerisinde başka bir sorgu tek değer üretebildiği gibi birden fazla değerde üretebilir. Bu tip durumlarda WHERE ile birlikte **IN** ifadesi kullanılarak alt sorgunun getirdiği değerlerin hepsi işleme tabi tutulur. Aşağıdaki örnekte bir SELECT ifadesi, içerisinde kullanılan alt sorgudan getirilen birden fazla değer kullanılarak bir sonuç kümesi oluşturulmaktadır.

```
SELECT OgrenciAdSoyad, OgrenciBolumId FROM Ogrenciler
WHERE OgrenciBolumId
IN (SELECT BolumId FROM Bolumler WHERE BolumAd LIKE 'B%')
```

```
SELECT OgrenciAdSoyad, OgrenciBolumId FROM Ogrenciler
WHERE OgrenciBolumId IN (SELECT BolumId FROM Bolumler WHERE BolumAd LIKE 'B%')
                                   ↓
  101, 102 ... gibi bölüm adı B harfi ile başlayan bölümlerin BolumId bilgileri geldi
                            Sorgu şu hale geldi
                                   ↓
SELECT OgrenciAdSoyad, OgrenciBolumId FROM Ogrenciler
WHERE OgrenciBolumId=101 OR OgrenciBolumId=102 OR ....
```

Şekil 150: In anahtar sözcüğünün örnek kullanımı

İç kısımdaki sorgu sonucunda getirilen değerler dıştaki SELECT ifadesinde kullanılarak bölüm adı B harfi ile başlayan bölümlerde okuyan öğrenciler listelenmiştir.

İç içe sorgulamalar SELECT ifadeleri dışında INSERT, UPDATE ve DELETE komutları ile birlikte de kullanılabilir. Örnek: **DELETE FROM Tablo WHERE Alan = (SELECT MAX(Alan) FROM Tablo)** gibi.

İç içe sorgulamalar kullanmak yerine bazı durumlarda JOIN ifadeleri kullanılabilir. JOIN ifadelerinden yararlanılarak oluşturulabilen sorgular için, içiçe sorgular kullanılması tercih edilmemelidir. Çünkü JOIN ifadeleri, içiçe sorgulardan performanslı çalışırlar.

Exists ve Not Exists İfadelerinin Kullanımı

WHERE ile IN ifadesinin kullanımına benzer olarak, **EXISTS** ve **NOT EXISTS** ifadeleri de alt sorgudan getirilen değerlerin içerisinde bir değerin olması veya olmaması durumunda işlem yapılmasını sağlar. EXISTS ifadesi kullanıldığında alt sorguda istenilen şartların yerine getirildiği durumlarda üstteki sorgu değer üretir. NOT EXISTS ise

EXISTS'in tam tersi olarak alt sorguda istenilen şartların sağlanmadığı durumlarda üstteki sorgu değer üretir.

Örnekte EXIST ifadesinin örnek kullanımı yer almaktadır.

```
SELECT OgrenciAdSoyad FROM Ogrenciler AS o WHERE EXISTS
(SELECT * FROM Bolumler AS b WHERE o.OgrenciBolumId = b.BolumId)
```

Veriyi Güncellemek

T-SQL'de Transactionlar

Şu ana kadar üzerinde durulan SQL ifadeleri bir tablodan kayıt getirmek gibi basit işlemleri içeriyorlardı. Bazı durumlarda veri tabanı üzerinde ardı ardına gelen işlemler yapılması istenebilir. Örneğin bir tablodan bir kayıt silindiğinde, bu silme işlemine bağlı olarak başka bir tablodaki bazı kayıtlar değiştirilmesi hatta silinmesi gerekebilir. Bu işlemlerin bir bütünlük gerektirdiği durumlar söz konusu olabilir. Yani sorgulardan birinin başarılı bir şekilde çalışması ancak başka bir sorgunun başarısız olması veri bütünlüğünü ya da iş mantığını bozabilir. Böyle bir durumda, yapılacak tüm işlemlerin tek bir işlem gibi düşünülmesi ve işlemlerden herhangi birinin gerçekleşmemesi durumunda diğer tüm işlemlerinde iptal edilmesi gerekebilir. Daha küçük parçalara ayrılamayan iş parçacıklarına **transaction**(işlem bloğu) ismi verilmektedir. Örneğin, bir banka veri tabanında, bir kullanıcıdan(Ahmet) diğer bir kullanıcının(Ayşe) hesabına para havale edileceği düşünülsün. Bu havale işlemi tek bir işlem içerisinde gerçekleştirilecektir. Yani, Ahmet isimli kullanıcının hesabındaki para miktarının eksiltilip, Ayşe isimli kullanıcının hesabındaki para miktarının arttırılması gerekecektir. Böylece havale işleminin gerçekleştirilmesi tek bir olay içerisinde ele alınacaktır. Olasılıklar analiz edilecek olursa, Ahmet isimli kullanıcının hesabında yeterli miktarda para olmaması durumunda, Ahmet'in hesabı eksiye düşecek ve Ayşe'nin hesabına para aktarılması ihtimali bulunmaktadır. Bu istemeyek bir durumdur. SQL Server'da bu işlem şu şekilde ele alınabilir. Önce ilk sorgu çalıştırılır; yani Ahmet isimli kullanıcının hesabından havale miktarını düşecek bir Update sorgusu çalıştırılır. Eğer Ahmet'in hesabında yeterli miktar var ise işlem başarıyla yapılacak ve bu sorguya bağlı olarak diğer sorgularda çalıştırılacaktır. Sonuç olarak para Ayşe'nin hesabına aktarılacaktır. İşlemin başarıyla sonuçlanması durumunda **transaction commit** edilir. Fakat Ahmet'in hesabında yeterli miktar yok ise ilk işlem gerçekleşmeyeceği için, ilk işleme bağlı olarak diğer işlemler de geri alınacaktır. Bunu sağlamak için **transaction rollback** edilir. Bu şekilde yapılabilecek hatalı işlemlerin önüne geçileceği gibi, veri tutarlılığı ve bütünlüğü de sağlanır.

SQL Server'da tek başına çalışabilen SELECT, INSERT, UPDATE, DELETE gibi komutlar da varsayılan olarak bir transaction içerisinde çalışmaktadır. Komut çalışırken bir hata oluşursa işlem geri alınır.

SQL Server'da yapılacak transaction işlemleri olabildiğince kısa tutulmalıdır. Aynı anda, aynı tablolar üzerinde birden fazla transaction talebinde bulunulursa, transction'lardan biri diğerini bekleyeceği için işlemin sonuçlandırılması daha fazla zaman alacaktır.

Veri Ekleme (Insert)

Bir tabloya yeni veri ekleme işlemi INSERT sorguları ile yapılmaktadır. INSERT sorguları, INTO (opsiyonel) ve VALUES ifadeleri ile birlikte kullanılarak tablolara yeni veri ekleme işlemlerini gerçekleştirir. Eklenecek veriler VALUES ifadesinden sonra gelen ( ) işaretlerinin içerisinde aralarında virgül olacak şekilde yazılır. INSERT ifadesinin örnek kullanımı şu şekildeydi.

```
INSERT INTO Ogrenciler (OgrenciId, OgrenciAdSoyad, OgrenciBolumId)
VALUES (128535, 'Uğur Umutluoğlu', 102)
```

Veri ekleme işlemlerinde öncelikli olarak **INSERT INTO Tablo** tanımlaması hangi tabloya veri ekleneceğini ifade eder. Tablo adından sonra eğer tablonun tüm alanlarına değil de, belirli alanlarına veri eklenecekse parantez içinde bu alanların isimleri yazılır. Daha sonra VALUES ifadesi ile eklenecek değerler parantez içinde belirtilir. Buradaki en önemli kriter verilerin sıralanışıdır. (OgrenciId, OgrenciAdSoyad, OgrenciBolumId) şeklinde sıralanmış alanlara uygun bir şekilde veri ekleyebilmek için, yine aynı sırayla eklenecek değerler yazılmalıdır.

Kayıt eklenecek tabloda eğer tüm alanlara değer girilecekse, tablo adından sonra alan isimlerinin yazılmasına gerek yoktur. Bu durumda SQL Sorgusu, tüm alanlara veri gireceğimizi varsayıp VALUES'tan sonraki parantez içindeki tüm alanlara sırasıyla veri eklememizi bekleyecektir.

Column Name	Data Type	Allow Nulls
OgrenciId	int	☐
OgrenciAdSoyad	varchar(40)	☑
OgrenciDogumTarih	smalldatetime	☑
OgrenciBolumId	int	☑
OgrenciEPosta	varchar(30)	☑
OgrenciSehir	nvarchar(40)	☑

```
INSERT INTO Ogrenciler VALUES (128535, 'Uğur Umutluoğlu', 102)

INSERT INTO Ogrenciler
VALUES ('Uğur Umutluoğlu', 128535, '1982-3-1', 102, 'ugur@nedirtv.com', 'İstanbul')
```

Şekil 151: Ogrenciler tablosunun yapısı ve bu tablo üzerinde başarısız olacak INSERT ifadelerine iki örnek

Yukarıda Ogrenciler tablosunun alanları ile alan tipleri yer almaktadır. Hemen alt kısmında ise bu tabloya, tablo içerisindeki alanlar yazılmaksızın veri eklenmeye çalışılan iki farklı sorgu bulunmaktadır. Buradaki iki sorgu cümlesiyle de veri eklenmeye çalışıldığımında hata alınır ve sorgular çalışmaz. İlk sorgu eksik parametreler olduğu için, ikinci sorgu girilen değerler tablodaki veri tipleri ile uyuşmadığı için çalışmayacaktır. Her iki hatada, VALUES (...) kısmındaki değerler doğru yazılmadığı için oluşmaktadır. Aşağıda INSERT ifadesinin tablodaki alan isimleri bildirilmeden nasıl kullanılacağı gösterilmektedir.

```
INSERT INTO Ogrenciler VALUES
(128535,'Uğur Umutluoğlu','1982-3-1',102,'ugur@nedirtv.com','İstanbul')
```

Eğer tablodaki bir alan NULL değer alabiliyorsa INSERT işlemi yapılırken bu kısımlar aşağıdaki gibi bir kullanımla boş bırakılabilir. NULL değer alamayan alanlara bir değer girmek zorundayız.

```
INSERT INTO Ogrenciler
VALUES (128544, 'Ensar MENGİ', NULL, NULL, NULL, 'İstanbul')
```

Kullanımına dikkat edilecek olursa boş bırakılmak istenilen alanlara *NULL* şeklinde bir atama yapılmıştır.

INSERT ifadesi içerisinde verilen değerlerin tabloya eklenmesini sağlanıldığı gibi başka tablolardan sonuç olarak getirilen bilgilerin de INSERT ile bir tabloya eklenmesini sağlanabilir.

```
INSERT INTO Tablo1 (Eklenecek Alanlar) SELECT Seçilecek Alanlar
FROM Tablo2
```

SELECT ifadesi ile seçilen alanların, INSERT ile eklenecek alanlar ile uyuşması gerekmektedir.

Veri Silme (Delete)

Bir tablodan belirli bir kaydı veya kayıtları silme işlemi DELETE ifadesi ile gerçekleştirilir. **DELETE** ifadesini kullanılırken **FROM** (opsiyonel) takısı ile birlikte tablo adını yazılarak hangi tablodan veri silineceğini belirtilir. **WHERE** ifadesi ile birlikte de silinecek kaydın hangi şartları sağlayan kayıt veya kayıtlar olduğu saptanmış olur. DELETE sorgularının örnek kullanımı aşağıdaki gibidir.

```
DELETE FROM Ogrenciler WHERE OgrenciId=128535
```

DELETE sorgusu çalıştırıldıktan sonra **SELECT * FROM Ogrenciler** ifadesi ile tablodaki kayıtlar tekrar sorgulanacak olursa OgrenciId'si 128535 olan kaydın silindiği görülecektir.

OgrenciId	OgrenciAdSoyad	OgrenciDog...	Ogre
128445	Yakup Şeref	05.05.1983 0...	114
128539	Refika Köseler	26.06.1983 0...	102
128488	Oğuzhan Alaşehir	13.06.1982 0...	102
129112	Elif Çakır	14.02.1983 0...	101
129116	Oğuz Özkavukçu	12.07.2053 0...	101
129302	Muhammet Turşak	03.01.1982 0...	102
129303	Mustafa Uysal	04.04.1982 0...	101
129419	Tayfun Akçay	11.06.1982 0...	114
128535	Uğur Umutluoğlu	01.03.1982 0...	102
128541	Ensar MENGİ	NULL	NULL

OgrenciId	OgrenciAdSoyad	OgrenciDog...	Og
128445	Yakup Şeref	05.05.1983 0...	114
128539	Refika Köseler	26.06.1983 0...	102
128488	Oğuzhan Alaşehir	13.06.1982 0...	102
129112	Elif Çakır	14.02.1983 0...	101
129116	Oğuz Özkavukçu	12.07.2053 0...	101
129302	Muhammet Turşak	03.01.1982 0...	102
129303	Mustafa Uysal	04.04.1982 0...	101
129419	Tayfun Akçay	11.06.1982 0...	114
128541	Ensar MENGİ	NULL	NUL

DELETE ifadesinden sonra
128535 numaralı kayıt silinmiştir.

Şekil 152: DELETE ifadesi çalıştırıldıktan sonra soldaki tabloda bulunan 128535 OgrenciID'sine sahip kayıt silinmiştir

Bu şekilde Ogrenciler tablosunda OgrenciId bilgisi 128535 olan kayıt silinecektir. Delete sorguları dikkatli kullanılmalıdır. Çünkü DELETE sorguları WHERE ifadesi ile birlikte bir şart belirtmeksizin kullanılırsa tablodaki tüm verilerin silinmesine neden olunulacaktır.

Böyle bir kullanım istenilmeyen sonuçlara sebep olacaktır. Dikkat edilmesi gereken bir diğer hususta WHERE ifadesi içerisinde belirtilen şart veya şartları sağlayan tüm kayıtların tablodan silineceğidir. Tek kaydın silinmesi istenilrn durumlarda tabloda primary key olan alan WHERE ifadesi içerisinde kullanılmalıdır.

```
DELETE FROM Ogrenciler WHERE OgrenciAdSoyad='Ahmet Yüksel'
```

Bu ifadenin kullanılması durumunda Ogrenciler tablosunda Ahmet Yüksel ismine sahip birden fazla kayıt varsa hepsi silinecektir. Eğer bu sonuç istenilmeyen bir durum ise tabloda primary key olan alanın şart ifadesinde kullanılması doğru olacaktır.

DELETE sorguları içerisinde JOIN, WHERE ... IN, TOP gibi ifadeler de kullanılabilmektedir.

Örnek Sorgu:

```
DELETE FROM Sales.SalesPersonQuotaHistory WHERE SalesPersonID IN
(SELECT SalesPersonID FROM Sales.SalesPerson WHERE SalesYTD > 2500000.00)
```

Veri Güncelleme (Update)

Tablolar üzerinde kayıtlı olan verileri güncellemek, veri tabanı uygulamalarında sıklıkla yapılan bir işlemdir. Tablolar üzerindeki güncelleme işlemi UPDATE komutu ile yapılır. UPDATE sorgularında SET ifadesi kullanarak güncellenecek alanlar ve bu alanların alacakları yeni değerler yazılır. DELETE işleminde olduğu gibi WHERE ifadesi ile birlikte şart cümlesi yazılarak, güncellenecek kayıt veya kayıtların belirlenmesi gerekmektedir. WHERE ifadesi bulunmayan bir UPDATE cümlesi, tablodaki tüm kayıtları güncelleyecektir. Bu nedenle DELETE işleminde olduğu gibi, UPDATE işleminde de WHERE ifadesinin kullanımına dikkat edilmesi gerekmektedir. UPDATE ifadesinin örnek kullanımı aşağıdaki gibidir.

```
UPDATE Ogrenciler
SET OgrenciSehir='Manisa', OgrenciEPosta='e128539@metu.edu.tr'
WHERE OgrenciId=128539
```

OgrenciId	OgrenciAdSoyad	OgrenciDog...	OgrenciBol...	OgrenciEPosta	OgrenciSehir
128539	Refika Köseler	26.06.1983 0...	102	refika@netron.com.tr	Ankara
128539	Refika Köseler	26.06.1983 0...	102	e128539@metu.edu.tr	Manisa

Şekil 153: UPDATE sorgusunun ardından Ogrenciler tablosunda 128539 numaralı kaydın bazı bilgileri güncellenmiştir

Bazı durumlarda ise tablodaki tüm kayıtların güncellenmesi istenebilir. Örneğin ürünlerle ilgili bilgilerin tutulduğu bir tabloda tüm ürünlerin fiyatlarına belli bir oranda zam yapılmak istenebilir. Tüm ürünlerin fiyatları aynı oranda değiştirilmek istenilirse WHERE ifadesi kullanılmadan bu işlem gerçekleştirilebilir. AdventureWorks veri tabanındaki Production.Product tablosunda, tüm ürünlere %10 zam yapılmasının istendiğini düşünelim. Aşağıdaki sorgu bu işlemi yerine getirecektir.

```
UPDATE Production.Product SET ListPrice = (ListPrice * 1.10)
```

Yine belirli bir şartı sağlayan ürünlerin fiyatlarının güncellenmesi istenebilir. Aşağıdaki sorguda ListPrice değeri 1000'den küçük olan ürünlerin fiyatları %10 arttırılıyor.

```
UPDATE Production.Product SET ListPrice = (ListPrice * 1.10)
WHERE ListPrice < 1000
```

ReorderPoint	StandardCost	ListPrice	Size
375	352,1394	594,8300	58
375	352,1394	594,8300	60
375	204,6251	337,2200	44
375	204,6251	337,2200	48
375	204,6251	337,2200	52
375	747,2002	1364,5000	42
375	706,8110	1364,5000	44
375	706,8110	1364,5000	48
375	747,2002	1364,5000	46
375	739,0410	1349,6000	42
375	699,0928	1349,6000	44

ReorderPoint	StandardCost	ListPrice	Size
375	352,1394	654,3130	58
375	352,1394	654,3130	60
375	204,6251	370,9420	44
375	204,6251	370,9420	48
375	204,6251	370,9420	52
375	747,2002	1364,5000	42
375	706,8110	1364,5000	44
375	706,8110	1364,5000	48
375	747,2002	1364,5000	46
375	739,0410	1349,6000	42
375	699,0928	1349,6000	44

Şekil 154: WHERE ifadesi ile sadece belirli kayıtların ListPrice değerleri değiştirilmiştir.

Sorgu sonucunda Production.Product tablosunda sadece ListPrice değeri 1000'den küçük olan kayıtların ListPrice alanları güncellendi.

DELETE ifadesinde olduğu gibi UPDATE işlemlerinde de sorgu içerisinde JOIN, WHERE ... IN, TOP gibi ifadeler kullanılabilmektedir.

Örnek Sorgu:

```
UPDATE Production.Product SET ProductModelID = 129 WHERE ProductModelID IN
(SELECT ProductModelID FROM Production.ProductModel
WHERE CatalogDescription IS NOT NULL)
```

KISIM SONU SORULARI:

1) Product tablosunda, ProductSubCategoryId'si 3 olan tüm ürünlerin liste fiyatlarını %10 arttıran SQL sorgusunu yazınız.

2) ProductCategory tablosunda Kategori ismi 'Bikes' olan ve rengi 'NULL' olmayan ürünlerin isimlerini ve renklerini Products tablosundan dönen SQL sorgusunu yazınız.

3) Product tablosunda, her bir alt kategoriye ait ürün sayılarını ayrı ayrı hesaplayıp, sayısı en çok olandan, en az olana göre sıralayan SQL sorgusunu yazınız.

4) Product tablosunda, her bir alt kategoriye ait ürün sayılarını ayrı ayrı hesaplayıp, her bir altkategoriye ait, toplam ürün sayısı 30'dan küçük olan altkategorinin Id'sini ve toplam ürün sayısını listeleyen SQL sorgusunu yazınız.

5) Count(*) ve Count('*kolonadi*') ifadeleri arasındaki farkı açıklayınız.

6) 'Bikes' kategorisindeki en pahalı 10 ürünün adı ve fiyatını listeleyen SQL sorgusunu yazınız.

7) Veri tiplerinden char, varchar, nvarchar, nchar arasındaki farkları açıklayınız.

8) Product tablosundaki, biribirinden farklı renk (Color) değerlerini listeleyen SQL sorgusunu yazınız.

9) Arkadaşlarınıza ait aşağıdaki bilgileri tutan tabloyu oluşturan SQL sorgusunu yazınız.
Id: int veri tipinde, otomatik artan (auto increment) ve Primary Key özelliklerine sahip.
Ad: 50 karakter uzunluğunda metin veri tipinde, boş olamaz.
PostaAdresi: 30 karakter uzunluğunda, boş olabilir.
Telefon: 15 karakter uzunluğunda, boş olabilir.
DoğumTarihi: Tarih veritipinde, boş olabilir.
Cinsiyet: Sadece 0 ve 1 değerlerini tutabilen.

10) Ürünler tablosundan, altkategori adı (ProductSubCategory tablosundaki Name kolonu) ilk harfi 'B' ve son harfi 's' olan, aynı zamanda, ProductID değerleri 850 ile 994 aralığında olan ve rengi 'Black', 'Silver', 'White', 'Yellow', 'Green' renklerinden herhangi biri olan ürünlerin ProductId, Name ve Color alanlarını listeleyen SQL sorgusunu yazınız. (Normal şartlarda 8 kayıt dönmesi gerekmektedir.)

KISIM IV: ADO.NET

YAZAR: OSMAN ÇOKAKOĞLU

BÖLÜM 0: VERİ ve VERİYE ERİŞİM TEKNOLOJİLERİ

Birçok uygulama bazı bilgileri geçici olarak tutup, daha sonra o bilgileri kullanarak işlemler yapar. Ancak bu bilgiler sadece uygulama çalıştığı sürece erişilebilir durumdadır. Çünkü bellek, uygulama çalışırken bilgileri geçici olarak tutmak için kullanılır. Bu şekilde çalışan uygulamalara hesap makinası örnek olarak verilebilir. Yani kullanıcıdan birkaç veri alıp bunları hemen işleyen ve bu bilgilerin daha sonra kullanılması gerekmediği için geçici olarak tutulmasında sakınca olmayan uygulamalardır.

Ancak her uygulama bu şekilde geliştirilemez. Çünkü alınan bilgilere uygulama kapatıldıktan sonra da erişilmesi gerekebilir. Bu durumda bilgileri bellek dışında, veriyi kalıcı olarak tutabilecek disklere kaydetmek gerekecektir. İşte bu ihtiyacı karşılamak için farklı yöntemler geliştirilmiş ve günümüzde çok büyük miktarlarda veri tutan sistemler tasarlanmıştır.

Kısaca bu gelişime bakacak olursak; ilk olarak, günümüzde de hala çok sık kullanılan veri tutulabilecek belki de en basit **yapısal olmayan** sistem "text dosya" olarak söylenebilir. Ancak bu yapılarda zamanla yetersiz kalmış ve **yapısal** fakat **hiyerarşik olmayan** çözümler geliştirilmiştir. Bu başlık altında Comma Separated Value (CSV) dosyalar, Microsoft Exchange dosyaları, Active Directory dosyaları örnek olarak verilebilir. Ancak zamanla sadece veriyi tutmuş olmak da yetersiz kalmış ve verileri **hiyerarşik** bir yapıda tutma gereksinimi duyulmuştur. Bu yapıya da verilecek en güzel örnek XML teknolojisi olacaktır. Fakat bu yapınında yetersiz kaldığı durumlar için **ilişkisel (Relational)** veri depolama sistemleri geliştirilmiştir. İlişkisel veri depolama modeli günümüzde yoğun olarak kullanılmaktadır. Bu modelle saklanan verileri yönetmek için geliştirilen uygulamalara da ilişkisel veritabanı yönetim sistemi (Relational Database Management System,RDBMS) adı verilmektedir.

Böylece **veritabanı**nı (Database), veriyi daha sonra kullanılabilmesi ya da sadece depolanması amacıyla belli bir formatta tutan sistem olarak açıklayabiliriz.

Günümüzde büyük miktarlardaki verileri depolamak için ve bunları yönetmek için bazı şirketlerin geliştirmiş olduğu uygulamalar vardır. Bu uygulamalara örnek olarak Microsoft SQL Server, Oracle, Microsoft Access verilebilir.

Aslında buraya kadar anlatılanlarla verinin nasıl tutulduğu incelenmiştir. Ancak veri, yukarıda bahsettiğimiz sistemlere uygulamalardan nasıl gönderilecek ya da bu sistemlerden uygulamalara nasıl alınacak konusuna hiç değinilmemiştir.

İşte bu aşamada artık, veriye erişim teknolojilerinden bahsedilmesi gerekmektedir. Peki nedir bu teknolojiler?

Veriye Erişim Teknolojileri

ODBC (Open Database Connectivity)

Microsoft ve diğer kuruluşların geliştirdiği bu teknoloji ile birçok veri kaynağına bağlanarak, veri alışverişi yapılabilmektedir. ODBC uygulama ortamlarına bir API sunmakta ve uygulamaların farklı sistemlere bağlanılabilmesini sağlamaktadır.

ODBC teknolojisi 1990 öncesi geliştirilmiş olmasına rağmen ADO.NET platformunda da yer alan bir teknolojidir. ODBC hem yerel (local) hem de uzaktaki (Remote) veri kaynaklarına erişmek için kullanılabilecek bir veri erişim teknolojisidir.

DAO (Data Access Objects)

DAO, ODBC'nin aşağı seviye diller için (C,C++) geliştirilmiş olması ve kullanımının zor olması nedeniyle, Microsoft tarafından Visual Basic 3 ile geliştirilmiş ve kullanımı çok

kolay bir teknolojidir. Microsoft'un, Microsoft Access veritabanlarına erişim standardı olan Jet için geliştirdiği bu model, halen VB6'dan MS Access'e erişim için en hızlı yöntemdir.

RDO (Remote Data Objects)

Microsoft'un Visual Basic 4 ile duyurduğu RDO; DAO'nun ODBC sağlayıcısındaki eksikliğini gidermek için geliştirilmiştir. Uzak veri kaynaklarına erişimde ODBC'nin daha performanslı kullanılmasını sağlamıştır.

OLE DB (Object Linking and Embedding DataBase)

ODBC'de olduğu gibi driver (sürücü) mantığıyla çalışan OLE DB, arka tarafta COM arayüzünü kullanan bir tekniktir. Kaydettiği gelişme ile bir çok sisteme bağlantı için kullanılabilmektedir. Bu başarısından dolayı da ADO.NET içinde önemli bir yeri vardır.

ADO (ActiveX Data Objects)

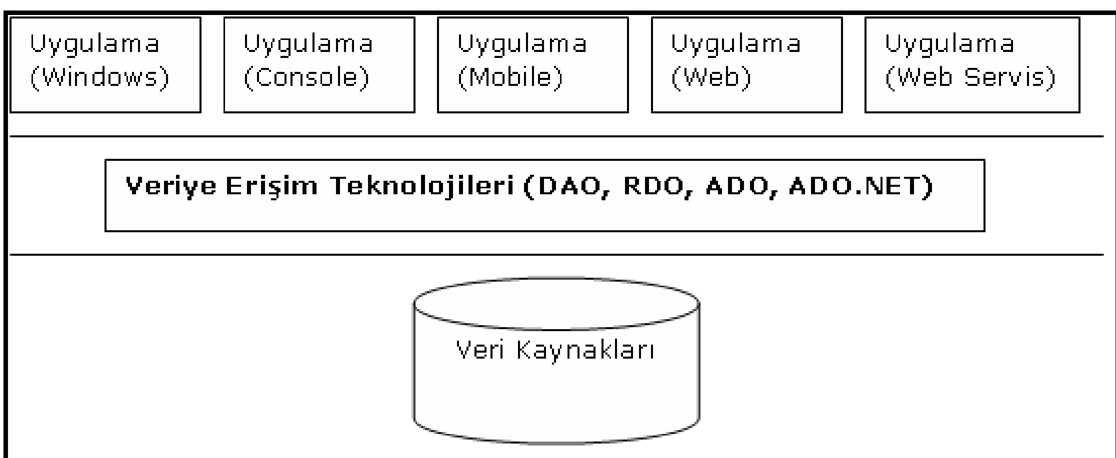
ADO aslında arka planda OLE DB teknolojisini kullanan ve veriye erişimi daha da kolaylaştıran, yüksek seviye programlama dilleri için veri erişiminde tercih edilen bir teknolojidir.

ADO, ADO.NET'e temel oluşturduğu söylenen bir teknoloji olmasına rağmen bu tam olarak doğru değildir. Bunun en büyük nedenlerinden birisi de ADO'nun COM desteğine gereksinim duymasıdır. Günümüzde bazı platformlarda halen kullanılan bu teknoloji, XML standardına destek vermek için eklentilerle güçlendirilmesine rağmen, XML formatındaki veriyi sadece taşıyabilmektedir. Tam aksine ADO.NET, XML standardı üzerine kurulmuş ve veriyi taşırken XML formatını kulllanan bir teknoloji olarak geliştirilmiştir.

ADO.NET

ADO.NET, Microsoft'un .NET platformunda çok önemli bir yere sahip, .NET uygulamalarından, her türlü veri kaynağına erişim için gerekli hazır tipleri olan, COM desteği gerektirmeyen, XML standardı üzerine kurulmuş ve en önemlisi de .NET Framework'ün imkanlarını kullanabilen, ADO'nun gelişmiş bir versiyonu olmaktan çok, yeni bir teknoloji olarak karşımıza çıkmaktadır.

ADO.NET, .NET Framework' ün bir parçası olarak 1.0, 1.1, 2.0 ve 3.0 versiyonlarında yer almaktadır. Ayrıca veri işlemlerini çok kolaylaştıran ve Nesneye Yönelik Programlama (Object Oriented Programming) modeline uygun yapısı nedeniyle son derece kullanışlı bir platformdur.

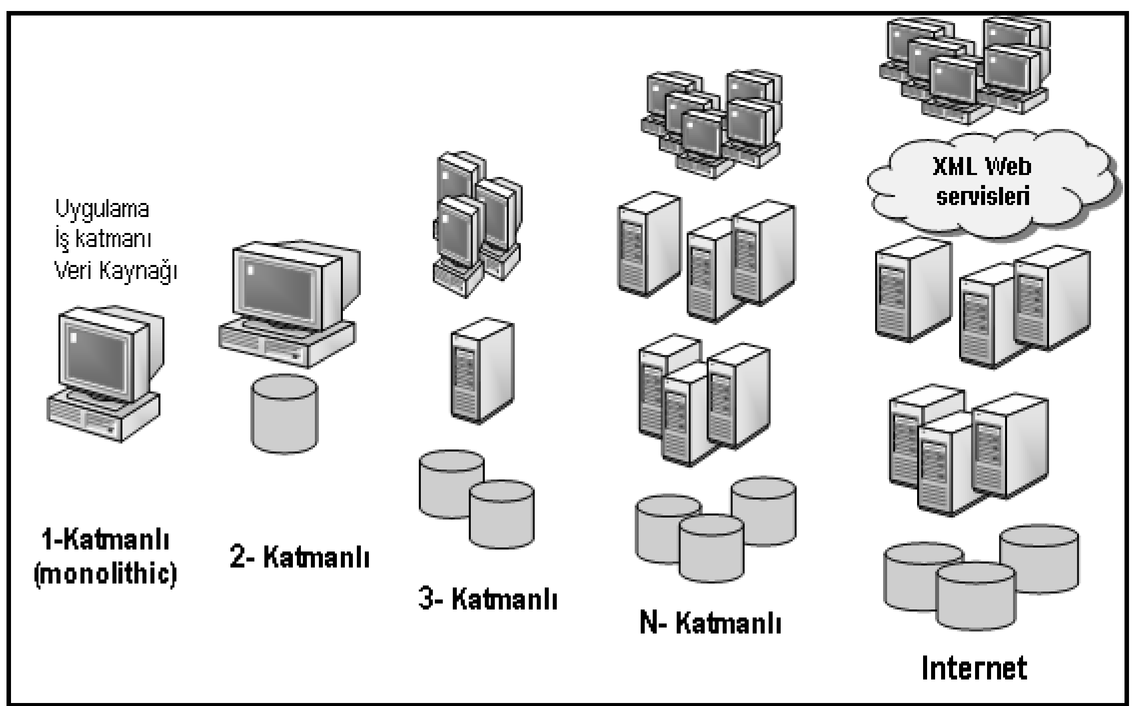


Şekil 155: Veri kaynakları ve Uygulamalar arasındaki bağlantı

Veriye Erişim Yöntemleri

Veriye erişim yöntemleri uygulamalar tasarlanırken belirlenen stratejiler olarak karşımıza çıkmaktadır. Bu noktada, verinin nerede ve nasıl saklanacağı, kimlerin bu veriler üzerinde neleri yapmaya yetkili olacağı gibi konular erişim yöntemlerini seçerken önemli olmaktadır.

Veriye erişim yöntemleri uygulamayı katmanlara ayırmak olarak da düşünülebilir. Bir uygulamadaki mantıksal her birim bir **Katman** olarak isimlendirilir.

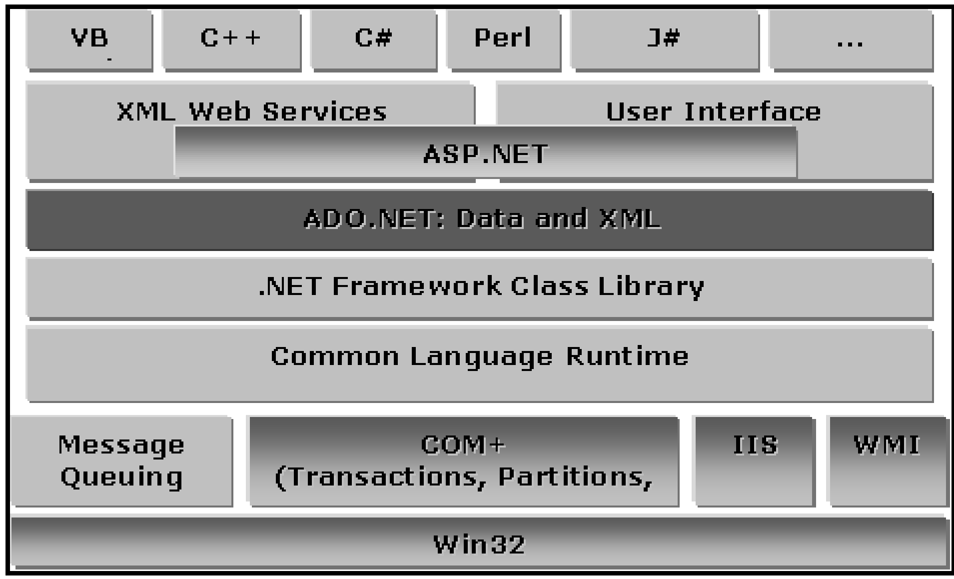


Şekil 155: Katmanlı mimarideki gelişim süreci

Yukarıda da görüldüğü gibi; uygulamalarda zaman içerisinde, verinin tutulduğu ortam ile, verinin kulllanıldığı ya da sunulduğu ortamın birbirinden ayrı olması gereksinimi ortaya çıkmış ve bu nedenle de katman sayısı giderek artmıştır. Tabi ki uygulamanın durumuna göre bu yapıların tümü kullanılabilmektedir. Ancak burada önemli olan, uygulamanın şu anda nasıl çalıştığı değil, ilerleyen dönemlerde ki gelişmelere kolay entegre olabilecek, farklı platformlarda en ufak bir değişiklik dahi yapmadan kullanılabilecek bir sistem olmasıdır. Bu durumda şöyle bir öneride bulunmak doğru olacaktır; en azından veriye erişim katmanı ile, uygulama katmanının ayrılması ve hatta araya da en az bir katmanın dahil edilmesi, uygulamanın ölçeklenebilirliği ve daha sonra tekrar kullanılabilmesi adına güzel bir adım olacaktır.

ADO.NET Mimarisi

ADO.NET, .NET platformunda çok önemli bir noktada yer almaktadır. Bunu aşağıdaki şekilde net bir biçimde görebilmekteyiz.



Şekil 156: ADO.NET'in .NET Framework içindeki yeri

Şekilde de görüldüğü gibi ADO.NET, .NET Framework'ün merkezinde ve bir çok platformda kullanılan ortak bir katmandır. O nedenle .NET ile geliştirilen tüm uygulamalar, veri kaynaklarına erişim için ADO.NET tiplerinden yararlanmaktadır. Bu da uygulama geliştiriciler için çok büyük bir avantaj sağlamaktadır. Çünkü ADO.NET platformu da, .NET Class Library üzerinde çalışmakta ve bu sayede .NET Framework içindeki yazılmış bir çok tipi kullanabilmektedir. Bu tipler aracılığıyla, farklı veri kaynaklarına bağlanmak, verileri alıp, alınan verileri uygulamalarda kullanmak, yeni veriler eklemek ve mevcut verileri güncellemek çok kolay bir hal almaktadır.

Tabiki ADO.NET platformundan yararlanırken, .NET Framework'ün yapısından dolayı dikkat etmemiz gereken noktalar olacaktır. Çünkü ADO.NET ile, çok farklı veritabanı ve veritabanı yönetim sistemleri kullanılabilmektedir. Böyle bir durumda farklı sistemlerle haberleşebilmek ve her biriyle en doğru, en performanslı şekilde çalışabilmek için farklı gereksinimler ortaya çıkmaktadır. Çünkü her sistemin kendi standartları, desteklediği farklı standartlar vardır. İşte bu durumu göz önüne alarak .NET geliştiricileri, farklı standartları destekleyen tipler yazmışlar ve bunları ADO.NET kütüphanesinde ayrı ayrı isim alanları (Namespace) içine yerleştirmişlerdir. Bu isim alanlarından, SQL Server'a bağlantı kurmak için yazılmış tipleri bulunduran isim alanına **Sql Server Veri Sağlayıcısı (Sql Server .NET Data Provider),** Oracle için olan isim alanına **Oracle Data Provider**, OleDb standartlarını destekleyen sistemlerle iletişim kurmayı sağlayan tiplerin yer aldığı isim alanına **OleDb .NET Data Provider,** ODBC standardı için olanlara da **ODBC .NET Data Provider** denilmektedir. Bu isim alanları, **System.Data** isim alanı altında yer almaktadır.

System.Data

ADO.NET ile uygulama geliştirirken kullanılabilecek, tüm veri sağlayıcılar için ortak olan bileşenlerin bulunduğu isim alanıdır. Mesela burada; çok güçlü ve çok da kullanışlı olan **DataSet, DataTable** gibi tipler vardır.

System.Data.SqlClient

SQL Server ile çalışabilmek için yazılmış tiplerin yer aldığı bu isim alanı, SQL Server 7.0 ve sonraki versiyonları ile birlikte kullanılabilmektedir.

System.Data.OleDB

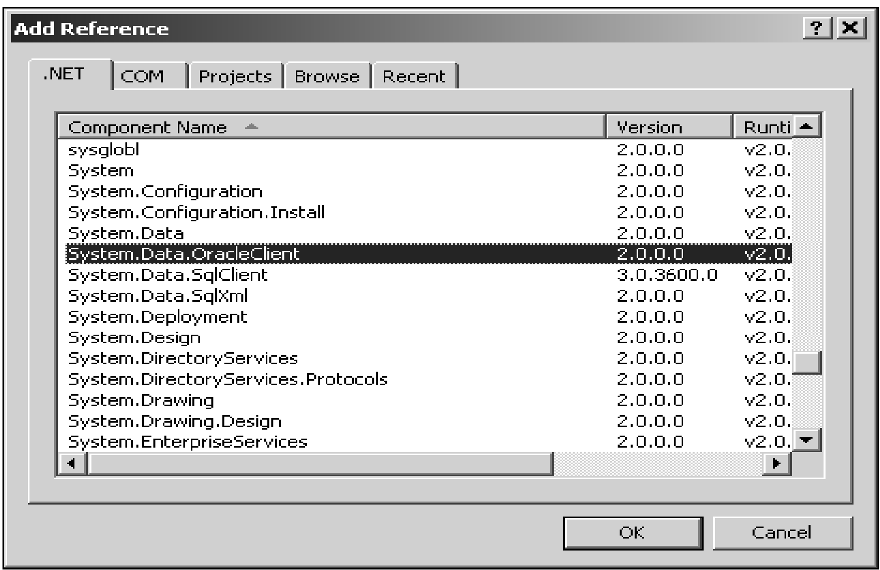
SQL Server 6.5 ve önceki sürümleri, Microsoft Access, Oracle, Text Dosyalar, Excel Dosyaları vb. gibi, OleDb arayüzü sağlayan tüm sistemlere bağlantı kurmayı sağlayan tipleri barındırmaktadır. Ayrıca SQL Server'ın 6.5 sonrası versiyonları için de kullanılabilmektedir.

System.Data.Odbc

ODBC (Open DataBase Connectivity) standartlarını destekleyen ve ODBC sürücüsü bulunan sistemlere bağlantı kurmayı sağlayan tipleri içermektedir.

System.Data.Oracle

.NET 2.0 ile birlikte .NET Framework içinde Oracle'a biraz daha güçlü bir destek gelmiştir. Ancak Oracle veri sağlayıcısını kullanabilmek için öncelikle aşağıda görülen, **System.Data.OracleClient** referansının projeye dahil edilmesi gerekmektedir.



Şekil 157: Oracle Provider Referansı

Bu referansı ekledikten sonra, uygulamalarda Oracle için yazılmış olan tipler kullanılabilmektedir. Bu tiplerin bazıları aşağıda görülmektedir. Aslında Oracle veri tabanlarına bağlanmak ile SQL Server veri tabanlarına bağlanıp veri almak, veriyi kullanmak ve hatta uygulama ortamındaki güncellemeleri veri kaynağına doğru gönderme işlemleri genel olarak aynı mantık üzerinde ilerlemektedir. Çünkü her ikisi de ve hatta diğer veri sağlayıcılarda aynı platformu kullandıkları için sadece nereye, hangi makinaya, hangi veritabanına, hangi güvenlik ayarlarıyla bağlanılması gerektiği gibi

bilgilerde değişiklik yapılması gerekmektedir. Bu nedenle, kitabın ilerleyen bölümlerinde yapılacak örnekler ve değinilecek konular ***Sql Server .NET Data Provider*** üzerinden yapılacaktır.

```
using System.Data.OracleClient;

namespace WindowsApplication1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
            System.Data.OracleClient.
        }
    }
}
```

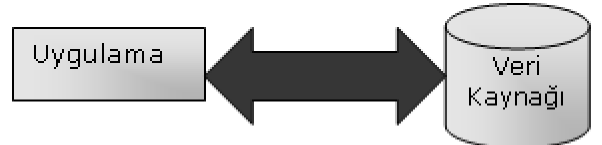
```
OracleBFile
OracleBinary
OracleBoolean
OracleClientFactory
OracleCommand
OracleCommandBuilder
OracleConnection
OracleConnectionStringBuilder
OracleDataAdapter
OracleDataReader
```

Şekil 158: Oracle Provider ile Kullanılabilecek Class'lar

Sql Server .NET Data Provider bileşenlerini incelemeden önce, herhangi bir veri kaynağı ile uygulamalar arasında veri taşıyabilmek için neler yapılması gerektiğini, provider'lardan bağımsız olarak incelemek, genel olarak ADO.NET mimarisine yukarıdan bakabilmek anlamına gelmektedir. Öncelikle en az bir adet **veri kaynağı** olması gerekmektedir. Burada veri kaynağı olarak Sql Server, Oracle, DB2, Interbase, Paradox, XML, Excel, Dbase, Access, CSV , Text vb... kullanılabilir.

Şekil 159: Veri Kaynağı (Data Source)

Veri kaynağı ile uygulamalar farklı makinalarda da olabileceği gibi, aynı makina üzerinde de olabilirler, ancak nerede olurlarsa olsunlar, sonuçta iki farklı platformdan bahsedildiği için; uygulama ile veri kaynağını konuşturabilmek ve onları iletişime hazırlamak gerekecektir. İşte bu noktada, veri kaynağı ile uygulamaları konuşturacak olan, hangi makinadaki, hangi veri kaynağına hangi güvenlik ayarlarıyla gidilmesi gerektiği bilgilerini taşıyan **bağlantı (Connection)** nesnesine ihtiyaç duyulacaktır.



Şekil 160: Veri Kaynağı ve Uygulama (Application)

Veri kaynağı ile bağlantının kurulmuş olması, veri taşımak için yeterli olmayacaktır. Bir veri kaynağına veri göndermek için ya da oradan veri almak için bazı komutlara (Sql Cümleleri, Stored Procedure (Saklı yordam) isimleri) ve hatta bu komutlarla birlikte taşınması gereken parametrelere ihtiyaç olacaktır. ADO.NET platformunda bu işlemleri yapabilmeyi sağlayan ve belli bir bağlantı üzerinden ilgili veri kaynağına müdahale edebilen, oraya veri gönderme veya oradan veri alma isteğini sistemler arasında taşıyan **Command** adı verilen bir bileşen yer almaktadır.

Şekil 161: Veri Kaynağı, Uygulama ve Command Nesnesi

Command adı verilen bileşen aracılığıyla, uygulama tarafından veri kaynağına veri gönderebilir. Bunun sonucunda herhangi bir işlem yapılmasına gerek yoktur. Ancak kullanılan sql cümlesi veri getirecek şekilde yazılmışsa; gelecek olan veriyi uygulama tarafında kullanabilmek için de bazı bileşenlere ihtiyaç olacaktır.

Bu bileşenlerden bir tanesi, yine provider'a göre değişen özelliklere sahip, verinin uygulamaya ileri yönlü ve sadece okunabilir olacak şekilde aktarılmasına imkan veren **DataReader** adındaki bileşendir.

Şekil 162: Veri Kaynağı, Uygulama , Command Nesnesi ve DataReader

Yukarıda genel olarak bahsedilen bu bileşenlerin Sql Server .NET Data Provider isim alanındaki karşılıklarını ve en sık kullanılan ayrıntıları diğer bölümde ele alınacaktır.

BÖLÜM 1: SQL SERVER .NET VERİ SAĞLAYICISI (DATA PROVIDER) ORTAK TİPLERİ

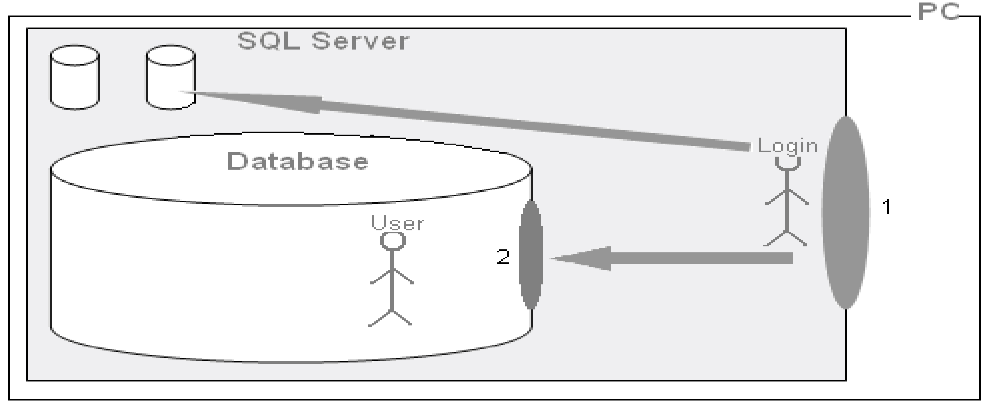
SqlConnection

Microsoft SQL Server 7.0 ve sonrası için kullanılan bağlantı sınıfıdır. Bu class ile bağlantı kurmak istenilen SQL Server örneğini, kullanılmak istenilen veritabanını, ve bağlantı için gerekli olan güvenlik ayarlarını belirttikten sonra, bağlantı açılarak veritabanı kullanılabilir hale getirilmiş olur. Daha sonra diğer tipler devreye girer ve veri taşıma işlemleri başlatılabilir. Ancak unutulmaması gereken çok önemli nokta; açılan bağlantının işlemler bittikten sonra tekrar kapatılması gerekliliğidir. Eğer açılan bağlantılar kapatılmazsa; bir süre sonra SQL Server'ın kaynaklarının gereksiz yere harcanmasından dolayı sorunlar yaşanabilir.

SqlConnection sınıfı ile çalışırken dikkat edilmesi gereken belki de en önemli nokta bağlantı cümlesidir. Çünkü SQL Server'a bağlantı kurulurken, belli yetkilere sahip, belli rolleri olan kullanıcılar adına bağlantı kurulması gerekmektedir. Bu nedenle kısaca SQL Server'ın güvenlik modellerinden bahsetmekte yarar vardır.

SQL Server iki oturum açma modelini desteklemektedir. Bunlardan ilki; Windows'a giriş yapıldıktan (oturum açtıktan) sonra tekrar yeni bir şifre girmeyi gerektirmeyen **Windows Authentication** modelidir. Bu modelde, SQL Server'a bağlanacak kullanıcıların (**Login**) tekrar tekrar kontrol edilmesine gerek yoktur. Bir başka deyişle, işletim sistemine giriş yapabilen yani makinada oturum açma yetkisi olan kişilerin SQL Server'a giriş yapabildiği durumdur. Ancak SQL Server tarafından, Windows kullanıcıları tanınıyor olmasına dikkat edilmelidir. Ardından bu kullanıcılar SQL Server'a bağlandıklarında, giriş yaptıktan sonra **nerelerde hangi** işlemleri **yapabilirler**, hangi işlemleri **yapamazlar** sorularına cevap aranmalıdır.

Bu noktada da odaklanılması gereken nokta kullanıcı **(user)** kavramıdır. Aşağıdaki şekilde login ve user kavramlarının geçerli oldukları alanlar belirtilmiştir.



Şekil 163: SQL Server Authencation Mekanizması

Yukarıdaki şekilde görülen 1 kapısı aslında; SQL Server'a giriş yapılan kapıdır. Bu kapıdan geçebilmek için, SQL Server'da kayıtlı kullanıcı (Login) olunmalıdır. Giriş yapıldıktan sonra, SQL Server içerisine dahil olunur ancak herhangi bir veritabanında işlem yapma yetkisine sahip olunmaz. Bu durumda da işlem yapılacak ya da çalışılacak veritabanına da giriş yapılması gerekmektedir. Bunun için 2 kapısında gereken kontroller yapılır. 2 kapısındaki kontrol de; içeriye girmiş olan yani login olan kullanıcının ilgili veritabanı içerisinde hangi isimle işlem yapacağı belirlenir. Aslında şöyle bir özet doğru olacaktır.

*"SQL Server'a giriş yapabilmek için **Login** olarak kayıtlı olmak gerekir. Login olarak giriş yapıldıktan sonra da herhangi bir veritabanı üzerinde işlem yapabilmek için içeriye giriş yapan o Login'i ilgili veritabanı içerisinde temsil edecek olan **User** olarak veritabanı içerisinde tanınmak gerekir."*

Bu bilgilerden sonra artık SQL Server'a kayıtlı Login'lerin SQL Server içerisine nasıl giriş yapabilecekleri noktalarına değinilebilir.

SqlConnection tipinin en çok kullanılan özellikleri ve metodları şunlardır.

ConnectionString	*"Data Source=Server; Initial Catalog=Veritabani; Integrated security=True;Persist Security Info=False"* *"Data Source=Server; Initial Catalog=Veritabani; Integrated security=False;Persist Security Info=False;User ID=loginadi; Password=sifre"* *"Data Source=Server; Initial Catalog=Veritabani; Integrated security=True;Persist Security Info=False;Pooling=True/False; Max Pool Size=100; Min Pool Size=0 "* ***OleDbConnection için;*** *"Provider=Microsoft.Jet.OleDb.4.0;Data Source=AccessDosyasi.mdb"* **Data Source** : SQL Server instance adı. **Initial Catalog** : Veritabanı adı. **Integrated Security** : True; Windows Authentication ile bağlantının yapılması, False ise; SQL Server Authentication ile bağlantının yapılmasıdır ve UserID-Password girilmelidir. **Persist Security Info** : SQL Server'ın güvenlik bilgilerini uygulama tarafına geriye göndermesidir. Varsayılan olarak ***False*** değerine sahiptir. **Pooling** : SQL Server bağlantı havuzunu destekler. Aynı ConnectionString isteklerinde bağlantının baştan oluşturulması için tekrar kaynak harcamak yerine, mevcut bağlantının işlemler bittikten sonra havuza atılmasını sağlamak için kullanılır. Varsayılan olarak ***True*** değerine sahiptir. **Max Pool Size** : Aynı bağlantı cümleleri için açılan havuzda, maksimum kaç tane bağlantının bulunabileceğini belirtir. **Min Pool Size** : Aynı bağlantı cümleleri için açılan havuzda, minimum kaç tane bağlantının bulunması gerektiğini belirtir. **Provider** : Sadece OleDbConnection için kullanılmaktadır.

Open()	Bağlantının kullanıma hazır olması ve açılmasını sağlar. SQL Server üzerinden kaynakların ayrılması ve kullanılabilmesini sağlayan metottur.
Close()	Bağlantının kapatılmasını sağlar. SQL Server tarafından ayrılan kaynakların bırakılmasını sağlar.
ConnectionState	Bağlantının durumunu verir. Alabileceği değerler ConnectionState. Broken Closed Connecting Executing Fetching Open şeklindedir. Böylece bağlantının durumuna göre yapılacak işleme karar verilebilir.

Tablo 22 : SqlConnection tipinin genel üyeleri

Örnek bağlantı cümleleri aşağıdadır.

```
/*Aşağıda oluşturulan baglanti nesnelerinin herbirinde Data Source
niteliğine localhost değeri atanarak, varsayılan Sql sunucusuna
bağlanılacağı ifade edilmiştir. Şartlara göre localhost yerine, sunucu
adı, ip numarası, sunucuAdi\örnekAdi gibi bilgiler girilmeside
gerekebilir.

   Initial catalog niteliği ise; bağlanılmak istenen sunucudaki hangi
veritabanına bağlanılacağı bilgisini tutmaktadır.

   Integrated Security niteliğine true,false veya SSPI değerlerinin
atanması halinde ise; yukarıda şekille de anlatılan SQL Server'a bağlantı
kurulurken gerekli olan güvenlik bilgileri belirtilmiş olur. Integrated
Security değerinin true ya da SSPI olması halinde, SQL Server'ın Windows
Authentication modu desteklemesi ve işletim sisteminde kayıtlı user'ın SQL
Server'a da kayıtlı olması gerekmektedir.*/

SqlConnection baglanti = new SqlConnection("Data Source=localhost; Initial
Catalog=ADONETDB; Integrated security=True");

SqlConnection baglanti = new SqlConnection("Data Source=localhost; Initial
Catalog=ADONETDB; Integrated security=SSPI");

SqlConnection baglanti = new SqlConnection("Data Source=.; Initial
Catalog=ADONETDB; Integrated security=True");

SqlConnection baglanti = new SqlConnection("Data Source=.; Initial
Catalog=ADONETDB; Integrated security=SSPI");

SqlConnection baglanti = new SqlConnection("Data Source=NETRON\\SQL2005;
Initial Catalog=ADONETDB; Integrated security=True");

SqlConnection baglanti = new SqlConnection("Data Source=NETRON\\SQL2005;
Initial Catalog=ADONETDB; Integrated security=SSPI");

SqlConnection baglanti = new SqlConnection("Data
Source=192.168.1.2\\SQLINSTANCE1; Initial Catalog=ADONETDB; Integrated
security=True");
```

```
SqlConnection baglanti = new SqlConnection("Data
Source=192.168.1.2\\SQLINSTANCE1; Initial Catalog=ADONETDB; Integrated
security=SSPI");


/* Aşağıdaki bağlanti cümlelerinde ise;
   * Güvenlik bilgilerinin farklı olduğu durumlar görülmektedir. Bu
cümlelerde, SQL Server'a Windows Authentication modu ile değil, SQL
Server'da açılmış kullanıcı hesapları ile nasıl oturum açılabileceği
gösterilmektedir.
   * Aşağıdaki cümlelerde güvenlik bilgileri yukarıdakilere göre
farklıdır. Windows Authentication modunun kullanılmayacağı Integrated
Security=False ifadesi ile belirtilmektedir. User ID ve Password bilgileri
girildiği için; Integrated Security niteliği otomatik olarak false
değerini alır.            */

SqlConnection baglanti = new SqlConnection("Data Source=.; Initial
Catalog=ADONETDB; Integrated Security=False; User ID=sa; Password=1234");

SqlConnection baglanti = new SqlConnection("Data Source=.; Initial
Catalog=ADONETDB; Integrated Security=False; User ID=netronlogin;
Password=1234");

SqlConnection baglanti = new SqlConnection("Data Source=.; Initial
Catalog=ADONETDB; User ID=netronlogin; Password=1234");

SqlConnection baglanti = new SqlConnection("Data Source=.; Initial
Catalog=ADONETDB; User ID=sa; Password=1234");
```

OleDbConnection için örnek bağlantı cümlesi aşağıdaki gibidir.

```
/*Aşağıdaki cümlede; OleDB Data Provider aracılığı ile herhangi bir Access
veritabanına geçerli bir bağlantı açılması için gerekli ifade yazılmıştır.
OleDbConnection nesnesinin aşağıdaki şekilde kullanılabilmesi için,
System.Data.OleDb isim alanının uygulamaya eklenmiş olması
gerekmektedir.*/

OleDbConnection baglanti = new OleDbConnection
("Provider=Mcrosoft.Jet.OleDb.4.0; Data Source=C:\\ADONETDB.mdb");
```

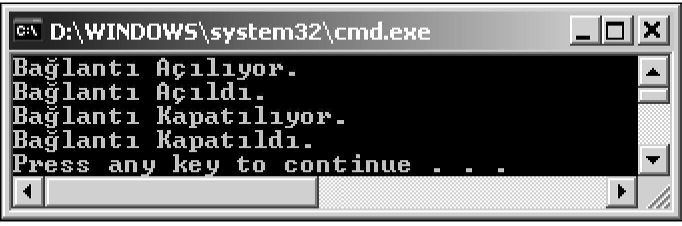
Bu bilgiler ışığında aşağıdaki örnekte, bir veritabanına bağlantı kurup, bağlantının veri taşınabilmesi için hazırlanmasını sağlayan işlemler incelenmektedir.

Örnek Uygulama

Bu örnekte, SQL Server'a bağlantı kurulmakta ve bu bağlantı açılarak veritabanı kullanıma hazır hale getirilmektedir. SqlConnection nesne örneğinin Open metodu çağırıldıktan sonra, 5 saniye boyunca ilgili bağlantı açık tutulmakta ve sonrasında bağlantı kapatılmakta ve böylece sistem kaynakları gereksiz yere tüketilmemektedir. Burada genel bir kuralı belirtmekte fayda vardır. Çoğunlukla söz konusu veri kaynaklarına olan bağlantılar mümkün olduğunca geç açılmalı ve işlemleri takiben mümkün olduğunca erken kapatılmalıdır.

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Data.SqlClient;

namespace VeriTabaniBaglantisi
{
    class Program
    {
        static void Main(string[] args)
        {
            SqlConnection baglanti = new SqlConnection();
            //baglanti nesnesini oluştururken, yapıcı metodun
(Constructor) aşırı yüklenmiş versiyonları da kullanılabilir.
            baglanti.ConnectionString = "Data Source=.;Initial
Catalog=SDS; Integrated Security=True";

            //bağlantı açılır. Böylece veritabanı kullanıma hazır hale
gelmiş olur.
            Console.WriteLine("Bağlantı Açılıyor.");
            baglanti.Open();
            Console.WriteLine("Bağlantı Açıldı.");
            System.Threading.Thread.Sleep(5000);
            Console.WriteLine("Bağlantı Kapatılıyor.");
            baglanti.Close();
            Console.WriteLine("Bağlantı Kapatıldı.");
        }
    }
}
```

Bu kodların çalışması sonucu aşağıdaki ekran görüntüsü oluşmaktadır.



SqlCommand

Uygulama ile SQL Server arasında sql cümlelerinin, saklı yordam (stored procedure) isimlerinin, gerekli parametre isimleri ve değerlerinin taşınmasını sağlayan sınıftır. Taşıdığı sql cümlesini ya da çağıracağı saklı yordamın ismini, SQL Server tarafında yürütme ve çıkan sonuçları ya da etkilenen kayıt sayısını geriye gönderme özelliklerine de sahiptir. SqlCommand sınıfının en sık kullanılan özellik ve metodları aşağıda kısaca incelenmektedir.

CommandText	Parametreli ya da parametresiz sql cümlelerinin, parametre bekleyen ya da beklemeyen saklı yordam isimlerinin atanabileceği özelliktir.
CommandType	CommandText özelliğinde belirtilen string ifadenin, saklı yordam olarak mı yoksa, Text ifade olarak mı değerlendirileceğine karar verilen özelliktir. Varsayılan olarak Text değerine sahiptir. Ancak saklı yordam (stored procedure) olarak kullanılacaksa; özellikle belirtilmesi gerekmektedir.
Connection	SqlCommand nesnesinin içerdiği sorguların hangi sunucu ve veritabanında çalıştırılması gerektiği ile ilgili bağlantı nesnesini tutan özelliktir.
ExecuteNonQuery()	Yazılan sql cümlesi ya da SQL Server tarafındaki saklı yordamın (strored procedure) çalıştırılması ve bu işlem ya da işlemler sonucunda etkilenen kayıt sayısının geriye alınabilmesini sağlayan metottur. Ancak burada dikkat edilmesi gereken nokta; geriye sadece tamsayı tipinde bir değer dönmesidir. Bu nedenle, genellikle veritabanına değer gönderirken yani, insert, update ve delete işlemleri için kullanılır, sonrasında da bu işlemlerden etkilenen kayıt sayısı geriye alınmaktadır. Böylece örneğin güncellenen satır sayısı elde edilip kullanılabilir.
ExecuteScalar()	ExecuteNonQuery metodu gibi, yazılan sql cümleleri veya saklı yordamları çalıştırmak için kullanılır. Bu metodun farklı olduğu nokta ise; geriye sadece tek bir **object** tipinde sonuç dönen sorgular için kullanılabilmesidir. Bu metotla, select işleminden dönecek olan scalar kayıtlar alınabilmektedir. Örneğin gruplama fonksiyonları içeren sorgu cümlelerinin sonuçlarını elde etmekte kullanılabilir.
ExecuteXmlReader()	Sadece SqlCommand sınıfında (class) bulunan bu metot; SQL Server üzerinden gelen verinin, XML (eXtensible Markup Language) formatında alınabilmesini sağlar.
ExecuteReader()	Geriye veri ya da veri blokları dönen sql cümleleri, saklı yordamları çalıştıran ve bunların sonuçlarının uygulama tarafına alınabilmesini sağlayan metottur. Bu metot ile uygulama tarafına gelecek sonuçlar, DataReader adı verilen ve veri sağlayıcıya (data provider) göre değişen bir tip yardımıyla okunabilmektedir. Özellikle içeriği çok fazla değişmeyen ve az sayıda satır içeren veri kümelerinin uygulama ortamına daha hızlı bir şekilde alınabilmesini sağlayan modellerde tercih edilmektedir.

Tablo 23: SqlCommand tipinin genel üyeleri

Örnek SqlCommand kullanımları aşağıdadır.

