Bu doküman, C# dilini kullanarak veriyi kalıcı olarak saklamanın en temel yollarından biri olan dosya okuma ve yazma işlemlerine odaklanmaktadır. Programların ürettiği verilerin, uygulama sonlandığında kaybolmamasını sağlamak, modern yazılım geliştirmenin temel gereksinimlerinden biridir. Bu rehber, .NET platformunun System.IO isim alanı altında yer alan ve sıkça kullanılan File sınıfını merkeze almaktadır. File.WriteAllText(), File.ReadAllText(), File.AppendAllText() ve File.Exists() gibi temel metotlar aracılığıyla bir metin dosyasının nasıl oluşturulacağı, okunacağı, üzerine nasıl veri ekleneceği ve dosya varlığının nasıl kontrol edileceği ayrıntılı örneklerle açıklanmıştır. Dosya işlemleri sırasında karşılaşılabilecek olası hataların yönetimi de dahil olmak üzere, bu konu tüm C# geliştiricileri için vazgeçilmez bir başlangıç noktası sunmaktadır.

BASİT DOSYA İŞLEMLERİ

Okuma, Yazma, Ekleme

Oğuzhan Karagüzel

İçindekiler

GİRİŞ

Bu dokümanda, C# programlama dili kullanılarak temel dosya okuma ve yazma işlemlerinin nasıl gerçekleştirileceği incelenecektir. Programların büyük bir kısmı, veriyi kalıcı olarak saklama ihtiyacı duyar. Program sonlandığında bellekteki (RAM) tüm veriler kaybolur. Verinin programın çalışma ömrü dışına taşınması ve kalıcı hale getirilmesi için en yaygın yöntemlerden biri dosyaları kullanmaktır.

Kullanıcı ayarlarını saklamak, program tarafından üretilen raporları kaydetmek, işlem günlüklerini (log) tutmak veya harici bir konfigürasyon dosyasını okumak gibi sayısız senaryoda dosya işlemlerine başvurulur. C#, System.IO isim alanı (namespace) altında dosya ve dizin işlemleri için zengin bir kütüphane sunar. Bu doküman, bu kütüphanenin en temel ve sık kullanılan sınıflarından biri olan System.IO.File sınıfı üzerine odaklanarak, metin tabanlı dosya işlemlerinin temellerini atmayı hedeflemektedir.

System.IO Namespace VE File SINIFI

C#'ta dosya işlemleriyle ilgili tüm sınıflar, metotlar ve yapılar genellikle System.IO isim alanı altında toplanmıştır. IO, "Input/Output" (Giriş/Çıkış) kelimelerinin kısaltmasıdır.

Bu isim alanı içindeki en önemli sınıflardan biri statik (static) bir sınıf olan File sınıfıdır. "Statik" olması, bu sınıfın bir örneğini (new File()) oluşturmamıza gerek olmadığı anlamına gelir. Metotlarını doğrudan sınıfın adı üzerinden (File.MetotAdi()) çağırabiliriz. File sınıfı, dosyaları oluşturma, kopyalama, silme, taşıma ve içeriklerini okuyup yazma gibi temel operasyonları gerçekleştiren metotlar sunar. Bu metotlar, özellikle basit ve tek seferlik dosya işlemleri için son derece pratiktir.

DOSYAYA METİN YAZMA: File.WriteAllText()

File.WriteAllText() metodu, bir metin dosyasının tüm içeriğini belirlediğiniz bir string değer ile oluşturmak veya üzerine yazmak için kullanılır.

Çalışma Mantığı:

1. Metoda bir dosya yolu ve yazılacak metin verilir.
2. Eğer belirtilen yolda bir dosya **yoksa**, metot bu dosyayı otomatik olarak oluşturur ve metni içine yazar.
3. Eğer belirtilen yolda bir dosya **varsa**, metot bu dosyanın **mevcut tüm içeriğini siler** ve yeni metni dosyaya yazar. Bu, "üzerine yazma" (overwrite) işlemidir.

