# 1.Ünite - Nesneye Yönelik Düşünme

---

**Nesneye Yönelik Düşünme (NYD)**, yazılım sistemlerini tıpkı gerçek dünyadaki gibi **nesneler** (özellikleri ve davranışları olan varlıklar) üzerinden modelleme yaklaşımıdır. Bu yöntem, karmaşık sistemleri anlamayı, tasarlamayı ve sürdürmeyi kolaylaştırır.

---

## Nesneye Yönelik Düşünme Yaklaşımı

- Her bir **bileşen** nesne olarak ele alınır.
- Gerçek dünyadaki nesneler (insan, masa, bilgisayar vb.) gibi yazılım nesneleri de **özelliklere (nitelikler)** ve **davranışlara (metotlar)** sahiptir.
- Nesneler arasındaki **ilişkiler ve etkileşimler**, sistemin bütününü oluşturur.
- Bu yaklaşım, yazılımların daha **modüler, esnek ve yeniden kullanılabilir** olmasını sağlar.

---

## Problem Çözme Adımları

1. **Problemin tanımlanması:** Sorunun açık biçimde ifade edilmesi.
2. **Soyutlama:** Gereksiz detayları ayırarak sadece önemli kısımlara odaklanmak.
3. **Model oluşturma:** Problemi temsil eden kavramsal bir yapı kurmak.
4. **Çözüm geliştirme:** Model üzerinden çözüm üretilmesi.
5. **Doğrulama ve geçerleme:**
    - **Doğrulama (Verification):** Modelin doğru uygulanıp uygulanmadığının kontrolü.
    - **Geçerleme (Validation):** Modelin gerçek sistemi doğru temsil edip etmediğinin sınanması.

---

## Programlama Paradigmaları

Programlama dilleri, **problemi çözme biçimine göre** farklı paradigmalara ayrılır:

- **Emirli (Imperative):** Ne yapılacağını değil, *nasıl yapılacağını* adım adım belirtir.
  Örnek: C, C++, Java
- **Bildirimli (Declarative):** *Ne yapılacağını* söyler, *nasıl yapılacağı* arka planda gerçekleşir.
  Örnek: SQL, HTML, Regex
- **Hibrit (Hybrid):** Her iki yaklaşımı da destekler.
  Örnek: Python, C#, JavaScript
- Emirli paradigmalar kendi içinde:
    - **Yordamsal (Procedural):** Adım adım işleyen yapı (C, Pascal).
    - **Nesneye Yönelik (OOP):** Nesneler üzerinden modelleme (C++, Java, C#).
    - **Paralel Programlama:** Birden fazla işlemi eşzamanlı yürütür.

---

## Yazılım ve Kalite Kriterleri

Yazılım; bilgisayarlara ne yapacağını söyleyen komut dizileridir.
Kaliteli yazılım hem kullanıcı hem geliştirici açısından belirli özellikleri taşımalıdır.

**Kullanıcı açısından:**

- Doğru sonuç verir.
- Kolay öğrenilir ve kullanılır.

- Hızlı ve hatasız çalışır.
- Kaynakları verimli kullanır.
- Verileri güvende tutar.
- Güncellenebilir ve belgelenmiştir.

**Geliştirici açısından:**

- Kod okunabilir ve düzenlidir.
- Modüler yapıdadır, bakımı kolaydır.
- Yeniden kullanılabilir modüller içerir.
- Maliyet ve zaman açısından verimlidir.
- Belgelerle desteklenmiştir.

---

### Yazılım Geliştirme Süreci

1. **Analiz:** Gereksinimlerin belirlenmesi.
2. **Tasarım:** Modelin ve mimarinin oluşturulması.
3. **Kodlama:** Tasarımın programlama diliyle hayata geçirilmesi.
4. **Dokümantasyon:** Sürecin kayıt altına alınması.
5. **Test:** Hataların tespiti ve düzeltilmesi.
6. **Dağıtım/Bakım:** Yazılımın kullanıma sunulması ve geliştirilmesi.

---

### Sonuç

Nesneye yönelik düşünme, gerçek dünyayı yazılıma taşımanın en doğal yoludur.
Bu yaklaşım:

- Yazılımları **modüler**, **anlaşılır** ve **sürdürülebilir** hale getirir.
- **Kod tekrarını azaltır**, **yeniden kullanılabilirliği artırır.**
- Yazılım kalitesini yükseltir ve geliştirme sürecini hızlandırır.

# 2.Ünite - Nesneye Yönelik Programlama

---

Nesneye yönelik programlama (OOP), yazılımları gerçek dünyadaki nesnelere benzer yapılarla modellemeyi hedefler. Bu yaklaşım, yazılımların daha modüler, esnek ve sürdürülebilir olmasını sağlar. Temel yapı taşları:

- **Sınıf (Class)**
- **Nesne (Object)**
- **Soyutlama (Abstraction)**
- **Kapsülleme (Encapsulation)**
- **Kalıtım (Inheritance)**
- **Çok biçimlilik (Polymorphism)**

**OOP'nin Avantajları**

- **Modülerlik:** Kod parçalarının bağımsız geliştirilebilmesi
- **Sürdürülebilirlik:** Uzun vadeli projelerde kolay bakım

- **Verimlilik:** Kod tekrarının azalması, yeniden kullanım
- **Güvenlik:** Kapsülleme sayesinde veri gizliliği
- **Ekip Çalışması:** Farklı geliştiriciler bağımsız modüllerde çalışabilir
- **Anlaşılabilirlik:** Gerçek dünya nesneleriyle benzerlik

**Sınıf ve Nesne**

- **Sınıf:** Nesnelerin özelliklerini ve davranışlarını tanımlayan şablondur.
  Örnek: Araba sınıfı → marka, model, renk özellikleri + sür, fren yap davranışları
- **Nesne:** Sınıfın bir örneğidir (instance). Sınıfta tanımlı özellik ve davranışlara sahiptir.

**Soyutlama (Abstraction)**

Bir nesnenin yalnızca gerekli özellik ve davranışlarının gösterilmesi, gereksiz ayrıntıların gizlenmesidir.
Örnek: **ATM cihazı** – kullanıcı yalnızca para çekme veya yatırma işlemini görür, arka plandaki karmaşık süreç gizlenir.

**Kapsülleme (Encapsulation)**

Veri ve bu veriye erişimi yöneten metotların tek bir yapı içinde toplanmasıdır.
Amaç:

- Veri gizliliği ve bütünlüğü sağlamak
- Dışarıdan doğrudan erişimi engellemek
- Kodun bakımını kolaylaştırmak

**Farkı:**

- *Soyutlama* karmaşıklığı gizler.
- *Kapsülleme* verilere erişimi sınırlar.

**Kalıtım (Inheritance)**

Bir sınıfın, başka bir sınıftan özellik ve davranışları miras almasıdır.

