

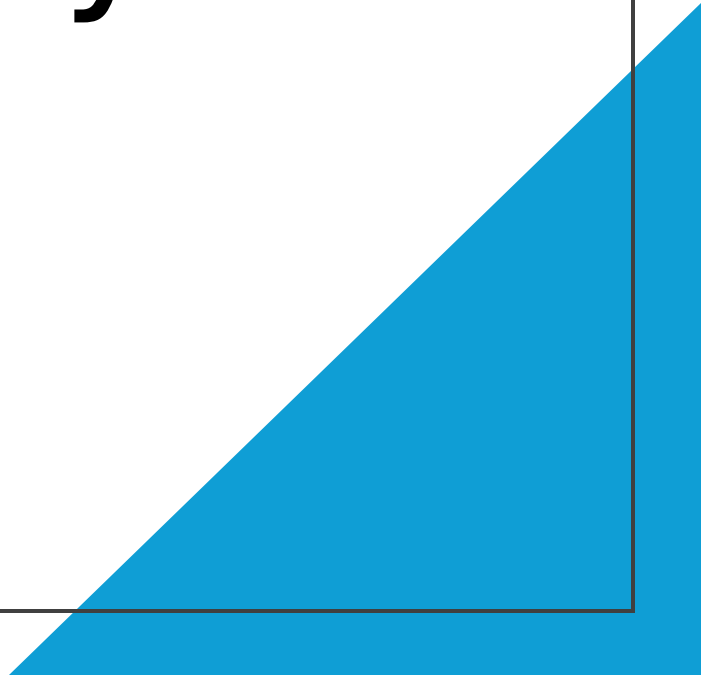
SOLID İlkeleri ve NYP İlkeleri

Hazırlayan:
Oğuz KIRAT

SOLID Principles

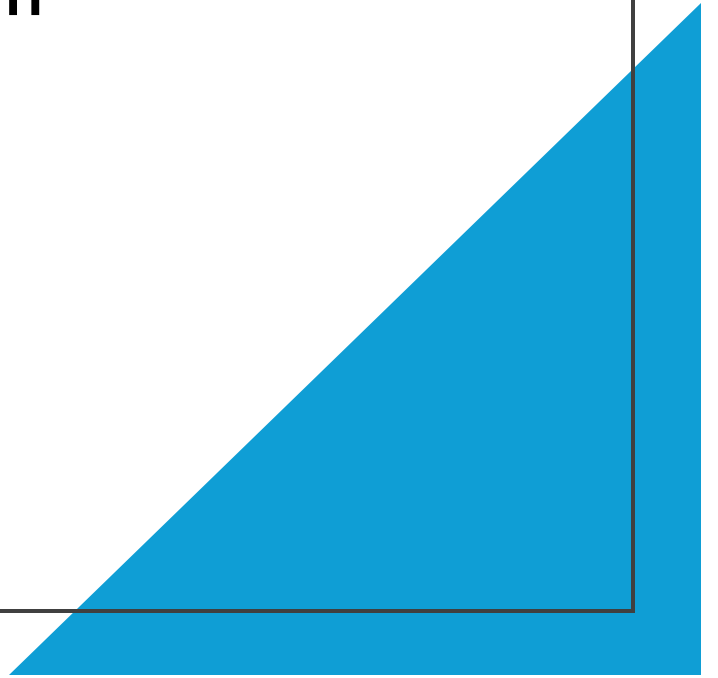
- **S** — Single-responsibility principle – Tek Sorumluluk
- **O** — Open-closed principle – Açık/Kapalı
- **L** — Liskov substitution principle – Linkov Yerine Geçme
- **I** — Interface segregation principle – Arayüz Ayrımı
- **D** — Dependency Inversion Principle – Bağımlılığın Tersine Çevrilmesi

Tek Sorumluluk Prensibi (Single Responsibility Principle)



Tek Sorumluluk Prensibi (Single Responsibility Principle)

