C#

WEB & YAZILIM GELİŞTİRME SERİSİ · MODÜL 11

# C#

Statik tipli, .NET dili: değişkenler ve tipler, koşullar, döngüler, diziler ve metotlar; sınıflar, özellikler, kalıtım, arayüzler ve çok biçimlilik; tip güvenliği, koleksiyonlar, LINQ, hata yönetimi ve async; .NET, ASP.NET, veritabanı ve güvenlik. Özgün diyagramlar ve gerçek C# örnekleriyle.

Tipler · OOP · LINQ · .NET · Eğitim amaçlıdır

# Bu Kitap Hakkında

Bu modül, Microsoft'un modern, statik tipli ve güçlü nesne yönelimli dili C#'ı .NET platformuyla birlikte sıfırdan öğretir. PHP ve Python'dan sonra üçüncü dil olarak, derlenen ve tip güvenli bir dilin getirdiği yapıyı ve güvenliği gösterir; aynı programlama kavramlarının nasıl daha katı ve açık bir biçimde ifade edildiğini ortaya koyar. Dört seviye ve yirmi altı bölüm boyunca Console.WriteLine, statik tipler, operatörler, koşullar, döngüler, diziler, koleksiyonlar ve metotlardan; sınıflar, özellikler (properties), kurucular, kalıtım, arayüzler ve çok biçimliliğe; tip güvenliği, generics, LINQ, hata yönetimi, dosyalar ve asenkron programlamaya; .NET ekosistemi, NuGet, ASP.NET, veritabanı erişimi, güvenlik ve yayınlamaya kadar uzanır.

Her bölümde konuyu görselleştiren özgün bir diyagram (kod → konsol çıktısı panelleri, statik tip tabloları, sınıf kutuları, akış şemaları), derlenebilir C# örnekleri, 'konsola ne yazar?' kartı ve bir alıştırma yer alır. Güvenlik ve sorumlu kullanım baştan sona vurgulanır: girdi doğrulama, parametreli sorgular (SQL enjeksiyonuna karşı), sırların yapılandırmada gizlenmesi ve şifrelerin hash'lenmesi. Modül, üç dili (PHP, Python, C#) tamamlayarak 'kavramlar ortaktır, sözdizimi değişir' ilkesini pekiştirir. Bu, on altı modüllük 'Web & Yazılım Geliştirme' serisinin on birinci modülüdür. Bu seri eğitim amaçlıdır.

Web & Yazılım Geliştirme Serisi · Modül 11

# İçindekiler

SEVİYE 1

# C#'a Giriş

Temeller: C# ve .NET nedir, ilk program, değişkenler ve statik tipler, operatörler, koşullar, döngüler, diziler ve koleksiyonlar, metotlar.

BÖLÜM 01

# C# ve .NET Nedir?

C#, Microsoft tarafından geliştirilen, modern, güçlü ve nesne yönelimli bir programlama dilidir. .NET platformu üzerinde çalışır; web, masaüstü, oyun (Unity) ve kurumsal uygulamalarda yaygındır. PHP ve Python'dan en büyük farkı: C# derlenen ve statik tipli bir dildir.

## C#'ı farklı kılan ne?

- **Derlenir:** kod, çalışmadan önce derleyiciden geçer (hatalar erken yakalanır).
- **Statik tipli:** her değişkenin sabit, belli bir tipi vardır.
- **.NET üzerinde çalışır:** zengin, olgun bir platform ve kütüphane.
- **Çok yönlü:** web (ASP.NET), masaüstü, oyun (Unity), bulut.


Şema 1.1 — C# derlenen bir dildir: yaz → derle → .NET çalıştırır.

**İPUCU**

PHP ve Python'da kodu doğrudan çalıştırıyordun (yorumlanan diller); C#'ta arada bir **derleme** adımı vardır. Bu fazladan adım bir avantajdır: derleyici, kod çalışmadan **önce** birçok hatayı (yanlış tip, yazım hatası, eksik tanım) yakalar — "çalıştırınca patlamak" yerine "derlerken uyarı almak". **Statik tipleme** de aynı felsefenin parçasıdır: bir değişkeni `int` tanımladıysan ona metin atayamazsın; bu, büyük projelerde hata oranını ciddi biçimde düşürür. C#, "güvenlik ve yapı" tarafında duran bir dildir.

**Konsola ne yazar?** ÇIKTI

C# kodu önce derlenir (denetlenir ve .NET'in anlayacağı ara koda çevrilir), sonra .NET çalışma zamanı tarafından çalıştırılır. Bir konsol uygulamasında sonuç terminalde görünür; bir web uygulamasında tarayıcıya, bir masaüstü uygulamasında pencereye gider. Derleme adımı, programın daha güvenli ve hızlı çalışmasını sağlar.

**Alıştırma** 8 dk

C#'ı konumla:

1. C#'ın "derlenen" ve "statik tipli" olmasının ne demek olduğunu yaz.
2. Derleme adımının sağladığı bir avantajı belirt.
3. C# ile Python'ın ortak yanı ve temel farkı nedir, kısaca yaz.

BÖLÜM 02

# İlk Program: Console.WriteLine

C#'ta ekrana yazı yazdırmanın yolu Console.WriteLine'dır. Modern C#'ta basit programlar çok sadeleşmiştir; tek satırla bir program yazabilirsin. Geleneksel yapıda ise kod bir Main metodunun içinde yer alır.

## Console.WriteLine ile çıktı

- `Console.WriteLine("metin")` — satır yazdırır.
- `Console.Write(...)` — satır atlamadan yazar.
- Her ifade **noktalı virgülle (;)** biter.


Şema 2.1 — Console.WriteLine, kodu derleyip çalıştırınca konsola yazar.

**İlk C# programın**

```
// Modern (üst düzey) — tek satır yeterli
Console.WriteLine("Merhaba, dünya!");

// Geleneksel yapı
class Program {
    static void Main() {
        Console.WriteLine("Merhaba!");
    }
}
```

**İPUCU**

C#'ta her ifade **noktalı virgülle** biter (Python'dan farklı, PHP'ye benzer) ve kod blokları **süslü parantezle** `{ }` tanımlanır (Python'ın girintisinden farklı). Geleneksel olarak her program bir `Main` metoduyla başlar — programın giriş noktasıdır. Modern C# (.NET 6+) bu kalıbı basitleştirdi: küçük programlarda doğrudan üst düzeyde kod yazabilirsin. `WriteLine` satır sonuna geçer, `Write` ise aynı satırda devam eder.

### Konsola ne yazar? ÇIKTI

`Console.WriteLine("Merhaba, dünya!");` kodu derlenip çalıştırıldığında konsola "Merhaba, dünya!" satırını yazar. Geleneksel yapıda aynı satır `Main` metodunun içinde yer alır; modern C#'ta tek başına da çalışır. Sonuç aynıdır: konsolda bir satır metin.

### Alıştırma 10 dk

İlk programı yaz:

1. Kendini tanıtan üç satırlık bir C# kodu yaz (WriteLine ile).
2. WriteLine ile Write arasındaki farkı bir örnekle göster.
3. Main metodunun görevini kendi cümlenle açıkla.

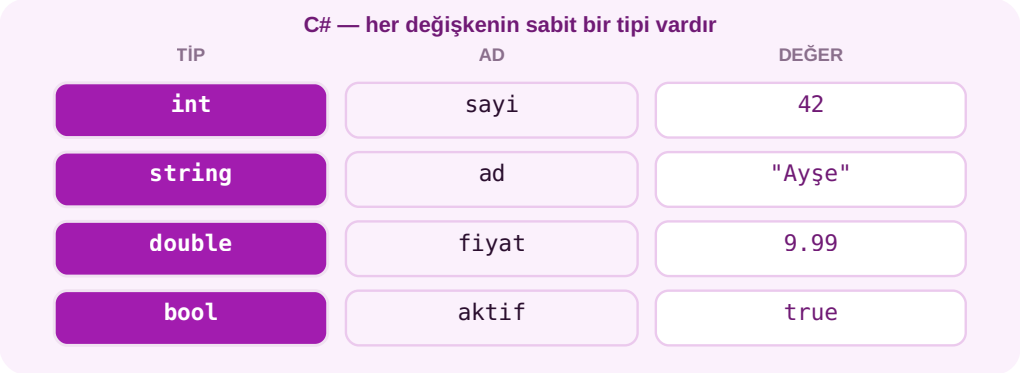
BÖLÜM 03

# Değişkenler ve Tipler

C#'ın kalbinde statik tipleme vardır: her değişken tanımlanırken bir tip alır ve o tipte kalır. Bu, C#'ı PHP ve Python'dan ayıran en temel özelliktir. Tipi açıkça yazabilir veya var ile derleyiciye çıkarttırabilirsin.

## Statik tipli değişkenler

- **Tip + ad + değer:** `int sayi = 42;`.
- Temel tipler: `int`, `double`, `string`, `bool`.
- `var` — tipi değerden çıkarır (yine de statiktir).
- Bir kez tip belirlendi mi, değişmez: `int`'e metin atayamazsın.




Şema 3.1 — C#'ta her değişkenin sabit, açık bir tipi vardır.

**Değişkenler ve tipler**

```
 int sayi = 42;
string ad = "Ayşe";
double fiyat = 9.99;
bool aktif = true;

var sehir = "Ankara";   // var: tip "string" olarak çıkarılır
// sayi = "metin";      // HATA: int'e metin atanamaz
```

**İPUCU**

Statik tipleme ilk başta "fazladan yazı" gibi görünebilir, ama büyük bir güvenlik sağlar: bir değişkene yanlış türde değer atamaya çalışırsan, program çalışmadan **derleyici seni uyarır**. Python/PHP'de bu hata ancak çalışma anında (belki kullanıcının önünde) ortaya çıkardı. `var` anahtar kelimesi yazımı kısaltır ama tipi ortadan kaldırmaz — derleyici tipi değerden çıkarır ve yine sabittir. C# isimlendirme geleneği: yerel değişkenler `camelCase`, tipler ve metotlar `PascalCase`.

