12. BÖLÜM:
Dizin ve Dosya İşlemleri

C# programlama dilinde, dizin ve dosya işlemlerini gerçekleştirebileceğimiz tüm metotlar ve özellikler "$System.IO$" kütüphanesi içinde yer alır. Bu kütüphanedeki dizin ($Directory$) ve dosya ($File$) sınıflarını kullanarak, klasör oluşturma, silme, taşıma ya da bir dosyayı okuma, yazma, kopyalama vb. birçok işlemi gerçekleştirmeniz mümkündür. Bununla birlikte, dizin ve dosyalar hakkında (oluşturma tarihi, son değişiklik tarihi, boyutu, uzantısı, yolu vb.) bilgi edinmek için $DirectoryInfo$ ve $FileInfo$ sınıflarını kullanabiliriz. Dosya veya dizinlerle ilgili bir işlem yapacağımız zaman bu kütüphaneyi "$using\ System.IO$;" projemize eklememiz gerekmektedir.

12.1. Dizin ve Dosya Sınıfları

Bu başlık altında $Directory$, $DirectoryInfo$, $File$ ve $FileInfo$ sınıflarının kullanımı hakkında açıklamalar ve bunlarla ilgili uygulamalı örnekler bulunmaktadır.

12.1.1. Directory Sınıfı

$Directory$ ve $File$ sınıfları "System.IO" kütüphanesi içerisinde yer alır. Dolayısıyla bu sınıfları kullanabilmek için, programımıza $using\ System.IO$; satırını eklemeliyiz. $Directory$ ve $File$ sınıflarında statik metotlar bulunur. Bu iki sınıfın birçok metodu birbirine benzerdir. Farklı olan tarafı ise $Directory$ sınıfı klasörler üzerinde işlem yaparken $File$ sınıfı dosyalar üzerinde işlem yapar. Bu metotlara örnek olarak $Delete$, $Exists$ ve $Move$'u verebiliriz.

$Directory.CreateDirectory$ metodu, Windows işletim sisteminin güvenliği ile belirtilen adrese bir dizin veya klasör oluşturmamıza imkan sağlar. Ayrıca bu yöntemle uzaktaki bir bilgisayarda da bir dizin oluşturabilirsiniz.

Aşağıdaki kod parçacığı, C:\ sürücüsü üzerinde $Elenium$ adında bir klasör olup olmadığını kontrol eder. Eğer böyle bir dizin yoksa $Elenium$ adında yeni bir klasör oluşturur. Burada Windows, C:\ sürücüsü üzerinde bir dizin oluşturulurken kullanıcı hesabının yönetici izinlerine sahip olmasını istemektedir.

```
using System;
using System.IO;

public class DizinIslemleri
{
    static void Main(string[] args)
    {
        string dizin = @"C:\Elenium";

        // Böyle bir dizinin olup olmadığını kontrol eder.
        bool dizinVarmi = Directory.Exists(dizin);

        if (!dizinVarmi) // Eğer böyle bir dizin yoksa;
        {
            // Yeni bir dizin oluştur.
            Directory.CreateDirectory(dizin);
        }
    }
}
```

$Directory.CreateDirectory$ metodu ile ayrıca bir alt dizin veya alt klasör oluşturabilmemiz de mümkündür. Tek yapmanız gereken, bu alt dizinin oluşturulacağı klasörün yolunu belirtmektir. Aşağıdaki kod parçacığı, C:\$Elenium$ dizini içerisinde $Framework$ isimli bir alt dizin oluşturur.

```
using System;
using System.IO;

public class DizinIslemleri
{
    static void Main(string[] args)
    {
        string altDizin = @"C:\Elenium\Framework";

        // Böyle bir dizinin olup olmadığını kontrol eder.
        bool altDizinVarmi = Directory.Exists(altDizin);

        if (!altDizinVarmi) // Eğer böyle bir dizin yoksa;
        {
            // Yeni bir dizin oluştur.
            Directory.CreateDirectory(altDizin);
        }
    }
}
```

$Directory.Delete$ metodu, belirtilen adresteki boş bir klasörü kalıcı olarak siler. Bu klasörde alt klasörler ve/veya dosyalar varsa, önce bunları silmelisiniz. Aksi halde bir hata mesajı alırsınız.

```
using System;
using System.IO;

public class DizinIslemleri
{
    static void Main(string[] args)
    {
        string dizin = @"C:\Elenium";

        if (Directory.Exists(dizin)) // Eğer böyle bir dizin varsa;
        {
            Directory.Delete(dizin); // Dizin sil.
        }
    }
}
```

$Directory.Move$ metodu, mevcut bir klasörü tam yolu belirtilen yeni bir dizine taşır. Move metodu, kaynak dizin adresi ve hedef dizin adresi olmak üzere iki parametre alır. Bu yöntem, orijinal dizini siler ve tüm içeriği ile birlikte hedeflenen adrese taşır.

```
using System;
using System.IO;

public class DizinIslemleri
{
    static void Main(string[] args)
    {
        string kaynakDizin = @"C:\Elenium";
        string HedefDizin = @"D:\newElenium";

        if (Directory.Exists(kaynakDizin)) // Eğer böyle bir dizin varsa;
        {
            // Dizini ve içindekileri taşır, ismini değiştirir.
            Directory.Move(kaynakDizin, HedefDizin);
        }
    }
}
```

$Directory$ sınıfının $GetDirectories$ metodu, belirtilen bir dizin içerisindeki tüm alt klasörlere erişebilmemizi sağlar. Tüm alt dizinlere erişebilmek için özyinelemeli ($recursive$) metotlar kullanabiliriz.

```
using System;
using System.IO;

public class DizinIslemleri
{
```

```
 6      static void Main(string[] args)
 7      {
 8          string dizin = @"C:\Elenium";
 9
10          if (Directory.Exists(dizin)) // Eğer böyle bir dizin varsa;
11          {
12              // Geçerli dizine ait alt dizinleri bulur.
13              string[] dizinler = Directory.GetDirectories(dizin);
14
15              foreach (string altDizin in dizinler)
16                  AltDizinleriGetir(altDizin);
17          }
18      }
19
20      Private static void AltDizinleriGetir(string dizin)
21      {
22          Console.WriteLine(dizin);
23          string[] altDizinler = Directory.GetDirectories(dizin);
24
25          foreach (string altDizin in altDizinler)
26          {
27              AltDizinleriGetir(altDizin);
28          }
29      }
30  }
```

Yukarıdaki örneğimizde, $GetDirectories$ metodu ile elde ettiğimiz alt dizinlerin her biri için sahip olduğu tüm alt dizinlerini listelemek için $foreach$ döngüsü altında $AltDizinleriGetir$ isimli metodumuzu çağırıyoruz. Bu metodumuzu özyinelemeli olarak oluşturmamızın sebebi tüm alt dizinlere erişebilmek içindir. Son olarak, $GetFiles$ metodu ile belirtilen bir dizin içerisindeki dosyaların listesini gösteren bir örnek yapalım.

```
 1  using System;
 2  using System.IO;
 3
 4  public class DizinIslemleri
 5  {
 6      static void Main(string[] args)
 7      {
 8          string dizin = @"C:\Elenium";
 9
10          // Geçerli dizindeki dosyaları getirir.
11          string[] dosyaListesi = Directory.GetFiles(dizin);
12
13          foreach (string dosya in dosyaListesi)
14          {
15              Console.WriteLine(dosya);
16          }
17      }
18  }
```