```
SqlCommand komut = new SqlCommand();
komut.CommandText = "Insert INTO Tablo(Alan1,Alan2,
Alan3) VALUES (deger1,deger2,deger3)";
komut.Connection = baglanti;
komut.CommandType = CommandType.Text;
baglanti.Open();
int etkilenenKayitSayisi = komut.ExecuteNonQuery();
baglanti.Close();

SqlCommand cmd = new SqlCommand();
cmd.CommandText = "UPDATE tablo SET alan1=deger1, alan2
=deger2 where alan=deger";
cmd.Connection=baglanti;
//commandtype belirtilmek zorunda değildir.
baglanti.Open();
cmd.ExecuteNonQuery();
baglanti.Close();

SqlCommand cmd = new SqlCommand();
cmd.CommandText = "select count(*) from tablo";
cmd.CommandType = CommandType.Text;
cmd.Connection = baglanti;
baglanti.Open();
int kayitsayisi = (int)cmd.ExecuteScalar();
baglanti.Close();

SqlCommand cmd = new SqlCommand();
cmd.CommandText = "SP_TumVerileriGetir";
cmd.CommandType = CommandType.StoredProcedure;
cmd.Connection = baglanti;
baglanti.Open();
SqlDataReader rdr = cmd.ExecuteReader();
baglanti.Close();
```

Şekil 164: SqlCommand Kullanım Örnekleri

BÖLÜM 2: BAĞLANTILI (CONNECTED) MODEL

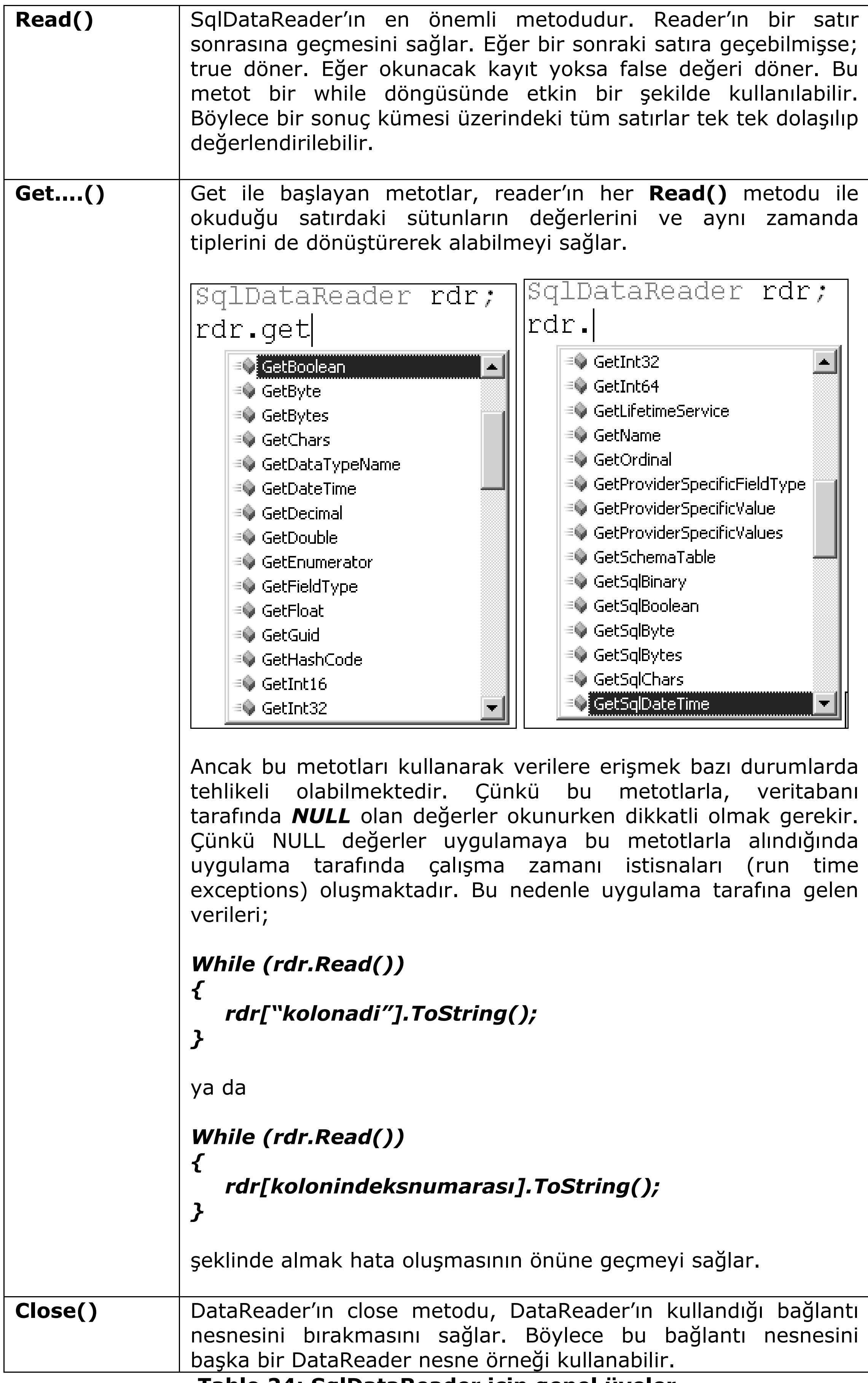
SqlDataReader

SQL Server üzerinde veya herhangi bir veri kaynağında duran veriler, veri sağlayıcıya bağlı connection ve command nesneleriyle alınabilmektedir. Ancak veri kaynağından gelen bu veriyi uygulama tarafında kullanabilmek için; bir şekilde o veriyi bellekte tutabilecek ve uygulamada ihtiyaç olduğu anda kullanılmasını sağlayacak başka bir bileşene, nesneye ihtiyaç vardır. Bu ihtiyacı karşılayan, veri sağlayıcıya özel **DataReader** sınıfları bulunmaktadır. Sql Server .NET Data Provider için bu sınıf, **SqlDataReader** olarak kullanılmaktadır.

SqlDataReader sınıfından oluşturulan bir nesne örneği, **SqlCommand**'ın **ExecuteReader**() metodundan dönen SqlDataReader örneğini yakalar. Bu nesne örneği uygulamalarda yazılan sql cümlesine bağlı olarak, gelen sonuç kümesini (result set) **sadece ileri yönlü (forward only) ve yanlız okunabilir (read only)** formatta kullanma şansı verir. Bu verinin daha hızlı bir şekilde uygulama ortamına alınmasını sağlar.

SqlDataReader'ın kullanımı sırasında dikkat edilmesi gereken önemli noktalar vardır. Örneğin; bir SqlDataReader, bir SqlConnection nesnesini kendine bağlar ve kayıtların sonuna gelinmeden bağlantının kapatılmasına izin vermez. Ayrıca bir SqlDataReader, üzerinde veri taşıdığı bağlantı nesnesini tamamen kendi kullanımına alır ve kapatılmadan önce bağlantı nesnesini bırakmaz. Ancak bu soruna .NET 2.0'da, MARS (Multiple Active Result Sets) adı verilen teknoloji ile çözüm bulunmuştur. Ne varki bu teknolojinin kullanılabilmesi için ilgili veritabanı sisteminin MARS'a destek veriyor olması gerekmektedir. Örneğin Ms Sql Server 2005' in doğrudan MARS desteği varken bu durum Ms Sql Server 2000 ve önceki sürümleri için geçerli değildir. SqlDataReader'ın en sık kullanılan özellik ve metotları şöyledir.

<table>
<tr><td>Read()</td><td>SqlDataReader'ın en önemli metodudur. Reader'ın bir satır sonrasına geçmesini sağlar. Eğer bir sonraki satıra geçebilmişse; true döner. Eğer okunacak kayıt yoksa false değeri döner. Bu metot bir while döngüsünde etkin bir şekilde kullanılabilir. Böylece bir sonuç kümesi üzerindeki tüm satırlar tek tek dolaşılıp değerlendirilebilir.</td></tr>
<tr><td>Get….()</td><td>Get ile başlayan metotlar, reader'ın her Read() metodu ile okuduğu satırdaki sütunların değerlerini ve aynı zamanda tiplerini de dönüştürerek alabilmeyi sağlar.



Ancak bu metotları kullanarak verilere erişmek bazı durumlarda tehlikeli olabilmektedir. Çünkü bu metotlarla, veritabanı tarafında NULL olan değerler okunurken dikkatli olmak gerekir. Çünkü NULL değerler uygulamaya bu metotlarla alındığında uygulama tarafında çalışma zamanı istisnaları (run time exceptions) oluşmaktadır. Bu nedenle uygulama tarafına gelen verileri;

```
While (rdr.Read())
{
    rdr["kolonadi"].ToString();
}
```

ya da

```
While (rdr.Read())
{
    rdr[kolonindeksnumarası].ToString();
}
```

şeklinde almak hata oluşmasının önüne geçmeyi sağlar.</td></tr>
<tr><td>Close()</td><td>DataReader'ın close metodu, DataReader'ın kullandığı bağlantı nesnesini bırakmasını sağlar. Böylece bu bağlantı nesnesini başka bir DataReader nesne örneği kullanabilir.</td></tr>
</table>

Tablo 24: SqlDataReader için genel üyeler

Buraya kadar bahsedilen SqlConnection, SqlCommand ve SqlDataReader sınıfları ile geliştirilen uygulamalar, **Bağlantılı (Connected)** veri erişim modeli olarak isimlendirilmektedir. Bağlantılı model olarak isimlendirilmesinin nedeni, ***DataReader nesnesiyle veriyi sunarken, son veriye gelene kadar bağlantının açık olmasının zorunlu olmasıdır.***

Örnek Uygulama

Bu örnekle bağlantılı modelde, veritabanından veri nasıl alınır, veritabanına nasıl veri gönderilir, mevcut veriler nasıl güncellenir ya da nasıl silinir, bu işlemler için neler yapılmalı, nelere ihtiyaç olacağı konuları incelenmektedir.

İlk olarak veritabanı ve tablolar oluşturulmalıdır. Senaryo, KATILIMCI kayıtlarının tutulacağı basit bir sistem üzerinden ilerleyecektir. Bu nedenle SQL Server 2005 üzerinde aşağıda görülen SDS adındaki veritabanı içerisinde KATILIMCI tablosu oluşturulmaktadır.

Column Name	Data Type	Allow Nulls
KatID	int	☐
KatIsim	nvarchar(50)	☐
KatSoyIsim	nvarchar(50)	☐
KatDogTar	datetime	☑
		☐

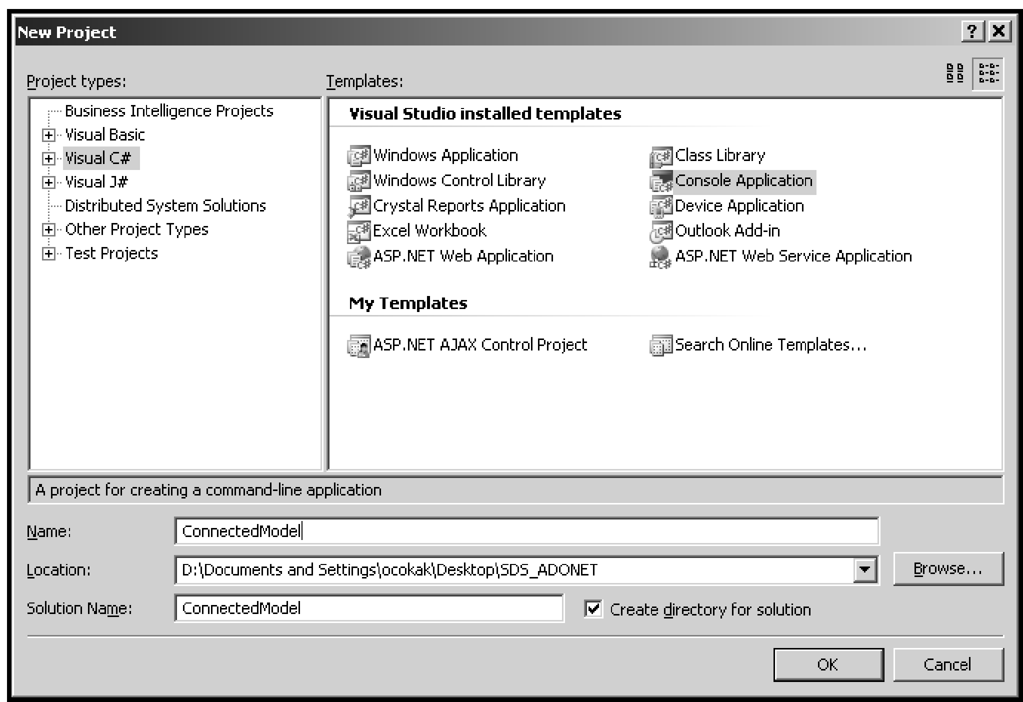
Şekil 165: KATILIMCI tablosunun alanları

Burada KatID kolonu Primary Key ve Identity yani otomatik olarak artan bir kolondur. Ayrıca dikkat edilmesi gereken diğer bir nokta da KatDogTar adındaki alanın NULL değerler alabileceğidir. Bu nokta önemlidir çünkü uygulama tarafından bu alana değer gönderilme zorunluluğu olmamasına rağmen, diğer alanlar için aynı şart geçerli değildir. Tabloya birkaç test verisi girilerek kodlamaya geçilebilir.

KatID	KatIsim	KatSoyIsim	KatDogTar
1	Osman	ÇOKAKOĞLU	01.01.2007...
2	Burak Selim	ŞENYURT	02.02.2006...
3	Bülent	SÖZGE	03.03.2005...
4	Özgür	ALTUNTAŞ	NULL
NULL	NULL	NULL	NULL

Şekil 166: KATILIMCI tablsou için örnek veriler

Uygulama konsol uygulaması (Console Application) olarak açılacak ve öncelikle mevcut veriler uygulama tarafına alınacaktır.



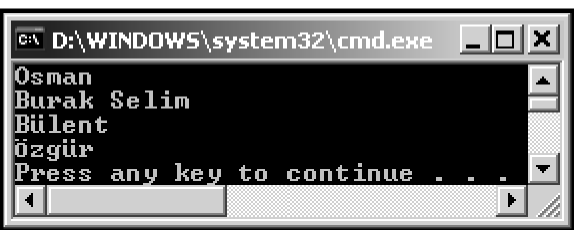
Şekil 167: Proje şablonunun seçimi

Öncelikle şunu belirtmek gerekir; veritabanı Sql Server 2005 olması nedeniyle kullanılacak veri sağlayıcısı, Sql Server .NET Data Provider'dır. Demek ki kullanılacak tipler, *System.Data.SqlClient* isim alanı altında yer almaktadır. Bu durumda ilk olarak bu isim alanı **using** bloğu ile uygulamaya eklenmelidir. Bu bir zorunluluk olmamakla beraber, kod içerisinde ilgi isim alanındaki tiplere daha kolay ulaşılmasını sağlayacaktır ki buda kodun okunabilirliğini arttırır.

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Data.SqlClient;

namespace ConnectedModel
{
    class Program
    {
        static void Main(string[] args)
        {
            SqlConnection baglanti = new SqlConnection();
            //baglanti nesnesi oluşturulurken, yapıcı metodun aşırı
yüklenmiş versiyonları da kullanılabilir.
            baglanti.ConnectionString = "Data Source=.;Initial
Catalog=SDS; Integrated Security=True";
            SqlCommand cmd = new SqlCommand();
            //Command nesnesi oluşturulurken, yapıcı metotlardan
(Constructor) yararlanılabilir.
            //cmd nesnesinin çalıştıracağı SQL cümlesi yazılır.
            cmd.CommandText = "SELECT * FROM KATILIMCI";
            //cmd nesnesinin kullanacağı yani gideceği yeri gösteren
bağlantı nesnesi atanır.
            cmd.Connection = baglanti;
            //bağlantı açılır. Böylece veritabanı kullanıma hazır hale
gelir.
            baglanti.Open();
            //burada dikkat edilemesi gereken önemli bir nokta vardır.
SqlDataReader sınıfının yapıcı metodu yoktur. Ancak kodlarken izin varmış
gibi görülür. Fakat uygulamayı build ederken hata alırız. Bunun nedeni
SqlDataReader sınıfına ait yapıcı metodun tanımlandığı assembly dışında
çağırılamamasıdır. Çünkü, internal erişim belirleyicisi ile
```

```
işaretlenmiştir.
            //SqlDataReader rdr = new SqlDataReader();  //HATALI

            SqlDataReader rdr;
            //SqlDataReader nesne örneğine,Command'ın veritabanından
execute işlemi sonucu elde edilen sonuç kümesi bağlanır.
            rdr = cmd.ExecuteReader();
            //HasRows özelliği, SqlDataReader'ın gösterebileceği satırı
olup olmadığı bilgisini verir.
            if (rdr.HasRows)
            {
                //Read-only ve forward-only olarak alınan verilerin
hepsine erişebilmek için; baştan sona kadar birer birer ilerlemeyi
sağlayan while döngüsü hazırlanır
                while (rdr.Read())
                {
           //KatIsim alanının değerlerini sırasıyla ekrana yazdırır.
                    Console.WriteLine(rdr.GetString(1));
                }
            }
            //bağlantı kapatılıyor. Böylece veritabanı kullanıma
kapatılmış oluyor ya da başka kullanıcılar için hazır hale geliyor.
            baglanti.Close();

        }
    }
}
```

Bu şekilde uygulamayı çalıştırırsak;



Şekil 168: Tüm Katılımcıların SqlDataReader ile okunması

şeklinde bir çıktı verecektir. Burada dikkat edilirse; **GetString()** metodu kullanılmıştır. Ancak herhangi bir hata ile karşılaşılmamıştır. Çünkü KatIsim alanında NULL değer yoktur. Eğer KatDogTar alanının değeri bu şekilde alınmak istenirse;

```
while (rdr.Read())
{
//KatIsim alanı ve KatDogTar değerlerini sırasıyla ekrana yazdırır.
      Console.WriteLine(rdr.GetString(1)+ " "
                        +rdr.GetDateTime(3).ToLongDateString());
}
```

```
D:\WINDOWS\system32\cmd.exe
Osman 01 Ocak 2007 Pazartesi
Burak Selim 02 Şubat 2006 Perşembe
Bülent 03 Mart 2005 Perşembe

Unhandled Exception: System.Data.SqlTypes.SqlNullValueException: Data is Null. T
his method or property cannot be called on Null values.
   at System.Data.SqlClient.SqlBuffer.get_DateTime()
   at System.Data.SqlClient.SqlDataReader.GetDateTime(Int32 i)
   at ConnectedModel.Program.Main(String[] args) in D:\Documents and Settings\oc
okak\Desktop\SDS_ADONET\ConnectedModel\ConnectedModel\Program.cs:line 39
Press any key to continue . . .
```

Şekil 169: Null değer istisnası

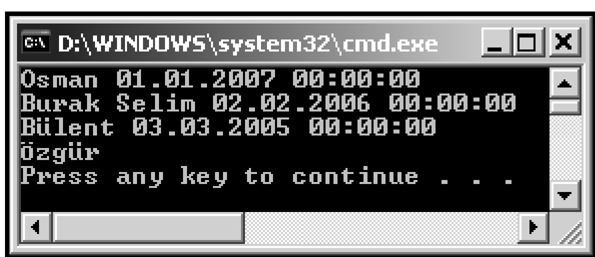
şeklinde bir hata mesajı alınır. Bu nedenle DataReader nesnesinden verileri alırken, her zaman bu olasılıklara karşı hazırlıklı olunmalıdır. Burada aslında yapılabilecek birkaç değişiklik vardır. Bunlardan bir tanesi,

rdr.IsDBNull(3)

metodu ile her değerin NULL olup olmama kontrolünü yapmaktır. Bu metod geriye Boolean bir değer dönmektedir. Diğer bir yol ise;

```
while (rdr.Read())
{
//KatIsim alanı ve KatDogTar değerlerini sırasıyla ekrana yazdırır.
      Console.WriteLine(rdr[1].ToString() + " " +
                                   rdr["KatDogTar"].ToString());
}
```

şeklinde kullanmaktır. Ancak burada da NULL olan değeri DateTime formatına dönüştürmeye çalışırken hata alınmaktadır. String olarak kullanıldığında ise sorun olmamaktadır.



Şekil 170: Null değer ile mücadele

Burada dikkat edilirse; SqlDataReader nesnesinden verileri alırken aslında bir kaç yol olduğu görülebilmektedir.

rdr.Get....(indeks) → Bu şekilde, değeri alınmak istenen kolon indeksi verilmektedir.

rdr[indeks] → Bu şekilde, değeri alınmak istenen kolon indeksi verilmektedir. Yukarıdaki kullanımdan farkı ise; NULL değerleri boş string değer olarak almasıdır.

rdr["kolonadi"] → Bu şekilde, değeri alınmak istenen kolonun adı belirtilmektedir. Ayrıca ikinci kullanımdaki gibi NULL değerler, boş string ifade olarak gelmektedir.

Burada incelenmesi gereken diğer bir nokta; While döngüsüdür. While döngüsü içinde bakılan koşul ifadesi aslında DataReader'a ait Read() metodundan dönen Boolean değerdir. Bu değer, her while döngü bloğu işletiminde bir satır okumak isteyecektir. Eğer yeni bir satır okuyabilirse; bu satırı, while bloğu içinde **rdr** ya da DataReader'a verilen değişken adı neyse o olarak kullanma imkanı verir. Demek ki her bir satır, **rdr** olarak ele alınmaktadır. Satır içindeki kolonları almak içinse; yukarıda bahsedildiği gibi bir kaç yol ve dikkat edilmesi gereken bazı noktalar vardır.

Bu şekilde verileri uygulama tarafına aldıktan sonra, uygulama tarafından veritabanına veri gönderme, güncelleme ve silme işlemlerinin nasıl yapılabileceğine değinilebilir. Ancak öncesinde SqlCommand, daha genel bir ifade olarak Command sınıfına ait **ExecuteScalar()** metodu için de, küçük bir örnek yapmak doğru olacaktır.

Bu örnekte, kayıtlı KATILIMCI sayısının alınacağı bir senaryo üzerinden ilerlenecektir. Temel olarak yapılması gereken işlemler yine aynıdır. SqlConnection nesnesini hazırladıktan sonraki kısmı tekrar yazılırsa;

```
SqlConnection baglanti = new SqlConnection();
//baglanti nesnesini oluştururken, Constructor'ın aşırı yüklenmiş versiyonları da kullanılabilir.
baglanti.ConnectionString = "Data Source=.;Initial Catalog=SDS; Integrated Security=True";
SqlCommand cmd = new SqlCommand();
//Command nesnesi oluşturulurken, Constructor'lardan yararlanılabilir.
//cmd nesnesinin çalıştıracağı SQL cümlesi yazılıyor.
cmd.CommandText = "SELECT COUNT(*) FROM KATILIMCI";
//cmd nesnesinin kullanacağı yani gideceği yeri gösteren connection nesnesi bağlanıyor.
cmd.Connection = baglanti;
//bağlantı açılıyor. Böylece veritabanı kullanıma açılmış oldu.
baglanti.Open();
int sayi=(int)cmd.ExecuteScalar();
//bağlantı kapatılıyor. Böylece veritabanı kullanıma kapatılmış oluyor ya da başka kullanıcılar için hazır hale geliyor.
baglanti.Close();
```

Kodlardan da anlaşılacağı gibi, geriye sadece bir tane sayı değeri gelecektir. Kayıt sayısı bir tam sayı olacaktır. Ancak Command'ın **ExecuteScalar()** metodu geriye **Object** tipinde bir değer dönmektedir. Bu durumda Cast işlemi yapmak gerekeceğini unutmamak gerekir.

Şimdide sırasıyla yeni kayıt ekleme, kayıt güncelleme ve kayıt silme işlemlerini gösteren örnekleri inceleyelim.

Yapılacak işlemler yine aynıdır. SqlConnection nesnesi oluşturulacak, ve bu bağlantı üzerinde, asıl işlemleri yapan SqlCommand nesnesinde değişiklikler yapılacaktır.

Kayıt Ekleme

```
SqlConnection baglanti = new SqlConnection();
baglanti.ConnectionString = "Data Source=.;Initial Catalog=SDS; Integrated Security=True";
SqlCommand cmd = new SqlCommand();
cmd.CommandText = "INSERT INTO KATILIMCI KatIsim,KatSoyIsim,KatDogTar) VALUES ('Selçuk','UZUN','05.05.2001')";
cmd.Connection = baglanti;
baglanti.Open();
int etkilenenKayitSayisi=cmd.ExecuteNonQuery();
Console.WriteLine(etkilenenKayitSayisi+ "Kayıt girildi.");
baglanti.Close();
```

Bu örnek için girilen veriler, bu şekilde elle yazılmak yerine kullanıcıdan ya da bir arayüzden de alınabilir. Bu bir windows, web uygulaması ya da mobil uygulama olabilir. Ancak verinin veritabanına gönderilmesi için mantık aynıdır. Sadece veriler dinamik bir şekilde dışarıdan alınmalıdır.

Kayıt Güncelleme

```
SqlConnection baglanti = new SqlConnection();
baglanti.ConnectionString = "Data Source=.;Initial Catalog=SDS; Integrated Security=True";
SqlCommand cmd = new SqlCommand();
cmd.CommandText = "UPDATE KATILIMCI SET (KatIsim='Değişen isim',KatSoyIsim='Değişen Soyisim' WHERE KatID=1)"
cmd.Connection = baglanti;
baglanti.Open();
int etkilenenKayitSayisi=cmd.ExecuteNonQuery();
Console.WriteLine(etkilenenKayitSayisi+ "Kayıt güncellendi.");
baglanti.Close();
```

Bu işlemde de kullanılan değerler, dinamik olarak dışarıdan alınabilmektedir.

Kayıt Silme

```
SqlConnection baglanti = new SqlConnection();
baglanti.ConnectionString = "Data Source=.;Initial Catalog=SDS;
Integrated Security=True";
SqlCommand cmd = new SqlCommand();
cmd.CommandText = "DELETE FROM KATILIMCI Where KatSoyIsim='UZUN'";
cmd.Connection = baglanti;
baglanti.Open();
int etkilenenKayitSayisi=cmd.ExecuteNonQuery();
Console.WriteLine(etkilenenKayitSayisi+ "Kayıt silindi.");
baglanti.Close();
```

Tüm bu işlemler sonucunda görülüyor ki; veritabanına veri eklemek, verileri güncellemek ya da veri silmek için gerekli olan temel bileşenler aynı, değişenler sadece veritabanı yönetim sisteminin bu işlemleri yapması için verilen sql komutlarıdır.

Bu işlemlerde dikkat edersek, **ExecuteNonQuery()** metodu kullanılmaktadır. Geriye herhangi bir sonuç getirmemesine rağmen, alınmak zorunda olunmayan ve bu işlemden etkilenen kayıt sayısını veren bir metoddur.

Tüm bu işlemlerin ardından veritabanındaki değerler aşağıdaki son halini almıştır.

	KatID	KatIsim	KatSoyIsim	KatDogTar
▶	1	Degisen isim	Değişen Soyisim	01.01.2007 00:...
	2	Burak Selim	ŞENYURT	02.02.2006 00:...
	3	Bülent	SÖZGE	03.03.2005 00:...
	4	Özgür	ALTUNTAŞ	NULL
*	NULL	NULL	NULL	NULL

Şekil 171: Silme işlemi sonucu tablonun son durumu

Böylece Connected (Bağlantılı) modelle ilgili temel bilgileri ve bu temel bilgilerin kullanıldığı örnekler tamamlanmış oldu.

Sırada **Disconnected (Bağlantısız)** katman hakkında temel bilgilerin ve örneklerin olduğu yeni bir bölüm var.

BÖLÜM 3: BAĞLANTISIZ (DISCONNECTED) MODEL

Bağlantısız model ilk duyulduğunda, veritabanı ile uygulama arasında bir bağlantı yokmuş gibi bir fikir oluşturabilir. Ancak hemen belirtmekte yarar var ki; Bağlantısız modelde de yine bir veritabanı, uygulama ile aynı makinada ya da uzak(Remote) bir makinada, yine bir uygulama ve aralarında mutlaka ve mutlaka bir bağlantı olmak zorunda. Bu bağlantı kablo bağlantısı, internet bağlantısı ya da aynı makinadalar ise; birbirlerini görebilen uygulamalar arasındaki bağlantı olmalıdır.
Peki nedir o zaman bu modelin bağlantısız olma özelliği?

Bu modelin bağlantısız olma özelliği, DataReader nesnesinde olduğu gibi ve o nesnenin en büyük sıkıntısı olan read-only, forward-only olması ve en önemlisi de **son satırı okuyana kadar, bağlantının açık kalması gerekliliğinin**, bu modelde olmayışıdır. Tabiki bu modelin, bu şekilde çalışabilmesini sağlamak için de farklı özelliklere sahip, çalışma mantığı farklı class'lar yazılmıştır.

Bu bölümde o yazılmış class'ları ve onların nasıl çalıştıkları incelenecektir. Ancak öncesinde bağlantısız katmanın çalışma modelinden bahsedilmelidir.

Günümüzde bir çok uygulamada aslında kullanılan model budur. Sabah iş yerine gelen ya da bir şekilde veritabanına bağlanan kişiler, veritabanından verileri alırlar ve akşama kadar o verilerle çalıştıktan sonra, bir şekilde veritabanına tekrar bağlantı kurup verileri veritabanına gönderirler. Bu senaryoya en güzel örnekler plasiyerlerin kullandığı, ecza deposu yetkililerinin kullandığı uygulamalardır. Mobil uygulamalar genellikle bu modelle çalışırlar.

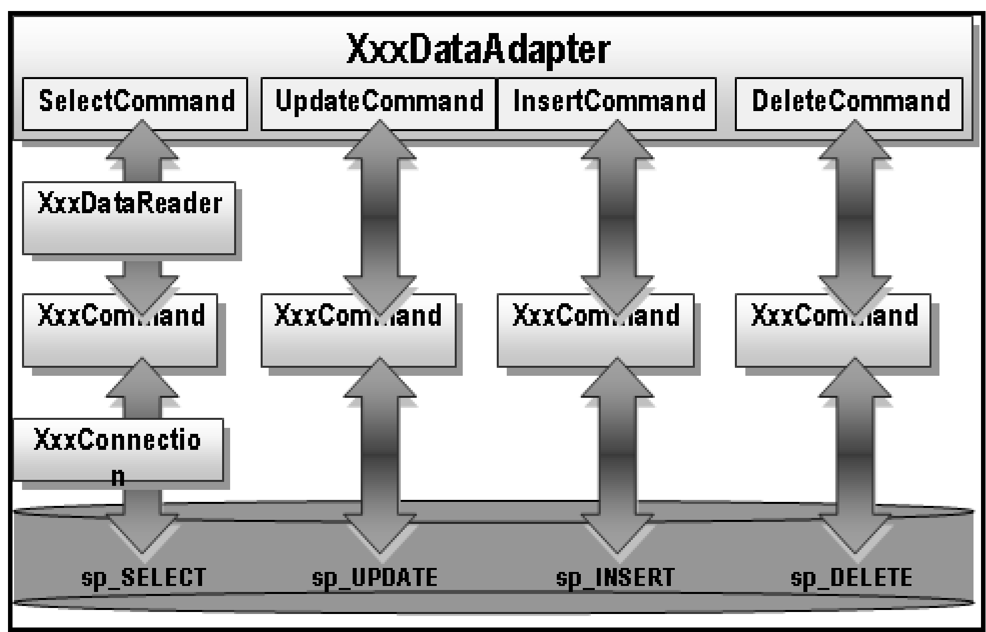
Bu modelde, veritabanıyla bağlantı kurulur. Ardından istenen verileri getirecek olan sql cümlesi yazılır. Bu cümleleri taşıyacak olan ve dönecek verileri tutacak olan nesneler yardımıyla veriler belleğe alındıktan sonra, veritabanıyla uygulama arasındaki bağlantı kapatılabilir. Ardından bellekteki veriler üzerinde istenilen şekilde çalışılabilir. İstenen veri kümesinin, istenen satırının, istenen kolonundaki değerine erişilebilir, değeri güncellenebilir ve hatta istenen satırlar silinebilmektedir. Ancak bu modelin artılarının yanında, verilerin güncelliğini kaybetmesi, verilerin tutarlı olmaması gibi eksi yönleride vardır.

Şimdi de sırasıyla bu model için gerekli olan temel bileşenleri inceleyelim. Bu modelde de SqlConnection nesnesi aynı görevleri üstlenmiştir. O nedenle tekrar incelemek gerekmemektedir.

SqlDataAdapter

SqlDataAdapter class'ı bağlantısız modelin en güçlü bileşenidir. Bu nesne ile veritabanına bağlantı kurulduktan sonra yazılan sql cümlesini çalıştırıp, dönecek olan sonuç kümesini, bu sonuçları bellekte tutacak olan nesnelere yükleme işlemi yapılmaktadır. Bu işlem aslında veri getirme işlemi olarak düşünülmektedir. Ancak SqlDataAdapter class'ı sadece veri getirmek için kullanılmaz, getirdiği bu veri üzerindeki değişiklikleri, yeni eklenen satırları, ve bellekte bulunan veri üzerinden silinen satırları tekrar veritabanına yansıtmak için de kullanılan bileşen budur.
Bu işlemleri yapabilmesi için bu bileşende, Select(Seçme) işlemleri için **SelectCommand**, insert (ekleme) işlemleri için **InsertCommand**, delete (silme) işlemleri için **DeleteCommand** ve update(güncelleme) işlemleri için **UpdateCommand** adı verilen 4 tane Command vardır. Bu command'lar belirli işlemleri yapmak için özelleşmişlerdir.

Burada bu command'lardan SelectCommand incelenecektir. Bu command'lardan herbiri, bağlantılı modelde incelenen SqlCommand'ın özelliklerini aynen taşımaktadır.



Şekil 172: DataAdapter Mimarisi

SelectCommand olarak bağlanan Command bir "**select**" cümlesi taşımalı ve DataAdapter daha sonra bu select cümlesinden gelecek sonucu bir yerlere yükleyebilmelidir. Bu yükleme işlemini DataAdapter, **Fill()** adı verilen metodu ile yapmaktadır.

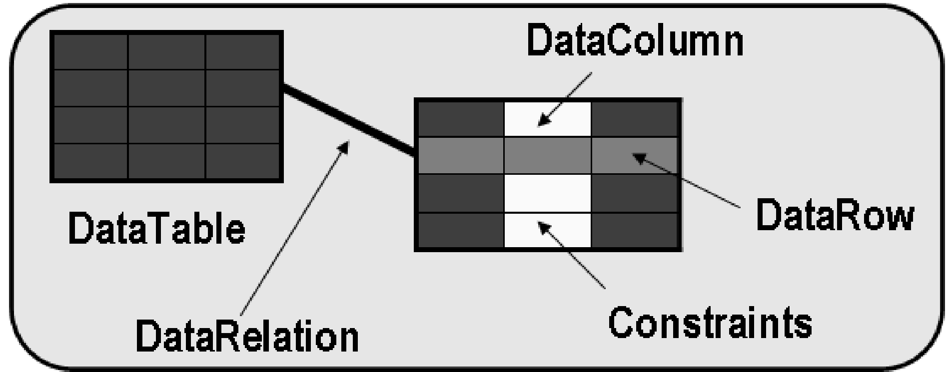
Yine bağlantılı modelde dikkat etmek gereken noktalardan bir tanesi bağlantının açılması ve kapatılmasıydı. Bu modelde ise; bağlantının açılması ve kapatılması işlemleriyle DataAdapter kendisi ilgilenmekte ve ihtiyaç duyduğunda yani verinin gelmesi istendiği anda bağlantıyı açar ve işlem tamamlandığında kendisi kapatır.

Ayrıca SqlDataAdapter'in **Update()** adında yine çok önemli bir metodu daha vardır. Bu metot burada incelenmemektedir. Ancak kısaca bahsetmek gerekirse; bu metod aracılığıyla, parametre olarak verilen bellekteki bir tablo, daha önce ayarlanmış insert, update ve delete command'lara göre veritabanına yansıtılmaktadır. Ayrıca unutulmamalıdır ki, DataAdapter class'ı da provider'a göre özelleşmiştir ve varsayılan command'ı, **SelectCommand**'dır.

```
SqlConnection baglanti = new SqlConnection();
baglanti.ConnectionString = "Data Source=.;Integrated Security=True;
initial catalog=SDS";
SqlDataAdapter adaptor = new SqlDataAdapter();
SqlCommand cmd = new SqlCommand("SELECT * FROM KATILIMCI",baglanti);
adaptor.SelectCommand = cmd;
DataSet ds = new DataSet();
adaptor.Fill(ds);
```

Yukarıdaki kod bloğunda da görüldüğü gibi, adaptor adında bir nesne oluşturulmuştur ve bu nesnenin selectcommand adındaki özelliğine, select işlemi yapan ve sonuc kümesi veren bir SqlCommand bağlanmıştır. Ardından bu işlemlerden çıkacak sonuçları tutmak için kullanılacak **DataSet** adındaki diğer bir class'tan nesne oluşturulmuştur ve bu nesneye gelen verileri doldurması için adaptor nesnesinin **Fill()** metodu kullanılmıştır. Peki buradaki DataSet class'ının özellikleri nelerdir?

DataSet



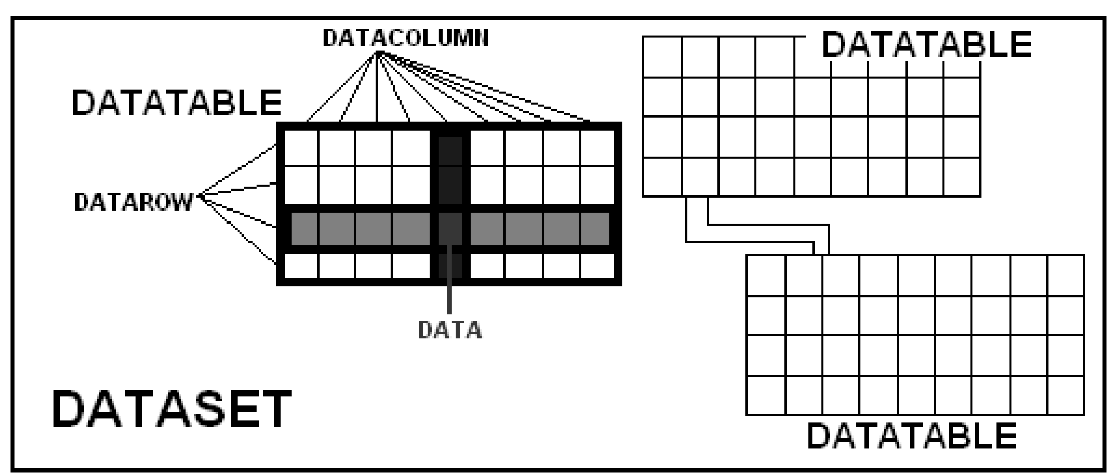
Şekil 173: DataSet Mimarisi

Yukarıdaki şekildende anlaşılacağı üzere DataSet, aslında veritabanının uygulama tarafındaki versiyonu olarak düşünülebilir. Veritabanının bire-bir kopyası olarak kullanılabilecek DataSet, uygulamanın çalıştığı makinanın belleğine yerleşir ve orada sanki veritabanıymış gibi kullanılabilir. Mesela bir DataSet üzerinde sorgulamalar, güncellemeler, eklemeler ve kayıt silme işlemleri yapılabilmektedir. Tabiki şunu da karıştırmamak gerekir. Bir DataSet veri tutmaz, veriyi DataSet içerisindeki **DataTable** adı verilen nesneler tutmaktadır. Bu nedenle, DataAdapter class'ının **Fill()** metoduna da DataSet'i doldurması söylendiğinde, aslında DataSet'in içerisine bir tane DataTable açar ve onun içerisine verileri atar. Ayrıca bir DataSet içerisinde birden çok DataTable olabilir ve bu tablolar arasında **DataRelation** adı verilen bağlantı nesneleri yer almaktadır.
DataTable nesnesi de aslında içinde bir çok bileşen bulundurmaktadır. Bunlardan en önemlisi ve tablonun yapısını belirten **DataColumn** class'ıdır. Bu classla, kolonun yapısı, özellikleri, adı, tipi, alabileceği verinin büyüklüğü vb. bilgiler tutulmaktadır. Diğer önemli bir bileşen de **DataRow'**dur. DataTable içerisindeki her bir satır, DataRow olarak belirtilmektedir. Veriler, DataColumn ve DataRow'ların kesiştiği **hücre**lerde tutulmaktadır.

Demek ki bir tek değere erişmek için geçilmesi gereken noktalar şu şekilde özetlenebilir.

DataSet'in tablolarından ilgili tablo, bu tablonun satırlarından ilgili satır, bu satırın ilgili kolonu şeklinde ilerlemek gerekmektedir.

Bir DataSet'in tabloları bir koleksiyon, bir tablonun satırları ve sütunları da birer koleksiyondur. Herşey listeler üzerinden çalışmaktadır.



Şekil 174: DataSet Mimarisi

Bir dataset'i kodlarken;

Öncelikle DataSet class'ının bulunduğu **System.Data** isim alanına giriş yapılmalıdır. Bu isim alanı altında, DataSet, DataTable, DataColumn, DataRow yani temel, provider'dan bağımsız ADO.NET bileşenleri yer almaktadır.

```
//ds adında, DataSet class'ından bir nesne oluşturuluyor. DataSet'in adı
ise; DsKATILIMCILAR.
//Buradaki DsKATILIMCILAR ismi, veritabanı ismi olarak düşünülebilir.
            DataSet ds = new DataSet("DsKATILIMCILAR");
//ds'nin tablolarından ilkinin, ilk satırının, ilk sütunundaki değeri
verir.
            ds.Tables[0].Rows[0][0].ToString();
//ds'tablolarından "KATILIMCI" tablosunun 6. sıradaki satırının, "KatIsim"
kolonuna karşılık gelen değeri verir.
            ds.Tables["KATILIMCI"].Rows[5]["KatIsim"].ToString();
//DataTable class'ından bir nesne oluşturuluyor. Adı "KATILIMCI2". Bu
isim, dataset'in tablolarından ilgili datatable'a gitmek gerektiğinde
kullanılabilir.
            DataTable dt = new DataTable("KATILIMCI2");
//Oluşturulan DataTable nesnesi, bir DataSet nesnesinin tablo
koleksiyonuna ekleniyor.
            ds.Tables.Add(dt);
//Ds'nin tablolarından yukarıda eklenen "KATILIMCI2" tablosunun
satırlarına yeni bir DataRow ekleniyor.
            ds.Tables["KATILIMCI2"].Rows.Add(yenisatir);
//Ds'nin tablolarından aslında yukarıda eklenen "KATILIMCI2" tablosuna,
yani 2. tablosunun kolonlarına yeni bir DataColumn ekleniyor.
            ds.Tables[1].Columns.Add(kolonadi);
```

Yukarıda görülen kodlarda, bir Dataset'e alınan veriye nasıl erişilebileceği konusunda farklı yöntemler görülmektedir.
Peki bu DataSet herhangi bir veri kaynağından gelen verilerle nasıl doldurulur?
Bu noktada, yukarıda sayılan bağlantısız model bileşenleri devreye girmektedir. Yani öncelikle SqlConnection ardından SqlDataAdapter ve onun SelectCommand'ı, oluşturulmalı ve tabiki bu command'ın CommandText'i yazılmalıdır. Aslında sadece bunların oluşturulması yeterli olacaktır. Sonrasında, SqlDataAdapter'in **Fill()** metoduna, içine verilerin doldurulması istenen, DataTable nesnesini ya da DataSet nesnesini parametre olarak vermek, verilerin veritabanından uygulamaya taşınması için yeterli olacaktır.

```
SqlConnection baglanti = new SqlConnection("Data source=.;initial
Catalog=SDS;integrated security=True");
//SqlDataAdapter'ın varsayılan Command'ı, SelectCommand'dır. O nedenle
Constructor aracılığıyla SelectCommand'ın commandtext'i ve kullanacağı
bağlantı bilgisi girilebilmektedir.
            SqlDataAdapter adaptor = new SqlDataAdapter("SELECT * FROM
KATILIMCI",baglanti);
//DataTable nesnesi oluşturuluyor.
            DataTable dt = new DataTable();
//adaptor aracılığıyla, dt tablosu gelecek olan verilerle dolduruluyor.
            adaptor.Fill(dt);
//DataSet nesnesi oluşturuluyor.
            DataSet ds = new DataSet();
//ds'nin tablo listesine, veritabanından gelen verileri tutan dt de
ekleniyor. Aslında eklenmeden de tek başına kullanılabilmektedir.
            ds.Tables.Add(dt);
//ds'nin tablolarından ilki olan dt tablosundaki tüm isimleri ekrana
yazdırılıyor.
            for (int i = 0; i < ds.Tables[0].Rows.Count; i++)
            {
//ilk tablonun her bir satırı için bu blok çalışacaktır.
Console.Write(ds.Tables[0].Rows[i]["KatIsim"].ToString());
//Katılımcıların KatDogTar kolonundaki bilgilerine erişmek için
//KatDogTar kolonu 3 nolu indeksteki kolondur.
Console.Write("\t" + ds.Tables[0].Rows[i][3].ToString());
            }
```

```
//sadece 5. katılımcı bilgileri alınabilir
            ds.Tables[0].Rows[4]["KatID"].ToString();
//sadece 5. katılımcının soyisim bilgisini verir.
            ds.Tables[0].Rows[4][2].ToString();
```

Bu örnek de gösteriyor ki; Bağlantısız katmanda, bağlantının açılması ya da kapatılması ile uğraşmaya gerek kalmaz. Ayrıca bağlantılı modelde olduğu gibi, istenilen veriye erişmek için, tüm kayıtları baştan tek tek geçmek gerekmemektedir. Direkt istenilen veriye odaklanılabilmekte ve hatta değeri de değiştirilebilmektedir.

Aşağıdaki örnekte, Bağlantısız modeli anlayabilmek için gerekli olan en temel bileşenler ve nasıl kullanılacakları görülebilmektedir.

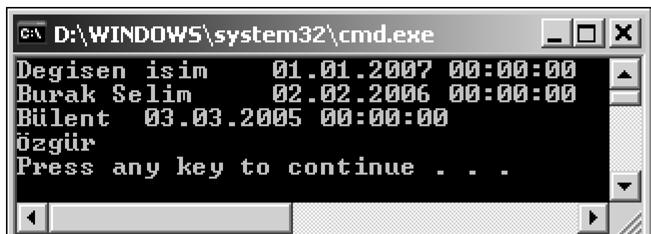
Örnek Uygulama

Kullanılacak veritabanı daha önceki örnekte de kullanılan SDS'dir. O nedenle bağlantı cümlesi yine aynıdır, yani aynı SqlConnection nesnesi kullanılabilir.
Bağlantısız katmanın en güçlü ve olmazsa olmaz bileşeni SqlDataAdapter class'ından bir tane nesne oluşturduktan sonra, o nesnenin SelectCommand özelliği için de bir tane SqlCommand oluşturulmalı CommandText ve Connection özellikleri ayarlanmalıdır. Ardından sadece SqlDataAdapter nesnesinin **Fill()** metodunu kullanarak gelecek olan verileri, ya DataTable nesnesine ya da DataSet nesnesine doldurup kullanmak kalmaktadır.

Unutulmamalıdır ki, bağlantısız katmanın bileşenlerinden DataSet, DataTable, DataRow ve DataColumn class'ları, provider bağımsız ve **System.Data** isim alanı içindedir. Ancak veritabanına bağlanmak ve hangi verilerin istendiğini söylemek için ve son olarak da bu verileri getirip ilgili nesnelere doldurmak için kullanılan, Connection, Command ve DataAdapter bileşenleri ise; provider'a özel olarak kullanılabilir. Bu örnekte, Sql Server .NET Data Provider kullanılarak ilerlenecektir.

```
using System.Data.SqlClient;
using System.Data;
……
…..

SqlConnection baglanti = new SqlConnection("Data source=.;initial
Catalog=SDS;integrated security=True");
//SqlDataAdapter'ın varsayılan Command'ı, SelectCommand'dır. O nedenle
Constructor aracılığıyla SelectCommand'ın commandtext'i ve kullanacağı
bağlantı bilgisi girilebilmektedir.
            SqlDataAdapter adaptor = new SqlDataAdapter("SELECT * FROM
KATILIMCI",baglanti);
//DataTable nesnesi oluşturuluyor.
            DataTable dt = new DataTable();
//adaptor aracılığıyla, dt tablosu gelecek olan verilerle dolduruluyor.
            adaptor.Fill(dt);
//DataSet nesnesi oluşturuluyor.
            DataSet ds = new DataSet();
//ds'nin tablo listesine, veritabanından gelen verileri tutan dt de
ekleniyor. Aslında eklenmeden de tek başına kullanılabilmektedir.
            ds.Tables.Add(dt);
//ds'nin tablolarından ilki olan dt tablosundaki tüm isimleri ekrana
yazdırılıyor.
            for (int i = 0; i < ds.Tables[0].Rows.Count; i++)
            {
//ilk tablonun her bir satırı için bu blok çalışacaktır.
Console.Write(ds.Tables[0].Rows[i]["KatIsim"].ToString());
//Katılımcıların KatDogTar kolonundaki bilgilerine erişmek için
//KatDogTar kolonu 3 nolu indeksteki kolondur.
                Console.Write("\t" + ds.Tables[0].Rows[i][3].ToString()+
"\n");
            }
```

Bu örneğin çalıştırılması sonucunda aşağıdaki çıktı oluşmaktadır.

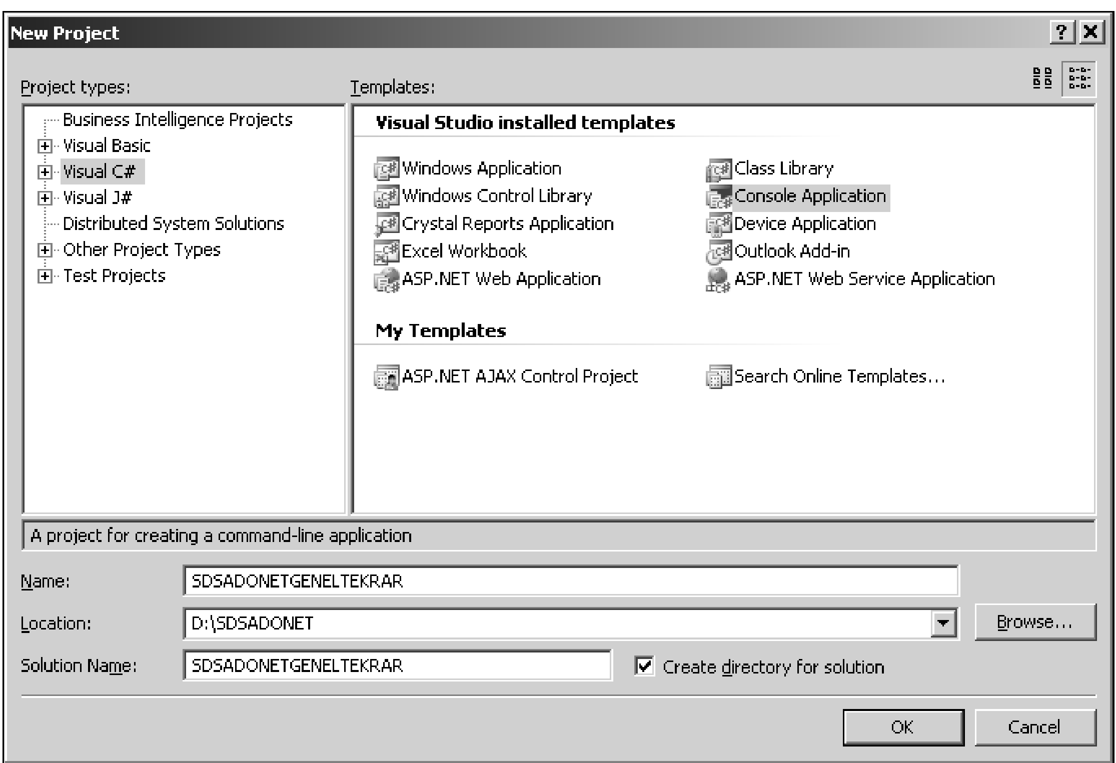


Şekil 175: DataTable içerisindeki satırların elde edilmesi

Tabiki bu konuda farklı örnekler verilebilir. Ancak senaryo hep aynı olacaktır. Değişen şey bağlantı cümlesi, ilgili veritabanına göre Provider ve bileşenler, istenen veriyi almak için de, Sql cümlesinin değişmesi yeterli olacaktır. Ardından hangi bilgilere ihtiyaç varsa; DataSet'in tablolarından ilgili tablonun ilgili satır ve sütunlarından veriler alınabilmektedir.

BÖLÜM 4: PROJE

Bu örnek, şu ana kadar anlatılan konuları aynı proje içerisinde kullanarak, birbirleriyle entegrasyonları tekrarlanmış olacaktır.
Örnek, Console Application projesi üzerinden yapılacaktır. Bu nedenle öncelikle Console Application projesi oluşturmak gerekmektedir.



Şekil 176: Proje şablonunun seçilmesi

Ardından, daha önce oluşturulan ve önceki örneklerde kullanılan SDS adındaki veritabanını kullananan ve bu veritabanı üzerinde farklı işlemler yapan kodlar aşağıdaki gibi oluşturulmaktadır.

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Data.SqlClient;

namespace SDSADONETGENELTEKRAR
{
    class Program
    {
        static void Main(string[] args)
        {
            TumKayitlar();
        }

        private static void TumKayitlar()
        {
            SqlConnection baglanti = new SqlConnection("Data
Source=.;Initial Catalog=SDS;Integrated Security=True");
            SqlCommand cmd = new SqlCommand("SELECT * FROM KATILIMCI",
baglanti);
            baglanti.Open();
            SqlDataReader rdr = cmd.ExecuteReader();
            Console.Write("KatID\t\tKatIsim\t\tKatSoyIsim\n\n");
```

```
            while (rdr.Read())
            {
                Console.Write("{0,-15}\t{1,-15}\t{2,-15}\n", rdr["KatID"],
rdr["KatIsim"], rdr["KatSoyIsim"]);
            }
            baglanti.Close();
        }
    }
}
```

Bu kodların çalıştırılması sonucu,

Şekil 177: SqlDataReader ile Katılımcı tablosundaki verilerin okunması

şeklinde bir çıktı alınmaktadır.

Daha sonra bu kodlara, yeni bir kayıt ekleyen aşağıdaki kodları eklenirse;

```
..............................
..............................
static void Main(string[] args)
        {
            TumKayitlar();
            YeniKayitEkle();
        }

        private static void YeniKayitEkle()
        {
            SqlConnection baglanti = new SqlConnection("Data
Source=.;Initial Catalog=SDS;Integrated Security=True");
            Console.Write("Kullanıcı Adı : ");
            string isim = Console.ReadLine();
            Console.Write("Kullanıcı SoyAdı : ");
            string soyisim = Console.ReadLine();

            SqlCommand cmd = new SqlCommand();
            cmd.CommandText = "INSERT INTO KATILIMCI (KatIsim,KatSoyIsim)
VALUES ('" + isim + "','" + soyisim + "')";
            cmd.CommandType = System.Data.CommandType.Text;
            cmd.Connection = baglanti;

            baglanti.Open();
            Console.WriteLine("Kayıt Veritabanına kaydediliyor.");
            int EtkilenenKayit = cmd.ExecuteNonQuery();
            Console.WriteLine(EtkilenenKayit + " adet kayıt kaydedildi.");
            baglanti.Close();
        }
............................
............................
```

```
D:\WINDOWS\system32\cmd.exe
KatID           KatIsim         KatSoyIsim

1               Degisen isim    Değişen Soyisim
2               Burak Selim     ŞENYURT
3               Bülent          SÖZGE
4               özgür           ALTUNTAŞ
Kullanıcı Adı : Osman
Kullanıcı SoyAdı : Çokakoğlu
Kayıt Veritabanına kaydediliyor.
1 adet kayıt kaydedildi.
Press any key to continue . . .
```

Şekil 178: Katılımcılar tablosuna kayıt ekleme

şeklinde bir çıktı oluşur ve SQL Server üzerindeki veritabanı aşağıdaki gibi değişmektedir.

	KatID	KatIsim	KatSoyIsim	KatDogTar
▶	1	Degisen isim	Değişen Soyisim	01.01.2007 00:...
	2	Burak Selim	ŞENYURT	02.02.2006 00:...
	3	Bülent	SÖZGE	03.03.2005 00:...
	4	Özgür	ALTUNTAŞ	NULL
	5	Osman	Çokakoglu	NULL
*	NULL	NULL	NULL	NULL

Şekil 179: Katılımcılar tablosunun son kayıt eklendikten sonraki hali

Kodlara girilen katılımcıID nin isim ve soyisim alanlarını güncelleyen kod blokları da aşağıdaki şekilde eklenirse;

```
     static void Main(string[] args)
        {
            TumKayitlar();
            //YeniKayitEkle();
            KayitGuncelle();
        }

        private static void KayitGuncelle()
        {
            SqlConnection baglanti = new SqlConnection("Data 
Source=.;Initial Catalog=SDS;Integrated Security=True");
            Console.Write("Güncellenecek Kullanıcı ID : ");
            int ID = Convert.ToInt32(Console.ReadLine());
            Console.Write("Yeni Kullanıcı Adı : ");
            string isim = Console.ReadLine();
            Console.Write("Yeni Kullanıcı SoyAdı : ");
            string soyisim = Console.ReadLine();

            SqlCommand cmd = new SqlCommand();
            cmd.CommandText = "UPDATE KATILIMCI SET KatIsim='" + isim + 
"',KatSoyIsim='" + soyisim + "' WHERE KatID="+ID;
            cmd.CommandType = System.Data.CommandType.Text;
            cmd.Connection = baglanti;

            baglanti.Open();
            Console.WriteLine("Kayıt Veritabanında Güncelleniyor.");
            int EtkilenenKayit = cmd.ExecuteNonQuery();
            Console.WriteLine(EtkilenenKayit + " adet Kayýt 
Güncellendi.");
            baglanti.Close();
        }
```

Bu kodlardan sonra çıktı aşağıdaki gibi oluşmaktadır.

```
D:\WINDOWS\system32\cmd.exe
KatID           KatIsim         KatSoyIsim

1               Degisen isim    Değişen Soyisim
2               Burak Selim     ŞENYURT
3               Bülent          SÖZGE
4               özgür           ALTUNTAŞ
5               Osman           Çokakoglu
Güncellenecek Kullanıcı ID : 1
Yeni Kullanıcı Adı : Güncellenen İsim
Yeni Kullanıcı SoyAdı : Güncellenen Soyisim
Kayıt Veritabanında Güncelleniyor.
1 adet Kayıt Güncellendi.
Press any key to continue . . .
```

Şekil 180: Katılımcılar tablosu üzerinde Update işlemi

Ayrıca veritabanında da değişiklik yapıldığı için, veritabanı da aşağıdaki gibi değişmektedir.

	KatID	KatIsim	KatSoyIsim	KatDogTar
▶	1	Güncellenen Isim	Güncellenen Soyisim	01.01.2007 00:...
	2	Burak Selim	ŞENYURT	02.02.2006 00:...
	3	Bülent	SÖZGE	03.03.2005 00:...
	4	Özgür	ALTUNTAŞ	NULL
	5	Osman	Çokakoglu	NULL
*	NULL	NULL	NULL	NULL

Şekil 181: Katılımcılar tablosunda güncelleme yapıldıktan sonraki hali

Son olarak da aşağıdaki kayıt silme kodları eklendiğinde;

```
      static void Main(string[] args)
        {
            TumKayitlar();
            //YeniKayitEkle();
            //KayitGuncelle();
            KayitSil();
        }

        private static void KayitSil()
        {
            SqlConnection baglanti = new SqlConnection("Data
Source=.;Initial Catalog=SDS;Integrated Security=True");
            Console.Write("Silinecek Kullanıcı ID : ");
            int ID = Convert.ToInt32(Console.ReadLine());

            SqlCommand cmd = new SqlCommand();
            cmd.CommandText = "DELETE FROM KATILIMCI WHERE KatID=" + ID;
            cmd.CommandType = System.Data.CommandType.Text;
            cmd.Connection = baglanti;

            baglanti.Open();
            Console.WriteLine("Kayıt Veritabanından Siliniyor.");
            int EtkilenenKayit = cmd.ExecuteNonQuery();
            Console.WriteLine(EtkilenenKayit + " adet Kayıt Silindi.");
            baglanti.Close();
        }
```

Bu kodların çalışmasından sonra ekran görüntüsü aşağıdaki şekilde olmaktadır.