### Konsola ne yazar? ÇIKTI

`int sayi = 42;` "sayi" adında, tipi kalıcı olarak `int` olan bir değişken oluşturur. Ona daha sonra `"metin"` atamaya çalışırsan kod **derlenmez** — hata alırsın. `var sehir = "Ankara"` ise tipi (string) otomatik çıkarır. Tablodaki her satır, bir tip–ad–değer üçlüsüdür.

### Alıştırma 12 dk

Tipli değişken kullan:

1. int, string, double ve bool tipinde birer değişken tanımla ve yazdır.
2. var ile bir değişken tanımla; tipinin ne olacağını söyle.
3. Statik tiplemenin sağladığı bir avantajı kendi cümlenle açıkla.

BÖLÜM 04

# Operatörler ve İfadeler

Operatörler değerler üzerinde işlem yapar. C#'ta metinleri birleştirmenin en şık yolu, Python'daki f-string'e çok benzeyen string interpolation'dır: $"..." içinde değişkenleri süslü parantezle gömersin.

## Operatörler ve string interpolation

- Aritmetik: `+ - * / %`.
- Karşılaştırma: `== != > < >= <=`.
- Mantıksal: `&&` (ve), `||` (veya), `!` (değil).
- String interpolation: `$"Merhaba {ad}"`.


Şema 4.1 — $"..." ile değişkenler metnin içine gömülür (interpolation).

**Operatörler ve interpolation**

```
int a = 10, b = 3;
Console.WriteLine(a + b);      // 13
Console.WriteLine(a % b);      // 1  (kalan)
Console.WriteLine(a / b);      // 3  (tam sayı bölme!)
Console.WriteLine(a / 3.0);    // 3.33 (ondalık)

string ad = "Ayşe";
Console.WriteLine($"Ad: {ad}");
```

**İPUCU**

**String interpolation** ( `$"...{degisken}..."` ), C#'ta metin oluşturmanın en okunur yoludur — Python'ın f-string'inin neredeyse aynısıdır. Tip konusunda bir tuzağa dikkat: iki `int`'i bölersen sonuç da `int` olur ve ondalık kısım atılır ( `10 / 3` → 3, 3.33 değil!). Ondalık sonuç istiyorsan en az bir tarafı `double` yap ( `10 / 3.0` ). Bu, statik tipli dillerde sık karşılaşılan ve dikkat gerektiren bir davranıştır.

**Konsola ne yazar?** ÇIKTI

`$"{ad}, {yas}"` ifadesi, `{ad}` ve `{yas}` yerine değişkenlerin değerlerini koyarak "Ayşe, 30" üretir. Aritmetikte tipe dikkat: `int / int` tam sayı bölme yapar (kalanı atar), ondalık için bir tarafı `double` olmalıdır.

**Alıştırma** 10 dk

Operatör kullan:

1. İki tam sayının bölümünü hem int hem double olarak hesapla, farkı gör.
2. String interpolation ile ad ve şehri tek cümlede yazdır.
3. 10 / 3'ün neden 3 verdiğini açıkla.

BÖLÜM 05

# Koşullar: if / else / switch

Koşullar programa karar verdirir. C#'ta if/else if/else yapısının yanında, çok sayıda durumu düzenli ele almak için switch deyimi de vardır. Bloklar süslü parantezle tanımlanır.

## if / else if / else ve switch

- `if (koşul) { ... }` — koşul doğruysa çalışır.
- `else if` / `else` — ek dallar.
- `switch` — bir değeri birçok olası duruma karşı sınar.


Şema 5.1 — if/else: koşula göre süslü parantezli bloklardan biri çalışır.

**if / else if / else ve switch**

```
 int not = 75;
if (not >= 85) {
    Console.WriteLine("Pekiyi");
} else if (not >= 60) {
    Console.WriteLine("Geçer");
} else {
    Console.WriteLine("Kaldı");
}

switch (gun) {
    case "Pzt": Console.WriteLine("Hafta başı"); break;
    default:    Console.WriteLine("Diğer"); break;
}
```

**İPUCU**

C#'ta koşullar süslü parantezle yazılır ve mantıksal operatörler PHP'deki gibidir ( `&&` , `||` , `!` ) — Python'ın `and/or/not` 'undan farklı. `switch` , bir değeri çok sayıda olası değere karşı sınarken if/else if zincirinden daha okunurdur; her `case` sonunda `break` unutma (yoksa derleyici uyarır). C#'ın koşulları "tip güvenli"dir: koşul mutlaka `bool` olmalıdır — bazı dillerdeki gibi sayıyı doğrudan koşul yapamazsın, bu da kazara hataları önler.

**Konsola ne yazar?** ÇIKTI

`not = 75` için kod sırayla bakar: 85'ten büyük mü (hayır), 60'tan büyük mü (evet) → "Geçer" yazdırılır. `switch` ise bir değeri (örn. günü) tek tek durumlara karşı sınar ve eşleşen `case` 'i çalıştırır. İkisi de programın koşula göre farklı yol izlemesini sağlar.

**Alıştırma** 12 dk

Koşul yaz:

1. Bir notu harf karşılığına çeviren if/else if/else yaz.
2. Bir switch ile haftanın gününe göre mesaj yazdır.
3. Koşulun neden bool olması gerektiğini açıkla.

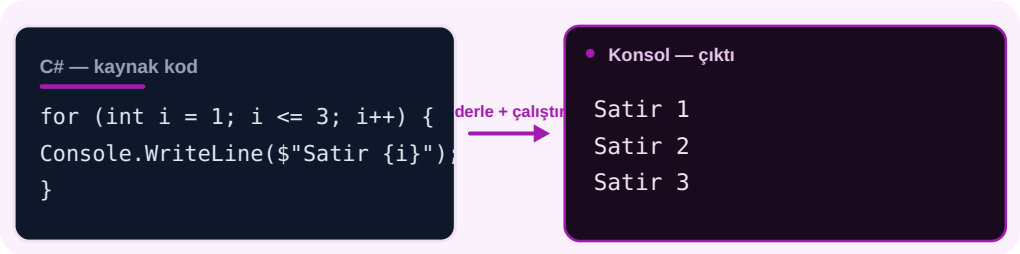
BÖLÜM 06

# Döngüler: for / while / foreach

Döngüler bir işi tekrar ettirir. C#'ta üç temel döngü vardır: sayaçlı for, koşullu while ve bir koleksiyonun her elemanını gezen foreach. Üçü de süslü parantezli bloklarla çalışır.

## for / while / foreach

- `for (int i = 0; i < 5; i++)` — sayaçlı tekrar.
- `while (koşul)` — koşul doğru oldukça.
- `foreach (var x in koleksiyon)` — her eleman için.




Şema 6.1 — for döngüsü: gövdeyi belirli sayıda çalıştırır.

**for, while ve foreach**

```
// for
for (int i = 1; i <= 3; i++) {
    Console.WriteLine(i);
}

// foreach: koleksiyonun her elemanı
string[] renkler = { "kırmızı", "yeşil" };
foreach (var renk in renkler) {
    Console.WriteLine(renk);
}
```

**İPUCU**

`foreach`, bir dizi veya listenin elemanlarını dolaşmanın en temiz yoludur (Python'ın `for x in`'ine, PHP'nin `foreach`'ine benzer). Klasik `for` ise sayaç üzerinde tam kontrol gerektiğinde kullanılır. Döngü değişkenini `foreach (var renk in ...)` derken `var` ile tanımlamak yaygındır — derleyici tipi koleksiyondan çıkarır. En sık `for` hatası, koşulu yanlış kurup **sonsuz döngüye** girmek veya dizi sınırını aşmaktır (C# bunu çalışma anında yakalayıp hata fırlatır).

### Konsola ne yazar? ÇIKTI

`for (int i = 1; i <= 3; i++)` döngüsü i'yi 1, 2, 3 yaparak gövdeyi üç kez çalıştırır ve "Satir 1/2/3" yazdırır. `foreach` ise bir koleksiyonun her elemanını sırayla alıp işler. İkisi de tekrar eden işi tek bir blokla ifade eder.

### Alıştırma 12 dk

Döngü yaz:

1. 1'den 10'a sayıları yazdıran bir for döngüsü yaz.
2. Bir string dizisini foreach ile dolaşıp yazdır.
3. for ile foreach'i hangi durumda tercih edersin, açıkla.

BÖLÜM 07

# Diziler ve Koleksiyonlar

Birden çok değeri bir arada tutmak için C# diziler ve koleksiyonlar sunar. Dizi (array) sabit boyutludur; List ise dinamik olarak büyüyüp küçülebilir. C#'ın tip güvenliği burada da geçerlidir: bir List bir tek tipte eleman tutar.

## Dizi ve List

- **Dizi:** `int[] sayilar = {1, 2, 3};` — sabit boyut.
- **List:** `List<string> adlar = new();` — dinamik.
- `.Add(...)` ile listeye eleman eklenir.
- Erişim indeksle: `sayilar[0]`.


Şema 7.1 — Dizi sabit boyutlu; List dinamik ve tip güvenlidir.

**Dizi ve List**

```
// Dizi: sabit boyut
int[] sayilar = { 10, 20, 30 };
Console.WriteLine(sayilar[0]);   // 10

// List: dinamik, tip güvenli
List<string> adlar = new();
adlar.Add("Ayşe");
adlar.Add("Veli");
Console.WriteLine(adlar.Count);  // 2
```

**İPUCU**

`List<T>` yazımındaki `<string>`, "bu liste yalnızca string tutar" demektir — buna **generics** (genel tipler) denir ve C#'ın tip güvenliğinin güçlü bir parçasıdır. Listeye yanlış tipte eleman eklemeye çalışırsan derleyici engeller. Sabit, boyutu belli veriler için **dizi**; sürekli değişen (eklenen/çıkarılan) veriler için **List** kullan. `List`, PHP'nin dizilerine ve Python'ın listelerine benzer ama tip güvenlidir. İndeksler 0'dan başlar; sınır dışı erişim çalışma anında hata verir.