- **Üst sınıf (base class):** Ortak özellik ve davranışları tanımlar.
- **Alt sınıf (derived class):** Üst sınıfı genişletir, kendi özelliklerini ekleyebilir.
  Örnek: Baba sınıfı → Oğul sınıfı üst sınıftan miras alır.
  **Çoklu Kalıtım:** Bir sınıf birden fazla üst sınıftan miras alabilir.

**Çok Biçimlilik (Polymorphism)**

Aynı metodun farklı nesnelerde farklı biçimlerde davranabilmesidir.
Örnek: “Görev yap” metodu;

- Aşçı için yemek pişirir
- Temizlikçi için temizlik yapar
- Bebek Bakıcısı için çocuk bakar

**Türleri:**

- **Statik Polimorfizm:** Derleme zamanında (metot/operatör aşırı yükleme)
- **Dinamik Polimorfizm:** Çalışma zamanında (metot geçersiz kılma – override)

**Mesaj Geçişi (Message Passing)**

Nesneler arasındaki iletişimi sağlar.
Bir nesne diğerine mesaj gönderir, hedef nesne mesajı alır ve uygun metodu çalıştırır.
Bu mekanizma nesnelerin bağımsızlığını artırır ve sistemin esnekliğini sağlar.

### Sonuç

Nesneye yönelik programlama, karmaşık yazılımları daha kolay yönetilebilir hale getirir.
Kodun yeniden kullanılabilirliği, güvenliği ve sürdürülebilirliği artar.
Modern dillerin çoğu (C++, Java, C#, Python) bu yaklaşımı temel alır.

# 3.Ünite - Tümleşik Modelleme Dili (UML)

---

**UML (Unified Modeling Language)**, yazılım sistemlerinin analiz, tasarım ve dokümantasyon süreçlerinde kullanılan**görsel bir modelleme dili**dir. Karmaşık sistemleri anlaşılır hale getirir ve yazılım geliştirme sürecinde iletişimi kolaylaştırır.

---

## UML'in Amacı

- Yazılım bileşenlerini ve ilişkilerini **görsel olarak ifade etmek**
- Karmaşık sistemleri **basitleştirmek ve analiz etmek**
- Geliştiriciler arasında **ortak bir dil** oluşturmak
- Kod yazmadan önce **tasarımı planlamak**

---

## UML Diyagram Türleri

#### 1. Yapısal Diyagramlar (Structural Diagrams)

Sistemin **statik yapısını** gösterir, sınıflar ve ilişkileri tanımlar.

- **Sınıf Diyagramı (Class Diagram):** Sınıflar, özellikler, metotlar ve ilişkiler.
- **Nesne Diyagramı (Object Diagram):** Çalışma anındaki nesne örnekleri.
- **Bileşen Diyagramı (Component Diagram):** Modüller arası bağlantılar.
- **Dağıtım Diyagramı (Deployment Diagram):** Donanım ve yazılım bileşenlerinin dağılımı.

#### 2. Davranış Diyagramları (Behavioral Diagrams)

Sistemin **dinamik yapısını**, olaylar ve süreçleri gösterir.

- **Kullanım Senaryosu Diyagramı (Use Case):** Kullanıcı ile sistem etkileşimi.
- **Etkinlik Diyagramı (Activity):** İş akışını veya süreci adım adım gösterir.
- **Durum Diyagramı (State Machine):** Nesnelerin olaylara göre durum değişimleri.
- **Dizi Diyagramı (Sequence):** Nesneler arası mesaj akışı.

---

## Sınıf Diyagramı (Class Diagram)

OOP'nin temel yapısını UML üzerinde temsil eder:

- **Sınıf (Class)** → Özellikler (attributes) + Davranışlar (operations)
- **İlişkiler:**
  - **Bağımlılık (Dependency)**
  - **Birliktelik (Association)**
  - **Kapsama (Aggregation)**
  - **Bütünleme (Composition)**
  - **Genelleme (Inheritance)**

**Örnek:**

```
+ Araba
  - marka: string
  - hız: int
  + hizlan(): void
  + dur(): void
```

---

### Kullanım Senaryosu Diyagramı (Use Case)

- Sistemi kullanıcı gözünden gösterir.
- **Aktör (Actor):** Sistemi kullanan kişi ya da sistem.
- **Use Case:** Kullanıcının gerçekleştirdiği işlev.
- **İlişkiler:**
    - **Include:** Zorunlu alt senaryo
    - **Extend:** Opsiyonel alt senaryo

**Örnek:**

```
Kullanıcı → (Giriş Yap)
        → (Ürün Ara)
        → (Sepete Ekle)
```

---

### Etkinlik Diyagramı (Activity Diagram)

- İş süreçlerini, karar noktalarını ve akış yönünü gösterir.
- **Başlangıç ve bitiş düğümleri**, **karar**, **birleşim** ve **faaliyet akışları** yer alır.

---

### Durum Diyagramı (State Diagram)

Bir nesnenin yaşam döngüsünü ve **durum değişimlerini** anlatır.
Örnek: Sipariş → "Alındı" → "Hazırlanıyor" → "Kargoda" → "Teslim Edildi"

---

### Dizi Diyagramı (Sequence Diagram)

- Nesneler arası **zaman sıralı etkileşimi** gösterir.
- Mesajların hangi sırayla gönderilip alındığı belirtilir.

---

### Sonuç

UML, yazılım geliştirmede soyut fikirleri görsel biçime dönüştüren **ortak bir modelleme dilidir.**
Sınıf, kullanım senaryosu, etkinlik ve dizi diyagramları, yazılım tasarımının en sık kullanılan araçlarıdır. UML sayesinde yazılım ekipleri sistemleri daha net anlayabilir, iletişim kurabilir ve hataları erken tespit edebilir.

# 4.Ünite - .NET Framework ve C

---

## 1. Giriş

C#, Microsoft tarafından geliştirilen, **nesne yönelimli, modern ve genel amaçlı** bir programlama dilidir.
2000 yılında Anders Hejlsberg liderliğinde oluşturulmuştur ve .NET Framework üzerinde çalışır.
Java'ya benzer sözdizimine sahiptir; masaüstü, web, mobil, oyun ve veritabanı uygulamaları geliştirmek için kullanılır.

**Temel özellikleri:**

- Güçlü tip güvenliği ve hata ayıklama yeteneği
- Zengin standart kütüphane (FCL)

- Geniş topluluk ve IDE desteği (özellikle Visual Studio)
- Kolay öğrenilebilir yapı

---

# 2. .NET Framework

Microsoft tarafından geliştirilen, Windows üzerinde çalışan bir **yazılım platformudur**.
Farklı dillerle uygulama geliştirme, çalıştırma ve dağıtımı destekler.

## 2.1. Temel Bileşenler

#### Common Language Runtime (CLR)

- .NET programlarını çalıştıran sanal makinedir.
- Bellek yönetimi, güvenlik, hata ayıklama gibi görevleri üstlenir.
- Farklı .NET dillerinde yazılmış kodların birlikte çalışmasını sağlar.

