## 1. Neden Nesne Yönelimli Programlama (OOP) Kullanmalıyız?

1. **Kodun Yeniden Kullanılabilirliği (Code Reusability):**
    - OOP'de **kalıtım (inheritance)** sayesinde mevcut kodları tekrar yazmadan kullanabiliriz.
    - Örneğin, bir **"Araç"** sınıfı oluşturup, "Araba" ve "Motosiklet" sınıflarının bu sınıftan türemesini sağlayabiliriz.
2. **Kodun Düzenli ve Yönetilebilir Olması (Code Organization & Maintainability):**
    - OOP, kodu küçük ve yönetilebilir parçalara (nesnelere) böler, böylece büyük projeler daha kolay yönetilir.
3. **Veri Güvenliği (Encapsulation - Data Security):**
    - **Kapsülleme (Encapsulation)** ile verileri gizleyebilir ve yalnızca belirli yöntemlerle erişime izin verebiliriz.
4. **Gerçek Dünya Modellemesi (Real-World Modeling):**
    - Nesneler ve sınıflar, gerçek dünyadaki varlıkları modellemeye uygundur.
    - Örneğin, bir "Öğrenci" nesnesi adı, yaşı ve notları gibi özelliklere sahip olabilir.
5. **Esneklik ve Modülerlik (Flexibility & Modularity):**
    - **Polimorfizm (Polymorphism)** sayesinde aynı kod farklı şekillerde çalışabilir.
    - Farklı nesneler için aynı metod ismi kullanılabilir.

- **Code Reusability**: Inheritance allows us to reuse existing code.
- **Maintainability**: OOP structures the code into manageable objects.
- **Encapsulation**: Ensures data security by restricting direct access.
- **Real-World Modeling**: Objects represent real-world entities.
- **Flexibility & Modularity**: Polymorphism allows different objects to use the same method.

## Bazı Önemli OOP Dilleri

1. **Java** – En popüler OOP dillerinden biri, özellikle büyük ölçekli uygulamalar için kullanılır.
2. **C++** – Performans açısından güçlüdür ve oyun geliştirme gibi alanlarda tercih edilir.
3. **Python** – OOP'yi destekler ve basit sözdizimiyle popülerdir.
4. **C#** – Microsoft ekosisteminde yaygın olarak kullanılır.
5. **Ruby** – Nesne yönelimli bir yapıya sahiptir ve web geliştirmede popülerdir.

## 2. Interface vs Abstract Class

### 1. Interface:

- **Tanım:** Bir interface, tamamen soyut bir yapıdır ve yalnızca metodların imzalarını (signature) tanımlar, metodların implementasyonunu içermez.
- **Kullanım Amacı:** Bir sınıfın, belirli bir fonksiyonellik sunmasını sağlamak için kullanılır, ancak bu fonksiyonelliğin nasıl uygulanacağı sınıfın kendisine bırakılır.
- **Özellikler:**
  - Sadece **metod imzaları** ve **özellikler** (sabitler) içerir.
  - Bir sınıf **birden fazla interface**'i implement edebilir.
  - **Multiple inheritance** (çoklu kalıtım) sağlar, yani bir sınıf birden fazla interface'i implement edebilir.
  - Java 8 ve sonrasında **default metodlar** eklenebilir.

**Örnek Interface:**

```
interface Animal {
    void sound();  // abstract method
}
class Dog implements Animal {
    public void sound() {
        System.out.println("Woof!");
    }
}
```

### 2. Abstract Class:

- **Tanım:** Soyut sınıf, hem **soyut metodlar** (implementasyonu olmayan) hem de **somut metodlar** (implementasyonu olan) içerebilen bir sınıftır.
- **Kullanım Amacı:** Temel bir sınıfın ortak davranışları sağlamak ve sınıfların aynı temel özellikleri taşımasını sağlamak için kullanılır.
- **Özellikler:**
  - **Hem soyut (abstract) metodlar** hem de **somut (concrete) metodlar** içerebilir.
  - **Bir sınıf yalnızca bir abstract sınıf**'ı extend edebilir.
  - **State (durum)** tutabilir, yani **field**'lar ve **constructor** içerir.
  - **Kalıtım (inheritance)** yoluyla **ortak işlevsellik** sağlar.

**Örnek Abstract Class:**

```
abstract class Animal {
    abstract void sound();  // abstract method
    void sleep() {          // concrete method
        System.out.println("Sleeping");
    }
}
class Dog extends Animal {
    void sound() {
```

```
        System.out.println("Woof!");
    }
}
```

### Farklar:

| Özellik | Interface | Abstract Class |
|---|---|---|
| **Metodlar** | Sadece abstract metodlar (Java 8 ve sonrası default metodlar da olabilir) | Hem abstract hem concrete metodlar içerir |
| **Kalıtım (Inheritance)** | Bir sınıf birden fazla interface implement edebilir | Bir sınıf yalnızca bir abstract sınıf extend edebilir |
| **Field** | Sadece sabitler (final fields) olabilir | Hem sabitler hem de instance (field) değişkenleri olabilir |
| **Constructor** | Yok | Vardır |
| **Kapsam (Access modifiers)** | Metodlar genellikle **public** olmak zorundadır | Metodlar **private**, **protected**, **public** olabilir |

Her ikisi de sınıfların ortak davranışlarını tanımlamak için kullanılır, ancak kullanım senaryoları farklıdır. Eğer birden fazla sınıfın aynı türde metodları uygulamasını istiyorsanız **interface** kullanmalısınız, ancak **paylaşılan bir davranış veya ortak özellik** sağlamak istiyorsanız **abstract class** daha uygun olacaktır.