Yukarıdaki örnekte, "$C:\Elenium$" dizini içerisindeki tüm dosyalar konsola yazdırılmaktadır. Bu kodu test etmek için sizlerde kendi oluşturduğunuz bir dizin içerisine dosya oluşturabilirsiniz

ya da mevcut dosyalarınızdan kopyalayabilirsiniz. Buraya kadar olan kısımda $Directory$ sınıfına ait yer vermediğimiz diğer metotları da tek bir örnek üzerinde açıklayalım;

```
using System;
using System.IO;

public class DizinIslemleri
{
    static void Main(string[] args)
    {
        string dizin = @"C:\Elenium";

        // Belirtilen dizinin kök dizinini döndürür.
        var kokDizin = Directory.GetDirectoryRoot(dizin);
        Console.WriteLine(dizin); // Konsola C:\ yazar.

        // SetCurrentDirectory, belirtilen dizini geçerli dizin olarak ayarlar.
        Directory.SetCurrentDirectory(dizin);
        // GetCurrentDirectory, geçerli dizini döndürür.
        var gecerliDizin = Directory.GetCurrentDirectory();
        Console.WriteLine(gecerliDizin);

        // GetLogicalDrives, sistemdeki tüm mantıksal sürücüleri listeler.
        string[] suruculer = Directory.GetLogicalDrives();
        foreach (string surucu in suruculer)
            Console.WriteLine(surucu);
    }
}
```

12.1.2. DirectoryInfo Sınıfı

$DirectoryInfo$ sınıfı, $Directory$ sınıfının aksine $static$ olmayan metot ve özellikleri içerir. $Directory$ sınıfı ile dizin üzerinde işlem yaparken, $DirectoryInfo$ sınıfı dizin özellikleri hakkında bilgiler verir. Şimdi bir örnekle "C:\$Elenium$" dizinine ait özelliklerini gösteren kodumuzu yazalım;

```
using System;
using System.IO;

public class DizinIslemleri
{
    static void Main(string[] args)
    {
        string dizin = @"C:\Elenium";
        var dir = new DirectoryInfo(dizin);

        Console.WriteLine("Özellikler: {0}", dir.Attributes);
        Console.WriteLine("Oluşturulma tarihi: {0}", dir.CreationTime);
        Console.WriteLine("Dizin var mı? {0}", dir.Exists);
        Console.WriteLine("Uzantı: {0}", dir.Extension);
        Console.WriteLine("Tam adres: {0}", dir.FullName);
        Console.WriteLine("Son erişim zamanı: {0}", dir.LastAccessTime);
        Console.WriteLine("Son değişiklik zamanı: {0}", dir.LastWriteTime);
```

```
        Console.WriteLine("Dizin adı: {0}", dir.Name);
        Console.WriteLine("Bir üst dizin: {0}", dir.Parent);
        Console.WriteLine("Kök dizin: {0}", dir.Root);
    }
}
```

Yukarıdaki örnekte görüldüğü üzere, "$C:\Elenium$" dizinine ait bazı özellikler konsola yazdırılmıştır. Dizinlerle ilgili yapacağımız işlemler öncesinde, klasör hakkında bilgi edinmek için $DirectoryInfo$ sınıfından yararlanabiliriz.

12.1.3. File Sınıfı

$File$ sınıfı, C# programlama dilinde dosya işlemleri için kullanılmaktadır. Bu sınıf ile yeni bir dosya oluşturabilir, bir dosyayı kopyalayıp taşıyabilir veya silebilirsiniz. Ayrıca akışları ($stream$) okumak ve yazmak için kullanılan $FileStream$ ile birlikte birçok dosya işlemi için statik yöntemler sağlar. Aşağıdaki tabloda File sınıfının en sık kullanılan metotları ve açıklamaları gösterilmektedir.

Tablo 1. $File$ sınıfının yaygın olarak kullanılan metotları.

Metot	Açıklamalar
Create	Belirtilen adreste (yol) bir dosya oluşturmak için kullanılır.
Open	Belirtilen konumdaki bir dosyayı açmak için kullanılır. Bu metodun geri dönüş tipi $FileStream$ sınıfıdır.
Copy	Bir dosyayı belirtilen yere kopyalamak için kullanılır.
Move	Belirtilen dosyayı yeni bir konuma taşır. Ayrıca yeni konumdaki dosya için farklı bir isim verilebilmektedir.
Delete	Bir dosyayı silmek için kullanılır.
Exists	Belirtilen dosyanın var olup olmadığını belirler.
OpenRead	Var olan bir dosyayı okumak için açar.
OpenWrite	Mevcut bir dosyayı açar veya yazmak için yeni bir dosya oluşturur.

Konuyla ilgili ilk örneğimizde, $Create$ metodunu kullanarak yeni bir dosya oluşturalım. Ancak bunun öncesinde oluşturacağımız dosya için aynı isimde başka bir dosya olup olmadığını kontrol etmemizde yarar vardır. Ayıca dosyalama ilgili geliştirdiğiniz kodları hata yakalama blokları arasında kullanmanız tavsiye edilir. Böylece muhtemel hataların önüne geçerek, programınızın akışını sağlıklı bir şekilde sürdürebilirsiniz.

```
using System;
using System.IO;

```

```
 4  public class DosyaIslemleri
 5  {
 6      static void Main(string[] args)
 7      {
 8          string dosya = @"C:\CSharp.txt";
 9
10          try
11          {
12              if (!File.Exists(dosya)) // Böyle bir dosya yoksa;
13              {
14                  File.Create(dosya); // Bu dosyayı oluştur.
15                  Console.Write("Dosya başarıyla oluşturuldu.");
16              }
17          }
18          catch
19          {
20              Console.Write("Dosya oluşturma hatası meydana geldi!");
21          }
22      }
23  }
```

Yukarıdaki örneğimizde, "$File.Create$" metodunu kullanarak "$C:\$" dizininde "$CSharp.txt$" adında bir metin dosyası oluşturduk. Bunun için önce "$File.Exists$" metodu ile böyle bir dosya olup olmadığını kontrol ettik ve eğer böyle bir dosya yoksa oluşturulmasını sağladık. Bu esnada herhangi bir hata meydana gelirse, konsola dosya oluşturma hatasını veren bir mesaj görüntülenecektir. Şimdi oluşturduğumuz dosyayı nasıl sileceğimizi görelim;

```
 1  using System;
 2  using System.IO;
 3
 4  public class DosyaIslemleri
 5  {
 6      static void Main(string[] args)
 7      {
 8          string dosya = @"C:\CSharp.txt";
 9
10          try
11          {
12              if (File.Exists(dosya)) // Böyle bir dosya varsa;
13              {
14                  File.Delete(dosya); // Bu dosyayı sil.
15                  Console.Write("Dosya başarıyla silindi.");
16              }
17          }
18          catch
19          {
20              Console.Write("Dosya silme hatası meydana geldi!");
21          }
22      }
23  }
```

Yazmış olduğumuz kodu, bir önceki örneğimizle karşılaştırdığımızda, "$File.Create$" metodu yerine "$File.Delete$" yazıldığı görülmektedir. Ayrıca böyle bir dosya olup olmadığını kontrol

ederken kullandığımız "$File.Exists$" metodunun önünde ünlem işareti olmadığına dikkat ediniz. Hatırlanacağı üzere, bu işaretin (!) anlamı boolean olarak tersini (değili) almaktı.