#### Framework Class Library (FCL)

- Yeniden kullanılabilir sınıf ve bileşenlerin koleksiyonudur.
- Veri tabanı, ağ, dosya, güvenlik, grafik işlemleri gibi alanlarda hazır sınıflar sunar.

#### Dil Entegrasyonu

- C#, VB.NET, F#, C++/CLI gibi birçok dil desteklenir.
- Derleme sonucu tüm diller **Common Intermediate Language (CIL)** koduna dönüştürülür.

#### Common Language Specification (CLS)

- Farklı dillerin uyum içinde çalışmasını sağlayan kurallar kümesidir.
- Veri türleri, erişim düzeyleri, isimlendirme ve istisna yönetimi gibi konularda standartlar belirler.

#### Common Type System (CTS)

- .NET dillerinde tür uyumluluğu sağlar.
- Farklı dillerde oluşturulan sınıflar birbiriyle etkileşebilir.
- Türlerin bellekte temsilini ve davranışını tanımlar.

#### Assembly

- Derlenmiş .exe veya .dll dosyalarıdır.
- Kod, türler ve metadata içerir.
- Türleri: Compile-Time, Runtime, Shared Assembly.

#### Application Domain

- Uygulamaların izole biçimde çalıştığı alanlardır.
- Bellek yönetimi, güvenlik ve hata izolasyonu sağlar.

#### Namespace (İsim Alanı)

- Kodun düzenlenmesini ve isim çakışmalarını önler.
- Örn: `System`, `System.IO`.
- Geliştiriciler özel namespace oluşturabilir.

---

### 2.2. .NET Framework Ozellikleri

- **IDE Desteği:** Visual Studio en yaygın geliştirme ortamıdır.
- **Çoklu Platform:** .NET Core ile Windows, macOS, Linux desteği.
- **Güvenlik:** Kod imzalama, yetkilendirme, izin kontrolleri.
- **Veritabanı Entegrasyonu:** ADO.NET ve LINQ desteği.
- **Kullanım Alanları:** Masaüstü, web, mobil, oyun, sunucu ve yapay zekâ uygulamaları.

---

## 3. C#'a Giriş

- Microsoft tarafından .NET platformu için geliştirilmiştir.
- C++ ve Java'dan etkilenmiştir.
- Basit, okunabilir, sürdürülebilir bir dil yapısına sahiptir.
- Mono ve .NET Core sayesinde Windows dışı sistemlerde de kullanılabilir.
- Nesneye Yönelik Programlama (OOP) prensiplerine dayanır:
    - Sınıflar, nesneler, kalıtım, çok biçimlilik, soyutlama, kapsülleme.

---

## 4. C# Dilinin Temel Özellikleri

- **Geniş Kütüphane:** Framework Class Library kullanımı.
- **Tip Güvenliği:** Statik tip kontrolü ile hataları önler.
- **Platform Bağımsızlık:** .NET Core sayesinde çoklu sistem desteği.
- **Hata Ayıklama:** Visual Studio ile güçlü debugging araçları.
- **Topluluk Desteği:** Geniş kaynak ve forum ekosistemi.

---

## 5. C#'ta Program Yazma Aşamaları

1. **Kaynak Kodun Oluşturulması**
    - `.cs` uzantılı dosyalar IDE veya metin editörü ile yazılır.
2. **Derleme**
    - C# derleyicisi hataları kontrol eder ve kodu derler.
3. **Ara Dile (CIL) Dönüştürme**
    - Kaynak kod, platformdan bağımsız **Common Intermediate Language**'e çevrilir.
4. **JIT (Just-In-Time) Derleme**
    - CIL kodu çalışma zamanında makine diline çevrilir.
    - **JIT türleri:**
        - *Normal JIT:* Gerektiğinde derler.
        - *Pre-JIT:* Tüm kodu baştan derler.
        - *Eco JIT:* Bellek sınırlı sistemler için optimize.
    - **Tiered Compilation (.NET Core):**
        - *Lower Tier:* Hızlı ama az optimize.

- *Upper Tier:* Daha optimize, yüksek performanslı.

5. **Yürütme**
    - CLR, bellek yönetimi, güvenlik ve hata ayıklama işlemlerini gerçekleştirir.
    - Makine kodu işletim sistemi üzerinde çalıştırılır.

---

## 6. Visual Studio ile İlk C# Örneği

```
using System;

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

**Önemli noktalar:**

- Her C# programı en az bir sınıf içerir.
- `Main()` metodu giriş noktasıdır.
- Kod satırları `;` ile biter.
- `using` ifadesi namespace eklemek için kullanılır.

---

### Sonuç

- **.NET Framework**, çok dilli destek, güvenlik, taşınabilirlik ve geniş kütüphaneler sunan bir platformdur.
- **C#**, bu platform üzerinde çalışan modern, güvenli ve nesne yönelimli bir dildir.
- Derleme süreci, kaynak koddan CIL'e ve JIT aracılığıyla makine koduna dönüşüm içerir.
- Visual Studio, C# geliştirme sürecini kolaylaştıran güçlü bir IDE'dir.

# 5.Ünite - C# Veri Türleri ve Değişkenler

---

## 1. Giriş

C#, verileri **saklamak ve işlemek** için farklı türlerde veri yapıları sunar.
Veri türleri iki ana grupta incelenir:

- **Önceden tanımlanmış türler**
    - *Değer tipleri (Value Types)*
    - *Referans tipleri (Reference Types)*
- **Kullanıcı tanımlı türler**
  (örneğin `class`, `struct`, `enum`, `interface`)

**Değer tipleri** doğrudan *stack* bellekte saklanır.
**Referans tipleri** ise *heap* bellekte tutulur ve stack sadece onların adresini taşır.

> Değer tipini `object` türüne dönüştürme işlemine **boxing**,
> tekrar orijinal haline çevirme işlemine **unboxing** denir.

Bir değişkenin erişim alanı **scope (kapsam)** olarak adlandırılır.
Değişmemesi gereken değerler `const` anahtar kelimesiyle **sabit (constant)** yapılır.

---

# 2. Temel Veri Türleri

## Değer Tipleri (Value Types)

Bellekte kendi değerini tutar (stack'te).
Örnekler:

- Tam sayılar → int, long, short, byte
- Ondalıklı sayılar → float, double, decimal
- Karakter → char
- Mantıksal → bool
- Tarih → DateTime

Her değer tipi, birbirinden bağımsız şekilde kopyalanabilir.

## Referans Tipleri (Reference Types)

Bellekteki adresi tutar (heap'te).
Örnekler:

- string, object, class, array, interface, delegate

Bir referans tipi başka bir değişkene atanırsa, **aynı nesneyi** işaret eder.

---

# 3. Bellek Bölgeleri

## Stack

- Yerel değişkenler ve metod parametreleri burada tutulur.
- LIFO (Last In, First Out) mantığıyla çalışır.
- Hızlıdır ve boyutu sınırlıdır.
- Değişkenin kapsamı bittiğinde bellek otomatik olarak temizlenir.

## Heap

- new anahtar sözcüğüyle oluşturulan nesneler burada saklanır.
- Dinamik bellek yönetimi sağlar.
- Garbage Collector tarafından temizlenir.
- Referans tipleri burada yaşar.

## Diğer Bellek Bölgeleri

- **Register:** CPU üzerindeki en hızlı geçici alan.
- **Static Bölge:** Program süresince var olan veriler.
- **Sabit Bölge:** Değişmeyen değerler.
- **RAM Dışı Bölge:** Disk veya harici depolama alanları.

---

# 4. Değişkenler (Variables)

Bir değişken, bellekte veri saklayan isimlendirilmiş alandır.

## 1. Tanımlama