Bir sınıf yalnızca bir işi yapmalı, tek bir sorumluluğa sahip olmalıdır.



```
// Tek sınıf birden fazla sorumluluk taşıyor

public class Kullanici {

    private String ad;

    private String email;

    // Kullanıcı verileri ile ilgili metotlar

    public void adDegistir(String yeniAd) {

        this.ad = yeniAd;

    }

    // Veritabanı işlemleri

    public void veritabaninaKaydet() {

    }

    // Email işlemleri

    public void hosgeldinEmailiGonder() {

    }

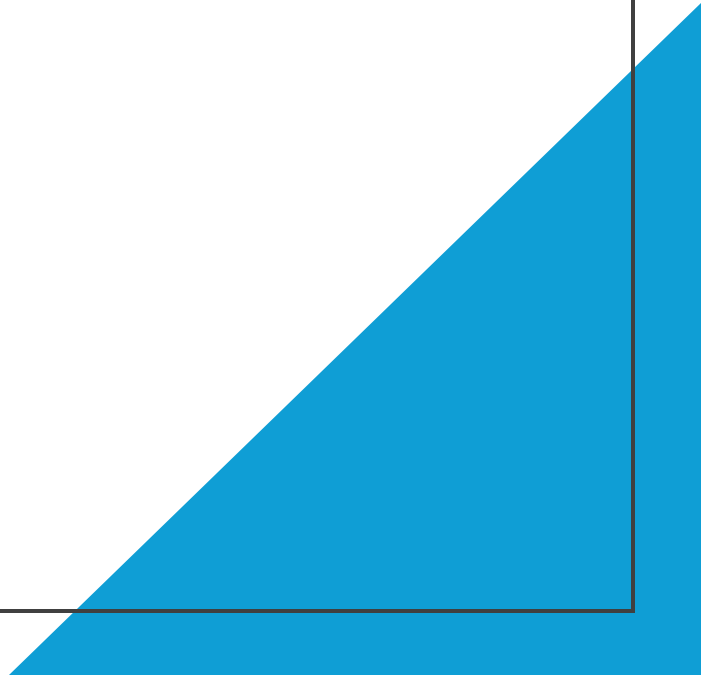
}
```

```
// Her sınıf tek bir sorumluluk taşır
public class KullaniciRepository {
    public void kullaniciKaydet(Kullanici kullanici) {
        // Veritabanına kullanıcı kaydetme işlemleri
    }

    public Kullanici kullaniciBul(int id) {
        // Kullanıcı bulma işlemleri
        return new Kullanici();
    }
}

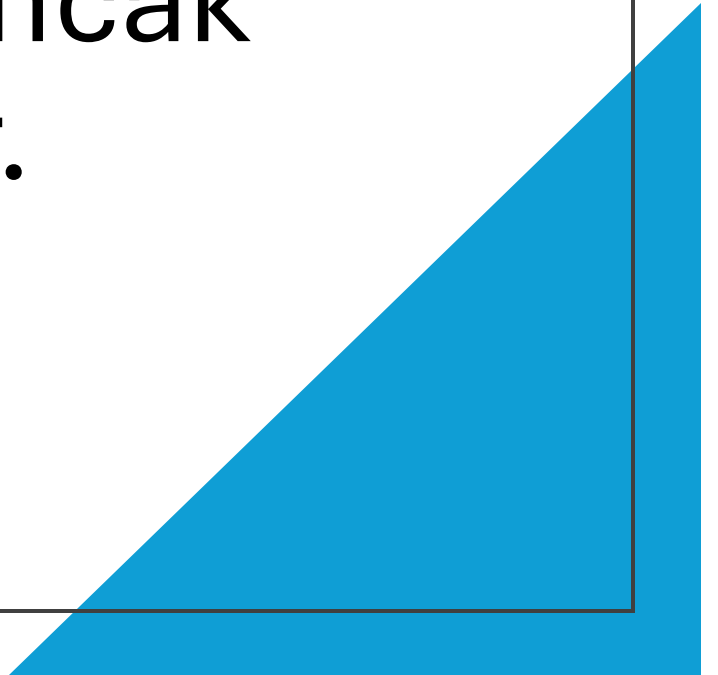
public class EmailServisi {
    public void emailGonder(String alici, String konu,
String icerik) {
        // Email gönderme işlemleri
    }
}
```

Açık/Kapalı Prensibi (Open/Closed Principle)



Açık/Kapalı Prensibi (Open/Closed Principle)

Sınıflar değişikliğe kapalı, ancak genişletmeye açık olmalıdır.




```
public class AlanHesaplayici {  
    public double alanHesapla(Object sekil) {  
        if (sekil instanceof Dikdortgen) {  
            Dikdortgen d = (Dikdortgen) sekil;  
            return d.getGenislik() * d.getYukseklik();  
        }  
        else if (sekil instanceof Daire) {  
            Daire d = (Daire) sekil;  
            return Math.PI * d.getYaricap() * d.getYaricap();  
        }  
        // Yeni şekil eklendiğinde bu sınıfı değiştirmek zorundayız  
        return 0;  
    }  
}
```

```
public abstract class Sekil {  
    public abstract double alanHesapla();  
}
```

```
public class Dikdortgen extends Sekil {  
    private double genislik;  
    private double yukseklik;
```

```
    public Dikdortgen(double genislik, double yukseklik) {  
        this.genislik = genislik;  
        this.yukseklik = yukseklik;  
    }
```