Şimdiki örneğimizde ise dosya kopyalama ($File.Copy$) ve taşıma ($File.Move$) işlemlerini yapan kodumuzu yazalım;

```
 1  using System;
 2  using System.IO;
 3
 4  public class DosyaIslemleri
 5  {
 6      static void Main(string[] args)
 7      {
 8          string dosyaOrijinal = @"C:\CSharp.txt";
 9          string dosyaKopyasi = @"C:\CSharp-Kopyalanan.txt";
10          string dosyaTasinan = @"D:\CSharp-Tasinan.txt";
11
12          try
13          {
14              if (File.Exists(dosyaOrijinal)) // Böyle bir dosya varsa;
15              {
16                  // Dosyanın bir kopyasını oluştur.
17                  File.Copy(dosyaOrijinal, dosyaKopyasi);
18
19                  // Orijinal dosyayı başka bir konuma taşı.
20                  File.Move(dosyaOrijinal, dosyaTasinan);
21              }
22          }
23          catch
24          {
25              Console.Write("Dosya silme hatası meydana geldi!");
26          }
27      }
28  }
```

Yukarıdaki örneğimizde, mevcut bir dosyanın tam yolunu bildiren $dosyaOrijinal$ isimli değişkeni kullanarak, kopyalama yapacağımız konumu ve dosya adını $dosyaKopyasi$ ve başka bir yere taşıyacağımız dosyanın tam yolunu ise $dosyaTasinan$ isimli değişkenlere atadık. Daha sonra kaynak kodumuzun 17. satırındaki gibi "$File.Copy$" metodu ile orijinal dosyamızın bir kopyasını aynı dizin üzerinde oluşturduk. Dosya taşıma işlemi için ise yine kaynak kodumuzun 20. satırında gösterildiği gibi "$File.Move$" metodu ile orijinal dosyamızı D sürücüsüne taşıdık.

Şu ana kadarki olan kısımda, $File$ sınıfı ile dosya oluşturma, kopyalama, taşıma, silme ve belirtilen konumdaki bir dosyanın olup olmadığının kontrolü üzerinde durduk. Dosya açma, okuma ve yazma işlemleri ile ilgili örneklerimizi ilerleyen bölümlerde farklı yöntemler kullanarak açıklayacağız.

12.1.4. FileInfo Sınıfı

$FileInfo$ sınıfı, statik yapıdaki $File$ sınıfı ile aynı işlevlere sahiptir. Ancak bu sınıfı kullanarak, bir dosyadaki verileri okuma/yazma işlemleri üzerinde daha fazla denetime sahip olursunuz. $Directory$ ve $File$ sınıfları klasör ve dosyalar üzerinde işlemler yaparken sadece dosya yolunu kullanırlar. Örneğin bir klasördeki tüm dosyaları listeleyen metodu çağırdığınızda size sadece dosyaların yolunu ($path$) döndürür. $DirectoryInfo$ ve $FileInfo$ sınıfları ise, dosya yolunun haricinde dosya adı, uzantısı, boyutu, oluşturulma zamanı gibi ekstra bilgiler verir. Bu iki sınıf statik değildir ve bir nesne üzerinden erişilebilir.

```
using System;
using System.IO;

public class DosyaIslemleri
{
    static void Main(string[] args)
    {
        string dosya = @"D:\CSharp.txt";
        var fInfo = new FileInfo(dosya);

        // Dosya oluşturulma tarihi.
        DateTime olusturmaTarihi = fInfo.CreationTime;
        // İçinde bulunduğu dizin bilgisi.
        DirectoryInfo dizinBilgisi = fInfo.Directory;
        // İçinde bulunduğu dizinin adı.
        string dizinAdi = fInfo.DirectoryName;
        string dosyaUzantisi = fInfo.Extension; // Dosya uzantısı.
        string dosyaYolu = fInfo.FullName; // Dosyanın yolu.
        long dosyaBoyutu = fInfo.Length; // Dosyanın byte cinsinden boyutu.
        string dosyaAdi = fInfo.Name; // Dosyanın adı.
    }
}
```

12.2. Dosya Okuma/Yazma Sınıfları

C# programlama dilinde, dosya işlemleri için farklı yöntemler bulunmaktadır. Bu yöntemleri kullanarak; yeni bir dosya oluşturma, mevcut dosyaya yazma ve bir dosyayı okuma işlemlerini gerçekleştirebilirsiniz.

12.2.1. FileStream

$FileStream$ sınıfı, dosyaların okuma ve yazma gibi temel işlemini gerçekleştirmek için kullanılır. $FileStream$ sınıfını kullanabilmek için, $System.IO$ kütüphanesini eklemeniz ve ardından yeni bir dosya oluşturmak ya da var olan bir dosyayı açmak için $FileStream$ sınıfının bir örneğini oluşturmanız gerekir.

$FileStream$ sınıfının farklı sürümlerde olan kurucu metotları bulunur. Şimdi bu kurucu metotların gereksinim duyduğu önemli parametreleri inceleyelim; $FileStream$ sınıfının kurucu metotları $enum$ tipinde olan $FileMode$, $FileAccess$ ve $FileShare$ parametreleri kullanır. Bunlar üzerinde çalışılan bir dosyanın ne şekilde açılacağını, üzerinde ne şekilde işlem yapılacağını ve dosyanın başka işlemlerle ortak kullanım durumlarını belirler.

$FileMode$ ile üzerinde çalışacağımız bir dosyanın hangi modda açılması gerektiğini belirleriz. Aşağıdaki tabloda ayrıntılı olarak anlatıldığı şekilde, dosyanıza veri eklemek, üzerine yazmak, içeriğini silmek ya da yeni dosya oluşturmak için bu modları kullanabilirsiniz.