```
int number;
```

### 2. Değer Atama

```
number = 10;
```

Değişken kullanılmadan önce mutlaka değer atanmalıdır.

### 3. Kullanım

```
Console.WriteLine(number); // 10
```

### İsimlendirme Kuralları

- Harf veya _ ile başlamalı, sayı ile başlayamaz.
- Boşluk ve özel karakter içeremez.
- Büyük/küçük harf duyarlıdır.
- Anlamlı isimler kullanılmalıdır.
- C# anahtar kelimeleri (`int`, `class`, `if` vb.) değişken adı olamaz.
- İsimlendirme stilleri:
  - camelCase → firstName
  - PascalCase → FirstName

---

# 5. Tür Dönüşümleri (Type Conversions)

Bir veri türünü başka bir türe çevirmeye **type conversion** denir.

### 1. Implicit (Kapalı / Otomatik)

- Veri kaybı riski yoksa C# otomatik dönüşüm yapar.
- Küçük tür → büyük türe

  csharp int i = 10; double d = i; // otomatik dönüşüm

### 2. Explicit (Açık / Manuel)

- Veri kaybı olasılığı varsa dönüşüm açıkça belirtilmelidir.

  csharp double d = 3.75; int i = (int)d; // cast

#### Convert Sınıfı

Dönüştürme işlemleri için yardımcı sınıftır:

```
int x = Convert.ToInt32("42");
string s = Convert.ToString(123);
```

#### Parse() Metodu

Metin verilerini sayısal değerlere dönüştürür:

```
int n = int.Parse("123");
```

#### TryParse() (Güvenli Alternatif)

Hatalı girişi yakalar:

```
int.TryParse("abc", out int result); // false döner
```

---

# 6. Boxing ve Unboxing

### Boxing (Kutulama)

Değer tipinin object türüne dönüştürülmesidir.

```
int value = 42;
object boxed = value;
```

### Unboxing (Kutudan Çıkarma)

object türündeki veriyi tekrar orijinal türe dönüştürür.

```
int number = (int)boxed;
```

**Not:** Bu işlemler performans maliyetlidir,
bu yüzden **generics (List<T>, Dictionary<TKey, TValue>)** kullanmak önerilir.

---

# 7. Değişkenlerin Faaliyet Alanı (Scope)

Bir değişkenin erişilebildiği kod bloğudur.

### Türleri:

- **Metot Scope:** Değişken sadece o metodun içinde geçerlidir.
- **Blok Scope:** {} ile çevrili yapılar (if, for, while vb.).
- **Sınıf Scope:** Tüm sınıf içinde erişilebilir.

Kapsam bittiğinde değişken bellekten silinir.

---

# 8. Sabitler (Constants)

Değeri değişmeyen, derleme zamanında belirlenen değişkenlerdir.
const anahtar kelimesiyle tanımlanır.

```
const double PI = 3.14;
const string APP_NAME = "MyApp";
```

- Program süresince değiştirilemez.
- Türleri: int, double, char, string, bool vb.
- Tanımlandığı kapsam içinde geçerlidir.

---

### Sonuç

- C#'ta veri türleri **değer** ve **referans** tiplerine ayrılır.
- Bellek yönetimi stack ve heap üzerinden yapılır.
- Tür dönüşümleri **implicit** ve **explicit** olarak gerçekleşir.
- **Boxing** ve **unboxing**, değer → referans ve tersine dönüşümleri tanımlar.
- **Scope**, değişkenin ömrünü ve erişim alanını belirler.
- **Const** değişkenler sabit değerleri temsil eder.

# C# Değişken Özeti Tablosu

| Kategori | Açıklama | Örnek Veri Tipleri | Saklandığı Bellek | Örnek Tanım | Özellikler |
|---|---|---|---|---|---|
| **Değer Tipleri (Value Types)** | Verinin kendisini tutar, doğrudan bellekte saklanır. | int, float, double, bool, char, decimal, DateTime | **Stack** | int age = 25; | Hızlı erişim sağlar, her kopya bağımsızdır. |

| Kategori | Açıklama | Örnek Veri Tipleri | Saklandığı Bellek | Örnek Tanım | Özellikler |
|---|---|---|---|---|---|
| **Referans Tipleri (Reference Types)** | Verinin adresini (referansını) tutar. | string, object, class, array, interface | **Heap (veri)**, **Stack (referans)** | string name = "Ahmet"; | Birden fazla değişken aynı nesneyi gösterebilir. |
| **Kullanıcı Tanımlı Tipler** | Geliştirici tarafından oluşturulan özel türler. | struct, enum, class, interface | Türüne göre (struct → Stack, class → Heap) | class Student { } | Modüler, yeniden kullanılabilir kod sağlar. |
| **Sabitler (Constants)** | Değeri derleme zamanında belirlenir ve değiştirilemez. | Tüm temel tipler (int, double, string, vb.) | **Static alan** | const double PI = 3.14; | Program süresince sabit kalır. |
| **Statik Değişkenler** | Tüm sınıf örnekleri arasında ortak değişkenlerdir. | Tüm tiplerde olabilir | **Static alan (RAM)** | static int counter = 0; | Sınıf yüklenirken bir kez oluşturulur. |
| **Yerel Değişkenler (Local)** | Sadece tanımlandığı blokta erişilebilir. | Herhangi bir tip | **Stack** | { int a = 10; } | Blok bitince bellekten silinir. |
| **Global / Alan Değişkenleri (Field)** | Sınıf seviyesinde tanımlanır. | Herhangi bir tip | Stack (değer) / Heap (referans) | private string name; | Tüm sınıf içinde erişilebilir. |
| **Readonly Değişkenler** | Sadece ilk atamada veya yapıcıda değiştirilebilir. | Tüm tipler | Stack / Heap | readonly int id = 5; | const gibi ama runtime'da atanabilir. |

---

## Ek Bilgiler

| Kavram | Açıklama | Örnek |
|---|---|---|
| **Boxing** | Değer tipinin object'e dönüştürülmesi | object obj = 42; |
| **Unboxing** | object tipinden orijinal değere dönüş | int x = (int)obj; |
| **Scope (Kapsam)** | Değişkenin erişim alanı | Metot, blok veya sınıf |
| **Type Conversion** | Tür dönüşümü (implicit / explicit) | double d = i; veya int i = (int)d; |

# 6.Ünite - Sınıflar ve Nesneler

---

Nesneye Yönelik Programlamanın (NYP) temeli olan sınıf/nesne yapısını, yaşam döngülerini (yapıcı/yıkıcı metotlar) ve veri güvenliği yöntemlerini (kapsülleme) pratik örneklerle açıklamaktadır.

## 1. Temel Kavramlar: Sınıf ve Nesne

Nesneye yönelik programlama, problemleri parçalara ayırıp her parçayı bir nesne olarak modelleme tekniğidir .

- **Sınıf (Class):** Bir nesnenin şablonudur. Nesnenin sahip olacağı özellikleri (alanlar) ve davranışları (metotlar) tanımlar . Soyut bir kavramdır.
- **Nesne (Instance):** Sınıftan türetilen somut örnektir. Bellekte yer kaplar ve gerçek dünya nesnelerini temsil eder .

### Uygulama Örneği: Araba Sınıfı

Bir sınıftan nesne türetmek için new anahtar sözcüğü kullanılır .