## `equals()` ve `hashCode()` Neden Gerekli?

1. **`equals()` Metodu:**
   - **Amacı:** İki nesnenin **mantıksal eşitliğini** karşılaştırmak için kullanılır. Varsayılan olarak, `equals()` metodu **referans eşitliği** kontrol eder (yani, nesnelerin bellek adreslerini karşılaştırır). Ancak çoğu zaman nesnelerin içeriğiyle (alanlarıyla) karşılaştırılmasını istersiniz.
   - **Neden Gerekli:** `equals()` metodunu ezerek, nesneleri **içeriklerine** göre karşılaştırabilirsiniz.
2. **`hashCode()` Metodu:**
   - **Amacı:** Nesnenin **hash kodunu** döndürür. Hash tabanlı koleksiyonlar (`HashMap`, `HashSet`, `Hashtable`) gibi yapılar, nesneleri verimli bir şekilde organize etmek için `hashCode()` metodunu kullanır.
   - **Neden Gerekli:** Eğer `equals()` metodunu ezerseniz, **`hashCode()`** metodunu da ezmelisiniz. Çünkü eşit kabul edilen nesneler aynı hash koduna sahip olmalıdır. Aksi takdirde, hash tabanlı koleksiyonlar düzgün çalışmaz.

## 3. Ne Zaman `equals()` ve `hashCode()` Override?

- **`equals()`** metodunu, nesnelerin **mantıksal eşitliğini** karşılaştırmak istiyorsanız ve nesneleriniz hash tabanlı koleksiyonlarda kullanılıyorsa ezmelisiniz. Yani, iki nesnenin bellekteki adreslerinin değil, **içeriklerinin eşit olup olmadığına bakılır**.

- **`hashCode()`** metodunu, `equals()` metodunu ezdiğinizde ve nesnelerin hash tabanlı koleksiyonlarda kullanılacağını bildiğinizde ezmelisiniz. Hash tabanlı koleksiyonlar (örneğin `HashMap` veya `HashSet`) nesneleri verimli bir şekilde bulabilmek için **hash kodları kullanır**. Yani, bir nesneyi bir koleksiyon içinde hızlıca ararken `hashCode()` önemli bir rol oynar.

## Örnek: `equals()` ve `hashCode()` Ezme

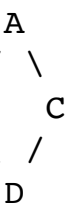
```
import java.util.Objects;

public class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null || getClass() != obj.getClass())
return false;
        Person person = (Person) obj;
        return age == person.age && Objects.equals(name,
person.name);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, age);
    }
}
```

- `equals()` metodu, `name` ve `age` alanlarına göre nesneleri karşılaştırır.
- `hashCode()` metodu, aynı alanlara göre hash kodu üretir.

## 4. Diamond Problem (Elmas Problemi) Java'da

**Elmas problemi** (Diamond Problem), özellikle çoklu kalıtımda ortaya çıkan bir durumdur. Bu problem, bir sınıfın birden fazla sınıftan miras aldığı durumlarda, her iki üst sınıfın da aynı metodu tanımlaması sonucu hangi metodun kullanılacağı konusunda belirsizlik oluşur. Java'da çoklu kalıtım doğrudan desteklenmediği için, bu sorun genellikle **interface** kullanırken karşılaşılır.

## Elmas Problemi Örneği:




- **A** sınıfı, **B** ve **C** sınıflarının ortak üst sınıfıdır.
- **B** ve **C** sınıfları, **D** sınıfı tarafından miras alınır.
- Hem **B** hem de **C**, aynı isimde bir metoda sahipse, **D** sınıfı hangi metodu kullanacağı konusunda belirsizlik yaşayabilir.

## Problem:

Eğer **B** ve **C** sınıflarında aynı isme sahip bir **`m()`** metodu varsa, **D** sınıfı hangi metodun kullanılacağına karar veremeyebilir. Çünkü Java, çoklu kalıtımı desteklemediği için birden fazla sınıfın aynı metodu miras alması durumunda çakışma yaşanır.

## Çözüm:

Java'da **çoklu kalıtım** doğrudan sınıflarla yapılmaz, ancak **interface** kullanılarak bu problem çözülür. Eğer iki interface aynı metodu tanımlıyorsa, Java, bu interface'leri **`default`** metodlarıyla kullanmanıza izin verir ve metodu override ederek çözüm sağlarsınız.

**Java 8 ve sonrasında**, interface'lerde **default metodlar** tanımlanabilir, bu da elmas probleminin çözülmesini sağlar.