Özellik	Açıklamalar
Append	Açılan bir dosyanın sonuna veri eklemek için kullanılır. Eğer dosya bulunmazsa oluşturulur.
Create	Yeni dosya oluşturmak için kullanılır. Zaten dosya varsa üzerine yazılır.
CreateNew	Yeni bir dosya oluşturmak için kullanılır, belirtilen dosya mevcutsa çalışma zamanı hatası verdirir.
Open	Bir dosyayı açmak için kullanılır.
OpenOrCreate	Belirtilen dosya varsa açılır, yoksa yenisi oluşturulur.
Truncate	Belirtilen dosya açılır ve içi tamamen silinir.

$FileMode$ kullanarak $FileStream$ sınıfının bir örneğini oluşturmak için aşağıda gösterildiği gibi kurucu fonksiyona dosyanın yolunu ve hangi modda açacağımızı belirtiyoruz.

```
// Kullanım şekli;
FileStream fs = new FileStream(string path, FileMode mode);

// D dizininde bulunan CSharp.txt dosyasını açmak için;
string dosyaYolu = @"D:\CSharp.txt";
FileStream fs = new FileStream(dosyaYolu, FileMode.Open);
```

$FileAccess$ ile bir dosyaya erişim modunu ayarlayabilirsiniz. Bir dosya üzerinde çalışırken, ihtiyaç duyulandan daha fazla ya da gereğinden az erişim yetkisi vermek tavsiye edilmez. Bir dosyadan okumak istediğinizde $Read$ ve bir dosyaya yazarken $Write$'ı seçebilirsiniz. Ancak, $Read$ erişimini belirlerseniz ve daha sonra dosyaya yazmaya çalışırsanız, bir hata mesajı alacağınızı unutmayın. Aynı şey $Write$ erişimi belirlediğinizde ve dosyayı daha sonra okumayı denediğinizde de geçerlidir. Aşağıdaki tabloda dosya erişim modları ve bunlara ait açıklamalar gösterilmektedir.

Özellik	Açıklamalar
Read	Erişilen dosya için sadece okuma yetkisi verilir.

ReadWrite	Erişilen dosya için hem okuma hem de yazma yetkisi verilir.
Write	Erişilen dosya için sadece yazma yetkisi verilir.

$FileAccess$ kullanarak $FileStream$ sınıfının bir örneğini oluşturmak için aşağıda gösterildiği gibi kurucu fonksiyona dosyanın yolunu, hangi modda açacağımızı ve erişim yetkisini belirtiyoruz.

```
// Kullanım şekli;
FileStream fs = new FileStream(string path, FileMode mode, FileAccess access);

// D dizininde bulunan CSharp.txt dosyasını yazma yetkisiyle açmak için;
string dosyaYolu = @"D:\CSharp.txt";
FileStream fs = new FileStream(dosyaYolu, FileMode.Open, FileAccess.Write);
```

$FileShare$, üzerinde çalışılan bir dosyanın eşzamanlı olarak diğer işlemlerle (proseslerle) paylaşılması gerektiği durumlarda yetki vermek için kullanılır. Örneğin, bir ASP.NET uygulaması için bir veritabanı dosyası görevi gören bir XML dosyanız olduğunu varsayalım. Eğer $FileAccess$ ile bir seçenek belirtmezseniz ya da varsayılan olarak kullanılan $FileShare.None$ seçilirse, bir seferde XML veritabanı dosyasından yalnızca bir kullanıcı okuyabilir. Bu veritabanına eşzamanlı olarak erişmeye çalışan diğer kullanıcılar bir hatayla karşılaşır. Aşağıdaki tabloda dosya paylaşım özellikleri ve bunlara ait açıklamalar gösterilmektedir.

Özellik	Açıklamalar
Inheritable	Dosyanın yavru ($child$) prosesler tarafından türetilebilmesini sağlar.
None	Dosyanın aynı anda başka prosesler tarafından açılmasını engeller.
Read	Dosyanın aynı anda başka proseslerce de okunabilmesini sağlar.
ReadWrite	Dosyanın aynı anda başka proseslerce de okunup yazılabilmesini sağlar.
Write	Dosyaya aynı anda başka proseslerce yazılabilmesini sağlar.

$FileShare$ kullanarak $FileStream$ sınıfının bir örneğini oluşturmak için aşağıda gösterildiği gibi kurucu fonksiyona dosyanın yolunu, hangi modda açacağımızı, erişim yetkisini ve paylaşım durumunu belirtiyoruz.

```
// Kullanım şekli;
FileStream fs = new FileStream(string path, FileMode mode,
    FileAccess access, FileShare share);

// Dosyayı yazma yetkisiyle açmak ve salt okunur olarak paylaşmak için;
string dosyaYolu = @"D:\CSharp.txt";
FileStream fs = new FileStream(dosyaYolu, FileMode.Open,
    FileAccess.Write, FileShare.Read);
```

Şimdi bir metin dosyası oluşturalım ve bu dosyanın içerisine bir veri yazarak kaydedelim; öncelikle oluşturacağımız dosyanın adı ve uzantısını içeren tam yolu belirterek bunu bir değişkene atayalım. Daha sonra $Encoding$ sınıfını kullanarak dosyaya yazacağımızı veriyi "UTF8" formatında $byte$ dizisine dönüştürelim. Ancak, $Encoding$ sınıfına erişebilmek için ilk önce projenize "$System.Text$" kütüphanesini eklemeniz gerekecektir.

```
1  string dosyaYolu = @"D:\CSharp.txt";
2
3  using (FileStream fs = File.Create(dosyaYolu))
4  {
5      byte[] veri = Encoding.UTF8.GetBytes("Yeni başlayanlar için C#.NET");
6      fs.Write(veri, 0, veri.Length);
7  }
```

Şimdi yukarıda oluşturduğumuz dosyayı açarak, içerisine yazmış olduğumuz verileri konsol ekranında görüntüleyelim. Bunun için dosya modunu $FileMode.Open$ olarak belirliyoruz.

```
1  var dosyaYolu = @"D:\CSharp.txt";
2
3  using (var fs = new FileStream(dosyaYolu, FileMode.Open))
4  {
5      byte[] buffer = new byte[1024];
6
7      while (fs.Read(buffer, 0, buffer.Length) > 0)
8      {
9          Console.Write(Encoding.UTF8.GetString(buffer));
10     }
11 }
```

Yukarıdaki dosya okuma örneğimizde, 1024 byte'lık yani 1 KB kapasitesine sahip bir tampon ($buffer$) tanımladık. Böylece $while$ bloğunun her yinelenmesinde konsol ekranına 1 KB'lık veri yazdıracaktır. Aşağıdaki örneğimizde ise dosya içerisinde bulunan karakterler bir for döngüsü içinde tek tek okunuyor.

```
1  var dosyaYolu = @"D:\CSharp.txt";
2
3  using (var fs = new FileStream(dosyaYolu, FileMode.Open))
4  {
5      for (int i = 0; i < fs.Length; i++)
6      {
7          Console.Write((char)fs.ReadByte());
8      }
9  }
```

Yukarıdaki örneğimizin yedinci satırında dikkat ettiyseniz, "$(\boldsymbol{char})fs.ReadByte()$" şeklinde okunan veriyi $char$ tipine dönüştürdük. Bu dönüşümü yapmadığımız taktirde karakterlerin ASCII kodlarını ekrana basacaktır.