```
using System;

// Sınıf Tanımı (Şablon)
public class Araba
{
    // Alanlar (Özellikler)
    public string Marka;
    public string Model;
    public int Yil;
    public string Renk;

    // Metotlar (Davranışlar)
    public void Calistir()
    {
        Console.WriteLine("Araba çalışıyor...");
```

```
    }
}

// Kullanım (Main Metodu)
class Program
{
    static void Main()
    {
        // Nesne Türetme (Instance Oluşturma)
        Araba Araba1 = new Araba();

        // Özelliklere Değer Atama
        // (Not: Orijinal belgedeki yazım hataları düzeltilmiştir)
        Araba1.Marka = "BMW";
        Araba1.Model = "5 Serisi";
        Araba1.Yil = 2018;

        // Metodu Çağırma
        Araba1.Calistir();
    }
}
```

---

# 2. Nesne Yaşam Döngüsü: Yapıcı ve Yıkıcı Metotlar

Nesneler oluşturulurken ve yok edilirken otomatik çalışan özel metotlar vardır.

## A. Yapıcı Metotlar (Constructors)

- Nesne `new` ile oluşturulduğunda **otomatik** çalışır .
- Sınıfın başlangıç durumunu (ilk değerlerini) ayarlamak için kullanılır .
- Sınıf ismiyle aynı isme sahiptir ve geriye değer döndürmez .
- **Aşırı Yükleme (Overloading):** Farklı parametreler alarak birden fazla yapıcı metot tanımlanabilir .

## B. Yıkıcı Metotlar (Destructors)

- Nesne artık kullanılmayacağı zaman (bellekten atılmadan hemen önce) otomatik çalışır .
- Sınıf isminin başına `~` (tilde) işareti konularak tanımlanır .
- Parametre almazlar ve sadece bir tane olabilirler .

## Uygulama Örneği: Kedi Sınıfı

```
public class Kedi
{
    public string Isim;

    // Yapıcı Metot: Nesne oluştuğunda çalışır
    public Kedi(string isim)
    {
        this.Isim = isim;
        Console.WriteLine(isim + " adlı kedi oluşturuldu.");
    }

    // Yıkıcı Metot: Nesne silinirken çalışır
    ~Kedi()
    {
        Console.WriteLine("Kedi bellekten atıldı.");
    }
}
```

---

# 3. Erişim Belirteçleri (Access Modifiers)

Kodun güvenliği ve düzeni için sınıf üyelerine erişimi kısıtlayabilir veya açabiliriz. 5 temel belirteç vardır :

1. **public:** Her yerden erişilebilir (Halka açık) .
2. **private:** Sadece tanımlandığı sınıf içinden erişilebilir (Gizli). Varsayılan değerdir .
3. **protected:** Kendi sınıfı ve miras alan alt sınıflardan erişilebilir .

4. **internal:** Sadece aynı proje (derleme) içinden erişilebilir .
5. **protected internal:** Aynı proje veya farklı projedeki alt sınıflardan erişilebilir .

---

## 4. this Anahtar Sözcüğü

Sınıfın o anki örneğini (kendisini) temsil eder . En yaygın kullanım amacı, metot parametresi ile sınıf alanının ismi aynı olduğunda karışıklığı önlemektir .

**Örnek:**

```
public class Ornek
{
   string ad;
   public void AdAta(string ad)
   {
      this.ad = ad; // 'this.ad' sınıfın alanıdır, 'ad' parametredir.
   }
}
```

---

## 5. Kapsülleme: Get ve Set (Properties)

Sınıf alanlarını (fields) doğrudan dışarı açmak yerine, get ve set blokları üzerinden kontrollü erişim sağlamak için kullanılır .

- **get:** Özelliğin değerini okumak için kullanılır .
- **set:** Özelliğe değer atamak için kullanılır. value anahtar kelimesi ile gelen değeri tutar .

### Uygulama Örneği: Toplama İşlemi

```
class Topla
{
   private int tpl; // Dışarıya kapalı alan

   public int ToplamaSonucu
   {
      get { return tpl; }      // Değeri okur
      set { tpl = tpl + value; } // Mevcut toplama yeni değeri ekler
   }
}

// Kullanım
// Topla t = new Topla();
// t.ToplamaSonucu = 5; (Set çalışır, toplama 5 ekler)
// Console.WriteLine(t.ToplamaSonucu); (Get çalışır)
```

# 7.Ünite - Sınıf Yapısına İlişkin Kavramlar

---

Nesneye Yönelik Programlamanın (NYP) karmaşıklığı yönetmek ve kodu güvenli hale getirmek için kullandığı iki temel yapı taşına odaklanır: **Soyutlama (Abstraction)** ve **Kapsülleme (Encapsulation)**.

## 1. Soyutlama (Abstraction)

Soyutlama, karmaşık bir sistemi basit ve anlaşılır bir şekilde modellemek için kullanılır1. Bir nesnenin "nasıl" çalıştığından ziyade "ne" yaptığına odaklanır; gereksiz ayrıntıları gizler ve sadece temel özellikleri vurgular.

C#'ta soyutlama iki ana yapı ile sağlanır:

### A. Soyut Sınıflar (Abstract Classes)

- **Tanım:** Diğer sınıflar için temel oluşturan (ata sınıf), doğrudan nesne üretilemeyen (new yapılamayan) sınıflardır.
- **İçerik:** Hem gövdesiz (soyut) metotlar hem de normal (gövdeli) metotlar barındırabilir.
- **Kural:** Soyut metotların sadece imzası yazılır (abstract anahtar kelimesi ile) ve alt sınıflar bu metotları override ederek (ezerek) doldurmak zorundadır.

**Uygulama Örneği: Araç Sınıfı**

```
using System;

// Soyut Sınıf
abstract class Arac
{
    [cite_start]// Soyut Metot: Gövdesi yok, alt sınıf doldurmak zorunda [cite: 118]
    public abstract double YakitGetir();