## Örnek Çözüm:

```
interface A {
    default void m() {
        System.out.println("A's m()");
    }
}

interface B extends A {
    default void m() {
```

```
        System.out.println("B's m()");
    }
}

interface C extends A {
    default void m() {
        System.out.println("C's m()");
    }
}

class D implements B, C {
    @Override
    public void m() {
        // Elmas problemini çözmek için her iki
interface'teki metodu çağırabilirsiniz.
        B.super.m();
        C.super.m();
    }
}

public class Main {
    public static void main(String[] args) {
        D d = new D();
        d.m();  // B's m() ve C's m() metodları çağrılır.
    }
}
```

## Açıklama:

- **`B.super.m()`** ve **`C.super.m()`** kullanılarak, **D** sınıfı her iki interface'in `m()` metodlarını çağırır. Bu sayede elmas problemini çözmüş oluruz.
- **D** sınıfı, her iki interface'teki default metodları özelleştirebilir ya da her ikisini de çağırabilir.

## Garbage Collector (Çöp Toplayıcı) Neden Gereklidir?

Java gibi dillerde, **Çöp Toplayıcı (GC)**, kullanılmayan bellek alanlarını otomatik olarak serbest bırakmak için gereklidir. Bu, **bellek sızıntıları** ve **bellek yetersizliği hataları** gibi sorunları engellemeye yardımcı olur. Eğer GC olmasaydı, geliştiriciler belleği manuel olarak yönetmek zorunda kalacaklardı, bu da hatalara ve verimsiz bellek kullanımına yol açabilirdi.

## Neden Gereklidir?

1. **Otomatik Bellek Yönetimi**: GC, kullanılmayan nesnelerin belleğini otomatik olarak serbest bırakır, geliştiricilerin işini kolaylaştırır.
2. **Bellek Sızıntılarını Engelleme**: Nesneler doğru şekilde serbest bırakılmazsa, bellek sızıntıları oluşur ve bu da uygulamanın belleğini gereksiz şekilde artırabilir.

## 5. Garbage Collection (GC) Nasıl Çalışır?

Garbage Collection, artık kullanılmayan nesneleri tespit eder ve bu nesnelerin belleğini geri kazandırarak yeniden kullanılabilir hale getirir.

**GC'nin Ana Adımları:**

1. **İşaretleme (Marking)**: GC, önce tüm canlı nesneleri tespit eder. Bu, aktif referansları takip ederek ve erişilebilen nesneleri işaretleyerek yapılır.
2. **Temizleme (Sweeping)**: Canlı nesneler işaretlendikten sonra, GC kullanılmayan nesneleri temizler (yani işaretlenmemiş nesneleri siler).
3. **Sıkıştırma (Optional)**: Temizleme işleminden sonra, bazı GC algoritmaları bellek bölgelerini sıkıştırarak bellek parçalılığına engel olur. Bu işlem opsiyoneldir.

**Java'daki Farklı GC Türleri:**

Java, farklı GC algoritmalarına sahiptir:

- **Seri GC (Serial GC)**: Tek bir iş parçacığı kullanır, küçük uygulamalar için uygundur.
- **Paralel GC (Parallel GC)**: Birden fazla iş parçacığı kullanarak paralel işleme yapar ve çok çekirdekli sistemlerde performansı artırır.
- **CMS (Concurrent Mark Sweep)**: Uygulama ile paralel çalışan bir algoritma olup duraklama sürelerini azaltmayı hedefler.
- **G1 GC**: Büyük uygulamalar için tasarlanmış olup, belleği bölgeler halinde yönetir ve GC işlemlerini daha verimli hale getirir.

## GC Ne Zaman Çalışır?

- **Garbage Collector** otomatik olarak arka planda çalışır ve JVM tarafından bellek düşük olduğunda veya uygulama boşta kaldığında tetiklenir.
- GC'nin ne zaman çalışacağına JVM karar verir, ancak geliştiriciler **`System.gc()`** komutunu kullanarak GC'nin çalışmasını önerebilirler, ancak bu genellikle tavsiye edilmez.

## 6. Java'da `static` Anahtar Kelimesi Kullanımı

Java'da **`static`** anahtar kelimesi, bir sınıfın üyelerinin (değişkenler, metotlar, bloklar) sınıfa ait olduğunu, ancak herhangi bir nesne örneği oluşturulmadan erişilebileceğini belirtir. Bu, sınıf seviyesinde yapılan bir tanımlamadır ve nesne seviyesinden bağımsızdır.