```
D:\WINDOWS\system32\cmd.exe
KatID           KatIsim         KatSoyIsim

1               Güncellenen Isim        Güncellenen Soyisim
2               Burak Selim     ŞENYURT
3               Bülent          SÖZGE
4               özgür           ALTUNTAŞ
5               Osman           Çokakoglu
Silinecek Kullanıcı ID : 1
Kayıt Veritabanından Siliniyor.
1 adet Kayıt Silindi.
Press any key to continue . . .
```

Şekil 182: Katılımcılar tablosunda silme işlemi

Ayrıca veritabanının son hali de aşağıdaki gibi olmaktadır.

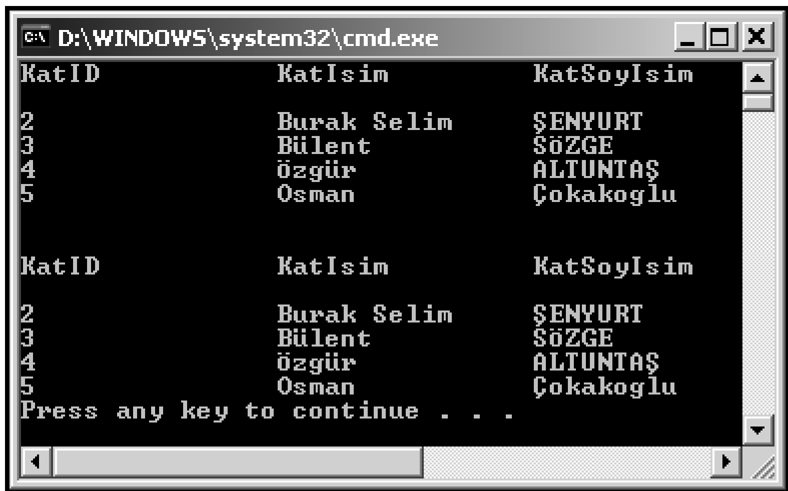
	KatID	KatIsim	KatSoyIsim	KatDogTar
▶	2	Burak Selim	ŞENYURT	02.02.2006 00:...
	3	Bülent	SÖZGE	03.03.2005 00:...
	4	Özgür	ALTUNTAŞ	NULL
	5	Osman	Çokakoglu	NULL
*	NULL	NULL	NULL	NULL

Şekil 183: Katılımcılar tablosunda silme işlemi yapıldıktan sonraki hali

Şimdi de son olarak veritabanındaki mevcut verileri bağlantısız model ile alan kodları uygulamaya eklenirse;

```
        static void Main(string[] args)
        {
            TumKayitlar();
            Console.WriteLine();
            Console.WriteLine();
            //YeniKayitEkle();
            //KayitGuncelle();
            //KayitSil();
            TumKayitlarBaglantisizModel();
        }

        private static void TumKayitlarBaglantisizModel()
        {
            SqlConnection baglanti = new SqlConnection("Data
Source=.;Initial Catalog=SDS;Integrated Security=True");
            SqlCommand cmd = new SqlCommand("SELECT * FROM KATILIMCI",
baglanti);

            SqlDataAdapter adaptor = new SqlDataAdapter();
            adaptor.SelectCommand = cmd;
            DataSet ds = new DataSet();
            baglanti.Open();
            adaptor.Fill(ds);
            baglanti.Close();
            Console.Write("KatID\t\tKatIsim\t\tKatSoyIsim\n\n");
            for (int i = 0; i < ds.Tables[0].Rows.Count; i++)
            {
                Console.Write("{0,-15}\t{1,-15}\t{2,-15}\n",
ds.Tables[0].Rows[i]["KatID"], ds.Tables[0].Rows[i]["KatIsim"],
ds.Tables[0].Rows[i]["KatSoyIsim"]);
            }
        }
```

Bu kodların çalıştırılması sonucu;



Şekil 184: DataTable üzerinden tablo satırlarının çekilmesi

şeklinde ekran görüntüsü oluşmaktadır.

Burada ilk liste bağlantılı modelle, alttaki ise; bağlantısız modelle hazırlanmıştır.

Böylece ADO.NET modülünde anlatılan bütün modüllerin tekrarlandığı örnek uygulamanında sonuna gelinmiştir.

KISIM SONU SORULARI:

1) Bağlantısız ortamın avantaj ve dezavantajlarını açıklayınız.

2) Bağlantılı ortamın avantaj ve dezavantajlarını açıklayınız.

3) Product tablosundan belirli bir alt kategorideki(ProductSubCategoryID' ye göre) ürünleri bağlantılı ortamı kullanaraktan çekip ekrana basan örnek kod parçasını bir konsol uygulaması üzerinden gerçekleştiriniz.

4) SQL Server kısmındaki ilgili soruda oluşturduğumuz Arkadas tablosuna kullanıcıdan aldığı bilgilerle insert, update, delete işlemlerini gerçekleştiren kod parçasını bir konsol uygulaması üzerinden gerçekleştiriniz.

5) Üçüncü soruyu bağlantısız ortam üzerinde yazınız.

6) Bir Connection nesnesinin işaret ettiği bağlantının o anki durumunu öğrenmek için, hangi sınıf üyesinden faydalanırız? Araştırınız ve örnek bir kod ile gösteriniz.

7) Product tablosundaki ürünlerden, rengi olanların ortalama fiyatını uygulama ortamına (konsol uygulaması) aktaran kod parçasını yazınız.

8) SqlDataAdapter' in kullandığı Command nesneleri ve görevleri nelerdir?

9) Bir DataSet' i ProductSubCategory ve Product tablosundaki verileride içerecek şekilde oluşturacak kod parçasını bir konsol uygulaması üzerinde yazınız.

10) Product tablosundaki bütün ürünlere %10 zam yapan kod parçasını bir konsol uygulaması üzerinde yazınız.

KISIM V: WINDOWS UYGULAMALARINA GİRİŞ

YAZAR: BURCU GÜNEL

BÖLÜM 0: WINDOWS UYGULAMALARI

Windows Uygulamaları Geliştirmek

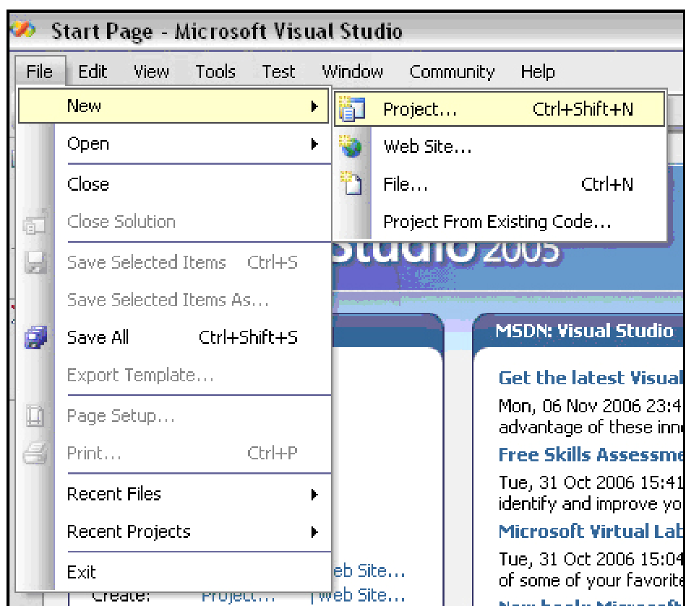
Windows formları, .NET Framework'ün bir parçasıdır ve ortak uygulama framework'ünü, yönetimsel çalışma ortamı, entegre güvenlik ve nesne yönelimli dizayn prensipleri dahil birçok teknolojiyi kullanır. Visual Studio 2005 uygulama geliştirme sistemi içerisindeki bu paylaşımlı uygulama geliştirme ortamında, .NET tabanlı herhangi bir dil kullanılabilir (C#, VB.NET gibi).

Formlar, uygulamalardaki kullanıcı arayüzlerinin basit ama temel elemanlarıdır. Windows uygulamalarında formlar, kullanıcıya bilgi sunmak ve kullanıcıdan bilgi almak için kullanılır. Formlar, görselliklerini tanımlamak için özellikler, davranışlarını tanımlamak için metotlar ve kullanıcı ile etkileşmek için olaylar (events) sunarlar. Uygulama geliştirici, uygulamanın ihtiyaçlarını karşılamak için bu üyeleri kullanarak formu istediği gibi düzenleyebilir. Nihayetinde .NET dünyasında her şeyin temeli sınıflardır (class) ve burada adı geçen form da aslında bir sınıftır. Uygulama geliştirici de, formu düzenlemek için, .NET framework uygulama geliştiricileri tarafından geliştirilen Form sınıfının üyelerini kullanır.

Kullanıcının uygulama içerisinde daha fazla aktif olması istenildiği durumlarda Windows uygulamaları kullanılır. Örneğin grafik çizimleri, veri girişi ve oyun gibi uygulamalarda kullanılabilir. Tüm bu uygulamaların kullanımı kullanıcının bilgisayarının ne kadar performanslı olduğuna bağlıdır. Bazı uygulamalar tam olarak kullanıcının bilgisayarına bağlı olarak çalışırken, bazıları için bağlantı (login) gerekmektedir. Örneğin, oyunlar kullanıcının bilgisayarına bağlı çalışan uygulamalarken, satış ile ilgili programlar bağlantı gerektirecek uygulamalara örnektir. Web uygulamalarından avantajı, daha performanslı olmasıdır. Fakat Web uygulamalarının dağıtımı, windows uygulamalarına göre daha kolaydır. Windows uygulamalarının kullanılabilmesi için kullanıcının bilgisayarına yüklenmesi gerekmektedir. Yüklendikten sonra, yalnızca kullanıcıda çalıştığı için Web uygulamalarından daha hızlı çalışmaktadır. Web uygulamalarından bir avantajı da, daha fonksiyonel ve daha güvenli olmasıdır.

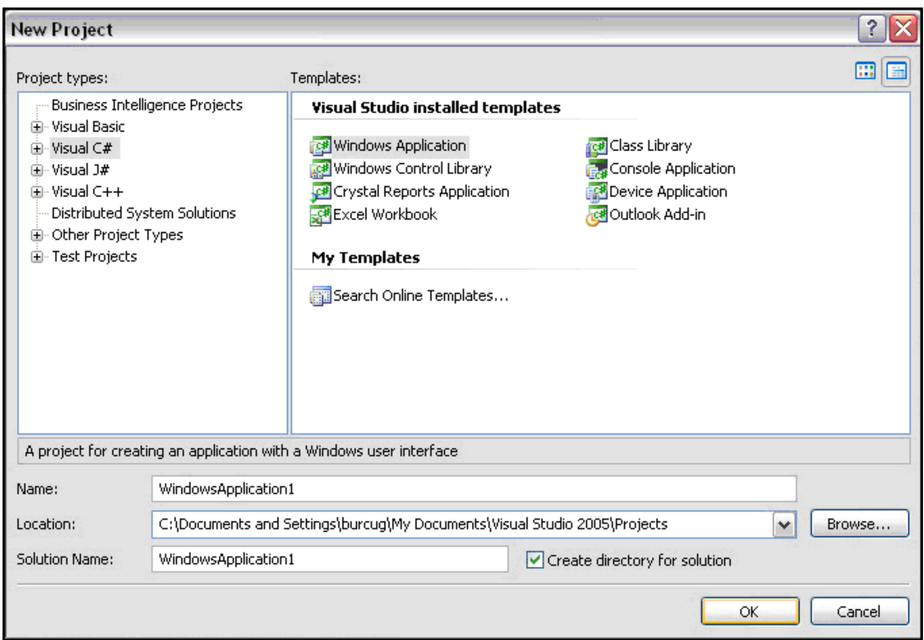
Bir Windows Uygulaması Oluşturmak

Yeni bir Windows uygulaması oluşturmak için Visual Studio 2005'de File → New → Project sekmesi takip edilir.



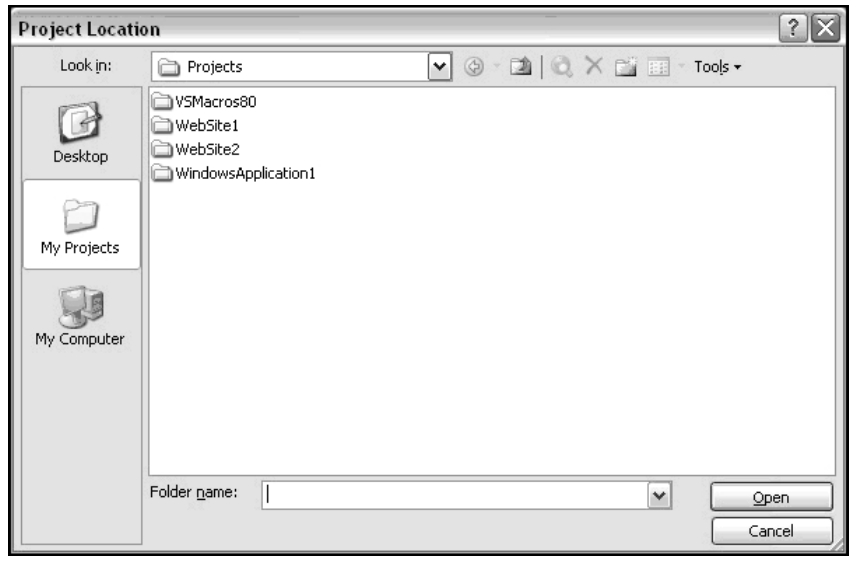
Şekil 185: Yeni proje oluşturmak

Daha sonra çıkan New Project penceresinden, Visual C# ve Windows Application seçilir.

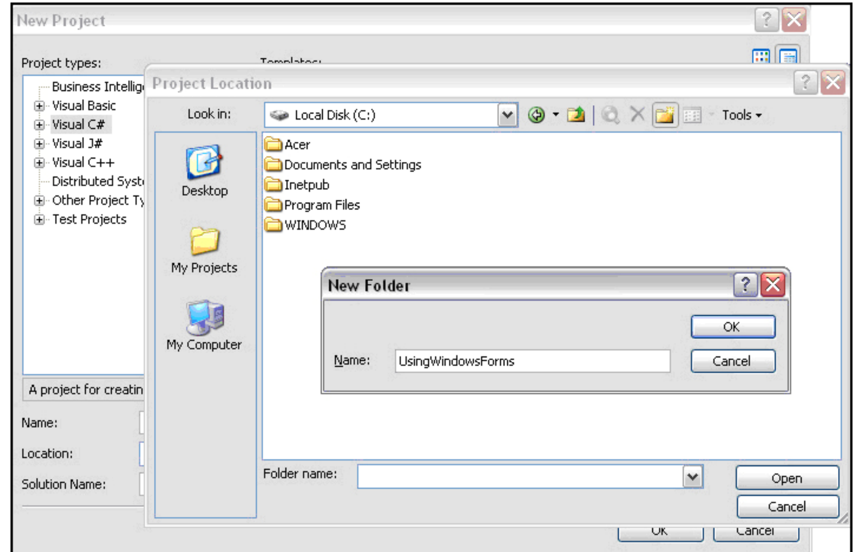


Şekil 186: Yeni bir Windows uygulaması oluşturmak

Yeni oluşturulan uygulama varsayılan olarak [root]:\Documents and Settings\...\My Documents\Visual Studio 2005\Projects altına kaydedilecektir. Eğer uygulama başka bir yere kaydedilmek istenirse, Location alanının yanında bulunan, Browse butonuna basılarak kaydedilecek yer belirlenebilir.

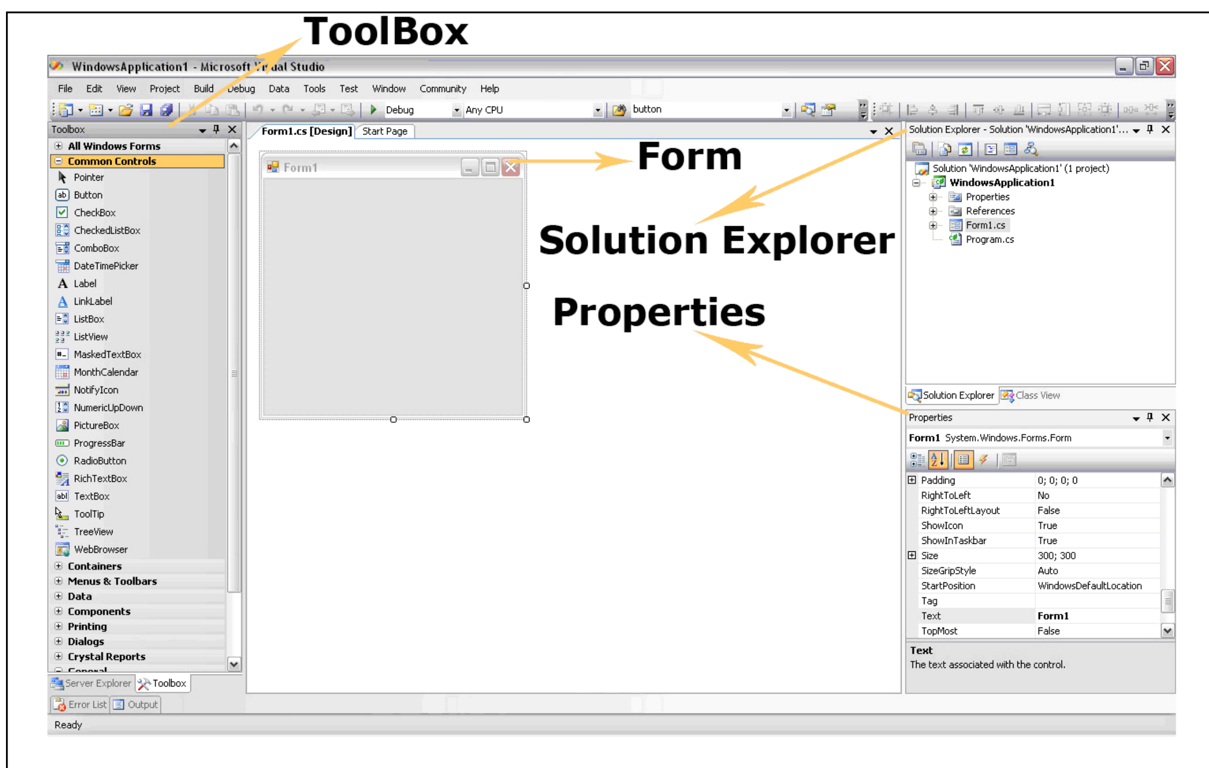


Şekil 187: Uygulamayı farklı bir yere kaydetmek



Şekil 188: Yeni Dosya oluşturmak

Project Location penceresinden kaydedilmek istenilen yer belirlendikten sonra, New Folder tıklanarak yeni bir dosya oluşturulur ve Project Location penceresinde Open diyerek, New Project penceresinde de OK butonuna basıldığı zaman yeni bir proje oluşturulmuş olur.



Şekil 189: Windows Uygulaması genel görünüm

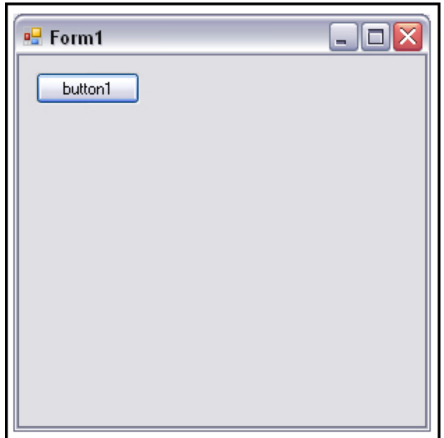
Bir Windows uygulaması oluşturulduğu zaman ekrana yeni bir Form gelir. Sol tarafta Windows kontrollerinin içerisinde bulunduğu ToolBox, sağ tarafta uygulamanın dosyalarının içerisinde bulunduğu Solution Explorer, kontrollerin ve forma ait özelliklerin değiştirilebileceği Properties penceresi bulunmaktadır.

Uygulamaya herhangi bir kontrol eklemek için Toolbox içerisinde yer alan kontrolleri sürükleyip bırakmak ya da kontrolün üzerine çift tıklamak yeterli olur.

Eğer uygulamaya yeni bir Form eklenmek istenirse Solution Explorer penceresinde Add → New Item diyerek yeni bir form eklenebilir.

Windows Formlarıyla Çalışmak

Bir windows formu ilk çalışırken, kullanıma hazır bir formun oluşması için **InitializeComponent** metodu çalışır. Formun üzerinde ki kontrollerin form üzerinde nerede konumlanacağı, kontrollere ait özellikler varsa özellikler, kontrollerin **olayları** (event) varsa olayları da, InitializeComponent metodunda bulunur. InitializeComponent metodu forma ait Form1 yapıcı metodu içerisinden çağırılır. (Yapıcı metodlar uygulama ilk çalıştığında, sınıflara ait nesneleri örneklerken sınıf içi alanlara ilk değerlerini vermek için kullanılır)



Şekil 190: Hazırlanacak Form örneği

```
public Form1()
{
    InitializeComponent();
}
```

InitializeComponent metodu Form1 yapıcı metodu içerisinden çağırılır.

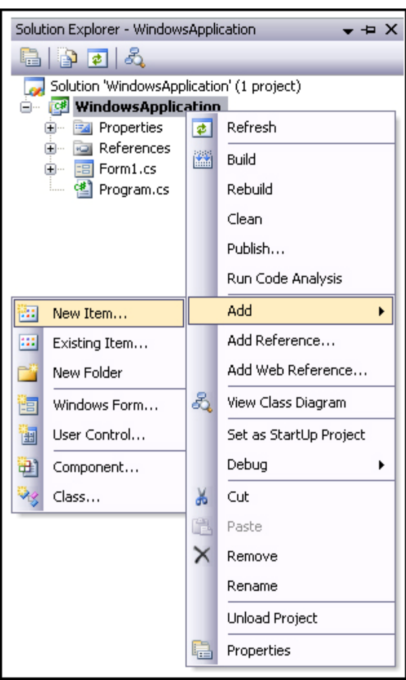
```
private void InitializeComponent()
{
    this.button1 = new System.Windows.Forms.Button();
    this.SuspendLayout();
    //
    // button1
    //
    this.button1.Location = new System.Drawing.Point(12, 12);
    this.button1.Name = "button1";
    this.button1.Size = new System.Drawing.Size(75, 23);
    this.button1.TabIndex = 0;
    this.button1.Text = "button1";
    this.button1.UseVisualStyleBackColor = true;
    this.button1.Click += new System.EventHandler(this.button1_Click);
    //
    // Form1
    //
    this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
    this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
    this.ClientSize = new System.Drawing.Size(292, 266);
    this.Controls.Add(this.button1);
    this.Name = "Form1";
    this.Text = "Form1";
    this.ResumeLayout(false);
```

```
}
```

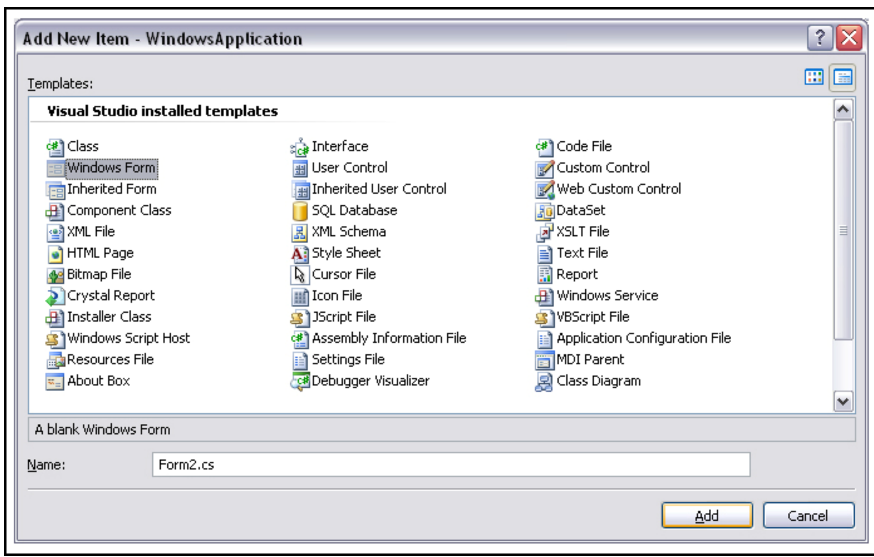
Forma ve form üzerinde bulunan butona ait konumlanacağı yer, özellikleri ve varsa olayları (event) InitializeComponent metodunda oluşur.

Bu noktada önerilen; bir nesnenin olay metoduna isim verilirken, ilgili olay ismi ile nesne isminin birlikte yer alması önerilir. Mesela Buton sınıfında alınan bir button1 nesnesine ait Click olayına bağlanacak olay metodu için; button1_click olarak belirtilmesi önerilir.

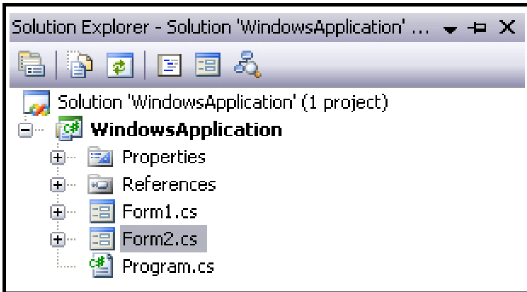
Bir windows uygulaması tek bir formdan oluşmak zorunda değildir. Birden fazla form olabilir ve bu formlar arasında geçiş yapılabilir. Örneğin bir formdan başka bir forma geçiş yapmak için öncelikle uygulamaya yeni bir form eklenmelidir. Uygulamaya yeni bir form eklemek için aşağıda yer alan adımlar izlenmelidir.



Şekil 191: Add → New Item



Şekil 192: Windows Form

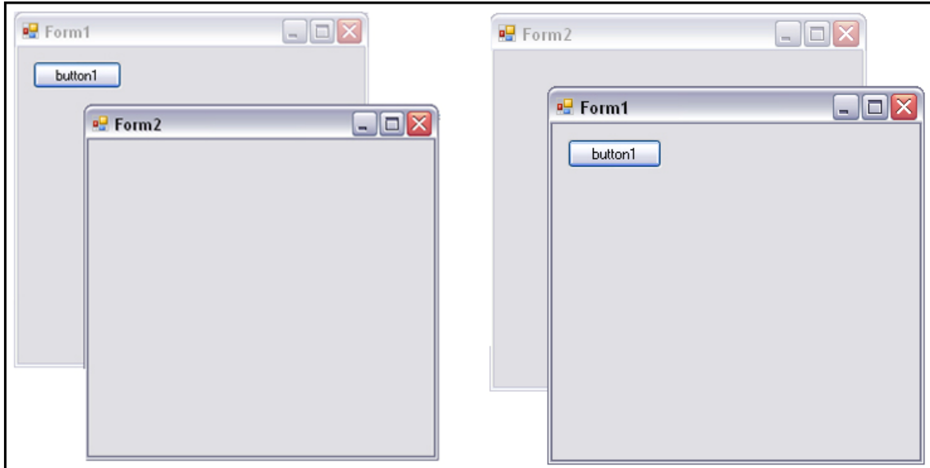


Şekil 193: Form eklendikten sonra Solution Explorer penceresinin son görünümü

Uygulamamızda yer alan Form1 formundan, Form2 formuna yönelmek için aşağıdaki kod kullanılabilir. Form1 formunda bulunan butonun click olayında (event) Form2 formunun bir nesne örneği alınır ve Show() metodu kullanılarak Form2 görüntülenir.

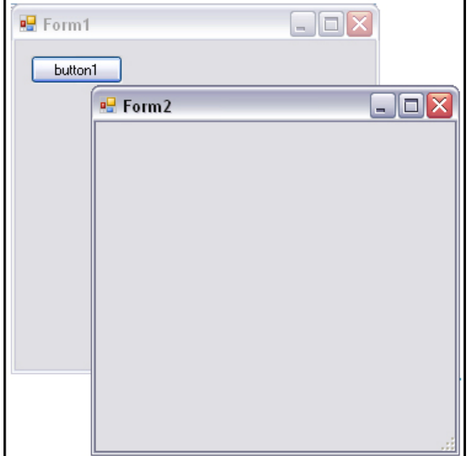
```
private void button1_Click(object sender, EventArgs e)
{
    Form2 frm2 = new Form2();
    frm2.Show();
}
```

Show() metodu ile Form2 formu çağırılınca Form1 formu hala kullanabilir ve erişilebilir olarak durmaktadır. Bu sıkıntının önüne geçmek için, yine show metodu gibi, başka bir formu görüntülemek için kullanılan ShowDialog() metodu vardır. ShowDialog() metodu ile Form2 formu açılırsa; Form1 formu, form2 kapatılmadığı sürece erişilemez olacaktır. Form1 formunu kullanabilmek için Form2 formunu kapatmak gerekmektedir. ShowDialog() metodu bir formu dialog box olarak gösterir.



Şekil 194: Show metodunun örnek kullanımı

```
private void button1_Click(object sender, EventArgs e)
{
    Form2 frm2 = new Form2();
    frm2.ShowDialog();
}
```



Şekil 195: ShowDialog metodunun örnek kullanımı

Show() metodu, ilgili formu görüntülemek için kullanılır ve bir yükleme (Load) süreci gerektirir. Show metodu çalışırken aşağıda yer alan yaşam döngüsü gerçekleşir.

- Load
- GotFocus
- Activated
- Closing
- Closed
- Deactive
- LostFocus
- Dispose

Load olayı (event) form gösterilmeden önce herhangi bir işlem yapmak için kullanılır. Burada forma ya da form üzerinde bulunan kontrollere özellikler verilebilir. Load olayı (event) bir form belleğe her yüklendiğinde tetiklenir. Bir uygulamanın çalışma sürecinde Load olayı (event) birçok kez çalışabilir.

Örneğin aşağıda yer alan örnekte olduğu gibi, form üzerinde bulunan butonun Text özelliği formun Load olayında (event), çalışma zamanında değiştirilebilir.

Şekil 196: Button nesnesinin Text özelliğinin çalışma zamanında değiştirilmesi

```
private void Form1_Load(object sender, EventArgs e)
{
    button1.Text = "BUTTON";
}
```

- Kullanıcı uygulamanın herhangi bir yerine odaklandığı zaman GotFocus olayı (event) çalışır.
- Kullanıcı uygulamayla olan etkileşimini kesince LostFocus olayı (event) çalışır.

- Kullanıcı uygulamayı kapatmak istediğinde, Closing olayı (event) gerçekleşir. Closing, uygulama kapanacağı zaman harekete geçer. Bu olayın oluşması sonucu tetiklenecek olay metodu içerisinde, kapatılma işlemi iptal edilebilir ve uygulama yine açık kalabilir. Ya da form açık

kalabilir.
- Closed olayı (event) ise; uygulama takatıldıktan sonra çalışan ve iptal edilme şansı olmayan olaydır.

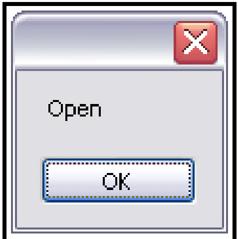
Uygulamada kullanıcıya bir hata vermek, uyarmak ya da yaptığı işlemin sonucuna ait bilgi vermek için, MessageBox sınıfının Show() metodu kullanılır.

Örneğin aşağıda yer alan örnekte olduğu gibi, bir bağlantı oluşturulup, MessageBox.Show() kullanılarak bağlantının açık mı, yoksa kapalı mı olduğuna bakılabilir.

Şekil 197: MessageBox sınıfının show metodunun kullanımı

```
private void btnUyari_Click(object sender, EventArgs e)
{
    SqlConnection con = new SqlConnection("data source=localhost;initial
catalog=AdventureWorks;integrated security=true");
    con.Open();
    string uyari = con.State.ToString();
    MessageBox.Show(uyari);
}
```

btnUyari butonuna tıklanınca bağlantının durumuna ait bilgi alınır. Sonucunda ekrana aşağıda yer alan Message Box çıkar.



Şekil 198: MessageBox' ın çalışmasının sonucu

MessageBox'ın iconu, üzerinde yer alan buton ve mesaj gibi özellikler değiştirilebilir. Aşağıda yer alan örnekte MessageBox'ın hata mesajı değiştirilmiştir. Show metodu, içerisine string tipinden uyarı çıkarmak için kullanıldığı gibi, MessageBox'ın görsel özelliklerini değiştirmek içinde kullanılır. Aşağıdaki örnekte MessajeBox'a bir isim verilmiştir.

```
private void btnUyari_Click(object sender, EventArgs e)
{
    MessageBox.Show("hata", "hata mesajı");
}
```

Şekil 199: MessageBox' ın çalışmasının sonucu

```
private void btnUyari_Click(object sender, EventArgs e)
{
    MessageBox.Show("hata", "hata mesajı",MessageBoxButtons.);
}
```

AbortRetryIgnore
OK
OKCancel
RetryCancel
YesNo
YesNoCancel

Şekil 200: MessageBoxButtons enum sabitinin değerleri

MessageBox üzerinde yer alan buton, Show() metodunun aşırı yüklenmiş versiyonları ile değiştirilebilir. MessageBoxButtons'dan sonra nokta operatörü kullanılarak OK, OKCancel gibi farklı butonlar seçilebilir.

Şekil 201: MessageBox ın üzerinde yer alan farklı tipte buton kombinasyonları

MessageBox üzerinde yer alan Icon, Show() metodunun aşırı yüklenmiş versiyonlarından ile değiştirilebilir. MesasageBoxIcon dan sonra nokta operatörü kullanarak Warning, Stop gibi farklı butonlar kullanılabilir.

```
private void btnUyari_Click(object sender, EventArgs e)
{
    MessageBox.Show("hata", "hata mesajı",MessageBoxButtons.OK,MessageBoxIcon.);
}
```

Asterisk
Error
Exclamation
Hand
Information
None
Question
Stop
Warning

Şekil 202: MessageBoxIcon enum sabitinin değerleri

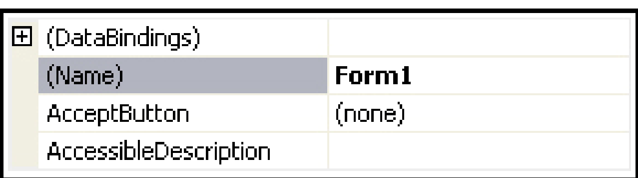
Şekil 203: MessageBox üzerinde yer alan farklı ikonlar

Form Özellikleri

Formlar Windows işletim sisteminin en temel kullanıcı arayüzleridir. Windows formları bilgi vermek ve almak için kullanılırlar.
Formlar uygulamanın içerisinde kullanılan ekranlardır. Kullanıcı ile etkileşim içerisinde olabilecek özelliklere sahiptirler. Yeni bir uygulama açılınca, kullanıcının karşısına bir form taslağı gelir. Bu taslak üzerinde ControlBox, Maximize, Minimize ve Close butonları bulunur. Aslında burada görülen form, yani arayüz, **System.Windows.Forms.Form** tipinden bir nesne örneğidir.

Name

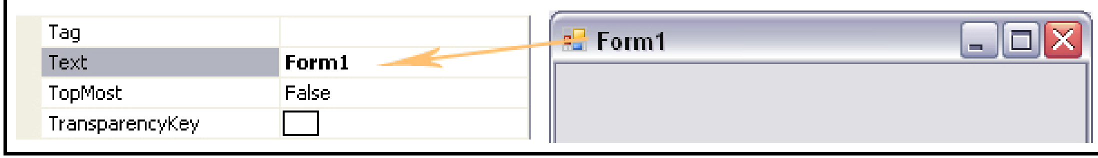
Formun adını değiştirmek için Name özelliği kullanılmalıdır. Kullanıcının göreceği isim değil, kodlamada kullanılacak isimdir.



Şekil 204: Name özelliği

Text

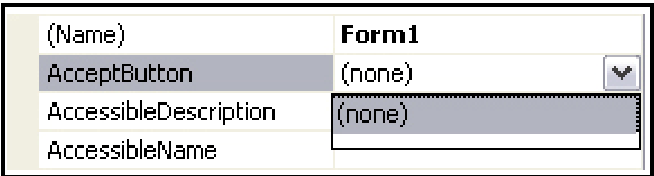
Formun ControlBox'da yer alan text alanını değiştirmek için Text özelliği kullanılır. Text özelliği formun kullanıcıya gösterilen ismidir.



Şekil 205: Text özelliği

AcceptButton

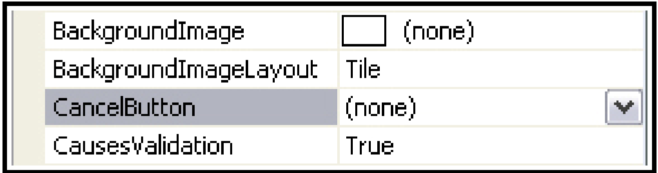
Kullanıcı, herhangi bir buton kontrolüne odaklanmamışsa(focus); Enter tuşuna bastığında hangi butonun tıklanmış olması isteniyorsa o buton, formun AcceptButton'u olarak belirlenir.



Şekil 206: AcceptButton özelliği

CancelButton

Kullanıcı Esc tuşuna bastığında, hangi butonun tıklanmış olması isteniyorsa; ilgili buton formun CancelButton'u olarak belirlenir.

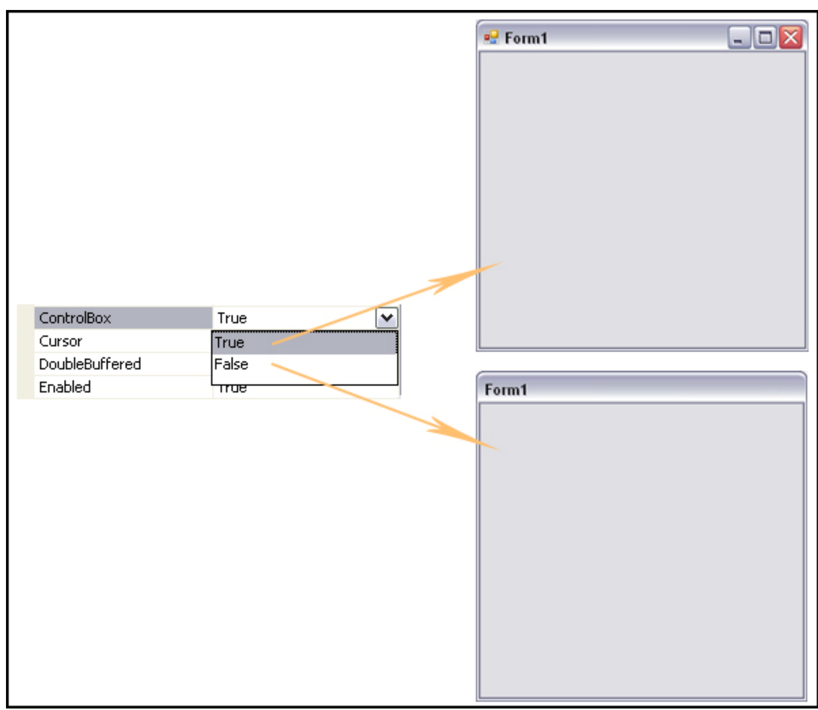


Şekil 207: CancelButton özelliği

AcceptButton ve CancelButton özelliklerini kullanabilmek için form üzerinde en az bir tane Button kontrolü bulunmalıdır.

ControlBox

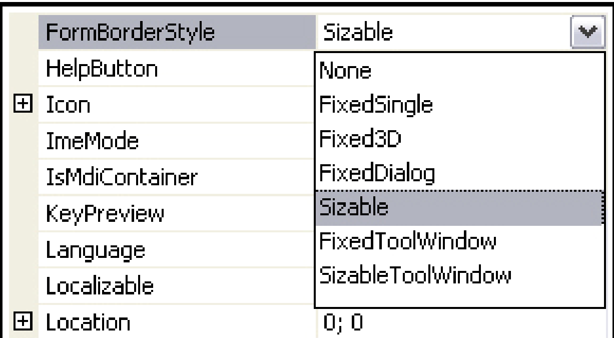
Formun Maximize, Minimize, Help ve Close buttonlarını üzerinde barındaran özelliktir. Varsayılan olarak değeri **true'**dur. Eğer false yapılırsa; Maximize, Minimize ve Close butonları görünmeyecektir.



Şekil 208: ControlBox özelliği

FormBorderStyle

Formun çerçevesinin görüntüsünü değiştirmek için ve ControlBox'daki hangi butonların kullanıcıya gösterileceğini belirlemek için kullanılır.

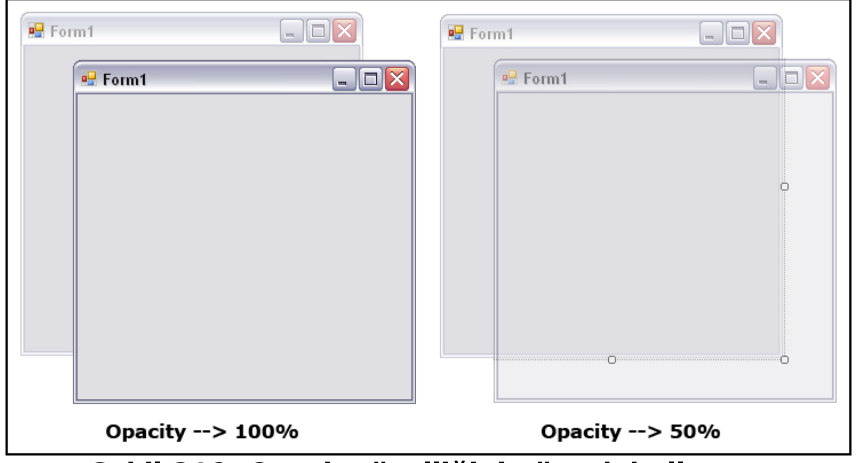


Şekil 209: FormBorderStyle özelliği

Varsayılan olarak bütün formlar için FormBorderStyle özelliği **Sizable** dır. Sizable olduğu durumda; kullanıcının uygulamayı kullanırken formun boyutunda değişiklik yapmasına izin verilir. **FixedSingle**, **Fixed3D**, **FixedDialog**, **FixedToolWindow** kullanıldığı durumlarda ise; boyutu ile oynanmasına izin verilmez. Ancak FixedToolWindow dışındaki değerlerde Maximize butonu kullanılarak formun boyutu büyütülebilir. FixedToolWindow ve SizableToolWindow değerleri, sadece Close butonunun görülmesi istenilen durumlarda kullanılır. SizableToolWindow değerinde kullanıcı, formun boyutu ile oynayabilir.

Opacity

Formun netliğini değiştirmek için kullanılmalıdır.

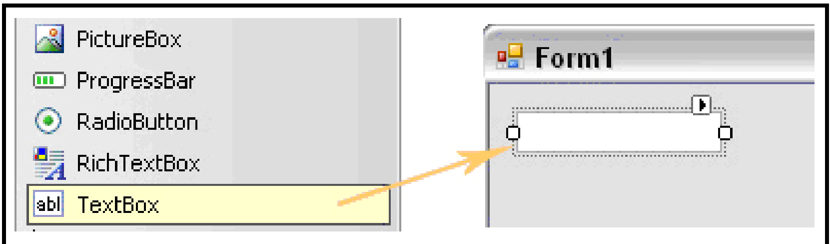


Şekil 210: Opacity özelliğinin örnek kullanımı

BÖLÜM 1: WİNDOWS KONTROLLERİ

TextBox

Windows formuna bir TextBox kontrolü eklemek için, sol tarafta yer alan Toolbox içerisindeki Common Controls bölümünde bir TextBox sürükleyip bırakılmalı ya da kontrolün üzerine çift tıklanmalıdır.

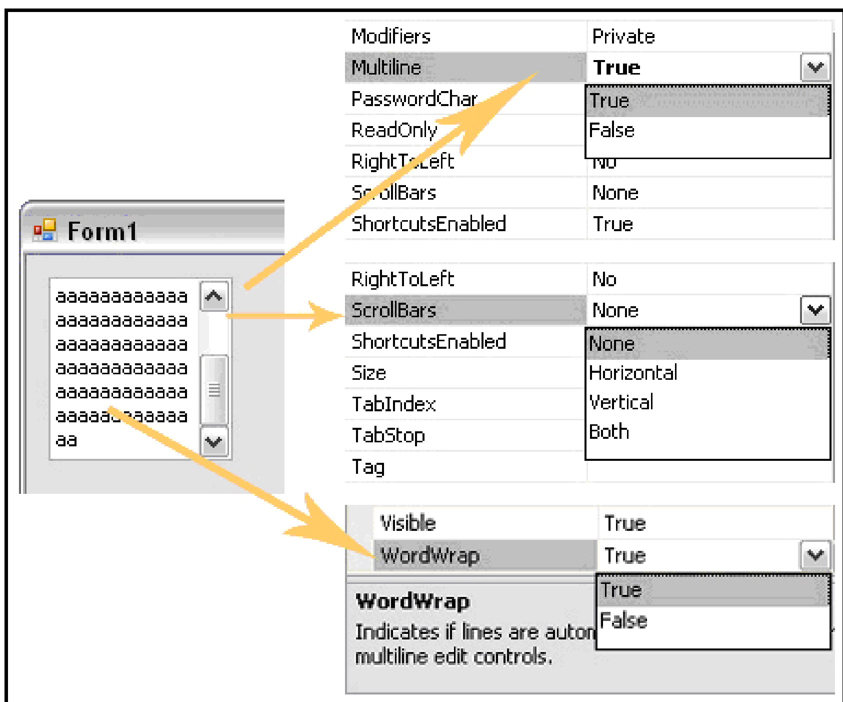


Şekil 211: TextBox kontrolü oluşturmak

Kullanıcıdan girdi almak için ya da bir metin alanı (text alanı) görüntülemek için TextBox kontrolü kullanılır. Genellikle güncellenebilen metin dosyaları(text dosyaları) için bu kontrolden faydalanılır. TextBox'daki text'ler birden fazla satırda görüntülenebilir, kontrolün boyutuna göre yazı düzenlenebilir ve biçimlendirilebilir(formatlanabilir).

MultipleLine, WordWrap ve ScroolBars Özelliği

Varsayılan hali ile TextBox, bilgileri tek satır halinde görüntüler ve kaydırma çubukları (scrool bars) görüntülenmez. Textbox'ın içerisine girilen metin Textbox'ın genişliğinden büyükse; metnin bir kısmı görünür. Bu durumu değiştirmek için **Multiline**, **WordWrap** ve **ScroolBars** özellikleri kullanılabilir. Property penceresinden Multiline özelliğini true, WordWrap özelliğini true ve ScroolBars özelliğini Horizantal, Vertical veya Both yaparak, bilgilerin birden fazla satırda görüntülenmesi sağlanabilir.



Şekil 212: Multiline, ScrollBars, WordWrap Özellikleri

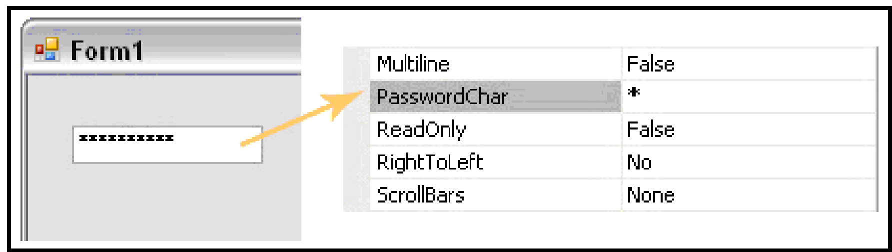
ScroolBars özelliği Horizantal yapılırsa, yazılan metinler yine aşağı doğru kayar, fakat kaydırma çubuğu görünmez. Vertical yapılırsa, yazılar aşağıya doğru kaydığı gibi kaydırma çubuğuda görünür. Both özelliği, WordWrap özelliği false iken yapılırsa kaydırma çubuğu hem yukarıdan aşağıya hemde soldan sağa görüntülenecektir.

Bu özellikler, dizayn zamanında yukarıdaki şekilde düzenlenebilirken; çalışma zamanında aşağıdaki kodlar kullanıarak da değiştirilebilir.

```
textBox1.ScrollBars = ScrollBars.Vertical;
textBox1.Multiline = true;
textBox1.WordWrap = true;
```

PasswordChar ve MaxLength Özelliği

Bir TextBox kontrolünün PasswordChar özelliğini değiştirmek için, Property penceresinden PasswordChar özelliğine kullanıcının girdiği şifrenin hangi karakterde görünmesi isteniyorsa o değer atanmalıdır. Bu özellik TextBox da görüntülenecek olan karakteri belirler. Eğer TextBox'da şifre yerine " * " görünmesi isteniyorsa; PasswordChar özelliğine " * " yazılır.

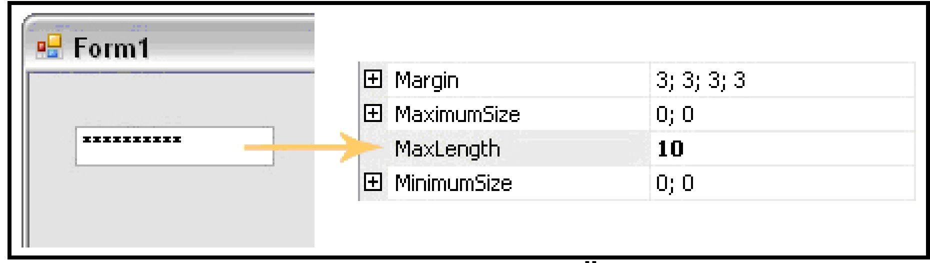


Şekil 213: PasswordChar Özelliği

Kullanıcının TextBox'a girdiği karaktere bakılmaksızın, TextBox içerisinde PasswordChar özelliğinde belirlenen karakter herbir karakterin yerine görüntülenir.

Şifre girilmesi istenilen alanlarda bu özellik kullanıldığı zaman kullanıcı şifre girerken bir başkasının şifreyi görmesine engel olunur. Fakat bu sadece yazım anında bir başkasının görmemesi için bir güvenlik oluşturur. Şifre olarak girilen yazı hiçbir şekilde şifrelenmediği için diğer bütün önemli veriler gibi, bu verileri için de önlem alınmalıdır.

PasswordChar özelliği kullanılırken alternatif olarak kullanıcının girdiği şifrenin karakteri de sınırlanmak istenirse, MaxLength özelliğinden faydalanılabilir. Bu özellik TextBox'ın içerisine maksimum kaç karakter girilebileceğini belirler.



Şekil 214: MaxLength Özelliği

Eğer MaxLength özelliğine 10 değeri atanırsa, kullanıcı TextBox kontrolüne 10 karakterden fazlasını yazamayacaktır. Sistem, belirlenen MaxLength sınırının aşılmasına izin vermeyecektir. Bunun için şifre gibi alanlarda, bir sınır belirtmiş olmamak için kullanılması tercih edilmez. Fakat telefon numarası ve kimlik numarası gibi MaxLength'i belli olan alanlarda kullanılabilir.

Çalışma zamanında PasswordChar ve MaxLength özelliklerini değiştirmek için aşağıdaki kodlar kullanılabilir.

```
textBox1.PasswordChar = '*';
textBox1.MaxLength = 10;
```

Read-Only Özelliği

Bazen TextBox kontrolünün değeri üzerinde, son kullanıcıların herhangi bir değişiklik yapmasına izin vermek istenmeyebilir. Bu durumda kontrolün ReadOnly özelliğini true yapmak yeterli olacaktır. Bunu sağlamak için, Property penceresinden kontrolün ReadOnly özelliği true yapılmalıdır.

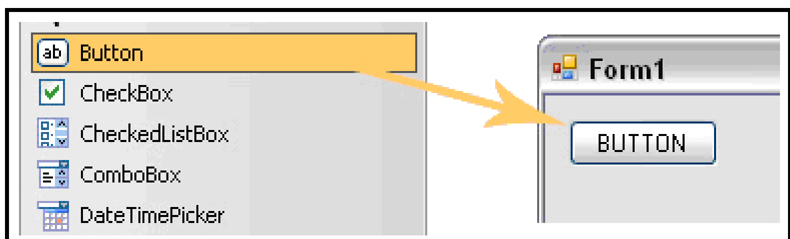
Şekil 215: ReadOnly Özelliği

```
textBox1.ReadOnly = true;
```

TextBox kontrolünün Read-Only özelliği true olduğu durumlarda, uygulama geliştirici TextBox kontrolüne değer atayabilir, fakat kullanıcı dışarıdan bir değer giremez. Kullanıcı dışarıdan TextBox kontrolünü seçebilir, çalışma zamanında verdiğimiz metin (text) değerini kopyalayabilir fakat kesemez. TextBox kontrolünün Read-Only özelliği çalışma zamanında da **true** yapılabilir.

Button

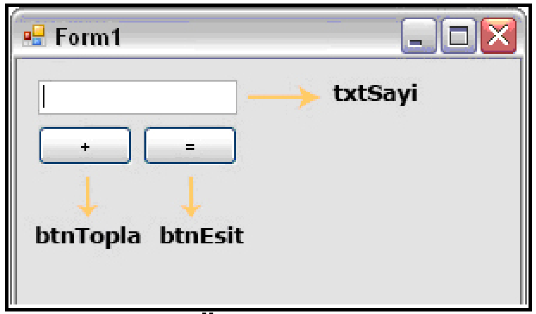
Windows formuna bir Button kontrolü eklemek için, sol tarafta yer alan Toolbox içerisindeki Common Controls sekmesinden bir Button sürükleyip bırakılır ya da kontrolün üzerine çift tıklanır.



Şekil 216: Button tipinden kontrol oluşturmak

Button kontrolü kullanıcının üzerine tıklayarak işlem yapmasını sağlar. Button kontrolü tıklandığında itilmiş ve geri bırakılmış olarak görünür. Kontrole tıklayarak yapılacak her işlem için, kontrolün **Click** olayına (event), bir olay metodu bağlanır. Kontrole her tıklamada bu metod çalışır.

Aşağıdaki örnekte Button kontrolünün Click olayı (event) için yazılmış, örnek bir olay metodu yer almaktadır. Örnekte bir TextBox kontrolü ve iki tane Button kontrolü kullanılıyor.

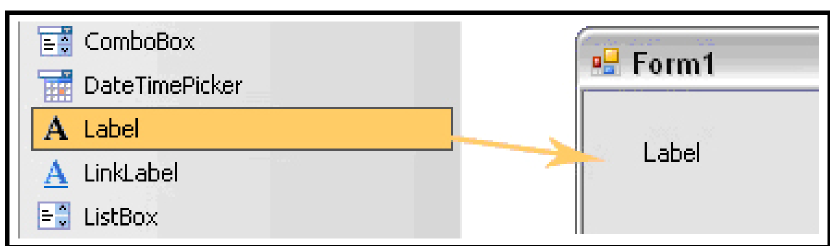


Şekil 217: Örnek form tasarımı

```
public partial class Form1 : Form
{
    int sayi = 0;
    public Form1()
    {
        InitializeComponent();
    }

    private void btnTopla_Click(object sender, EventArgs e)
    {
        sayi = Convert.ToInt32(txtSayi.Text); // TextBox kontrolünün
içerisinde bulunan Text değeri, int tipinden sayi değişkenine atanır.
        txtSayi.Clear(); // Kullanıcı birinci sayıyı girdikten sonra
ikinci sayıyı girmesi için, TextBox kontrolünün Clear() metodu
kullanılarak içinde bulunan değer temizlenir.
    }

    private void btnEsit_Click(object sender, EventArgs e)
    {
         sayi += Convert.ToInt32(txtSayi.Text); // Kullanıcının girdiği
ilk değer sayi değişkenine atanmıştı, şimdi ise son girilen değer ilk
değere eklenir.
        txtSayi.Text = sayi.ToString(); //Toplama işleminin sonucu,
TextBox kontrolüne atanır.
    }
}
```

Label

Windows formuna bir Label kontrolü eklemek için, sol tarafta yer alan Toolbox içerisindeki Common Controls sekmesinden bir Label sürükleyip bırakılır ya da kontrolün üzerine çift tıklanır.

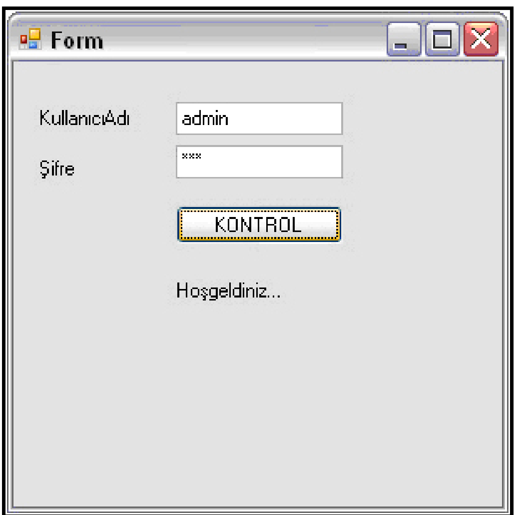


Şekil 217: Label tipinden kontrol oluşturmak

Label kontrolleri genellikle açıklayıcı metin alanlar ve başka kontrollere anahtar oluşturmak için kullanılır. Örneğin, bir Label kontrolü başka bir kontrol ile ilgili açıklayıcı bilgi vermesi için kullanılabilir. Aynı zamanda kullanıcıya yaptığı işlemin sonucunda bilgi vermek için de kullanılabilir.

Aşağıdaki örnekte kullanıcının TextBox kontrollerine girdiği Kullanıcı Adı ve Şifre değerlerinin kontrolü yapılmaktadır. txtSifre textBox'ının PassWordChar özelliğine "*"

karakteri atanır, böylece kullanıcı şifresini girerken başkasının şifreyi görmesi engellenmiş olur.



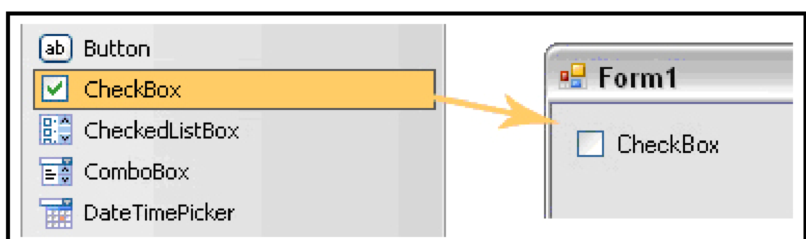
Şekil 218: Uygulamanın çalışma zamanı hali

lblKullaniciAdi ve lblSifre Label kontrolleri, TextBox kontrolleri ile ilgili açıklayıcı bilgi vermesi için kullanılır. lblSonuc Label'ı ise, kullanıcının doğru kullanıcı olup olmadığı sonucunu göstermek için kullanılır.

```
private void btnKontrol_Click(object sender, EventArgs e)
{
    if ((txtKullaniciAdi.Text == "admin") && (txtSifre.Text == "123")) // TextBox kontrollerine girilen değerlerin doğru olup olmadığı kontrol edilir.
    {
        lblSonuc.Text = "Hoşgeldiniz..."; // Doğru Kullanıcı olduğu için Label kontrolüne " Hoşgeldiniz... " yazılır.
     }
    else
     {
        lblSonuc.Text = "Hatalı Kullanıcı..."; // Doğru Kullanıcı olmadığı için Label kontrolüne " Hatalı Kullanıcı... " yazılır.
     }
}
```

CheckBox

Windows formuna bir CheckBox kontrolü eklemek için, sol tarafta yer alan Toolbox içerisindeki Common Controls sekmesinden bir CheckBox sürükleyip bırakılır ya da kontrolün üzerine çift tıklanır.



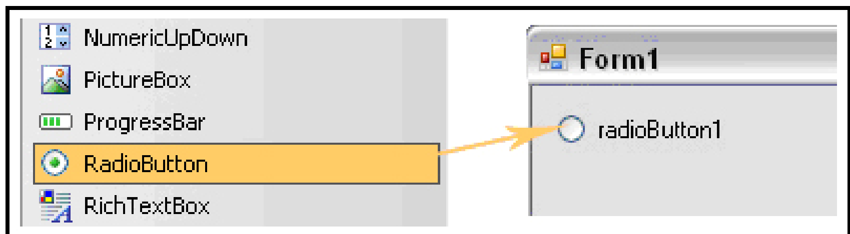
Şekil 219: CheckBox kontrolü oluşturmak

Kullanıcıya birden fazla seçenek sunup bir ya da birden fazla seçim yapabilmesini sağlamak için CheckBox kontrolü kullanılır. Genellikle "EVET" , "HAYIR" gibi seçeneklerde kullanılır.

RadioButton kontrolü ile benzerlikleri vardır.

RadioButton

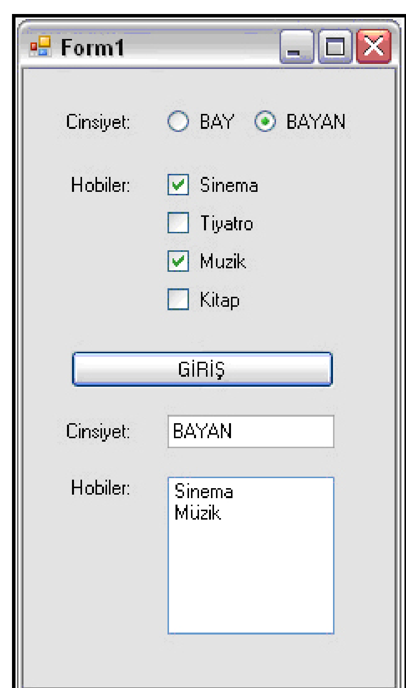
Windows formuna bir RadioButton kontrolü eklemek için, sol tarafta yer alan Toolbox içerisindeki Common Controls'den bir RadioButton sürükleyip bırakılır ya da kontrolün üzerine çift tıklanır.



Şekil 220: RadioButton kontrolü oluşturmak

Kullanıcıya birden fazla seçenek sunup bir seçim yapmasını sağlamak için RadioButton kontrolü kullanılır. Genellikle cinsiyet seçimi gibi işlemler yaptırmak için kullanılabilir.

Aşağıdaki örnekde Checkbox ve RadioButton arasındaki fark görülebilir:



Şekil 221: Uygulamanın çalışma zamanı hali

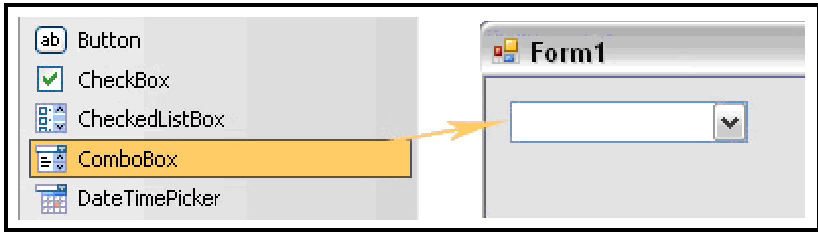
Örnek formda, son kullanıcının cinsiyet ve hobi seçimi yapmasına imkan verilmektedir. Cinsiyet seçimi için RadioButton, hobi seçimi için ise CheckBox kontrolleri kullanılmıştır. RadioButton kontrolünde sadece tek seçim yapılabilirken, CheckBox kontrolünde birden fazla seçim yapılabillir. Bu sebeple çoklu şeçim yapılmasına izin veren Checkbox kontrolü, hobi seçiminde kullanılmıştır. Tekli seçim yapılmasına izin veren RadioButton kontrolü ise, cinsiyet seçimi yapabilmek amacıyla kullanılmıştır.

```
private void button1_Click(object sender, EventArgs e)
{
    string hata = ""; //Kullanıcı Buton'a bastığı zaman eğer herhangi bir
seçim yapmazsa kullanıcıya hata vermek için bu değişken oluşturulur.

    txtCinsiyet.Text = ""; //Kullanıcı cinsiyet seçimi yapıp Giriş
```

```
butonuna bastıktan sonra seçilen cinsiyeti yazdırmak için kullanılır.

    if (radioBay.Checked == true) // Eğer kullanıcı radioBay kontrolünü
işaretlerse BAY, radioBayan kontrolünü işaretlerse BAYAN yazar.
        txtCinsiyet.Text = "BAY";
    else if (radioBayan.Checked == true)
        txtCinsiyet.Text = "BAYAN";
    else
        hata+="Cinsiyet Seçmelisiniz...";

    listBox1.Items.Clear(); //Butona her basıldığında kontrollerden
seçilen değerlerin tekrar ListBox kontrolüne alt alta eklenmemesi için her
seferinde ListBox kontrolünün öğeleri temizlenir.
    if (chkKitap.Checked == false && chkMuzik.Checked == false &&
chkTiyatro.Checked == false && chkSinema.Checked == false)
    {
        hata += "\nHobi Seçiniz..."; // Eğer kullanıcı herhangi bir hobi
seçimi yapmadıysa kullanıcıya uyarı çıkarılır.
        MessageBox.Show(hata); //Kullanıcıya hata mesajı gösterilir.
    }
    else //Kullanıcının seçim yaptığı hobiler listbox kontrolüne eklenir.
    {
        if (chkSinema.Checked == true)
            listBox1.Items.Add("Sinema");
        if (chkTiyatro.Checked == true)
            listBox1.Items.Add("Tiyatro");
        if (chkMuzik.Checked == true)
            listBox1.Items.Add("Müzik");

        if (chkKitap.Checked == true)
            listBox1.Items.Add("Kitap");
    }
}
```

ComboBox

Windows formuna bir ComboBox kontrolü eklemek için, sol tarafta yer alan Toolbox içerisindeki Common Controls sekmesinden bir ComboBox sürükleyip bırakılır ya da kontrolün üzerine çift tıklanır.



Şekil 222: ComboBox tipinden kontrol oluşturmak

ComboBox kontrolü, bir kontrolün içerisinde data görüntülemek, bu yolla kullanıcıya bilgi girişi sırasında kolaylık sağlamak ve girilecek değerlerin bir standart haline gelmesi için kullanılmaktadır. Kontrol aşağıya doğru açılır ve iki bölümden oluşur. Üst tarafı bir TextBox kontrolü gibi düşünülebilir ve son kullanıcının metin girebilmesini sağlar. İkinci bölüm ise bir ListBox kontrolü gibi düşünülebilir. Kullanıcının seçebileceği öğelerin (item) listesini oluşturur

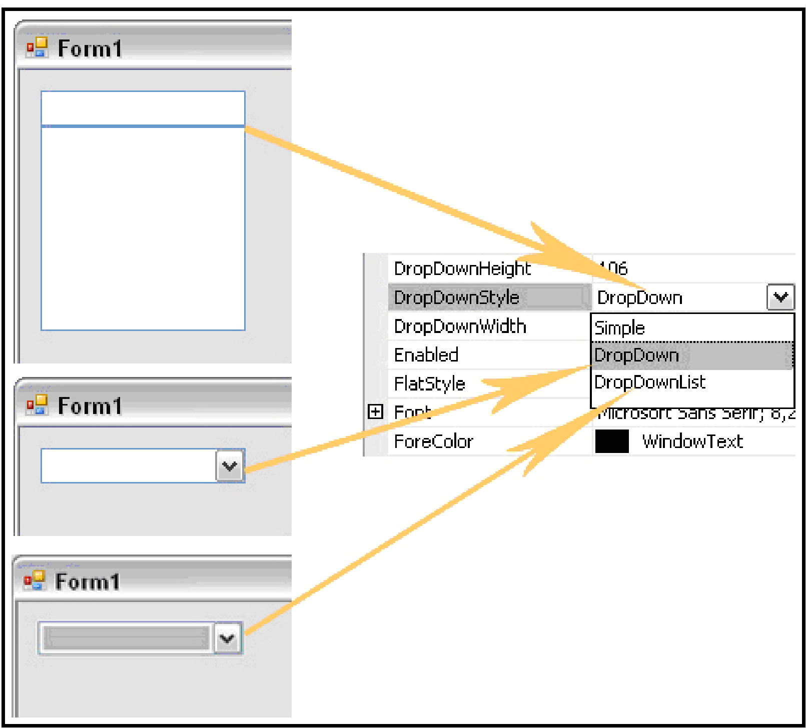
ComboBox, ListBox kontrolü ile benzer davranış gösteren bir kontroldür. Fakat bazı durumlarda ComboBox kontrolünü kullanmak daha avantajlıdır. Örneğin kontrolde yer alan listenin içerisine kullanıcı tarafından ekleme yapılması isteniyorsa ComboBox kontrolünü kullanmak daha uygundur. ComboBox metin seçenekler içerdiğinden istenilen öğeler kullanıcı tarafından eklenebilir.

Aşağıda yer alan örnekte ComboBox ile ListBox arasındaki bu fark gösterilmektedir. Bunun için windows formuna bir tane Button kontrolü eklenir. Formun yapıcı metodu (Constructor) içerisinde, ComboBox kontrolüne, çalışma zamanında, yeni bir öğenin nasıl eklenebileceği de gösterilmektedir. Daha sonra btnEkle kontrolünün Click olay metodunda ise, ComboBox kontrolüne bir kullanıcının dışarıdan nasıl öğe ekleyebildiği örneklenmiştir.

```
public Form1()
{
    InitializeComponent();
    comboBox.Items.Add("windows"); //comboBox1'e bir tane değer (item)
ekleme işlemi.
}

private void btnEkle_Click(object sender, EventArgs e)
{
    comboBox.Items.Add(comboBox.Text.ToString()); //Kullanıcının ComboBox
kontrolüne ekleyeceği öğeyi kontrolün TextBox bölümüne girdikten sonra
Ekle butonuna tıklayarak ComboBox'ın öğeleri arasına ekledik.
}
```

DropDownStyle Özelliği

Bir ComboBox kotrolünün DropdownStyle özelliği, Property penceresinde yeralan DropDownStyle özelliğinden değiştirilir. DropDownStyle özelliği, bir ComboBox'ın görünümünü değiştirmek için kullanılır.



Şekil 223: DropDownStyle Özelliği

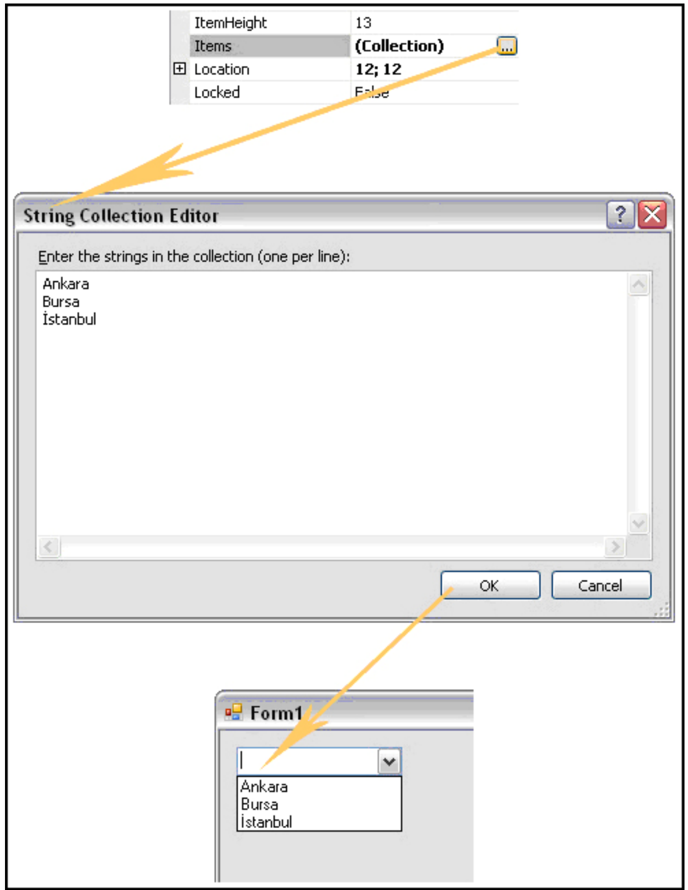
Bir ComboBox için DropDownStyle özelliğinin varsayılan değeri DropDown'dır. Özelliğin değeri DropDown olduğu durumda, ComboBox kontrolünün TextBox ve ListBox özelliklerinden yararlanılabilir ve kullanıcı dışarıdan kontrol içerisine öğe ekleyebilir.

DropDownStyle özelliği DropDownList yapıldığı zaman, ComboBox kontrolünün sadece ListBox özelliğinden faydalanılır ve daha önceden eklenmiş öğelere erişilebilir. Kullanıcı istediği veriyi çalışma zamanında TextBox özelliği aracılığıyla ekleyemez.

DropDownstyle özelliği Simple yapıldığı zaman ise; yine TextBox ve ListBox özelliklerinden faydalanılabilir. Fakat form üzerinde daha fazla yer kaplar.

Items Özelliği

Bir ComboBox'ın öğeleri, çalışma zamanında eklenebileceği gibi tasarım zamanında Property penceresinde yer alan Items koleksiyonundan da eklenebilir.



Şekil 224: Items Özelliği

ComboBox kontrolüne, yeni öğelerini eklemek için, uygulama geliştiricinin karşısına çıkan String Collection Editor penceresinde, istenilen öğelerin yazılması yeterlidir.

Çalışma zamanında ComboBox kontrolüne öğeler aşağıdaki gibi eklenebilir.

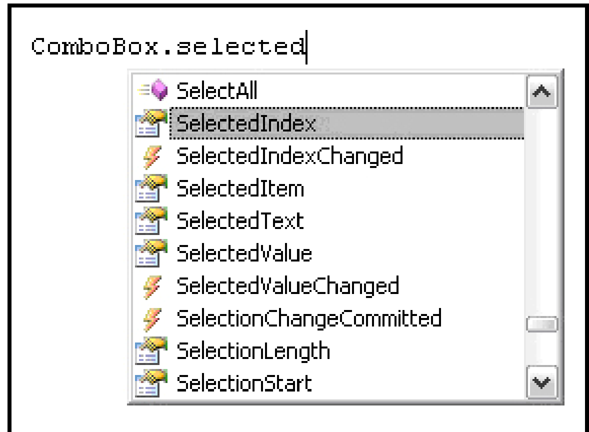
```
ComboBox.Items.Add("İstanbul");
ComboBox.Items.Add("Bursa");
ComboBox.Items.Add("Ankara");
```

Şekil 225: Çalışma zamanında ComboBox kontrolüne öğe eklemek

SelectedIndex ve SelectedItem Özellikleri

SelectedIndex özelliği, bir ComboBox kontrolünün içerisinde seçilen elemanın, ComboBox içerisindeki kaçıncı öğe olduğunun bilinmesi istendiği durumlarda kullanılan bir özelliktir. Hiçbir öğe seçilmemiş ise; SelectedIndex değeri -1 dir. Eğer seçilmiş ise; comboBox içerisindeki bulunduğu indeks değeri sonuç olarak gelecektir. Ancak unutulmamalıdır ki, ComboBox ve ListBox kontrollerinin elemanları indekslenirken; 0'dan başlanılmaktadır.

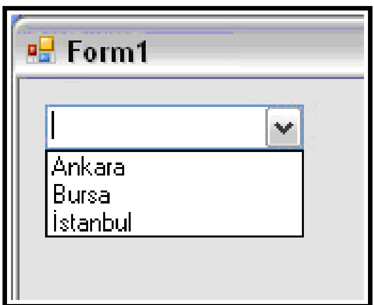
SelectedItem özelliği ise SelectedIndex özelliğine benzer. Farkları ise; SelectedItem özelliğinin ComboBox içerisinde bulunan öğelerin string değerlerini vermesidir.



Şekil 226: SelectedIndex özelliği

Bir ComboBox kontrolünün SelectedIndex ve SelectedItem özellikleri kontrolün adının yanına konulan nokta operatörü ile çağırılır.

ComboBox üzerindeki seçili eleman çalışma zamanında değiştirilebilir. Bir ComboBox kontrolüne property penceresinden üç tane öğe eklendiği durumda ilgili combobox kontrolünün görünümü aşağıdaki gibi olacaktır.



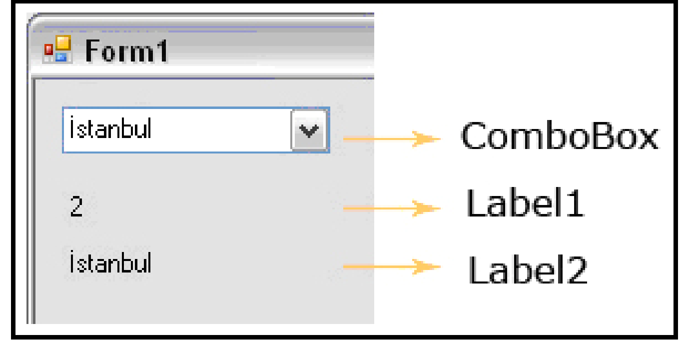
Şekil 227: Dinamik yüklenen ComboBox kontrolünün çalışma zamanındaki hali

Öğeler eklendikten sonra Form1() yapıcı metodunun (constructor) içerisinde ComboBox kontrolünün SelectedIndex özelliği değiştirilsin. Daha sonra ise Formun üzerine daha önceden bırakılan Label1 ve Label2 adındaki label konrollerine ComboBox'ın SelectedIndex ve SelectedItem özelliklerinin değerleri yazdırılsın.