Son olarak, aynı dosyamız için bu sefer yazma yetkisi verelim ve dosyanın sonuna başka bir veri ekleyelim. Bu işlem için dosya modunu "$FileMode.Append$" olarak belirtmemiz gerekir.

```
1  var dosyaYolu = @"D:\CSharp.txt";
2
3  using (var fs = new FileStream(dosyaYolu, FileMode.Append, FileAccess.Write))
4  {
5      byte[] veri = Encoding.UTF8.GetBytes("\nwww.elenium.net");
6      fs.Write(veri, 0, veri.Length);
7  }
```

Bu işlemin ardından programı çalıştırdığınızda, dosya içeriğini konsola aşağıdaki gibi yazacaktır.

```
Yeni başlayanlar için C#.NET
www.elenium.net
```

12.2.2. TextReader & TextWriter

$TextReader$ sınıfı, $StreamReader$ ve $StringReader$ sınıfları için temel bir sınıftır ($base\ class$). $StreamReader$, bir akış kaynağından ($stream$) karakterleri okumak için kullanılır. $StringReader$ ise bir dizgeden ($string$) karakterleri okumak için kullanılır. Benzer şekilde $TextWriter$ sınıfı da $StreamWriter$ ve $StringWriter$ sınıfları için temel bir sınıf niteliğindedir. Dosyalama işlemlerinde bir metni ($text$) okumak için $StreamReader$ veya $StringReader$ kullanmanız gerekir. Bu sınıflar da $System.IO$ isim uzayında tanımlanmıştır. Bu sınıfları kullanmadan önce projenize $System.IO$ kütüphanesini eklemeniz gerekir. Aşağıdaki tabloda $TextWriter$ sınıfının yaygın olarak kullanılan metotları gösterilmiştir.

Tablo 2. TextWriter sınıfının yaygın olarak kullanılan metotlar ve açıklamaları.

Metot	Açıklamalar
Write	Verilerin $TextStream$'e yazdırılmasını sağlar. Bu metodun aşırı yüklenmiş 17 farklı sürümü bulunmaktadır.
WriteLine	Verilerin $TextStream$'e yeni bir kayıt satırı olarak yazdırılmasını sağlar. Bu metodun aşırı yüklenmiş 18 farklı sürümü bulunmaktadır.
Flush	Geçerli $TextWriter$ için tüm arabelleği ($buffer$) temizler ve arabellek verilerinin fiziksel ortama yazdırılmasını sağlar.
Close	$TextWriter$'ı kapatır ve bununla ilişkili tüm sistem kaynaklarını serbest bırakır.
Dispose	$TextWriter$ tarafından kullanılan tüm kaynakları serbest bırakır.

Daha önce de belirtildiği gibi, $StreamWriter$ türü $TextWriter$ adlı bir temel sınıftan türemiştir. Aşağıdaki örneğimizde, belirtilen konumda ve isimde bir metin dosyası oluşturarak içerisine $string$ tipinde verilerin nasıl yazılacağı gösterilmektedir.

```
 1  using System;
 2  using System.IO;
 3
 4  public class DosyaIslemleri
 5  {
 6      static void Main(string[] args)
 7      {
 8          string dosya = "C:\\Elenium\\CSharp.txt";
 9
10          try
11          {   // Konumu ve adı belirtilmiş olan dosyayı oluştur.
12              using (TextWriter writer = File.CreateText(dosya))
13              {
14                  // Oluşturulan dosyaya verileri yaz.
15                  writer.WriteLine("Yeni başlayanlar için");
16                  writer.WriteLine("C# .NET");
17              }
18          }
19          catch (IOException exp)
20          {
21              Console.WriteLine(exp.ToString());
22          }
23      }
24  }
```

Burada "$File.CreateText$" metodu ile oluşturulan dosya $StreamWriter$ olarak geri döndürülür. Böylece $TextWriter$ tipinde tanımlamış olduğumuz $writer$ değişkenine aktarabiliyoruz. Şimdi de $TextReader$ sınıfını inceleyelim ve daha önce oluşturmuş olduğumuz dosyayı kullanarak verileri okuyan örnek kodumuzu yazalım. Aşağıdaki tabloda $TextReader$ sınıfının yaygın olarak kullanılan metotları gösterilmiştir.

Tablo 3. TextReader sınıfının yaygın olarak kullanılan metotlar ve açıklamaları.

Metot	Açıklamalar
Read	$TextReader$'dan bir karakteri okur ve karakter konumunu bir karakter adımı ilerletir. Eğer okunacak karakter bulunamazsa -1 değerini döndürür.
ReadBlock	$TextReader$'dan belirtilen $index$ değerinden başlayarak, $count$ parametresinin değeri kadar karakteri okur ve belirtilen aralıktaki verileri bir arabelleğe ($buffer$) yazar.
ReadLine	$TextReader$'dan bir karakter satırını okur ve verileri bir dize olarak döndürür.
ReadToEnd	Geçerli konumdan başlayarak $TextReader$'ın sonuna kadar olan tüm karakterleri okur ve bunları tek bir dize olarak döndürür.
Peek	Okuyucu durumunu veya karakter kaynağını değiştirmeden bir sonraki karakteri okur. $TextReader$'dan kullanılabilir bir sonraki karakteri okumadan döndürür.
Close	$TextReader$'ı kapatır ve bununla ilişkili tüm sistem kaynaklarını serbest bırakır.

Dispose	$TextReader$ tarafından kullanılan tüm kaynakları serbest bırakır.

Bu örneğimizde de daha önce "$C:\Elenium\CSharp.txt$" olarak oluşturduğumuz dosyanın içeriğini nasıl okuyacağımızı görelim; belirtilen konumdaki dosyamızı $StreamReader$ ile $TextReader$ nesnesine yüklüyoruz. Daha sonra dosyadaki tüm kayıtları okuyabilmek için $while$ döngüsü kullanarak, okunabilir her kayıt satırını konsola yazdırıyoruz.

```
 1  using System;
 2  using System.IO;
 3
 4  public class DosyaIslemleri
 5  {
 6      static void Main(string[] args)
 7      {
 8          string dosya = "C:\\Elenium\\CSharp.txt";
 9
10          try
11          {
12              string data = null; // Değişkene null değerini ata.
13              // Belirtilen dosya ile TextReader'ın bir örneğini oluştur.
14              TextReader reader = new StreamReader(dosya);
15
16              // Dosyada okunabilen kayıt olduğu sürece çalışan döngü.
17              while ((data = reader.ReadLine()) != null)
18              {
19                  // Dosyadaki her bir kayıt satırını konsola yazdır.
20                  Console.WriteLine(data);
21              }
22
23              reader.Close(); // TextReader'ı kapat.
24          }
25          catch (IOException exp) // Bir hata oluştuysa;
26          {
27              Console.WriteLine(exp.ToString());
28          }
29      }
30  }
```

Burada dikkat edilirse, bir önceki dosya yazma örneğimizden farklı olarak kodumuzu try-catch bloğu içerisine yazdık. Bu nedenle dosya okuma işimiz bittikten sonra kullanmış olduğumuz sistem kaynaklarını serbest bırakmamız gerekecektir. Bunun için dosya okuma işimizin bittiği yerde $Close$ metodunu çağırdık. Eğer bu işlemi $using$ bloğu içerisinde gerçekleştirmiş olsaydık, $Close$ metodunu çağırmamıza gerek kalmayacaktı.