## `static` Anahtar Kelimesi ile Yapılabilecekler:

1. **Static Değişkenler (Class Variables)**:
    - **Static değişkenler**, sınıfın bir örneği oluşturulmadan önce kullanılabilir. Tüm nesneler için ortak olan bir değeri tutarlar.
    - Her nesne, **`static`** değişkene aynı değeri paylaşır.

```
class MyClass {
    static int counter = 0;

    MyClass() {
        counter++;
    }
}

public class Main {
    public static void main(String[] args) {
        new MyClass();
        new MyClass();
        System.out.println(MyClass.counter);  // 2
    }
}
```

**Static Metodlar (Class Methods)**:

**Static metotlar**, nesneye bağlı olmadan sınıf adıyla çağrılabilir. Bu metotlar yalnızca `static` değişkenlerle etkileşime girebilir ve nesne örneği oluşturmak gerekmez. **`static`** metotlar, nesne üyelerine (örneğin, instance değişkenlerine) doğrudan erişemez.

```
class MyClass {
    static void greet() {
        System.out.println("Hello from static method!");
    }
}

public class Main {
    public static void main(String[] args) {
        MyClass.greet();  // Static method çağrısı
    }
}
```

**Static Bloklar (Static Blocks)**:

**Static bloklar**, sınıf ilk kez yüklendiğinde bir defaya mahsus çalıştırılmak üzere kullanılır. Genellikle **static** değişkenlerin ilk değerlerinin ayarlanması için kullanılır.

```
class MyClass {
    static {
        System.out.println("Static block executed");
    }
}

public class Main {
    public static void main(String[] args) {
        new MyClass();  // Static block çalışacak
    }
}
```

**Static İsim Alanları (Static Import)**:

**Static import**, sınıf üyelerini sınıf adını yazmadan doğrudan kullanmanıza olanak tanır.

```
import static java.lang.Math.PI;
import static java.lang.Math.pow;

public class Main {
    public static void main(String[] args) {
        System.out.println(PI);  // Static import
sayesinde Math.PI yazmak gerekmez
        System.out.println(pow(2, 3));  // Static import
sayesinde Math.pow yazmak gerekmez
    }
}
```

## Özet:

- **Static değişkenler** ve **metotlar**, sınıf seviyesinde tanımlanır ve nesne örneği oluşturulmadan kullanılabilir.
- **Static bloklar**, sınıf ilk kez yüklendiğinde çalıştırılır.
- **Static import**, sınıf üyelerinin adını yazmadan doğrudan erişim sağlar.

## 7. Immutability Nedir?

**Immutability**, bir nesnenin durumunun (özelliklerinin) yaratıldıktan sonra değiştirilemez olmasını ifade eder. Yani, bir nesne oluşturulduktan sonra o nesnenin içerdiği veriler üzerinde herhangi bir değişiklik yapılamaz. Bu, o nesnenin **değiştirilemez** olduğu anlamına gelir.

## Nerelerde Kullanılır?

Immutability, genellikle aşağıdaki durumlarda kullanılır:

1. **Thread-Safety (İş Parçacığı Güvenliği)**:
    - **Immutable** nesneler, aynı anda birden fazla iş parçacığı tarafından güvenli bir şekilde kullanılabilir. Çünkü bir nesnenin durumu değiştirilemediğinden, paralel çalıştırma sırasında veri bozulması (race condition) riski yoktur.
2. **Veri Tutarlılığı**:
    - Değiştirilemez nesneler, verilerin tutarlı kalmasını sağlar. Özellikle **data transfer object (DTO)** ya da **value objects** gibi yapılar için ideal bir tercihtir.
3. **Hashing ve Caching**:
    - Immutable nesneler, **hash tablosu** gibi yapılarla (örneğin, `HashMap`) daha verimli çalışabilir. Çünkü nesnelerin durumları değişmediği için, hash kodları sabittir ve nesneler güvenle kullanılabilir.
4. **Fonksiyonel Programlamada**:
    - Fonksiyonel programlamada, değiştirilemez nesneler tercih edilir çünkü yan etkiler (side effects) en aza indirilir ve fonksiyonlar saf (pure) olur.

## Nasıl Kullanılır?

Immutability sağlamak için aşağıdaki adımlar takip edilebilir:

1. **Sadece final alanlar kullanmak**:
    - Nesne oluşturulduğunda, tüm değişkenler **`final`** olarak tanımlanır ve sonradan değiştirilemez.
2. **Yalnızca getter metodları kullanmak**:
    - Değişkenlere dışarıdan doğrudan erişim sağlamazsınız. Sadece getter metodları ile veriyi alabilirsiniz.
3. **Constructor ile değer atamak**:
    - Nesne, yaratılırken tüm alanları **constructor** aracılığıyla alır ve sonra değiştirilemez hale gelir.
4. **Deep Copy kullanmak**:

- Eğer nesne başka nesnelerle ilişkili (örn. koleksiyonlar, diziler) ise, bu nesnelerin de **deep copy** (derin kopya) yapılması gerekmektedir. Böylece iç nesnelerin değiştirilmesi engellenir.

## Örnek:

```
public final class Person {
    private final String name;
    private final int age;

    // Constructor
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Getter Methods
    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }
}
```

Yukarıdaki örnekte:

- **`Person`** sınıfı **`final`** olarak tanımlanmıştır, bu da sınıfın miras alınamayacağı anlamına gelir.
- Tüm değişkenler **`final`** olarak tanımlanmış, yani değerleri bir kez atanabilir ve sonrasında değiştirilemez.
- Sadece getter metodları bulunur, böylece dışarıdan değişkenlere erişilemez.