```
public Form1()
{
    InitializeComponent();

    ComboBox.SelectedIndex = 2;
    Label1.Text = ComboBox.SelectedIndex.ToString();
    Label2.Text = ComboBox.SelectedItem.ToString();
}
```

Şekil 228: SelectedIndex ve SelectedItem özellikleri için örnek kullanım

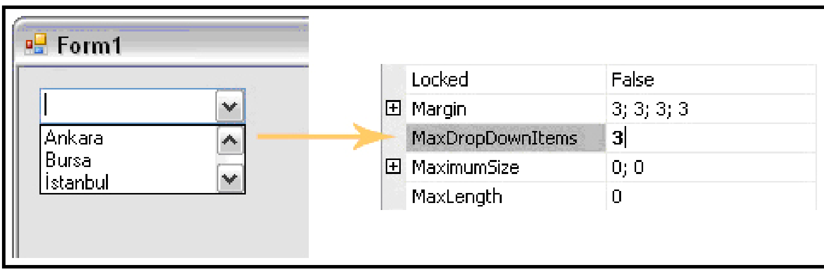
Bu işlemler yapıldığı zaman, ComboBox kontrolünün SelectedIndex ve SelectedItem özelliklerinin değerleri Label kontrollerine atandığında, SelectedIndex özelliği ***2***, SelectedItem özelliği ise ***İstanbul*** olacaktır. Eğer çalışma zamanında kod ile seçili öğe değiştirilmeseydi, SelectedIndex özelliği -1, SelectedItem özelliği ise boş olurdu.



Şekil 229: Uygulamanın çalışma zamanındaki hali

MaxDropDownItems Özelliği

Bir ComboBox kontrolünün MaxDropDownItems özelliğine Property penceresinden değer verilebilir.

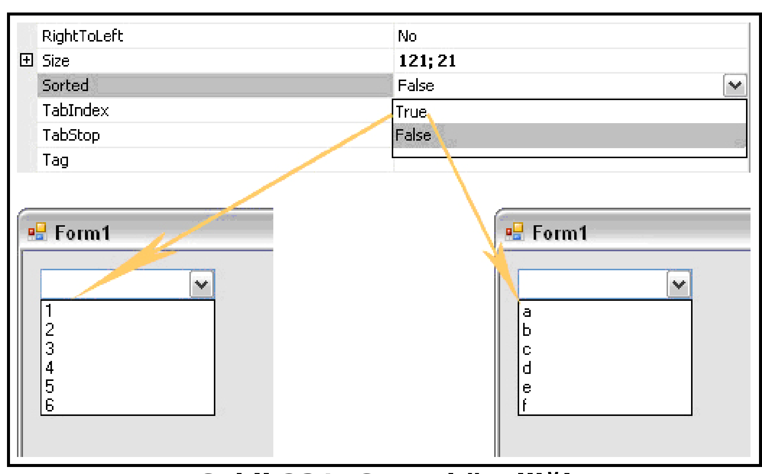


Şekil 230: MaxDropDownItems özelliği

Eğer bu özelliğe üç değeri verilirse; ComboBox kontrolü üçüncü öğeden sonrasını göstermez ve ComboBox kontrolünün ListBox bölümünde bir kaydırma çubuğu (ScroolBars) çıkar. Kaydırma çubuğu aşağıya çekildikçe diğer öğeler de görülebilir. MaxDropDownItems özelliğine herhangi bir değer verilmezse, varsayılan olarak sekiz öğeden sonra kaydırma çubuğu oluşur. Eğer ComboBox kontrolü içerisinde çok fazla öğe varsa, bu özelliği kullanarak kontrolün fazla yer kaplaması engellenmiş olur.

Sorted Özelliği

Bir ComboBox kontrolünün Sorted özelliğini değiştirmek için, Property penceresinde yer alan Sorted özelliği kullanılır. Sorted özelliği, bir ComboBox'ın içerisine girilen öğeleri sıralamak için kullanılır. Kontrolün içerisinde yer alan öğeler nümerik ise; küçükten büyüğe, alfabetik ise a'dan z'ye sıralanır.

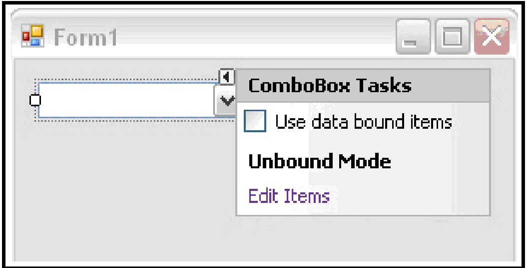


Şekil 231: Sorted özelliği

ComboBox'ın içerisinde yer alan öğelerin sıralı bir şekilde yer alması isteniyorsa; **Sorted** özelliği **true** yapılmalıdır.

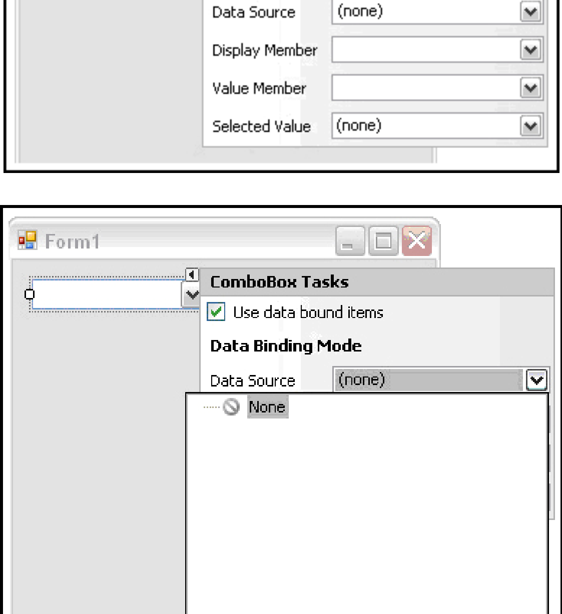
ComboBox Kontrolüne Veritabanından Veri Eklemek

Bir ComboBox kontrolüne veritabanından veri eklemek için ComboBox Tasks ekranı kullanılır.



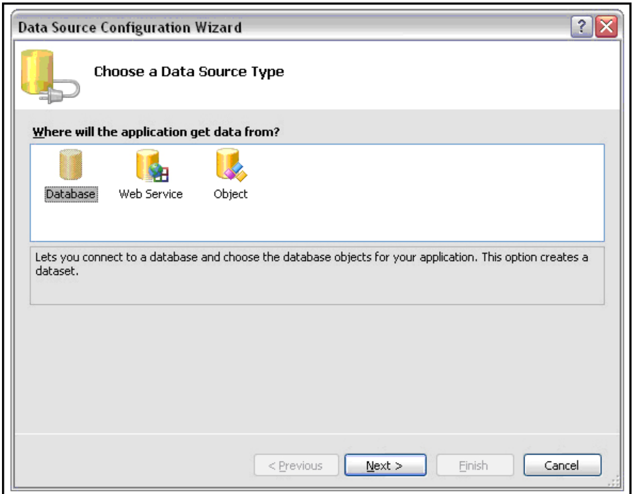
Şekil 232: ComboBox kontrolüne veri tabanından veri eklemek

Use data bound items'ı tıkladıktan sonra çıkan pencereden **DataSource**, **DisplayMember**, **ValueMember** ve **SelectedValue** özellikleri belirlenmelidir. DisplayMember özelliği Combobox kontrolünün **göstereceği alanın** adı olarak atanmalıdır. ValueMember özeliği ise; **arka planda** ComboBox'da seçilen öğeye atanmasını istenilen alanın adıdır. SelectedValue ise; seçili olan öğenin Value Member'idir.



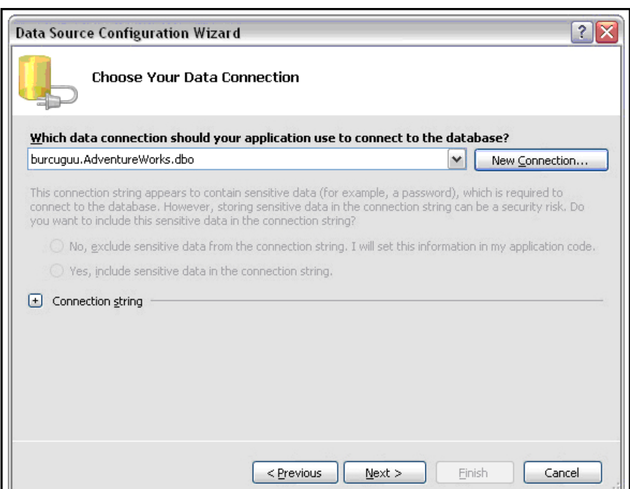
Şekil 233: ComboBox kontrolüne datasource eklemek

DataSource'u belirlemek için; **Add Project Data Source** dedikten sonra çıkan Data Source Configuration Wizard ekranından **Database** seçilip Next butonuna basılır.



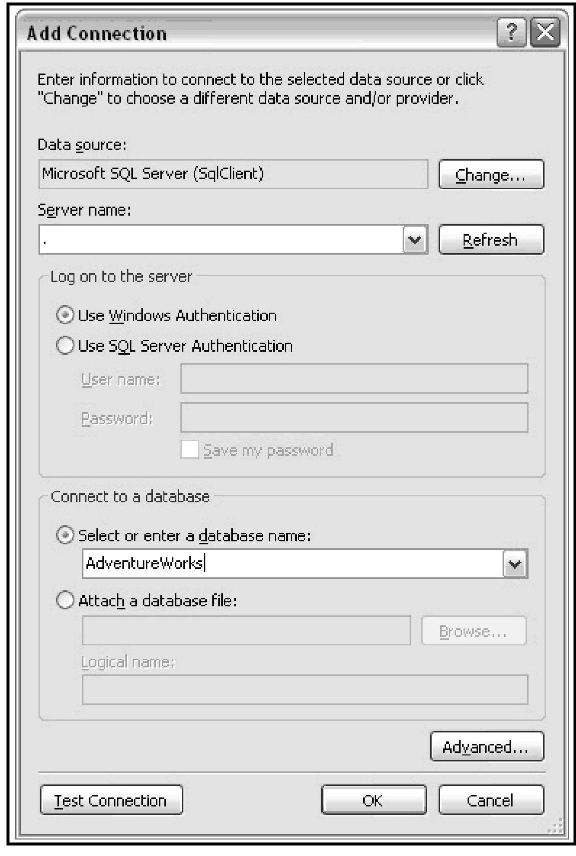
Şekil 234: Datasource bağlantısı

Daha sonra veri kaynağına bağlanmak için kullanılacak bağlantı tipini belirlemek için New Connection denilebilir ya da önceden oluşturulmuş bir connection varsa o bağlantı kullanılabilir.



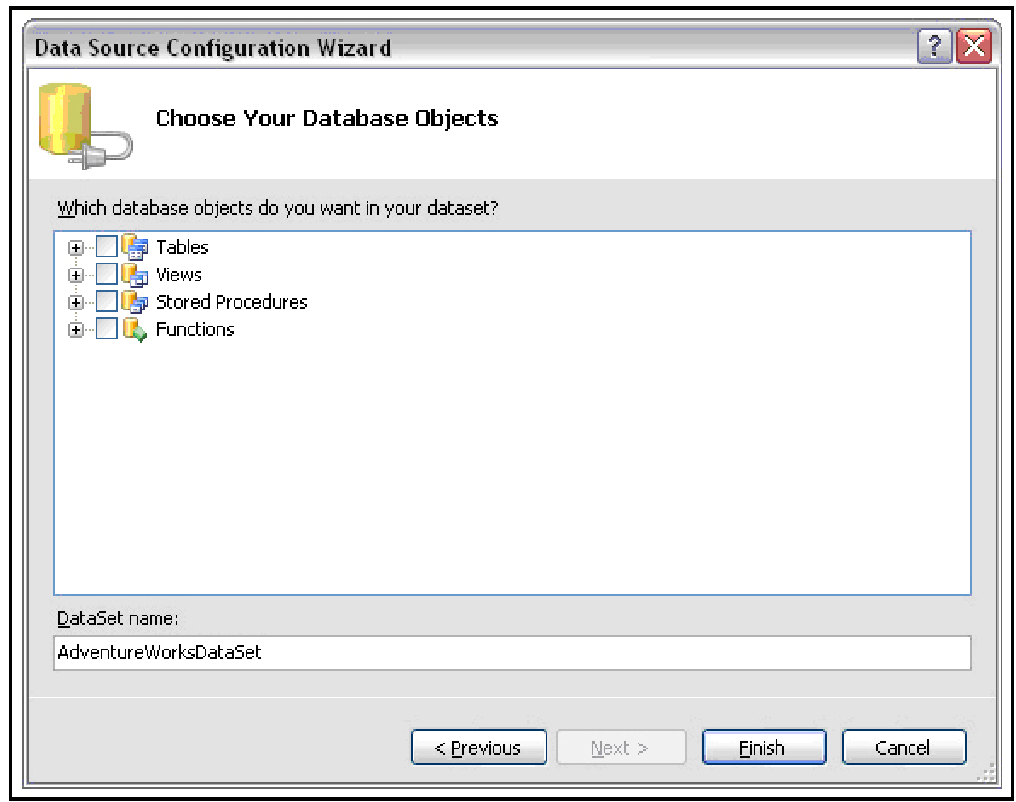
Şekil 235: ConnectionString özelliğinin belirlenmesi

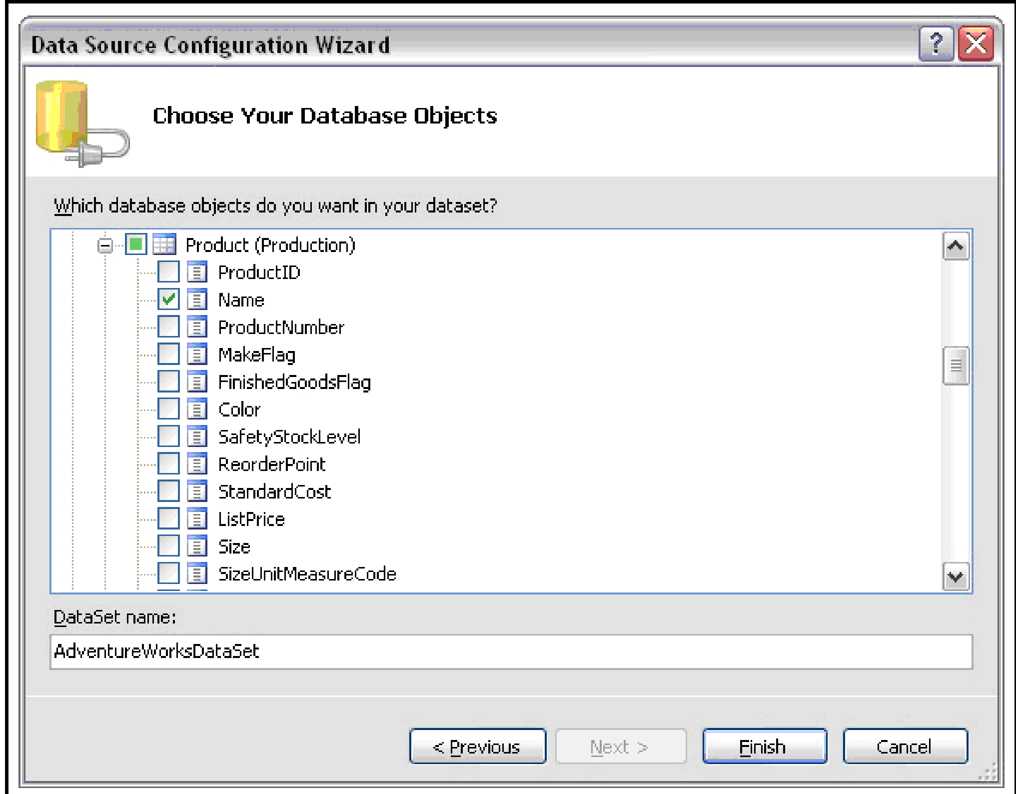
"New Connection" seçeneği seçildikten sonra "Add Connection" ekranından, Data source özelliği, Server name özelliği ve hangi veritabanına bağlanılacağı belirlendikten sonra OK butonuna basılarak yeni bir bağlantı kurulmuş olur. Artık ComboBox kontrolü ile yapılacak işlemlerde bu bağlantı kullanılabilir.



Şekil 236: ConnectionString bilgisinin konfigürasyon dosyasında saklanıp saklanmayacağına karar verilmesi adımı

OK butonuna basıldıktan sonra tekrar Data Source Configuration Wizard ekranı ile karşılaşılır. Daha sonra Next butonuna basılarak veritabanında bulunan hangi tablo ile ve bu tablodaki hangi kolon ile işlem yapılmak istendiği seçilir.

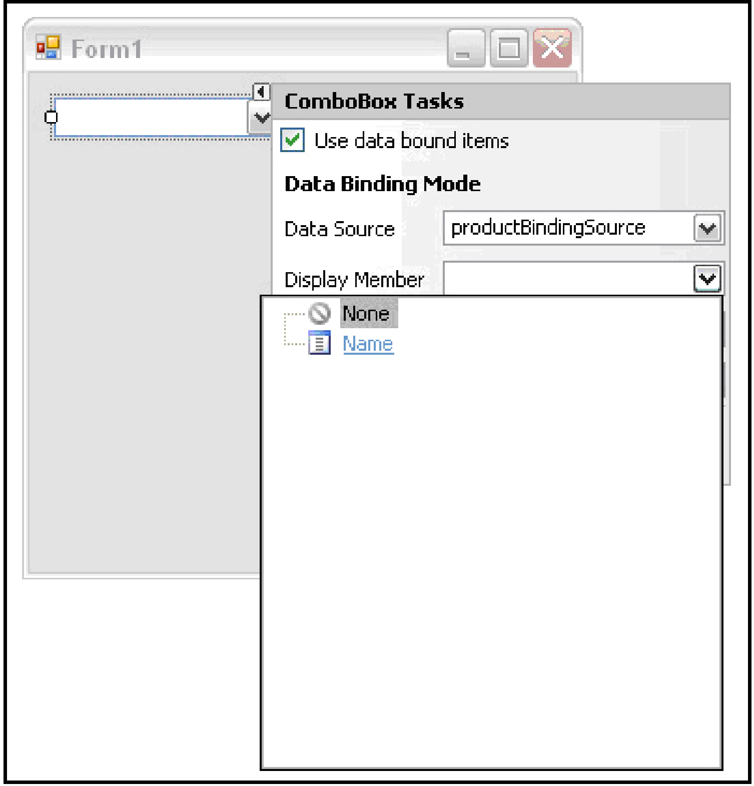




Şekil 237: ComboBox'da görüntülenecek Tabloya ait kolonların belirlenmesi

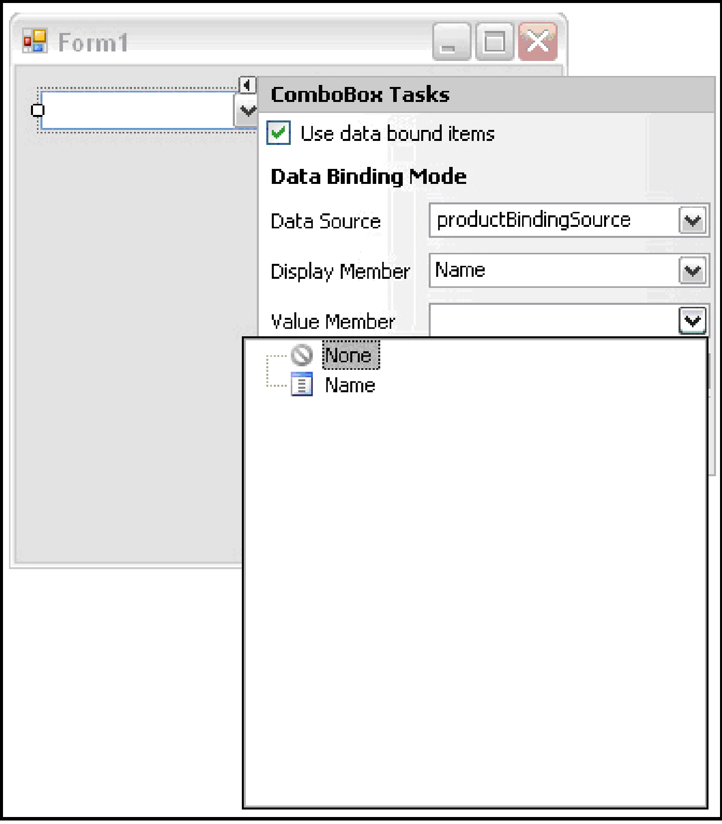
Bunlar belirlendikten sonra Finish butonuna basılır. Böylece ComboBox kontrolü için hangi bağlantıyı kullanarak, ilgili veritabanında bulunan ilgili tablodaki hangi kolonlar üzerindeki verilerin kullanacağı belirlenmiş olur.

Şimdi ComboBox kontrolünün Display Member, Value Member, Selected Value özellikleri belirlenebilir.



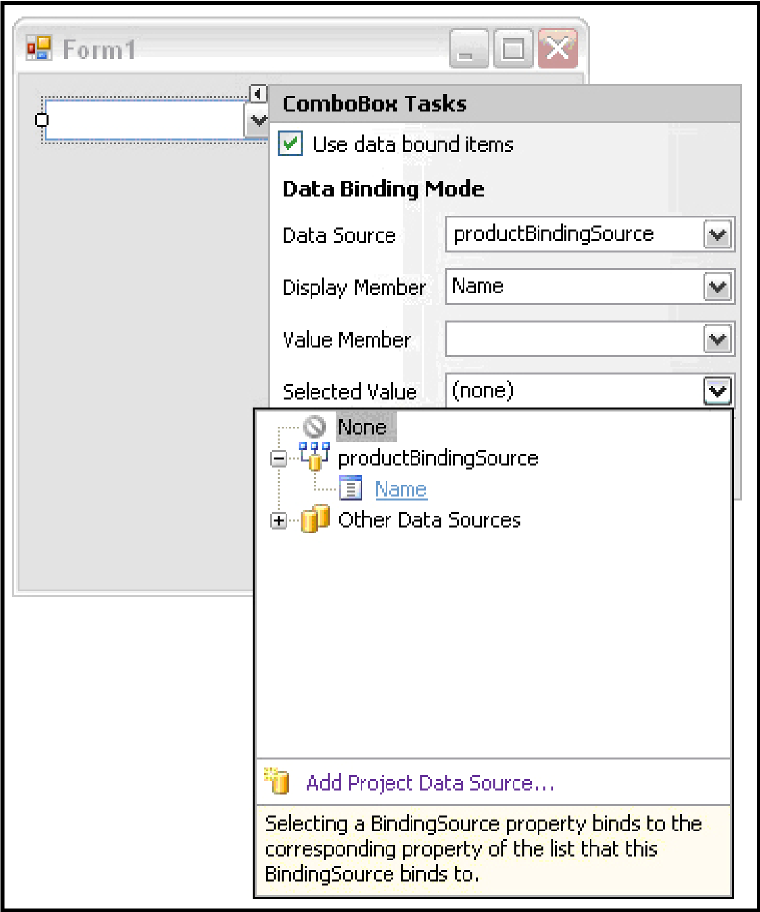
Şekil 238: Display Member özelliği

DataSource'da seçilen tablo içerisindeki Name kolonu ComboBox kontrolünün DisplayMember özelliği olarak belirlenir.



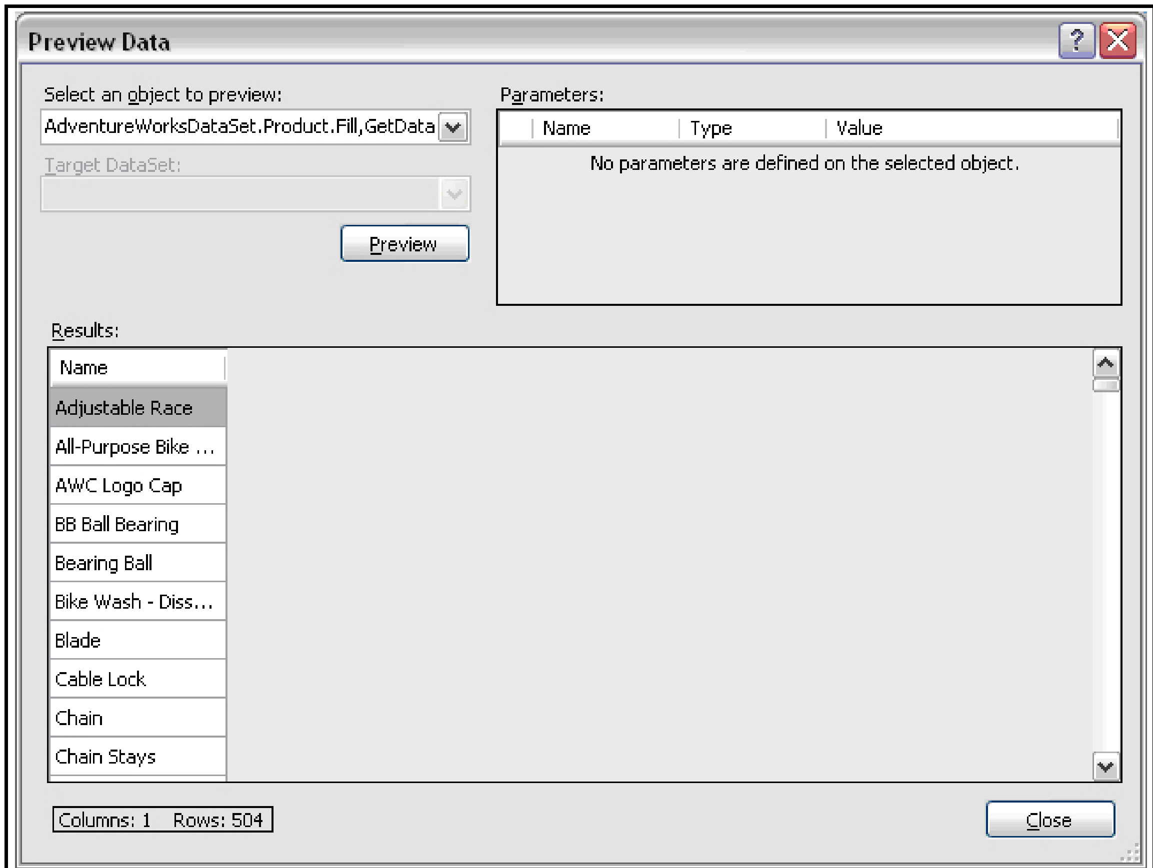
Şekil 239: Value Member özelliği

Kontrolün DisplayMember üyesi Name olarak belirlendikten sonra son olarak SelectedValue değerine productBindingSource içerinde yer alan Name alanı verilir.



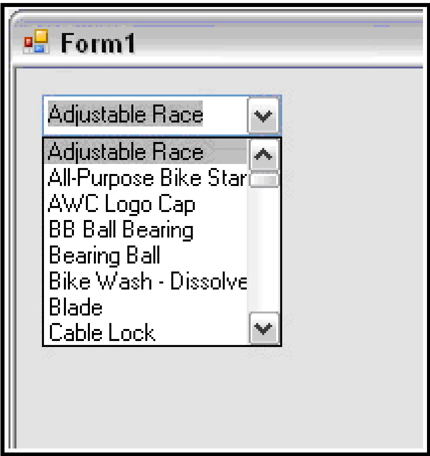
Şekil 240: Selected Value özelliği

ComboBox Tasks ekranında yer alan Preview Data tıklanarak yapılan işlemlerin doğruluğu kontrol edilebilir. Bunun için, çıkan Preview Data ekranında bulunan Preview butonuna basmak yeterlidir.



Şekil 241: Preview Data adımı

Uygulama çalıştırıldığında seçilen veritabanı içerisinde yer alan tablodaki ilgili kolonun verileri, ComboBox kontrolü içerisine dolar.



Şekil 242: Uygulamanın çalışma zamanındaki hali

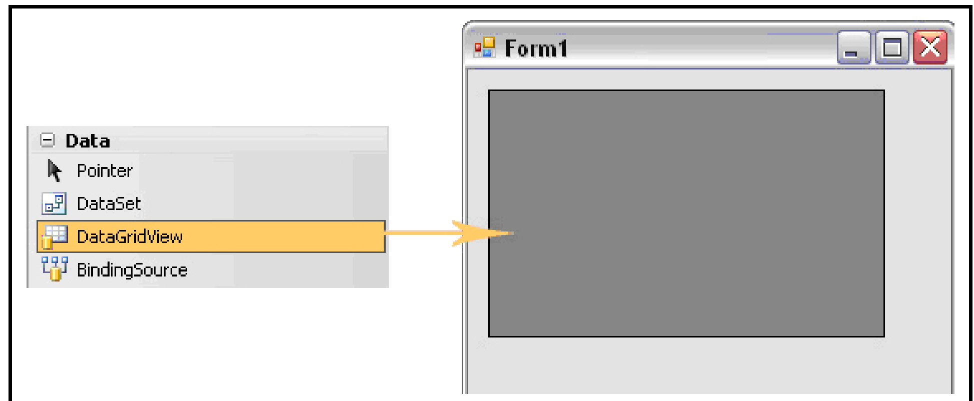
ComboBox kontrolüne çalışma zamanında veri eklemek için aşağıdaki kod kullanılır.

```
public Form1()
{
            InitializeComponent();
            SqlConnection con = new SqlConnection("data source=.;initial
catalog=AdventureWorks;integrated security=true"); //Veritabanına bağlantı
oluşturulur..
            SqlCommand cmd = new SqlCommand("select Name from
Production.Product",con); // Hangi tablodan hangi veri ile işlem
yapılacağı belirlenir.
```

```
            con.Open();
            SqlDataReader dr = cmd.ExecuteReader();
            while (dr.Read())
            {
                ComboBox.Items.Add(dr[0].ToString()); // DataReader dan
gelen veriler ComboBox kontrolüne eklenir.
            }
            dr.Close();
            con.Close();
}
```

DataGridView

Windows formuna bir DataGridView kontrolü eklemek için, sol tarafta yer alan Toolbox içerisindeki Data sekmesinden bir DataGridView sürükleyip bırakılır ya da kontrolün üzerine çift tıklanır.

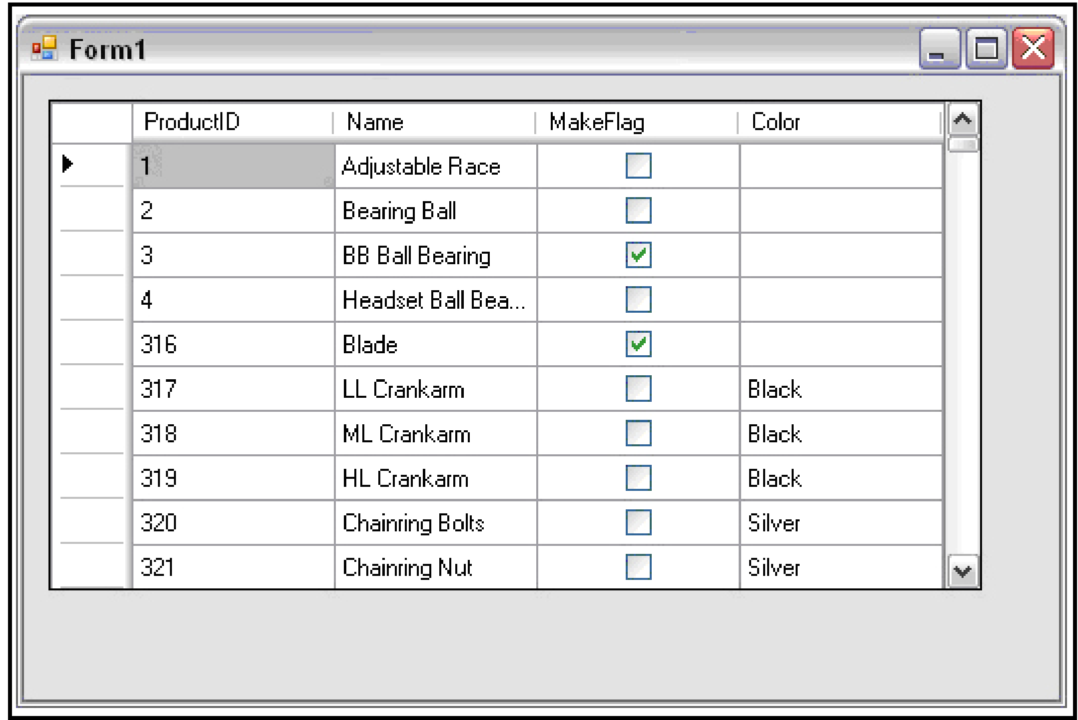


Şekil 243: DataGridView tipinden kontrol oluşturmak

GridView kontrolü databaseden çekilen verilerin veya geçiçi olarak oluşturulan verilerin gösterilmesine yarayan bir windows kontrolüdür. GridView kontrolünde verileri göstermek için izlenmesi gereken bazı adımlar bulunmaktadır.

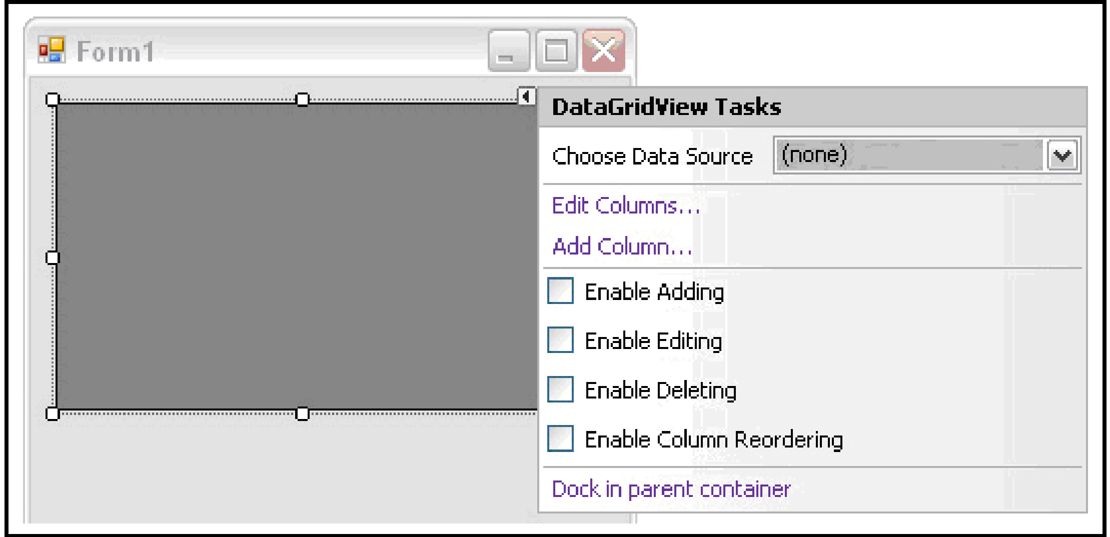
Öncelikle bir veri kaynağı olmalıdır. Bu veri yaknağı **DataTable** veya **DataSet** olabilir. Sonra bu veri kaynakları bir şekilde doldurulmalıdır. Aşağıda yer alan örnekte, bir veri kaynağından verilerin nasıl GridView kontrolüne alınacağı anlatılmaktadır.

```
            SqlConnection con = new SqlConnection("Data Source=.; Initial
Catalog =  AdventureWorks; Integrated Security = SSPI"); // Bağlantı
oluşturulur.
            SqlDataAdapter adaptor = new SqlDataAdapter("Select
ProductID,Name,MakeFlag,Color from Production.Product", con); //Adaptöre
gereken select komutu ve bağlantı yollanır.
            DataTable dt = new DataTable(); //DataTable oluşturulur.

            adaptor.Fill(dt); //Adaptörün Fill metodu sayesinde gereken
kaynak doldurulur.

            dataGridView1.DataSource = dt; //DataTable source olarak
bağlanıyor.
```



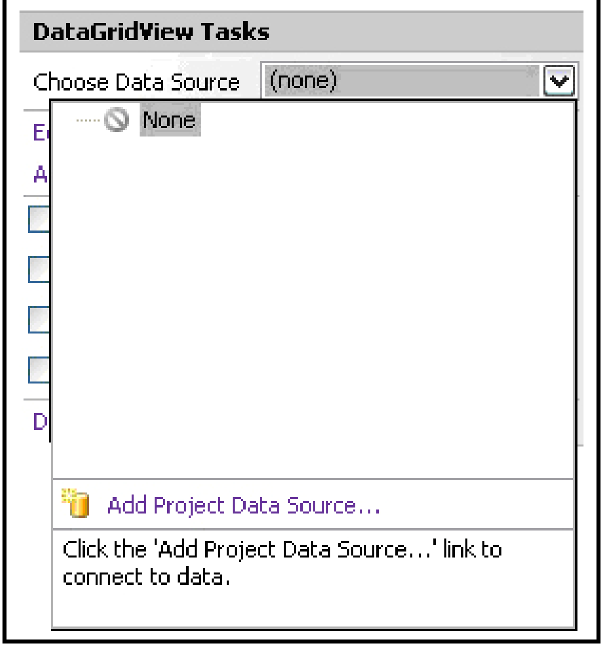
Şekil 244: DataGridView kontrolünün veri yüklenmiş hali

Çalışma zamanında kod yazarak kontrolü veri kaynağından gelen veriler ile doldurmak yerine, DataGridView'in Task ekranı da kullanılabilir. Bunun için DataGridView kontrolünü form üzerinde oluşturduktan sonra, DataGridView Task ekranı seçilmelidir.



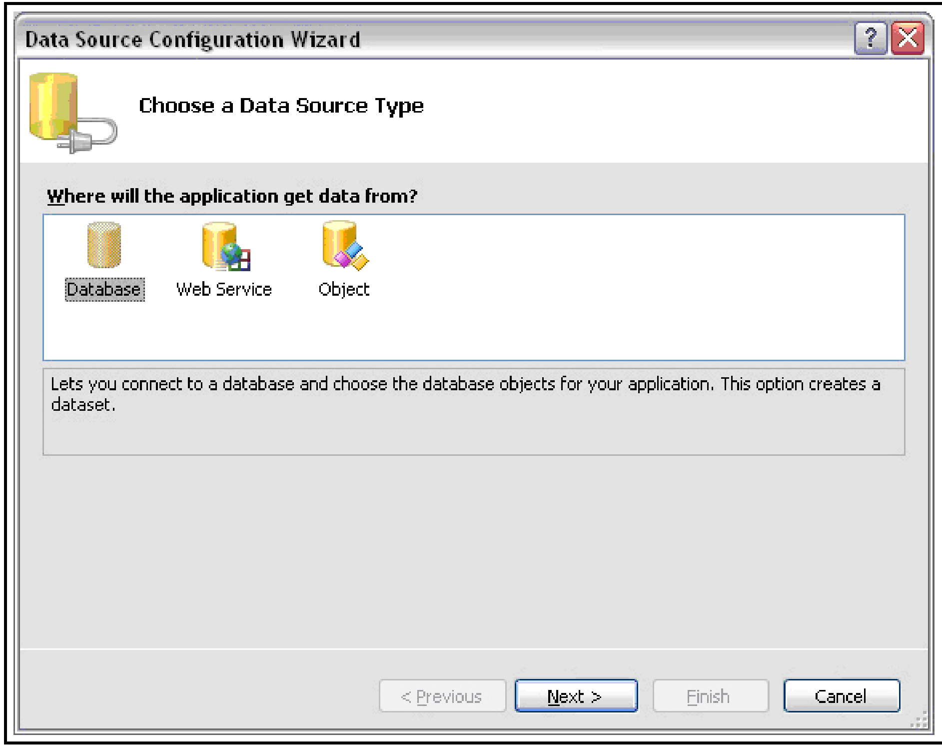
Şekil 245: DataGridView kontrolü için Tasks seçenekleri

Çıkan ekrandan, Choose Data Source bölümünden yeni bir DataSource seçilmelidir.

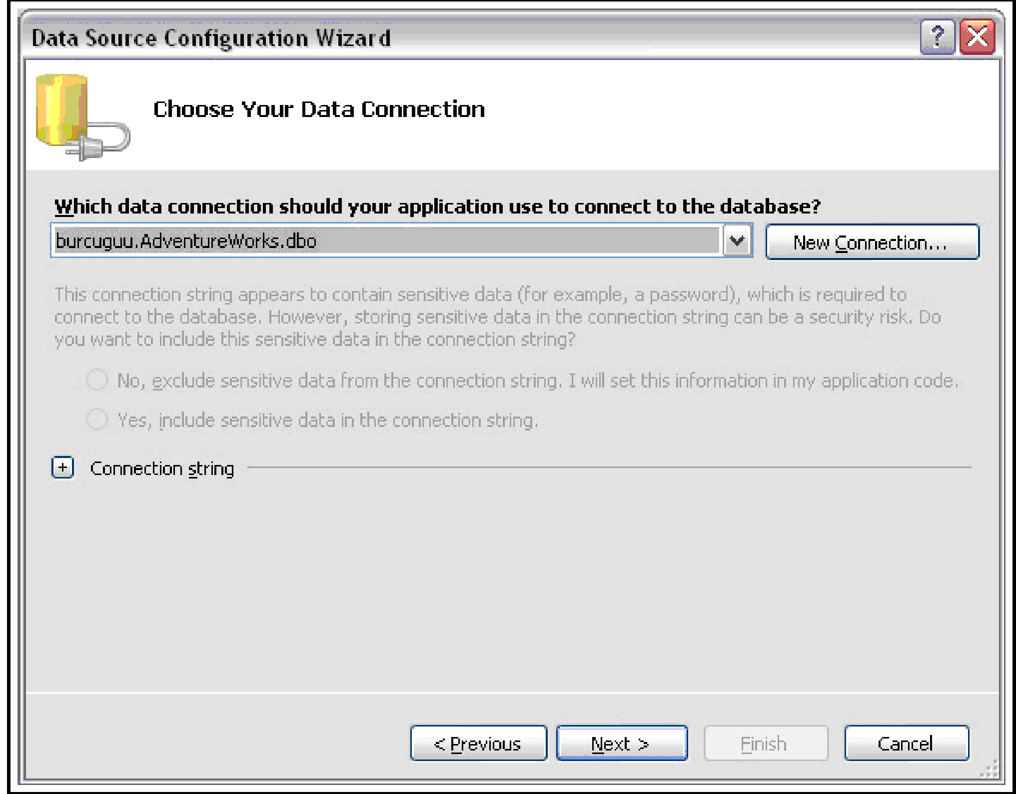


Şekil 246: Data Source seçim adımı

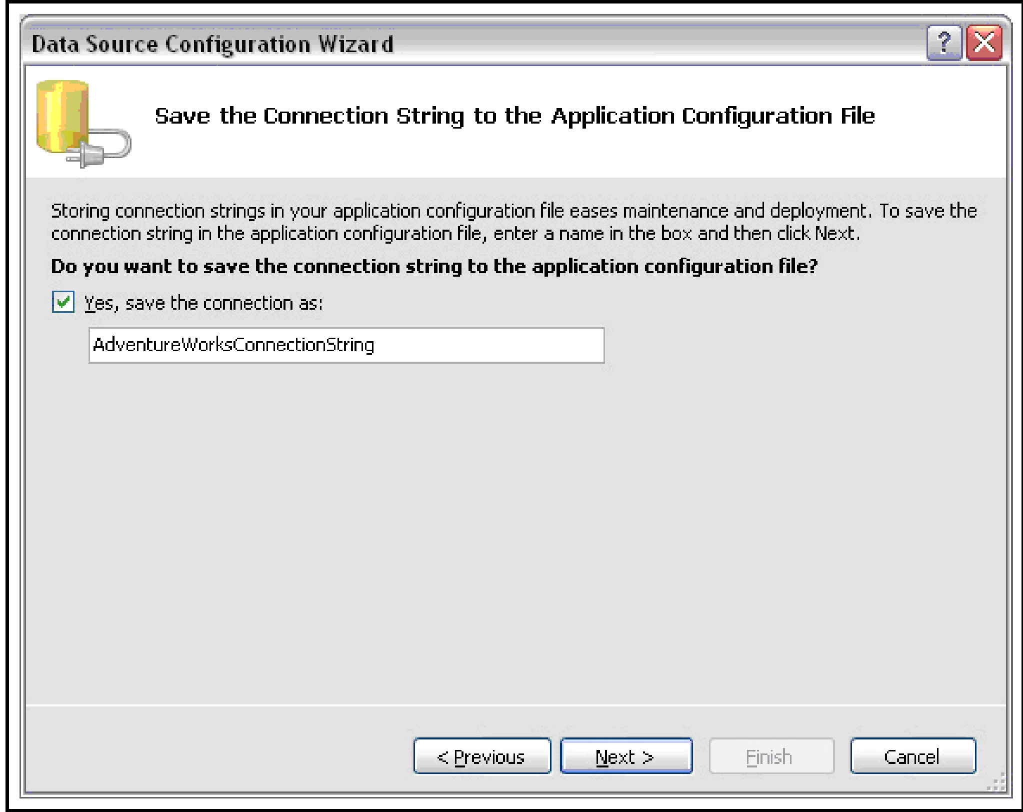
Add Project Data Source denildikten sonra yeni bir Data Source oluşturulmalıdır. Yeni bir Data Source oluşturmak için aşağıda yer alan adımlar takip edilmelidir.



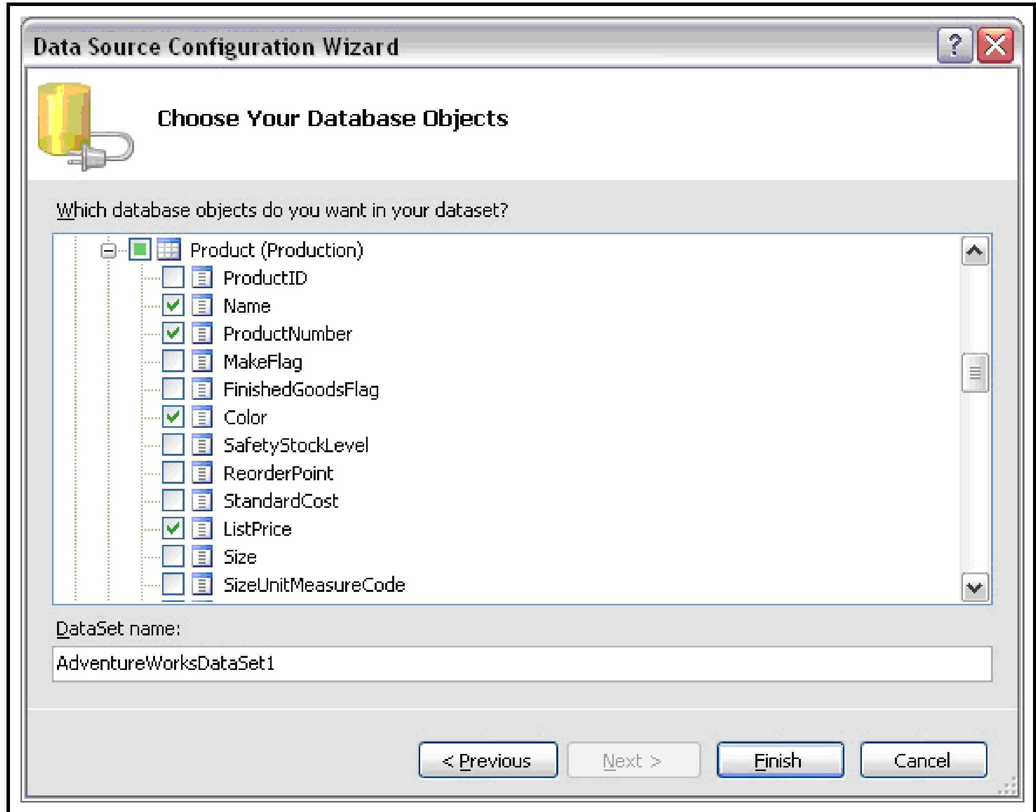
Şekil 247: Veri kaynağı belirlenmesi adımı



Şekil 248: New Connection ile bağlantı oluşturma adımı



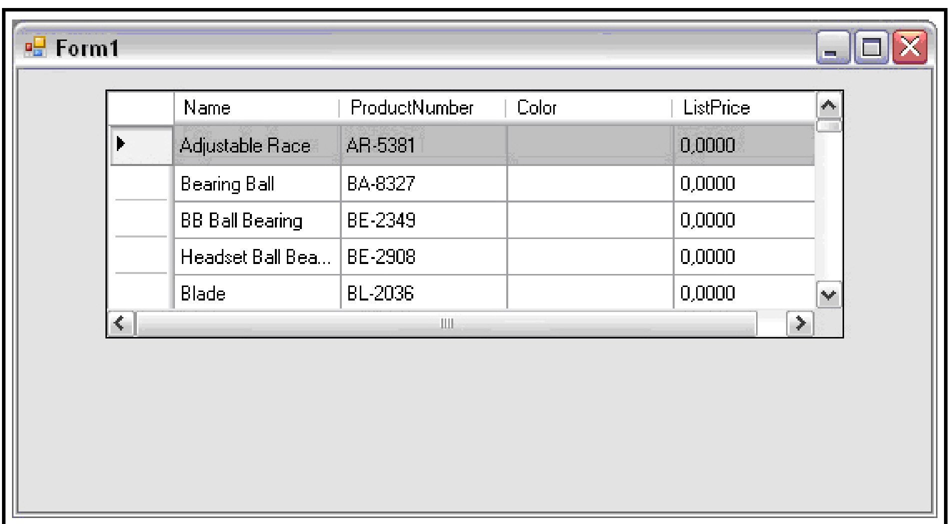
Şekil 249: Connection String bilgisinin konfigurasyon dosyasına eklenmesi



Şekil 250: Tablo ve alanlarının seçimi

Veri kaynağında yer alan hangi tablodaki kolonların çağırılacağı belirlenmelidir. Daha sonra Finish butonuna basılmalıdır.

Bu işlemleri yaptıktan sonra DataGridView kontrolünün içerisine veri kaynağından veriler çekilmiş olmaktadır.



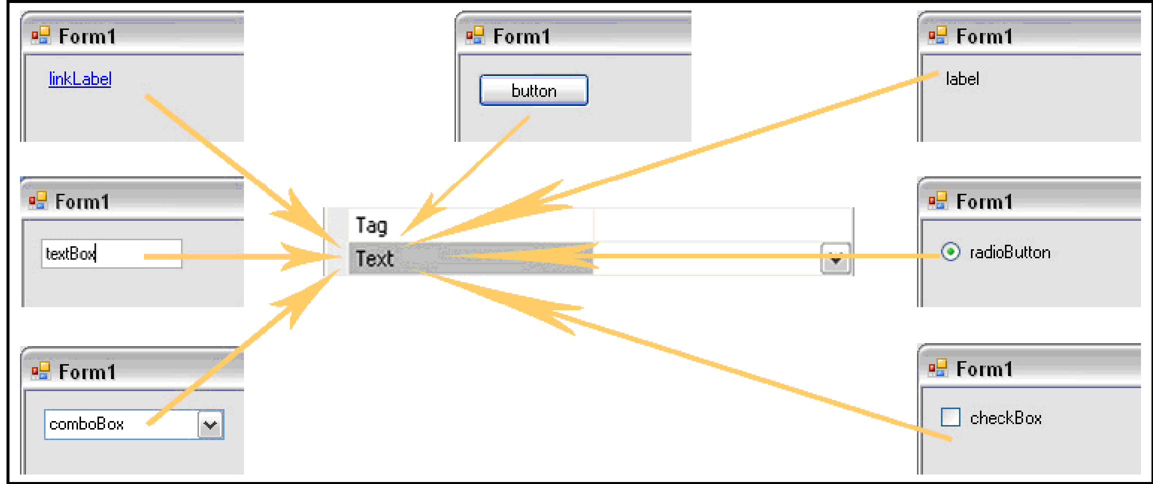
Şekil 251: Uygulamanın çalışma zamanındaki hali

Kontroller İçin Ortak Özellikler

Windows kontrollerinin, Control sınıfından türemesi nedeniyle, bu sınıftan türeyen windows kontrollerinin tümünün sahip olduğu bazı ortak özellikler vardır. Bu özellikler genellikle, uygulama geliştirirken çok sık kullanılacak özelliklerdir.

Text Özelliği

Windows kontrollerinden bazıları içerisinde bulunan metin (text) özelliği, Text özelliğinden (property) gelmektedir. TextBox, Label, LinkLabel, CheckBox, RadioButton, ComboBox, Button vb. kontrollerin Text özellikleri vardır. Bu kontrollerin Text özelliklerine bir değer girmek için Property penceresinde yer alan Text özelliği kullanılır.



Şekil 252: Text özelliğinin kullanıldığı örnek ortak kontroller

TextBox kontrolünün, text özelliğine çalışma zamanında (runtime) kod kısmından değer atanabilir ve yine çalışma zamanında (runtime) kullanıcıya dışarıdan da değer girdirilebilir. TextBox kontrolünün mevcut içeriği çalışma zamanında (runtime) text özelliğini (property) okuyarak da elde edilebilir. Aynı zamanda bir TextBox kontrolüne en fazla 2147483647 (integer veri tipinin maksimum değeri) karakter girilebilir.

Text özelliğine eklenen bir metin ComboBox kontrolünün bir öğesi değildir. Eğer kontrolün Text özelliğini boş bırakırsak varsayılan olarak kontrolün TextBox bölümü boş olacaktır. Eğer kullanıcı yeni bir öğe eklemek isterse; Text alanının dolu olması yeni bir öğe girilmesini engellemeyecektir. ComboBox içerisine öğeler eklendiğin de bu öğeler kontrolün TextBox bölümünde görülmeyecektir. ComboBox içerisinde örneğin "seçiniz" gibi bir uyarı çıkarmak için Text özelliği kullanılır.

Button kontrolünün Text özelliğine verilen metin değeri Button kontrolünün boyutlarını geçerse, metin bir alt satıra kayar. Bundan dolayı eğer uzunluk ayarlanmazsa metin kesilecektir. Text özelliği kullanıcıya Button kontrolünün ne iş yapacağını belirtir.

Bir kontrolün çalışma zamanında Text özelliğine değer atamak için aşağıdaki kodlar kullanılır.

```
Button.Text = "Button";
CheckBox.Text = "CheckBox";
CheckedListBox.Text = "CheckedListBox";
ComboBox.Text = "ComboBox";
Label.Text = "Label";
LinkLabel.Text = "LinkLabel";
ListBox.Text = "ListBox";
RadioButton.Text = "RadioButton";
TextBox.Text = "TextBox";
```

Şekil 253: Text özelliğine çalışma zamanında değer atanması

Bir kontrolün herhangi bir özelliğine ulaşmak için kontrolün adının yanında nokta operatörü kullanılır. Burada nokta operatörünü kullanılarak kontrollerin Text özelliğine erişilir ve kontrollerin Text özelliğine istenilen metin atanır.

Location Özelliği

Bir kontrolün Location özelliğini değiştirmek için Property penceresinde yer alan Location özelliğinden x ve y koordinatlarını değiştirmek gerekir. x ve y değerleri kontrolün form üzerinde yatay ve dikey konumlarıdır. Yatayda yerini değiştirmek için x, dikeyde y, her ikisinide değiştirmek için x ve y kooordinatlarını değiştirmek gerekmektedir.

⊟ Location	30; 30
X	30
Y	30

Şekil 254: Location özelliği

Kontrolün form üzerinde konumlanacağı yer Property penceresinden değiştirileceği gibi, form üzerinde seçilen kontrol sürüklenerek de istenilen yere yerleştirilebilir.

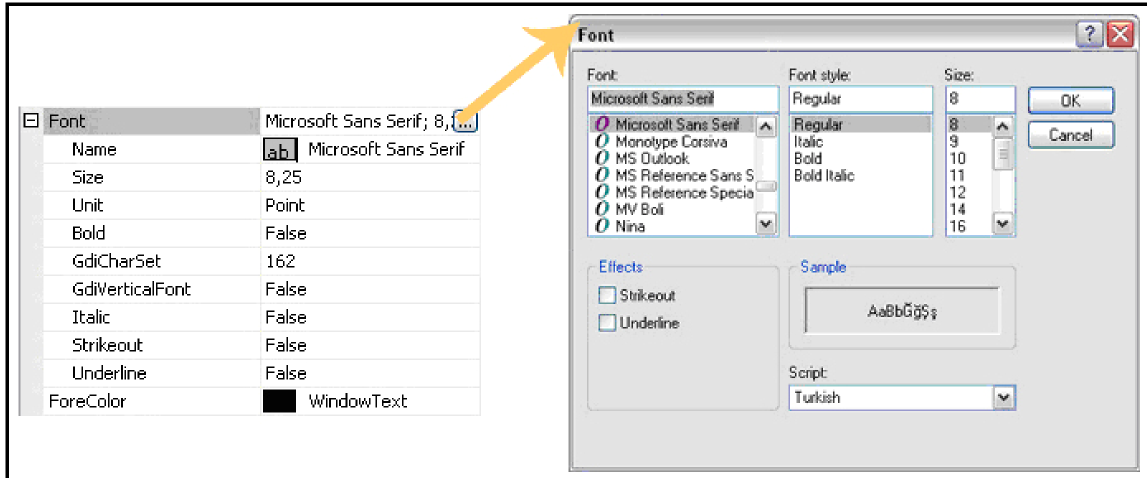
Bir kontrolün çalışma zamanında Location özelliğini değiştirmek için aşağıdaki kodlar kullanılır.

```
Button.Location = new System.Drawing.Point(80, 100);
CheckBox.Location = new System.Drawing.Point(80, 100);
CheckedListBox.Location = new System.Drawing.Point(80, 100);
ComboBox.Location = new System.Drawing.Point(80, 100);
Label.Location = new System.Drawing.Point(80, 100);
LinkLabel.Location = new System.Drawing.Point(80, 100);
ListBox.Location = new System.Drawing.Point(80, 100);
RadioButton.Location = new System.Drawing.Point(80, 100);
TextBox.Location = new System.Drawing.Point(80, 100);
```

Şekil 255: Location özelliğinin çalışma zamanında değiştirilmesi

Font Özelliği

Bir kontrolün Font özelliği Property penceresinde yer alan Font özelliğinden değiştirilebilir. Font özelliği kontrollerin metininin görünümü ile ilgili değişiklik yapmak için kullanılır.



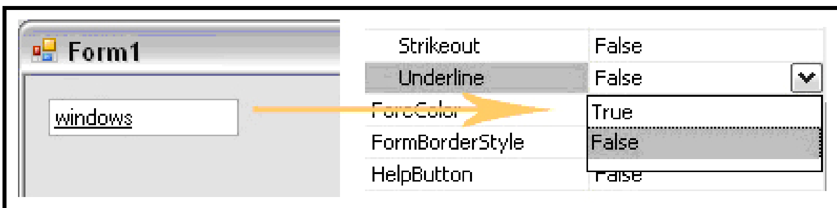
Şekil 256: Font özelliği

Kontrol içerisine girilen metinin Size, Italic, Underline ve Bold gibi özellikleri Font penceresinden değiştirilebileceği gibi Property penceresinden de değiştirilebilir.

```
// Button kontrolünün Font özelliklerini Property
//penceresinden değiştirdiğimizde formumuzun *.Designer.cs
//uzantılı dosyasında oluşan kod

Button.Font = new System.Drawing.Font("Verdana", 8F,
((System.Drawing.FontStyle)
(((System.Drawing.FontStyle.Bold | System.Drawing.FontStyle.Italic)
| System.Drawing.FontStyle.Underline))),
System.Drawing.GraphicsUnit.Point, ((byte)(162)));
```

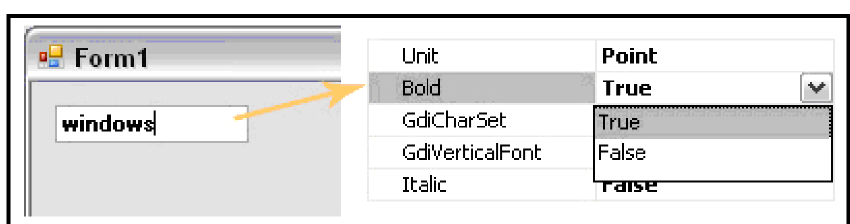
Şekil 257: Font özelliklerinin çalışma zamanında belirlenmesi

Yukarıdaki kodda kontrolün Font özelliğinin Name, Bold, Italic, UnderLine, Size gibi özellikleri değiştirilir. Name, metin alanların FontStyle'ını; Bold, kalın ya da ince yazılmasını; Underline, yazılan metinin altının çizili olmasını; Size ise boyutunu belirler.



Şekil 258: Underline özelliği

Kontrol içerisine girilen metinin altının çizili olması isteniyorsa Underline özelliği true yapılır.

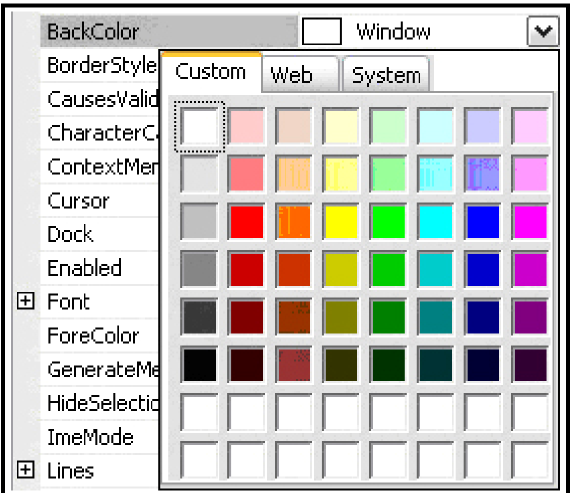


Şekil 259: Bold özelliği

Kontrol içerisine girilen metinin bold olması isteniyorsa; Bold özelliği true yapılmalıdır. Yine Property penceresinde yer alan Font özelliğinden, kontrollerin içerisine girilen metinin diğer özellikleri de değiştirilip, biçimlendirilebilir.

BackColor Özelliği

Bir kontrolün BackColor özelliği Property penceresinde yer alan BackColor özelliğinden değiştirilebilir. BackColor özelliği bir kontrolün arka fonunun rengini değiştirmektedir. TextBox, Label, LinkLabel, CeheckBox, RadioButton, ComboBox, Button vb. kontrollerinin BackColor özellikleri vardır.



Şekil 260: BackColor özelliği

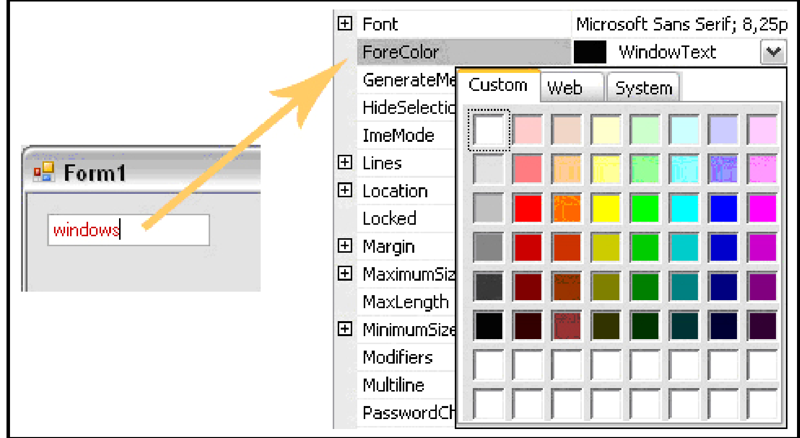
Bir kontrolün çalışma zamanında BackColor özelliğini değiştirmek için aşağıdaki kodlar kullanılabilir.