Dikkat!
Bu tür, $IDisposable$ arabirimini uygular. Bu sınıftan türetilen herhangi bir türü kullanmayı bitirdiğinizde, kaynakları serbest bırakmak için $Dispose$ yöntemini bir $try/catch$ bloğu içerisinde çağırınız ya da $using$ gibi bir dil yapısı kullanınız.

12.2.3. StreamReader & StreamWriter

Karakter tabanlı verileri okumak veya yazmak istediğimizde, $StreamReader$ ve $StreamWriter$ sınıflarını kullanabiliriz. $StreamReader$, $TextReader$ sınıfından, $StreamWriter$ ise $TextWriter$ sınıfından türetilmiştir. Ek olarak, bu sınıflar karakter kodlamaya duyarlıdır ve farklı formatlarda kodlanmış metinleri okumak ve yazmak için kullanılabilir. Varsayılan olarak kullanılan kodlama formatı UTF8'dir. Aşağıdaki tabloda $StreamReader$ sınıfına ait bazı metotlar ve bunların açıklamaları verilmiştir.

Metot	Açıklamalar
Read()	Bu yöntem dosyadan sadece bir karakter okumak istenildiği zaman kullanılır. Okunacak karakter yoksa "-1" değeri döndürür.
Read(char[], int, int**)**	$Read()$ metodunun diğer bir sürümü olan bu yöntem, belirli bir sayıdaki ($count$) karakteri dosyadan okuyarak tampon belleğe ($buffer$) aktarır.
ReadBlock(char[], int, int**)**	Geçerli akıştan belirtilen maksimum karakter sayısını okur ve verileri belirtilen dizinden başlayarak tampon belleğe yazar.
ReadLine()	Geçerli akıştan bir kayıt satırı okur ve verileri bir dizge ($string$) olarak döndürür.
ReadToEnd()	Geçerli konumdan itibaren dosyanın sonuna kadar olan bütün bilgileri tek seferde okumak için kullanılır. Geçerli konum akışın ($stream$) sonundaysa, boş bir dize ("") döndürür.
Peek()	Bir dosyada okunacak karakter olup olmadığını kontrol etmek için kullanılır. Eğer okunacak herhangi bir karakter yoksa geriye negatif bir değer döndürür.
Close()	$StreamReader$'ı kapatır ve bununla ilişkili tüm sistem kaynaklarını serbest bırakır.
Dispose()	$StreamReader$ tarafından kullanılan tüm kaynakları serbest bırakır.

İlk örneğimizde, $StreamWriter$ sınıfını kullanarak bir metin dosyası oluşturalım ve bu dosyanın içerisine bir veri yazarak kaydedelim; öncelikle oluşturacağımız dosyanın adı ve uzantısını içeren tam yolu belirterek bunu bir değişkene atayalım. Daha sonra $using$ deyimi içerisinde bir $StreamWriter$ nesnesi oluşturarak dosya içerisine birkaç satır veri yazalım.

```
1  using System;
2  using System.IO;
3
4  public class Program
5  {
6      static void Main(string[] args)
7      {
8          string dosyaYolu = @"D:\CSharp.txt";
9
```

```
10          using (var sw = new StreamWriter(dosyaYolu))
11          {
12              sw.WriteLine("Yeni başlayanlar için C#.NET");
13              sw.WriteLine("Abdullah ELEN");
14              sw.WriteLine("Emre AVUÇLU");
15          }
16      }
17  }
```

Şimdi, oluşturduğumuz dosyadaki bilgileri $StreamReader$ sınıfını kullanarak, farklı şekillerde okuyabileceğimiz metotları deneyelim. Her bir metodun farkını anlayabilmek için $while$ bloğu içinde döngü yinelemesini işaret etmek için '*' karakterini de konsola yazdırdık. Böylece $Peek()$ ile her seferinde dosyadan ne kadar veri okunduğunu gözlemleyebileceğiz. İlk örneğimizde $Read()$ metodunu inceleyelim.

```
 1  using System;
 2  using System.IO;
 3
 4  public class Program
 5  {
 6      static void Main(string[] args)
 7      {
 8          string dosyaYolu = @"D:\CSharp.txt";
 9
10          using (var reader = new StreamReader(dosyaYolu))
11          {
12              while (reader.Peek() >= 0)
13              {
14                  Console.Write((char)reader.Read());
15                  Console.Write('*'); // Yineleyici işareti.
16              }
17          }
18      }
19  }
```

Programın çıktısı aşağıdaki gibidir; bu örneğimizde mevcut dosyadaki verileri $Read()$ metodu ile harf harf yani karakter olarak okuduk. Kaynak kodumuzda $while$ döngüsü, dosyadan her bir karakterlik veri okuduğunda yinelenmektedir. Program çıktısında görünen yıldız (*) karakterleri ile ayrılan yerler bunları işaret etmektedir.

```
Y*e*n*i* *b*a*ş*l*a*y*a*n*l*a*r* *i*ç*i*n* *C*#*.*N*E*T*
*A*b*d*u*l*l*a*h* *E*L*E*N*
*E*m*r*e* *A*V*U*Ç*L*U*
```

Kaynak kodunun 15. satırındaki "$Console.Write('*');$" ifadesini kaldırarak programı çalıştırırsanız, konsol ekranındaki görüntü aşağıdaki gibi görünecektir. Bunu aşağıdaki tüm dosya okuma örnekleri için test edebilirsiniz.

```
Yeni başlayanlar için C#.NET
Abdullah ELEN
Emre AVUÇLU
```

$Read(char[\,], int, int)$ metodu ile dosyanın okunması;

```
 1  using System;
 2  using System.IO;
 3
 4  public class Program
 5  {
 6      static void Main(string[] args)
 7      {
 8          string dosyaYolu = @"D:\CSharp.txt";
 9
10          using (var reader = new StreamReader(dosyaYolu))
11          {
12              char[] parca = null;
13
14              while (reader.Peek() >= 0)
15              {
16                  parca = new char[15];
17                  reader.Read(parca, 0, parca.Length);
18                  Console.WriteLine(c);
19                  Console.Write('*'); // Yineleyici işareti.
20              }
21          }
22      }
23  }
```