## Neden Kullanılır?

1. **Güvenlik**:
   - Immutable nesneler, dışarıdan değiştirilemeyeceği için daha güvenlidir. Nesneler üzerinde istenmeyen değişikliklerin önüne geçilir.
2. **Kolay Hata Ayıklama**:
   - Nesnelerin durumu değişmediği için, hataları izlemek daha kolaydır. Bir nesnenin durumunun ne zaman değiştiğini takip etmek gerekmez.
3. **Performans ve Optimizasyon**:
   - Immutable nesneler daha verimli olabilir çünkü bir nesne, bellekteki yerini değiştirmediği için bazı optimizasyonlar yapılabilir (örn. eşitlik karşılaştırmalarında, hashing işlemlerinde).Yan Etkisiz Programlama:

- Immutable nesneler, fonksiyonel programlamanın temel prensiplerinden biri olan yan etkisizliği sağlar.

## 8.Composition ve Aggregation, her ikisi de nesne yönelimli programlamada (OOP) ilişki türlerini tanımlar. Bu iki terim, bir sınıfın başka bir sınıfla olan ilişkisinin gücünü ve bağlılık derecesini belirtir. İşte anlamları ve arasındaki farklar:

### Composition (Bileşim)

- **Tanım**: Composition, bir sınıfın bir başka sınıfın parçası olduğu ve bu parçanın yaşam döngüsünün, ana sınıfın yaşam döngüsüne bağlı olduğu bir ilişki türüdür. Yani, bir nesne, bir başka nesnenin "içine yerleştirilmiş" ve o nesnenin varlığına tamamen bağımlıdır.
- **Bağımlılık**: Composition ilişkisi güçlü bir bağlılık gerektirir. Ana nesne yok olduğunda, içinde barındırdığı nesne de yok olur.
- **Örnek**: Bir **Araba** sınıfı, **Motor** sınıfına sahip olabilir. Araba yok olduğunda, motor da yok olur çünkü motor araba için bir parçadır.

```
class Motor {
    // Motor özellikleri
}

class Araba {
    private Motor motor;

    public Araba() {
        this.motor = new Motor(); // Motor araba ile birlikte
yaratılır
    }
}
```

### Aggregation (Toplama)

- **Tanım**: Aggregation, bir nesnenin başka bir nesneyle ilişkili olduğu, ancak bu iki nesnenin bağımsız olarak var olabileceği bir ilişki türüdür. Yani, bir nesne bir başka nesneyi içerebilir, ancak bu içeren nesne, içerilen nesne yok olduğunda da varlığını sürdürebilir.
- **Bağımlılık**: Aggregation ilişkisi daha zayıf bir bağımlılık gösterir. Ana nesne yok olsa bile, içinde barındırdığı nesne bağımsız olarak varlığını sürdürebilir.
- **Örnek**: Bir **Okul** sınıfı, **Öğrenciler** sınıfını içerebilir. Bir okul yok olsa bile öğrenciler başka okullara kaydolabilirler. Öğrencilerin okula bağımlılığı, sadece okulda eğitim alıyor olmalarıyla ilgilidir.

```
class Ogrenci {
    // Öğrenci özellikleri
}

class Okul {
    private List<Ogrenci> ogrenciler;
```

```
    public Okul() {
        this.ogrenciler = new ArrayList<>();
        // Öğrenciler okulda olabilir, ancak okul yok olsa
bile öğrenciler hala var
    }
}
```

**Farklar**

| Özellik | Composition | Aggregation |
|---|---|---|
| **Bağımlılık Derecesi** | Yüksek. Ana nesne yok olduğunda bağlı nesne de yok olur. | Düşük. Bağlı nesne, ana nesneden bağımsız olarak varlığını sürdürebilir. |
| **Yaşam Döngüsü** | İç içe nesnelerin yaşam döngüsü birbirine bağlıdır. | İç içe nesnelerin yaşam döngüsü birbirinden bağımsızdır. |
| **Örnek** | Araba ve motor (araba yoksa motor da yok olur). | Okul ve öğrenciler (okul yok olsa da öğrenciler başka okullarda var olabilir). |

## Özet:

- **Composition**: İçsel ve güçlü ilişki, nesnelerin yaşam döngüleri birbirine bağlıdır.
- **Aggregation**: Daha zayıf ilişki, nesneler bağımsız olarak var olabilirler.

## 9. Cohesion ve Coupling, yazılım mühendisliğinde, özellikle nesne yönelimli programlamada (OOP) iki önemli kavramdır. Bu kavramlar, sistemlerin tasarım kalitesini ve modüllerin birbirleriyle olan ilişkilerini anlamamıza yardımcı olur.

## Cohesion (Kohezyon)

- **Tanım**: Cohesion, bir sınıfın veya modülün içinde bulunan öğelerin, birbirleriyle ne kadar güçlü bir şekilde ilişkili olduğunu ifade eder. Yüksek cohesion, bir sınıfın veya modülün kendi işini tek başına ve belirli bir sorunu çözme amacına yönelik odaklandığını gösterir.

- **Özellik**: Yüksek cohesion, modülün içinde yer alan bileşenlerin, yalnızca belirli bir sorunu çözmeye yönelik olması anlamına gelir. Böylece, modül daha anlaşılır, bakımı kolay ve yeniden kullanılabilir olur.