```
Button.BackColor = System.Drawing.Color.Red;
CheckBox.BackColor = System.Drawing.Color.Red;
CheckedListBox.BackColor = System.Drawing.Color.Red;
ComboBox.BackColor = System.Drawing.Color.Red;
Label.BackColor = System.Drawing.Color.Red;
LinkLabel.BackColor = System.Drawing.Color.Red;
ListBox.BackColor = System.Drawing.Color.Red;
RadioButton.BackColor = System.Drawing.Color.Red;
TextBox.BackColor = System.Drawing.Color.Red;
```

Şekil 261: BackColor özelliğinin çalışma zamanında elde edilmesi

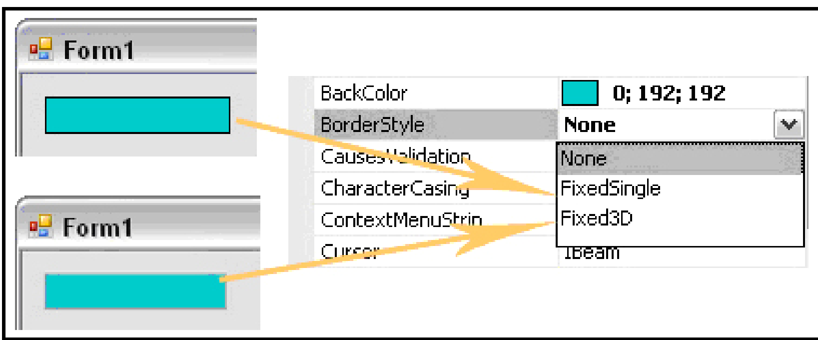
Herhangi bir kontrolün BackColor özelliği olduğu gibi formunda BackColor özelliği vardır. Eğer formun üzerinde bir Label kontrolü varsa ve formun BackColor özelliği değiştirildiyse Label'ın da BackColor' değeri formun BackColor değerini alır.

Bir kontrolün arka fonunun rengini değiştirmek, içerisine girilen metinin rengini değiştirmez. Bunun için kontrolün ForeColor özelliği kullanılır.



Şekil 262: ForeColor özelliği

Kontrolün fonunun rengi değiştirilebildiği gibi çerçevesinin (border) biçiminde de değişiklik yapılabilir. Bunun için de BorderStyle özelliğini değiştirmek gerekir.

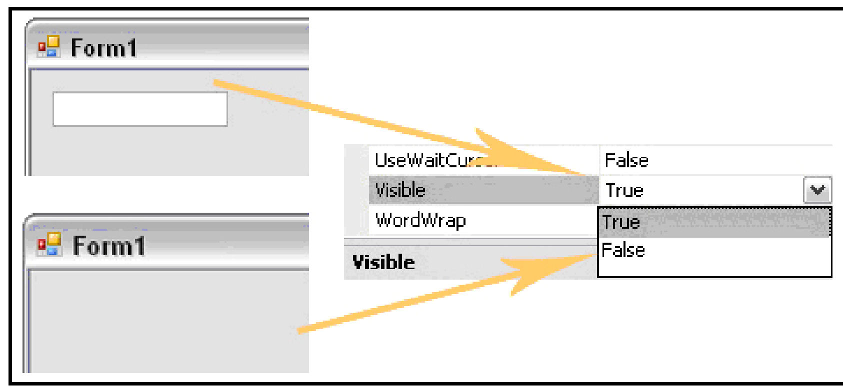


Şekil 263: BorderStyle özelliği

Button, CheckBox, ComboBox ve RadioButton kontrollerinin BorderStyle özelliği yoktur.

Visible Özelliği

Bir kontrolün Visible özelliği Property penceresinde yer alan Visible özelliğinden değiştirilir. Bu özellik kontrolün görünümü ile ilgilidir. Varsayılan olarak bütün kontrollerin visible özelliği true dur. Bu, kontrolün çalışma zamanında form üzerinde görüneceğini belli eder. Visible özelliği false olduğu zaman ise çalışma zamanında kontrol form üzerinde görünmez.



Şekil 264: Visible özelliği

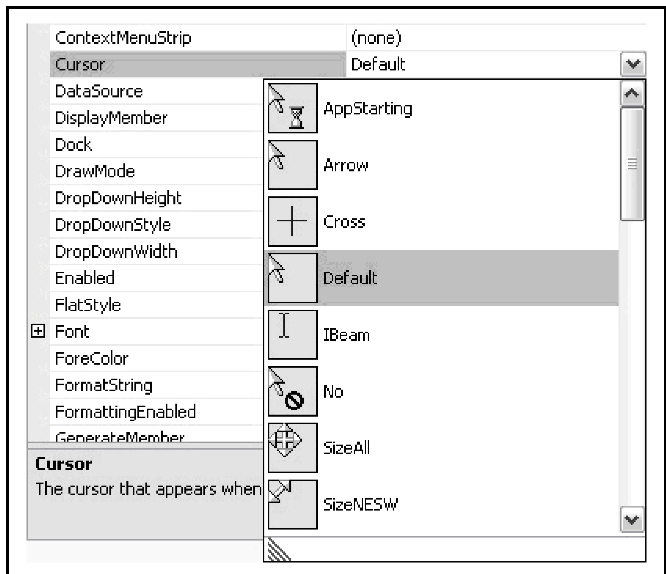
Çalışma zamanında kontrollerin Visible özelliğini değiştirmek için aşağıdaki kod kullanılır.

```
Button.Visible = false;
CheckBox.Visible = false;
CheckedListBox.Visible = false;
ComboBox.Visible = false;
Label.Visible = false;
LinkLabel.Visible = false;
ListBox.Visible = false;
RadioButton.Visible = false;
TextBox.Visible = false;
```

Şekil 265: Visible özelliğinin çalışma zamanında belirlenmesi

Cursor Özelliği

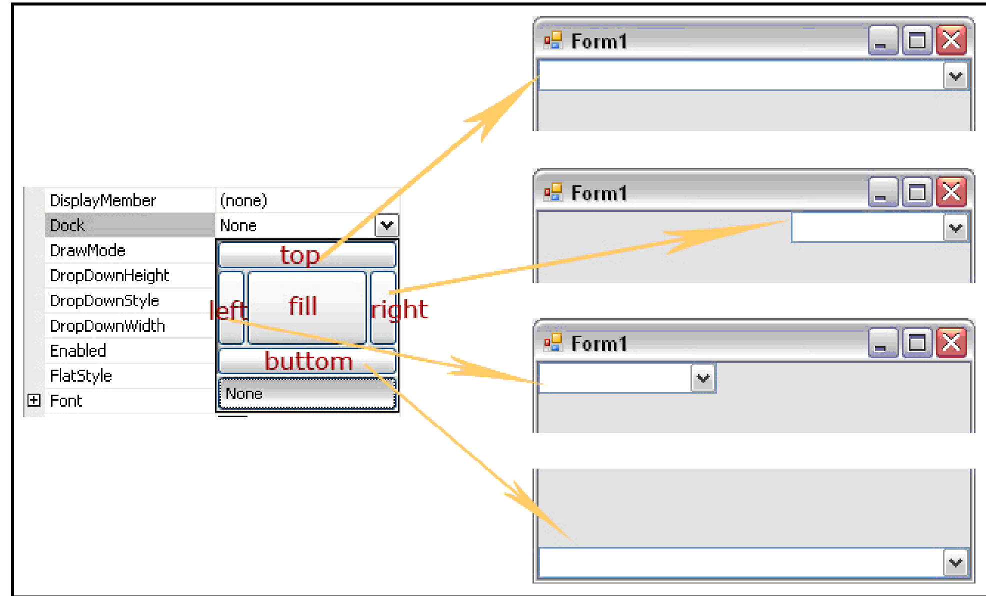
Bir kontrolün Cursor özelliğini değiştirmek için Property penceresinde yer alan Cursor özelliği kullanılır. Cursor özelliği farenin (mouse) kontrolün üzerine geldiği andaki görünümünü değiştirir. Bütün kontroller için kullanılan bir özelliktir. Fakat TextBox kontrolünde varsayılan hali Default değil IBeam dır.



Şekil 266: Cursor özelliği

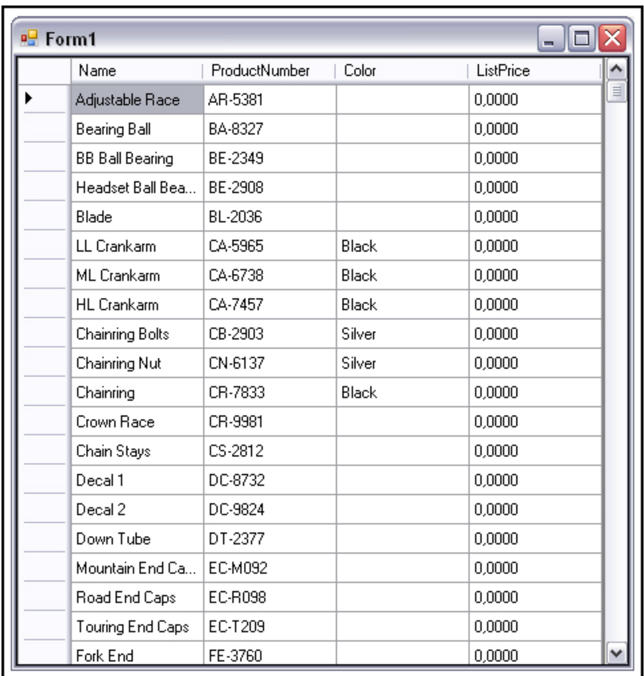
Dock Özelliği

Bir kontrolün Dock özelliğini değiştirmek için Property penceresinde yer alan Dock özelliği kullanılır. Dock özelliği bir kontrolün form üzerinde nerede konumlanacağını belirler, fakat dock özelliğine None dışında bir değer verildiğinde Location özelliğinden farklı olarak kontrolün yeri değişmez. Location özelliği kontrolün form üzerinde hangi konumda bulunacağını belli eder, Dock özelliği ise formun hangi ucunda konumlanacağını belli etmenin yanısıra kontrolün form üzerinde hareket ettirilmesini de engellemektedir.



Şekil 267: Dock özelliği

Dock özelliği DataGridView kontrolü için diğer kontrollerden farklı davranış gösterir. Eğer bir DataGridView kontrolünün Dock özelliği Fill değerinde kullanılırsa kontrol formun bütününü kaplayacaktır. Formun boyutunu değiştirdikçe DataGridView kontrolünün boyutuda form ile birlikte değişecektir.

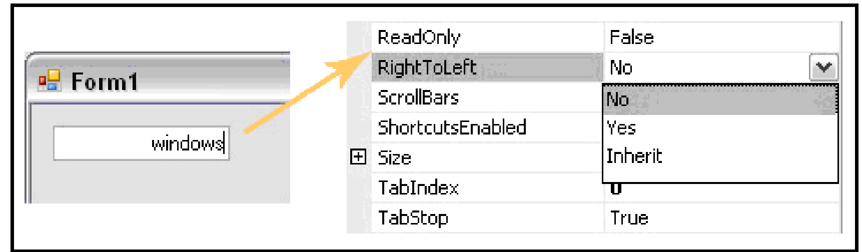


Name	ProductNumber	Color	ListPrice
Adjustable Race	AR-5381		0,0000
Bearing Ball	BA-8327		0,0000
BB Ball Bearing	BE-2349		0,0000
Headset Ball Bea...	BE-2908		0,0000
Blade	BL-2036		0,0000
LL Crankarm	CA-5965	Black	0,0000
ML Crankarm	CA-6738	Black	0,0000
HL Crankarm	CA-7457	Black	0,0000
Chainring Bolts	CB-2903	Silver	0,0000
Chainring Nut	CN-6137	Silver	0,0000
Chainring	CR-7833	Black	0,0000
Crown Race	CR-9981		0,0000
Chain Stays	CS-2812		0,0000
Decal 1	DC-8732		0,0000
Decal 2	DC-9824		0,0000
Down Tube	DT-2377		0,0000
Mountain End Ca...	EC-M092		0,0000
Road End Caps	EC-R098		0,0000
Touring End Caps	EC-T209		0,0000
Fork End	FE-3760		0,0000

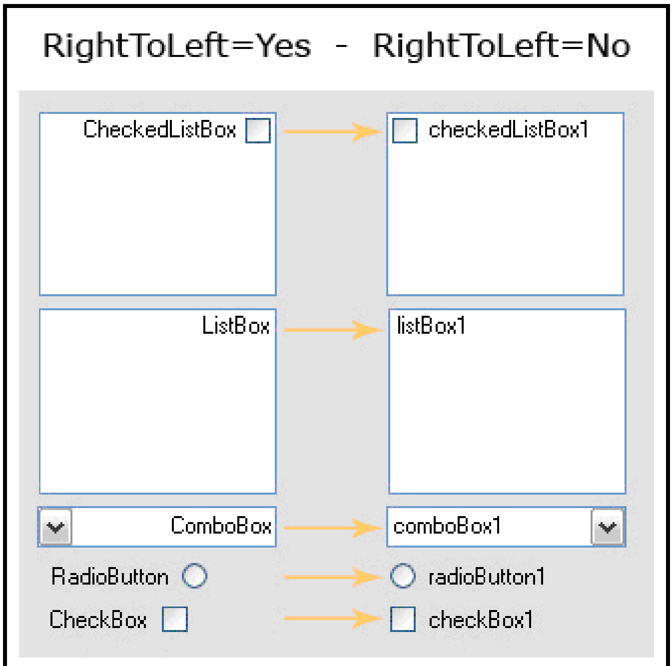
Şekil 268: DataGridView kontrolü için Dock özelliği kullanılmasının sonucu

RightToLeft Özelliği

Bir kontrolün RightToLeft özelliği Property penceresinde yer alan RightToLeft özelliğinden değiştirilebilir. Varsayılan olarak TextBox içerisine girilen metinin her bir karakteri soldan sağa sıralanırken, RightToLeft özelliği Yes yapıldığı zaman sağdan sola sıralanmaktadır.



Şekil 269: RightToLeft özelliği

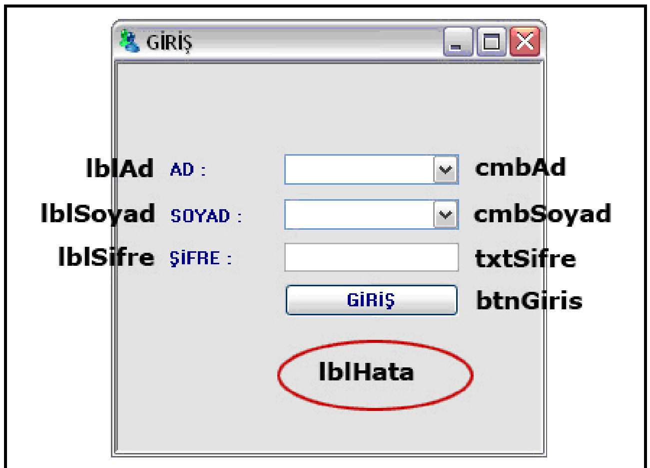


Şekil 270: RightToLeft özelliğinin farklı kontrollerdeki etkisi

BÖLÜM 2: PROJE

Projede, kullanıcının adı, soyadı ve şifresi ile giriş yaptıktan sonra adam asmaca oyununu oynayacağı bir uygulama anlatılıyor.

UsingWindows adında yeni bir windows uygulaması oluşturulur. Oluşturulan uygulamada yer alan Form1 formuna kullanıcının, adı, soyadı ve şifresi ile giriş yapabilmesi için kontroller eklenir. Kullanıcının adını ve soyadını seçebilmesi için iki tane ComboBox, şifresini girebilmesi için TextBox, bilgilerini girdikten sonra kontrol edilmesi için Button ve hatalı giriş yaptıysa ya da eksik bir bilgi girdiyse kullanıcıyı uyarmak için Label kontrolleri kullanılır. Form1 formunun text özelliği de "GİRİŞ" yapılır.

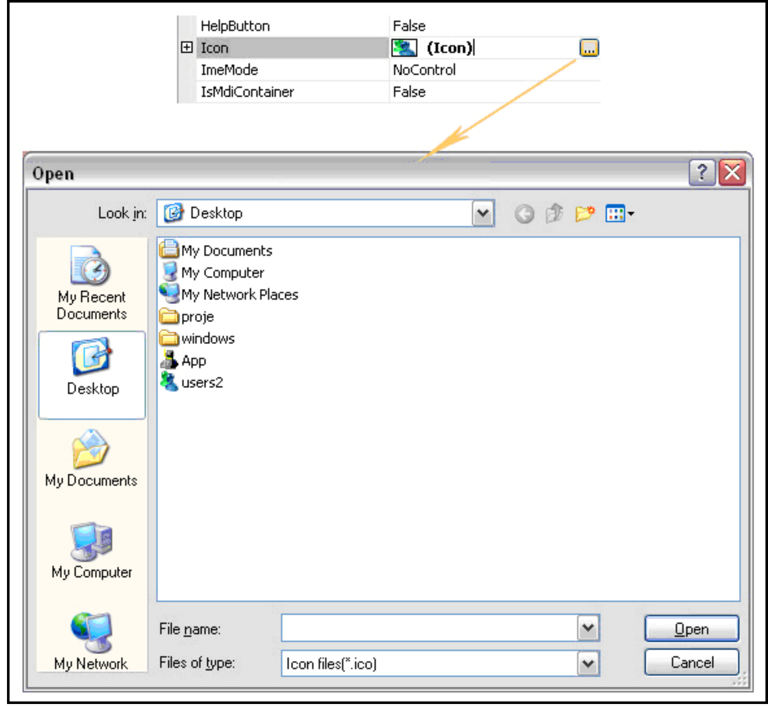


Şekil 271: Giriş formu için örnek ekran tasarımı ve kontrolleri

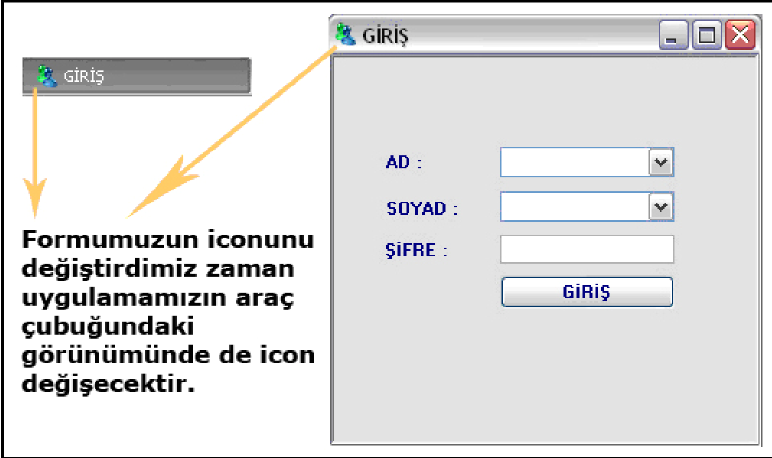
Hatalı Kullanıcı!

Şekil 272: Eğer kullanıcı yanlış giriş yaparsa çıkan hata mesajı

Icon ise özellikler penceresinde yer alan Icon özelliğinden forma eklenir. Bunun için Formun özellikler penceresinden Icon özelliğine tıklanır ve çıkan pencereden iconu seçtikten sonra Open butonuna basılır.



Şekil 273: Forma icon eklemek



Şekil 274: Icon görünümü

Kullanıcının bilgileri NETRON veritabanında oluşturulan Kullanici tablosundan kontrol edilir. Kullanici tablosunda yer alan KullaniciAdi, cmbAd ComboBox'ına, KullaniciSoyadi, cmbSoyad ComboBox'ına doldurulur. Kullanıcı adını, soyadını seçtikten sonra şifresini yazar ve GİRİŞ butonuna tıklar. Girilen bu veriler veritabanından kontrol edilerek kullanıcı eğer doğru kullanıcıysa oyun sayfasına geçmesi eğer değilse hata mesajı çıkması sağlanır.

Table - dbo.Kullanici

KullaniciID	KullaniciAdi	KullaniciSoyadi	Sifre
1	Burcu	Günel	Netron@01
2	Bülent	Sözge	Netron-01
3	Burak	Şenyurt	NeTrOn@1
4	Özgür	Altuntaş	Norten@01
5	Osman	Çokakoğlu	Netron@0
6	Emrah	Uslu	Uslu@01
7	Ercan	Bozkurt	NetErcan@02

Şekil 275: Kullanıcı giriş kontrolünün yapıldığı Kullanici tablosu
(Kullanıcının bilgileri NETRON adında oluşturulan database içerisindeki Personel tablosundan kontrol edilir.)

Kullanıcının Ad ve Soyad bilgilerini ComboBox kontorollerine eklemek için, öncelikle veri kaynağına bir bağlantı oluşturulmalıdır. Oluşturulan bağlantıda veri kaynağı daha önceden oluşturulan NETRON database'i olarak belirlenir. SqlCommand nesnesi yardımı ile veri kaynağındaki hangi tablodan ad ve soyad bilgilerinin çekileceği belirlenir. Gelen bilgiler SqlDataReader nesnesine atandıktan sonra, cmbAd ve cmbSoyad kontrollerine öğeler eklenir. Bu işlemler cmbAdSoyadDoldur isimli metodda yapılır. cmbAdSoyadDoldur metodu Form1 metodu içerisinde çağırılarak uygulama ilk çalıştığında görüntülenmesi sağlanır.

Kullanıcının şifresini gireceği TextBox kontrolünün PasswordChar özelliğine * karakteri atanır, böylece kullanıcının şifresinin başkası tarafından görünmesi engellenmiş olur.

Kullanıcının bilgilerinin doğruluğu KullaniciSorgu isimli metodda kontrol edilir. Veri kaynağına erişim sağlanarak, Kullanici tablosundan ad ve soyad bilgileri çekilir. Daha sonra, kullanıcının uygulamaya girdiği ad, soyad ve şifre değerleri ile karşılaştırılır ve eğer kullanıcı doğru kullanıcı ise adam asmaca oyunun olduğu diğer forma yönlendirilir, değilse "Hatalı Kullanıcı" uyarısı verilir. Kullanıcının bilgileri btnGiris butonuna tıklandığı an kontrol edildiği için, KullaniciSorgu metodu btnGiris butonunun Click olayında (event) çağırılır. Burada KullaniciSorgu metoduna kullanıcının kotrollere girdiği değerler gönderilir. btnGiris butonunun Click olayında (event) ilk önce girilen bilgilerin eksik olup olmadığı kontrol edilir. Eğer eksik bir bilgi girilmişse "Boş alanlar var uyarısı verilir", girilen eksik bir bilgi yoksa KullaniciSorgu metodu çağırılır ve kullanıcının girdiği bilgilerin doğruluğu kontrol edilir. Veri kaynağında önceden tanımlanmış bir kullanıcı ise, adam asmaca oyununa yönlendilir, değilse "Hatalı Kullanıcı" uyarısı verilir.

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
        cmbAdSoyadDoldur();
    }

    private void cmbAdSoyadDoldur() // ComboBox kontrollerinin öğelerini
eklemek için yazılan metot
    {
        SqlConnection con = new SqlConnection("data source=.;initial
catalog=Netron;integrated security=true"); // Bağlantı oluşturulur.
        SqlCommand cmd = new SqlCommand("select
KullaniciAdi,KullaniciSoyadi from dbo.Kullanici", con); // Veri
kaynağından istenilen veriler belirlenir.
        con.Open();
        SqlDataReader dr = cmd.ExecuteReader(); // Belirtilen veriler
```

```
SqlDataReader a alınır.

        while (dr.Read()) // Kullanici tablosundan istenilen veriler
alındıktan sonra bu veriler ComboBox kontrollerine eklenir.
        {
            cmbAd.Items.Add(dr[0]);
            cmbSoyad.Items.Add(dr[1]);
        }
        dr.Close();
        con.Close();
    }

    private void KullaniciSorgu(string ad,string soyad,string sifre) //
Giriş yapan kullanıcının doğru kullanıcı olup olmadığını kontrol eden
metot.
    {
        SqlConnection con = new SqlConnection("data source=.;initial
catalog=Netron;integrated security=true"); // Bağlantı oluşturulur.
        SqlCommand cmd = new SqlCommand("select * from
dbo.Kullanici",con); //Arama yapılacak verilerin neler olduğu veri
kaynağına söylenir.
        con.Open();
        SqlDataReader dr = cmd.ExecuteReader(); // İstenilen veriler veri
kaynağından alınır.

        while (dr.Read()) // Kullanıcının ComboBox ve TextBox
kontrollerine girdiği veriler veri kaynağındakiler ile karşılaştırılır ve
kullanıcının veri kaynağında olan bir kullanıcı olup olmadığı kontrol
edildikten sonra eğer doğru kullanıcı ise oyun sayfasına yönlendirilir,
doğru kullanıcı değilse hata mesajı çıkarılır.
        {
            if ((ad == dr[1].ToString()) && (soyad == dr[2].ToString()) &&
(sifre == dr[3].ToString()))
            {
                Form2 frm2 = new Form2();
                frm2.FormClosed += new
                FormClosedEventHandler(frm2_FormClosed);
                this.Hide();
                frm2.Show();
            }
            else
            {
                lblHata.Text = "Hatalı Kullanıcı!";
            }
        }
        dr.Close();
        con.Close();
    }

    void frm2_FormClosed(object sender, FormClosedEventArgs e)
    {
        Form2 frmSender = (Form2)sender;

        if (!frmSender.Exit)
        {
            Form2 frm2 = new Form2();
            frm2.FormClosed += new
            FormClosedEventHandler(frm2_FormClosed);
            frm2.Show();
        }
    }


    private void btnGiris_Click(object sender, EventArgs e)
    {
        if ((cmbAd.Text != "") && (cmbSoyad.Text != "") && (txtSifre.Text
!= "")) // Boş alanların olup olmadığı kontrol edilir. Eğer boş alanlar
yoksa, doğru kullanıcı kontrolü yapılır.
        {
```

```
            KullaniciSorgu(cmbAd.SelectedItem.ToString(),
cmbSoyad.SelectedItem.ToString(), txtSifre.Text.ToString(); // Kullanıcı
kontrolünün yapıldığı KullaniciSorgu metodunu çağırarak, eğer doğru
kullanıcı ise oyuna giriş yapması, değilse hata mesajı çıkması sağlanır.
        }
        else
        {
            lblHata.Text = "Boş alanlar var!";
        }
    }
}
```

Adam asmaca formunda, kullanıcının buton kontrollerini kullarak harflerini girebilmesi sağlanır. Bunun için a dan z ye butonlara ihtiyaç duyulur. Form üzerinde bulunan butonlar ToolBox dan sürükleyip bırakılmak yerine çalışma zamanında oluşturulur. Aynı işi yapacak birden fazla buton varsa bunları sürükleyip bırakıp, özelliklerini ayrı ayrı değiştirip, herbirinin Click olayında (event) aynı kodları yazmak yerine, çalışma zamanında buton kontrollerini tek bir kod bloğunda oluşturmak daha kolay olur. Bunun için buton kontrollerini oluşturacak bir ButtonOlustur() metodu yazılır ve bu metot Form1 metodu içerisinde çağrılarak formun açıldığı an görüntülenmesi sağlanır. Button kontrolleri oluşturulurken, öncelikle kaç tane butona ihitiyaç duyulacağı belirlenmelidir. Uygulamada seçilecek kelimelerin karakterleri birtek Türkçe olmayacağından dolayı 29 yerine 32 tane buton oluşturulur. Form üzerinde konumlanacakları x ve y konumları her bir kontrol için farklı olcağından, önceden x ve y adında iki tane değişken tanımlanır. Aynı zamanda buttonlar beşer tane yanyana olmak üzere sıralanacağı için, sıralarının kontrol edilebileceği bir sira değişkeni tanımlanır. Bu değişkenler tanımlandıktan sonra, for döngüsü içerisinde her bir butonun özellikleri, önceden oluşturulan harfler dizisinin elemanları kullanılarak oluşturulur. Buttonlar, Text ve Name özelliklerini harfler isimli diziden alırlar. Döngü içerisinde herbir buton için button sınıfından bir nesne örneği alınır ve böylelikle button kontrolllerinin özelliklerine erişilerek Location, Text, Name ve Size gibi özelliklerine erişilebilinir.

```
private void ButtonOlustur() // Button kontrollerinin oluşturulduğu metod
{
    int x = 30;
    int y = 30;
    int sira = 0;
    string[] harfler = new string[] { "A", "B", "C", "Ç", "D", "E", "F",
"G", "Ğ", "H", "I", "İ", "J", "K", "L", "M", "N", "O", "Ö", "P", "Q", "R",
"S", "Ş", "T", "U", "Ü", "V", "W", "X" , "Y", "Z" }; // Bütün harfleri
içinde bulunduran bir dizi tanımlanır. Bu dizinin eleman sayısı kadar
Button oluşturulacaktır.

    for (int i = 0; i < 32; i++)
    {
        Button button1 = new System.Windows.Forms.Button(); // Button
sınıfından bir nesne örneklenir.

        button1.Location = new System.Drawing.Point(x, y); // Button ların
herbirinin form üzerinde konumlanacağı yer belirlenir. Konumlanacakları
yer için sabit bir değer verilmez. Eğer burada x ve y değişkenleri yerine
sabit bir değer kullanılırsa, bütün butonlar üstüste oluşturulur. Bunun
için for döngüsü içerisinde her bir butonun yeri değiştirilir. Burada x
değişkeni butonun form üzerinde konumlanacağı x değerini, y değişkeni ise
y değerini belirlemektedir.

        button1.Click += new EventHandler(button1_Click); // Her bir
Button un Click olayı (event) tetiklendiğinde çalışacak olan olay metodu
bildirilir.

        this.Controls.Add(button1); // Yarttılan Button kontrolleri forma
dahil edilir.

        sira++; // Button kontrolleri beşer tane yanyana sıralanır. Beş
```

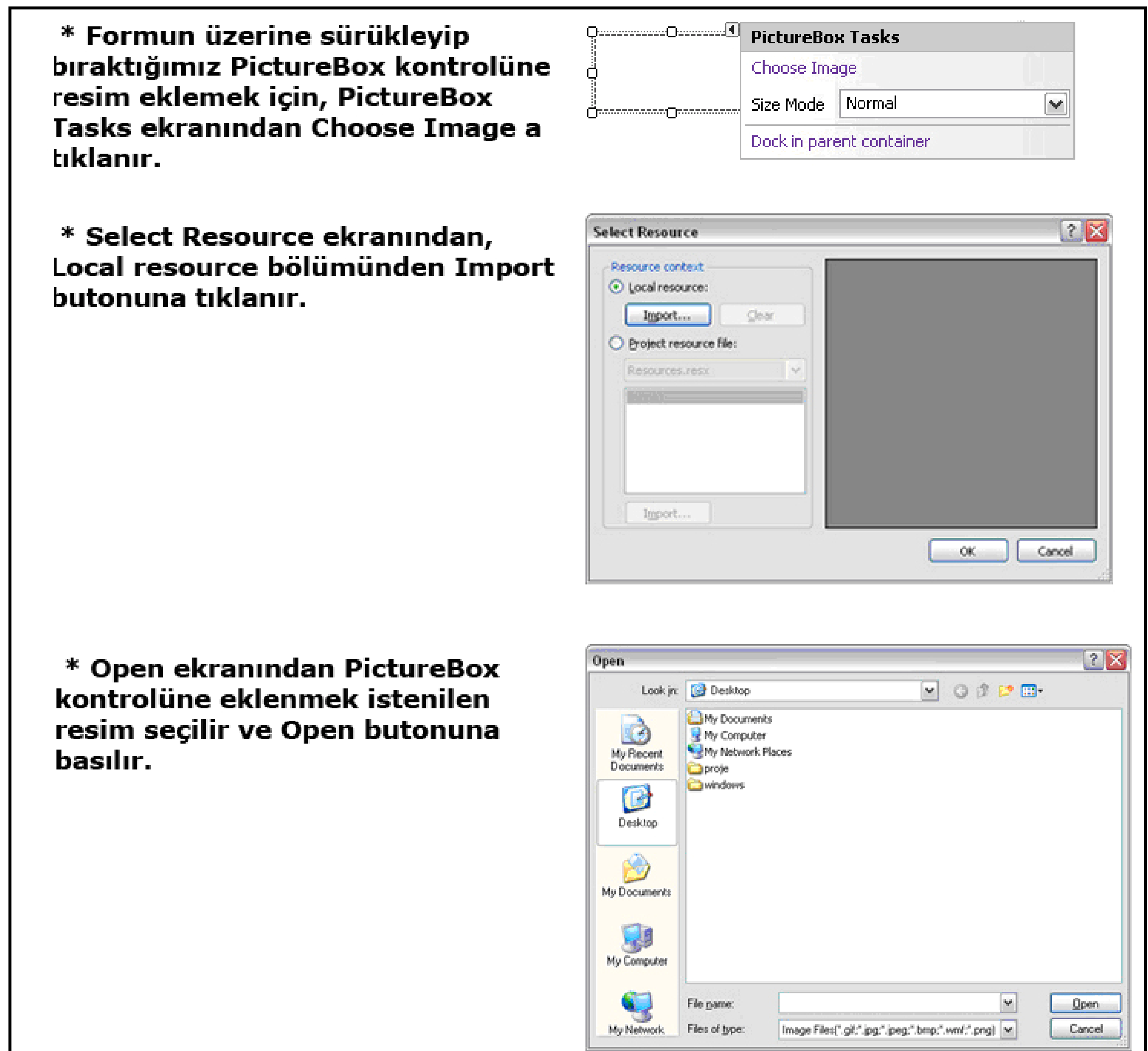
```
taneden sonra bir alta geçip diğer butonlar oluşturulur. Bunun için,
oluşturulan butonların kaçıncı sırada olduğu kontrol edilir.

        if (sira == 5) // Eğer sıra beş ise bir alt satırdan devam
edilecek. Button un Size ı y konumumda 25 olduğu için, y i 25 arttırıldı.
sira değişkeninin değeri tekrar sıfırlandı. Çünkü her bir satır için 5
tane yanyana buton oluşturulacak.
        {
            x = 30;
            y += 25;
            sira = 0;
        }
        else // Eğer sıra beş değilse yanyana butonlar sıralanır. Button
un Size ı x konumumda 30 olduğu için, x 30 arttırılır.
        {
            x += 30;
        }

        button1.Name = "btn" + harfler[i].ToString(); // Herbir butonun
Name özelliğine harfler dizisinin o anki değerini  atanır.
        button1.Size = new System.Drawing.Size(30, 25); //Butonların Size
ı belirlenir.
        button1.Text = harfler[i]; // Herbir butonun Name özelliğine
harfler dizisinin o anki değeri  atanır.
    }
}
```

Form üzerinde yer alan adam Picturebox kontrolü kullanarak oluşturulur. Önceden hazırlanmış resimler kontrole eklenir ve uygulmanın ilk çalışmasında PictureBox kontrollerinin Visible özellikleri false yapıır. Kullanıcının her yanlış tahmin ettiği harf için PictureBox kontrollerinin visible özellikleri true yapılarak adam asılır.

Windows formuna bir PictureBox kontrolü eklemek için, sol tarafta yer alan Toolbox içerisindeki Common Controls den bir PictureBox sürükleyip bırakılır ya da kontrolün üzerine çift tıklanır. Bu kontrol uygulamada yer alan resimleri düzgün bir formatta görüntülemek için kullanılır.

Şekil 276: PictureBox kontrolü

PictureBox kontrolüne resim eklemek için izlenmesi gereken adımlar aşağıda gösterilmektedir.



Şekil 277: PictureBox kontrolüne resim ekleme adımları

Bu işlemler gerçekleştirilince form üzerinde ki PictureBox a bir image (resim) eklenmiş olunur.

Kullanıcı eğer oyunu kaybederse ya da tekrar oynamak isterse oyunu bitirdiği zaman tekrar oynaması için btnTekrar adında bir buton kontrolü kullanılır. Oyun ilk başladığı zaman btnTekrar butonunun Visible özelliği false'dur. Kullanıcı oyunu kaybettiği ya da tekrar oynamak istediği zaman Visible özelliği true olur.

Formun iconu için, adam asmaca oyununa uygun bir icon seçilir ve formun özelliklerinden forma eklenir. Formun ControlBox özelliği false yapıldığından icon formun sol üst köşesinde görünmeyecektir, fakat uygulama çalıştığı zaman görev çubuğunda icon görünecektir. Controlbox özelliği forma ait icon, kapatma ve büytüme gibi özelliklerinin görünümlerini değiştirmek için kullanılır.

Forma ait MaximizeBox özelliği de false yapılır. Böylece formun sağ üst köşesinde bulunan büyütme kotrolü kullanılamaz. Formun FormBorderStyle özelliği de Fixed3D yapılır. Bu özellikte formun kullanıcı tarafından değiştirilmesini engeller.

Controlbox özelliği false yapıldığından, forma ait kapatma işlemi yapılamayacaktır. Bunun için btnCikis adında bir button kontrolü eklenir ve bu butonun Click olayında (event) uygulamadan çıkılır.

Şekil 278: Form için ControlBox, MaximizeBox, FormBorderStyle özellikleri

*** ControlBox özelliğini false yaparak formumuzun kapatma butonunu iptal etmiş oluruz. ControlBox özelliği formun sağ tarafında bulunan kapatma gibi özelliklerini kontrol eder.**

*** FormBorderStyle özelliği formun boyutunu sabitlemek için kullanılır. Formun FormBorderStyle özelliğini Fixed3D yaparak boyutu ile oynanmasını engellemiş olduk**

*** Formumuza bir icon verdiğimiz halde form üzerinde görünmemektedir. Çünkü ControlBox özelliğini false yaptık. Fakat uygulamayı açtığımızda araç çubuğunda icon görünecektir.**

Şekil 279: İkinci formun genel görünümü

Kullanıcı uygulamayı her çalıştırdığında ya da tekrar oynamak istediğinde yeni bir kelime seçilir. Bunun için kelimeler adında bir dizi oluşturulur. Uygulama her çalıştığında bu dizi içerisinden bir kelime seçilir ve bu seçilen kelimenin harf sayısı kadar Label kontrolleri oluşturulur. Kelimenin harf sayısı kadar oluşturulan Label'ların text özelliklerine "_" değeri verilir, kullanıcının her doğru tahmininde doğru tahmin ettiği harfe ait label kontrolünün text özelliği "_" yerine harf ne ise o olur. Bunun için oluşturulan Labelların herbirini Label tipinden bir diziye atayarak daha sonra o dizinin text değerlerinde değişiklik yapılır. Label dizisi tanımlanırken boyutu belirlenmez. Rastgele kelime seçildikten sonra boyutu belirlenir. Çünkü kelimenin harf sayısı kadar Label oluşturulacak ve Labellar oluşturulurken herbir Label, Label tipinden boyut dizisine atanacak. Labelların bir diziye atılmasının nedeni ise daha sonra uygulamanın herhangi bir yerinden istenildiği zaman ulaşılmak istenmesidir. Label tipinden oluşturduğumuz dizinin içerisine oluşturulan her bir Label kontrolü eleman olarak atanır ve her bir eleman, dizi Label tipinden olduğu için Label özelliklerine sahiptir.

LabelOlustur() metodunda, öncelikle rastgele bir sayı üretilir. Üretilen bu sayı dizinin elemanı olarak belirlenir ve daha önceden oluşturulan secilenKelime adlı değişkene dizinin o on üretttiği elemanın değeri atanır. secilenKelime değişkeninin uzunluğu Label tipinde oluşturulan boyut isimli dizinin eleman sayısı olarak belirlenir. Label kontrollerinin form üzerinde nerede konumlanacaklarını belirlemek için a ve b adında iki tane değişken tanımlanır. Tanımlanan bu iki değişken Label kontrolleri oluşturulurken Location özelliğide

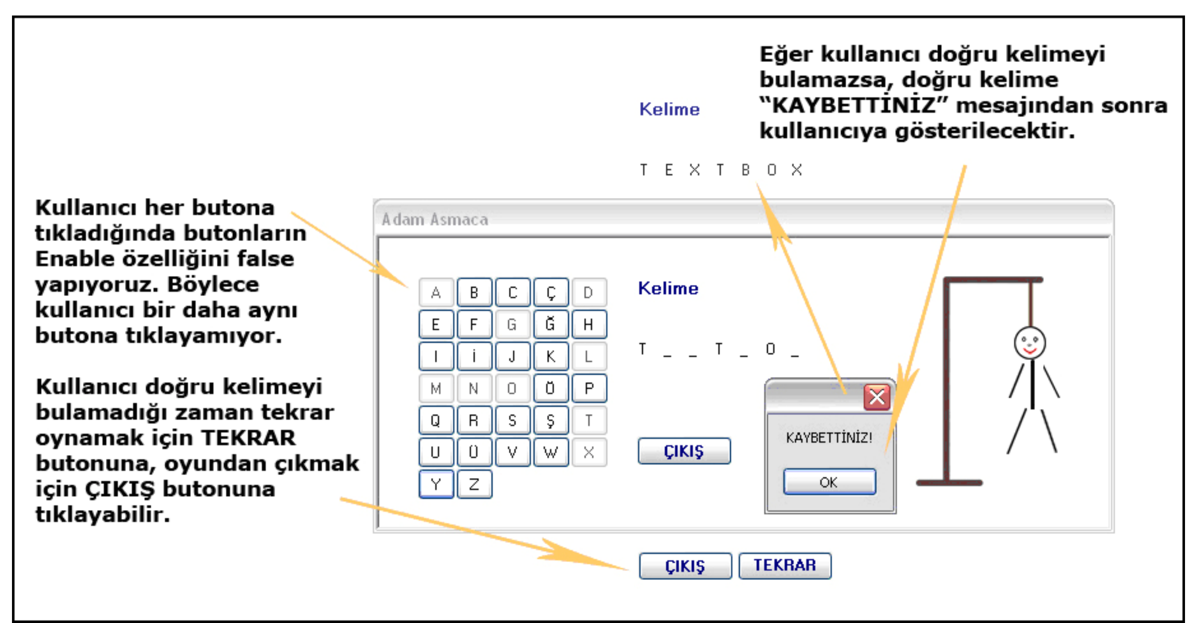
kullanılır. Her bir Label kontrolü yanyana sıralandığı için herhnagibir sıra belirlemeden labellar yanyana oluşturulur.

```
string[] kelimeler = new string[] { "WINDOWS", "AJAX", "TEXTBOX",
"COMBOBOX", "RADIOBUTTON", "CHECKBOX", "BUTTON", "VISTA", "INTRANET",
"BİLGİSAYAR", "YAZILIM", "MONİTÖR", "ANAKART" }; // Uygulama her
çalıştığında kelimeler dizisinin içerisinden bir kelime seçilir

. . .

Label[] boyut; // Oluşturulan Label kontrolllerini atamak için Label
tipinden bir dizi tanımlanır.

. . .

public void LabelOlustur() // Label kontrollerinin oluşturulduğu metot
{
    secilenKelime = kelimeler[rnd.Next(kelimeler.Length)];
    boyut=new Label[secilenKelime.Length];
    int a = 200;
    int b = 80;
    for (int i = 0; i < secilenKelime.Length; i++)
    {
        Label label1 = new System.Windows.Forms.Label();
        label1.Location = new System.Drawing.Point(a, b);
        this.Controls.Add(label1);
        a += 20;
        label1.Name = "lbl" + kelimeler.ToString();
        label1.Size = new System.Drawing.Size(17, 17);
        label1.Text = secilenKelime[i].ToString();
        boyut[i] = label1;
        boyut[i].Text = "_";
    }
}
```

Labelların oluşturulduğu LabelOlustur() metodu

Kullanıcı oyunu oynarken, tahmin ettiği bir kelimeyi bir daha seçemesin diye her tıkladığı butonun Enable özelliği false yapılır. Böylece tıklanan her bir butona bir daha tıklanamayacaktır. Kullanıcı tahminde bulunurken button click olayında (event) secilen kelimenin harfleri ile tahmin edilen harfler karşılaştırılır. Eğer seçilen kelime içerisinde yeralan bir harf tahmin edildiyse, kelimenin o harfinin visible'ı true olur, yanlış bir tahminde bulunulduysa AdamAs() metodu çağırılır ve sıra ile daha önceden oluşturulan PictureBox ların visible ları true olur. Her yanlış tahminde daha önceden sayac adında tanımladığımız değişkenin değeri bir arttırılır, böylece PictureBox kontrollerinin Visible özellikleri sıra ile true yapılmış olunur. AdamAs metodu içerisinde sayac değişkeninin değeri switch koşulu ile kontrol edilir ve kullanıcının kaçıncı yanlış tahmini olduğuna göre PictureBox'ların Visible'ları true olur.

```
btn.Enabled = false; // Tıklanan her butonun bir daha tıklanmasını
engellemek için enabled özelliği false yapılır.
```



Şekil 280: Uygulamanın Çalışma zamanındaki görünümü

Adam Asmaca Formu kodları

```
public partial class Form2 : Form
{
    Random rnd = new Random();
    string[] kelimeler = new string[] { "WINDOWS", "AJAX", "TEXTBOX",
"COMBOBOX", "RADIOBUTTON", "CHECKBOX", "BUTTON", "VISTA", "INTRANET",
"BİLGİSAYAR", "YAZILIM", "MONİTÖR", "ANAKART" }; // Uygulama her
çalıştığında kelimeler dizisinin içerisinden bir kelime seçilir
    Label[] boyut; // Oluşturulan Label kontrollerini atamak için Label
tipinde bir dizi tanımlanır.
    string secilenKelime; // Kelimelr dizisinden rasgele üretilen kelime,
secilenKelime değişkenine atanır.
    int sayac = 0; //Kullanıcı kelimenin herhangi bir harfini her
bilemediğinde sayac bir artar. Böylece adam, belli bir sıra takip edilerek
asılabilir.
    int sayac2 = 0; //Kullanıcı kelimenin herbir harfini bildiğinde bir
artar.

    private bool _exit = false;

    public bool Exit
    {
        get { return _exit; }
        set { _exit = value; }
    }

    public Form2()
    {
        InitializeComponent();
        ButtonOlustur(); // Button kontrollerinin oluşturulduğu metot
çağrımı
        LabelOlustur(); // Label kontrollerinin oluşturulduğu metot
çağrımı
    }

    private void Form2_Load(object sender, EventArgs e)
    {
        Form1 frm1 = new Form1();
        frm1.Close();
    }
```

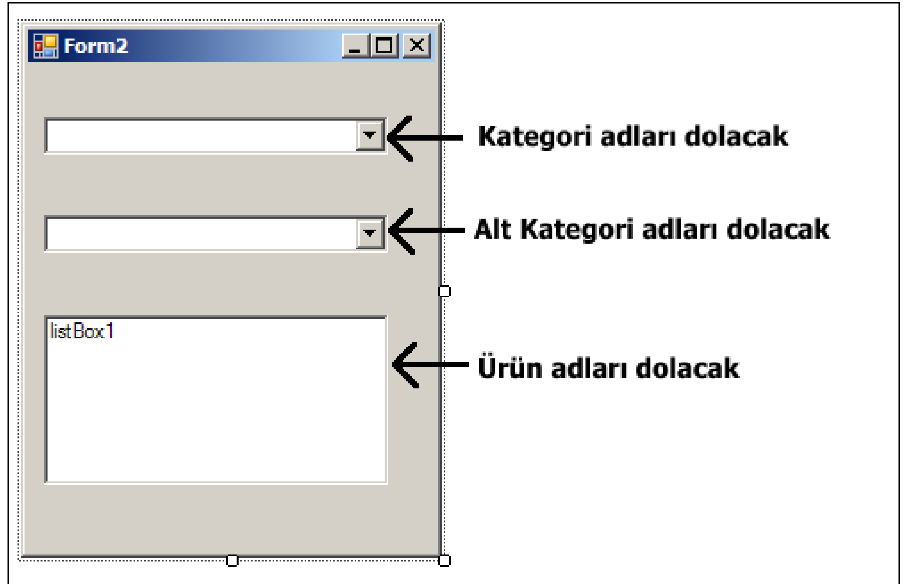
```
    private void ButtonOlustur() // Button kontrollerinin oluşturulduğu
metot
    {
        int x = 30;
        int y = 30;
        int sira = 0;
        string[] harfler = new string[] { "A", "B", "C", "Ç", "D", "E",
"F", "G", "Ğ", "H", "I", "İ", "J", "K", "L", "M", "N", "O", "Ö", "P", "Q",
"R", "S", "Ş", "T", "U", "Ü", "V", "W", "X" , "Y", "Z" }; // Bütün
harfleri içinde bulunduran bir dizi tanımlanır. Bu dizinin eleman sayısı
kadar Button oluşturulur.

        for (int i = 0; i < 32; i++)
        {
            Button button1 = new System.Windows.Forms.Button(); // Button
sınıfından bir nesne örneklenir.
            button1.Location = new System.Drawing.Point(x, y); // Button
ların herbirinin form üzerinde konulanacağı yer belirlenir.
            button1.Click += new EventHandler(button1_Click); // Her bir
Button un Click olayı (event) tetiklendiğinde çalışacak olan olay metodu
bildirilir.
            this.Controls.Add(button1); // Oluşturulan Button kontrolleri
forma dahil edilir.
            sira++; // Button kontrollerini beşer tane yanyana sıralanır.
Beş taneden sonra bir alta geçip diğer butonlar oluşturulur. Bunun için
oluşturulan butonların kaçıncı sırada olduğu kontrol edilir.

            if (sira == 5) // Eğer sıra beş ise bir alt satırdan devam
edilir. Button un Size ı y konumumda 25 olduğu için, y 25 arttırılır. Sira
tekrar sıfırlanır. Çünkü her bir satır için 5 tane yanyana buton
oluşturulur.
            {
                x = 30;
                y += 25;
                sira = 0;
            }
            else // Eğer sıra beş değilse yanyana butonlar sıralanır.
Button un Size ı x konumumda 30 olduğu için, x 30 arttırılır.
            {
                x += 30;
            }

            button1.Name = "btn" + harfler[i].ToString(); // Herbir
butonun Name özelliğine harfler dizisinin o anki değeri atanır.
            button1.Size = new System.Drawing.Size(30, 25); //Butonların
Size ı belirlenir.
            button1.Text = harfler[i]; // Herbir butonun Name özelliğine
harfler dizisinin o anki değerini  atanır.
        }
    }
    private void button1_Click(object sender, EventArgs e)
    {
        Button btn = (Button)sender;

        bool bulundu = false; // Eğer kullanıcı doğru harfi bulduysa true
değerini alır.
        btn.Enabled = false; // Tıklanan her butonun bir daha tıklanmasını
engellemek için enabled özelliği false yapılır.

        if (sayac >= 6) // Eğer kullanıcı adamı astıysa KAYBETTİNİZ mesajı
çıkarılır. Aynı zamanda kullanıcının bir daha başka bir buttona bastığında
oyunun devam etmemesi sağlanır.
        {
            MessageBox.Show("KAYBETTİNİZ!");
            for (int i = 0; i < secilenKelime.Length; i++)
            {
                boyut[i].Text = secilenKelime[i].ToString();
            }
            btnTekrar.Visible = true; // Eğer oyun kaybedildiyse
```

```
kullanıcının tekrar oynaması için visible özelliği true yapılır.
        }
        else if (sayac < 7) // Kullanıcı oyun hakkını kaybetmediyse
oynamaya devam edebilmesi için bu kontrol yapılır.
        {
            for (int i = 0; i < secilenKelime.Length; i++)
            {
                if (secilenKelime[i].ToString() == btn.Text)
                {
                    boyut[i].Text = secilenKelime[i].ToString();
                    bulundu = true;
                    sayac2++;
                }
            }
        }

        if (!bulundu) // Eğer kullanıcı doğru harfi girmediyse AdamAs
metodu çağırılır.
        {
            bulundu = false;
            sayac++;
            AdamAs();
        }
        if (sayac2 == secilenKelime.Length) // Eğer kullanıcı doğru
kelimeyi bulursa tekrar oynama şansı vermek için btnTekrar butonunun
visible özelliği true yapılır.
        {
            MessageBox.Show("TEBRİKLER");
            btnTekrar.Visible = true;
        }
    }

    public void LabelOlustur() // Label kontrollerinin oluşturulduğu metot
    {
        secilenKelime = kelimeler[rnd.Next(kelimeler.Length)];
        boyut=new Label[secilenKelime.Length];
        int a = 200;
        int b = 80;
        for (int i = 0; i < secilenKelime.Length; i++)
        {
            Label label1 = new System.Windows.Forms.Label();
            label1.Location = new System.Drawing.Point(a, b);
            this.Controls.Add(label1);
            a += 20;
            label1.Name = "lbl" + kelimeler.ToString();
            label1.Size = new System.Drawing.Size(17, 17);
            label1.Text = secilenKelime[i].ToString();
            boyut[i] = label1;
            boyut[i].Text = "_";
        }
    }

    private void AdamAs() // Kullanıcının her yanlış girişinde adamı asmak
için yazılan metot
    {
        switch (sayac) // Kullanıcının bilemediği durumlarda arttırılan
sayac değişkenin değerine göre sıra ile PictureBox kontrollerinin
görünümleri değiştirilir.
        {
            case 1:
                pcbKafa.Visible = true;
                break;
            case 2:
                pcbGovde.Visible = true;
                break;
            case 3:
                pcbSolKol.Visible = true;
                break;
            case 4:
```

```
                pcbSagKol.Visible = true;
                break;
            case 5:
                pcbSolBacak.Visible = true;
                break;
            case 6:
                pcbSagBacak.Visible = true;
                break;
        }
    }

    private void btnTekrar_Click(object sender, EventArgs e)
    {
        this.Close();
    }

    private void btnCikis_Click(object sender, EventArgs e) //
Kullanıcının uygulamadan tamamen çıkması imkanı verilir.
    {
        _exit = true;
        Application.Exit();
    }
}
```

KISIM SONU SORULARI:

1) AdventureWorks veritabanında yer alan ProductCategory, ProductSubCategory ve Product tablolarından faydalanarak aşağıdaki ekran görütünsüne benzer bir windows uygulaması yazmaya çalışınız.



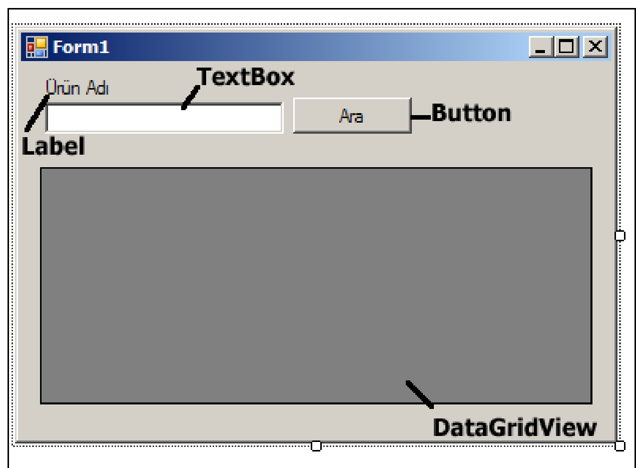
Şekil 281: Form tasarımı

2) Kullanıcıdan aldığı liste fiyatı bilgisine göre, Product tablosundan ListPrice değeri ilgili fiyat değerinden küçük olan satırların Name ve ProductSubCategoryId değerlerini bir ListBox kontrolüne dolduracak Windows uygulamasını yazınız.

3) Bir windows formu üzerine A' dan Z' ye kadar Button nesnelerini dinamik olarak ekleyecek kod parçasını yazınız.

4) Bir windows formu üzerindeki TextBox kontrollerinin tamamının içeriğini dinamik olarak temizleyecek kod parçasını yazınız.

5) Product tablosundan Name alanının değeri bir TextBox' a girilen değere benzer olan ürünleri bulan ve Form üzerindeki DataGridView kontrolünde gösterecek olan windows uygulamasını yazınız.



Şekil 282: Form tasarımı

KISIM VI: ASP.NET ile WEB UYGULAMALARI GELİŞTİRMEK

YAZARLAR:
UĞUR UMUTLUOĞLU
EMRAH USLU

BÖLÜM 0: WEB UYGULAMALARINA GİRİŞ

Temel Web Teknolojileri

Web Uygulamalarının Gelişimi

İnternet ilk ortaya çıktığında en büyük sorun, web sayfalarının nasıl ifade edileceği problemiydi. Takvimler 1990 yılını gösterdiğinde Tim Berners-Lee isimli bir programcı **HTML (HyperText Markup Language)** ismini verdiği bir dilin temellerini atmaya başladı. HTML, içinde metinler ve başka sayfalara bağlantılar olan bir dildi ve bu dil tarayıcı (browser) adı verilen programlar aracılığı ile yorumlanabiliyordu.

HTML, 1990 yılından bugüne kadar bazı güncellemeler ve yeniliklerle geliştirildi. Fakat HTML tarafında yapılan bu yenilikler ve güncellemeler, web uygulamalarının istenilen seviyeye gelmesi için yeterli olmadı. Çünkü HTML yapısı itibariyle web sayfalarını sadece durağan (statik) olarak sunabiliyor ve web sayfaları ziyaretçisi ile etkileşim içerisinde olamıyordu. Bu sebeple zaman içerisinde HTML'in geliştirilmesinden ziyade başka teknolojilerin geliştirilmesi zorunlu hale geldi. **CSS (Cascading Style Sheets)** şablonları ve **JavaScript** dili gibi istemci tarafındaki gelişmeler ile **CGI, PERL, PHP, ASP** ve **ASP.NET** teknolojileri gibi sunucu tarafındaki gelişmeler web uygulamalarının bugüne gelmesindeki en büyük rolleri üstlendi.

HTML'in dinamik sayfalar oluşturmaya imkan vermemesi nedeni ile bu konudaki eksiği kapatmak için sahneye ilk çıkarılan teknoloji **CGI (Common Gateway Interface)** oldu. CGI bir çok sitede HTML'in eksikliklerini kapatmak ve dinamik sayfalar oluşturmak için kullanıldı. Ancak, gerek yapısındaki eksiklikler, gerekse kullanılması çok zor olan bir dil olması nedeniyle zaman içerisinde programcılar başka arayışlar içerisinde girdi. Bu arayışlar sonucunda ortaya PHP ve ASP teknolojileri çıktı. **PHP** (Personal Home Page veya Personal Hypertext Preprocessor), kendine kişisel bir site yapmak isteyen *Rasmus Lerdorf* tarafından 1993 yılında geliştirilmeye başlandı. Adının ilk duyulmasından sonra oldukça popüler olan PHP, geçen sürede bu özelliğinden hiçbir şey yitirmeden bugüne kadar geldi. **ASP** ise Microsoft tarafından internet dünyasındaki bu gelişmelerin ardından 1996 yılında 1.0 sürümü ile duyuruldu. Genelde **VBScript** kodları ile yazılan ASP uygulamaları da, kısa zamanda hak ettiği ilgiyi buldu ve geniş bir kullanıcı kitlesine yayıldı. ASP teknolojisi, versiyon 3.0'a kadar geliştirilmeye devam edildi. 2000 yılında duyurulan ASP 3.0, "Klasik ASP"nin son sürümü oldu. Sıra ASP.NET'teydi...

Microsoft ASP'nin alt yapısındaki bazı problemler ve eksikliklerden dolayı, klasik ASP'yi geliştirmeyi bıraktı ve .NET platformu üzerinde çalışacak, web teknolojilerinde yeni bir yaklaşım olan **ASP.NET**'i geliştirdi. ASP.NET, .NET platformunda yer alan birçok dille yazılabilen, tam olarak nesneye yönelimli yeni bir teknoloji olarak ortaya çıktı. ASP.NET bugünkü yapısıyla, yazılım geliştiricilere çok kısa sürede, ileri seviye web uygulamaları geliştirebilmelerini sağlayan sunucu taraflı bir teknolojidir. 2005 yılı içerisinde 2.0 versiyonu duyurulan ASP.NET, Microsoft tarafından geliştirilmeye devam edilmektedir.

Web uygulamaları geliştirilirken, farklı teknolojilerden yararlanılabilmektedir. HTML ile sadece web sayfalarının nasıl görüntüleceği belirlenirken, istemci ve sunucu tarafındaki bazı teknolojiler de kullanılarak, web sayfalarının nasıl çalışacağı, ziyaretçilerle nasıl etkileşimde bulunacağı belirlenebilir. Web programcısı, profesyonel web uygulamaları geliştirebilmek için, birden fazla teknolojiye hakim olmalıdır. Genel olarak HTML, XML, JavaScript ve sunucu taraflı en az bir teknolojiye hakim olmak yeterlidir.

XML (eXtensible Markup Language)

Genişletilebilir **İşaretleme Dili** olan XML, hem insanlar, hem de bilgi işletim sistemleri tarafından kolayca okunabilen dokümanların oluşturulabilmesini sağlayan bir işaretleme dilidir. Bilgi teknolojilerinin gelişmesiyle birlikte, farklı standartlar oluşturulmakta ve bu standartlar arasında bazı uyumsuzluklar ortaya çıkmaktadır. XML'in asıl amacı farklı sistemler arasında veri alışverişi sağlamaktır. XML teknolojisi, metinsel içeriğe sahip olduğu ve bir **standart** haline geldiği için, günümüzde bir çok platform ve programlama dili tarafından tanınabilmektedir. Bu sayede, XML tüm platformlar arasında verinin taşınabilmesini sağlayan köprü vazifesi görmektedir.

XML dokümanları ağaç yapısında, yani hiyerarşik bir yapıdadırlar. Aşağıda basit bir XML kodu bulunmaktadır.

```
<diller>
   <dil isim="HTML" yon="Istemci" />
   <dil isim="ASP.NET" yon="Sunucu" />
   <dil isim="JS" yon="Istemci" />
   <dil isim="PHP" yon="Sunucu" />
</diller>
```

Standartlara uygun olarak yazılmış olan bir HTML dokümanı, aynı zamanda XML yapısında bir doküman olarak düşünülebilir.

JavaScript

JavaScript **istemci tarafında** çalışan ve amacı HTML'in statik yapısına zenginlik kazandırmak olan, istemci taraflı bir script dilidir. JavaScript ilk olarak Netscape tarafından geliştirilmiş ve piyasaya sürülmüştür. 1995 yılında ise C dilinin tarayıcılara uyarlanmış haline çevrilerek güçlenen JavaScipt, kısa zamanda büyük ilgi toplamıştır. HTML gibi statik bir dünyadan dinamik bir dünyaya geçiş, Microsoft'un da bu teknolojiye ilgi duymasına sebep oldu. Microsoft, JavaScript diline gerek yapı, gerek işlevsellik açısından çok benzeyen ve **JScript** adını verdiği bir dil geliştirdi. Ancak standartlaşmamış JScript kodları, Internet Explorer dışındaki tarayıcılarda istenen şekilde çalıştırılamadı. Aşağıda bir web sayfasına "Merhaba Dünya" yazılmasını sağlayan basit bir JavaScript kodu bulunmaktadır.

```
<script type="text/javascript">
      document.write("Merhaba Dünya");
</script>
```

HTML (HyperText Markup Language)

HTML, etiket (tag) adı verilen yapılardan oluşan, metin tabanlı bir işaretleme dilidir. HTML ile sayfada yer alacak içeriklerin (yazılar, resimler, animasyonlar, form bileşenleri), sayfanın neresinde, nasıl ve ne şekilde gösterileceği belirlenir. HTML dili, Internet Explorer veya diğer tarayıcılar (browser) tarafından yorumlanarak görsel bir içerik şeklinde sunulur. Tarayıcıların yaptığı işlem, sayfa içerisindeki bileşenleri alarak, HTML içerisinde tanımlandığı şekilde görüntülenmesini sağlamaktır.

FrontPage ve Dreamweaver gibi web sayfası editörleri kullanılarak, HTML kodu yazmadan da web sitesi oluşturabilmek mümkündür. Çünkü bu editörler bir arayüzden tasarım yapma imkanı verirler ve HTML kodlarını arka planda kendileri oluştururlar.

HTML'in bazı önemli özellikleri şöyle sıralanabilir.

- HTML, Notepad gibi basit bir metin editörü ile yazılabilecek bir dildir. Yazılan kodlar, uzantısı **.htm** ya da **.html** olan bir dosya olarak kaydedilirse, bu dosya bir tarayıcı tarafından sadece yorumlama işlemi sonucunda görüntülenebilir. HTML dosyaları herhangi bir derleyiciye (compiler) ihtiyaç duymaz. Web sunucusu gelen

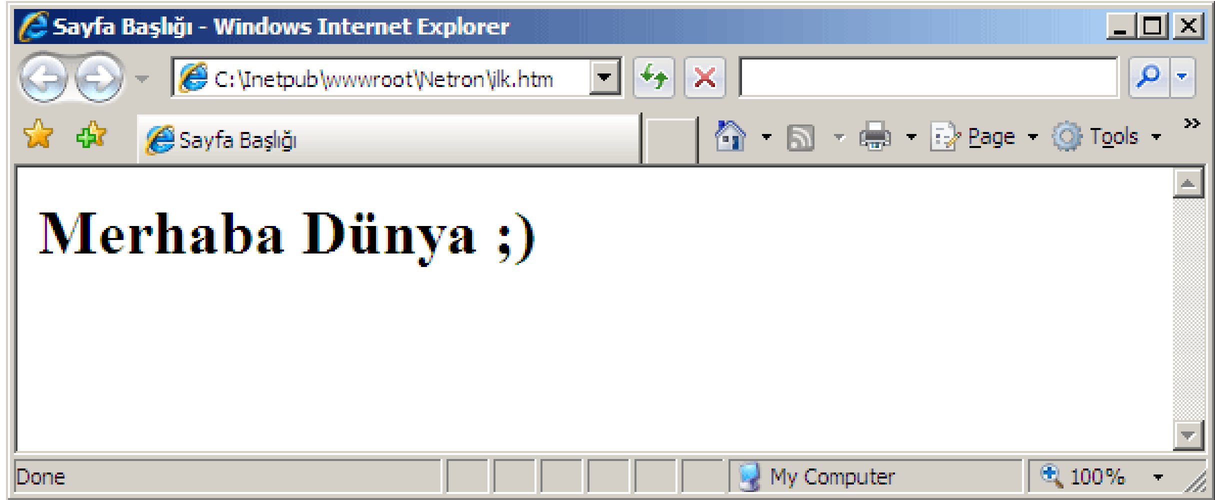
HTML isteği doğrudan cevapladığı için derleme yapılmaz ve tarayıcıda sadece HTML kodlarının çözümlenmesi (parse) yeterli olur.