**Konsola ne yazar?** ÇIKTI

`int[] sayilar = {10, 20, 30}` sabit boyutlu bir dizidir; `sayilar[0]` ilk elemanı (10) verir. `List<string>` ise boş başlar, `.Add(...)` ile büyür ve `.Count` eleman sayısını verir. Dizi sabit, List dinamiktir; ikisi de yalnızca belirtilen tipte eleman tutar.

**Alıştırma** 12 dk

Koleksiyon kullan:

1. Üç sayıdan oluşan bir dizi oluştur ve ikincisini yazdır.
2. Bir List<string> oluştur, iki eleman ekle ve sayısını yazdır.
3. Dizi ile List arasındaki farkı kendi cümlenle açıkla.

BÖLÜM 08

# Metotlar

Metot, bir işi yapan, isimlendirilmiş ve tekrar kullanılabilir bir kod bloğudur (diğer dillerdeki "fonksiyon"). C#'ta metotlar tip güvenlidir: hem parametrelerin hem de döndürdüğü değerin tipi bellidir.

## Tipli metotlar

- `dönüş_tipi Ad(tip parametre) { ... }`.
- `return` ile belirtilen tipte değer döndürür.
- `void` — değer döndürmeyen metot.


Şema 8.1 — Metot: tipli parametre alır, tipli bir değer döndürür.

**Metot tanımı**

```
// int alır, int döndürür
int Topla(int a, int b) {
    return a + b;
}

// değer döndürmez (void)
void Selamla(string ad) {
    Console.WriteLine($"Merhaba, {ad}");
}

Console.WriteLine(Topla(10, 20));  // 30
Selamla("Ayşe");
```

**İPUCU**

C# metotlarında **her şeyin tipi bellidir**: `int Topla(int a, int b)` ifadesi "iki int alır, bir int döndürür" der. Bu, metodu yanlış kullanmanı (örn. metin verip sayı beklemeni) derleyici aşamasında engeller — Python/PHP'de bu hata çalışma anında ortaya çıkardı. Değer döndürmeyen metotlar `void` ile işaretlenir. Metot ve sınıf isimleri C#'ta `PascalCase` (büyük harfle başlar) yazılır. Metotlar, Modül 6'daki "problemi parçalara bölme" ilkesinin C#'taki karşılığıdır.

### Konsola ne yazar? ÇIKTI

`Topla(3, 5)` çağrısı metodu çalıştırır, `3 + 5` hesaplar ve `return` ile 8 döndürür; `Console.WriteLine` bunu yazdırır. `Selamla(...)` ise bir şey döndürmez (void), doğrudan ekrana yazar. Metodun parametre ve dönüş tipleri sabittir; yanlış tip verirsen kod derlenmez.

### Alıştırma 12 dk

Metot yaz:

1. İki sayının çarpımını döndüren (int) bir metot yaz ve çağır.
2. Bir ismi alıp selam yazan bir void metot yaz.
3. void ile değer döndüren metot arasındaki farkı açıkla.

SEVİYE 2

# Nesne Yönelimli C#

C#'ın kalbi OOP: sınıflar ve nesneler, özellikler (properties), kurucular, kalıtım, arayüzler ve çok biçimlilik.

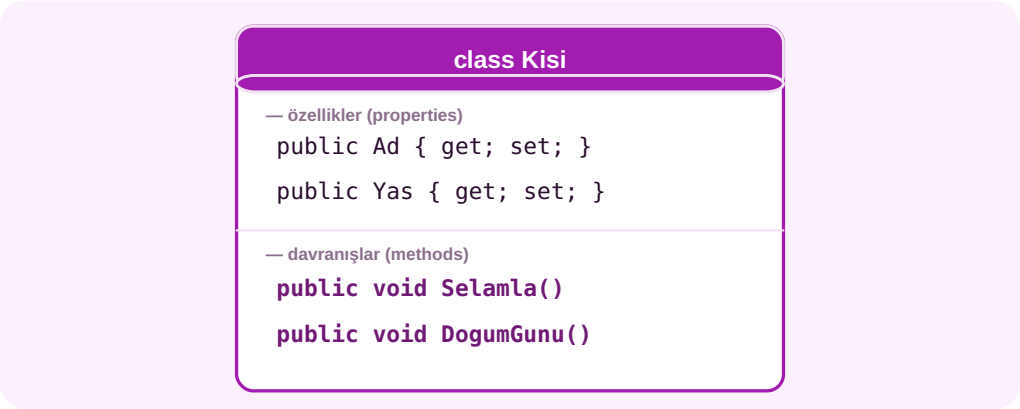
BÖLÜM 09

# Sınıflar ve Nesneler

Nesne yönelimli programlama (OOP) C#'ın merkezindedir. Bir sınıf (class), ilgili veriyi (alanlar/özellikler) ve davranışları (metotlar) tek bir kalıpta birleştirir. Nesne (object) ise bu kalıptan üretilen somut bir örnektir.

## Sınıf ve nesne

- **Sınıf:** bir kalıp — özellikler + metotlar.
- **Nesne:** sınıftan `new` ile üretilen örnek.
- `nesne.Ozellik` ve `nesne.Metot()` ile erişilir.




Şema 9.1 — Bir sınıf: özellikler (veri) ve metotlar (davranış) bir arada.

**Sınıf ve nesne**

```
class Kisi {
    public string Ad;
    public int Yas;
    public void Selamla() {
        Console.WriteLine($"Merhaba, {Ad}");
    }
}

Kisi k = new Kisi();
k.Ad = "Ayşe";
k.Selamla();              // Merhaba, Ayşe
```

**İPUCU**

OOP, Modül 6'daki **soyutlama** ve "ilgili şeyleri bir arada tutma" ilkelerinin C#'taki güçlü hâlidir. PHP ve Python'da da sınıflar gördün; C#'ın farkı, OOP'nin dilin **her yerine** işlemiş olmasıdır — neredeyse her şey bir nesnedir. Bir sınıf bir kez tanımlanır, ondan `new` ile istediğin kadar nesne üretirsin; her nesne kendi verisini taşır. C# geleneğinde sınıf adları ve genel üyeler `PascalCase` yazılır. Sonraki bölümlerde alanları doğrudan açmak yerine **özellikler (properties)** kullanmanın neden daha iyi olduğunu göreceksin.

**Konsola ne yazar?** ÇIKTI

`new Kisi()`, sınıf kalıbından somut bir nesne üretir; `k.Ad = "Ayşe"` ile verisini ayarlar, `k.Selamla()` ile davranışını çağırırsın — konsola "Merhaba, Ayşe" yazar. Aynı sınıftan farklı nesneler (farklı kişiler) üretebilir, her birine kendi verisini verebilirsin.

**Alıştırma** 14 dk

Sınıf yaz:

1. Bir "Urun" sınıfı tasarla: alanları (Ad, Fiyat) ve bir metodu.
2. Sınıftan iki nesne üret ve metodunu çağır.
3. Sınıf ile nesne arasındaki farkı kendi cümlenle açıkla.

BÖLÜM 10

# Özellikler (Properties)

C#'a özgü ve çok kullanılan bir özellik: properties. Bir alanı doğrudan dışarı açmak yerine, ona kontrollü erişim sağlayan get/set bloklarıyla sararsın. Böylece veriyi okurken/yazarken araya kural koyabilirsin.

## get ve set

- `public string Ad { get; set; }` — otomatik özellik.
- **get:** değeri okur. **set:** değeri yazar.
- set içinde doğrulama yapabilirsin (örn. negatif yaşı engelle).


Şema 10.1 — Özellik: set ile yaz, get ile oku — kontrollü erişim.

**Özellikler (properties)**

```
class Kisi {
    // Otomatik özellik
    public string Ad { get; set; }

    // Kurallı özellik
    private int _yas;
    public int Yas {
        get => _yas;
        set => _yas = (value < 0) ? 0 : value;
    }
}
```

**İPUCU**

Özellikler, OOP'nin **kapsülleme** (encapsulation) ilkesinin C#'taki şık aracıdır: verinin nasıl okunup yazıldığını sınıf kontrol eder. Basit durumlarda `{ get; set; }` (otomatik özellik) yeterlidir; kural gerektiğinde `set` bloğuna doğrulama eklersin (örn. negatif yaşı sıfıra çekmek). `value`, `set` içinde "atanmak istenen değer" anlamına gelen özel bir kelimedir. Yalnızca okunabilir özellik için `{ get; }`, dışarıdan değiştirilemeyen ama içeride ayarlanabilen için `{ get; private set; }` kullanılır. Bu, alanları doğrudan `public` yapmaktan çok daha güvenlidir.

**Konsola ne yazar?** ÇIKTI

`k.Ad = "Ayşe"` aslında özelliğin `set` bloğunu, `Console.WriteLine(k.Ad)` ise `get` bloğunu çalıştırır — dışarıdan basit bir değişken gibi görünse de araya kural koyabilirsin. `Yas` özelliğine negatif bir değer atanırsa, set bloğu onu sıfıra çevirir; veri her zaman geçerli kalır.

**Alıştırma** 12 dk

Özellik yaz:

1. Bir sınıfa otomatik özellik (Ad) ekle.
2. Fiyat için, negatif değeri engelleyen kurallı bir set yaz.
3. Özelliğin alanı doğrudan public yapmaya göre avantajını açıkla.