- **Örnek**: Bir **BankAccount** sınıfı, yalnızca banka hesabıyla ilgili işlemleri yönetiyorsa (para yatırma, çekme, bakiye sorgulama), bu sınıf yüksek cohesion'a sahip olur.

## Coupling (Bağlantı)

- **Tanım**: Coupling, bir modülün veya sınıfın, başka bir modül veya sınıfla olan bağımlılığını ifade eder. Düşük coupling, modüllerin bağımsız olmasını, yani bir modülün değişmesinin diğerlerini etkilememesini sağlar.

- **Özellik**: Düşük coupling, sistemin daha esnek, bakımı daha kolay ve modüler olmasını sağlar. Yüksek coupling ise modüllerin birbirine sıkı bir şekilde bağlı olması, değişikliklerin bir modülde diğerlerini etkilemesi anlamına gelir.

- **Örnek**: Bir **Order** sınıfı, yalnızca **Customer** sınıfı ile ilişkiliyse ve diğer sınıflara bağımlı değilse, bu durumda düşük coupling vardır.

## Farklar

| Özellik | Cohesion (Kohezyon) | Coupling (Bağlantı) |
|---|---|---|
| **Tanım** | Modül içindeki öğelerin birbiriyle olan ilişkisi. | Modüller arasındaki bağımlılık. |
| **İdeal Durum** | Yüksek cohesion istenir (modüller odaklanmış ve belirli bir amaca hizmet eder). | Düşük coupling istenir (modüller bağımsız ve esnek olmalı). |
| **Etkisi** | Yüksek cohesion, modülün bakımını ve anlaşılabilirliğini artırır. | Düşük coupling, sistemin esnekliğini ve bakımını artırır. |
| **Örnek** | Bir sınıfın yalnızca bir soruna odaklanması (örn. **BankAccount** sınıfı). | Bir sınıfın yalnızca gerekli sınıflara bağımlı olması (örn. **Order** ve **Customer** arasındaki ilişki). |

## Özet:

- **Cohesion**: Bir sınıfın veya modülün içindeki bileşenlerin birbirleriyle ne kadar uyumlu çalıştığını ifade eder. Yüksek cohesion, modülün odaklanmış ve sorumluluğunu net bir şekilde yerine getirdiği anlamına gelir.

- **Coupling**: Modüller arasındaki bağımlılığı ifade eder. Düşük coupling, modüllerin birbirlerinden bağımsız olarak çalışabilmesi anlamına gelir ve yazılımın esnekliğini artırır.

## 10. Heap ve Stack Nedir ve Aralarındaki Farklar?

**Heap** ve **Stack**, bilgisayarın bellek yönetiminde kullanılan iki farklı alanı ifade eder. Her biri belirli amaçlarla kullanılır ve programın çalışması sırasında farklı görevleri yerine getirir.

## Stack (Yığın) Nedir?

**Stack**, son giren ilk çıkar (LIFO - Last In, First Out) prensibiyle çalışan bir bellek yapısıdır. Yani, en son eklenen veri ilk olarak çıkar. Stack, genellikle fonksiyon çağrıları, yerel değişkenler ve fonksiyonların çalışma sürecinde kullanılır.

**Özellikleri:**

1. **LIFO (Last In, First Out)**: Son eklenen öğe ilk çıkar.
2. **Yerel Değişkenler**: Fonksiyonlar arasında veri taşımak için kullanılır. Fonksiyon çalışırken yerel değişkenler bu alanda saklanır.
3. **Küçük ve Hızlı**: Stack üzerinde veri ekleme ve çıkarma işlemleri çok hızlıdır.
4. **Sınırlı Bellek**: Stack sınırlı bir alandır ve çok büyük veriler için uygun değildir.

**Örnek:**

- Fonksiyonların çağrılması ve geri dönüşü stack ile yönetilir.

```
void functionA() {
    int a = 5; // 'a' stack üzerinde saklanır
}

void functionB() {
    functionA(); // functionA çağrıldığında, fonksiyon
stack'e eklenir
}
```

## Heap (Yığın) Nedir?

**Heap**, dinamik bellek tahsisi için kullanılan bir bellek alanıdır. Heap, verilerin yerleştirilmesi için serbest bir alandır ve bu alandaki verilerin yaşam süreleri kontrol edilebilir.

**Özellikleri:**

1. **Dinamik Bellek Yönetimi**: Bellek, gerektiği zaman tahsis edilir ve serbest bırakılır.
2. **Büyük Veri için Uygun**: Stack'ten daha büyük veri yapıları için uygundur.
3. **Daha Yavaş**: Heap, stack'e göre daha yavaş çalışır çünkü bellek tahsisi ve serbest bırakma işlemleri karmaşıktır.
4. **Bellek Sızıntısı Riski**: Eğer bellek doğru şekilde serbest bırakılmazsa, heap'te bellek sızıntıları olabilir.