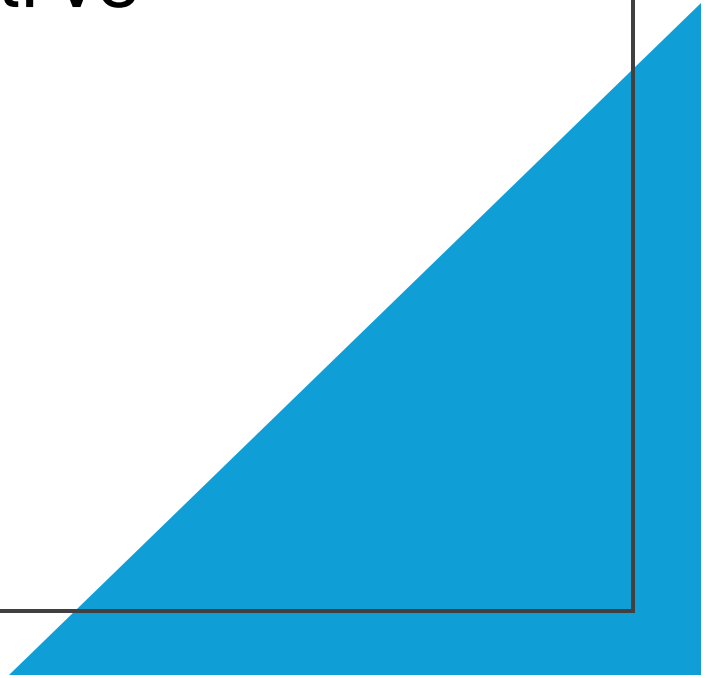
```
@Override
```

```
    public double alanHesapla() {  
        return genislik * yukseklik;  
    }
```

```
}
```

Liskov Yerine Geme Prensipleri (Liskov Substitution Principle)

Alt sınıflar, st sınıfların yerine geebilmeli ve programın doėruluėunu bozmadan alıřabilmelidir.



```
public class Kus {  
    public void uc() {  
        System.out.println("Kuş uçuyor");  
    }  
  
    public void yemekYe() {  
        System.out.println("Kuş yemek yiyor");  
    }  
}
```

```
public class Penguen extends Kus {  
    @Override  
    public void uc() {  
        // Penguen uçamaz, bu yüzden bu metot mantıksız  
        throw new UnsupportedOperationException("Penguenler uçamaz!");  
    }  
}
```

```
// Kullanım  
public static void main(String[] args) {  
    Kus kus = new Penguen();  
    // Bu çağrı hata verecek ve programın  
    akışını bozacak. Penguen, Kuş tipinin yerine  
    geçemiyor.  
    kus.uc();  
}
```

```
// Uçabilen kuşlar için ayrı arayüz
public interface Ucabilir {
    void uc();
}

public class Serce implements Ucabilir {
    @Override
    public void uc() {
        // serce uçuş kodu
    }
}

public class Penguen {
    public void yuru() {
        // penguen yürüme kodu
    }
}

// Her sınıf yalnızca uygun arayüzü implement eder; LSP korunur.
```

Arayüz Ayrımı Prensipli (Interface Segregation Principle)

- Sınıflar kullanmayacakları metotları içeren arayüzleri uygulamaya zorlanmamalıdır.



// Tek büyük arayüz

```
public interface CihazIslemleri {  
    void yazdir(String belge);  
    void tara(String belge);  
    void faksGonder(String belge);  
    void renkliYazdir(String belge);  
}
```

// Basit yazıcı sınıfı, kullanmadığı metotları da uygulamak zorunda

```
public class BasitYazici implements CihazIslemleri {  
    @Override  
    public void yazdir(String belge) {  
        System.out.println("Belge yazdırılıyor: " +  
belge);  
    }  
  
    @Override  
    public void tara(String belge) {  
        // Kullanılmayan işlem  
        throw new UnsupportedOperationException("Basit  
yazıcı tarama yapamaz");  
    }  
  
    @Override  
    public void faksGonder(String belge) {  
        // Kullanılmayan işlem  
        throw new UnsupportedOperationException("Basit  
yazıcı faks gönderemez");  
    }  
}
```

✗ Yanlış Örnek

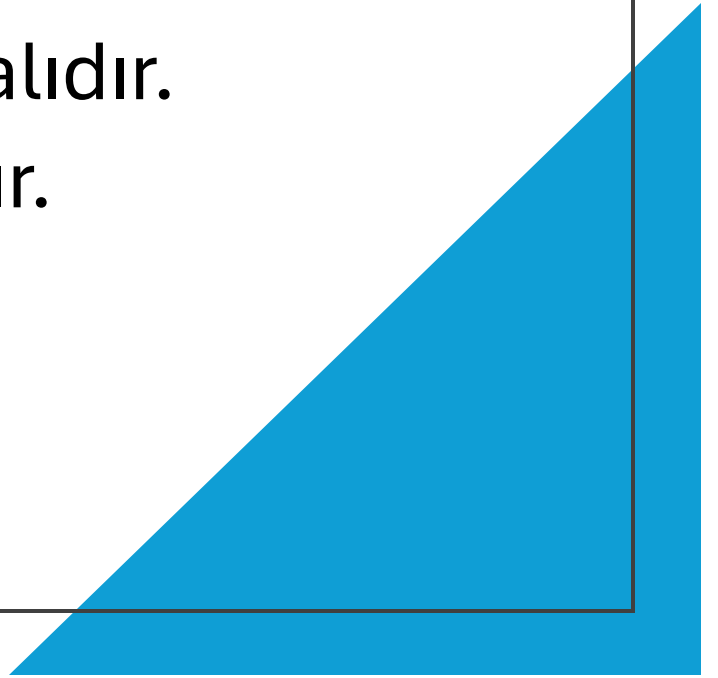
// Ayrı arayüzler

```
public interface Yazici {  
    void yazdir(String belge);  
}  
  
public interface Tarayici {  
    void tara(String belge);  
}  
  
public interface Faksci {  
    void faksGonder(String belge);  
}  
  
public class BasitYazici implements  
    Yazici {  
    @Override  
    public void yazdir(String belge) {  
        System.out.println("Belge  
yazdırılıyor: " + belge);  
    }  
}
```

```
public class CokFonksiyonluYazici implements Yazici,  
    Tarayici, Faksci {  
    @Override  
    public void yazdir(String belge) {  
        System.out.println("Belge yazdırılıyor: " +  
belge);  
    }  
  
    @Override  
    public void tara(String belge) {  
        System.out.println("Belge taranıyor: " + belge);  
    }  
  
    @Override  
    public void faksGonder(String belge) {  
        System.out.println("Faks gönderiliyor: " +  
belge);  
    }  
}
```