BÖLÜM 11

# Kurucular ve this

Bir nesne oluşturulurken başlangıç değerlerini ayarlamak için kurucu (constructor) kullanılır. Kurucu, sınıfla aynı adı taşıyan özel bir metottur ve nesne new ile üretilirken otomatik çalışır.

## Kurucu ve this

- Kurucu, sınıfla **aynı adı** taşır; `new` ile çalışır.
- Başlangıç değerlerini parametre olarak alır.
- `this` — nesnenin kendisine işaret eder.


Şema 11.1 — Kurucu: nesne üretilirken başlangıç değerlerini ayarlar.

**Kurucu (constructor)**

```
class Kisi {
    public string Ad { get; set; }
    public int Yas { get; set; }

    // Kurucu: sınıfla aynı ad
    public Kisi(string ad, int yas) {
        this.Ad = ad;
        this.Yas = yas;
    }
}

var k = new Kisi("Ayşe", 30);
```

**İPUCU**

Kurucu, bir nesnenin **geçerli bir başlangıç durumuyla** doğmasını sağlar: `new Kisi("Ayşe", 30)` dediğinde, Ad ve Yas daha en baştan ayarlanmış olur — "yarım kurulmuş" nesne riski azalır. `this` , parametre adıyla özellik adı aynı olduğunda hangisinin hangisi olduğunu netleştirir ( `this.Ad = ad` ): "bu nesnenin Ad'ı, parametredeki ad olsun". PHP'deki `$this` ve Python'daki `self` ile aynı fikirdir. Bir sınıfta birden çok kurucu olabilir (farklı parametrelerle) — buna aşırı yükleme (overloading) denir.

**Konsola ne yazar?** ÇIKTI

`new Kisi("Ayşe", 30)` kurucuyu çağırır; kurucu, gelen "Ayşe" ve 30 değerlerini `this.Ad` ve `this.Yas`'a atar. Böylece nesne, oluşturulduğu anda geçerli verilerle hazır olur. `k.Ad` okunduğunda "Ayşe" döner.

**Alıştırma** 12 dk

Kurucu yaz:

1. Bir "Araba" sınıfına marka ve hız alan bir kurucu yaz.
2. Kurucu içinde this kullanarak değerleri ata.
3. Kurucunun nesneyi "geçerli başlangıç durumu" ile kurmasının faydasını açıkla.

BÖLÜM 12

# Kalıtım (Inheritance)

Kalıtım, bir sınıfın başka bir sınıfın özelliklerini ve davranışlarını devralmasıdır. Ortak olanı bir üst sınıfta toplar, özel olanı alt sınıflarda eklersin — kod tekrarını önlersin. C#'ta bir sınıf : ile üst sınıfını belirtir.

## Üst ve alt sınıf

- `class Kopek : Hayvan` — Hayvan'dan miras alır.
- Alt sınıf, üstün üyelerini otomatik kullanır.
- `base` ile üst sınıfa erişilir.


Şema 12.1 — Kalıtım: ortak üyeler üst sınıfta, özel olanlar alt sınıflarda.

**Kalıtım**

```
class Hayvan {
    public string Ad { get; set; }
    public void Bilgi() =>
        Console.WriteLine($"Ben {Ad}");
}

class Kopek : Hayvan {       // miras alır
    public void Havla() =>
        Console.WriteLine("Hav!");
}

var k = new Kopek { Ad = "Karabaş" };
k.Bilgi();                   // Ben Karabaş (devralındı)
```

**İPUCU**

Kalıtım, "bir X bir Y türüdür" ilişkisini ifade eder: bir köpek bir hayvandır, bu yüzden `class Kopek : Hayvan` doğaldır. Üst sınıftaki kod alt sınıflarda **tekrar yazılmadan** kullanılır. C#'ta bir sınıf yalnızca **tek** bir sınıftan miras alabilir (çoklu kalıtım yoktur) — bu kısıtın yarattığı boşluğu bir sonraki bölümdeki **arayüzler** doldurur. Çok derin kalıtım zincirlerinden kaçın; gereğinden fazla kalıtım kodu kırılgan yapar. Modern tasarımda çoğu zaman arayüzler ve kompozisyon tercih edilir.

**Konsola ne yazar?** ÇIKTI

`Kopek` sınıfı `Hayvan`'dan miras aldığı için `Bilgi()` metodunu yeniden yazmadan kullanır: `k.Bilgi()` "Ben Karabaş" yazar. `Havla()` ise yalnızca Kopek'e özeldir. Ortak davranış üstte tanımlanır, özel davranış altta eklenir — tekrar önlenir.

**Alıştırma** 14 dk

Kalıtım kur:

1. Bir "Hayvan" üst sınıfı ve ondan türeyen bir alt sınıf yaz.
2. Üst sınıftaki bir metodun alt sınıfta kullanıldığını göster.
3. C#'ta neden tek kalıtım olduğunu ve bunu neyin tamamladığını yaz.

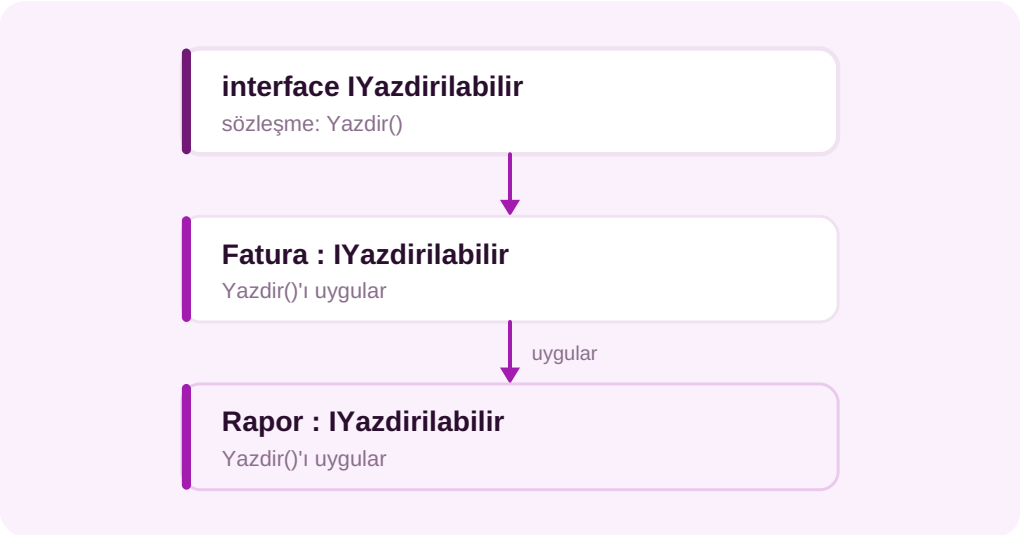
BÖLÜM 13

# Arayüzler (Interfaces)

Arayüz (interface), bir sınıfın "ne yapabileceğini" tanımlayan bir sözleşmedir — ama nasıl yapacağını söylemez. Bir sınıf bir arayüzü uyguladığında, o arayüzdeki tüm metotları sağlamayı taahhüt eder. Arayüzler, C#'ta esnek ve gevşek bağlı tasarımın anahtarıdır.

## Sözleşme olarak arayüz

- `interface IYazdirilabilir { void Yazdir(); }` — sözleşme.
- `class Fatura : IYazdirilabilir` — sözleşmeyi uygular.
- Bir sınıf **birden çok** arayüz uygulayabilir.




Şema 13.1 — Arayüz bir sözleşmedir; farklı sınıflar onu kendince uygular.

**Arayüz (interface)**

```
 interface IYazdirilabilir {
    void Yazdir();          // gövdesiz: sözleşme
}

class Fatura : IYazdirilabilir {
    public void Yazdir() =>
        Console.WriteLine("Fatura yazdırıldı");
}

IYazdirilabilir nesne = new Fatura();
nesne.Yazdir();             // Fatura yazdırıldı
```

**İPUCU**

Arayüzleri bir **sözleşme** olarak düşün: "Beni uygulayan her sınıf şu metotları sağlamalı." Bu, kodun **gevşek bağlı** olmasını sağlar — bir metot, somut bir sınıf yerine bir arayüz isteyebilir ( `void Yazdir(IYazdirilabilir x)` ) ve o arayüzü uygulayan **herhangi bir** sınıfla çalışır. Bu esneklik, test edilebilir ve değiştirilebilir kodun temelidir. Arayüz adları C#'ta geleneksel olarak `I` ile başlar (IYazdirilabilir, IComparable). Kalıtımdan farkı: bir sınıf tek bir sınıftan miras alır ama **birçok** arayüz uygulayabilir.

**Konsola ne yazar?** ÇIKTI

`IYazdirilabilir` arayüzü yalnızca "bir `Yazdir()` metodu olmalı" der; nasıl yazdırılacağını `Fatura` ve `Rapor` sınıfları kendi içinde belirler. `IYazdirilabilir nesne = new Fatura()` ile bir nesneyi arayüz tipi üzerinden tutup `Yazdir()` çağırırsın — hangi sınıf olursa olsun sözleşmeyi yerine getirir.

**Alıştırma** 14 dk

Arayüz yaz:

1. Bir ISekil arayüzü tanımla (örn. Alan() metodu).
2. Bu arayüzü uygulayan iki sınıf yaz (Daire, Kare).
3. Arayüz ile kalıtım arasındaki farkı kendi cümlenle açıkla.

BÖLÜM 14

# Çok Biçimlilik (Polymorphism)

Çok biçimlilik, aynı çağrının farklı nesnelerde farklı davranmasıdır. Bir üst sınıf metodunu alt sınıflar kendilerine göre yeniden tanımlar (override); sen tek bir çağrı yaparsın, doğru davranış nesnenin gerçek tipine göre seçilir. OOP'nin en güçlü fikirlerinden biridir.