    [cite_start]// Normal Metot: Gövdesi var, miras alanlar aynen kullanabilir [cite: 119]
    public void Durdur()
    {
        Console.WriteLine("Araç durduruldu...");
    }
}

// Alt Sınıf
class Otobus : Arac
{
    [cite_start]// Soyut metodu ezerek (override) gövdesini yazıyoruz [cite: 132]
    public override double YakitGetir()
    {
        return 300;
    }
}
```

### B. Arayüzler (Interfaces)

- **Tanım:** Sadece metot ve özellik imzalarını içeren, tamamen soyut bir yapıdır. İçerisinde kod gövdesi bulunmaz.
- **Amaç:** Farklı sınıfların ortak davranışları paylaşmasını sağlar ve C#'taki çoklu kalıtım eksikliğini giderir (Bir sınıf birden fazla arayüzü uygulayabilir).
- **İsimlendirme:** Genellikle isimlerinin başına "I" harfi getirilir (Örn:`IKayit`).

#### Uygulama Örneği: Dosya İşlemleri

```
// Interface Tanımı
interface IDosyaIslemleri
{
    [cite_start]// Sadece imza var, erişim belirteci (public vs) yazılmaz [cite: 172]
    void DosyaAc(string dosyaAdi);
    void DosyaKaydet();
}

// Interface Uygulayan Sınıf
class MetinEditoru : IDosyaIslemleri
{
    // Interface metotlarını public olarak gerçekleştirmek zorundadır
    public void DosyaAc(string dosyaAdi)
    {
        Console.WriteLine(dosyaAdi + " dosyası açıldı.");
    }

    public void DosyaKaydet()
    {
        Console.WriteLine("Dosya kaydedildi.");
    }
}
```

---

## 2. Kapsülleme (Encapsulation)

Kapsülleme, bir nesnenin iç yapısını (verilerini) dış dünyadan gizleyerek, sadece izin verilen yollarla erişim sağlanmasıdır. Bir kapsül gibi veriyi korur.

- **Neden Kullanılır?** Veri güvenliğini sağlamak ve hatalı veri girişini engellemek için.
- **Nasıl Yapılır?**
    1. Alanlar (fields) `private` yapılarak gizlenir.
    2. Bu alanlara erişim `public` özellikler (Properties - Get/Set) üzerinden kontrollü sağlanır.

#### Uygulama Örneği: Öğrenci Kaydı

```
class Ogrenci
{
    [cite_start]// 1. Alanları gizle (Private) [cite: 237]
    private string isim;
    private int yas;

    // Yapıcı Metot (Constructor)
    public Ogrenci(string isim, int yas)
    {
        [cite_start]// "this" kullanımıyla karışıklığı önle [cite: 242]
        this.isim = isim;
        this.yas = yas;
    }

    // 2. Kontrollü Erişim (Property)
    public string Isim
    {
        get { return isim; }
        // Set bloğunda istersek veri kontrolü yapabiliriz
        set { isim = value; }
    }

    public void BilgiGoster()
    {
        // String interpolation kullanımı düzeltildi
        Console.WriteLine($"İsim: {isim}, Yaş: {yas}");
    }
}
```

# 8.Ünite - Sınıf Hiyerarşisi

---

Sınıflar arasındaki ilişkileri düzenleyen, kod tekrarını önleyen ve yazılımın esnekliğini artıran temel hiyerarşik yapıları ele almaktadır.

## 1. Kalıtım (Inheritance)

Kalıtım, bir sınıfın (türetilmiş sınıf/alt sınıf) başka bir sınıfın (temel sınıf/üst sınıf) özelliklerini ve davranışlarını miras almasıdır.

- **Amaç:** Kod tekrarını önlemek, bakımı kolaylaştırmak ve sınıflar arasında hiyerarşik bir düzen kurmaktır.
- **İlişki:** "is-a" (bir ...) ilişkisidir. (Örn: SporAraba *bir* Arabadır).
- **Kural:** C# dilinde bir sınıf sadece **tek bir** sınıftan miras alabilir (Single Inheritance).
- **Sözdizimi:** `class AltSinif : UstSinif` şeklinde tanımlanır.

## 2. Çoklu Kalıtım ve Arayüzler

C# doğrudan çoklu kalıtımı (bir sınıfın birden fazla sınıftan miras almasını) desteklemez. Bunun yerine **Arayüzler (Interfaces)** kullanılır.

- Bir sınıf birden fazla arayüzü (`interface`) uygulayabilir (implement edebilir).
- Bu sayede farklı yetenekler tek bir sınıfta toplanabilir.

## 3. Mühürlü Sınıflar (Sealed Classes)

Kalıtımın engellenmesi gereken durumlar için kullanılır.

- **`sealed`** anahtar kelimesi ile tanımlanır.
- Bu sınıflardan başka bir sınıf türetilemez. Genellikle güvenlik veya sınıf bütünlüğünü korumak için tercih edilir.

## 4. Bileşim (Composition)

Kalıtıma güçlü bir alternatiftir. Bir sınıfın, başka bir sınıfın nesnesini kendi içinde barındırmasıdır.

- **İlişki:** "has-a" (sahiptir) ilişkisidir. (Örn: Bilgisayar işlemciye *sahiptir*).

- **Avantajı:** Sınıflar arasındaki bağımlılığı azaltır (loose coupling) ve esnekliği artırır.

# 5. Çok Biçimlilik (Polymorphism)

Nesnelerin farklı formlarda davranabilme yeteneğidir. 4 temel türü vardır:

1. **Alt Tür (Runtime):** `virtual` ve `override` kullanılarak, alt sınıfların ana sınıfın metodunu kendine göre değiştirmesidir.
2. **Parametrik (Overloading):** `GenericList<T>` gibi yapılarla, kodun farklı veri tipleriyle çalışabilmesidir.
3. **Geçici (Ad Hoc / Compile-time):** Metot aşırı yüklemesidir (Method Overloading). Aynı isimli metodun farklı parametrelerle yazılmasıdır.
4. **Tür Dönüşümü (Coercion):** Bir veri tipinin diğerine dönüştürülmesidir (Örn: `(int)3.14`).

---

## Kod Örnekleri

### Örnek 1: Kalıtım ve Mühürlü Sınıf

```
// Temel Sınıf
class Araba
{
    public string Marka { get; set; }
    public void Calistir()
    {
        Console.WriteLine("Araba çalışıyor...");
    }
}

// Türetilmiş Sınıf (Miras Alan)
// 'sealed' olduğu için bu sınıftan başka sınıf türetilemez.
sealed class SporAraba : Araba
{
    public void TurboAc()
    {
        Console.WriteLine($"{Marka} turbo moduna geçti!");
    }
}

// KULLANIM
// SporAraba s = new SporAraba();
// s.Marka = "Ferrari";
// s.Calistir(); // Miras aldığı metot
// s.TurboAc();  // Kendi metodu
```

### Örnek 2: Bileşim (Composition)