Sözdizimi:
File.WriteAllText(string path, string contents);

ÖRNEK:

```
using System;

using System.IO;
```

```
class Program
{
    static void Main(string[] args)
    {
        // Dosyanın kaydedileceği yolu belirtiyoruz.
        // @ işareti, string içindeki '\' karakterlerinin özel bir anlam taşımamasını sağlar.
        string dosyaYolu = @"C:\CSharpDersleri\notlar.txt";

        // Yazmak istediğimiz metin içeriği
        string icerik = "Bu, C# ile dosyaya yazılan ilk metindir.\nİkinci satıra hoş geldiniz.";

        try
        {
            // Belirtilen yola, belirtilen içeriği yaz.
            File.WriteAllText(dosyaYolu, icerik);

            Console.WriteLine("Dosyaya yazma işlemi başarıyla tamamlandı.");
            Console.WriteLine($"Dosya konumu: {dosyaYolu}");
        }
        catch (Exception ex)
        {
            // Yazma işlemi sırasında bir hata oluşursa (örn: izin hatası)
            Console.WriteLine($"Bir hata oluştu: {ex.Message}");
        }
    }
}
```

İşlem Sonucu:
Eğer C:\CSharpDersleri dizini mevcutsa (değilse bir DirectoryNotFoundException alırsınız), bu kod notlar.txt adında bir dosya oluşturur (veya üzerine yazar) ve içeriği aşağıdaki gibi olur:

Bu, C# ile dosyaya yazılan ilk metindir.

İkinci satıra hoş geldiniz.

Önemli Not: File.WriteAllText() metodu, mevcut bir dosyanın içeriğini sorgusuz sualsiz sileceği için dikkatli kullanılmalıdır. Veri kaybına yol açma potansiyeli taşır. Eğer mevcut veriyi koruyup sonuna ekleme yapmak istiyorsanız File.AppendAllText() kullanılmalıdır.

DOSYANIN VARLIĞINI KONTROL ETME: File.Exists()

Bir dosya üzerinde okuma veya silme gibi bir işlem yapmadan önce, o dosyanın belirtilen yolda gerçekten var olup olmadığını kontrol etmek iyi bir programlama pratiğidir. File.Exists() metodu bu amaçla kullanılır.

Çalışma Mantığı:

- Metoda bir dosya yolu verilir.
- Eğer dosya belirtilen yolda mevcutsa true, değilse false değerini döndürür.

Sözdizimi:
bool dosyaVarMi = File.Exists(string path);

Örnek:

```
string dosyaYolu = @"C:\CSharpDersleri\notlar.txt";


if (File.Exists(dosyaYolu))
{
    Console.WriteLine("Belirtilen yolda dosya bulundu.");
    // Dosya bulunduğuna göre okuma veya diğer işlemler yapılabilir.
}
else
{
    Console.WriteLine("Dosya bulunamadı! Lütfen yolu kontrol ediniz.");
}
```

Not: File.Exists() kullanımı, programınızı daha sağlam hale getirir ve FileNotFoundException gibi hataları proaktif olarak önlemenize yardımcı olur. Özellikle bir dosyayı okumadan önce bu kontrolü yapmak kritik önem taşır.

DOSYADAN METİN OKUMA: File.ReadAllText()

File.ReadAllText() metodu, bir metin dosyasının **tüm içeriğini** tek bir string değişkenine okur.

Çalışma Mantığı:

1. Metoda okunacak dosyanın yolu verilir.
2. Metot dosyayı açar, içindeki tüm metni okur ve bu metni tek bir string olarak döndürür.
3. Eğer dosya bulunamazsa FileNotFoundException fırlatır.