- HTML hatalara karşı dirençsiz bir dildir. HTML'de yazılmış bir sayfa, tamamen hatalarla dolu olsa bile tarayıcı yorumlayabileceği kısmı görüntüler. Bu da sayfaların istenilmeyen bir şekilde görüntülenmesine sebep olabilir.
- HTML iç içe yer alan etiketlerden oluşur. Bir sayfanın düzgün görüntülenebilmesi için açılan her etiketin kapatılması ve etiketlerin aralarındaki hiyerarşiye uygun bir şekilde yazılması gereklidir. Bir etiketi başlatmak için **<etiket_adi>** ifadesi yazılır, o etiketi bitirmek için ise **</etiket_adi>** ifadesi yazılır. Böylece iki ifade arasındaki kısım o etiketin etki alanı içerisindedir. Etiketlerin bir çoğunda üst element - alt element (parent-child) ilişkisi bulunmaktadır ve alt elementlerin üst elementler içerisinde yer alması gerekmektedir.
- Bir HTML dosyası **<html>** etiketi ile başlayıp **</html>** etiketi ile kapatılmalıdır.

Aşağıda, HTML ile yazılmış basit bir web sayfası bulunmaktadır.

```
<html>
      <head>
            <title>Sayfa Başlığı</title>
      </head>
      <body>
            <h1>Merhaba Dünya ;)</h1>
      </body>
</html>
```

Yukarıdaki kodlar, Notepad gibi bir metin editöründe yazılarak, uzantısı **htm** veya **html** olacak şekilde kaydedilip, bir tarayıcıda çalıştırıldığında aşağıdaki gibi bir sonuç elde edilir. Buradaki **<h1>** elementi, paragraf içerisindeki metinin büyüklüğünü belirler.



Şekil 283: Oluşturulan HTML dosyasının tarayıcıdaki görüntüsü

Bir HTML dosyasının, standart olarak <html> etiketi ile başlayıp </html> etiketi ile bitmesi gerekir. <html> etiketinin içerisinde iki ana kısım bulunur. Bunlar **<head>** ve **<body>** kısımlarıdır. Bir web sayfası ile ilgili tanımlamalar, JavaScript ve CSS gibi HTML dışında yazılacak kod kısımları, <head> ve </head> ifadeleri arasında yer almalıdır. <body> ise bir web sayfasının gövde kısmını oluşturmakta ve web sayfasının bileşenleri (yazılar, resimler vs.) bu kısımda bulunmaktadır. İçerik istenilen şekilde düzenlendikten sonra </body> ifadesi ile daha önce açılmış olan <body> etiketi kapatılır.

```
<html>
```

```
    <head>
       Sayfa ile ilgili tanımlamalar, JavaScript ve CSS kodları...
    </head>
    <body>
       Sayfanın içeriğini oluşturacak kısımlar(yazılar, resimler, vb...)
    </body>
</html>
```

HTML Formları

Web sayfaları üzerinde ziyaretçilerle iletişim ve etkileşim formlar aracılığıyla yapılmaktadır. Örneğin, bir ziyaretçinin siteye üye olması için hazırlanan üyelik formu ile kullanıcıdan bazı bilgiler toplanabilir. Web sayfalarında sıklıkla karşılaşılan anketler, iletişim formları ve ziyaretçi defterleri gibi kısımlar formlar aracılığıyla yapılmaktadır. Bu tip formlar metin kutusu (textbox), açılır liste (dropdownlist), seçme kutusu (radiobutton-checkbox) ve buton (button) gibi bileşenlerden oluşabilir. Buradaki her bileşen aslında birer HTML form bileşenidir.

Şekil 284: Basit bir web formu

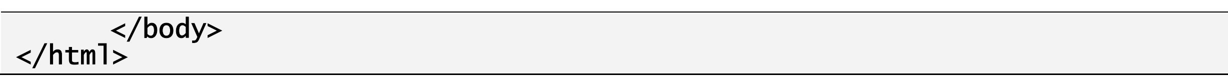
HTML içerisinde formlar <form> etiketi ile oluşturulur. Formlar içerisinde;

- **Metin kutusu (Textbox)**
- **Şifre kutusu (Password)**
- **Radyo butonları (Radiobutton)**
- **Seçme kutusu (Checkbox)**
- **Buton (Button)**
- **Açılır liste (DropdownList)**
- **Liste kutusu (Listbox)**

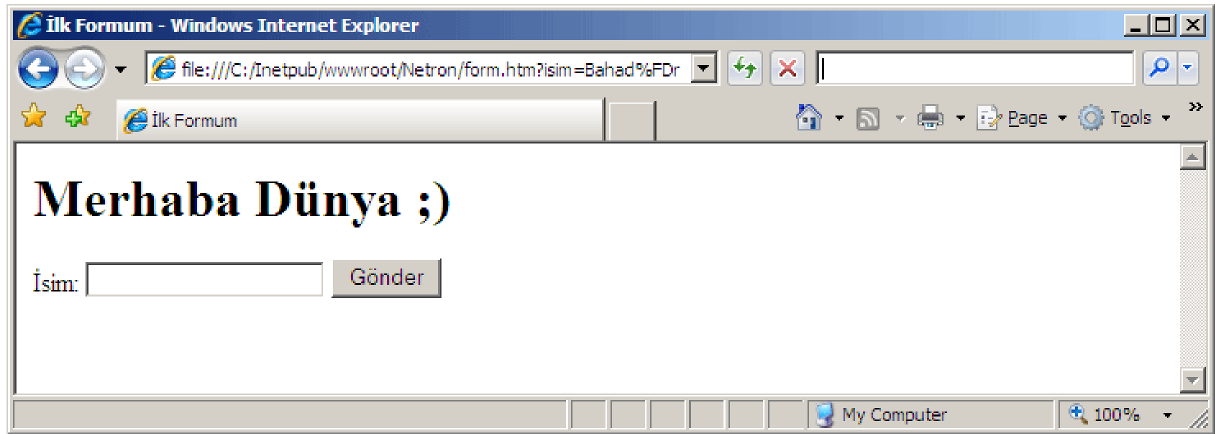
gibi bileşenler bulunabilir.

Aşağıda, şu ana kadar bahsedilen form elementleri ile ilgili bir örnek yer almaktadır.

```
<html>
      <head>
            <title>Sayfa Başlığı</title>
   </head>
      <body>
            <h1>Merhaba Dünya ;)</h1>
            <form>
                  <p>İsim:
                        <input type="text" name="isim" size="20">
                        <input type="submit" value="Gönder">
                  </p>
            </form>
```



```
      </body>
</html>
```



Şekil 285: Hazırlanan formun tarayıcıdaki görüntüsü

Yukarıdaki örnekte görüldüğü gibi, metin kutusu (textbox) ve buton (button) <form> etiketi içerisinde yer almaktadır. Sayfa bu durumdayken, metin kutusuna Bahadır yazıp **Gönder** butonuna tıklanırsa, adres çubuğunun aşağıdaki gibi değiştiği görülecektir.

file:///C:/Inetpub/wwwroot/Netron/form.htm?isim=Bahadir

Bu noktada, ortaya çıkan **URL** (Uniform Resource Locator) adresinin neden bu şekilde oluştuğu, az sonra detaylı şekilde ele alınacaktır. <form> tanımlaması yapılırken bazı önemli özelliklerine atamalar yapılması gerekmektedir.

Bunlardan ilki, form verilerinin hangi sayfaya yollanacağı bilgisidir. Bu özelliğin ismi **action**'dır. Action özelliği web sayfasında doldurulan formun işlenmek üzere hangi dosyaya ya da adrese yollanacağını belirtir. Buraya bir dosya adı yazılabileceği gibi adres olarak bir URL'de yazılabilir. Şimdilik bu özelliğe sayfanın kendi ismi verilsin.

Form etiketlerinin ikinci özelliği ise **method** özelliğidir. Bu özellik iki farklı değer alır, **POST** veya **GET**. POST değeri, form verilerinin kullanıcıya gösterilmeden iletilmesini sağlar. Formu doldurup yollayan kullanıcı hangi verilerin iletildiğini görmez. GET değeri ise, form verilerinin URL aracılığı ile yollanmasını sağlar. Bu şekilde yollanırken dosya isminden sonra "?" ile başlayan veriler iletilir ve kullanıcı bu verileri görebilir. Eğer gönderilmek istenilen verilerin kullanıcı tarafından görülmemesi gerekiyor ise POST, görülmesinde bir sakınca yok ise GET değeri atanabilir. Bu konuda diğer bir ayrım ise verinin büyüklüğü ile yapılmaktadır. GET metodu ile kısıtlı miktarda veri yollanabilirken, POST metodu ile teorik olarak sınırsız veri yollanabilmektedir. Eğer method özelliği belirlenmezse bu değer otomatik olarak GET olur. Yukarıda yapılan örnekte, formun method özelliğine herhangi bir değer atanmadığı için, GET değeri otomatik olarak seçilir ve veriler URL ile gönderilir. Örnekteki formun method özelliği POST olarak ayarlanırsa; farklı bir sonuç elde edilir. Aşağıdaki örnekte formun method özelliği POST, action özelliği ise sayfanın kendisi olarak belirlenmiştir.

```
<html>
      <head>
            <title>Sayfa Başlığı</title>
   </head>
      <body>
            <h1>Merhaba Dünya ;)</h1>
            <form action="form.htm" method="POST">
                  <p>İsim:
                        <input type="text" name="isim" size="20">
                        <input type="submit" value="Gönder">
                  </p>
            </form>
      </body>
```

```
</html>
```

Örnek çalıştırıldığında, metin kutusuna bir değer girerek butona tıklanırsa adres satırında herhangi bir değişiklik gerçekleşmez. Çünkü, form üzerindeki bilgiler POST metodu aracılığıyla gizli olarak taşınmıştır.

Form özelliğinin GET metodu istemciden sunucuya doğru sınırlı boyutta içerik gönderebilir. Bu sınır çoğu zaman tarayıcı tipine ve sürümüne göre değişiklik göstermektedir. Örneğin Internet Explorer tarayıcısında, dördüncü versiyondan itibaren bu sınır 2083 karakter iken, Opera tarayıcısında yaklaşık olarak 4050 karakterdir.

Web Sayfalarını Programlama

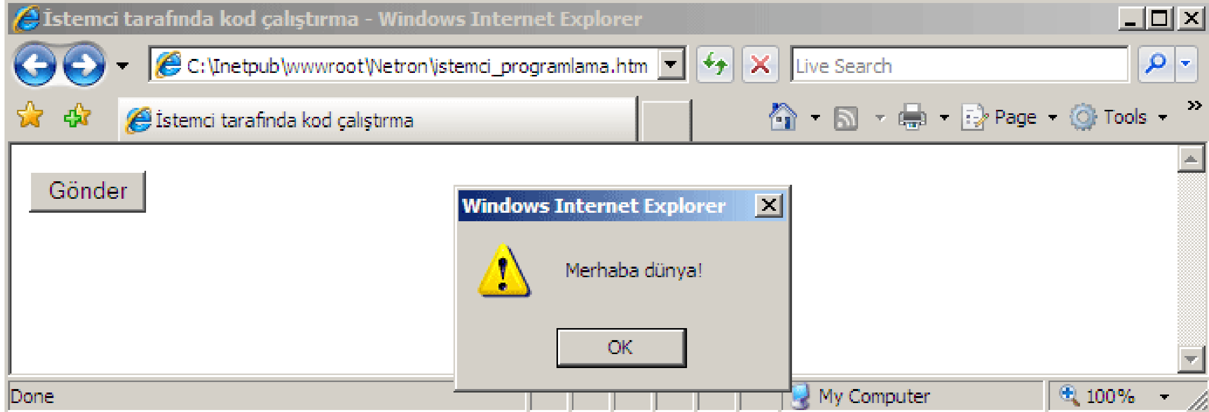
Web sayfalarının çalışma mantığı, masaüstü uygulamalara göre çok farklıdır. Masaüstü uygulamalar tek bir bilgisayar üzerinde çalışır. Çalışma esnasında oluşan çıktılar, ekran görüntüleri ve değişiklikler yine aynı bilgisayar üzerinde olmaktadır. Web sayfalarında ise çok farklı bir durum söz konusudur. Web sayfaları sunucu üzerinde tutulmakta ve çalıştırılmaktadır. Çalışma sonucunda HTML kodları üretilmekte ve gelen istekler sonucunda ziyaretçilerin bilgisayarında görüntülenmektedir. Böyle bir çalışma modelinde web programcısı olarak sunucu tarafında ve istemci tarafında bazı işlemler yapılabilir. Web uygulamaları geliştirme sürecinde, bazı diller ve teknolojiler kullanılır. Bu yapıların bazıları sadece istemci tarafında çalışabilirken, bazıları ise sadece sunucu tarafında çalışabilmektedir.

İstemci Taraflı Programlama (Client-Side Programming)

Bir web sayfası hazırlanırken bazı işlemlerin istemci tarafında yapılması sağlanabilir. Örneğin, ziyaretçinin bilgisayarındaki tarihi ve saati sayfaya yazdırmak, tarayıcı tipine göre işlem yapmak veya bir butona basıldığında ziyaretçiye bir uyarı mesajı göndermek gibi işlemler istemci tarafında yapılabileceklere birer örnektir. Bu tip işlemlerde yazılan kodlar istemci tarafında, yani tarayıcı üzerinde yorumlanmakta ve çalıştırılmaktadır. İstemci tarafında programlama yapılırken kullanılan temel dil JavaScript dilidir. HTML ve CSS ise yine istemci tarafında yorumlanırlar.

Aşağıdaki örnekte, önceki bölümde oluşturulan formdaki metin kutusuna yazılan ismin, butona basılınca bir mesaj kutusu olarak gösterilmesini sağlayan kodlar bulunmaktadır.

```
<html>
<head>
<script language="javascript" type="text/javascript">
   function tikladim() {
      alert('Merhaba dünya!');
   }
</script>
<title>İstemci tarafında kod çalıştırma</title>
<body>
      <input type="button" value="Gönder" onClick="tikladim()">
</body>
</html>
```



Şekil 286: İstemci tarafında çalıştırılan bir kod ile uyarı penceresi çıkarılması

Örnekteki kodlar bir dosyaya yazılıp çalıştırıldığında, metin kutusuna bir metin yazıp Gönder butonuna tıklanırsa ekran görüntüsündeki gibi bir mesaj kutusu görülür. Burada yazılan JavaScript kodları istemci tarafında çalıştırılarak üretilen çıktı kullanıcıya görüntülenmiştir. Eğer sayfanın boş bir yerinde fareye sağ tıklanıp **Kaynak Kodunu Göster (View Source)** seçilirse, yazılan JavaScript kodlarına erişilebilir.

```
istemci_programlama.htm - Notepad
File  Edit  Format  View  Help
<html>
<head>
<script language="javascript" type="text/javascript">
        function tikladim() {
                alert('Merhaba dünya!');
        }
</script>
<title>İstemci tarafında kod çalıştırma</title>
<body>
    <input type="button" value="Gönder" onClick="tikladim()">
</body>
</html>
```

Şekil 287: Sayfanın kaynak kodları içerisinde istemci tarafında çalışan JavaScript kodları görünmektedir

İstemci tarafında kod yazmanın en önemli dezavantajlarından biri yazılan kodu saklayamamaktır. Burada uyarı penceresi çıkarmak yerine daha önemli işlemler yapılabileceği düşünülürse, yazılacak kodların ziyaretçiler tarafından görüntülenmesi oldukça sakıncalı olabilir. Bazen bu kodların görüntülenebilmesi güvenlik açıklarına bile sebep olabilmektedir. Bu nedenle uygulamalarda önemli işlemlerin istemci tarafında yapılması tercih edilmemektedir.

Sunucu Taraflı Programlama (Server-Side Programming)

Sunucu taraflı programlama daha farklı bir yapı içerisinde çalışmaktadır. İstemci taraflı programlamada yazılan kodlar kullanıcıya gönderilerek, kullanıcının bilgisayarındaki tarayıcı üzerinde çalıştırılır ve sonuç yine tarayıcı üzerinde görüntülenir. Sunucu taraflı programlama da ise, yazılan kodlar kesinlikle istemciye gönderilmemekte ve sadece sunucu üzerinde tutulmaktadır. İstemciden sunucuya gelen isteğe göre kodlar sunucu üzerindeki ilgili programlar aracılığıyla çalıştırılır ve sonuç olarak üretilen HTML kodları ile sayfanın içerikleri istemciye gönderilir. İstemci tarafında sadece HTML kodları ve istemci taraflı script kodları görüntülenebilmektedir.

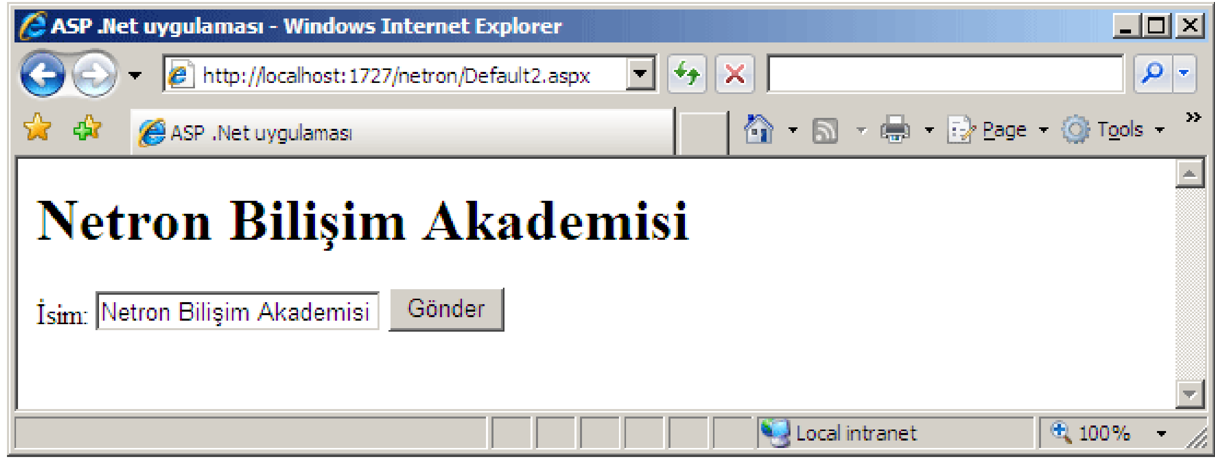
Sunucu tarafında PHP, ASP veya ASP.NET gibi bir teknoloji ile yazılmış olan kodlar, sunucu üzerindeki bazı yapılar tarafından çalıştırılır. Örneğin ASP ile geliştirilen kodlar sunucu tarafında bulunan **ASP.DLL** tarafından ele alınır ve HTML çıktı oluşturulur. Benzer olarak PHP de sunucu taraflı bir programlama dilidir ve o da sunucu üzerindeki PHP motoru tarafından çalıştırılır.

ASP.NET sunucu taraflı bir yazılım geliştirme teknolojisidir ve yazılan kodlar sunucu tarafında kullanılan .NET tabanlı dilin derleyicisi ile derlenip, ortak dil çalışma zamanı (CLR - Common Language Runtime) tarafından çalıştırılır. Elde edilen HTML çıktı istemciye gönderilir. ASP.NET uygulamalarının çalışma mekanizması sonraki bölümlerde anlatılacaktır. Aşağıda ASP.NET ile hazırlanan bir sayfa yer almaktadır.

Aşağıdaki ASP.NET örneğini çalıştırabilmek için, Visual Studio 2005 üzerinde bir web uygulamasının nasıl geliştirileceği bilinmelidir. Visual Studio 2005 üzerinde bir web uygulamasının nasıl geliştirileceği Bölüm-2'de detaylı bir şekilde anlatılacaktır.

```
<%@ Page Language="C#" %>
<html>
    <head>
        <title>ASP .NET uygulaması</title>
        <script runat="server">
            public void tikladim(object sender, EventArgs e)
            {
                lblYazi.Text = txtIsim.Text;
            }
        </script>
    </head>
    <body>
        <form id="Form1" runat="server">
            <h1><asp:Label id="lblYazi" runat="server"></asp:Label></h1>
            İsim: <asp:TextBox id="txtIsim" runat="server"></asp:TextBox>
            <asp:Button id="btnDugme" onclick="tikladim" runat="server"
text="Gönder" />
        </form>
    </body>
</html>
```

Örnek Visual Studio 2005 ortamında çalıştırıldığında txtIsim metin kutusuna (<asp:TextBox> ile belirtilen kısım) bir metin yazılıp Gönder butonuna tıklandığında, metin kutusu içerisine yazılan yazılar lblYazi isimli metin alanına (<asp:Label> ile belirtilen kısım) aktarılacaktır.



Şekil 288: İlk ASP .NET sayfası

Buradaki önemli hususlardan biri de sayfanın kaynak kodlarıdır. Sayfa üzerindeyken fareye sağ tıklayıp **Kaynak Kodunu Göster (View Source)** seçildiğinde aşağıdaki gibi bir sonuç elde edilir.

Default2[1] - Notepad

File Edit Format View Help

```
<html>
    <head>
        <title>ASP .Net uygulaması</title>

    </head>
    <body>
        <form name="Form1" method="post" action="Default2.aspx" id="Form1">
<div>
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE"
value="/wEPDwUKLTU5MjM1OTI1Ng9kFgICAg9kFgICAQ8PFgIeBFRleHQFGU5ldHJvbiBCaWxpxZ9pbSBB
a2FkZW1pc2lkZGRSA2tgGx1YpWi0YsKpZOzh8YjFaQ==" />
</div>

            <h1><span id="lblYazi">Netron Bilişim Akademisi</span></h1>
            İsim: <input name="txtIsim" type="text" value="Netron Bilişim
Akademisi" id="txtIsim" />
            <input type="submit" name="btnDugme" value="Gönder" id="btnDugme" />

<div>

        <input type="hidden" name="__EVENTVALIDATION" id="__EVENTVALIDATION"
value="/wEWAwLYhMzAAwKuzubvCQLogPvbCiFOU2V9vylMVzOyuhthjR1+Ou1c" />
</div></form>
    </body>
</html>
```

Şekil 289: Çalışmış olan ASP .NET sayfasının ürettiği kodlar

Görüldüğü gibi yazılan kodlar ile ortaya çıkan kodlar farklıdır. Ayrıca **<script runat=”server”></server>** etiketleri arasında yazılan C# kodları da hiçbir şekilde görünmemektedir.

HTTP (HyperText Transfer Protocol)

HTTP, web ortamında bilgi taşımak ve transfer etmek için kullanılan bir protokoldür. HTTP’nin asıl amacı, HTML sayfalarını yayınlama, sunucu ile istemci arasındaki dosya alışverişini sağlamak için bir yol sunmaktır. HTTP birçok web teknolojisinde olduğu gibi **World Wide Web Consortium (W3C)** tarafından standartlaştırılmaktadır ve bugün 1.1 versiyonu kullanılmaktadır.

HTTP, istemci ve sunucu arasındaki bir istek/cevap protokolüdür. Burada istemciden kasıt Internet Explorer, Firefox gibi tarayıcılar, sunucudan kasıt ise HTML, ASP, PHP, ASP.NET dosyalarının, resimlerin ve diğer dosyaların tutulduğu bir bilgisayar olarak düşünülebilir. İstemci uygulaması, **TCP (Transmission Control Protocol)** protokolünü kullanarak sunucunun bir portuna (genelde web için 80 numaralı porttur) ulaşır ve isteğini bu porta iletir. Sunucu ise bu portu devamlı dinleyerek kendisine gelen istekleri değerlendirir.

HTTP’nin avantajları olduğu kadar bazı dezavantajları da vardır. Örneğin bir kullanıcı internet sitesine girmek için sunucudan bir dosya isteğinde bulunur. Sunucudaki yazılım, dosyayı bulup gerekli çıktıyı istemciye geri yollar. İstemci ile sunucu arasındaki bağlantı bu noktada kopar ve sunucu istemcinin hala kendisine bağlı olup olmadığını tespit edemez. Dolayısıyla uygulamalar tasarlanırken bir ziyaretçinin o an içerisinde sitede olup olmadığı bilinemez.

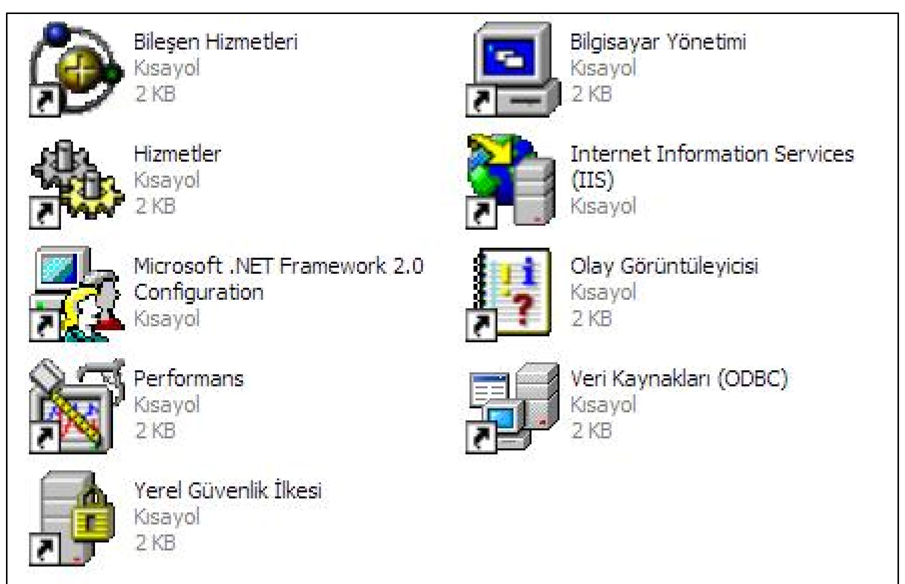
IIS (Internet Information Services)

Web sunucusu (web server), web uygulamalarının yönetildiği ve dosyaların tutulduğu sistemdir. **IIS**, Microsoft tarafından geliştirilen web sunucusu yazılımıdır. Dünyada en sık kullanılan sunucu yazılımlarından biri olan IIS, HTTP başlığında bahsedilen TCP portunu dinleyen bir yazılımdır. İçerisinde **FTP** (File Transfer Protocol), **SMTP** (Simple Mail Transfer Protocol), **NNTP** (Network News Transfer Protocol) ve **HTTP/HTTPS** (HTTP Secure) protokollerini içeren IIS, ilk olarak Windows NT 3.51 ile kullanıma sunulmuştur.

IIS'te Sanal Klasörler

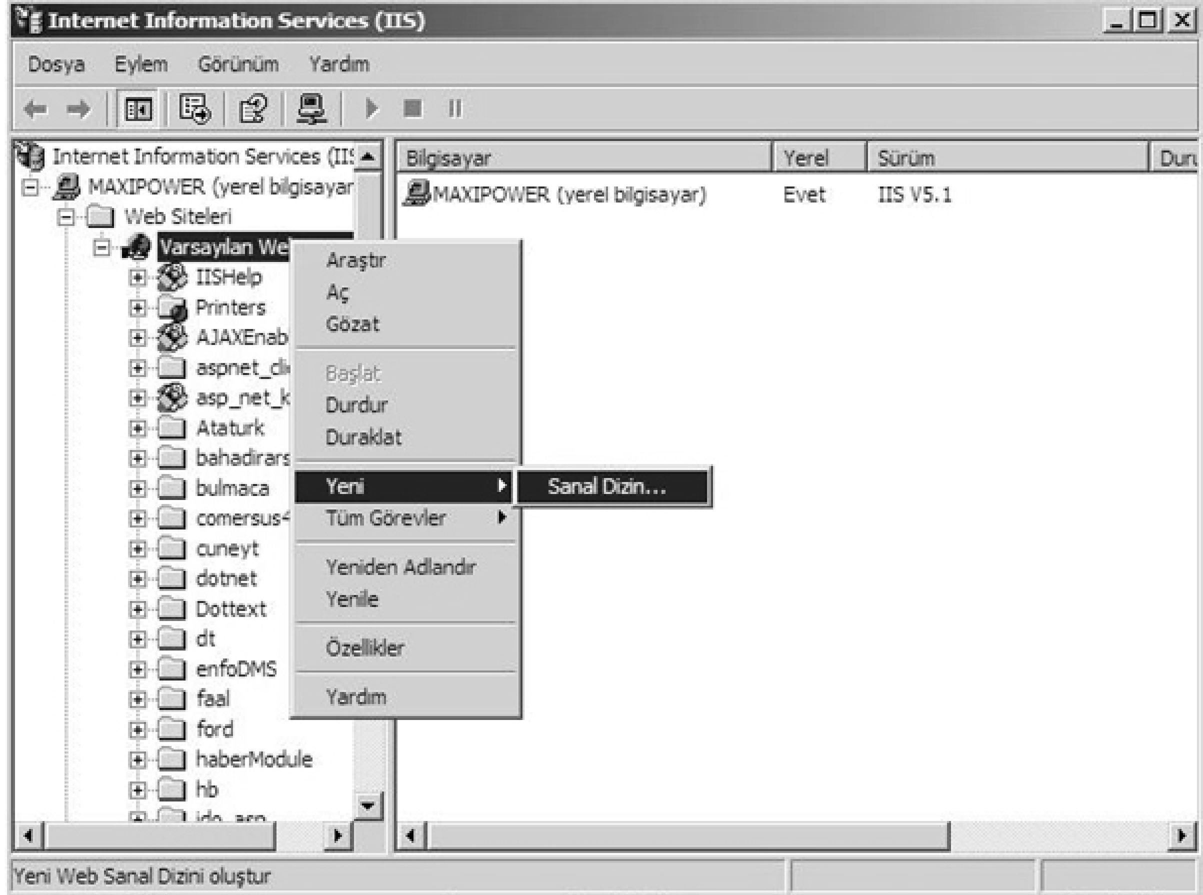
Sanal klasör IIS'e kaydedilen uygulama çalıştırma klasörleridir. ASP.NET ile oluşturulan siteler birer uygulama olduğu için, kendine ait bir sanal klasör içinde çalıştırılması gerekmektedir. Aşağıda IIS üzerinde sanal bir klasörün nasıl oluşturulacağı adım adım anlatılmıştır.

1. Sanal klasör oluşturmak için **Denetim Masası**'ndan, **Yönetimsel Araçlar** içindeki **Internet Information Services** kısmı kullanılır.



Şekil 290: Denetim Masası'ndan IIS kontrol edilebilir

2. Açılan yönetim panelinde, sol tarafta ağaç yapısında bulunan öğelerden **Varsayılan Web Sitesi**'ne gelene kadar artılara (+) tıklanarak alt öğelere geçilir. Son olarak, Varsayılan Web Sitesi'ne sağ tıklanarak **Yeni** seçeneğinden **Sanal Dizin** öğesi seçilir.

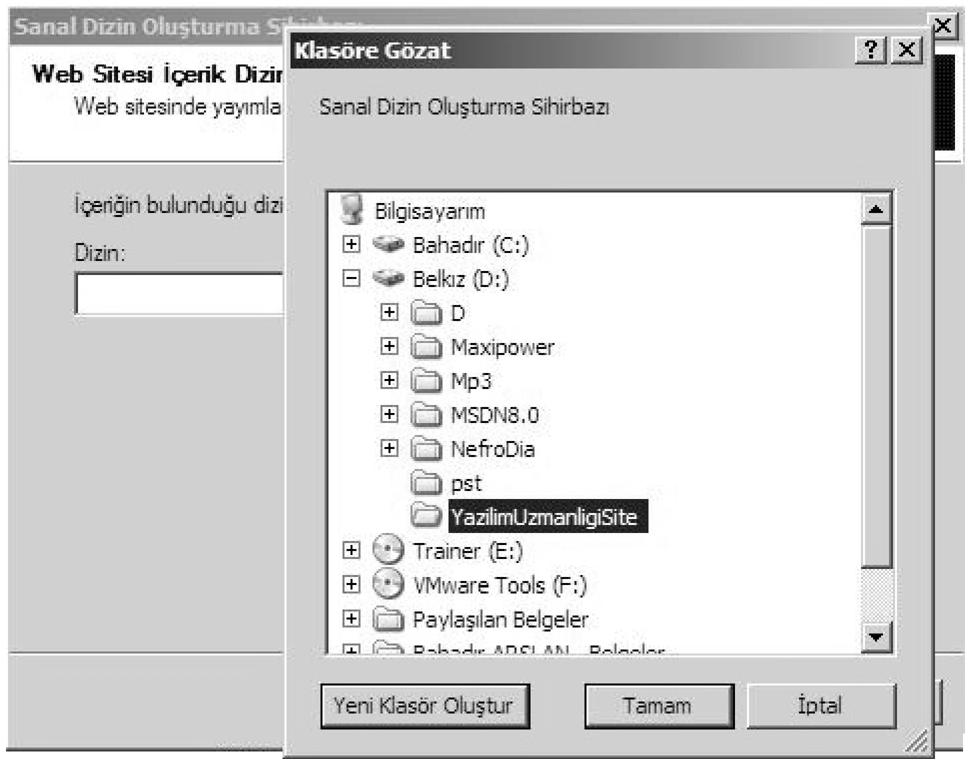


Şekil 291: IIS üzerinden yeni bir sanal dizin açılması

3. İlk pencere **İleri** seçilerek geçilir. Açılan pencereye oluşturulacak sanal dizinin adı girilir. İsim olarak uygulamayı temsil edecek ve **http://localhost/uygulama_adi** gibi bir adresten uygulamaya ulaşılmasını sağlayacak bir isim girilir. **İleri** seçeneği seçilerek devam edilir.

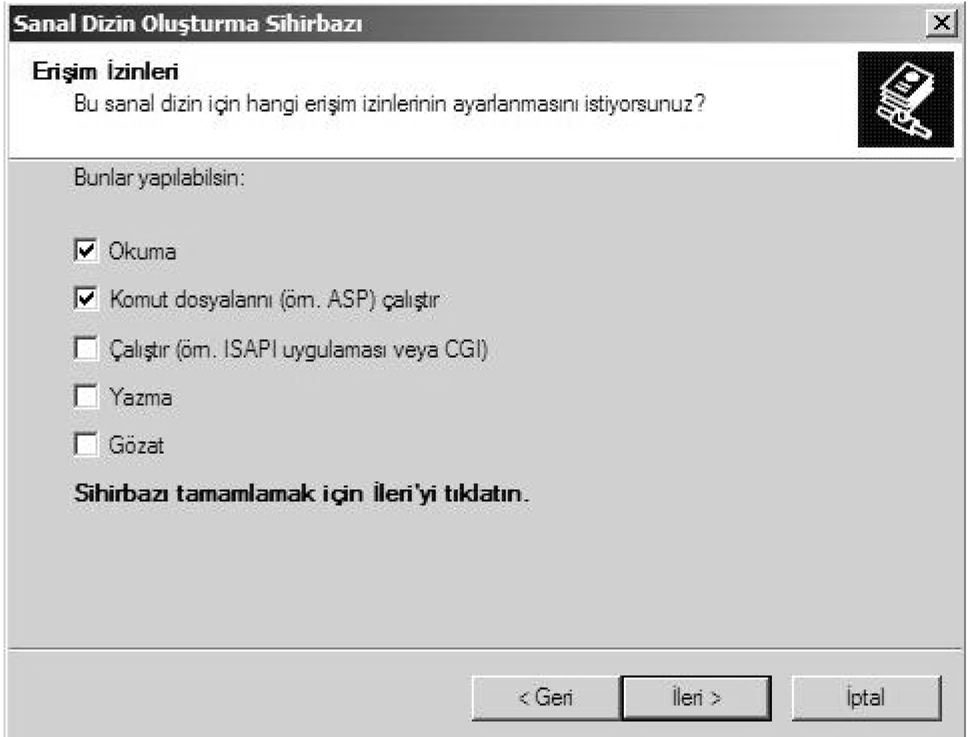
Şekil 292: Oluşturulacak sanal dizine isim verilmesi

4. Gelen ekranda sanal klasör olması istenilen dizin seçilir. Bu klasör IIS'in kök dizininde (varsayılan C:\Inetpub\wwwroot\) olabileceği gibi bilgisayarın herhangi bir yerindeki klasörde olabilir. **Gözat** kısmında bir klasör seçilerek **İleri** butonuna tıklanıp devam edilir.



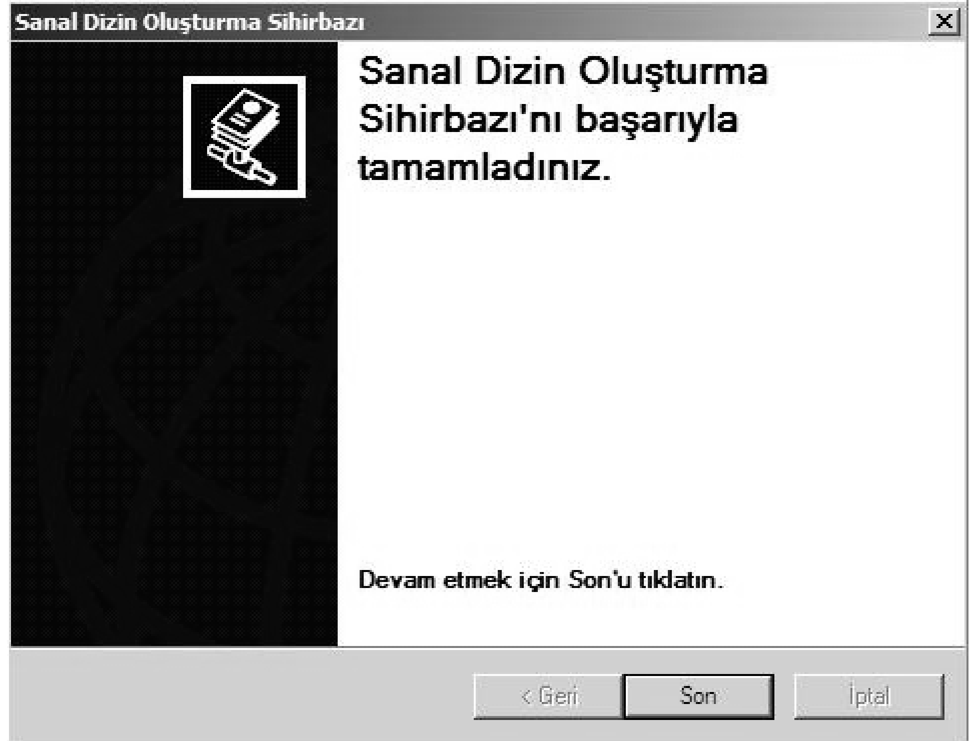
Şekil 293: Sanal dizin için bilgisayardan bir klasörün seçilmesi

5. Gelen ekrandan oluşturulan sanal klasör için haklar belirlenecektir. İlk iki seçenek seçili durumda gelmektedir. Bunların dışında eğer bu klasörde bir dosyaya veya veritabanına yazma işlemi yapılacaksa; **Yazma** seçeneği de seçilir.



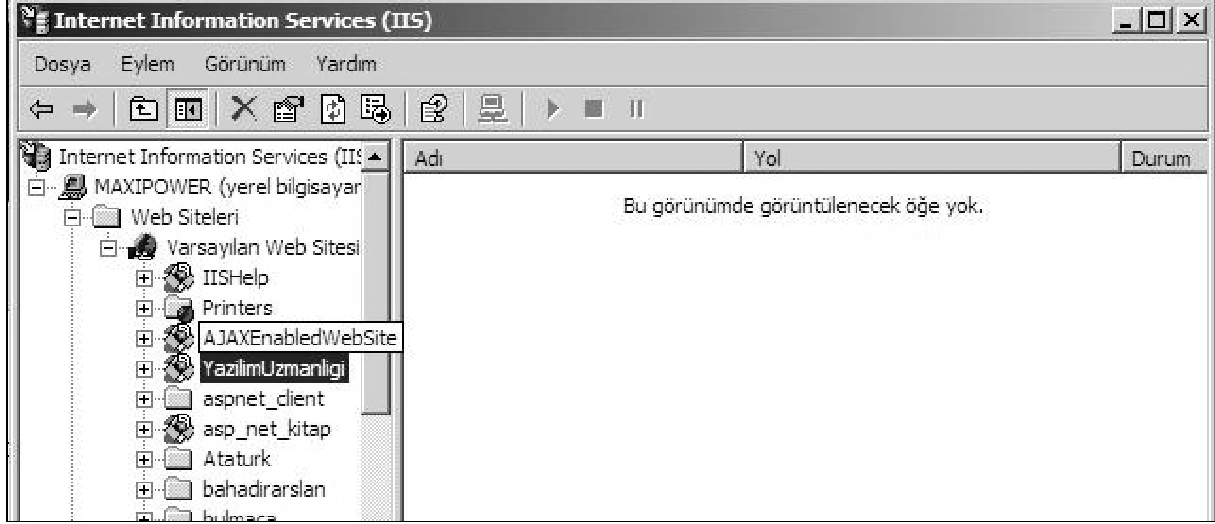
Şekil 294: Oluşturulan sanal dizin için hakların belirlenmesi

6. İleri butonuna tıklandığında yeni bir sanal dizin oluşturulacaktır. Herhangi bir sorun veya hata olmazsa; aşağıdaki gibi bir ekran ile karşılaşılır.



Şekil 295: Sanal dosya oluşturma başarılıyla tamamlandı ekranı

7. Sihirbaz penceresi kapandığında IIS'in sol tarafındaki ağaç yapısı içerisinde oluşan yeni sanal klasör görüntülenir.



Şekil 296: Oluşturulan sanal klasör IIS içerisinde görüntülenebilir

Yukarıdaki adımlar tamamlandığında yeni bir sanal klasör kullanıma hazır hale gelmiş olur.

BÖLÜM 1: MERHABA ASP.NET

ASP.NET Mimarisi ve Temelleri

ASP.NET Çalışma Modeli

ASP.NET'in çalışma mantığı ASP ve PHP gibi betik (script) tabanlı programlama ortamlarına göre oldukça farklıdır. Bu fark da ASP.NET'e çalışma zamanında hissedilir düzeyde performans artışı sağlar.

Basit düzeyde ASP.NET çalışma mekanizması şu şekildedir: Öncelikle kullanıcı, bir internet tarayıcısı yardımıyla adres penceresine istediği web sayfasının adresini yazıp, ilk talebini (request) yaptığında, HTTP protokolünün GET metodu ile ilgili web sunucusuna ulaşır. Kullanıcıdan sunucuya talebin GET metodu ile gitmesinin anlamı, talebin url'den gönderilmesidir. Web sunucusuna ulaşan talep IIS tarafından karşılanır. IIS'in görevi, gelen talebin hangi tipte sayfaya yapıldığına bağlı olarak kontrolü, o uzantılı sayfayı ele alacak olan yazılıma devretmektir. Örneğin, buradaki talebin anasayfa.aspx gibi aspx uzantılı bir ASP.NET sayfasına yapıldığı varsayılırsa, IIS kendi listesinde varolan bilgilerden faydalanarak kontrolün, talebi değerlendirecek olan .NET Framework'e devredilmesini sağlar. Kontrol, .NET Framework'e geçtiğinde, talep ortak dil çalışma zamanına (Common Language Runtime) gönderilir. Burada anasayfa.aspx'e ait sınıf nesne örneği ile birlikte sayfanın üzerinde yer alan bütün kontroller oluşturulur. Bu esnada, başka kullanıcılardan da aynı sayfa için gelecek sonraki taleplere daha hızlı cevap verebilmek için sayfaya ait sınıf, bir .dll dosyası olarak sunucunun belleğinde Assembly Cache adında bir yere atılır ve buradan kullanılır. Kullanıcıya geri dönecek cevabın (response) HTML olarak gönderilmesi gerekmektedir. Sınıfın çalışması sona erdiğinde, HTTP protokolünün çalışma zamanı tarafından, anasayfa.aspx'in HTML çıktısı oluşturulur (HTML render işlemi). Sınıfla ilgili işlemler sona erdiğinde, ortak dil çalışma zamanı (CLR) tarafından sınıf nesne örneği bellekten düşürülür ve yine IIS üzerinden kullanıcıya HTML ve varsa istemci taraflı script kodları gönderilir.

ASP.NET ile hazırlanmış bir web sayfasının arkasında .NET'i destekleyen herhangi bir dilde (C#, VB.NET gibi) yazılmış bir sınıf yer alır. ASP.NET çalışma modelinde bir web sayfasına yapılan her talepte, o sayfaya ait sınıfın nesne örneği oluşturulur ve kullanılır. Bu nedenle, web uygulamalarının doğası gereği sayfalar arası bilgi taşıma için daha fazla çaba sarfetmek gerekir.

Web Sitesi Oluşturma Yolları

Visual Studio 2005 platformunda, bir web sitesi oluşturmak için üç farklı yöntem bulunmaktadır.

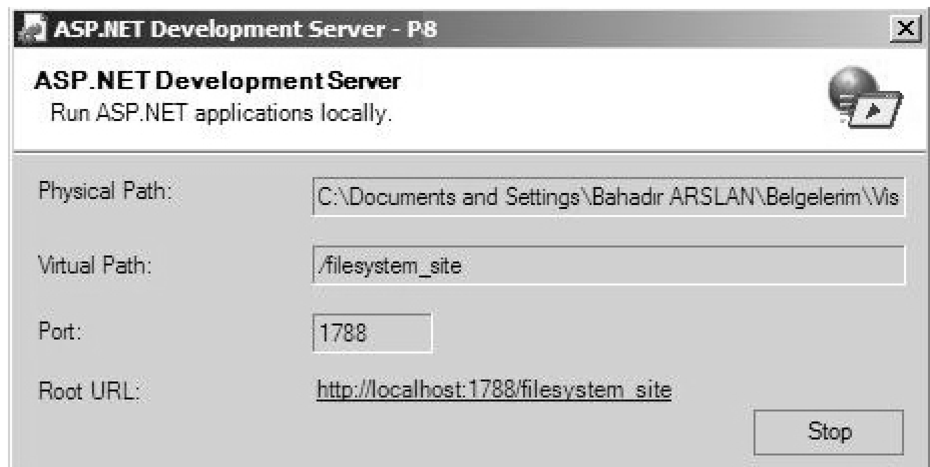
- File System
- HTTP
- FTP

File System

File System, ASP.NET 2.0 ile gelen yeni bir yöntemdir. ASP.NET 1.x sürümlerinde, bir uygulamanın geliştirilebilmesi için IIS'e ihtiyaç duyuluyordu. Ayrıca Windows XP Home edition kullanıcıları gibi bilgisayarına IIS kurulamayacak olan veya bilgisayarına IIS

kurmak istemeyen kullanıcıların ASP.NET sayfaları geliştirmelerine engel teşkil ediyordu. Bu nedenle Microsoft, Visual Studio 2005 içerisine gömülü (embeded) bir web sunucusu yerleştirdi ve ASP.NET ile hazırlanan web sayfalarının IIS olmadan da geliştirilebilmelerini sağladı.

Eğer File > New Web Site menüsünden File System seçeneği seçilirse, belirlenen bir klasör içerisinde yeni bir web sitesi projesi oluşturulacaktır. Proje çalıştırıldığında ise Visual Studio 2005'in **kendi web sunucusu** devreye girecektir. Bu web sunucusu sadece yerel makinadan (local) gelen istekleri cevapladığı için güvenlik açığı oluşturmamakla birlikte, IIS'e oranla daha performanslı çalışmaktadır; çünkü bu sunucu sadece yazılım geliştirme amaçlı bir kullanım için geliştirilmiştir ve üzerinde ayar yapılamamaktadır.



Şekil 297: Visual Studio 2005'in yapısında kendi web sunucusu bulunmaktadır

File System ile site oluşturma yöntemi özellikle proje başka bilgisayarlarda da çalıştırılacaksa çok kullanışlıdır; çünkü çalıştırılacağı makinada tekrar ayar yapmaya gerek kalmayacaktır. Bu yöntemin dezavantajı ise uygulamayı çalıştırabilmek için Visual Studio 2005'in gerekli olmasıdır.

File System ile oluşturulan siteler gerekli durumlarda IIS üzerinde açılacak sanal bir dizine taşınabilir ve buradan da çalıştırılabilir.

HTTP

HTTP, önceki ASP.NET sürümlerinde de kullanılan klasik site oluşturma yöntemidir. Yeni bir web sitesi oluşturulurken bu yöntem kullanılırsa, Visual Studio 2005 verilen isme uygun olarak IIS üzerinde bir sanal dizin (virtual directory) oluşturur ve sitenin IIS üzerinden çalışmasını sağlar. (Sanal dizin konusu ile ilgili daha detaylı bilgiler 1. bölümde anlatılmıştır.) Bu yöntem ile çalışırken web sitesine ulaşmak için Visual Studio 2005'in açık olma şartı yoktur, ancak IIS hizmetlerinin başlatılmış olması gereklidir.

Uygulama geliştirmek için bir de **Remote Site** seçeneği vardır. Bu seçenek kullanılarak **HTTP** ile uzak bir site üzerinden çalışmak mümkün olmaktadır. Ancak uzak makinede **FrontPage Server Extension**'larının kurulu olması gerekmektedir.

FTP

Bu yöntemde site belirtilen bir FTP (File Transfer Protocol) adresi üzerinde oluşturulacaktır. Bu yöntem ile çalışırken uygulama dosyaları sunucu üzerinde oluşturulduğu için, dosyaları yeniden ftp üzerinden yüklemeye gerek yoktur. Bu yöntemin dezavantajı ise internet bağlantısına ihtiyaç duyulmasıdır. Ayrıca yapılan değişiklikler site

üzerinde ziyaretçiler tarafından anında görüntülenebileceği için, uygulama geliştirme sürecinde daha dikkatli çalışmak gerekir.

ASP.NET Dosya Tipleri

ASP veya PHP gibi programlama ortamlarında web sayfaları geliştirenler için ASP.NET'teki önemli değişikliklerden biri de farklı dosya tipleridir. ASP.NET'te kullanıcıların görüntüleyebileceği temel dosya tipleri aspx olmakla birlikte, yine uygulamalar içerisinde sıklıkla kullanılan başka dosya tipleri de bulunmaktadır. ASP.NET uygulamalarında kullanılan dosya tipleri ve işlevleri aşağıda detaylı bir şekilde açıklanmıştır.

- **aspx**: Temel işlemlerin yapılacağı ve kullanıcıların görüntüleyebileceği dosyalardır. Metin tabanlı bir dosya biçimi olup web formlarının oluşturulmasında kullanılır. aspx dosyaları, HTML kodları ile sunucu kontrolleri ve kullanıcı kontrollerini içerir. HTML kısmında sayfa içerisindeki içeriklerin nerede ve nasıl bulunacağı belirlenir. HTML kısımları içerisinde normal HTML etiketleri dışında ASP.NET sunucu ve kullanıcı kontrolleri de kullanılabilir. aspx sayfalarında <script> kısımları içerisinde C#, VB.NET veya .NET ortamındaki farklı bir dil ile kodlar yazılabileceği gibi, bu kod kısımları farklı dosyalarda da tutulabilir.

.aspx dosyaları ile birlikte çalışacak olan C#, VB.NET vb. kodların aspx dosyasının içerisinde tutulmasına **in-line kodlama yöntemi**, farklı dosyalar içerisinde tutulmasına ise **code-behind kodlama yöntemi** denir. Code-behind kodlama yönteminde, yazılan C# kodları **aspx.cs**, VB.NET kodları ise **aspx.vb** uzantılı farklı dosyalarda tutulur.

- **ascx**: aspx sayfaları içerisinde kullanılabilecek kullanıcı kontrolü (user control) dosyalarıdır. Yapısı bir aspx dosyası ile hemen hemen aynıdır. Aspx dosyalarında bulunan <html>, <head> ve <body> gibi HTML elementleri ascx dosyalarında yer almamalıdır. Bu dosyalar tek başlarına çalıştırılamazlar, sadece bir aspx dosyasının içinde bir element olarak kullanılabilirler.
- **asax**: Uygulamanın çalıştığı süre boyunca bazı olayları yakalamak için kullanılan dosya türüdür. Yakalanan olaya göre işlemler yapılmasını sağlar. Uygulamalarda varsayılan olarak Global.asax dosyası kullanılır.
- **master**: ASP.NET 2.0 ile gelen ve aspx sayfaları için temel şablon olarak kullanılan dosyalardır. MasterPage adı verilen bu sayfaların genel yapısı aspx sayfaları ile hemen hemen aynıdır.
- **sitemap**: ASP.NET 2.0 ile gelen ve menü kontrollerinin çalışması için gerekli bilgileri içeren XML tabanlı bir dosyadır.
- **skin**: ASP.NET 2.0 ile gelen ve sunucu kontrollerinin görünümlerinin ayarlanabileceği stil dosyalarıdır.
- **config**: Projenin kullanacağı genel ayarları içeren XML tabanlı bir dosyadır.
- **dll**: Derlenmiş kod kütüphanesi dosyalarıdır. Uygulama içerisinde **Bin** klasöründe bulunur.
- **asmx**: Web servisleri dosyalarıdır. Web servisleri farklı platformlardaki, farklı uygulamaların kullanabildiği servislerdir. aspx sayfalarında olduğu gibi asmx dosyalarının da arka tarafında C# ve VB .NET gibi dillerle yazılmış kodlar yer almaktadır.

ASP.NET Klasör Tipleri

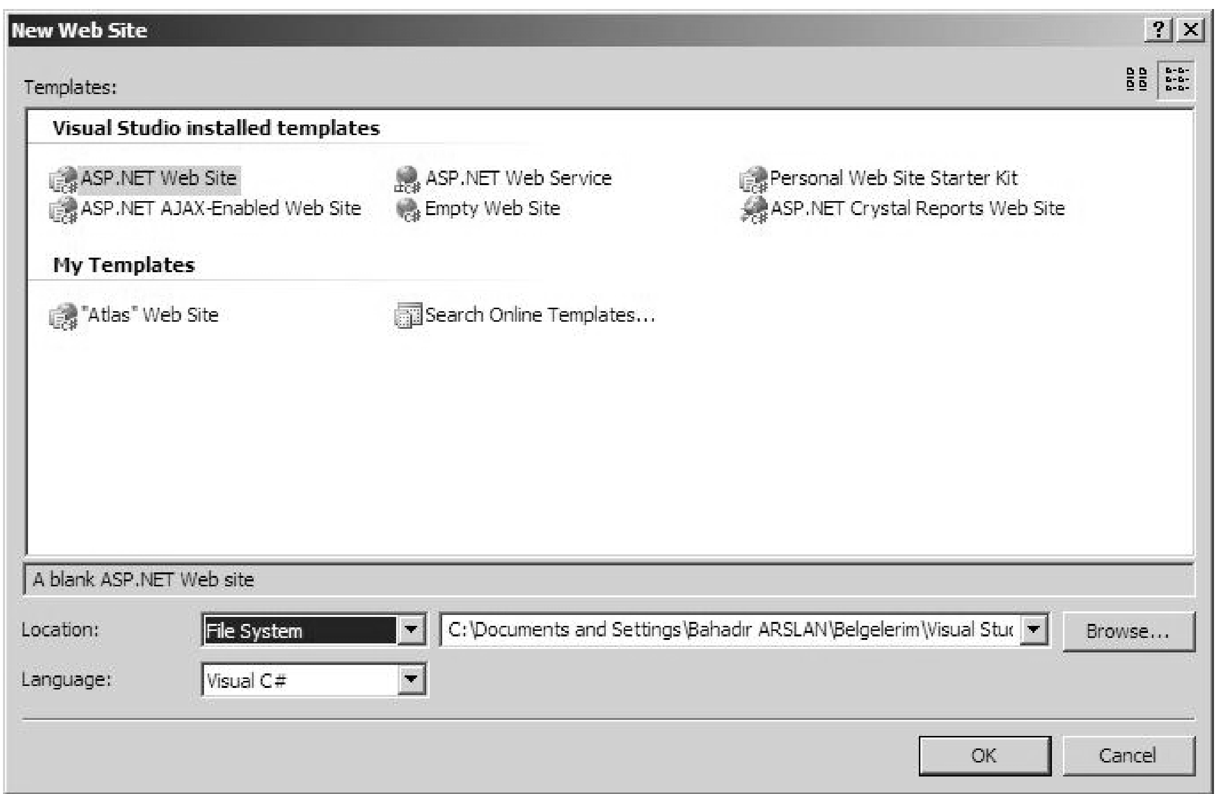
ASP.NET uygulamalarında özel amaçları olan ve belirli tipteki dosyaları içerebilen klasörler bulunmaktadır. ASP.NET'in 1.0 ve 1.1 versiyonlarında özel klasör olarak **sadece**

<u>Bin</u> klasörü varken, 2.0 versiyonu ile beraber özel klasörlerin sayısı sekize çıkmıştır. Bu klasörlerden bazıları şunlardır:

- **Bin**: İçerisinde web uygulamasında kullanılacak derlenmiş .NET assembly dosyalarını (genellikle dll dosyaları) bulundurmaktadır. Bir web sayfası çalışırken, sunucu ilk olarak bu klasör içerisindeki kod kütüphanelerine bakarak sayfaları arayacaktır.
- **App_Code**: Bin klasörüne benzer bir görev üstlenmekle beraber, derlenmemiş kod dosyalarını içerisinde tutar. Bu klasörde **.cs**, **.vb** gibi sınıf ve kod dosyaları, **.wsdl** veya **.xsd** türünde dosyalar bulunabilir. Çalışma zamanında bu klasördeki kodlar derlenir ve çalıştırılır.
- **App_Themes**: Web uygulaması içerisinde tanımlanan ve sayfalardaki elementler tarafından kullanılan temaları saklar. Bu temaların içerisinde de **.skin** ve **.css** dosyaları bulunur.
- **App_Data**: Uygulama içerisinde kullanılabilecek olan **SQL Server veritabanları, Microsoft Access** ve **XML** gibi veri ile ilgili dosyaları içerir.
- **App_LocalResources**: Sayfa bazında kaynak (resource) tanımlanırken kullanılan dosyaları saklar. Genellikle yerelleştirme (localization) işlemlerinin yapılacağı kaynak dosyalarını saklamak için kullanılır.
- **App_GlobalResources**: Bu dizin ise bir web uygulamasındaki her sayfa tarafından erişilebilen global kaynakların saklanması için kullanılır.

Visual Studio 2005'te Web Projesi Oluşturma

Visual Studio 2005'te yeni bir web sitesi projesi oluşturmak için **File** menüsündeki **New** sekmesinden **Web Site** seçilir.



Şekil 298: Visual Studio 2005'te yeni bir web sitesi projesinin oluşturulması

Şekil 298'deki ekranda bir web sitesi oluşturmak için kullanılabilecek farklı şablonlar bulunmaktadır. Normal bir web uygulaması oluşturmak için bu kısımdan **ASP.NET Web Site** seçeneği seçilir. Bu ekranda bulunan **Location** kısmındaki seçenekler, bir önceki bölümde **Web Sitesi Oluşturma Yolları** başlığında detaylı bir şekilde anlatılmıştır.

Language kısmından, web sitesinin hangi dil kullanılarak hazırlanacağı seçilir. Bu kısımda varsayılan olarak Visual C#, Visual Basic ve Visual J# seçenekleri bulunmaktadır. Location kısmından **File System**'i, language kısmından **Visual C#**'ı seçip, projenin hangi klasörde yer alacağını belirtip, **OK** butonuna tıklandığında yeni bir web sitesi oluşturulacaktır.

Oluşturulan web sitesi içerisinde **Default.aspx** ve bu dosya ile birlikte çalışacak olan **Default.aspx.cs** dosyası bulunmaktadır. aspx dosyası **Design** (Tasarım) ve **Source** (HTML Kodları) olmak üzere iki kısımdan oluşur. Bu iki kısımda da aslında aynı işler yapılmaktadır. Design kısmında web sayfası görsel öğeler kullanılarak hazırlanırken, Source kısmında web sayfası HTML kodları yazılarak hazırlanır. Design kısmından görsel öğeler kullanarak web sayfası hazırlandığında, Visual Studio 2005 source kısmında gerekli HTML kodlarını kendisi üretmektedir. Şekil 299'da Default.aspx sayfasının Source kısmı ve projedeki dosyalar görülmektedir.

```
1  <%@ Page Language="C#" AutoEventWireup="true"  CodeFile="Default.
2
3  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "
4
5  <html xmlns="http://www.w3.org/1999/xhtml" >
6  <head runat="server">
7      <title>Untitled Page</title>
8  </head>
9  <body>
10     <form id="form1" runat="server">
11     <div>
12
13     </div>
14     </form>
15 </body>
16 </html>
17
```

Şekil 299: Yeni açılmış site'deki Default.aspx dosyasının Source kısmı ve projedeki dosyalar

İlk ASP.Net Web Sayfası: Merhaba ASP.Net

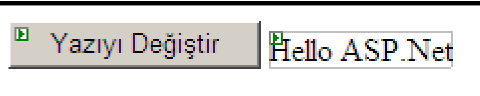
Design kısmında Toolbox üzerinden sayfaya kontroller eklenip, bu kontrollerin özellikleri Properties penceresinden değiştirilebilir. Benzer şekilde sayfanın Source kısmından HTML kodları değiştirebilir, kontroller eklenebilir ve kontrollerin özellikleri değiştirebilir. Örneğin, bir sayfaya **btnTikla** isimli bir Button ve **lblYazi** isimli bir Label kontrolü ekleyip, btnTikla butonuna tıklandığında lblYazi içerisindeki "Hello ASP.Net" yazısının "Merhaba ASP.Net" olarak nasıl değiştirilebileceği adım adım şu şekilde incelenebilir.

1. Açılan yeni bir web sitesi projesi içerisindeki Default.aspx sayfasının Source kısmından <body> kısmındaki **<form id="form1" runat="server">** ile başlayan form alanı içerisine bir Button ve bir Label kontrolü eklensin.

```
<form id="form1" runat="server">
```

```
    <asp:Button ID="btnTikla" runat="server" Text="Yazıyı Değiştir" />
    <asp:Label ID="lblYazi" runat="server" Text="Hello
    ASP.Net"></asp:Label>
</form>
```

Yukarıdaki kodların sayfanın Source kısmına eklenmesinin ardından, Design kısmında sayfanın görünümü kontrol edilirse aşağıdaki gibi bir görüntü elde edilir.



Şekil 300: Sayfadaki Button ve Label nesnelerinin Design kısmında görünümü

ASP.NET sayfalarının doğru bir şekilde çalışabilmesi için, ASP.NET kontrollerinin **<form id="form1" runat="server">** ile başlayan form alanı içerisinde yazılması gerekmektedir. Burada **id** kısmındaki form ismi değişebilir, ancak form etiketi içerisinde mutlaka **runat="server"** ifadesi yer almalıdır.

2. Butona tıklandığında etiketin içeriğini değiştirebilmek için öncelikle butonun tıklanma olay (click event) metoduna geçilmeli ve yapılması istenilen işlemler bu kısımda gerçekleştirilmelidir (Olay (event) kavramına ilerleyen konularda daha detaylı şekilde değinilecektir). Butonun tıklanma olayını ele alabilmek için, Design kısmında iken butonun üzerine çift tıklanır. Visual Studio 2005 arayüzü, otomatik olarak sayfanın kaynak kodlarının bulunduğu **Default.aspx.cs** sayfasına giderek, bu kısımda butona bağlanacak olan bir metot oluşturur. HTML kısmında ise btnTikla isimli butona bu metot bağlanır.

```
Default.aspx.cs*   Start Page   Default.aspx*
_Default                                   btnTikla_Click(obj
 4  using System.Web;
 5  using System.Web.Security;
 6  using System.Web.UI;
 7  using System.Web.UI.WebControls;
 8  using System.Web.UI.WebControls.WebParts;
 9  using System.Web.UI.HtmlControls;
10
11  public partial class _Default : System.Web.UI.Page
12  {
13      protected void Page_Load(object sender, EventArgs e)
14      {
15
16      }
17      protected void btnTikla_Click(object sender, EventArgs e)
18      {
19
20      }
21  }
```

Şekil 301: Default.aspx.cs dosyası içerisinde butonun click olayı için bir metot yazıldı

Şekil 301'de görünen kısımda btnTikla_Click isimli metodun içerisine **lblYazi** nesnesinin içerisindeki metni değiştirmek için gerekli olan kodların yazılması gerekir. Bunun için bu kısıma aşağıda kalın puntolarla yazılmış kodlar eklenmelidir. "lblYazi" yazıldığında açılan intelli-sense penceresinden Text özelliği seçilmelidir.

```
protected void btnTikla_Click(object sender, EventArgs e)
```

```
{
    lblYazi.Text = "Merhaba ASP.Net";
}
```

C# büyük-küçük harf duyarlı bir dil olduğu için, kod yazarken genellikle açılan intelli-sense penceresinin sunduğu seçeneklerin kullanılması tavsiye edilmektedir. Böylece hatalı kod yazma ihtimali en aza inecektir.

Bu şekilde gerekli eklemeler ve kodlamalar yapılarak sayfanın son hali hazırlanmış olur. Web sayfasının çalıştırılması için **Debug** menüsünden **Start Debugging** seçeneği seçilebilir veya **F5** kısayol tuşu kullanılabilir. İlk çalıştırma esnasında Visual Studio 2005 web sayfasını hata ayıklama modunda (debugging mode) açmak için onay isteyecektir. Penceredeki ilk şık olan **Modify the web.config file to enable debugging** seçeneği seçilerek OK butonuna tıklayıp sayfa çalıştırılabilir. Bu şekilde web.config dosyasında gerekli değişiklikler yapılarak ilerleyen çalışmalarda hata ayıklama işlemlerinin yapılması sağlanabilir. Sayfa çalıştıktan sonra butona tıklanarak Label üzerindeki değişiklik gözlemlenebilir.