```
class Islemci
{
    public string Model { get; set; }
    public void IslemYap()
    {
        Console.WriteLine($"{Model} işlem yapıyor.");
    }
}

class Bilgisayar
{
    // Bileşim: Bilgisayar bir işlemciye "sahiptir" (Has-a)
    private Islemci _cpu;

    public Bilgisayar()
    {
        _cpu = new Islemci(); // Nesne içinde nesne oluşturma
        _cpu.Model = "Intel i7";
    }

    public void BilgisayariAc()
    {
        Console.WriteLine("Bilgisayar açılıyor...");
        _cpu.IslemYap(); // İçindeki nesnenin metodunu kullanıyor
    }
}
```

### Örnek 3: Çok Biçimlilik (Polymorphism)

```
// 1. Geçici Çok Biçimlilik (Method Overloading)
class Hesapla
{
    // Aynı metot ismi, farklı parametreler
    public int Topla(int a, int b) { return a + b; }
    public double Topla(double a, double b) { return a + b; }
}

// 2. Alt Tür Çok Biçimliliği (Override)
class Hayvan
{
    public virtual void SesCikar() // 'virtual' ezilebilir demek
    {
        Console.WriteLine("Hayvan ses çıkarıyor.");
    }
}

class Kedi : Hayvan
{
    public override void SesCikar() // 'override' ile eziyoruz
    {
        Console.WriteLine("Miyav!");
    }
}
```

# 9.Ünite - Operatör Aşırı Yükleme

---

C# programlama dilinde operatörlerin (matematiksel, mantıksal, karşılaştırma vb.) kullanımını ve kullanıcı tanımlı sınıflar (class) için bu operatörlerin nasıl özelleştirileceğini (aşırı yükleme/overloading) ele almaktadır.

## 1. Operatörler ve Temel Kavramlar

Operatörler, değişkenler ve değerler üzerinde işlem yapmamızı sağlayan özel sembollerdir.

- **Aritmetik Operatörler:** `+`, `-`, `*`, `/`, `%` (mod), `++` (arttırma), `--` (azaltma).
- **Karşılaştırma Operatörleri:** `==`, `!=`, `<`, `>`, `<=`, `>=`. Sonuç her zaman `true` veya `false` döner.
- **Mantıksal Operatörler:** `&&` (VE), `||` (VEYA), `!` (DEĞİL)
- **Bit Tabanlı Operatörler:** Bit seviyesinde işlem yapar. `&` (VE), `|` (VEYA), `^` (XOR), `~` (Tersleme), `<<` (Sola kaydırma), `>>` (Sağa kaydırma)
- **Özel Amaçlı Operatörler:**
    - `?:` (Ternary): Kısa if-else yapısıdır (`koşul ? doğruysa : yanlışsa`)5.
    - `checked` / `unchecked`: Tamsayı işlemlerinde taşma (overflow) kontrolü yapar.
    - `typeof`: Bir nesnenin tür bilgisini (`System.Type`) verir.

### Tip Kontrolü ve Dönüşümü

C#'ta tip güvenliği için iki önemli operatör vardır:

1. **`is` Operatörü:** Nesnenin belirtilen tiple uyumlu olup olmadığını kontrol eder. `true` / `false` döner.
2. **`as` Operatörü:** Güvenli dönüşüm yapar. Dönüşüm başarısız olursa hata fırlatmaz, `null` değeri döndürür.

## 2. Operatör Aşırı Yükleme (Operator Overloading)

Kendi tanımladığımız sınıfların (örneğin `KarmasikSayi` veya `Vektor`), standart operatörlerle (`+`, `-`, `==` vb.) işlem yapabilmesini sağlamak için kullanılır.

- **Kural:** Operatör metotları mutlaka `public` ve `static` olmalıdır.
- **Syntax:** `public static DönüşTipi operator OperatörSembolü (Parametreler)`

### A. Aritmetik Operatörlerin Aşırı Yüklenmesi

İki nesneyi toplamak veya çıkarmak için kullanılır.

### Uygulama Örneği: Nokta (Point) Toplama

```
public class Nokta
{
    public int X { get; set; }
    public int Y { get; set; }

    public Nokta(int x, int y)
    {
        X = x;
        Y = y;
    }

    // '+' operatörünü aşırı yükleme
    public static Nokta operator +(Nokta nokta1, Nokta nokta2)
    {
        // İki noktanın X ve Y değerlerini toplayıp yeni bir Nokta döner
        return new Nokta(nokta1.X + nokta2.X, nokta1.Y + nokta2.Y);
    }
}

// Kullanım:
// Nokta n1 = new Nokta(3, 5);
// Nokta n2 = new Nokta(2, 4);
// Nokta toplam = n1 + n2; // (5, 9) olur
```

## B. İlişkisel (Karşılaştırma) Operatörlerin Aşırı Yüklenmesi

Nesnelerin eşitliğini veya büyüklüğünü karşılaştırmak için kullanılır.

- **Önemli Kural:** İlişkisel operatörler çiftler halinde yüklenmelidir. Eğer `==` yüklenirse, `!=` de yüklenmelidir. Aynı şekilde `<` yüklenirse `>` de yüklenmelidir.

### Uygulama Örneği: Öğrenci Karşılaştırma

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

    // '==' operatörü
    public static bool operator ==(Ogrenci o1, Ogrenci o2)
    {
        // Null kontrolü önemlidir
        if (ReferenceEquals(o1, null) || ReferenceEquals(o2, null))
            return ReferenceEquals(o1, o2);

        return (o1.Ad == o2.Ad && o1.Yas == o2.Yas);
    }

    // '!=' operatörü (Çift olma zorunluluğu)
    public static bool operator !=(Ogrenci o1, Ogrenci o2)
    {
        return !(o1 == o2);
    }

    // Not: == operatörü ezildiğinde Equals() ve GetHashCode() metotlarının da
    // ezilmesi (override) önerilir, ancak basitlik adına burada gösterilmemiştir.
}
```

## C. Dönüşüm Operatörleri (Implicit / Explicit)

Bir sınıfın başka bir tipe nasıl dönüştürüleceğini belirler.

- **`implicit` (Kapalı/Otomatik):** Veri kaybı riski olmayan, güvenli dönüşümler için kullanılır. Derleyici otomatik yapar.
- **`explicit` (Açık/Zorunlu):** Veri kaybı olabilecek durumlarda kullanılır. Parantez içinde tür belirtmek `(int)nesne` gerekir.