Bağımlılığın Ters Çevrilmesi Prensibi (Dependency Inversion Principle)

- Yüksek seviyeli modüller, düşük seviyeli modüllere bağımlı olmamalıdır.
- Her ikisi de soyutlamalara bağılı olmalıdır.



```
// Soyutlama (arayüz)

public interface VeriTani {
    void kaydet(String veri);
    String getir(int id);
}

// Düşük seviyeli modül

public class PostgreSQLVeriTani implements
    VeriTani {
    @Override
    public void kaydet(String veri) {
        System.out.println("PostgreSQL'e
kaydediliyor: " + veri);
    }

    @Override
    public String getir(int id) {
        return "PostgreSQL'den " + id + " numaralı
veri";
    }
}
```

```
// Yüksek seviyeli modül
public class KullaniciServisi {
    private final VeriTani veriTabani;

    // Bağımlılık enjeksiyonu
    public KullaniciServisi(VeriTani veriTabani)
    {
        this.veriTani = veriTabani;
    }

    public void kullaniciKaydet(String
kullaniciVerisi) {
        veriTabani.kaydet(kullaniciVerisi);
    }

    public String kullaniciBul(int id) {
        return veriTabani.getir(id);
    }
}
```

```
// Düşük seviyeli modül
public class PostgreSQLVeriTabani {
    public void kaydet(String veri) {
        System.out.println("PostgreSQL'e
kaydediliyor: " + veri);
    }

    public String getir(int id) {
        return "PostgreSQL'den " + id + "
numaralı veri";
    }
}
• }
```

```
// Yüksek seviyeli modül doğrudan düşük seviyeli
modüle bağımlı
public class KullaniciServisi {
    private final PostgreSQLVeriTabani veriTabani;

    public KullaniciServisi() {
        // Doğrudan bağımlılık - kötü
        this.veriTabani = new PostgreSQLVeriTabani();
    }

    public void kullaniciKaydet(String
kullaniciVerisi) {
        veriTabani.kaydet(kullaniciVerisi);
    }

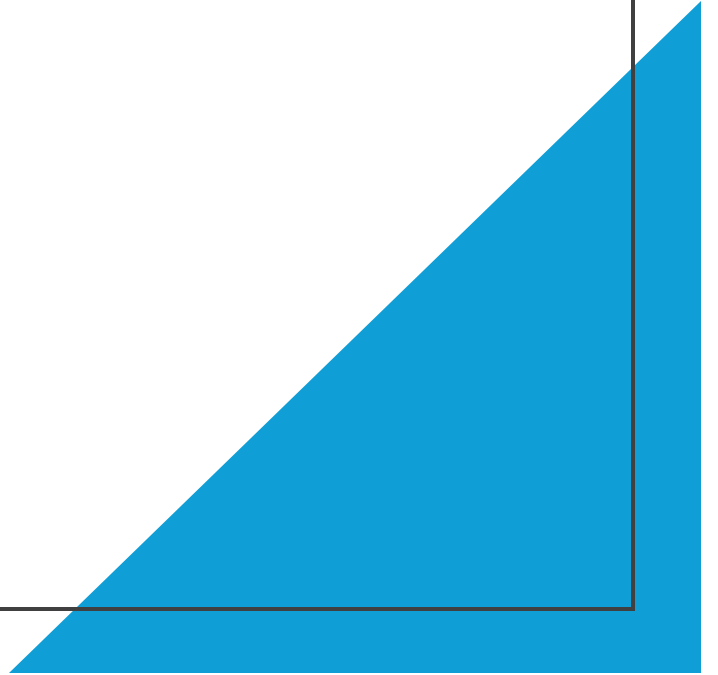
    public String kullaniciBul(int id) {
        return veriTabani.getir(id);
    }
}
```

Nesne Nedir?

Nesne Nedir?