## virtual ve override

- Üst sınıf: `virtual` ile metodu "geçersiz kılınabilir" yapar.
- Alt sınıf: `override` ile kendi davranışını verir.
- Aynı çağrı, nesneye göre farklı sonuç üretir.


Şema 14.1 — Çok biçimlilik: aynı çağrı, nesnenin tipine göre farklı davranır.

**virtual / override**

```
 class Hayvan {
    public virtual void SesCikar() =>
        Console.WriteLine("...");
}
class Kopek : Hayvan {
    public override void SesCikar() =>
        Console.WriteLine("Hav!");
}

Hayvan h = new Kopek();
h.SesCikar();               // Hav!  (Kopek'in davranışı)
```

**İPUCU**

Çok biçimliliğin gücü şudur: bir `List<Hayvan>` içinde farklı türden hayvanlar (Kopek, Kedi, Kus) tutabilir, hepsini tek bir `foreach` ile dolaşıp `SesCikar()` çağırabilirsin — her biri kendi sesini çıkarır. Sen "nasıl"ı bilmek zorunda değilsin; doğru davranış çalışma anında seçilir. Üst sınıfta metot `virtual` olmalı, alt sınıfta `override` ile değiştirilir. Bu, `if/else` zincirleriyle tip kontrol etmekten çok daha temiz ve genişletilebilir bir tasarımdır — yeni bir hayvan türü eklemek, var olan kodu değiştirmeden mümkün olur.

**Konsola ne yazar?** ÇIKTI

`Hayvan h = new Kopek()` ile değişkenin tipi Hayvan ama gerçek nesne bir Kopek'tir. `h.SesCikar()` çağrıldığında C#, nesnenin gerçek tipine (Kopek) bakar ve onun override ettiği davranışı ("Hav!") çalıştırır. Aynı çağrı, Kedi nesnesinde "Miyav!" üretirdi — davranış nesneye göre değişir.

**Alıştırma** 14 dk

Çok biçimlilik kur:

1. Bir üst sınıfta virtual bir metot, iki alt sınıfta override yaz.
2. Üst tip bir değişkenle farklı nesneleri tutup metodu çağır.
3. Bu yaklaşımın if/else tip kontrolüne göre avantajını açıkla.

SEVİYE 3

# Güçlü Araçlar

C#'ı güçlü kılan araçlar: tip güvenliği ve dönüşümler, koleksiyonlar, LINQ ile veri sorgulama, hata yönetimi, dosyalar ve asenkron programlama.

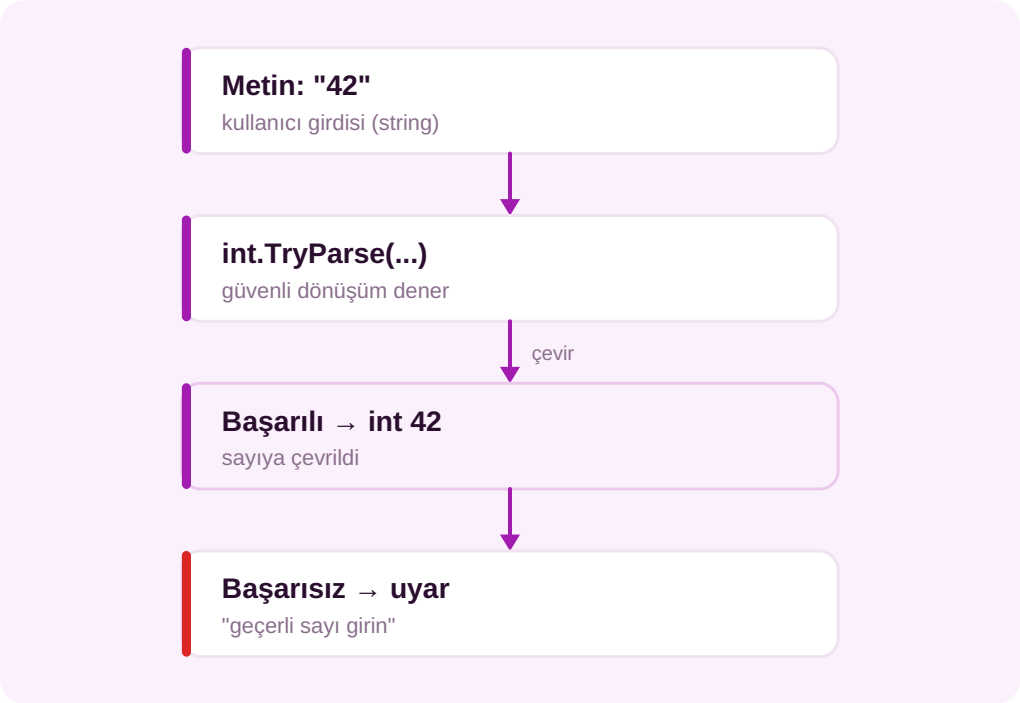
BÖLÜM 15

# Tip Güvenliği ve Dönüşümler

Statik tipleme güçlüdür ama bazen tipler arası geçiş gerekir: kullanıcıdan gelen metni sayıya çevirmek, bir tipi diğerine dönüştürmek gibi. C# bunu güvenli ve açık yollarla yapar; "olmayabilir" değerler için de nullable tipler sunar.

## Dönüşüm ve null

- `int.Parse("42")` — metni sayıya çevir (hata olabilir).
- `int.TryParse(...)` — güvenli çevir (başarı/başarısız).
- `int?` — null olabilen tip (değer yoksa).




Şema 15.1 — TryParse: dönüşümü güvenle dener, çökmeden sonucu bildirir.

**Tip dönüşümleri**

```
// Riskli: hatalı metinde çöker
int n = int.Parse("42");      // 42

// Güvenli: başarıyı bildirir
if (int.TryParse(girdi, out int sayi)) {
    Console.WriteLine(sayi);
} else {
    Console.WriteLine("Geçerli sayı girin.");
}

int? yas = null;              // null olabilen int
```

**İPUCU**

Kullanıcıdan veya dosyadan gelen veri her zaman metindir; onu sayıya çevirirken **`TryParse` tercih et**: `Parse` hatalı bir metinde programı çökertir, `TryParse` ise sessizce "başarısız" döner ve sen nazikçe baş edersin. `out` anahtar kelimesi, dönüşen değeri geri almanı sağlar. **Nullable tipler** ( `int?` , `string?` ) "bu değer olmayabilir" demenin tip-güvenli yoludur — modern C#, null kaynaklı hataları derleyici aşamasında uyararak azaltır. Tip güvenliği, "boş/yanlış değer" hatalarını kullanıcının önünde değil, kodu yazarken yakalamana yardım eder.

**Konsola ne yazar?** ÇIKTI

`int.TryParse(girdi, out int sayi)` , girdi geçerli bir sayıysa `true` döner ve sayıyı `sayi` 'ya yazar; değilse `false` döner ve program çökmez. Böylece "12" girilince 12 yazdırılır, "abc" girilince "Geçerli sayı girin." mesajı gösterilir. Dönüşümü her zaman güvenli yoldan yaparsın.

**Alıştırma** 12 dk

Dönüşüm yap:

1. Bir metni TryParse ile sayıya çeviren, hatayı ele alan kod yaz.
2. Parse ile TryParse arasındaki farkı açıkla.
3. int? (nullable) ne işe yarar, bir örnekle yaz.

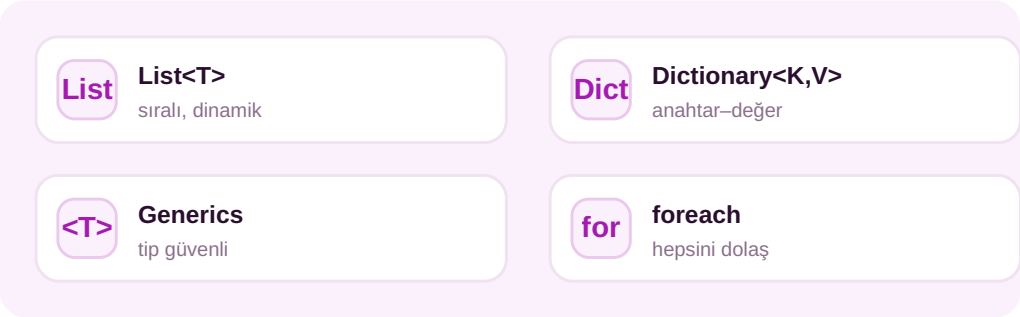
BÖLÜM 16

# Koleksiyonlar Derinlemesine

Diziler ve List'in ötesinde, C# zengin bir koleksiyon ailesi sunar. En çok kullanılanı, anahtar–değer çiftlerini tutan Dictionary'dir. Hepsi generics ile tip güvenlidir: ne tür anahtar ve değer tutacaklarını baştan belirtirsin.

## List ve Dictionary

- `List<T>` — sıralı, dinamik liste.
- `Dictionary<K, V>` — anahtar–değer (PHP/Python sözlüğü gibi).
- Generics `<...>` tip güvenliğini sağlar.




Şema 16.1 — C# koleksiyonları generics ile tip güvenlidir.

**Dictionary kullanımı**

```
var stok = new Dictionary<string, int>();
stok["kalem"] = 100;
stok["defter"] = 50;

Console.WriteLine(stok["kalem"]);   // 100

foreach (var cift in stok) {
    Console.WriteLine($"{cift.Key}: {cift.Value}");
}
```

**İPUCU**

`Dictionary<string, int>`, "anahtarları string, değerleri int olan bir sözlük" demektir — PHP'nin ilişkisel dizilerinin ve Python'ın sözlüklerinin tip-güvenli karşılığıdır. Bir anahtara hızlıca erişmek (örn. ürün koduyla stok bulmak) için listeyi baştan sona aramaktan çok daha verimlidir. Var olmayan bir anahtara erişmek hata verir; `stok.TryGetValue(...)` veya `stok.ContainsKey(...)` ile güvenle kontrol edersin. Generics ( `<...>` ) sayesinde yanlış tipte anahtar/değer eklemeye çalışırsan derleyici engeller — koleksiyonların içeriği her zaman beklediğin tiptedir.