Şekil 302: Butona tıklandığında sayfada oluşan değişiklik

Bu şekilde ilk web uygulaması gerçekleştirilmiş oldu. Web sayfalarına kod ilave etmek iki farklı amaç için yapılmaktadır. İlki; sayfaya kontrolleri eklemek, kontrollerin özelliklerini değiştirmek ve görününüm ayarlarını değiştirmek gibi işlemler için, diğeri ise sayfanın ve kontrollerin fonksiyonelliklerini sağlamak içindir.

Kaynak kodu dosyalarının (aspx.cs uzantılı dosyalar) görünümü sadece C# ile yazılmış kodlardan oluşmaktadır. Bir web sayfası üzerinde boş bir alana sağ tıklayıp veya Solution Explorer'da ilgili dosyanın üzerinde sağ tıklayıp View Code seçeneği seçilirse; o dosyanın kaynak kodlarına geçilir. Aynı şekilde kaynak kodları görünümünde iken sağ tıklayıp View Designer seçilirse de tasarım kısmına dönülür.

Web Sayfalarında Event (Olay) Kullanımı

.Net tabanlı dillerde, olay kavramı uygulamaların önemli temellerinden birini oluşturmaktadır. Bu durum, hem web uygulamaları, hem de Windows uygulamaları için geçerlidir.

Olaylar (events) uygulamanın çalışması sırasında bir düğmenin tıklanması, bir formun açılması ya da bir seçeneğin değiştirilmesi gibi eylemlerden sonra istenilen bir metodun çalışmasını sağlayan sınıf (class) üyeleridir. ".Net'te her şey bir sınıftır" sözüne uygun olarak sayfaya bir buton sürüklenip bırakıldığında, aslında .NET Framework uygulama geliştiricileri tarafından yazılmış olan **Button** sınıfından bir nesne örneği oluşturulmuş olur. Uygulamayı geliştiren, bu nesne örneğinin görsel kısmını görür. Ayrıca ID, Text gibi değiştirilebilen ve elde edilebilen birçok özellik, yine bu sınıfın üyeleridir. Sayfadaki bir butona çift tıklayarak yazılan metodun, çalışma zamanında kullanıcı tarafından tıklandığında çalışmasını garanti eden ise Click olay (event)'ıdır. Butona çift tıklandığı anda ilgili butonun html koduna **onclick = "MetotAdi"** şeklinde bir atama yapılır.

Bir olay metodunun genel yapısı aşağıdaki gibidir.

```
public void Olay_Adi (object sender, EventArgs e)
{

}
```

Bu özel metodun iki tane parametresi vardır. Bunlardan ilki **object** türündeki **sender** parametresidir. sender, olayın (event), üyesi olduğu nesnedir. Bir başka deyişle olayın tetiklenmesine neden olan kontroldür. Bu olaya göre sender, bazen bir Button, bazen bir LinkButton ya da bir GridView olabilir. İkinci parametre ise; **EventArgs** türündeki **e** parametresidir. Bu parametre ise o olay ile ilgili oluşan bazı önemli bilgileri içerir. Bu, olaya göre bazen sorgudan etkilenen kayıtların sayısı olabileceği gibi, bazen de yapılan işlemin durdurulmasını sağlayan bir boolean değer olabilir.

Visual Studio 2005'de Designer ekranında, bir kontrole çift tıklayarak o kontrolün varsayılan olayına ulaşabilir ve ilgili metoduna istenilen kodlar yazılabilir. Bir kontrolün olaylarına ulaşmanın diğer bir yolu da; istenilen kontrolün özelliklerini görüntüleyip, Properties penceresindeki simgesine tıklamaktır. Böylece listelenen olaylardan herhangi birini seçerek yine Visual Studio 2005'in otomatik oluşturacağı olay metodu içerisinde gerekli işlemler yapılabilir.

Her kontrolün birden fazla olayı olabilir. Bir kontrole çift tıklandığında ele alınacak olay, o kontrolün varsayılan olayıdır. Button kontrolü için varsayılan olay **Click** iken, diğer kontroller için farklı olaylar olabilir.

Bir olaya ait olay metodunu oluşturmak ve kontrole bağlamak için Visual Studio 2005'e ihtiyacımız yoktur. Gerekli ifadeler yazılarak da, olay için olay metodu oluşturulabilir ve bu olay metodu kontrole bağlanabilir. Tabii ki bunun daha zahmetli bir işlem olacağı unutulmamalıdır.

Bir olay için olay metodu yazılmak ve kontrole bağlamak istenilirse, şu adımlar takip edilebilir.

- Örneğin bir Button kontrolüne ait Click olayına, Tiklama isimli bir olay metodu yazmak için aşağıdaki gibi bir kod örneği yazılmalıdır.

```
public void Tiklama (object sender, EventArgs e)
{

}
```

- Daha sonra bu olay metodunu, kontrole bağlamak gerekmektedir. Bu işlem için sayfanın Source kısmına geçip HTML kodları içerisinde Button kontrolüne, **onclick="Tiklama"** ifadesini eklemek gerekecektir. Böylece sayfa içerisindeki bir butona bir olay metodu bağlanmış olur.

Web Sayfalarının Olay Tabanlı Yaşam Döngüsü

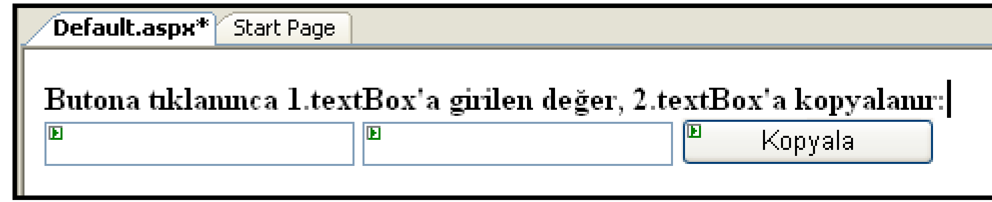
Bir web sayfası, talep edildiğinde sunucu tarafında Common Language Runtime tarafından ele alınırken bir yaşam döngüsü gerçekleşir. Bu yaşam döngüsünün parçalarından bazıları **Page_Load** olayı ve kontrollerin **Click**/**Change** olaylarıdır. .aspx uzantılı sayfaların bir yaşam döngüsü vardır ve sayfalar her talep edildiğinde bu döngü, kontrollerin olay metodları da eklenerek tekrarlanır. Öncelikli olarak Page_Load ve Click/Change olaylarının ele alınışı üzerinde durulursa; sayfanın yaşam döngüsünde Page_Load, kontrollerin Click/Change olay metotlarından önce çalışır. Bu da dikkat edilmesi gereken bir ayrıntıdır. Hem bu önceliği kanıtlamak, hem de nerede dikkatli olunması gerektiğini görmek adına küçük bir uygulama yapılabilir.

Bir web sayfasının yaşam döngüsünde kontrollerin click/change olayları, sayfa ilk talep edildiğinde tetiklenmez. Kullanıcı ilk talebi sonrasında istediği sayfayı gördükten sonra kontrollere ait olay metotlarını tetikletebilir.

Öncelikle dosya sisteminde veya IIS üzerinde bir uygulama açarak çalışılacak web sayfasının hazır olması sağlanmalıdır. Web sayfasını, iki tane TextBox ve bir tane Button

kontrolü içerecek şekilde tasarlayıp, daha sonra kontrollere kolay erişim için; bu kontrollerin ID bilgilerinin düzenlenmesi önerilir. Örneğin TextBox'ların ID özelliklerine "txtKaynak" ve "txtHedef", butona ise "btnKopyala" adı verilebilir. Butonun Text özelliğine ise "Kopyala" verilerek aşağıdaki görüntü elde edilir. (Kontrollerin bu özellikleri, ilgili kontrolün üzerine bir kez tıklanarak "Property" penceresinden ya da F4'e basılarak verilebilir. Eğer "Property" penceresi görünmüyorsa View menüsünden görünür hale getirilebilir.)

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Default.aspx.cs"
Inherits="_Default" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>Untitled Page</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>
        <strong>Butona tıklanınca 1.textBox'a girilen değer, 2.textBox'a
kopyalanır</strong><br />
         
        <asp:TextBox ID="txtHedef" runat="server"></asp:TextBox>
        <asp:TextBox ID="txtKaynak" runat="server"></asp:TextBox>
        <asp:Button ID="btnKopyala" runat="server" Text="Kopyala" /><br />
        <br />
        </div>
    </form>
</body>
</html>
```



Şekil 303: Bir sayfanın yaşam döngüsü analizi için gerekli ekranın hazırlanması

Senaryoya göre "Kopyala" butonuna tıklandığında, "txtKaynak" adlı ilk textBox'a yazılan değer, "txtHedef" adlı ikinci textBox'a taşınacaktır. Bunun için butonun Click olayı ele alınır. İlgili olay metodu, butona çift tıklanarak ya da event listesinden ilgili olaya çift tıklanarak oluşturulur.

```
public partial class _Default : System.Web.UI.Page
{
   protected void Page_Load(object sender, EventArgs e)
   {
   }
   protected void btnKopyala_Click(object sender, EventArgs e)
   {
      //soldaki textBox'a girilen değer, sağdaki textBox'a kopyalanır.
      txtHedef.Text = txtKaynak.Text;
   }
}
```

Bu olay metodunun hazırlanmasının ardından uygulama F5 ile çalıştırılabilir. Uygulama çalıştığında 1. TextBox'a bir veri girilir.(Şekil 304) Ardından butona basılarak, değerin ikinci TextBox'a taşınması sağlanır. (Şekil 305)

Butona tıklanınca 1.textBox'a girilen değer, 2.textBox'a kopyalanır:
ASP.NET Kopyala

Şekil 304: İlk TextBox'a değer girilir

Butona tıklanınca 1.textBox'a girilen değer, 2.textBox'a kopyalanır:
ASP.NET ASP.NET Kopyala

Şekil 305: İlk textBox'a girilen değer, ikincisine kopyalanır

Şimdi de sayfa her yüklendiğinde txtKaynak isimli metin kutusunun içinin boş olması sağlansın. Sayfanın yüklenme olayının olay metodu olan **Page_Load** metodunda bu işlemi yapmak için, Page_Load kısmına aşağıda kalın harflerle yazılmış kısım eklenmelidir.

```
public partial class _Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        //sayfa yüklendiğinde soldaki metin kutusunun boş olmasını sağlar.
         txtKaynak.Text = "";
    }
    protected void btnKopyala_Click(object sender, EventArgs e)
    {
         //soldaki textBox'a girilen değer, sağdaki textBox'a kopyalanır.
         txtHedef.Text = txtKaynak.Text;
    }
}
```

Uygulamayı yeniden çalıştırıp neler olacağını test etmeden önce, düşünüp neler olacağını tahmin etmeye çalışmak iyi bir deneyim olacaktır. Butonun click olayında soldaki TextBox'a girilen değer sağdaki metin kutusuna taşınırken, sayfanın yüklenişi aşamasında; yani Page_Load olayında soldaki textBox'ın değeri silinecektir. Acaba butona basıldığında iki metin kutusunda ne gibi değişiklikler olacaktır? Uygulamayı tekrar çalıştırıp soldaki TextBox'a bir değer yazılarak butona tıklanmalı ve değişiklikler gözlemlenmelidir.

Butona tıklanınca 1.textBox'a girilen değer, 2.textBox'a kopyalanır:
ASP.NET Kopyala

Şekil 306: İlk TextBox'a değer girilir

Soldaki metin kutusuna bir değer girilir ve ardından butona basılarak, değerin ikinci TextBox'a taşınmasına çalışılmaktadır.

Butona tıklanınca 1.textBox'a girilen değer, 2.textBox'a kopyalanır:
Kopyala

Şekil 307: İlk TextBox'taki değer, ikinci TextBox'a kopyalanır

Buradaki beklenti, soldaki TextBox'a girilen değerin sağdakine kopyalandıktan sonra, soldaki TextBox'da bulunan değerin silinmesi, sağdakinin ise kalması yönünde olabilir. Ancak bir ASP.NET sayfasının yaşam döngüsünde Page_Load olayı, kontrollerin Click/Change olaylarından önce çalıştığı için butona basılınca sayfanın yaşam döngüsü gereği, önce Page_Load olay metodundaki kod çalışır. Dolayısıyla soldaki metin kutusu (txtKaynak) içerisindeki değer silinir, ardından butonun Click olay metodu içerisindeki kodlar çalışır ve txtKaynak'taki yazı txtHedef'e kopyalanır. txtKaynak'ın Text'i boş olduğu için txtHedef'e de boş olarak kopyalama yapılır ve butona basıldığında sunucudan gelen cevap içerisinde, her iki TextBox da boş olarak gelir.

PostBack Kavramı

PostBack, ASP.Net ile web teknolojilerinde kullanılmaya başlanan bir kavramdır. Bir ASP.Net sayfası içerisinde, bir olayın tetiklenmesi sonucunda sayfanın, içerisindeki bilgilerle birlikte sunucuya gönderilmesi ve sunucuda yeniden üretilen sayfanın geri getirilmesi işlemine **PostBack** denilmektedir. Burada tetiklenen olay bir butona tıklanması veya açılır listeden bir seçeneğin seçilmesi olabilir. Ya da sayfanın ilk defa sunucudan istenmesi de olabilir. Bir .aspx sayfasına yapılan ilk talep dışındaki bütün taleplerde sayfa postback işlemini gerçekleştirmiş olur.

ASP.Net web sayfaları üzerinde her istek gerçekleştiğinde Page_Load olay metodu çalışacağı için bu işlemin sayfanın her çağrılmasında yapılması bazen sakıncalı olabilir. Bu nedenle PostBack işlemi her yapıldığında sayfadaki ilgili metodlarda, sayfanın ilk kez çağrılıp çağrılmadığı kontrol edilip, duruma göre işlemler yapılması sağlanabilir.

Bir sayfanın ilk istek veya PostBack işlemi sonucunda çağırıldığını kontrol etmek için, **Page** sınıfının **IsPostBack** özelliği kullanılır. Aşağıda **Page.IsPostBack** ifadesinin Page_Load içerisinde örnek kullanımı yer almaktadır.

```
public void Page_Load(object sender, EventArgs e)
{
	if(Page.IsPostBack == false) // sayfa postback yapılmamış ise
	{
		// sadece bir sefer yapılması gereken işler
	}
}
```

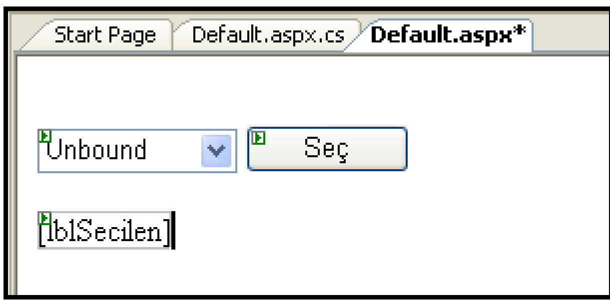
Page.IsPostBack ifadesi boolean bir değer döndürecektir. Eğer sayfa ilk defa çalışıyorsa **False**, PostBack işlemi sonucunda çalışıyorsa **True** değeri döner. IsPostBack özelliği ve PostBack kavramını daha iyi anlayabilmek için önceki örnekte kullanılan sayfanın, sadece Page_Load kısmına aşağıdaki kodlar yazılarak tekrar çalıştırılabilir.

```
public partial class _Default : System.Web.UI.Page
{
	protected void Page_Load(object sender, EventArgs e)
	{
		Response.Write("Sayfa PostBack yaptı mı : ");
		Response.Write(Page.IsPostBack.ToString());
	}
}
```

Sayfa ilk çalıştırıldığında oluşacak olan metin, "**Sayfa PostBack yaptı mı : False"** şeklinde olacaktır. Sayfa üzerindeki butona tıklandığında, buton sayfadaki bilgileri POST metoduyla sunucuya göndereceği için; sayfada PostBack işlemi gerçekleşecek ve sayfadaki yeni metin "**Sayfa PostBack yaptı mı : True"** şeklinde olacaktır.

Bir sayfaya yapılan ilk talepten sonraki taleplerde PostBack işleminin gerçekleştiği görüldü. Bu kavramdan faydalanacak şekilde aşağıdaki örnek uygulama yararlı olacaktır. Öncelikle dosya sisteminde ya da IIS üzerinde yeni bir uygulama daha açarak çalışılacak web sayfasının hazır olması sağlanmalıdır. Bu sayfaya, Toolbox'dan **ddlListe** ID'li bir DropDownList, **lblSecilen** ID'li bir Label (Label'ın Text'i silinebilir), **btnSecilen** ID'li ve Text değeri "Seç" olan bir buton atılmalıdır.

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Default.aspx.cs"
Inherits="_Default" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>Untitled Page</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>
        <br />
        <asp:DropDownList ID="ddlListe" runat="server" Width="100px"
AutoPostBack="True"
OnSelectedIndexChanged="ddlListe_SelectedIndexChanged">
        </asp:DropDownList>
        <asp:Button ID="btnSec" runat="server" OnClick="btnSec_Click"
Text="Seç" Width="82px" />
        <br />
        <br />
        <asp:Label ID="lblSecilen" runat="server"></asp:Label></div>
    </form>
</body>
</html>
```



Şekil 308: PostBack kavramı için hazırlanan sayfanın Design kısmında görünümü

Daha sonra sayfa yüklenirken DropDownList'e kod yoluyla üç yeni eleman eklenmektedir. Bunu yaparken DropDownList'in Items koleksiyonunun Add() metodu kullanıldığına dikkat edilmelidir.

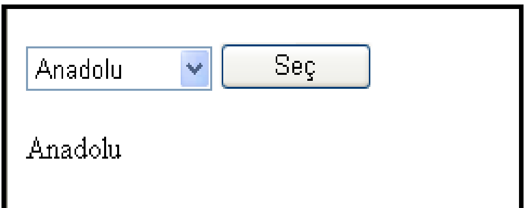
```
public partial class _Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        ddlListe.Items.Add("Murat 131");
        ddlListe.Items.Add("Anadolu");
        ddlListe.Items.Add("Kartal");
    }
}
```

Butona basıldığında kullanıcının DropDownList'den seçtiği değer Label'a yazdırılır. Bunun için DropDownList'in **SelectedItem** özelliğinin **Text** değeri kullanılabilir. Butonun Click olayına bağlı olay metodu içerisine aşağıdaki kalın harflerle yazılmış kodlar eklenebilir.

```
public partial class _Default : System.Web.UI.Page
{
    ...
```

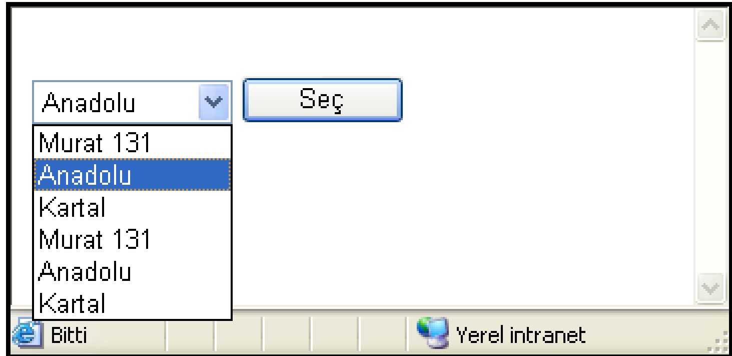
```
    protected void btnSec_Click(object sender, EventArgs e)
    {
        lblSecilen.Text = ddlListe.SelectedItem.Text;
    }
}
```

Bu şekilde uygulama çalıştırılırsa; DropDownList'ten bir araba seçip butona basıldığında değeri label'a yazdırılacaktır.



Şekil 309: ddlListe ID'li DropDownList'den değer seçilip butona basıldığı durum

Uygulama sorunsuz çalışıyor gibi gözükse de, aslında önemli bir sorun var. Butona basıldıktan sonra dropDownList'den yeni bir değer seçmek üzere listelendiğinde değerlerin ikiye katlandığı görülür. Butona her basılıp sayfa PostBack yapıldığında, bu listedeki değerlerin sayısı artmaya devam edecektir.



Şekil 310: Her butona basılışında liste artar

Burada sayfa, sunucuya her gönderildiğinde kontroller dahil bütün sayfa içeriği yok edilip yeniden oluşturulmaktadır. Buna rağmen dropDownList'in önceki talepten kalan verileri hatırlaması, yeni gelen taleplerde de aynı verileri yeniden üzerine eklemesinin sebebi **VIEWSTATE'**dir. Bir kontrole eklenen verilerin bir sonraki talepde ve sayfa oluşumunda hatırlanmasının nedeni; viewstate nesnesidir. Viewstate nesnesi, arka planda istemci ile sunucu arasında gizli bir nesne olarak verileri taşır. Yukarıdaki ikiye-üçe... katlanmanın sebebi de budur. Bütün kontrollerin **EnableViewState** özellikleri varsayılan olarak **true**'dur. İstenirse **false** yapılarak kontrollerin önceki talepde kendisine yüklenen verileri hatırlamasının önüne geçilebilir. (TextBox, RadioButton ve CheckBox kontrolleri hariç)

DropDownList'e eleman eklenilen kısım Page_Load olay metodudur. Butona basılarak veya başka bir yolla sayfa sunucuya her gönderilişinde o kod çalışır ve listeye aynı elemanları sürekli ekler. Bunun önüne geçmenin yolu ise; Page_Load'daki listeye eleman ekleme kodunun sadece sayfaya yapılan ilk talepte çalışmasını sağlamak ve sonraki taleplerde de çalışmasını engellemektir. **Page.IsPostBack** özelliğini kullanarak bu kontrolü yapmak sorunu çözecektir.

```
public partial class _Default : System.Web.UI.Page
{
```

```
    protected void Page_Load(object sender, EventArgs e)
    {
        if (Page.IsPostBack == false)
        {
            ddlListe.Items.Add("Murat 131");
            ddlListe.Items.Add("Anadolu");
            ddlListe.Items.Add("Kartal");
        }
    }
}
```

Sayfa ilk talep edildiğinde Page.IsPostBack özelliği false döneceğinden, kontrol if bloğuna girecek ve dropDownList'e istenilen elemanlar eklenecektir. Sayfaya yapılan diğer taleplerde Page_Load olay metodu çalışmasına rağmen, Page.IsPostBack özelliği true döneceği için kontrol if bloğuna girmeyecek ve dropDownList'e elemanların yeniden eklenmesi önlenmiş olacaktır.

Butona basıldığında DropDownList'de seçilen değerin Label'a yazdırılması yerine DropDownList'den bir değer seçildiği anda Label'a yazdırılması istenirse; ele alınması gereken olay DropDownList'in **SelectedIndexChanged** olayıdır. Bu olayın, DropDownList'de seçilen değer değiştiği anda tetiklenmesi beklenir. Olay metodunu ele almak için DropDownList'e çift tıklayabilir veya **ddlListe** ID'li DropDownList'e sağ tıklanıp "Property" penceresinden yıldırım işaretli "Events" kısımına geçilip, SelectedIndexChanged'e çift tıklanabilir. Butonun Click olay metodunda yazılan kod, aynen ddlListe'nin SelectedIndexChanged olayına bağlı olan olay metoduna yazılır.

```
public partial class _Default : System.Web.UI.Page
{
    ...

    protected void btnSec_Click(object sender, EventArgs e)
    {
        //lblSecilen.Text = ddlListe.SelectedItem.Text;
    }
    protected void ddlListe_SelectedIndexChanged(object sender, EventArgs e)
    {
        lblSecilen.Text = ddlListe.SelectedItem.Text;
    }
}
```

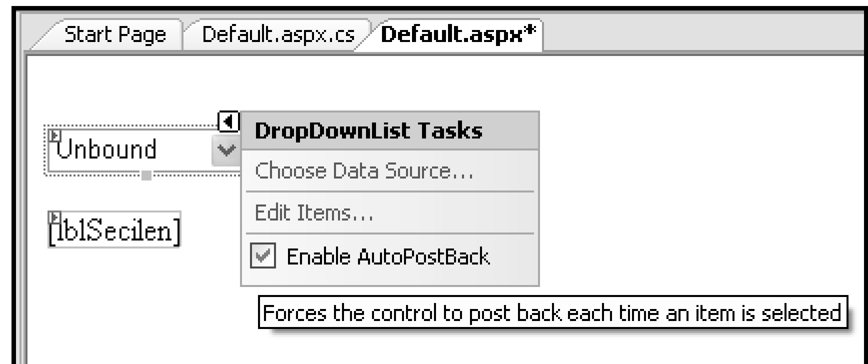
Bu olay metodu hazırlandıktan sonra, butonun Click olayında çalışan kod, sağlıklı test için yorum satırı haline getirilir. Uygulama bu haliyle çalıştırılır.

Şekil 311: DropDownList'de seçilen değer değiştiğinde çalışması gereken olay metodunun tetiklenmediği görülür

Uygulama bu haliyle çalıştırıldığında DropDownList'den bir araba seçildiğinde değerini Label'a yazdıran olay tetiklenmez. Ancak Click olayının olay metodunda herhangi bir kod bulunmayan butona basılması durumunda ise; Label'a istenen değer yazılır. Bunun nasıl gerçekleştiği adım adım düşünülürse;

- DropDownList için yazılan SelectedIndexChanged olayına bağlı olay metodu sunucuda çalışması gereken bir koddur. Dolayısıyla sayfanın sunucuya gönderilmesi gerekir.
- Buton, varsayılan olarak Click olayı ile bütün sayfayı sunucuya gönderir.

- DropDownList'de seçilen değer her değiştiğinde sayfa sunucuya gönderilmemektedir. Bu yüzden SelectedIndexChanged olay metodu çalışmaz. Ancak butona basıldığında sayfa sunucuya gönderildiği için, SelectedIndexChanged olayı tetiklenir ve olay metodu çalışır.
- O zaman çözüm; DropDownList'e, buton gibi sayfayı postback etme yeteneği kazandırmak olacaktır. Bu da DropDownList'in "Property"lerinden **AutoPostBack** özelliğini **True** yaparak sağlanır. Ya da DropDownList'in smart tag'ından (sağ üst köşesinde çıkan etiket) "EnableAutoPostBack" CheckBox'ı tıklanarak da gerçekleştirilir.



Şekil 312: DropDownList'in AutoPostBack özelliği true yapılır

Uygulama bu değişiklikten sonra çalıştırılırsa, DropDownList'de seçilen değer her değiştiğinde sayfa sunucuya gönderilecektir ve SelectedIndexChanged olay metodu çalışacaktır.

Şekil 313: DropDownList'in AutoPostBack'inin açık olmasından sonraki durum

Görüldüğü gibi, DropDownList'e eleman eklerken ve bir olay metodunu kodlarken aktif bir şekilde postback kavramından faydalanıldı. Bu küçük örnekler, postback kavramının önemini göstermesi açısından oldukça faydalıdır.

BÖLÜM 2: WEB KONTROLLERİ

ASP.NET ile geliştirilen web sayfalarında kullanılan, bazı önemli işlevleri ve özellikleri olan ve birçok işlemin kolay bir şekilde yapılmasını sağlayan hazır kontroller bulunmaktadır. ASP.NET 2.0 sürümü ile gelen yeni kontrollerle birlikte 60'ın üzerinde kontrol vardır. Bu kontroller değişik amaçları yerine getirmekte ve uygulamalarda birçok kolaylık sağlamaktadır. ASP.NET'te kullanılabilecek web kontrolleri, **sunucu kontrolleri** ve **HTML(istemci) kontrolleri** olmak üzere iki sınıfa ayrılmaktadır. Bu kontrollerin tamamına, Visual Studio 2005 içerisindeki **Toolbax**(Araç Kutusu) penceresinden ulaşılabilir. Bu kontroller, Toolbox içerisinde görevlerine ve özelliklerine göre oluşturulmuş farklı gruplar içerisinde toplanmıştır. Bu gruplar şunlardır:

- **Standart**: Temel web programlama işlemlerini yerine getirebilecek olan kontrollerin yer aldığı gruptur.
- **Data**: Veri ile ilgili işlemler için kullanılan kontrollerin yer aldığı gruptur.
- **Validation**: Veri doğrulaması ve veri kontrolü için kullanılan kontrollerin yer aldığı gruptur.
- **Navigation**: Site içi dolaşımın sağlanması için kullanılan kontrollerin yer aldığı gruptur.
- **Login**: Üyelik sistemi için kullanılan kontrollerin yer aldığı gruptur.
- **WebParts**: Uygulamanın yönetilebilen ve kişiselleştirilebilen parçalara ayrılmasını sağlayan gruptur.
- **HTML**: ASP.NET kontrolleri dışındaki sıklıkla kullanılan HTML kontrollerinin bulunduğu kontrol grubudur.
- **General**: Tüm bunlar dışında kalan, genelde sonradan eklenen kontrollerin yer aldığı gruptur.

Sunucu Kontrolleri (Server Controls)

ASP.NET uygulamalarında sunucu tarafındaki işlemlerin gerçekleşmesinde rol alacak kontrollere sunucu kontrolleri denir. Toolbax içerisinde HTML grubu dışında yer alan tüm kontroller birer sunucu kontrolüdür. Sunucu kontrolleri sayfanın Source kısmında tanımlanırken sunucu tarafında çalışacağı **runat="server"** ifadesi ile belirtilir. Web uygulamalarında sıklıkla kullanılan sunucu kontrolleri bu konu içerisinde incelenecektir.

Standart Kontroller (Standard Controls)

Web uygulamalarında kullanılabilecek temel işlemleri yerine getiren sunucu kontrolleri, Toolbox içerisinde standart kontroller grubunda toplanmıştır. Standart kontroller kullanım sıklıklarına göre incelenecek olursa;

- Label (Etiket): Web sayfası üzerindeki durağan metinler için kullanılabileceği gibi, sayfanın herhangi bir yerinde bir butonun tıklanmasıyla veya herhangi bir olayın tetiklenmesiyle de içeriği değiştirilebilir. Bu elemanın en çok kullanılan özelliği **Text** özelliğidir. Text özelliği, etiketin içinde görüntülenecek yazıyı tutan özelliktir. Basit olarak bir Label şu şekilde tanımlanabilir.

```
<asp:Label ID="lblIsim" runat="server" Text="Merhaba Dünya"/>
```

- TextBox (Metin Kutusu): Bir metnin görüntülenmesi için kullanılabileceği gibi, daha çok metin girişleri için kullanılan bir kontroldür. Basit olarak bir TextBox şu şekilde tanımlanabilir.

```
<asp:Textbox   ID="txtIsim"   runat="server"   Text="Metin   kutusu   içinde
```

```
yazı"></asp:Textbox>
```

Textbox kontrolüne, tek satırlık metin girilecekse **TextMode** özelliğini **SingleLine**, birden fazla satırdan oluşan bir metin girilecekse **MultiLine**, eğer şifre girişi için kullanılacaksa (girilen karakterlerin görüntülenmesini engellemek için) **Password** moduna ayarlamak gerekmektedir.

- Button (Buton-Düğme): Butonlar web formlarında yaygın olarak kullanılan elemanlardır. Bir sayfaya gitmek, bir formu göndermek, doldurduğumuz formu temizlemek gibi farklı işlerin yapılmasını sağlarlar. ASP.NET'te bir buton üzerine tıklandığında kendisine atanan olayı tetikler ve tetiklenen olaya göre de farklı işlemler yaptırılabilir. Butonun üzerindeki yazıyı değiştirmek için **Text** özelliği kullanılır. Basit olarak bir Button şu şekilde tanımlanabilir.

```
<asp:Button ID="btnIsim" runat="server" Text="Tıkla"/>
```

- HyperLink (Bağlantı): Bu kontrol bir sayfadan başka bir sayfaya geçmek veya aynı sayfa içinde yapılacak bir işleme bağlantı kurmak amacı ile kullanılır. Gidilecek olan sayfanın adresi **NavigateUrl** özelliği ile belirtilir. Bu kontrolün üzerine tıklandığı zaman doğrudan istenen sayfaya yönlenme olayı gerçekleşir, sunucu tarafından herhangi bir olay tetiklenmez. Basit olarak bir HyperLink şu şekilde tanımlanabilir.

```
<asp:HyperLink ID="hypIsim" NavigateUrl="anasayfa.aspx" runat="server"
Text="Anasayfaya Git"/>
```

- LinkButton (Link Butonu): LinkButton kontrolü HyperLink kontrolü ile benzerlik göstermektedir. Ancak HyperLink belirtilen bir adresin açılmasını sağlarken, LinkButton aynen Button kontrolü gibi üzerine tıklandığında kendisine atanan olayı tetikler. Kısacası LinkButton, HyperLink görünümünde bir buton türüdür. LinkButton kontrolü basit olarak şu şekilde tanımlanabilir.

```
<asp:LinkButton ID="linkIsim" runat="server">Lütfen Tıklayınız
</asp:LinkButton>
```

- ImageButton (Resim Butonu): ImageButton kontrolü de Button kontrolü gibi tıklandığında sunucu tarafında bir olayı tetikler. Ancak Button'dan farkı, görüntü olarak istenilen resmin kullanabilmesidir. Ayrıca ImageButton'a bir ImageMap tanımlayıp, resmin değişik yerlerine tıklandığında, değişik işlemler yapması sağlanabilir. ImageButton kontrolü basit olarak şu şekilde tanımlanabilir.

```
<asp:ImageButton ID="ImageButton1" runat="server" ImageUrl="~/resim.jpg"/>
```

- Image (Resim): Sayfada GIF, JPG, PNG veya kabul edilebilir diğer formatlarda resim görüntülemek için kullanılır. ImageButton'a benzer; ancak üzerine tıklandığında sunucu tarafında bir olay tetiklenmez.(HTML'deki <img> etiketi ile benzerdir) Image kontrolü basit olarak şu şekilde tanımlanabilir.

```
<asp:Image ID="imgIsim" runat="server" ImageUrl="~/resim.jpg" />
```

- DropDownList (Açılır Liste): Sayfada aşağıya doğru açılır bir liste ile çeşitli seçenekler arasından bir tanesini kullanıcıya seçtirmek amacıyla kullanılabilecek olan kontroldür. İçerisinde kullanılacak ListItem kontrolü ile seçilecek bilgiler tanımlanabilir. DropDownList kontrolü ListItem ile birlikte basit olarak şu şekilde tanımlanabilir.

```
<asp:DropDownList ID="ddlIller" runat="server">
   <asp:ListItem>Ankara</asp:ListItem>
   <asp:ListItem>İstanbul</asp:ListItem>
```

```
   <asp:ListItem>İzmir</asp:ListItem>
</asp:DropDownList>
```

- ListBox (Liste Kutusu): DropDownList kontrolüne benzeyen bu kontrolde, menü açıktır ve seçeneklerin hepsi görünür bir haldedir. Kullanıcı bir veya daha fazla seçeneği seçebilir. ListBox kontrolü basit olarak şu şekilde tanımlanabilir.

```
<asp:ListBox ID="lboxIlceler" runat="server">
   <asp:ListItem>Bakırköy</asp:ListItem>
   <asp:ListItem>Beşiktaş</asp:ListItem>
   <asp:ListItem>Pendik</asp:ListItem>
   <asp:ListItem>Üsküdar</asp:ListItem>
</asp:ListBox>
```

- CheckBox (Seçim Kutusu): Kullanıcının üzerine tıklayarak seçebileceği bir kontroldür. Kullanıcıya seçim yaptırmak istenildiğinde kullanılabilir. CheckBox kontrolü basit olarak şu şekilde tanımlanabilir.

```
<asp:CheckBox ID="cbIsim" runat="server" Text="Seçmek için tıklayın" />
```

- CheckBoxList (Seçim Kutusu Listesi): CheckBox kontrolünün gruplanmış ve listelenmiş halidir. Kullanıcı listedeki seçeneklerden bir veya daha fazlasını seçebilir. CheckBoxList kontrolü ListItem ile birlikte basit olarak şu şekilde tanımlanabilir.

```
<asp:CheckBoxList ID="cblUrunler" runat="server">
   <asp:ListItem>Fare</asp:ListItem>
   <asp:ListItem>Klavye</asp:ListItem>
   <asp:ListItem>Monitör</asp:ListItem>
</asp:CheckBoxList>
```

- RadioButton: Kullanıcının seçenekler arasından sadece bir tanesini seçmesi istenildiği durumlarda kullanılabilecek bir kontroldür. Üzerine tıklanan seçenek işaretlenir.

```
<asp:RadioButton ID="rbIsim" runat="server" Text="Seçmek için tıklayın" />
```

- RadioButtonList: RadioButton kontrolünün gruplanmış ve listelenmiş halidir. Kullanıcı listedeki seçeneklerden sadece bir tanesini seçebilir. RadioButtonList kontrolü ListItem ile birlikte basit olarak şu şekilde tanımlanabilir.

```
<asp:RadioButtonList ID="RadioButtonList1" runat="server">
   <asp:ListItem>Kırmızı</asp:ListItem>
   <asp:ListItem>Mavi</asp:ListItem>
   <asp:ListItem>Yeşil</asp:ListItem>
</asp:RadioButtonList>
```

- ImageMap (Resim Haritası): Bir resim üzerinde koordinatlara göre başka sayfalara veya sayfa içerisindeki bir noktaya bağlantı atamak için kullanılır. Koordinatlar, poligonal, dairesel veya karesel olabilir.

```
<asp:ImageMap ID="imIsim" runat="server" ImageUrl="~/resim.gif">
</asp:ImageMap>
```

- Table (Tablo): Tasarımın daha düzgün görünmesi ve verilerin karışmaması için kullanılan HTML tablonun sunucu kontrol halidir.

```
<asp:Table ID="tblIsim" runat="server" />
```

- BulletedList: Sayfada verilerin maddeli bir şekilde listelenmesini sağlayan sunucu kontrolüdür.

```
<asp:BulletedList ID="BulletedList1" runat="server">
   <asp:ListItem>Beşiktaş</asp:ListItem>
   <asp:ListItem>Fenerbahçe</asp:ListItem>
   <asp:ListItem>Galatasaray</asp:ListItem>
   <asp:ListItem>Trabzonspor</asp:ListItem>
</asp:BulletedList>
```

- HiddenField: Aynı sayfa için arka planda veri taşıma işleminde kullanılan ve sunucu taraflı kodlardan da erişilebilen ve yönetilebilen sunucu kontrolüdür.

```
<asp:HiddenField ID="hfIsim" runat="server" Value="Taşınacak değer" />
```

- Literal: İçine başka kontroller alabilen bir "Container Control"'dür. Gizlendiği zaman içindeki tüm kontrol ve veriler ile gizlenir. Çalışma zamanında dinamik olarak sayfaya elementler eklemek istenildiğinde kullanılır.

```
<asp:Literal ID="Literal1" runat="server"></asp:Literal>
```

- Panel: Literal kontrolü gibi diğer kontrolleri ve içerikleri gruplamak için kullanılan bu kontrol de gizlendiği zaman, içindeki tüm kontroller gizlenmiş olur.

```
<asp:Panel ID="Panel1" runat="server" />
```

- PlaceHolder: Literal ve Panel kontrollerine çok benzeyen bu kontolde aynı işlevleri görür.

```
<asp:PlaceHolder ID="PlaceHolder1" runat="server"></asp:PlaceHolder>
```

- Calendar: Sayfaya takvim eklemek için kullanılır. Birçok ayarı değiştirilebilir ve özelleştirilebilir.

```
<asp:Calendar ID="Calendar1" runat="server"></asp:Calendar>
```

- AdRotator: ASP.NET'in zengin kontrollerinden bir diğeridir. Sayfaya bir reklam göstericisi ekler. Bir XML dosyasından aldığı bilgilere göre, sayfada reklamların değiştirilerek gösterilmesini sağlar. Şu şekilde tanımlanır.

```
<asp:AdRotator ID="AdRotator1" runat="server" />
```

- FileUpload: ASP.NET 2.0 ile gelen bu kontrol sunucu tarafına dosya yüklenmesi işleminde büyük bir kolaylık sağlamaktadır. FileUpload şu şekilde tanımlanabilir.

```
<asp:FileUpload ID="FileUpload1" runat="server" />
```

- Wizard: ASP.NET 2.0 ile gelen bu kontrol ise sayfada adım adım çalışacak bir sihirbaz oluşturmak için kullanılır. WizardStep kontrolü ile Wizard içerisine adımlar tanımlanmaktadır. Örnek olarak şu şekilde tanımlanabilir.

```
<asp:Wizard ID="wzIsim" runat="server">
   <WizardSteps>
         <asp:WizardStep runat="server" Title="1.Adım"></asp:WizardStep>
         <asp:WizardStep runat="server" Title="2.Adım"></asp:WizardStep>
   </WizardSteps>
</asp:Wizard>
```

Veri Kontrolleri (Data Controls)

- GridView: ASP.NET 1.x versiyonundaki DataGrid'in daha gelişmiş versiyonu olarak ASP.NET 2.0 ile gelen bu kontrol, verilerin listeleme işini çok basit bir şekilde gerçekleştirmekte ve birçok işlevselliği yapısında barındırmaktadır. Veri silme, ekleme, güncelleme gibi işlemlerin dışında verileri sayfalama, sıralama gibi işlemleri de çok basit bir şekilde yapmaktadır

```
<asp:GridView ID="gvIsim" runat="server" />
```

- DataList: Veri listelemek için kullanılan bu kontrol verileri yatay veya dikey olarak verilen şablona göre tekrarlı olarak sıralar.

```
<asp:DataList ID="dlIsim" runat="server" />
```

- Repeater: Veri listeleme kontrollerinin en basiti olan bu kontrol sadece verilen şablona göre verileri sıralar.

```
<asp:Repeater ID="rpIsim" runat="server" />
```

- DetailsView: Diğer veri listeleme kontrollerinin aksine satırlar ve sutünlar yerine, bir satırda bir veriye ait tüm sütunları sıralar. Veri ekleme, silme ve güncelleme için kullanılır.

```
<asp:DetailsView ID="dvIsim" runat="server" />
```

Site içi Dolaşım Kontrolleri (Navigation Controls)

SiteMapPath: Web.sitemap dosyasındaki sayfa ve bölümleri yatay olarak üst kategoriden alt kategoriye göre sıralandıran, kullanıcının site içinde kaybolmaması veya nerede olduğunu görmesi için kullanılan kontroldür.

```
<asp:SiteMapPath ID="SiteMapPath1" runat="server" />
```

Menü: Web.sitemap dosyasındaki verileri kullanarak çok basit bir şekilde istemci taraflı dinamik menüler oluşturmak için kullanılır.

```
<asp:Menu ID="Menu1" runat="server" />
```

TreeView: Web.sitemap dosyasındaki verilerin ağaç yapısı şeklinde gösterilmesini sağlayan kontroldür. Birçok uygulamada dinamik menüler oluşturmak için kullanılır.

```
<asp:TreeView ID="TreeView1" runat="server" />
```

Site içi dolaşım kontrolleri, **web.sitemap** isimli bir dosyayı kaynak olarak kullanır. Bu dosya XML biçiminde hazırlanmış bir dosya olup, sitedeki bölümlerin ve sayfaların hiyerarşik yapısını saklar. Site içi dolaşım kontrolleri, ASP.NET 2.0 ile gelmiş yeni kontrollerdir.

Web Sayfalarına Kontrol Ekleme

ASP.NET'te mevcut olarak bulunan bir kontrolü uygulama içerisine eklemek için yapılması gereken yol, kontrolü ifade eden kodları ilgili sayfanın kodları arasında, istenilen yere eklemektir. Bu aşamada dikkat edilmesi gereken kurallar şöyledir;

- Tüm kontrollerde **ID** özelliği bir isim ile belirlenmiş olmalı ve hepsi **runat="server"** ifadesini içermelidir.
- Kontrollerin kodları eklenirken XML'e uygun biçimde etiketleri kapatılmalıdır. Bunun için iki yöntem mevcuttur;
    1. Etiket içinde sonlandırma:
       ```
       <asp:TreeView ID="TreeView1" runat="server" />
       ```
    2. Etiket dışında sonlandırma:
       ```
       <asp:TreeView ID="TreeView1" runat="server"></asp:TreeView>
       ```
- Bazı kontrollerin mutlaka belirtilmesi gereken özellikleri vardır. Kullanmak isterken bunları mutlaka belirtmek gerekir.
- Aynı isimde (ID) iki tane kontrol, aynı sayfaya eklenemez. Source kısmından kontrol eklemek istenmiyorsa; Toolbox'taki kontroller üzerinden kod yazmadan da ekleme işlemi yapılabilir. Toolbox'tan kontrol eklemek için iki yöntem vardır.
    - Kontrole çift tıklayarak: Toolbox'taki bir kontrolün üzerine fare ile çift tıklandığında, kontrol o an dosya üzerinde imlecin bulunduğu yere eklenir.
    - Sürükleyip bırakarak: Toolbox'ta bulunan bir kontrolü, fare ile seçip sayfa üzerinde bırakılan noktaya eklenmesi sağlanabilir.

HTML Sunucu Kontrolleri

ASP.NET içerisinde sunucu tarafında olaylarda kullanılan sunucu kontrolleri dışında HTML dilinden alışık olunna kontroller de bulunmaktadır. HTML sunucu kontrolleri, HTML'de sıklıkla kullanılan bazı elementlerin hem istemci hem de sunucu tarafında çalışabilecek olan biçimleridir. Standart HTML kodlarına eklenen **runat="server"** ifadesi ile bu elemanın sunucu tarafında çalışacağı, yani sunucu tarafında bu kontrolün yönetilebilir olduğu bildirilir. **id="kontrol_adi"** kodu ile de bu kontrole sunucu tarafından **kontrol_adi** ismi ile ulaşılacağı belirtilir. HTML kontrollerine aşağıda kısaca değinilmiştir.

HTML Button

HTML butonu, standart HTML de hiç bir görevi olmayan, tıklanması sonucu çalışacak kodların, istemci tarafında çalışan istemci taraflı kodlar olduğu, ancak sunucu kontrolüne çevrildiği zaman sunucu tarafından kontrol edilebilen bir kontroldür.

```
<input id="btnIsim" type="button" value="button" runat="server" />
```

HTML Reset

HTML reset butonu, HTML bir formun içerisindeki elementlerin içeriklerini temizlemek için kullanılır. Bu kontrol de HTML Button gibi sunucu kontrolüne çevrilince sunucu tarafından kontrol edilebilir hale gelir.

```
<input id="resIsim" type="reset" value="reset" runat="server" />
```

HTML Submit

HTML submit butonu, bir HTML formunu sunucuya göndermek için kullanılır. Bu kontrol de sunucu kontrolüne çevrilince, sunucu taraflı bir tıklanma olayına (event) sahip olur.

```
<input id="submIsim" type="submit" value="submit" runat="server" />
```

HTML TextBox

HTML textbox, HTML formlarında kullanıcının tek satırlık metin girişi yapabileceği metin kutusu kontrolüdür. Sunucu kontrolüne çevrildiğinde ise sunucu tarafından, içindeki yazının değişmesi durumunda tetiklenebilecek bir olaya sahip olur.

```
<input id="txtIsim" type="text" runat="server" />
```

HTML File

HTML file kontrolü, istemciden sunucuya dosya transfer işlemi sırasında kullanılan bir kontroldür.

```
<input id="fileIsim" type="file" runat="server" />
```

HTML Password

HTML password kontrolü, içerisine girilen metini gizlenmiş bir formatta maskeleyen ve genellikle şifre girişlerinde kullanılan metin kutusu kontrolüdür.

```
<input id="passIsim" type="password" runat="server" />
```

HTML CheckBox

HTML checkbox kontrolü, kullanıcının işaretleyebileceği bir seçme kutusudur.

```
<input id="cbIsim" onsetype="checkbox" runat="server" />
```

HTML Radio

HTML radio kontrolü, kullanıcının işaretleyebileceği bir seçme elemanıdır. CheckBox elemanından farkı; bir grup içerisindeki radio elemanlarından sadece bir tanesinin seçilebilmesidir. Sunucu kontrolüne çevrildiğinde ise; kullanıcının seçimi değiştirmesi durumunda sunucu tarafında bir olay tetiklenebilir.

```
<input id="radIsim" type="radio" runat="server" />
```

HTML Hidden

HTML hidden kontrolü, sayfalar arasında kullanıcının göremeyeceği verileri taşımak için kullanılan kontroldür. Bu kontrol içerisinde taşınan veriler sadece sayfanın arka planında, HTML kodları arasında taşınır ve sayfanın görünüm kısmında yer almaz.

```
<input id="hidIsim" type="hidden" runat="server"/>
```

HTML TextArea

HTML TextArea kontrolü, metin girişi yapılabilecek çok satırlı bir metin kutusudur. Sunucu kontrolüne çevrildiği zaman ise içindeki metinde değişiklik olması durumunda tetiklenen bir olaya sahip olabilir.

```
<textarea id="taIsim" cols="20" rows="2" runat="server">
```

HTML Table

HTML table kontrolü, çıktı olarak HTML tablosu üreten kontroldür. Verilerin satırlar ve sütunlar içerisinde düzenli şekilde görüntülenmesini sağlar.

```
<table id="tabIsim" runat="server">
        <tr>
                <td> </td>
        </tr>
</table>
```

HTML Image

HTML image elemanı, bir kaynaktaki resmi görüntüleyen kontroldür.

```
<img id="imgIsim" src="images/resim.gif" runat="server"/>
```

HTML Select

HTML select kontrolü, birden fazla seçeneği tek bir eleman içerisinde göstermeyi sağlayan, normalde aşağı doğru açılarak içindeki seçenekleri listeleyen bir elemandır. Sunucu kontrolüne çevrildiği zaman ise seçili olan seçenekte değişiklik olması durumunda tetiklenen bir olaya sahip olabilir. (Sunucu kontrollerinden DropDownList ile benzerdir)

```
<select id="selectIsim" runat="server" >
      <option selected="selected">Seçenek-1</option>
      <option selected="selected">Seçenek-2</option>
</select>
```

HTML Div

HTML div kontrolü, içerisinde web kontrollerini ve HTML etiketlerini bulundurabilen ve bu elementleri gruplamaya yarayan kontroldür. Ayrıca DHTML ve CSS kullanımlarında aktif roller üstlenir.

```
<div id="divIsim" style="width: 100px; height: 100px" runat="server">
```

HTML Sunucu kontrolleri, sunucu tarafında çalışmalarına rağmen sağladıkları imkanlar az olan kontrollerdir. Genel olarak var olmalarının iki sebebi vardır:

- Daha önce ASP, PHP gibi standart HTML elemanları kullanılarak oluşturulmuş sistemlerin ASP.NET'e uyumlu şekilde geçişini kolaylaştırabilir.
- Bir sayfada JavaScript, form elemanları ile birlikte aktif olarak kullanılıyorsa bazı durumlarda HTML sunucu kontrolleri daha faydalı olabilir.

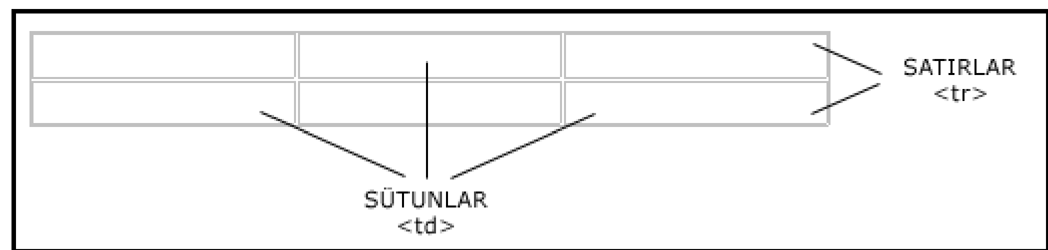
HTML Tablolar

HTML tablolar sitenin görsel olarak düzenli bir şekilde görünmesini sağlayan yapılardır. Masaüstü uygulamalarda kontrolleri, formlar üzerinde istenilen bir noktaya

konumlandırmak mümkündür. Bu, web sayfalarında mümkün olsa da, birçok durumda sayfanın görünümü ve tarayıcıda düzgün görüntülenmesi açısından sakıncalı olabilir. Web sayfalarında tablolar kullanılarak, sayfa içerisindeki elementlerin daha düzenli şekilde tasarlanması sağlanır.

HTML tabloları, yapısal olarak Microsoft Excel dosyaları içerisindeki satırlar ve sütunlara benzetilebilir. Ortaya çıkacak tablo istenilen genişlikte, yükseklikte ve sayıda satırlar ile sütunlardan oluşur. Satırlar ve sütünlar içerisine tüm ASP.NET kontrolleri ve HTML elementleri eklenebilir.

Tablolar **<table>** etiketi ile oluşturulur. Bir tablonun genişliği, hücrelerinin arasındaki boşluk ve kenarlıklarının kalınlığı gibi özellikler <table ....> etiketi içerisindeki tanımlamalarla yapabilir. **<table>....</table>** etiketi içine satırlar ve sütunlar tanımlanır. Tablolar satırlardan, satırlar ise sütunlardan oluşmaktadır. Tablo içerisine satırlar, **<tr>** etiketiyle eklenir. Satırlar içerisine sütunlar ise; **<td>** etiketiyle eklenir.



Şekil 314: Bir tablo içerisindeki satır ve sütunlar

Şekil 314'deki tablo 2 satır ve 3 sütundan oluşmaktadır. Tablo içerisindeki her kutuya hücre denilmektedir. Şekil 314'deki gibi bir tablo oluşturmak için HTML kısmında şu kodların yazılması gerekmektedir.

```
<table border="1" width="400">
      <tr>
            <td></td>
            <td></td>
            <td></td>
      </tr>
      <tr>
            <td></td>
            <td></td>
            <td></td>
      </tr>
</table>
```

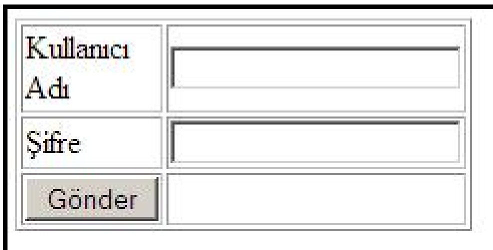
Yukarıda oluşturulan tabloda; <table> kısmındaki **border="1"** ifadesi tablonun kenar kalınlıklarının 1 piksel kalınlığında olması, **width="400"** ifadesi ise; bu tablonun yatay genişliğinin 400 piksel olması gerektiğini belirtir. <table> içerisinde 2 tane <tr> ifadesi bulunmaktadır. Bu ifade tablonun 2 satırdan oluşacağını gösterir. Her iki satır içerisinde de 3 tane <td> ifadesi bulunmaktadır; yani tablodaki satırlar 3 sütundan oluşacaktır. Bir tabloda sadece **<td></td>** etiketleri arasına içerik eklenebilir.

Bir form içerisinde tablo kullanarak, form elementlerinin tablo içerisinde nasıl kullanılabileceği aşağıdaki örnekte görülebilmektedir.

```
<form id="frmTablolar" runat="server">
<table border="1" width="200px">
        <tr>
                <td>Kullanıcı Adı</td>
                <td><asp:TextBox id="txtAd" runat="server" /></td>
        </tr>

        <tr>
                <td>Şifre</td>
                <td><asp:TextBox id="txtSifre" runat="server" /></td>
        </tr>
```

```
        <tr>
                <td><asp:Button id="btnGonder" Text="Gönder"
runat="server" /></td>
                <td> </td>
        </tr>
</table>
</form>
```

Yukarıdaki örnekteki **<form ID="frmTablolar" runat="server"></form>** kısmı kullanılacak formun sunucu tarafında çalışacağını belirler. Bir form içerisinde ASP.NET kontrolleri kullanabilmek için <form> içerisinde **runat="server"** ifadesinin yer alması gerekmektedir. 3 satır ve 2 sütündan oluşan tablonun içerisine iki tane TextBox ve bir tane Button kontrolü eklenmiştir. Yine bu kontrollerin görsel olarak anlamlı görünmesi için diğer hücreler içerisinde bazı metin ifadeleri yer almaktadır. Ayrıca içi boş bırakılan hücreler bazı durumlarda görünümü bozabileceği için son hücrenin içini boş bırakmamak için HTML dilinde boşluk karakteri anlamına gelen ** ** ifadesi eklenmiştir. Yukarıdaki örneğin görüntüsü aşağıdaki gibi olacaktır.



Şekil 315: Tablo kullanarak oluşturulan form

HTML tablolar üzerinde görsel olarak satır ve sutünların yüksekliği, kenar çizgilerinin kalınlığı veya rengi, tablonun arka plan rengi gibi birçok ayar yapmak mümkündür. Ancak tablonun bütün özelliklerini burada anlatmak mümkün olmayacağı için bu konu ile ilgili farklı kitaplardan veya internet kaynaklarından faydalanabilirsiniz.

BÖLÜM 3: ASP.NET İLE TEMEL VERİ İŞLEMLERİ

Masaüstü uygulamalarda olduğu gibi, web uygulamalarında da verilerle ilgili işlemler yapılabilmektedir. ASP.NET web uygulamalarında veri kaynaklarıyla ilgili temel işlemler yapılırken, .NET Framework içerisinde bulunan ADO.NET platformu kullanılır.

Veri Kaynakları (Data Sources)

Veri kaynakları, veritabanı ile veri kontrolleri arasında veri taşıyan elemanlardır. Kullanılan veri kaynağının türüne göre uygun olan kontrol seçilerek veri kaynağına bağlanılır ve verinin nasıl çekileceği ya da verinin nasıl ekleneceği belirtilir. Ardından da veri kaynağı kontrolü, istenilen verileri veri kontrollerine aktarır.

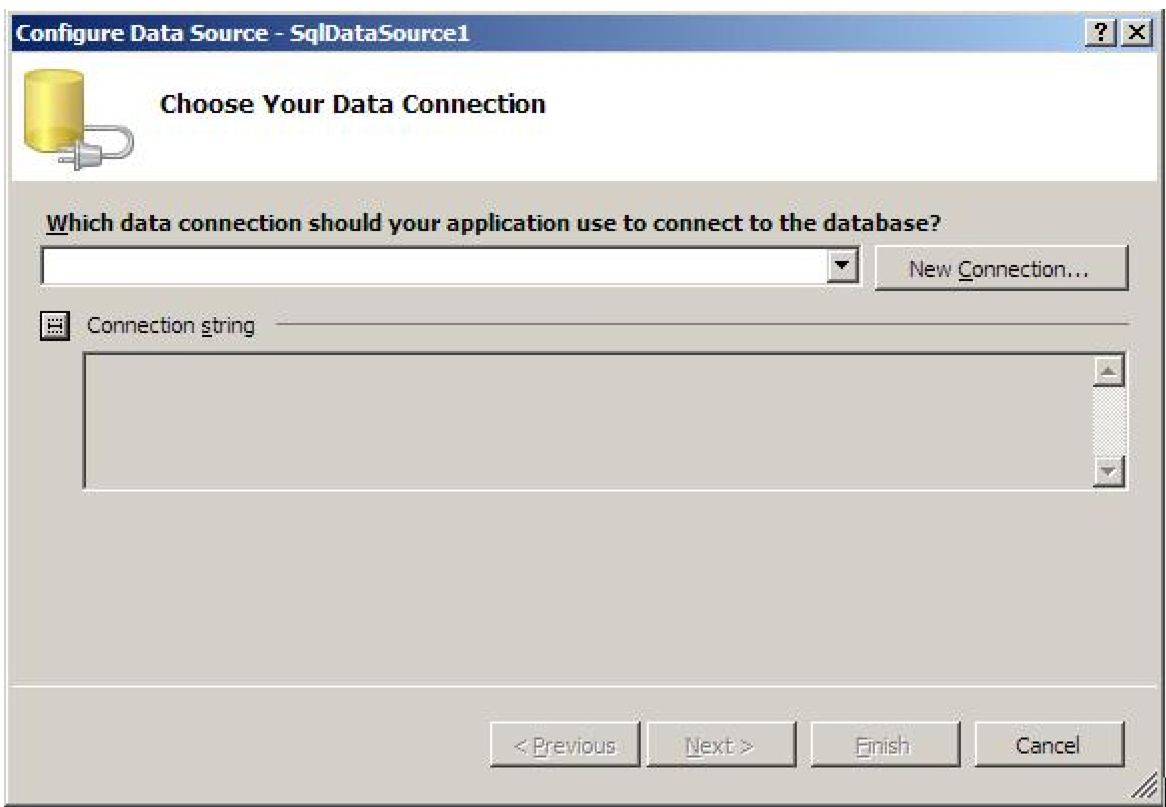
ASP.NET'te beş tane veri kaynağı bulunmaktadır. Bu veri kaynakları kontrol olarak ToolBox içerisindeki **Data** kategorisinde bulunmaktadır. Bu kontrollerden ObjectDataSource ve SiteMapDataSource ileri düzey uygulamalarda kullanılabileceği için bu kısımda sadece SqlDataSource, AccessDataSource ve XmlDataSource kontrolleri anlatılacaktır

SqlDataSource

Kullanılan veritabanı SQL Server ailesinden ise; SqlDataSource kontrolü kullanılarak SQL Server'dan veri çekilebilir, eklenebilir, güncellenebilir veya silinebilir. Ayrıca SQL Server içerisindeki Stored Procedure'leri de aktif olarak destekleyen bu kontrol sayesinde veritabanı işlemleri çok kolay bir şekilde yapılabilir.

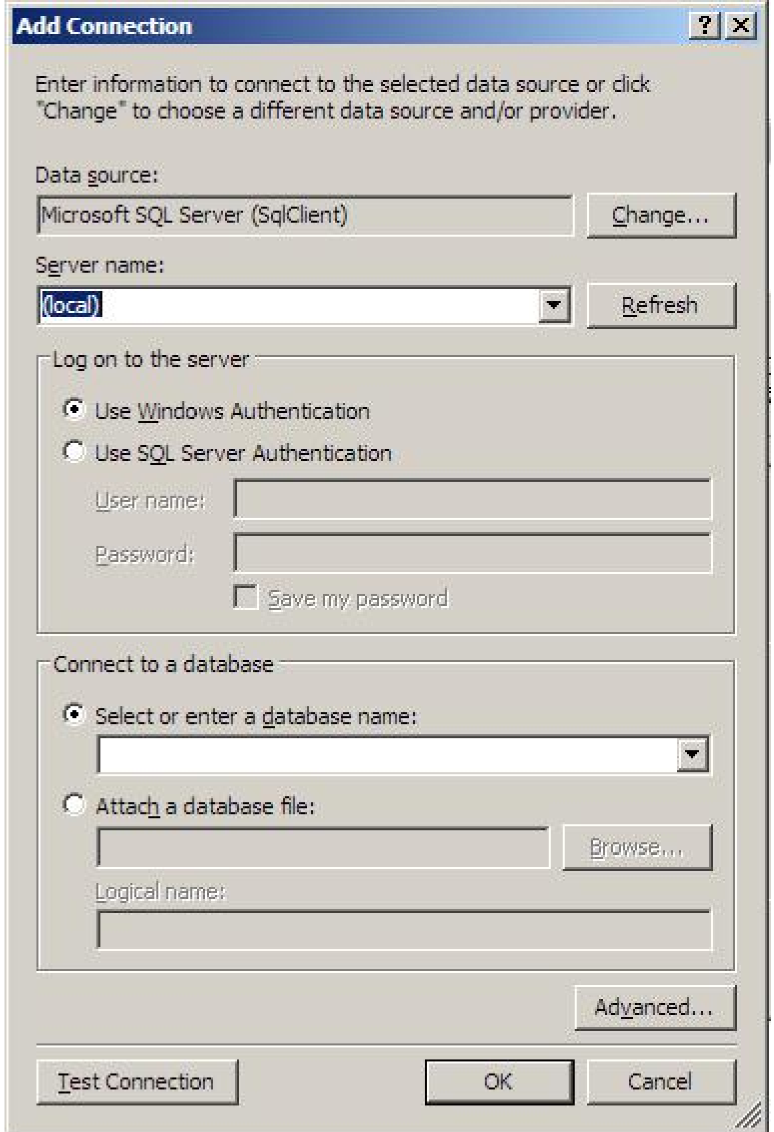
SqlDataSource ile SQL Server'da bulunan Northwind veritabanına nasıl bağlanılacağı adım adım aşağıda yer almaktadır.

- Öncelikle SqlDataSource kontrolü sayfaya eklenir ve kontrolün sol üst köşesindeki ok simgesine (Smart Tag) tıklanarak görevler penceresi açılır. Burada **Configure Data Source...** bağlantısına tıklanır.