Programın çıktısı aşağıdaki gibidir; kaynak kodumuzun 16. satırında "$parca$" isimli değişken 15 karakter uzunluğunda bir $char$ dizisi tanımlanmıştır. Böylece $while$ döngüsü, dosyadan her 15 karakterlik veri okuduğunda yinelenmektedir. Program çıktısında görünen yıldız (*) karakterleri ile ayrılan yerler bunları işaret etmektedir.

```
Yeni başlayanla*r için C#.NET
*Abdullah ELEN
*Emre AVUÇLU
 *
```

$ReadBlock$ metodu ile dosyanın okunması;

```
 1  using System;
 2  using System.IO;
 3
 4  public class Program
 5  {
 6      static void Main(string[] args)
 7      {
```

```
 8          string dosyaYolu = @"D:\CSharp.txt";
 9
10          using (var reader = new StreamReader(dosyaYolu))
11          {
12              char[] parca = null;
13
14              while (reader.Peek() >= 0)
15              {
16                  parca = new char[15];
17                  reader.ReadBlock(parca, 0, parca.Length);
18                  Console.Write(parca);
19                  Console.Write('*'); // Yineleyici işareti.
20              }
21          }
22      }
23  }
```

Programın çıktısı aşağıdaki gibidir;

```
Yeni başlayanla*r için C#.NET
*Abdullah ELEN
*Emre AVUÇLU
 *
```

ReadLine metodu ile dosyanın okunması;

```
 1  using System;
 2  using System.IO;
 3
 4  public class Program
 5  {
 6      static void Main(string[] args)
 7      {
 8          string dosyaYolu = @"D:\CSharp.txt";
 9
10          using (var reader = new StreamReader(dosyaYolu))
11          {
12              while (reader.Peek() >= 0)
13              {
14                  Console.WriteLine(reader.ReadLine());
15                  Console.Write('*'); // Yineleyici işareti.
16              }
17          }
18      }
19  }
```

Programın çıktısı aşağıdaki gibidir;

```
Yeni başlayanlar için C#.NET*Abdullah ELEN*Emre AVUÇLU*
```

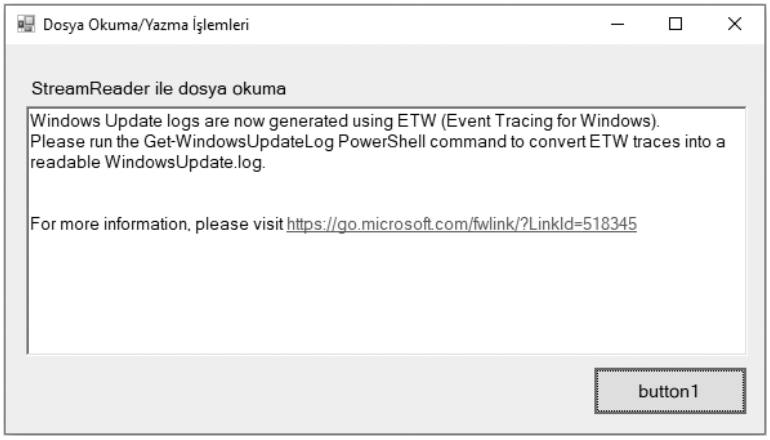
ReadToEnd metodu ile dosyanın okunması;

```
1  using System;
2  using System.IO;
3
4  public class Program
5  {
6      static void Main(string[] args)
7      {
8          string dosyaYolu = @"D:\CSharp.txt";
9
10         using (var reader = new StreamReader(dosyaYolu))
11         {
12             Console.WriteLine(reader.ReadToEnd());
13             Console.Write('*'); // Yineleyici işareti.
14         }
15     }
16 }
```

Programın çıktısı aşağıdaki gibidir;

```
Yeni başlayanlar için C#.NET
Abdullah ELEN
Emre AVUÇLU
*
```

Son örneğimizi bir "Form Application" projesi üzerinden gerçekleştirelim. Bunun için yeni bir Form Application projesi başlatın ya da mevcut projeniz üzerinde de değişiklik yapabilirsiniz. Aşağıdaki şekilde görüldüğü üzere, form üzerine bir $RichTextBox$ ve $Button$ kontrollerini ekleyin ve okumak istediğimiz bir dosyayı açabilmek için $OpenFileDialog$ kontrolünü ekleyin.



Daha sonra, $Button$ kontrolü üzerine $mouse$ ile çift tıklatarak kod editöründe, "$Click$" olayının gerçekleşeceği "$button1\_Click$" adlı metoda aşağıda gösterildiği şekilde kodlamayı yaparak projenizi çalıştırabilirsiniz. Yukarıda konsol uygulaması olarak verdiğimiz örnekleri de bu proje üzerinde test edebilirsiniz.

```
 1  using System;
 2  using System.IO;
 3  using System.Windows.Forms;
 4
 5  namespace WindowsFormsApp1
 6  {
 7      public partial class Form1 : Form
 8      {
 9          public Form1()
10          {
11              InitializeComponent();
12          }
13
14          private void button1_Click(object sender, EventArgs e)
15          {
16              var fileDialog = new OpenFileDialog();
17
18              if (fileDialog.ShowDialog() == DialogResult.OK)
19              {
20                  string fileName = fileDialog.FileName;
21                  var stream = new StreamReader(File.OpenRead(fileName));
22                  richTextBox1.Text = stream.ReadToEnd();
23                  stream.Dispose();
24              }
25          }
26      }
27  }
```

Dikkat!
Bu tür, $IDisposable$ arabirimini uygular. Bu sınıftan türetilen herhangi bir türü kullanmayı bitirdiğinizde, kaynakları serbest bırakmak için $Dispose$ yöntemini bir $try/catch$ bloğu içerisinde çağırınız ya da $using$ gibi bir dil yapısı kullanınız.

12.2.4. BinaryReader & BinaryWriter

$BinaryReader$ ve $BinaryWriter$ sınıfları, ilkel .NET veri tiplerini belirli bir kodlama biçiminde (ASCII, Unicode, UTF32, UTF7 ve UTF8) $binary$ (ikili) olarak okumak ve yazmak için kullanılır. Yalnızca ilkel türlerle ilgileniyorsanız, bu sizin için kullanılacak en uygun yöntem olacaktır. Verilerin $binary$ (ikili) formatta okunduğundan, dosya içeriğinin okunabilir bir formatta olmadığı bilinmesi gerekir. Bu sınıfların kurucu metotları, $Stream$ (akış) ve $Encoding$ (karakter kodlama formatı) parametreleri ile tanımlanır. Aşağıdaki kod parçacığı, $BinaryWriter$ sınıfının bir örneğini oluşturur.

```
BinaryWriter writer;
var stream = new FileStream("BinaryDosyamiz.bin", FileMode.Create);
```

```
writer = new BinaryWriter(stream, Encoding.ASCII);
```