**Konsola ne yazar?** ÇIKTI

`Dictionary<string, int>` ürün adını stok sayısına eşler; `stok["kalem"]` ile 100 değerine doğrudan erişirsin. `foreach` ile her çiftin `.Key` (anahtar) ve `.Value` (değer) bilgisini dolaşıp yazarsın. Dictionary, anlamlı anahtarlarla hızlı erişim gereken her yerde idealdir.

**Alıştırma** 12 dk

Koleksiyon kullan:

1. Bir Dictionary<string, int> oluştur ve üç çift ekle.
2. Bir anahtarın değerini yazdır.
3. foreach ile tüm anahtar–değer çiftlerini yazdır.

BÖLÜM 17

# LINQ ile Veri Sorgulama

LINQ (Language Integrated Query), C#'ın en sevilen ve güçlü özelliklerinden biridir: koleksiyonları, neredeyse SQL gibi, doğrudan kod içinde sorgulamanı sağlar. Süzme, sıralama, dönüştürme — hepsi okunur tek satırlarla.

## Where, Select, OrderBy

- `.Where(x => koşul)` — süz (filtrele).
- `.Select(x => ifade)` — dönüştür.
- `.OrderBy(x => ...)` — sırala.


Şema 17.1 — LINQ: koleksiyonu kod içinde, okunur biçimde sorgular.

**LINQ sorgusu**

```
int[] sayilar = { 5, 30, 8, 25, 12 };

var buyukler = sayilar
    .Where(x => x > 10)      // süz
    .OrderBy(x => x)          // sırala
    .ToList();

foreach (var s in buyukler)
    Console.WriteLine(s);     // 12, 25, 30
```

**İPUCU**

LINQ, Modül 8'deki SQL'in mantığını (**süz, sırala, seç**) C# koleksiyonlarına taşır — ama veritabanı yerine bellekteki listeler üzerinde çalışır. `x => x > 10` yazımı bir **lambda**'dır: "bir x al, x'in 10'dan büyük olup olmadığını döndür" demenin kısa yoludur. LINQ ifadeleri zincirlenebilir ( `.Where(...).OrderBy(...).Select(...)` ) ve her adım okunur kalır. `Select` ile dönüştürür (örn. her kişiden yalnızca adını al), `Sum` / `Count` / `Average` ile özetlersin. LINQ, döngülerle yazılacak çok şeyi kısa ve net hâle getirir — ama karmaşıklaşınca okunabilirliği koru.

**Konsola ne yazar?** ÇIKTI

`sayilar.Where(x => x > 10)` yalnızca 10'dan büyük sayıları seçer; `.OrderBy(x => x)` onları küçükten büyüğe sıralar. Sonuç 12, 25, 30 olur ve foreach ile yazdırılır. LINQ, "şu koşula uyanları, şöyle sıralı getir" demenin kod içindeki en temiz yoludur.

**Alıştırma** 14 dk

LINQ yaz:

1. Bir sayı dizisinden çiftleri Where ile süz.
2. Sonucu OrderBy ile sırala ve yazdır.
3. LINQ ile SQL arasındaki benzerliği kendi cümlenle açıkla.

BÖLÜM 18

# Hata Yönetimi

Statik tipleme birçok hatayı önler ama hepsini değil: dosya bulunamaz, ağ kesilir, kullanıcı beklenmedik veri girer. Bu çalışma-anı hatalarını C#'ta try/catch/finally ile yakalar, programın çökmesini önlersin.

## try / catch / finally

- `try { ... }` — riskli kodu dene.
- `catch (Exception e) { ... }` — hatayı yakala.
- `finally { ... }` — her hâlükârda çalışır (temizlik).


Şema 18.1 — try/catch/finally: hatayı yakala, çökmeyi önle, temizle.

**Hata yakalama**

```
try {
    int n = int.Parse(girdi);
    Console.WriteLine(100 / n);
} catch (FormatException) {
    Console.WriteLine("Geçerli sayı girin.");
} catch (DivideByZeroException) {
    Console.WriteLine("Sıfıra bölünemez.");
} finally {
    Console.WriteLine("İşlem tamam.");
}
```

**İPUCU**

**Belirli hata tiplerini** yakala ( `FormatException` , `DivideByZeroException` ) — her şeyi tek bir genel `catch (Exception)` ile yutmak, gerçek sorunu gizleyip hata ayıklamayı zorlaştırır. Birden çok `catch` bloğu farklı hatalara farklı yanıt verir. `finally` bloğu, hata olsa da olmasa da **her zaman** çalışır — dosya kapatmak, bağlantı serbest bırakmak gibi temizlik işleri için idealdir (ya da `using` deyimini kullan, bir sonraki bölümde göreceksin). Backend ve PHP modüllerindeki ilkeyle aynı: hata ayrıntısını kullanıcıya gösterme, logla ve nazik bir mesaj ver.

**Konsola ne yazar?** ÇIKTI

Kullanıcı "abc" girerse `int.Parse` bir `FormatException` fırlatır; ilgili `catch` "Geçerli sayı girin." yazar. Sıfır girilirse bölme hatası yakalanır. Ne olursa olsun `finally` bloğu "İşlem tamam." yazarak çalışır. Program hiçbir durumda çökmez; her senaryoya uygun bir yanıt verir.

**Alıştırma** 12 dk

Hata yönet:

1. Bir dönüşümü ve bölmeyi try/catch ile saran kod yaz.
2. İki farklı hata tipini ayrı catch bloklarıyla ele al.
3. finally bloğunun ne işe yaradığını açıkla.

BÖLÜM 19

# Dosyalar ve Veri

Programlar çoğu zaman kalıcı veriyle çalışır. C#, System.IO ad alanıyla dosya okuma/yazmayı sade hâle getirir. Küçük işler için tek satırlık yardımcılar, büyük dosyalar için akış (stream) tabanlı yöntemler vardır.

## Dosya okuma ve yazma

- `File.WriteAllText(yol, metin)` — metni dosyaya yaz.
- `File.ReadAllText(yol)` — dosyayı oku.
- `File.ReadAllLines(yol)` — satır dizisi olarak oku.


Şema 19.1 — Dosya: yaz → diskte sakla → geri oku → kullan.

**Dosya okuma/yazma**

```
using System.IO;

// Yaz
File.WriteAllText("notlar.txt", "İlk notum");

// Oku
string icerik = File.ReadAllText("notlar.txt");
Console.WriteLine(icerik);    // İlk notum

// Satır satır
string[] satirlar = File.ReadAllLines("notlar.txt");
```

**İPUCU**

Küçük dosyalar için `File.WriteAllText` / `ReadAllText` en pratik yoldur — tek satırda işi bitirir. Büyük dosyalarda (tümünü belleğe almak istemediğinde) akış tabanlı yöntemler ve `using` deyimi kullanılır: `using var sr = new StreamReader(...)`, blok bitince dosyayı **otomatik kapatır** (Python'ın `with`'i gibi). Dosya işlemleri sık hata kaynağıdır (dosya yok, izin yok) — bu yüzden bir önceki bölümdeki `try/catch` ile sarmak iyi olur. Türkçe karakterler için kodlamayı (UTF-8) belirtmek güvenlidir.

**Konsola ne yazar?** ÇIKTI

`File.WriteAllText("notlar.txt", "İlk notum")` metni dosyaya yazar (yoksa oluşturur, varsa üzerine yazar); `File.ReadAllText(...)` aynı dosyayı geri okur ve konsola "İlk notum" yazdırılır. `ReadAllLines` ise dosyayı bir satır dizisine çevirir — her satıra ayrı erişebilirsin.

**Alıştırma** 12 dk

Dosya kullan:

1. Bir dosyaya metin yazan ve geri okuyan kod yaz.
2. ReadAllText ile ReadAllLines arasındaki farkı açıkla.
3. Dosya işlemlerini neden try/catch ile sarmak iyi olur, yaz.

BÖLÜM 20

# Asenkron Programlama (async/await)

Bir program, yavaş bir işi (dosya okuma, ağ isteği, veritabanı sorgusu) beklerken donmamalıdır. Asenkron programlama, bu bekleme sırasında programın başka işler yapabilmesini sağlar. C#'ta bunun aracı async ve await'tir.

## async ve await

- `async` — metodun asenkron olduğunu belirtir.
- `await` — yavaş işin sonucunu, bloklamadan bekler.
- Bekleme sırasında program tepkisiz kalmaz.


Şema 20.1 — async/await: yavaş işi beklerken program tepkisiz kalmaz.

**async / await**

```
async Task<string> VeriGetir() {
    // ağ isteği — yavaş olabilir
    string sonuc = await httpClient
        .GetStringAsync("https://api.ornek.com");
    return sonuc;
}

string veri = await VeriGetir();
Console.WriteLine(veri);
```

**İPUCU**

Asenkron programlamayı bir restorandaki garson gibi düşün: bir masanın yemeğini beklerken (mutfakta hazırlanıyor) öylece durmaz, diğer masalara bakar. `await`, "bu yavaş işin sonucunu bekle ama bu sırada programı kilitleme" der. Bu, özellikle web sunucularında (aynı anda binlerce isteği karşılarken) ve arayüzlerde (kullanıcı beklerken donmaması için) kritiktir. `async` metotlar genelde `Task` veya `Task<T>` döndürür. Modern C# ve .NET'te ağ, dosya ve veritabanı işlemlerinin çoğu asenkron sürümleriyle (`...Async`) kullanılır. Başlangıçta soyut gelebilir; bol pratikle yerleşir.