Sözdizimi:
string icerik = File.ReadAllText(string path);

Örnek:
(Önceki WriteAllText örneğinde oluşturulan notlar.txt dosyasını okuyalım)

```
string dosyaYolu = @"C:\CSharpDersleri\notlar.txt";


if (File.Exists(dosyaYolu))
{
    try
    {
        string okunanIcerik = File.ReadAllText(dosyaYolu);
        Console.WriteLine("--- Dosya İçeriği Başlangıcı ---");
        Console.WriteLine(okunanIcerik);
        Console.WriteLine("--- Dosya İçeriği Sonu ---");
    }
    catch (Exception ex)
    {
        Console.WriteLine($"Dosya okunurken bir hata oluştu:
{ex.Message}");
    }
}
else
{
    Console.WriteLine("Okunacak dosya bulunamadı.");
}
```

İşlem Sonucu (Konsol Çıktısı):

--- Dosya İçeriği Başlangıcı ---

Bu, C# ile dosyaya yazılan ilk metindir.

İkinci satıra hoş geldiniz.

--- Dosya İçeriği Sonu ---

Uyarı: File.ReadAllText() tüm dosyayı belleğe (RAM) yükler. Bu, küçük ve orta boyutlu metin dosyaları için son derece pratiktir. Ancak çok büyük dosyalar (Gigabyte boyutunda log dosyaları vb.) için kullanılması, yüksek bellek tüketimine ve performans sorunlarına yol açabilir. Büyük dosyalarla çalışırken satır satır okuma imkanı sunan StreamReader sınıfı tercih edilmelidir. Bunu ileride öğreneceğiz. Dilerseniz kendi başınıza bunu araştırabilirsiniz.

DOSYAYA VERİ EKLEME: File.AppendAllText()

Bu metot, mevcut bir dosyanın içeriğini silmeden, belirtilen metni dosyanın **sonuna eklemek** için kullanılır.

Çalışma Mantığı:

1. Eğer belirtilen yolda dosya **yoksa**, WriteAllText gibi davranır: dosyayı oluşturur ve metni içine yazar.
2. Eğer belirtilen yolda dosya **varsa**, mevcut içeriği korur ve yeni metni dosyanın en sonuna ekler.

Sözdizimi:
File.AppendAllText(string path, string contents);

Örnek:

```
string logDosyasi = @"C:\CSharpDersleri\gunluk.log";
string yeniLogMesaji = $"{DateTime.Now}: Kullanıcı sisteme giriş yaptı.\n";

try
{
    // Dosyanın sonuna yeni log mesajını ekliyoruz.
    File.AppendAllText(logDosyasi, yeniLogMesaji);

    // İşlemi tekrarlayalım.
    yeniLogMesaji = $"{DateTime.Now}: Kullanıcı profili güncelledi.\n";
    File.AppendAllText(logDosyasi, yeniLogMesaji);
```

```
    Console.WriteLine("Log dosyasına yeni kayıtlar eklendi.");
}
catch (Exception ex)
{
    Console.WriteLine($"Loglama sırasında hata: {ex.Message}");
}
```

İşlem Sonucu (gunluk.log dosyasının içeriği):

29.06.2025 15:30:15: Kullanıcı sisteme giriş yaptı.

29.06.2025 15:30:18: Kullanıcı profili güncelledi.

Not: File.AppendAllText() özellikle loglama (işlem günlüğü tutma) gibi, geçmiş verinin korunması gereken ve sürekli yeni veri eklenen senaryolar için idealdir.

BONUS BİLGİLER

DOSYA YOLLARI VE @ İŞARETİNİN KULLANIMI

Windows işletim sisteminde dosya yolları \ (ters taksim) karakteri ile ayrılır. Ancak C#'ta \ karakteri, özel karakterleri (kaçış dizileri - escape sequences) belirtmek için kullanılır (örneğin \n yeni satır, \t sekme anlamına gelir).

Bu nedenle, bir dosya yolunu string içinde normal şekilde yazmaya çalışırsanız derleyici hata verir:
string yol = "C:\YeniKlasor\notlar.txt"; // HATA!