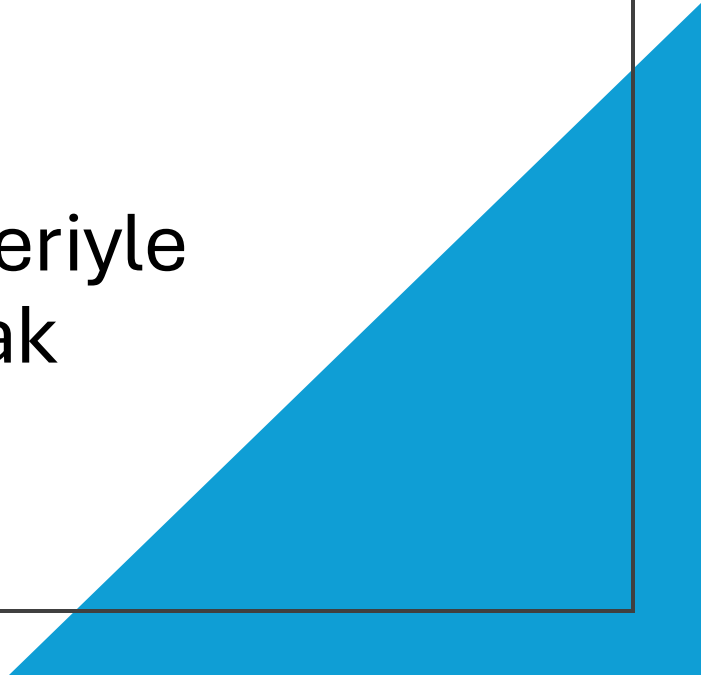
- Nesne belirli bir kavramı veya gerçek dünya varlığını modellemek amacıyla oluşturulan bağımsız yapılardır.
- Nesne (Object), yazılım geliştirme bağlamında, özellikler (attributes/properties) ve davranışlar (behaviors/methods) içeren bağımsız bir yapıdır
- Özellikler, nesnenin durumunu tanımlayan verileri temsil ederken; davranışlar, nesnenin gerçekleştirebildiği işlemleri belirtir.
- Örneğin, bir "Araba" nesnesi; renk, model, hız gibi özelliklere ve hızlan, fren yap gibi davranışlara sahip olabilir.

Nesne Yönelik Programlama (OOP) Nedir?

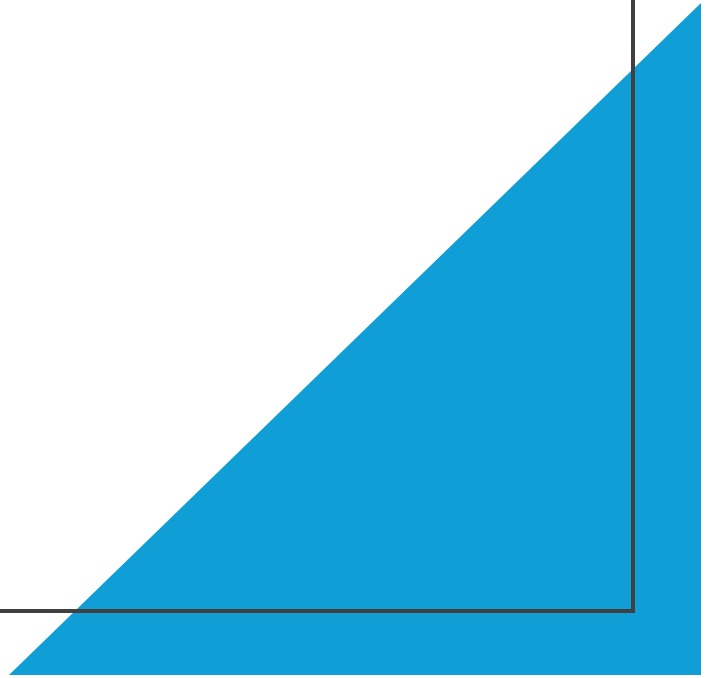


NYP Nedir?

- Nesneye yönelik programlama (Object-Oriented Programming, OOP), yazılımı nesneler aracılığıyla modelleyen bir programlama paradigmasıdır.
- Bu paradigmada, yazılım sistemi, birbirleriyle etkileşim kuran nesneler topluluğu olarak organize edilir.



Neden OOP
kullanıyoruz? Neden
ihtiyaç var?

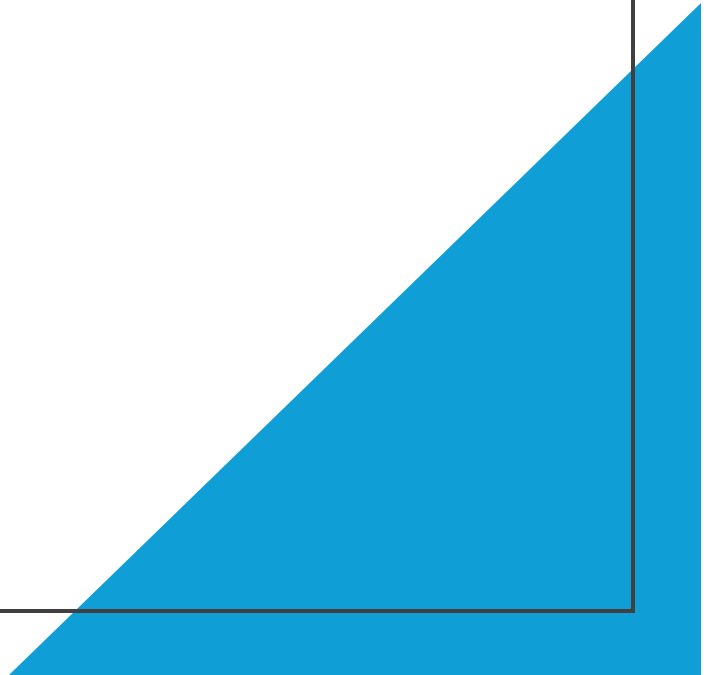


NYP'ye Neden İhtiyaç Var?