**Konsola ne yazar?** ÇIKTI

`await httpClient.GetStringAsync(...)`, bir ağ isteğini başlatır ve sonucu beklerken programın bloklanmasını önler — bu sürede başka işler yürüyebilir. İstek tamamlanınca kod kaldığı yerden devam eder, dönen veri `veri`'ye atanır ve yazdırılır. Kullanıcı veya sunucu, bekleme boyunca tepkisiz kalmaz.

**Alıştırma** 12 dk

Asenkronu kavra:

1. async/await'in çözdüğü problemi kendi cümlenle açıkla.
2. Hangi tür işler (örnekler) asenkron yapılmaya uygundur, yaz.
3. await'in "beklerken bloklamaması"nın neden önemli olduğunu belirt.

SEVİYE 4

# C# ile Üretim

Gerçek uygulamalar: .NET ekosistemi ve NuGet, ASP.NET ile web, veritabanı, güvenlik, yayınlama ve küçük bir C# uygulaması.

BÖLÜM 21

# .NET Ekosistemi ve NuGet

C# tek başına değil, .NET platformuyla birlikte gelir: zengin standart kütüphane, proje yapısı ve NuGet paket yöneticisi. Bu ekosistem, C#'ı web'den buluta, masaüstünden oyuna geniş bir alanda güçlü kılar.

## Proje yapısı ve paketler

- **.csproj:** proje dosyası (ayarlar, bağımlılıklar).
- **NuGet:** .NET'in paket yöneticisi (pip/Composer gibi).
- `dotnet add package ...` ile paket eklenir.


Şema 21.1 — Tipik bir .NET projesi: kod, proje dosyası ve derleme çıktısı.

**İPUCU**

**NuGet**, .NET dünyasının paket yöneticisidir (Python'daki pip, PHP'deki Composer gibi): `dotnet add package Newtonsoft.Json` gibi bir komutla hazır kütüphaneleri projene eklersin ve bağımlılıklar `.csproj` dosyasına kaydedilir. `bin/` ve `obj/` klasörleri derleme çıktısıdır ve **Git'e konmaz** (başka makinede `dotnet build` ile yeniden üretilir) — diğer dillerdeki `vendor/` ve `venv/` mantığının aynısı. .NET, tek bir komut satırı aracıyla (`dotnet`) proje oluşturma, derleme, çalıştırma ve paket yönetimini birleştirir; bu tutarlılık, büyük projelerde işi kolaylaştırır.

**Konsola ne yazar?** ÇIKTI

Bir .NET projesinde `Program.cs` giriş noktasıdır, `.csproj` projenin ayar ve bağımlılıklarını tutar, `bin/` ve `obj/` ise derleme sırasında üretilir. NuGet ile eklediğin paketler `.csproj` 'ye işlenir; böylece proje başka bir yerde tek komutla aynı bağımlılıklarla kurulur. Düzen, projeyi büyümeye ve ekip çalışmasına hazır kılar.

### Alıştırma

10 dk

Ekosistemi tanı:

1. Bir .NET projesinin temel dosyalarını ve görevlerini yaz.
2. NuGet'in ne işe yaradığını kendi cümlenle açıkla.
3. bin/ ve obj/ klasörlerinin neden Git'e konmadığını belirt.

BÖLÜM 22

# C# ile Web: ASP.NET'e Bakış

C#, web uygulamaları ve API'ler geliştirmek için ASP.NET çatısını kullanır. Backend ve API modüllerinde öğrendiğin kavramlar (istek, yanıt, rota, JSON) burada C#'ın tip güvenli dünyasında hayat bulur.

## İstekten yanıta

- Tarayıcı/istemci bir **istek** gönderir (Modül 7).
- ASP.NET, isteği doğru **rotaya** yönlendirir.
- Kod, veriyi işler ve bir **yanıt** (HTML/JSON) döner.


Şema 22.1 — ASP.NET: istek → rota → işle → yanıt (Modül 7'nin C# hâli).

**Basit bir API uç noktası (minimal API)**

```
var app = WebApplication.Create();

app.MapGet("/urunler", () => {
    var urunler = new[] { "Kalem", "Defter" };
    return Results.Json(urunler);   // JSON döner
});

app.Run();
```

**İPUCU**

ASP.NET, Modül 7'deki (Backend) ve Modül 9'daki (PHP web) kavramların C#'taki olgun karşılığıdır: rotalar, istek/yanıt, durum kodları, JSON. Modern "minimal API" yaklaşımı, küçük servisleri çok az kodla yazmanı sağlar; büyük uygulamalar için MVC (Model-View-Controller) deseni vardır. C#'ın tip güvenliği web'de de avantaj sağlar: gelen/giden veriyi sınıflarla (modellerle) temsil edersin ve derleyici birçok hatayı yakalar. Güvenlik ilkeleri burada da değişmez — bir sonraki bölümlerde göreceğin gibi girdiyi doğrula, parametreli sorgu kullan, sırları gizle.

**Konsola ne yazar?** ÇIKTI

`app.MapGet("/urunler", ...)`, "/urunler adresine gelen GET isteğini şu kod karşılasın" der; kod bir ürün listesini `Results.Json(...)` ile JSON olarak döner. Bir tarayıcı veya başka bir uygulama bu adrese istek atınca, ASP.NET isteği bu koda yönlendirir ve dönen JSON'u istemciye iletir — tam da Modül 7'deki API mantığı.

**Alıştırma** 10 dk

Web'i kavra:

1. ASP.NET'te bir isteğin yanıta dönüşme akışını adım adım yaz.
2. Minimal API ile bir uç noktanın ne yaptığını açıkla.
3. C#'ın tip güvenliğinin web'de sağladığı bir avantajı belirt.

BÖLÜM 23

# Veritabanı ve C#

C# uygulamaları veritabanıyla konuşur: veri okur, yazar, sorgular. Modül 8'deki SQL bilgisi burada doğrudan işe yarar. Ve oradaki en kritik güvenlik kuralı C#'ta da aynen geçerlidir: kullanıcı verisini asla doğrudan sorguya gömme — parametreli sorgu kullan.

## Güvenli sorgu


Şema 23.1 — Parametreli sorgu: kullanıcı verisini koddan ayırır (Modül 8).

**Parametreli sorgu**

```
// Parametre (@ad) ile — güvenli
var cmd = new SqlCommand(
    "SELECT * FROM Kisiler WHERE Ad = @ad", baglanti);
cmd.Parameters.AddWithValue("@ad", girdi);

var okuyucu = cmd.ExecuteReader();
// ... sonuçları oku
```

**İPUCU**

Bu, tüm seri boyunca tekrarlanan **en kritik güvenlik kuralıdır** ve C#'ta da değişmez: kullanıcıdan gelen değeri içeren her sorgu, **parametreli** yazılmalıdır ( `@ad` gibi yer tutucular + ayrı verilen değerler). Böylece kullanıcı girdiye SQL yazsa bile o yalnızca aranan bir metin olur, sorgunun yapısını değiştiremez — Modül 8'deki SQL enjeksiyonu ve Modül 9'daki hazırlanmış ifadelerle **aynı** ilke. Pratikte çoğu C# projesi, veritabanıyla daha yüksek seviyede konuşmak için bir ORM (örneğin Entity Framework) kullanır; o da arka planda parametreli sorgular üretir. Hangi yöntemi kullanırsan kullan: girdiyi asla ham olarak sorguya katma.

**Konsola ne yazar?** ÇIKTI

`WHERE Ad = @ad` sorgunun iskeletini tanımlar; `cmd.Parameters.AddWithValue("@ad", girdi)` ise kullanıcı verisini ayrı olarak yerleştirir. Kullanıcı ada zararlı bir SQL parçası yazsa bile, o değer yalnızca aranan bir metin olarak ele alınır — sorgunun mantığını bozamaz. Sonuç hem doğru hem güvenlidir.

**Alıştırma** 12 dk

Güvenli sorgula:

1. Kullanıcı verisini birleştiren tehlikeli bir sorguyu parametreli hâle getir.
2. Parametrenin (@ad) enjeksiyonu nasıl önlediğini açıkla.
3. Bu kuralın Modül 8 ve 9 ile bağını kendi cümlenle yaz.

BÖLÜM 24

# Güvenlik İlkeleri

Statik tipleme ve derleyici, birçok hatayı önler — ama güvenlik açıklarını değil. C# uygulamalarında da, tüm seri boyunca öğrendiğin temel güvenlik ilkeleri geçerlidir. Güvenlik, dilin değil, geliştiricinin sorumluluğudur.

## Olmazsa olmaz önlemler

- **Girdiyi doğrula:** kullanıcı/dış veriye güvenme.
- **Parametreli sorgu:** SQL enjeksiyonunu önle (Bölüm 23).
- **Sırları gizle:** bağlantı bilgisi/anahtar koda değil, yapılandırmaya.
- **Şifreleri hash'le:** asla düz metin sakla.


Şema 24.1 — C# güvenliğinin dört temel taşı (seri geneliyle aynı).

**İPUCU**

Bu ilkeler dilden bağımsızdır; C#'ta da aynen geçerlidir: kullanıcı/dış veriyi **doğrula** (Modül 9), veritabanı sorgularında **parametre** kullan (Modül 8), bağlantı bilgisi ve API anahtarı gibi sırları **koda gömme** — .NET'in yapılandırma sistemini (appsettings + ortam değişkenleri) kullan (Modül 7). Şifreleri güçlü bir algoritmayla **hash'le**, düz metin saklama. ASP.NET, kimlik doğrulama ve yetkilendirme için olgun, hazır bileşenler sunar — bunları kendin sıfırdan yazmak yerine kullan. Statik tipleme sana güvenlik vermez; o yalnızca tip hatalarını önler. Güvenlik, bilinçli bir alışkanlıktır.