Burada $FileStream$ nesnesi ile oluşturulacak dosya adını (tam yolunu) ve bu dosyanın hangi modda açılması gerektiğini belirtiyoruz. Böylece $BinaryWriter$ sınıfını için gerekli olan akışımız hazır hale gelecektir. Son işlem adımında ise dosyada saklayacağımız veri için bir karakter kodlama formatı belirtiyoruz. Herhangi bir kodlama türü belirtilmediği takdirde "UTF-8" karakter kodlaması varsayılan olarak kullanılır. Şimdi örneğimizi biraz daha geliştirerek, oluşturduğumuz yeni dosyaya nasıl veri yazacağımızı görelim;

```
 1  using System;
 2  using System.IO;
 3  using System.Text;
 4
 5  public class Program
 6  {
 7      static void Main(string[] args)
 8      {
 9          BinaryWriter writer = null;
10          string path = @"D:\CSharp.bin";
11
12          try
13          {
14              writer = new BinaryWriter(
15                  new FileStream(path, FileMode.Create), Encoding.ASCII);
16
17              string yazarAdi = "Abdullah ELEN";
18              int dogumYili = 1981;
19              bool aktifKayit = true;
20
21              writer.Write(yazarAdi);
22              writer.Write(dogumYili);
23              writer.Write(aktifKayit);
24
25              writer.Close();
26          }
27          finally // Bir hata meydana gelirse, kaynaklar serbest bırakılsın.
28          {
29              if (!ReferenceEquals(null, writer)) writer.Dispose();
30          }
31      }
32  }
```

Örneğimizde görüldüğü gibi $BinaryWriter$ sınıfının $write()$ metodunu kullanarak dosyamıza farklı tipte verileri yazabiliyoruz. Şimdi aynı örneğimizi daha kısa kod yazmak amacıyla $using$ kullanarak kodlayalım.

```
1  using System;
2  using System.IO;
3  using System.Text;
4
5  public class Program
```

```
 6  {
 7      static void Main(string[] args)
 8      {
 9          string path = @"D:\CSharp.bin";
10
11          using (var writer = new BinaryWriter(
12              new FileStream(path, FileMode.Create), Encoding.ASCII))
13          {
14              string yazarAdi = "Abdullah ELEN";
15              int dogumYili = 1981;
16              bool aktifKayit = true;
17
18              writer.Write(yazarAdi);
19              writer.Write(dogumYili);
20              writer.Write(aktifKayit);
21          }
22      }
23  }
```

$BinaryWriter$ sınıfındaki $write$ metodunun aşırı yüklenmiş 18 farklı sürümü bulunmaktadır. Buna göre sizlerde oluşturulan dosyaya farklı tipte veriler ekleyebilirsiniz. Aşağıdaki tabloda bu sınıfa ait yaygın olarak kullanılan metotlar açıklanmaktadır.

Metot	Açıklamalar
Write(bool value**)**	Geçerli akışa bir baytlık $Boolean$ değeri yazar; burada 0 $false$ ve 1 ise $true$ değerini gösterir.
Write(byte value**)**	Geçerli akışa işaretsiz bir bayt veri yazar ve akış konumunu bir bayt ilerletir.
Write(byte[] buffer**)**	Geçerli akışa bir bayt dizisi yazar.
Write(char ch**)**	Geçerli akışa bir karakter uzunluğunda veri yazar.
Write(char[] chars**)**	Geçerli akışa bir karakter dizisi yazar.
Write(double value**)**	Geçerli akışa sekiz baytlık ondalıklı bir değer yazar ve akışın konumunu sekiz bayt ilerletir.
Write(int value**)**	Geçerli akışa dört baytlık işaretli bir tam sayı yazar ve akış konumunu dört bayt ilerletir.
Write(string value**)**	$BinaryWriter$'ın karakter kodlayıcısı ($Encoding$) ile ilişkili olan akışa bir dize yazar. Kullanılan karakter kodlama yöntemine ve akışa yazılan karakter uzunluğuna göre akışı konumunu ilerletir.
Close()	Geçerli $BinaryWriter$'ı ve buna bağlı olan akışı kapatır.
Flush()	Geçerli $BinaryWriter$'ın tüm arabelleğini temizler ve tamponlanmış verilerin fiziksel ortama yazılmasını sağlar.
Seek(int offset, SeekOrigin origin**)**	Geçerli akış içindeki konumu ayarlar.

Aynı örneğimiz üzerinden devam edecek olursak; şu ana kadar yapılan işlemlerde bir dosya oluşturup içerisine çeşitli tiplerde veri eklemişti. Şimdi ise "$CSharp.bin$" dosyasından verileri nasıl okuyacağımızı görelim;

```
 1  using System;
 2  using System.IO;
 3  using System.Text;
 4
 5  public class Program
 6  {
 7      static void Main(string[] args)
 8      {
 9          using (var reader = new BinaryReader(
10              new FileStream(path, FileMode.Open), Encoding.ASCII))
11          {
12              string yazarAdi = reader.ReadString();
13              int dogumYili = reader.ReadInt32();
14              bool aktifKayit = reader.ReadBoolean();
15
16              Console.WriteLine("String veri: {0}", yazarAdi);
17              Console.WriteLine("Integer veri: {0}", dogumYili);
18              Console.WriteLine("Boolean veri: {0}", aktifKayit);
19          }
20      }
21  }
```

Yukarıdaki örneğimizde, dosyadan verileri okumak için bu sefer $BinaryReader$ sınıfının bir örneğini oluşturduk. Bunun için gerekli olan tanımlar $BinaryWriter$ sınıfı ile aynıdır. Ancak burada akışı oluştururken dosya modunu "$FileMode.Open$" olarak belirlendiğine dikkat ediniz. Dosyamızda kayıtlı olan verileri okumak için $BinaryReader$ sınıfı $ReadString$, $ReadInt32$ vb. farklı veri tiplerine özgü yöntemler sağlar. Ancak burada dikkat edilmesi gereken önemli bir nokta vardır; $Binary$ olarak yazılan dosyamızdan verileri okuma yaparken aynı sırada ve aynı veri tipinde olması gerekir. Aksi taktirde programınız çalışma esnasında dosya okuma hatası verecektir. Aşağıdaki tabloda $BinaryReader$ sınıfının yaygın olarak kullanılan metotları açıklanmaktadır.

Metot	Açıklamalar
Read()	Dosyadan belirtilen miktarda veri okumak için kullanılır.
ReadBoolean()	Dosyadan $bool$ tipindeki verileri okumak için kullanılır.
ReadByte()	Dosyadan sadece bir byte veri okumak için kullanılır.
ReadBytes(int count**)**	Dosyadan $count$ ile belirtilen büyüklükte byte veri okumak için kullanılır.
ReadChar()	Dosyadan sadece bir karakter okumak için kullanılır.
ReadChars(int count**)**	Dosyadan $count$ ile belirtilen miktar kadar karakter okumak için kullanılır.
ReadDouble()	Dosyadan $double$ tipindeki verileri okumak için kullanılır.
ReadInt32()	Dosyadan int tipindeki verileri okumak için kullanılır.
ReadString()	Dosyadan $string$ tipindeki verileri okumak için kullanılır.
Close()	Geçerli okuyucuyu ve buna bağlı olan akışı kapatır.