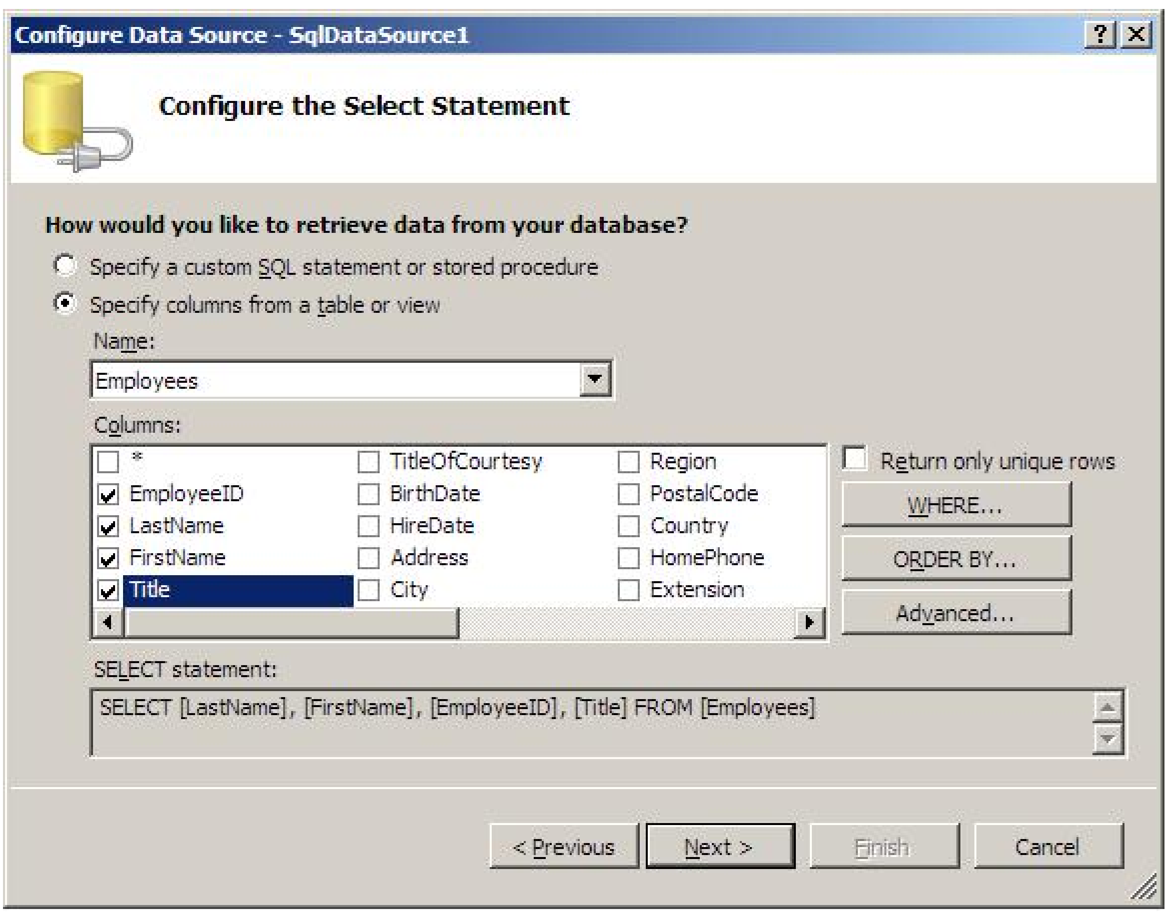
Şekil 316: SqlDataSource kontrolünde Configure Data Source menüsü seçilir

- Ilk önce bağlanılacak bir veri kaynağı ile ilgili bilgilerin oluşturulması gereklidir; bu yüzden **New Connection...** butonuna tıklanır ve yeni bir bağlantı cümlesi (**connection string**) oluşturulur.. Açılan pencerede eğer Data source kısmında **"Microsoft SQL Server (SqlClient)"** yazmıyor ise **Change** butonuna basılır ve açılan pencereden bu seçenek seçilip aşağıdaki ekranla karşılaşılır.



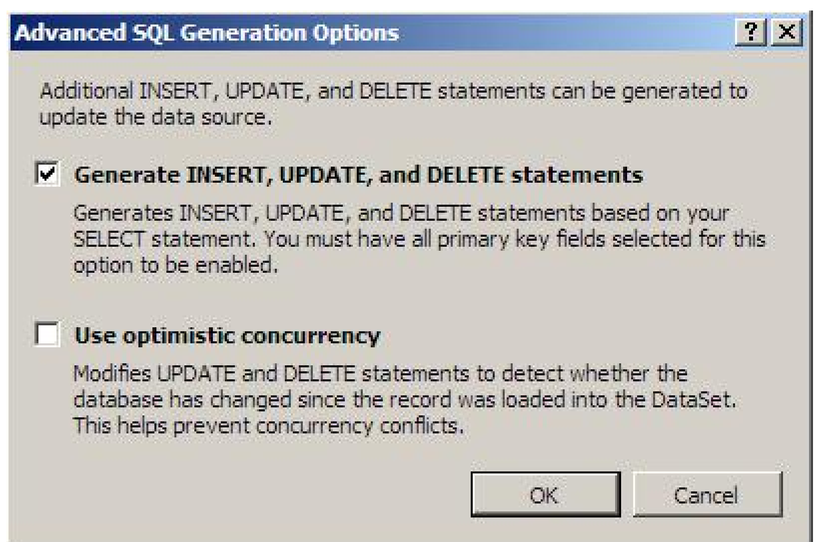
Şekil 317: Bağlanılacak veri kaynağı belirlenir

- Server name kısmına yerel makineyi temsil edecek (local) ifadesini yazılır. Ya da ".(nokta)" koyulabilir. (Bu kısıma, çalışılan makinenin adı da yazılabilir)
- Sql Server sunucu adını yazınca aktif hale gelen **"Select or enter a database name:"** seçeneğinden Northwind seçeneği seçilir ve **"OK"** butonuna tıklanır.
- Artık veritabanı bilgileri ekrana gelmiştir. Next düğmesine tıklayarak devam edilir.
- Bir sonraki ekran bu bilgilerin bir ayarlama dosyasında (.config) kaydedilip edilmeyeceğini soracaktır. **Next** diyerek kabul edilip geçilir.
- Artık veritabanındaki hangi tablo üzerinde çalışılacağının belirlenmesi gerekir. **"Name"** alanından **Employees** tablosu seçilir. Tablo seçilince **"Columns"** bölümünde o tablonun kolonları görünür. Üzerinde çalışılmak istenen kolonlar seçilir. Tablodan gelecek veri üzerinde bir şart belirterek sorgulama yapmak istenirse; **"Where..."** butonuna, sıralama yapmak istenirse; **"Order By.."** düğmesine tıklanmalıdır.



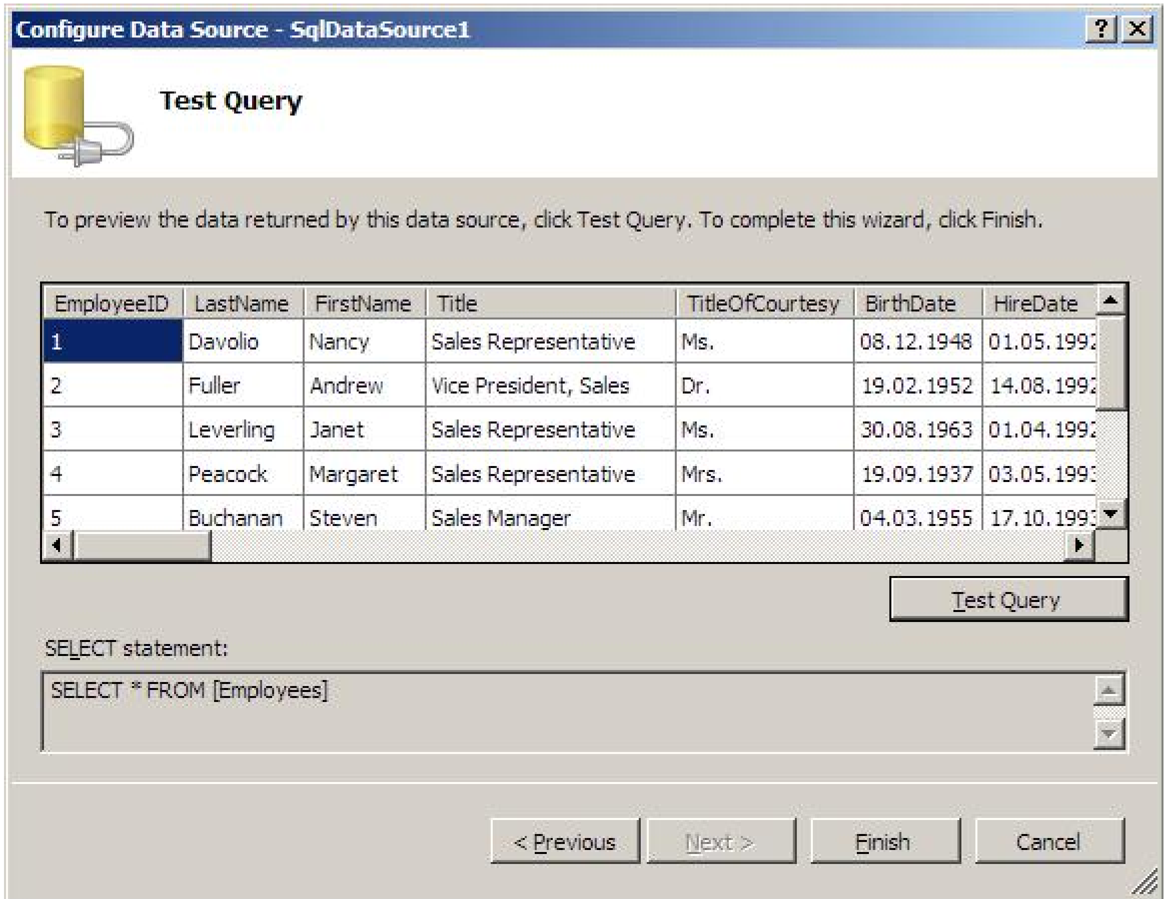
Şekil 318: Veri kaynağından çekilecek verinin sql sorgusu hazırlanır

- Advanced... butonuna tıklandığında açılan pencerede **"Generate INSERT, UPDATE, and DELETE statements"** seçeneği seçilir. Bu seçenek seçildiği zaman daha sonra eklenecek veri kontrollerinin veri ekleme, silme ve güncelleme yapabilmesini sağlayacak olan SQL sorgularının oluşturulması sağlanır. OK butonuna basıp bu pencere kapatılıp, ardından Next düğmesine tıklanarak bir sonraki ekrana geçilir.



Şekil 319: Gelişmiş teknikler için "Advenced" seçeneği seçilir

- Bir sonraki ekran, oluşturulan sorguların test edilme ekranıdır. **Test Query** butonuna tıklanarak oluşturulan sorgunun sonucu elde edilebilir.



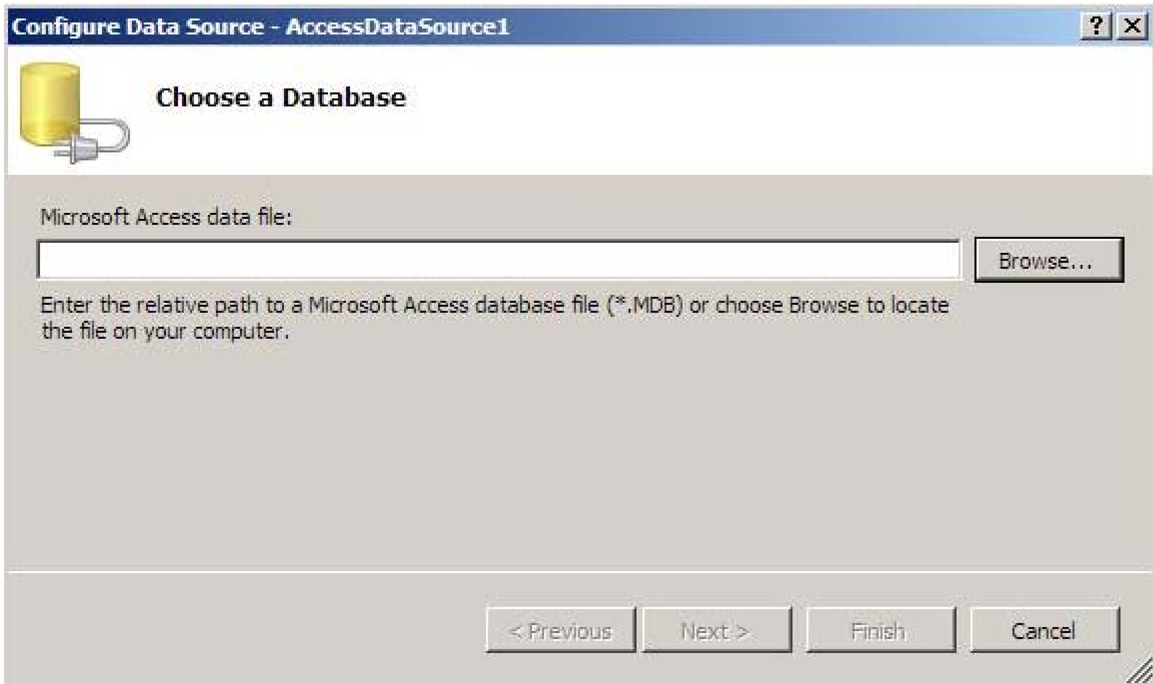
Şekil 320: Oluşturulan SqlDataSource test edilir

- **Finish** butonuna basıldığında artık veri kaynağını ayarlama işlemi bitmiş demektir.

AccessDataSource

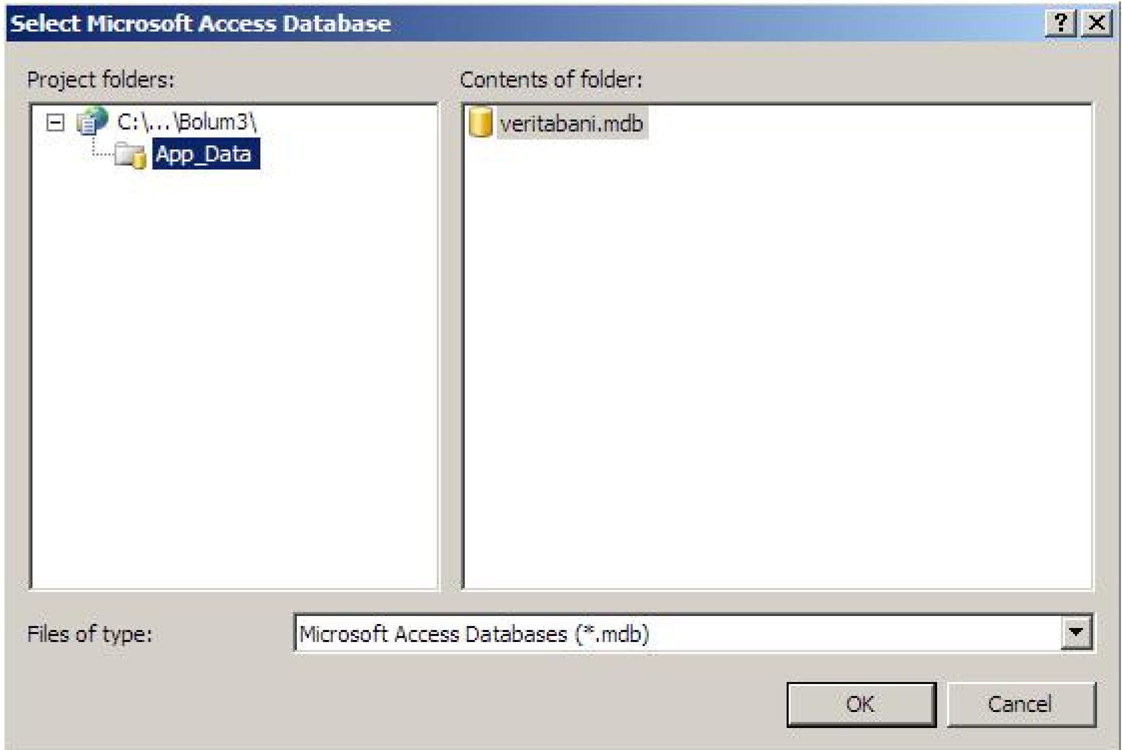
Uygulama içerisinde veritabanı olarak Access dosyası ile işlemler yapılmak istenirse, AccessDataSource kontrolü kullanılmalıdır. AccessDataSource kontrolü aracılığıyla, bir Access veritabanına bağlanılabilir ve veriler üzerinde işlemler yapılabilir. Sayfaya AccessDataSource kontrolü şu şekilde eklenebilir:

- Sayfaya bir tane AccessDataSource kontrolü eklenir ve eklenen kontrolün Smart Tag'ına tıklanır. Açılan pencereden **Configure Data Source...** bağlantısı seçilir.



Şekil 321: AccessDataSource kontrolünde "Configure Data Source" menüsü seçilir

- Açılan pencereden kullanılmak istenilen Access dosyası seçilir. Bunun için **Browse...** butonuna tıklanır.



Şekil 322 Bağlanılacak *.mdb Access veritabanı dosyası belirlenir

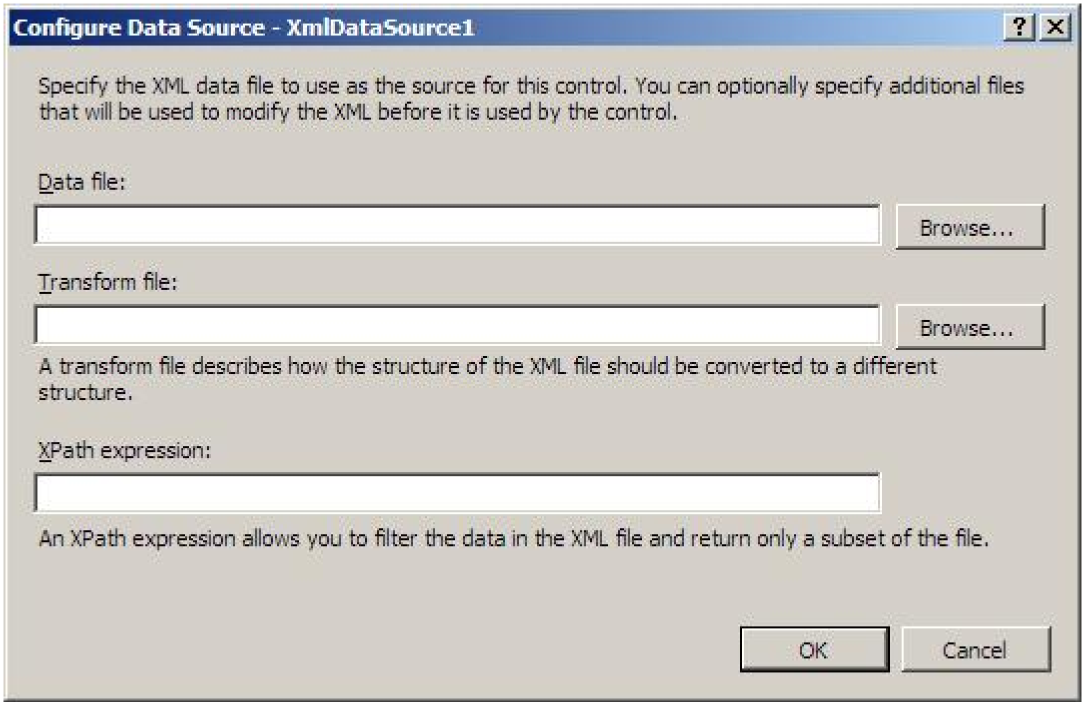
- Access dosyasının geliştirilen proje klasörü içerisinde yer alması gerekmektedir. Veri ile ilgili dosyaların ASP.NET'te App_Data klasörü içerisinde tutulması daha uygun olacağı için dosyayı daha önceden bu klasöre eklemek iyi olacaktır. Dosyayı seçtikten sonra **OK** butonuna tıklanır.

- Son olarak **Next** butonuna tıklanarak. SqlDataSource başlığında anlatılan ekranlara dönülür. Bu aşamadan sonra yapılacak işlemler SqlDataSource kontrolünde yapılan işlemlerin hemen hemen aynısıdır.

XmlDataSource

Veri kaynağı olarak XML dosyası kullanıldığı durumlarda, **XmlDataSource** kontrolü kullanılabilir. Bir web sayfasına XmlDataSource şu şekilde eklenebilir:

- Sayfaya eklenen XmlDataSource kontrolünün Smart Tag'ına tıklanır. Açılan pencereden **Configure Data Source...** bağlantısı seçilir.



Şekil 323: XmlDataSource için kaynak dosyanın eklenmesi

- Veritabanı olarak kullanılacak XML dosyası en üstteki **Browse** butonuna tıklanınca açılan pencereden seçilir. XML dosyasının proje ile aynı klasörde olması gerekmektedir. Bu dosya da, veri ile ilgili dosyaların tutulacağı App_Data klasörü içerisinde tutulabilir. Dosyayı seçtikten sonra OK butonuna basarak işlem tamamlanır.
- Eğer XML dosyası ile beraber, o XML dosyasının yapısında bir değişiklik yapılması gerekiyorsa bunun kurallarını belirleyen, XSL (XML Transform Files) dosyası da kullanılmak isteniyorsa; ikinci seçim ekranından da XSL dosyası seçilir. Ayrıca aynı ekranda XML dosyasındaki verileri filtrelemek için gerekirse Xpath ifadesi de belirlenebilir.

Veri (Data) Kontrolleri

Veri kontrolleri, veri ile ilgili işlemlerin yapılmasını sağlayan, veritabanından alınan verinin, kullanıcıya esnek ve düzgün bir biçimde sunulmasını sağlayan kontrollerdir. ASP.NET 2.0 ile gelen yeni veri kontrolleri ile birlikte veri işlemlerinde büyük kolaylıklar sağlanmaktadır. Veri kontrolleri en basit anlamda verileri tekrarlı olarak göstermekten, en gelişmiş işlemler olan sıralama, sayfalama, veri ekleme, veri güncelleme ve veri silme gibi birçok işlemi yapabilmektedir. Başlıca veri kontrolleri şunlardır:

Repeater

Repeater kontrolü veri kontrollerinin en basitidir. Çalışma prensibi, kendisine bağlanan veriyi, verilen biçime göre tekrarlı bir şekilde görüntülemektir. Örneğin önceki bölümde oluşturulan SqlDataSource'u, Repeater'e bağlayarak veriler gösterilmek istenildiğinde şu adımlar takip edilebilir.

Repeater kontrolü sayfaya eklediğinde aşağıdaki gibi bir kod sayfanın Source kısmına eklenir.

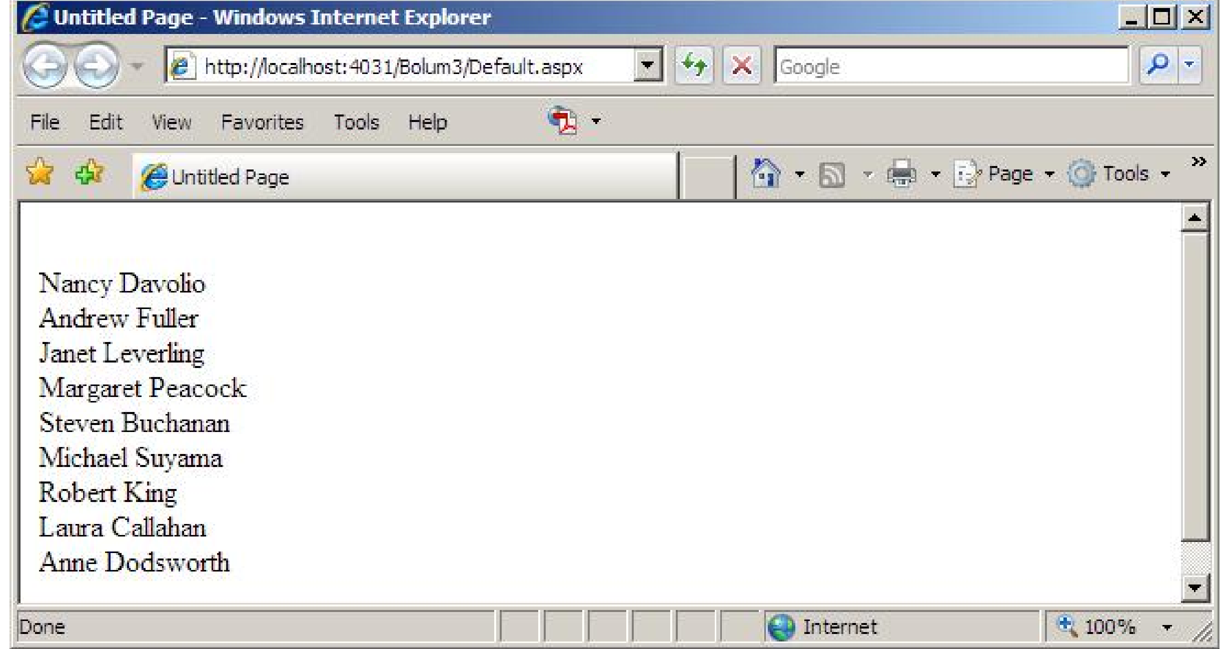
```
<asp:Repeater ID="Repeater1" runat="server"></asp:Repeater>
```

Bu kod üzerinde aşağıdaki değişiklikler yapılarak, verilerin alt alta yazılması sağlanmaktadır

```
<asp:Repeater ID="rpVeriler" runat="server" DataSourceID="sdsKaynak">
      <ItemTemplate>
            <%# Eval("FirstName")%> <%# Eval("LastName")%>
      </ItemTemplate>
      <SeperatorTemplate>
            <br>
      </ SeperatorTemplate>
</asp:Repeater>
```

Repeater kontrolüne **rpVeriler** ismini verip, **DataSourceID** özelliğine SqlDataSource veri kaynağının ID değeri atanarak bağlanmaktadır. Bu işlemden sonra, ItemTemplate tanımlayarak verilerin nasıl gösterileceği belirlenir. Burada FirstName ve LastName verileri yan yana gösterilecek, ardından da SeperatorTemplate kullanarak, her bir satır verinin birbirinden nasıl ayrılacağı belirlenmektedir. **
** bir HTML etiketi olup alt satıra geçilmesini sağlayacaktır.

<ItemTemplate>.....</ ItemTemplate> kısmı içerisinde görüntülenecek bir kaydın nasıl görüneceğini belirlerken, **<SeperatorTemplate>.....</ SeperatorTemplate>** kısmı içerisinde de görüntülenecek verilerin arasında nasıl bir kısmın yer alacağı belirleniyor. Bu iki kısım içerisinde HTML etiketleri ile birlikte birçok ASP.NET kontrolünü de kullanılabilir.



Şekil 324: Repeater kontrolü ile görüntülenen veriler

Bu işlemlerden sonra sayfa çalıştırıldığında Şekil 324'deki gibi bir sonuç elde edilir. Dikkat edilecek olursa veritabanında seçilen her kayıt bir satıra yazılmıştır.

DataList

DataList kontrolü, Repeater kontrolüne benzemekle beraber verileri sütunlar halinde de gösterebilme özelliği vardır. Daha önce sayfaya eklenen SqlDataSource veri kaynağı, bir DataList'e bağlanarak DataList'in kullanımı incelenebilir. Sayfaya bir DataList eklediğinde, Source kısmına aşağıdaki kodlar eklenecektir.

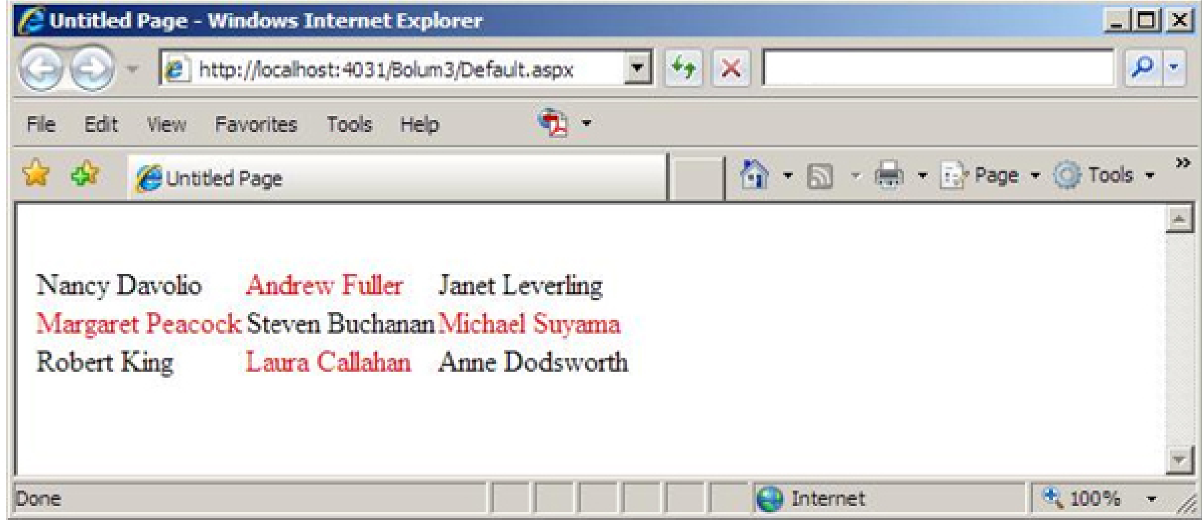
```
<asp:DataList ID="DataList1" runat="server"></asp:DataList>
```

Şimdi veritabanındaki Employees tablosunda bulunan isimleri 3 kolon halinde yan yana yazdıralbilmek için,

```
<asp:DataList ID="dlVeriler" runat="server" DataSourceID="sdsKaynak"
RepeatColumns="3" RepeatDirection="Horizontal">
            <ItemTemplate>
            <%# Eval("FirstName")%> <%# Eval("LastName")%>
      </ItemTemplate>
            <AlternatingItemTemplate>
            <font color="red"><%# Eval("FirstName")%> <%#
            Eval("LastName")%></font>
      </AlternatingItemTemplate>
</asp:DataList>
```

Kodları yazılmalıdır.

DataList'in adı dlVeriler olarak değiştirildi ve DataSourceID özelliği sdsKaynak olarak belirlendi ve SqlDataSource'a bağlandı. **RepeatColumns** özelliği 3, **RepeatDirection** özelliği Horizantal olarak ayarlandı. **RepeatColumns** yanyana kaç kaydın geleceğini, **RepeatDirection** ise verilerin yanyana mı yoksa alt alta mı geleceğini belirler. Bu örneğin sonucunda üç sütun içerisine veriler yanyana getirilecektir. **<ItemTemplate>.....</ ItemTemplate>** kısmında verilerin nasıl gösterileceği ayarlanıyor.**<AlternatingItemTemplate>.....</AlternatingItemTemplate>** kısmında ise; alternatif veri gösterim şekli belirlenir. Böylece görüntülenecek verilerin biri ItemTemplate kısmındaki formatta, diğeri AlternatingItemTemplate kısmındaki formatta olacak şekilde sıralanacaktır.



Şekil 325: DataList kontrolüyle görüntülenen veriler

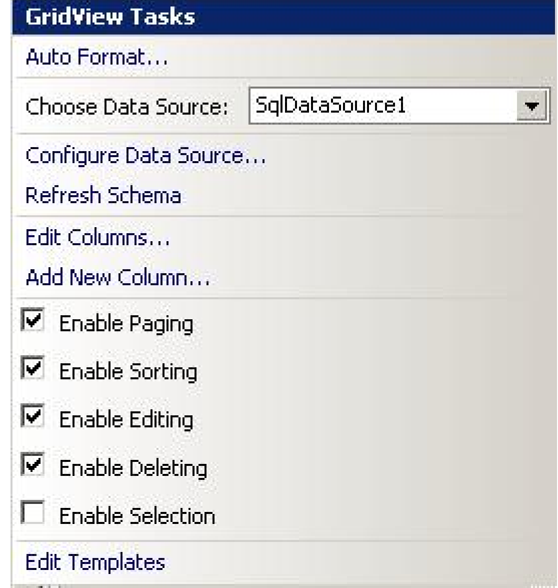
GridView

GridView, veri kontrolleri arasında en gelişmiş özelliklere sahip olan kontroldür. ASP.NET 1.x versiyonlarında bulunan DataGrid kontrolüne oranla gelişmiş birçok özelliği bulunmaktadır. Yine kullanım kolaylığı açısından da DataGrid'e göre daha pratik bir kullanıma sahiptir. GridView kontrolü, kendisine bağlanan veriler üzerinde listeleme, seçme, güncelleme ve silme imkanı sağlar. Bunun dışında verilerin liste halinde sayfalanması ve sıralanması gibi, kontrolü zor olan birçok işlemin çok basit bir şekilde yapılabilmesine olanak sağlayan özellikleri hazır olarak gelmektedir. Önceki örneklerde kullanılan SqlDataSource'u bu kez de GridView'a bağlayarak GridView'in nasıl kullanılacağını incelenecektir.

- Sayfaya eklenen GridView'ın üzerindeki akıllı etikete tıklanarak, Choose Data Source bölümünden daha önce eklenen sdsKaynak seçilebilir. Veri kaynağı seçildiği anda, GridView'ın şeklinde değişiklik olacaktır. Tekrar akıllı etikete tıkladığında açılan GridView Tasks ekranından Şekil 326'daki seçenekler seçilebilir olacaktır. Ve şekilde görüldüğü gibi ilgili seçimler yapılabilir.

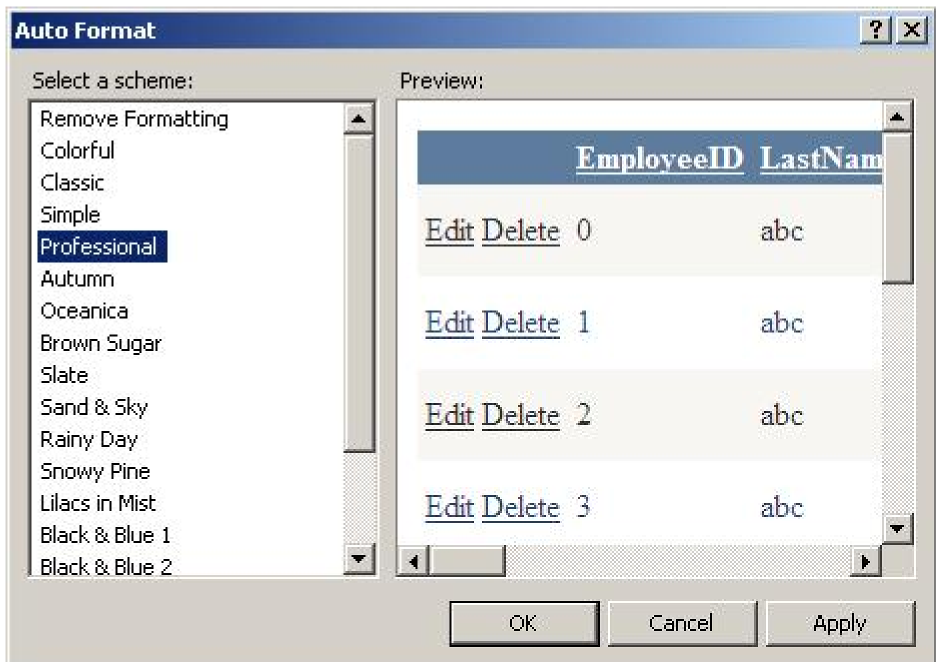
- Enable Paging (Sayfalama)
- Enable Sorting (Sıralama)
- Enable Editing (Güncelleme, Düzenleme)
- Enable Deleting (Silme)

Bu özellikler az sonra görüntülenecek olan veriler üzerinde sayfalama, sıralama, düzenleme ve silme gibi işlemlerin yapılabilmesini sağlayacaktır. Yine bu pencereden GridView üzerinde bazı ayarlamalar ve eklemeler yapılabilmektedir.



Şekil 326: Gridview Tasks penceresi

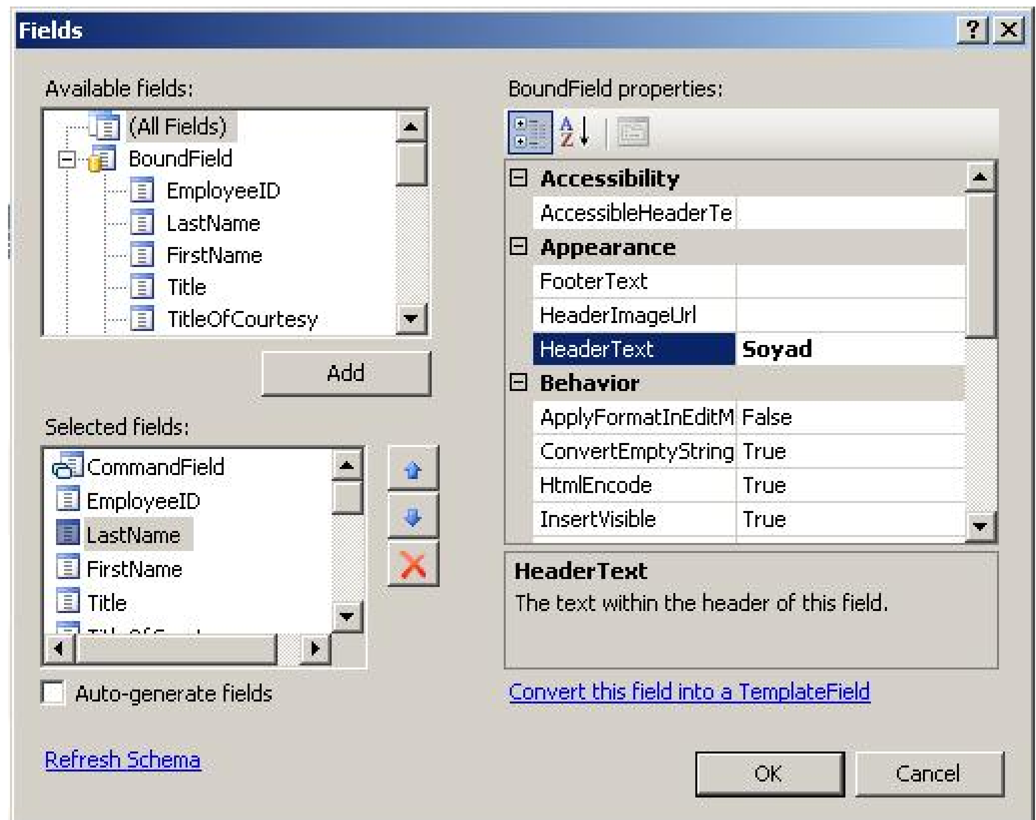
- Bu işlemleri yaptıktan sonra GridView üzerinde tüm satırlarda, Güncelleme (Edit) ve Sil (Delete) butonları görüntülenir. Yine akıllı etikete tıklayıp ve GridView Tasks ekranından Auto Format seçeneği seçilerek, varolan bazı şablonlar kontrol üzerine uygulanabilir ve GridView'in görselliğinde değişiklikler yapılabilir.



Şekil 327: Auto Format penceresinden GridView'in görünümü değiştirilebilir

ASP.NET'teki bazı kontrollerinin üzerindeki akıllı etiketlere tıklandığında **Auto Format** seçeneği çıkmaktadır. Bu kısımdan, önceden hazırlanmış olan görüntü formatlarından birisinin seçilmesi durumunda kontrolün içeriği otomatik olarak yeniden şekillendirilecektir. Calendar, DataList, GridView gibi kontrollerde Auto Format özelliği kullanılabilir.

- Temel işlemler ve görünüm değişiklikleri dışında GridView üzerinde bazı eklemeler ve çıkarmalar da yapılabilir. Örneğin GridView'de görüntülenmesi istenilmeyen alanları çıkarma ve sütun isimlerini değiştirme işlemleri şu şekilde gerçekleştirilebilir. GridView'in akıllı etiketine tıklayarak, GridView Tasks ekranından Edit Columns'a tıklanmalıdır.



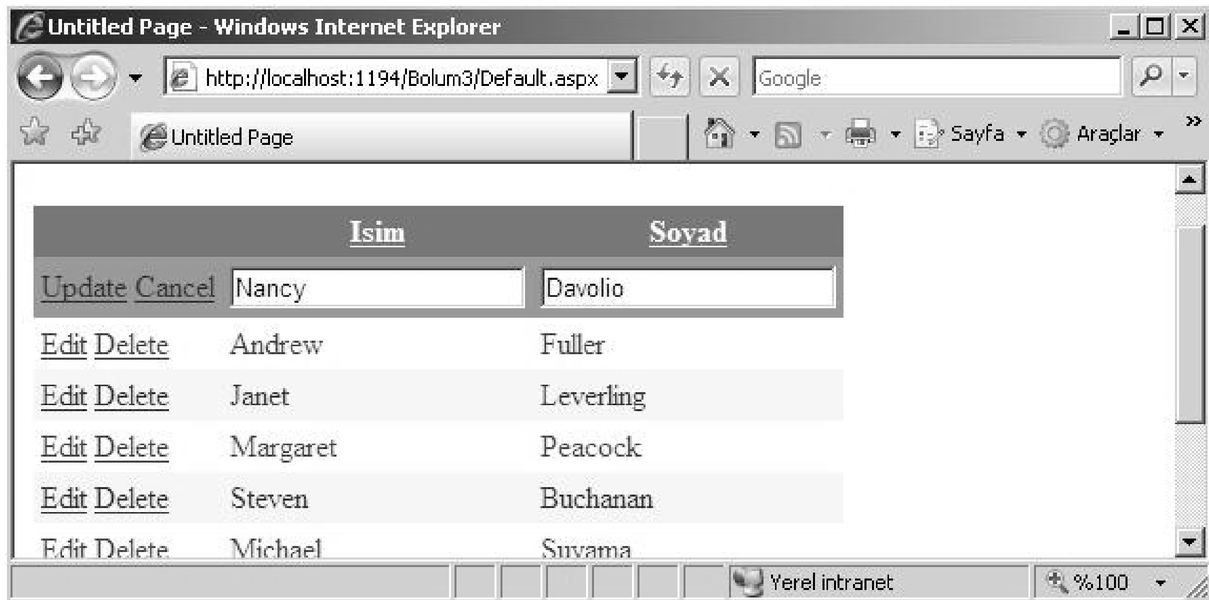
Şekil 328: GridView üzerindeki sütunlarda değişiklikler yapılabilir

- Açılan Fields ekranında, ekranda görülmesi istenilen alanlar; **Selected fields** bölümünde yer almalıdır. Görüntülenmek istenilmeyen alanlar bu kısımdan silinebilir. Yine gösterilmesi istenilen bir alan Available fiels pencesi içerisinden bu kısma eklenebilir. Selected fields kısmında seçilen bir alan ile ilgili bilgiler sağ kısımdaki **BoundField properties** penceresinde görüntülenir. Buradan yapılacak ayarlamalarla görüntülenecek alan üzerinde değişiklikler yapılabilir. Örneğin, görüntülenecek bir sütunun başlığı HeaderText kısmından değiştirilebilir.

Yapılan işlemlerden sonra örneği çalıştırarak GridView içerisinde verilerin nasıl görüntüleceğine bakılabilir.

Şekil 329: GridView kontrolü ile veriler görüntülenir

Şekil 329'da da görüldüğü gibi veriler, GridView üzerinde tablo içerisinde renklendirilmiş olarak görüntülenir. Herhangi bir kaydın yanındaki **Edit** butonuna tıklanırsa, seçilen verinin güncellenebilir hali görüntülenir.



Şekil 330: GridView kontrolü üzerindeki Edit butonu ile seçilen veriler güncellenebilir (Update işlemi)

Edit butonu ile seçilen kaydın üzerindeki verileri değiştirdikten sonra **Update** butonuna tıklanırsa; veriler otomatik olarak güncellenecektir. **Delete** tuşu ise bulunduğu sıradaki kaydın veritabanından silinmesini sağlar.

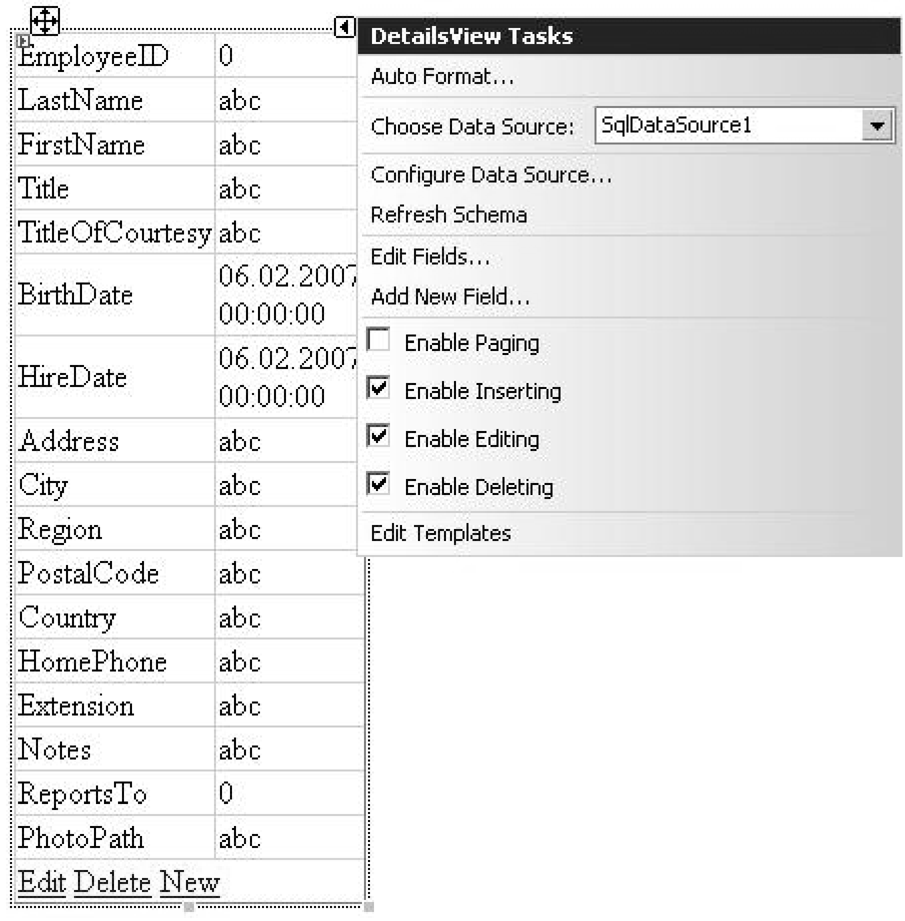
DetailsView

Şu ana kadar incelenen Repeater, DataList ve GridView kontrolleri genelde, tüm verilerin listelenmesi amacını taşımaktadır. DetailsView kontrolü diğer veri kontrollerden biraz daha farklı olarak, tek bir kaydın ayrıntılarını göstermektedir. Bir örnek ile sayfaya DetailsView kontrolü eklenerek bu kontrol incelenebilir.

Eklenen DetailsView'ın üzerindeki akıllı etikete tıklayarak **DetailsView Tasks** ekranı açılır ve bu ekrandaki, **Choose Data Source** kısmından sdsKaynak seçilir. Kontrol veri kaynağına bağlandığında DetailsView güncellenerek yeni alanlar görüntülenecektir. Bu işlemin ardından aynı pencerenin alt kısmından,

- Enable Inserting (Ekleme)
- Enable Editing (Düzenleme)
- Enable Deleting (Silme)

seçeneklerini seçilebilir olacak ve alanların seçilmesi ile birlikte ilgili kontrol bu işlemlere destek verir hale gelecektir.



Şekil 331: DetailsView kontrolünün veri kaynağına bağlanması ve üzerinde yapılacak işlemlerin seçilmesi

DetailsView'ın görünümünün ve görünmesi istenilen alanların düzenlenmesi işlemi de, aynı GridView'daki gibidir. Benzer işlemleri yapıp görünümünü aşağıdaki hale getirilebilir.

Şekil 332: DetailsView kontrolün çalışma zamanındaki görünümü

Dikkat edilecek olursa, DetailsView'in içerisinde sadece önceki veri listeleme kontrollerinde ilk sırada görünen kayıt görünüyor. Çünkü DetailsView veritabanındaki tabloda bulunan tek bir satırı gösterir. Bu nedenle DetailsView kontrolünün tek kayıt getiren ifadeler ile kullanılması daha mantıklı olacaktır.

GridView kontrolünde olduğu gibi **Edit** butonu düzenleme işlemine geçilmesini sağlarken, **Delete** butonu silme görevini yapacaktır. Ancak burada GridView'den farklı olarak **New** (Yeni) butonu bulunmaktadır. Bu butona tıkladığında DetailsView ekrana getireceği kontrollerle, yeni bir kaydın girilmesini sağlar.

Şekil 333: DetailsView üzerindeki New butonu ile yeni bir kayıt girişi yapılabilir

BÖLÜM 4: DURUM YÖNETİMİ (STATE MANAGEMENT)

ASP.NET'in genel mimarisi ve sağladığı sınırlı olanaklar, uygulamalarda durum yönetimi yapılmasını gerekli hale getirmektedir. İlk bölümde bahsedildiği gibi, HTTP alt yapısında sunucu kendisine ulaşan isteği cevapladıktan sonra, isteği yapan istemciye tekrar ulaşamaz. Teknik açıdan bakılacak olursa, bir .aspx sayfasına yapılan talep sırasında oluşturulan ilgili sınıf (class) tipinden nesne, sayfa istemciye gönderildikten sonra yok edilecektir. Aynı sayfaya yapılan başka bir talepte, nesne tekrar oluşturulur ve bu işlemler bu şekilde devam eder. Kullanıcının sunucu ile bağlantısının her cevaptan sonra kopması nedeniyle bir sayfadaki bilginin, diğer sayfaya taşınması mümkün olmayacaktır. Örneğin, bir web sitesinde A.aspx ve B.aspx isimli iki sayfa varsa, A.aspx sayfası içerisinden, B.aspx sayfasındaki bir değere veya nesneye ulaşmak, şu ana kadar ki bilgiler dahilinde mümkün olmayacaktır. Masaüstü uygulamalarda ise, programın içerisinde tanımlanan global bir değişken ve o değişkene atanan değer, program açık olduğu sürece program içerisindeki tüm formlar tarafından kullanılabilmektedir. Web üzerinde bu tip sorunları gidermek amacıyla durum yönetimi kavramı geliştirilmiştir.

Oturum Nesnesi ile Durum Yönetimi (Session)

Bir web sunucusu, kendisine ASP.NET sayfaları ile ilgili bir talep geldiği zaman, talebi yapan istemciye SessionID isiminde bir anahtar verir. Bu anahtar rastgele ve tekil olarak oluşturulan bir değerdir ve böylece SessionID değerleri hiçbir zaman çakışmazlar. Bir sayfa üzerinde oluşturulan SessionID'nin belirlenmiş bir ömrü vardır. Sayfaya istek geldikten sonra oluşturulan SessionID, o sayfa üzerinde belli bir süre tekrar işlem yapılmazsa ömrünü tamamladıktan sonra silinir. Ancak bu süre içerisinde, sayfaya istek gelmeye devam ederse SessionID yaşamaya devam eder.

Web sunucusu, SessionID değerleri ile kendisine gelen binlerce isteği birbirinden ayırabilir. Böylece istekleri birbirinden ayırdıktan sonra her isteği yönetebilir hale gelir. Bir kullanıcının bir tarayıcı penceresinde açmış olduğu ekrandan bir web uygulamasına yaptığı tüm talepler, tarayıcı penceresi kapatılana kadar veya üretilen SessionID değerinin ömrü sona erene kadar bir oturumu (session) ifade eder. Dolayısıyla her bir tarayıcı penceresinden bir web uygulamasına bağlı kalındığı sürece, sunucu ve kullanıcı arasındaki iletişimin tutarlılığı, bu SessionID tarafından sağlanır. SessionID'ye özel bilgiler sunucu belleğinde saklanabilir. Böylece sayfaya erişen her kullanıcıya özel sunucu belleğinde tutulan bilgiler, sayfalar arasında paylaştırılabilir. Örneğin, bir alışveriş sitesinde, alışveriş sepetine atılan ürünler, sayfalar arasında gezildiğinde kaybolmamakta ve yine alışveriş sepeti içerisinde yer almaktadır. Alışveriş sepetindeki ürünler, alışverişi yapan kullanıcıya özel SessionID ile sunucunun belleğinde tutulurlar.

Bir ASP.NET uygulamasında, tüm oturum bilgileri **Session** adı verilen bir nesne tarafından tutulur. Session nesnesi içerisindeki bilgilere ulaşılabileceği gibi, istenirse Session nesnesine veri de atılabilir. Sitenin başka bir sayfasından Session içerisindeki tüm verilere ulaşılabilir. Bu özelliği Session nesnesini, sayfalar arasında veri taşıma ve belli durumlarda veri saklama işlemleri için çok uygun bir hale getirmektedir. Session nesnesinin bir diğer özelliği ise taşıdığı verilerin diğer kullanıcılar tarafından görüntülenememesidir. Böylece çok önemli bilgiler bile Session nesnesi içerisinde tutulabilir.

Session Nesnesine Veri Eklemek

Bir Session nesnesi içerisinde teorik olarak sonsuz veri tutabilir. Session nesnesine içerisine Object tipinden türeyen her türlü veri atılabilir; tam sayı(int), metin(string) veya bir nesne örneği gibi... Session ifadesi sonuna gelecek olan köşeli parantezler içerisine

yazılacak bir isim ile belirlenir. Belirlenen bu isimdeki session'a değer atılabilir. Session nesnesine değer atama işlemi şu şekilde yapılabilir.

```
Session["degiskenAdi"] = Alacagi_deger;
```

Session nesnesi kolleksiyon özelliğine de sahiptir. Yukarıdaki şekilde Session içerisine bir değer eklenebileceği gibi, Session sınıfına ait Add metodu aracılığıyla da Session nesnesi içerisine değer ataması yapılabilir. Aşağıda Add metodunun kullanım şekli yer almaktadır.

```
Session.Add("degiskenAdi", Alacagi_deger);
```

Bu iki kullanımı bir örnek içerisinde incelenecek olursa; oluşturulacak bir sayfaya **session_ekle.aspx** ismi verilerek içerisine bir TextBox kontrolü ile bir Button kontrolü eklenmelidir. Butonun Click olayı içerisine aşağıdaki ifadeleri yazılabilir.

```
protected void Button1_Click(object sender, EventArgs e)
{
      Session["isim"] = TextBox1.Text;
      Session.Add("yas", 25);
}
```

TextBox içerisinde bir değer girip butona tıklandığın da, Session nesnesi içerisine iki değer eklenecektir. Bunlar; TextBox1 kontrolü içerisine girilen değeri taşıyacak **isim** değeri ve rakamsal olarak 25 değerini taşıyacak olan **yas** değişkenidir. Bu şekilde bir Session nesnesi içerisine farklı değerler eklenebilir.

Session nesnelerin sunucunun belleğinde taşınmasından dolayı, bu nesnelere eklenecek olan verilerin uygulamaların performansını olumsuz yönde etkilememesi için, çok büyük boyutlara sahip olmaması önerilir.

Session Nesnesinden Veri Okumak

Session nesnesi içerisindeki veriler, uygulama içerisindeki her sayfadan okunabilir. Veri okuma işlemi, veri ekleme işlemine benzer. Önceki örnekte hazırlanan uygulama içerisine **session_oku.aspx** isimli yeni bir sayfa daha ekleyerek bu durum görülebilir. Bu sayfa içerisine ekleyenecek bir Label kontrolü içerisine, Session nesnesinden okunacak değerler yazılabilir. Sayfanın boş bir yerine fare ile çift tıklayarak veya fareye sağ tıklayıp View Code seçeneğinden sayfanın kod kısmına yani session_oku.aspx.cs dosyasına geçerek ilgili kodlar yazılabilir. Bu kısımda sayfanın Page_Load metodu içerisinde Session içerisindeki değerler Label kontrolüne yazdırılabilir.

```
protected void Page_Load(object sender, EventArgs e)
{
      Label1.Text = "İsim: " + Session["isim"].ToString() + "<br>Yaş: " +
Session["yas"].ToString();
}
```

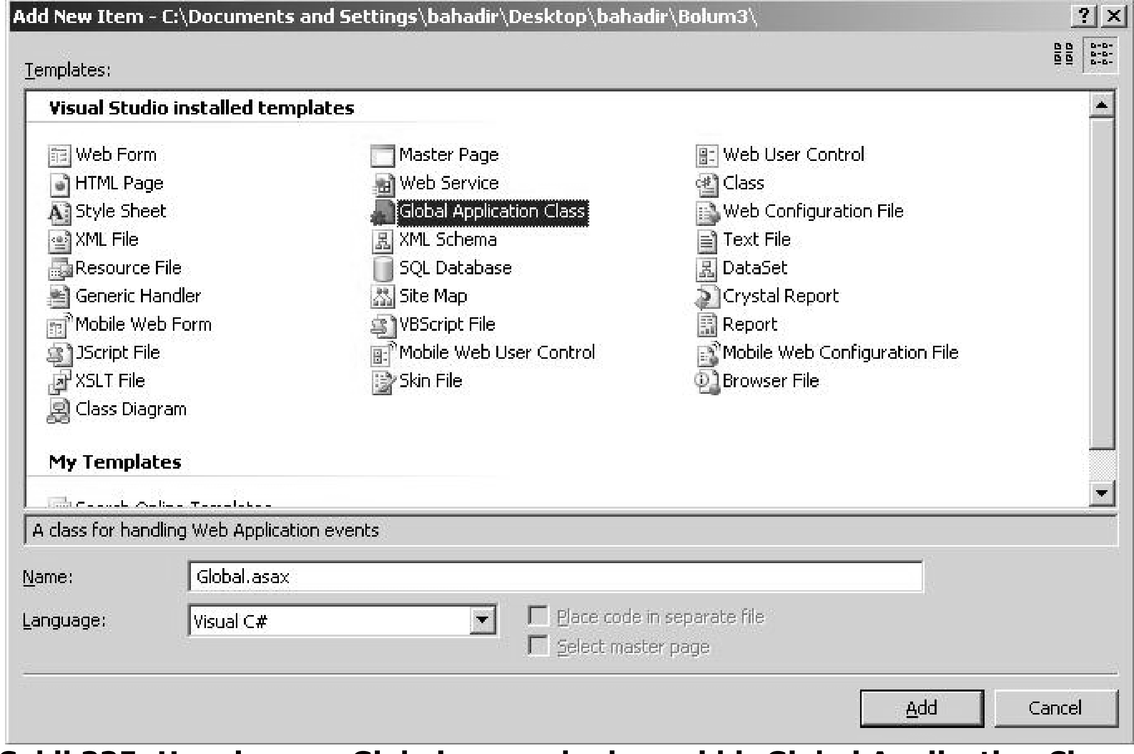
Bu örneğin düzgün bir şekilde çalışabilmesi için öncelikle **session_ekle.aspx** sayfasını çalıştırıp TextBox kontrolüne bir değer girip butona tıklamak gerekmektedir. Daha sonra tarayıcıdaki adres satırında, ilgili kısmı **session_oku.aspx** olarak değiştirip sayfa açıldığın da, Label kontrolü içerisine Session'daki değerlerin yazdırıldığı görülebilir.

Şekil 334: Session içerisindeki değerleri aynı uygulama içerisinden okuyabiliriz

Uygulama Nesnesi ile Durum Yönetimi (Application)

Bir ASP.NET uygulamasında, oturum bazında bilgilerin taşınabileceği gibi uygulama bazında da bilgiler tutulabilir ve taşınabilir. Oturum yönetimi, her kullanıcı için özel bilgiler taşınabilirken, uygulama yönetiminde hazırlanan ASP.NET bazında herkes için ortak değerler tutulup taşınabilmektedir. ASP.NET'te uygulama yönetimi **Application** adlı nesne ile yapılmaktadır. Application nesnesi, Session nesnesinden farklı olarak tüm kullanıcıları kapsayacak ve uygulamanın tümü için geçerli olacak şekilde bilgi saklama yeteneğine sahiptir. Böylece uygulama tümü ile ilgili bilgileri tutulabilir, gerektiği yerde bu bilgilere ulaşılabilir ve bu bilgilere göre işlemler yapılabilir. Örneğin, o an siteyi ziyaret eden ziyaretçilerin sayısı bir uygulama için genel olacak bir bilgidir, dolayısı ile Application nesnesinde tutulabilir.

Application nesnesini, ziyaretçi sayısını tutan bir örnek ile açıklanabilir. Bu işlem için öncelikle projeye **Global Application Class** (Global.asax) dosyası eklenmelidir.



Şekil 335: Uygulamaya Global.asax adında yeni bir Global Application Class dosyasının eklenmesi

Global.asax dosyası görsel arayüzü olmayan, sadece C# kodlarından oluşan bir dosyadır. Genel olarak uygulama ile ilgili

- **Uygulamanın başlaması**
- **Uygulamanın sonlanması**
- **Uygulamada hata oluşması**
- **Oturumun başlaması**
- **Oturumun bitmesi**

gibi olayları (events) kontrol eder. Aşağıda Global.asax dosyasının içeriğinin bir kısmı görünmektedir.

```
<%@ Application Language="C#" %>
<script runat="server">

    void Application_Start(object sender, EventArgs e)
    {
    }

    void Application_End(object sender, EventArgs e)
    {
    }

    void Application_Error(object sender, EventArgs e)
    {
    }

    void Session_Start(object sender, EventArgs e)
    {
    }

    void Session_End(object sender, EventArgs e)
    {
    }
</script>
```

Bu örnekte yapılmak istenen işlem ziyaretçi sayısını bulmak olacağı için; açılan her Session ziyaretçilerin sayısının bir artması, kapatılan her Session da ziyaretçi sayısının bir azalmasını ifade edecektir. Yani bu dosya içerisindeki **Application_Start**, **Session_Start** ve **Session_End** kısımları bu örnek için gerekli olan kısımlardır. Session_Start, sitede Session açıldığında yani her yeni browser açıldığında, Session_End ise Session kapatıldığında tetiklenen olaylar olacağı için, bu metodlar içerisine yazılacak ifadelerle, ziyaretçilerin anlık olarak sayıları bulunabilir. Bu işlem için Session'da olduğu gibi Application sınıfı içerisine de değer eklenebilecektir.

```
void Application_Start(object sender, EventArgs e)
{
      Application["ziyaretci"] = 0;
}
```

Görüldüğü gibi Application nesnesinin kullanımı da, Session nesnesine benzemektedir. Application_Start metodu içerisinde Application nesnesine, ziyaretçi adında bir değer eklendi ve bu değer başlangıç olarak 0'a eşitlendi. Bu metot sadece uygulama ilk çalıştığında tetiklenecektir. Bu kısımdan sonra siteye giriş yapacak her kullanıcı için açılacak olan Session üzerinden ziyaretçi sayısını bir arttırma işleminin yapılması gerekmektedir. Session_Start bu işlemin yapılacağı yerdir.

```
void Session_Start(object sender, EventArgs e)
{
      Application.Lock();
      Application["ziyaretci"] = (int)Application["ziyaretci"] + 1;
      Application.UnLock();
```

```
}
```

Bir Application verisini değiştirirken, Application nesnesi önce **Lock()** metodu çağrılarak kilitlenmelidir. Böylece Application nesnesi üzerinde aynı anda iki işlem yapılması olasılığına yani bir çakışma olasılığına karşı önlem alınmış olacaktır. Kilitleme işlemi tamamlandıktan sonra veri değiştirilmelidir. Veri değiştirme işlemi bittikten sonra **UnLock()** metodu çağrılarak kilit açılmalıdır.

Session ve Application nesneleri içerlerindeki değerleri **object** türünden tuttuğu için, bu nesnelerin taşıdığı değerler üzerinde tip bazlı olarak işlem yapılmak istendiğinde, tür dönüşümü yapılması gerekmektedir. **Session["degiskenAdi"].ToString()** veya **(int)Application["degiskenAdi"]** gibi...

Bu işlemleri tamamladıktan sonra, Session nesnesi kapatıldığında (ziyaretçi siteden çıkış yaptığında veya Session nesnesinin ömrü bittiğinde) zyaretçi sayısını azaltmak gerekecektir. Bu işlemi de **Session_End** metodu içerisinde yapmakdoğru olacaktır.

```
void Session_End(object sender, EventArgs e)
{
      Application.Lock();
      Application["ziyaretci"] = (int)Application["ziyaretci"] -  1;
      Application.UnLock();
}
```

Application nesnesindeki veri değiştirilmek istenildiğinde; önce Application nesnesi kilitlenip veri değiştirilir ve ardından kilit açılır. Veri üzerinde aritmetik bir işlem yapılacağı için yine object türünden int türüne çevirme işlemi yapılmalıdır. Bu örnek için Global.asax dosyasında yapılması gereken işlemler aslında temel olarak bu kadardır. Sitedeki ziyaretçi sayısını bir sayfa içerisinde görüntüleyebilmek için de, bir sayfa hazırlanması yeterli olacaktır.

Ziyaretçi sayısını hazırlayacağımız sayfada gösterebilmek için sayfaya bir Label kontrolü ekleyelim. Sayfanın Page_Load kısmında Application["ziyaretci"] değerine ulaşıp bu değeri Label içerisine yazdıralım.

```
protected void Page_Load(object sender, EventArgs e)
{
      Label1.Text = "Aktif ziyaretçi sayısı: " +
                     Application["ziyaretci"].ToString();
}
```

Oluşturulan sayfa çalıştırılarak sonuç görülebilir.

Şekil 336: Application nesnesi içerisindeki ziyaretçi sayısı ekrana yazdırıldı

Uygulama ilk kez çalıştığı için öncelikle ziyaretçi sayısı 0 olarak başlatılacaktır. Sayfanın açılmasıyla birlikte yeni bir oturum açılacağı için ziyaretçi sayısı 1 olacaktır. Bu sayfa kapatılmadan, tarayıcıdaki adres satırından adres kopyalanarak, tarayıcıda yeni bir ekran açılıp, adres satırına kopyalanan adres yapıştırılarak sayfada yeni bir tane daha açılırsa; ziyaretçi sayısının 2 olarak arttığı görülecektir.

Şekil 337: Aynı uygulama içerisinde yeni bir sayfa açıldığı için ziyaretçi sayısı arttırıldı

Session nesnesinin ömrü varsayılan olarak 20 dakika içerisinde biteceği için, bu örnekte ziyaretçi sayısının düştüğünü görebilmek için bir sayfada 20 dakika işlem yapmamak gerekecektir. Application bazında, bu 20 dakikalık süre düşürülebilir ve değişiklik daha kısa sürede gözlemlenebilir. Fakat bu ileri düzeyde bir işlem olacağı için burada üzerinde durulmayacaktır.

KISIM SONU SORULARI:

1) Application ve Session nesneleri arasındaki temel farklar nelerdir? Araştırınız.

2) Masaüstü programlama ve web tabanlı programlama arasındaki temel farklar nelerdir? Araştırınız.

3) İstemci taraflı ve sunucu taraflı web programlama arasındaki temel farklar nelerdir?

4) Html, Html Server ve Web Server kontrolleri arasındaki farklar nelerdir? Araştırınız.

5) Sayfalar arası geçiş için kaç yol vardır? Bu yollar arasındaki farklar nelerdir? Araştırınız.

6) Basit bir Forum uygulaması yazmaya çalışınız.

7) Sayı tahmin oyunu yazınız.

8) Windows kısım sorularında yer alan birinci sorunun Web tabanlı versiyonunu yazmaya çalışınız.

9) Global.asax dosyası içerisindeki olay metodları hangi durumlarda çalışır. Araştırıp örnekler ile ispatlayınız.

10) Bir sayfada oluşturulan ve Personel isimli bir sınıfa ait nesne örneği içerisinde tutulan verileri, diğer bir sayfaya Session nesnesi yardımıyla taşımaya çalışınız.
(Yardımcı Not: Sayfalar arası hareket için Response sınıfına ait static Redirect metodu kullanılabilir.)