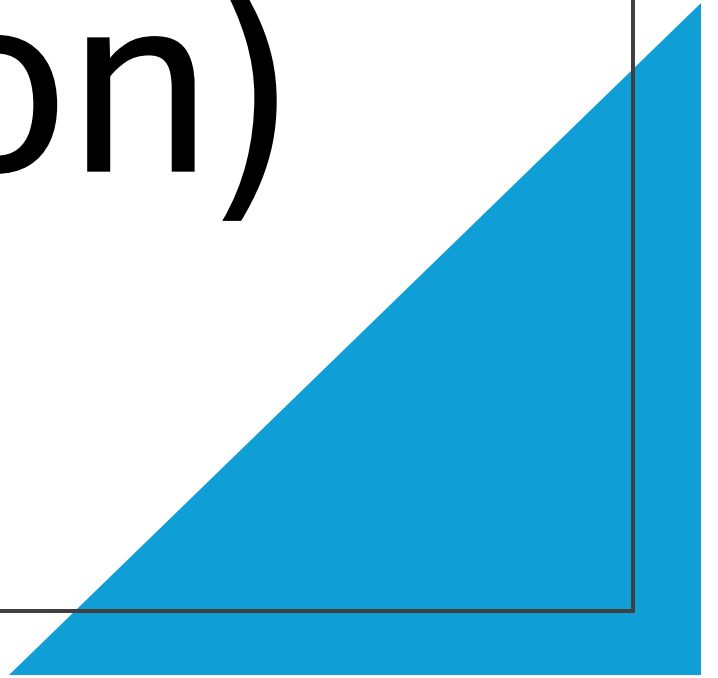
- **Gerçek Dünya'yı Daha Kolay Modelleme**
- **Yeniden Kullanılabilirlik**
- **Bakım ve Geliştirilebilirlik**
- **Düşük Bağımlılık/Yüksek Bütünlülük**
*(DB – Low Coupling): Bir modülün başka modüllere olan bağımlılığının **az olması** anlamına gelir. Bir modül kendi başına çalışabilir; diğer modüllerde yapılan değişikliklerden en az düzeyde etkilenir.*
(YB – High Cohesion: Bir modülün yalnızca tek ve net bir sorumluluğa sahip olması, içindeki tüm işlemlerin bu sorumlulukla doğrudan ilgili olması)
- **Daha İyi Hata Yönetimi ve Test Edilebilirlik**
- **Kodun Daha Rahat Okunması/Anlaşılması**

NYP'nin Temel İlkeleri

- Kapsülleme (Encapsulation)
- Kalıtım (Inheritance)
- Çok Biçimlilik (Polymorphism)
- Soyutlama (Abstraction)



Kapsülleme (Encapsulation)



Kapsülleme (Encapsulation)

Nesne içindeki veri (alan) ve metotlar, erişim belirleyiciler (private, public, protected) ile korunarak dışarıdan doğrudan müdahale engellenir.

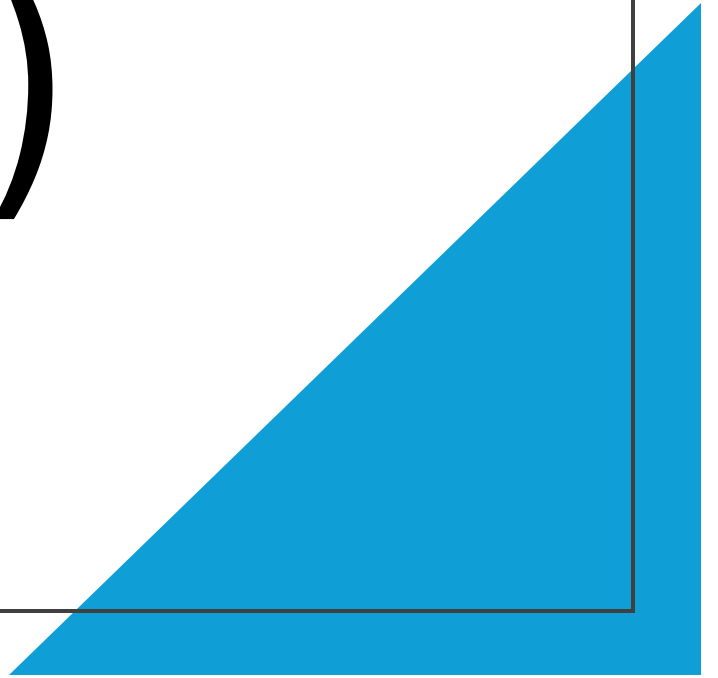
Bu sayede nesnenin iç durumu kontrol altında tutulur.