Bunu çözmenin iki yolu vardır:

1. **Çift Ters Taksim Kullanımı:** Her \ karakterini, kendisinden sonra gelenin özel bir anlamı olmadığını belirten ikinci bir \ ile yazmak.
   string yol = "C:\\YeniKlasor\\notlar.txt"; // Geçerli ama okunması zor.
2. **@ (Verbatim String Literal) Kullanımı:** String'in başına @ işareti koymak. Bu, derleyiciye string içindeki hiçbir karakteri özel olarak yorumlamamasını, her şeyi olduğu gibi kabul etmesini söyler. Bu yöntem daha modern, temiz ve yaygın olarak tercih edilendir.
   string yol = @"C:\YeniKlasor\notlar.txt"; // GEÇERLİ ve TAVSİYE EDİLEN.

Göreli (Relative) ve Özel Klasör Yolları

- **Mutlak Yol (Absolute Path):** @"C:\Users\KullaniciAdi\Desktop\dosya.txt" gibi kök dizinden başlayan tam yoldur.

- **Göreli Yol (Relative Path):** @"data\config.json" gibi, programın çalıştığı dizine göreli olan yoldur. Genellikle proje içindeki dosyalara erişmek için kullanılır.
  Programınız bin\Debug altından çalışıyorsa, bu yol bin\Debug\data\config.json anlamına gelir.
- **Özel Klasörler:** .NET size Masaüstü, Belgelerim gibi özel klasörlerin yolunu dinamik olarak alma imkanı sunar.

```
// Kullanıcının masaüstü yolunu alalım.

string masaustuYolu =
Environment.GetFolderPath(Environment.SpecialFolder.Desktop);

string tamYol = Path.Combine(masaustuYolu, "rapor.txt");

Console.WriteLine(tamYol); // Çıktı örn:
C:\Users\KullaniciAdi\Desktop\rapor.txt
```

HATA YÖNETİMİ: try-catch İLE GÜVENLİ DOSYA İŞLEMLERİ

Dosya işlemleri, programın kontrolü dışındaki birçok faktöre bağlı olduğu için istisna fırlatmaya çok müsaittir:

- Dosya kullanımda olabilir (IOException).
- Programın o yola yazma izni olmayabilir (UnauthorizedAccessException).
- Disk dolu olabilir (IOException).
- Yol geçersiz olabilir (DirectoryNotFoundException, PathTooLongException).

Bu nedenle, **TÜM** dosya işlemleri istisnasız bir şekilde try-catch bloğu içine alınmalıdır.

```
// Geçersiz bir yolu okumaya çalışarak hata senaryosu oluşturalım.

string gecersizYol = @"Z:\varolmayan_dizin\gizli_belge.txt";


try
{
    Console.WriteLine("Dosya okunmaya çalışılıyor...");
    string icerik = File.ReadAllText(gecersizYol);
    Console.WriteLine(icerik);
}
catch (DirectoryNotFoundException ex)
{
    Console.WriteLine("Hata: Belirtilen dizin bulunamadı!");
```

```
    Console.WriteLine($"Detay: {ex.Message}");
}
catch (FileNotFoundException ex)
{
    Console.WriteLine("Hata: Belirtilen dosya bulunamadı!");
    Console.WriteLine($"Detay: {ex.Message}");
}
catch (Exception ex) // Diğer tüm olası hatalar için genel bir yakalayıcı
{
    Console.WriteLine("Beklenmedik bir dosya hatası oluştu!");
    Console.WriteLine($"Detay: {ex.Message}");
}
finally
{
    Console.WriteLine("Dosya okuma operasyonu denemesi tamamlandı.");
}
```

Önemli Tavsiye: Profesyonel uygulamalarda, dosya işlemleri gibi harici kaynaklara bağımlı olan operasyonlar asla try-catch bloğu olmadan yazılmaz. Bu, uygulamanızın beklenmedik durumlarda çökmesini engelleyerek kararlı ve güvenilir olmasını sağlar.