**Konsola ne yazar?** ÇIKTI

Güvenli bir C# uygulaması: dış veriyi doğrular, veritabanı sorgularını parametreli yazar, sırları koddan ayrı (yapılandırmada) tutar ve şifreleri hash'ler. Bu dört önlem, gerçek dünya saldırılarının büyük çoğunluğunu durdurur ve tüm seri boyunca tekrarlanan ortak ilkelerdir — dil değişse de güvenlik kuralları değişmez.

**Alıştırma** 12 dk

Güvenliği uygula:

1. Dört temel güvenlik önlemini ve neye karşı koruduğunu yaz.
2. Sırların C#'ta nasıl saklanması gerektiğini açıkla.
3. "Statik tipleme güvenlik sağlamaz" ifadesini kendi cümlenle yorumla.

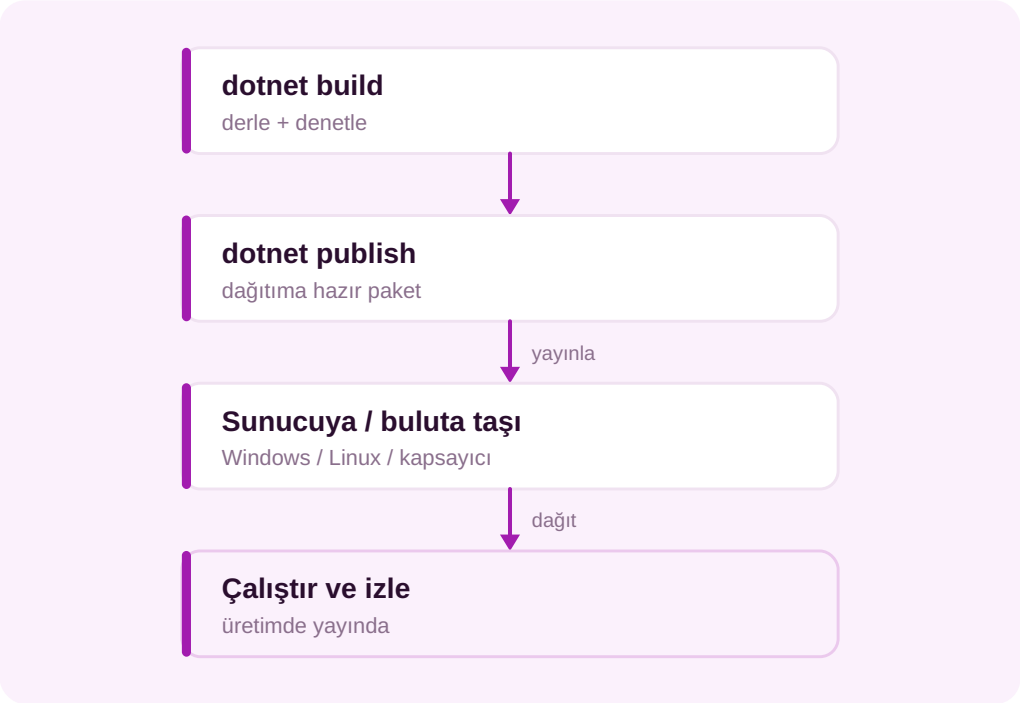
BÖLÜM 25

# Yayınlama ve Dağıtım

Uygulaman hazır olduğunda, onu çalıştırılabilir bir hâle getirip dağıtırsın. .NET, tek bir komut satırı aracıyla derleme, yayınlama ve dağıtımı kolaylaştırır; uygulaman birçok platformda (Windows, Linux, macOS) çalışabilir.

## Derle, yayınla, dağıt

- `dotnet build` — projeyi derle.
- `dotnet publish` — dağıtıma hazır çıktı üret.
- Çıktıyı sunucuya/buluta taşı ve çalıştır.




Şema 25.1 — Üretim yolu: derle → yayınla → dağıt → izle.

**İPUCU**

`dotnet publish`, uygulamanı çalıştırılabilir bir pakete dönüştürür; .NET'in çapraz platform desteği sayesinde aynı kod Windows, Linux ve macOS'ta çalışır. Modern dağıtımda uygulamalar genelde **kapsayıcılarda** (Docker) paketlenip buluta taşınır — bu, "benim makinemde çalışıyordu" sorununu ortadan kaldırır. Üretime almadan önce kontrol listesi diğer modüllerdekiyle aynıdır: sırlar koda gömülü değil, HTTPS aktif, hata ayrıntısı kullanıcıya gösterilmiyor (loglanıyor), bağımlılıklar güncel, veritabanı yedekleniyor (Modül 8). Yayınlama, "çalışıyor" ile "güvenle üretimde" arasındaki köprüdür.

**Konsola ne yazar?** ÇIKTI

`dotnet build` projeyi derleyip hataları yakalar; `dotnet publish` ise dağıtıma hazır, çalıştırılabilir bir paket üretir. Bu paketi bir sunucuya veya bulut ortamına taşıyıp çalıştırırsın — .NET'in çapraz platform desteğiyle çoğu işletim sisteminde sorunsuz koşar. Son adım, üretimde uygulamayı izlemek ve sağlıklı tutmaktır.

**Alıştırma** 10 dk

Dağıtıma hazırlan:

1. build, publish ve deploy adımlarının ne yaptığını yaz.
2. Üretime almadan önce kontrol edeceğin dört maddeyi listele.
3. Kapsayıcıların (Docker) sağladığı bir faydayı belirt.

BÖLÜM 26

# Bitirme: Küçük Bir C# Uygulaması

Öğrendiklerini birleştirip baştan sona küçük bir C# uygulaması tasarlıyorsun: tipli veriyi al, sınıflarla modelle, işle, sakla ve sun. Bu, gerçek bir C# uygulamasının çekirdeğidir ve seriyi diller açısından tamamlar.

## Bir "görev yöneticisi" akışı


Şema 26.1 — Küçük uygulama: al → modelle → işle → sakla → sun.

**Uygulama kontrol listesi**

```
// 1. Girdiyi al ve DOĞRULA (TryParse)
// 2. Veriyi SINIFLARLA modelle (class Gorev)
// 3. List<Gorev> + LINQ ile süz/sırala
// 4. try/catch ile hataları ele al
// 5. Dosya veya parametreli sorgu ile SAKLA
// 6. Sırlar yapılandırmada + okunur kod
```

**İPUCU**

Gerçek bir C# uygulaması yazarken bu sırayı izle: girdiyi **doğrula** (TryParse), veriyi **sınıflarla modelle**, koleksiyonlar ve **LINQ** ile işle, hataları **try/catch** ile yakala, sonucu dosya veya **parametreli sorgu** ile sakla, sırları **yapılandırmada** tut. Bu modülle birlikte üç dili (PHP, Python, C#) tamamladın ve önemli bir şeyi gördün: **kavramlar ortaktır, sözdizimi değişir**. Değişkenler, koşullar, döngüler, fonksiyonlar/metotlar, OOP, güvenlik — hepsi her dilde var. C# sana tip güvenliğini ve yapıyı; Python sadeliği; PHP web'in pratikliğini öğretti. Artık yeni bir dile geçmek çok daha kolay; çünkü öğrenilen şey dil değil, programlama düşüncesidir.

**Konsola ne yazar?** ÇIKTI

Tasarladığın görev yöneticisi: kullanıcıdan görevleri alır (doğrulayarak), her görevi bir `Gorev` sınıfıyla modeller, `List<Gorev>` içinde tutup LINQ ile süzer/sıralar, hataları yakalar ve sonucu dosyaya veya veritabanına güvenle saklar. Tipli modeller, koleksiyonlar, LINQ ve güvenlik bir araya gelince — çalışan, sağlam, gerçek bir C# uygulaması ortaya çıkar.

**Alıştırma** 20 dk

Uygulamanı tasarla:

1. Küçük bir uygulama seç (görev listesi, kişi rehberi, basit envanter).
2. Veriyi hangi sınıf(lar)la modelleyeceğini yaz.
3. Al → modelle → işle (LINQ) → sakla akışını fonksiyonlara böl.
4. Hangi güvenlik ve hata yönetimi önlemlerini ekleyeceğini belirt.

EK

# C# Terimleri ve Komutları Sözlüğü

En sık kullanılan C# yapıları ve komutları. Bir başvuru kaynağı olarak saklayabilirsin.

| | | | |
|---|---|---|---|
| Console.WriteLine | `Konsola yazdır` | int / string / bool | `Temel tipler` |
| var | `Tip çıkarımı` | $"...{x}..." | `String interpolation` |
| if / switch | `Koşul` | for / foreach | `Döngü` |
| int[] / List<T> | `Dizi / liste` | void Ad(...) | `Metot` |
| class | `Sınıf (OOP)` | { get; set; } | `Özellik (property)` |
| interface | `Arayüz` | LINQ (Where/Select) | `Veri sorgulama` |
| try / catch | `Hata yönetimi` | async / await | `Asenkron` |

**C#'ın özeti** ÇIKTI

C# derlenen, statik tipli ve güçlü nesne yönelimli bir dildir. **Console.WriteLine** ile yazdırır, **tipli değişkenlerle** (int, string...) veri taşır, **if/switch** ve **döngülerle** akışı yönetir. **Diziler** ve **List<T>** koleksiyonları tip güvenlidir; **metotlar**, **sınıflar**, **özellikler**, **arayüzler** ve **kalıtım** OOP'nin temelidir. **LINQ** ile veriyi sorgular, **async/await** ile asenkron çalışır, **.NET** platformunun zengin kütüphanesinden yararlanırsın. Derleyici ve statik tipler birçok hatayı çalışmadan önce yakalar — ama güvenlik yine senin sorumluluğun: girdiyi doğrula, parametreli sorgu kullan, sırları gizle.