A large blue right-angled triangle is positioned in the bottom right corner of the slide, pointing towards the top right.

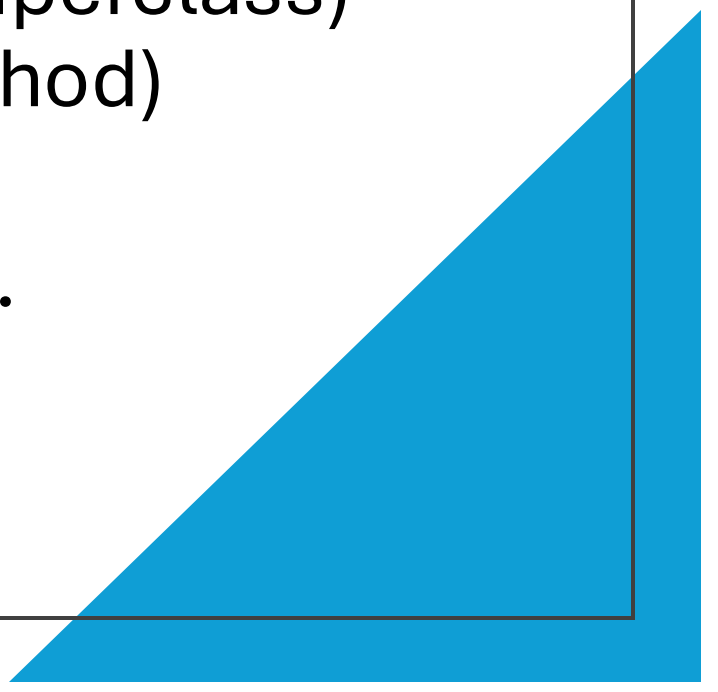
```
public class Ogrenci {  
    private String isim;  
    // dışarıdan doğrudan erişilemez  
    private int yas;  
    public String getIsim() {  
    // kontrollü erişim  
        return isim;  
    }  
    public void setIsim(String isim) {  
        if (!isim.isEmpty())  
    // basit bir doğrulama  
            this.isim = isim;  
    }
```

```
        public int getYas() {  
            return yas;  
        }  
        public void setYas(int yas) {  
            if (yas > 0)  
                this.yas = yas;  
        }  
    }
```

Kalıtım (Inheritance)



Kalıtım (Inheritance)

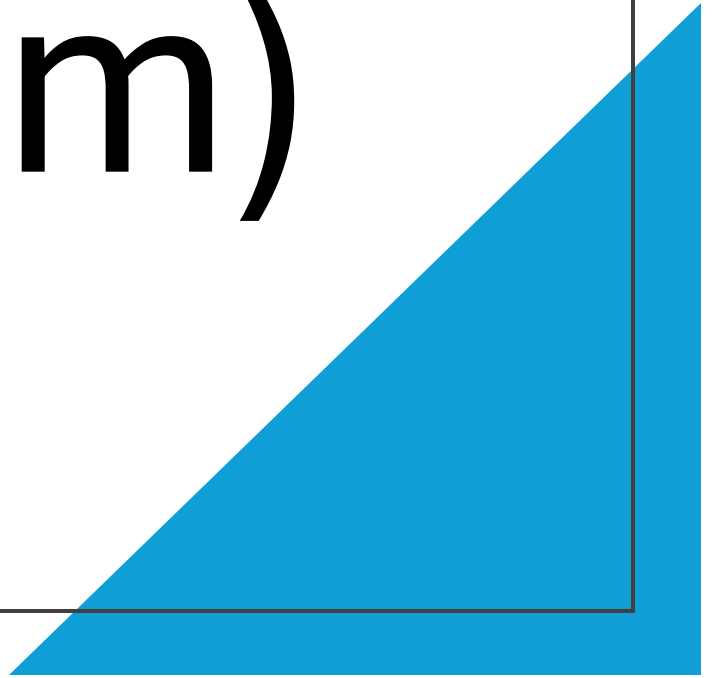
- Bir sınıfın (subclass) başka bir sınıfın (superclass) özelliklerini ve davranışlarını (field ve method) devralmasına izin verir.
 - Kod tekrarını önler ve hiyerarşi oluşturur.
- 

```
// Üst sınıf (superclass)
public class Canli {
    public void nefesAl() {
        System.out.println("Canlı nefes alıyor.");
    }
}
```

```
// Alt sınıf (subclass) Canli'den türetilir
public class Kedi extends Canli {
    public void miyavla() {
        System.out.println("Kedi miyavlıyor.");
    }
}
```

```
// Kullanım
Kedi k = new Kedi();
k.nefesAl();    // Canlı sınıfından miras
k.miyavla();    // Kedi'ye ait metot
```


Çok Biçimlilik (Polymorphism)