### Uygulama Örneği: Mesafe Dönüşümü (Metre <-> Santimetre)

```
public class Mesafe
{
    public double Metre { get; set; }

    public Mesafe(double metre)
    {
        this.Metre = metre;
```

```
    }

    // Implicit: Mesafe -> double (Otomatik dönüşüm)
    // Metreyi santimetreye çevirip double döner
    public static implicit operator double(Mesafe m)
    {
        return m.Metre * 100;
    }

    // Explicit: double -> Mesafe (Zorunlu dönüşüm)
    // Santimetreyi metreye çevirip Mesafe nesnesi döner
    public static explicit operator Mesafe(double santimetre)
    {
        return new Mesafe(santimetre / 100);
    }
}

// Kullanım:
// Mesafe m = new Mesafe(5);
// double cm = m; // Implicit (500 olur)
// Mesafe m2 = (Mesafe)450.0; // Explicit (4.5 metre olur)
```

# 10.Ünite - Metotlar ve Aşırı Yükleme

---

C# programlamanın temel yapı taşlarından olan metotları, parametre yönetimini ve Nesneye Yönelik Programlamanın (NYP) önemli kavramlarından olan Metot Aşırı Yükleme (Overloading) ve Geçersiz Kılma (Overriding) konularını ele almaktadır.

## 1. Metotların Temel Yapısı

Metotlar, belirli bir işlemi gerçekleştiren ve tekrar tekrar kullanılabilen kod bloklarıdır. Kodun modüler olmasını ve tekrarın önlenmesini sağlarlar.

- **Tanımlama:** [Erişim Belirteci] [Dönüş Tipi] [Metot Adı] (Parametreler)
- **Void Metotlar:** Geriye bir değer döndürmeyen metotlardır. `return` ifadesi değer almadan kullanılabilir veya hiç kullanılmaz.
- **Static Metotlar:** Sınıftan nesne türetilmesine gerek kalmadan, sınıf ismiyle doğrudan çağrılabilen metotlardır.

## 2. Parametre Aktarım Yöntemleri

C#'ta metotlara veri gönderirken değişkenin türüne ve kullanılan anahtar kelimeye göre bellek yönetimi değişir.

### A. Değer ve Referans Türleri

- **Değer Türü (Value Type):** `int`, `double`, `bool` gibi türlerdir. Metoda gönderildiğinde **kopyası** oluşturulur. Metot içindeki değişiklik ana değişkeni etkilemez.
- **Referans Türü (Reference Type):** Sınıflar (`class`), diziler ve string gibi türlerdir. Metoda **bellek adresi** gönderilir. Metot içindeki değişiklik ana nesneyi değiştirir.

### B. `ref` ve `out` Anahtar Kelimeleri

Değer türlerinin de referans gibi davranmasını (orijinal verinin değişmesini) sağlamak için kullanılırlar.

- **`ref`:** Değişkenin metoda gönderilmeden önce **mutlaka ilk değerinin atanmış olması** gerekir.
- **`out`:** Değişkenin ilk değerinin olmasına gerek yoktur, ancak metodun içerisinde **mutlaka bir değer ataması yapılmalıdır**.

## 3. Metot Aşırı Yükleme (Method Overloading)

Aynı isme sahip fakat parametre yapısı (sayısı veya türü) farklı olan birden fazla metodun tanımlanmasıdır.

- **Amaç:** Kod okunabilirliğini ve esnekliği artırmaktır.
- **Kural:** Parametre imzası farklı olmalıdır. Sadece geri dönüş tipinin farklı olması aşırı yükleme sayılmaz ve hata verir.

## 4. Metot Geçersiz Kılma (Method Overriding)

Kalıtım alınan bir sınıftaki metodun, alt sınıfta (türetilen sınıf) yeniden yazılarak davranışının değiştirilmesidir.

- **Base Class (Temel Sınıf):** Metot `virtual` olarak işaretlenmelidir.
- **Derived Class (Alt Sınıf):** Metot `override` olarak işaretlenmelidir.

# 5. Değişken Sayıda Parametre (`params`)

Bir metoda kaç adet parametre gönderileceğinin bilinmediği durumlarda kullanılır.

- **Kural:** `params` anahtar kelimesi ile tanımlanır ve bir dizi (`array`) tipinde olmalıdır.
- **Kısıt:** Metodun parametre listesinde en sonda yer almalıdır ve sadece bir tane `params` kullanılabilir.

# 6. Özyinelemeli (Recursive) Metotlar

Bir metodun, belirli bir şart sağlanana kadar kendi kendisini çağırmasıdır. Genellikle matematiksel hesaplamalarda (Faktöriyel, Fibonacci vb.) kullanılır. Sonsuz döngüye girmemesi için mutlaka bir "durdurma koşulu" (base case) olmalıdır.

---

## Kod Örnekleri

### Örnek 1: `ref` ve `out` Kullanımı

```
class ParametreOrnekleri
{
    // ref: Değişkenin değeri metoda girmeden atanmış olmalı
    public static void SayiyiDegistir(ref int sayi)
    {
        sayi = 100; // Orijinal değişkeni değiştirir
    }

    // out: Değer metot içinde atanmak zorundadır
    public static void KisiOlustur(out string ad)
    {
        ad = "Ahmet"; // Dışarıya bu değeri fırlatır
    }
}

// Kullanım:
// int a = 5;
// ParametreOrnekleri.SayiyiDegistir(ref a); // a artık 100 olur.
// string isim;
// ParametreOrnekleri.KisiOlustur(out isim); // isim "Ahmet" olur.
```

### Örnek 2: Aşırı Yükleme (Overloading)

```
class Hesapla
{
    // İki tamsayıyı toplar
    public int Topla(int s1, int s2)
    {
        return s1 + s2;
    }

    // Üç tamsayıyı toplar (Aşırı yükleme)
    public int Topla(int s1, int s2, int s3)
    {
        return s1 + s2 + s3;
    }

    // İki ondalıklı sayıyı toplar (Aşırı yükleme)
    public double Topla(double s1, double s2)
    {
        return s1 + s2;
    }
}
```

### Örnek 3: Geçersiz Kılma (Overriding)

```
class Sekil
{
    // Alt sınıflar bu metodu ezebilir (virtual)
    public virtual void Ciz()
```

```
    {
        Console.WriteLine("Bir şekil çizildi.");
    }
}

class Daire : Sekil
{
    // Temel sınıftaki metodu geçersiz kılıyoruz (override)
    public override void Ciz()
    {
        Console.WriteLine("Daire çizildi.");
    }
}
```

### Örnek 4: `params` ve Recursive (Faktöriyel)

```
class OzelIslemler
{
    // Değişken sayıda parametre toplama
    public int CokluTopla(params int[] sayilar)
    {
        int toplam = 0;
        foreach (int sayi in sayilar)
        {
            toplam += sayi;
        }
        return toplam;
    }

    // Recursive Faktöriyel
    public int Faktoriyel(int n)
    {
        if (n <= 1) return 1; // Durdurma koşulu (Base case)
        return n * Faktoriyel(n - 1); // Kendini çağırma
    }
}

// Kullanım:
// OzelIslemler islem = new OzelIslemler();
// int sonuc = islem.CokluTopla(10, 20, 30, 40); // İstediğimiz kadar sayı gönderebiliriz.
// int fakt = islem.Faktoriyel(5); // 120
```