**Örnek:**

- Heap, dinamik olarak belleği tahsis etmek için kullanılır (örneğin, `new` operatörü ile nesne oluşturulması).

```
class Person {
    String name;
}

Person p = new Person(); // Person nesnesi heap'te
oluşturulur
```

## Heap ve Stack Arasındaki Farklar:

| Özellik | Stack | Heap |
|---|---|---|
| **Veri Yapısı** | LIFO (Last In First Out) | Dinamik Bellek Yönetimi |

| | | |
|---|---|---|
| **Kullanım Alanı** | Yerel değişkenler, fonksiyon çağrıları | Nesne oluşturma ve büyük veri yapıları |
| **Hız** | Çok hızlı (Veri ekleme ve çıkarma hızlı) | Daha yavaş (Daha karmaşık yönetim) |
| **Bellek Alanı** | Sınırlı (Genellikle küçük boyutlu) | Büyük veri için uygundur (Büyük nesneler) |
| **Yaşam Süresi** | Fonksiyon sona erdiğinde yok olur | Bellek manuel serbest bırakılana kadar var |
| **Bellek Yönetimi** | Otomatik (stack bitince veri silinir) | Manuel (garbage collection veya manuel serbest bırakma) |

## Sonuç:

- **Stack**, küçük ve hızlı veri işlemleri için kullanılır, genellikle fonksiyonel ve yerel veri yönetiminde etkilidir.
- **Heap**, daha büyük veri yapıları ve dinamik bellek tahsisi için kullanılır, ancak daha karmaşık ve daha yavaş çalışır.

## 11. Exception Nedir?

**Exception** (İstisna), bir programın çalışma sırasında karşılaştığı beklenmeyen bir durumu ifade eder. Bu durum, programın normal akışını bozar ve bu tür hatalarla başa çıkmak için özel bir mekanizma gereklidir. Java'da, exceptionlar genellikle hataların yönetilmesi ve hata ayıklama için kullanılır.

Bir **exception**, bir hatanın meydana geldiği ve programın bu hatayı uygun bir şekilde ele alabilmesi için bir yol açan olaydır.

## Exception Türleri

Java'da iki ana türde exception vardır:

1. **Checked Exceptions (Kontrol Edilen İstisnalar)**
2. **Unchecked Exceptions (Kontrol Edilmeyen İstisnalar)**

### 1. Checked Exceptions (Kontrol Edilen İstisnalar):

Bu tür istisnalar, programcı tarafından açıkça yakalanması veya yönetilmesi gereken hatalardır. **Compile time** (derleme zamanında) hataları olarak da bilinir. Eğer bir metot, checked exception fırlatıyorsa, o metodun çağrıldığı yerin bu exception'ı yakalamak için ya `try-catch` bloğu kullanması ya da `throws` ifadesiyle bu hatayı üst metoda iletmesi gerekir.

- **Örnekler:**
  - `IOException`
  - `SQLException`
  - `ClassNotFoundException`

**2. Unchecked Exceptions (Kontrol Edilmeyen İstisnalar):**

Bu tür istisnalar, genellikle programın mantığındaki hatalardan kaynaklanır ve **runtime** (çalışma zamanında) hataları olarak bilinir. Unchecked exceptions, programcı tarafından yakalanmak zorunda değildir, ancak genellikle hata ayıklama sırasında bu tür hatalar dikkate alınır. Bunlar genellikle, yazılımla ilgili mantıksal hatalar, yanlış girişler veya hatalı veri kullanımı gibi durumları ifade eder.

- **Örnekler:**
  - `NullPointerException`
  - `ArrayIndexOutOfBoundsException`
  - `ArithmeticException`
  - `IllegalArgumentException`

## Exception Handling (İstisna Yönetimi)

Java'da exception'lar, **try-catch** bloğu ile yönetilir. Ayrıca, istisna fırlatmak için **throw** anahtar kelimesi kullanılır.

```
try {
    // Hata olabilecek kod
    int result = 10 / 0;  // Bu satırda ArithmeticException
fırlatılacak
} catch (ArithmeticException e) {
    // Hata yakalanınca yapılacak işlemler
    System.out.println("Hata: Bölme hatası");
} finally {
    // Bu blok her durumda çalışır
    System.out.println("Finally bloğu çalıştı");
}
```

- **try**: Hata oluşması muhtemel kod bloğu.
- **catch**: Hata yakalandığında yapılacak işlemler.
- **finally**: Hata olsa da olmasa da çalışacak kod.

## 12. Clean Code Nedir?

**Clean Code**, yazılım geliştirme sürecinde kodun okunabilir, anlaşılır ve sürdürülebilir olmasını sağlayan bir yazılım prensibidir. Clean code, aşağıdaki temel ilkelere dayanır:

- **Okunabilirlik**: Kod, başkaları (ve gelecekteki siz) tarafından kolayca anlaşılabilir olmalı.
- **Basitlik**: Gereksiz karmaşıklıklardan kaçınılmalı, en basit çözüm tercih edilmelidir.
- **Tek Sorumluluk Prensibi (Single Responsibility Principle)**: Her sınıf ve fonksiyon yalnızca bir sorumluluğa sahip olmalıdır.
- **Yeniden Kullanılabilirlik**: Kod modüler olmalı ve yeniden kullanılabilir olmalıdır.
- **Yazım ve İsimlendirme**: Anlamlı ve tutarlı isimler kullanılmalıdır.