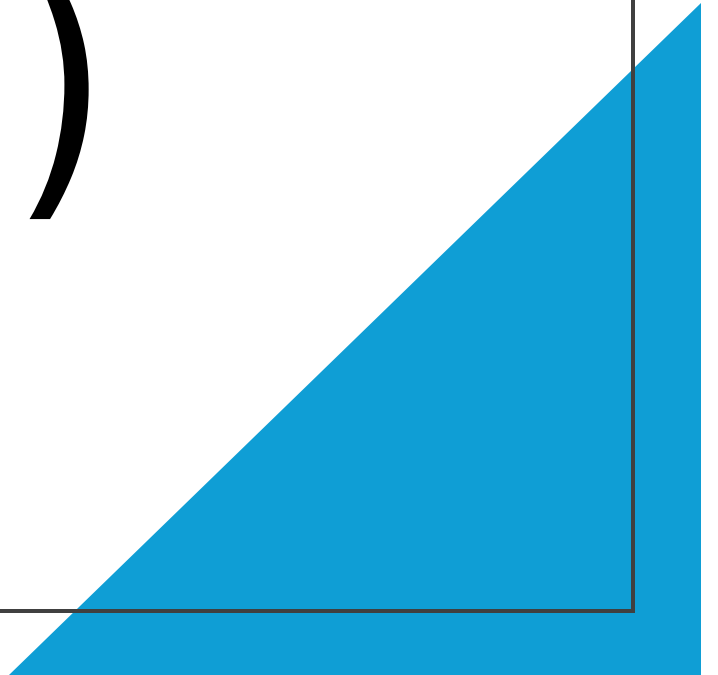
Çok Biçimlilik (Polymorphism)

- Nesnelerin aynı arayüzü (interface) veya üst sınıfı kullanarak farklı biçimlerde davranabilmesidir.
 - Aynı metot adının farklı parametrelerle tanımlanması
 - Alt sınıfların üst sınıftan devraldıkları metodu kendilerine özgü olarak yeniden tanımlaması.

```
public class HesapMakinesi {  
    public int topla(int a, int b) {  
        return a + b;  
    }  
    public double topla(double a, double b, double c) {  
        return a + b + c;  
    }  
}
```

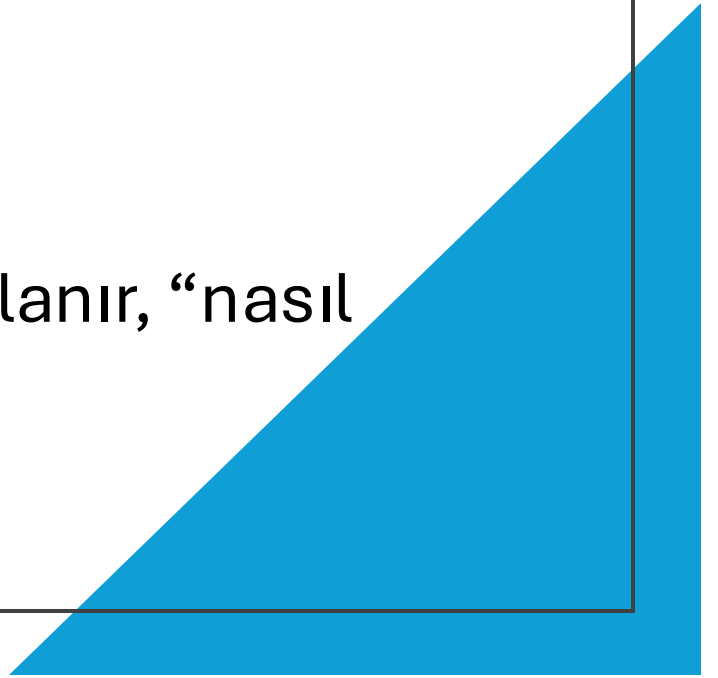
```
class Hayvan {  
    public void sesCikar() {  
        System.out.println("Hayvan ses çıkarıyor.");  
    }  
}  
class Kus extends Hayvan {  
    @Override  
    public void sesCikar() {  
        System.out.println("Kuş cıvıldıyor.");  
    }  
}
```

Soyutlama (Abstraction)



Soyutlama (Abstraction)

- Karmaşık sistemleri, sadece gerekli özellik ve işlemleri ortaya koyarak sadeliğe indirger.
- Java'da abstract sınıflar ve interface yapıları ile gerçekleştirilir;
- Detaylar gizlenir, sadece “ne yapılacağı” tanımlanır, “nasıl yapılacağı” alt sınıflara bırakılır.



```
// Soyut sınıf örneği
public abstract class Sekil {
    public abstract double alanHesapla();
    // nasıl hesaplanacağı alt sınıf belirler

    public void bilgiYaz() {
        System.out.println("Şekil alanı: "
+ alanHesapla());
    }
}
```

```
// Alt sınıf: Daire
public class Daire extends Sekil {
    private double yariCap;
    public Daire(double r) { this.yariCap = r; }

    @Override
    public double alanHesapla() {
        return Math.PI * yariCap * yariCap;
    }
}

// Kullanım
Sekil s = new Daire(5);
s.bilgiYaz(); // "Şekil alanı: 78.5398..."
```