## 13. Java’da Method Hiding (Metod Gizleme)

**Method hiding**, Java'da bir alt sınıfın, üst sınıfındaki aynı isme ve parametrelere sahip statik bir metodunu gizlemesi durumudur. Statik metodlar, derleme zamanında (compile-time) çözümlenir, bu nedenle hangi metodun çağrılacağı, referans türüne bağlıdır, gerçek nesne türüne değil.

### Nasıl Çalışır?

- **Statik metodlar**, sınıf ile ilişkilidir, nesne ile değil.
- Alt sınıf, üst sınıfın statik metoduyla aynı isme ve parametrelere sahip bir metod tanımlarsa, alt sınıfın metodu üst sınıfın metodunu gizler.
- Metod çağrısı, referans türüne göre belirlenir, nesne türüne göre değil.

### Method Hiding Örneği:

```
class Parent {
    static void display() {
        System.out.println("Parent class static method");
    }
}

class Child extends Parent {
    static void display() {
        System.out.println("Child class static method");
    }
}

public class Main {
    public static void main(String[] args) {
        Parent parent = new Parent();
        Parent child = new Child(); // Referans tipi Parent

        parent.display();  // Parent'ın display() metodunu
çağırır
        child.display();   // Parent'ın display() metodunu
çağırır, Child'ınki değil
    }
}
```

### Açıklama:

- Hem `Parent` hem de `Child` sınıflarında `display()` adında statik metodlar bulunuyor.
- `Parent` türünde bir referans, `Child` türünde bir nesneye işaret etse de, statik metodlar referans türüne bağlı olarak çözülür, bu yüzden `Parent` sınıfının metodu çağrılır.

## 14. Abstraction vs Polymorphism in Java

**Abstraction** ve **Polymorphism**, Java'da nesne yönelimli programlamada (OOP) temel kavramlardır. Her biri, yazılımın farklı yönlerini soyutlamak ve daha esnek hale getirmek için kullanılır. Ancak, bu iki kavramın farklı işlevleri ve kullanımları vardır.

## Abstraction (Soyutlama)

**Abstraction**, karmaşık bir sistemin gereksiz detaylarından kurtulup sadece önemli özelliklerini ve işlevselliğini vurgulamaktır. Soyutlama, bir nesnenin veya sınıfın sadece gerekli bilgileri ortaya koyarak, kullanıcıya veya diğer sınıflara sadece gerekli işlevleri sunmasına olanak tanır.

- **Amaç:** Gereksiz detaylardan kaçınarak, yalnızca önemli özellikleri ve işlevsellikleri sağlamak.
- **Nasıl Yapılır:**
  - Soyut sınıflar (`abstract class`) ve arayüzler (`interface`) kullanılarak yapılır.
  - Bir sınıfın bazı metodlarını soyut (abstract) hale getirerek, bu metodların alt sınıflar tarafından uygulanmasını sağlamak.

**Örnek:**

```
abstract class Animal {
    abstract void sound();  // Soyut metot, alt sınıf
tarafından implement edilecek
}

class Dog extends Animal {
    void sound() {
        System.out.println("Bark");
    }
}
```

## Polymorphism (Çok Biçimlilik)

**Polymorphism**, bir nesnenin birden fazla biçimde davranabilmesidir. Yani, aynı isimdeki bir metod, farklı nesnelerle farklı şekillerde çalışabilir. Polymorphism, hem **method overriding** (metodun üst sınıf tarafından yeniden yazılması) hem de **method overloading** (aynı isimde ama farklı parametrelerle metotların oluşturulması) şeklinde yapılabilir.

- **Amaç:** Aynı işlevin farklı biçimlerde çalışabilmesini sağlamak.
- **Nasıl Yapılır:**
  - **Method Overriding:** Alt sınıfın, üst sınıfın metodunu yeniden tanımlaması.
  - **Method Overloading:** Aynı isme sahip farklı parametrelerle metodlar oluşturulması.

**Örnek (Method Overriding):**

```
class Animal {
    void sound() {
        System.out.println("Animal makes a sound");
    }
}

class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("Bark");
    }
}

class Main {
    public static void main(String[] args) {
        Animal myDog = new Dog();  // Polymorphism: Dog
nesnesi Animal türünde referansla tutuluyor
        myDog.sound();  // Bark
    }
}
```

## Farklar:

| Özellik | Abstraction | Polymorphism |
|---|---|---|
| **Tanım** | Gereksiz detayları gizleyerek sadece önemli özellikleri ortaya koyma. | Aynı metodun farklı şekillerde çalışması. |
| **Amaç** | Sistemin karmaşıklığını gizleyip, sadece önemli özellikleri sunmak. | Aynı işlemi farklı nesnelerle gerçekleştirmek. |
| **Kullanım** | Soyut sınıflar ve arayüzler kullanılır. | Method overriding ve method overloading ile yapılır. |
| **Örnek** | Soyut metodlar ve sınıflar. | Aynı isimde fakat farklı parametreler ya da farklı nesneler ile metod çağrıları. |