Inheritance

Java'da kalıtım (inheritance); extends, constructor zinciri, method overriding, field hiding, upcasting/downcasting ve composition vs inheritance karşılaştırması.

Orta 50 dk

Son güncelleme: 9 Kasım 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/inheritance

---

Inheritance

Buraya kadar gördüğümüz Abstract Class ve Interface derslerinin ikisi de, aslında hiç durup tanımını yapmadığımız bir mekanizmayı sürekli kullandı: `extends`, `super`, `@Override`. Şimdi bu mekanizmanın — **inheritance**'ın (kalıtım) — kendisine, en temelinden başlayarak eğileceğiz. Bir sınıfın başka bir sınıfın alanlarını ve davranışını nasıl devraldığından, constructor zincirinin nasıl işlediğine, `Object` sınıfının her Java nesnesinin neden ortak bir atası olduğuna ve son olarak "ne zaman inheritance, ne zaman composition" sorusuna kadar uçtan uca ilerleyeceğiz.

Konu Nedir?

Inheritance, bir sınıfın (alt sınıf / subclass) başka bir sınıfın (üst sınıf / superclass) alanlarını ve metotlarını devralmasını sağlayan bir mekanizmadır. Gerçek hayattan bir örnek: her `Dog` bir `Animal`'dır — bir köpeğin sahip olduğu her özellik (isim, yaş) ve davranış (uyumak) aslında "hayvan" kavramının bir parçasıdır, köpeğe özgü değildir. Java'da bu ilişkiyi `extends` anahtar kelimesiyle kurarsın:

```
class Animal {
    String name;
}

class Dog extends Animal {
    // Dog, Animal'ın "name" alanını otomatik olarak devralır
}
```

Bu ilişkiye **"is-a" ilişkisi** denir — "Dog is an Animal" (Dog bir Animal'dır) cümlesi anlamlıdır ve doğrudur. Bu, ileride "Inheritance vs Composition" bölümünde göreceğimiz **"has-a"** ilişkisinden (örneğin bir `Car`'ın bir `Engine`'i "vardır", ama `Car` bir `Engine` "değildir") temelden farklıdır.

Neden Var?

Diyelim ki `Student`, `Teacher` ve `Admin` sınıflarını ayrı ayrı yazıyorsun — üçünde de `name`, `email` alanları ve bir `login()` davranışı var. Bu alanları ve metodu üç sınıfa da tek tek kopyalamak hem tekrar (duplication) hem de bir hata düzeltmesi gerektiğinde üç yeri birden güncelleme riski demek. Inheritance, bu ortak kısmı bir `Person` üst sınıfında **bir kez** tanımlayıp, `Student extends Person`, `Teacher extends Person`, `Admin extends Person` diyerek üçüne de bedava kazandırmanı sağlar — her biri yalnızca kendine özgü olanı (örneğin `Student`'ın `enrolledCourses`'u) ekler.

Tarihçe

Inheritance kavramı Java'ya özgü değil — kökleri 1967'de Simula 67 diline, oradan da 1970'lerin Smalltalk'ına kadar uzanır; nesne yönelimli programlamanın (OOP) en temel taşlarından biridir. Java'nın 1996'daki ilk sürümünde alınan kritik bir tasarım kararı, sınıflar için **tek kalıtımı** (single inheritance) desteklemek oldu — bir sınıf yalnızca bir üst sınıftan `extends` edebilir. Bu, o dönem popüler olan C++'ın **çoklu kalıtım** (multiple inheritance) desteğine bilinçli bir tepkiydi: C++'da iki üst sınıftan aynı adlı bir alan ya da metot miras alındığında ortaya çıkan belirsizlik (**Diamond Problem**, "Çoklu Kalıtımın Olmayışı" bölümünde detaylı göreceğiz) C++ kod tabanlarında ciddi karmaşaya yol açıyordu. Java'nın çözümü: sınıflar için tek kalıtım, ama Interface dersinde gördüğümüz gibi interface'ler için sınırsız çoklu "implementasyon" — bu ayrım, kalıtımın gücünü korurken belirsizliği ortadan kaldırdı.

Alt Sınıf Oluşturma ve Temel Terminoloji

Bir alt sınıf, `class` bildirimine `extends ÜstSınıfAdı` eklenerek tanımlanır. Bu ilişkideki iki tarafın birden fazla adı vardır ve hepsi birbirinin eş anlamlısıdır — kaynaklarda hangisiyle karşılaşırsan karşılaş aynı şeyden bahsediliyor demektir:

- Üst sınıf: **superclass**, **parent class**, **base class**
- Alt sınıf: **subclass**, **child class**, **derived class**

```
class Animal {
    String name;

    Animal(String name) {
        this.name = name;
    }

    void eat() {
        System.out.println(name + " is eating");
    }
}

class Dog extends Animal {
    Dog(String name) {
        super(name);
    }

    void bark() {
        System.out.println(name + " says: Woof!");
    }
}

class FirstInheritanceExample {
    public static void main(String[] args) {
        Dog dog = new Dog("Rex");
        dog.eat();  // inherited from Animal
        dog.bark(); // Dog's own method
    }
}
```

Beklenen Çıktı

```
Rex is eating
Rex says: Woof!
```

`Dog`, `Animal`'ı `extends` ederek `name` alanını ve `eat()` metodunu hiç yazmadan devralıyor; kendi özgü davranışı olan `bark()`'ı ise ayrıca ekliyor. Bir `Dog` nesnesi, hem `Animal`'ın hem kendi üyelerine sahip — inheritance, kod **kopyalamak** değil, kod **paylaşmak**tır.

Constructor'lar ve super()

Bir alt sınıfın constructor'ı, ilk satırında (açıkça yazılmasa bile örtük olarak) her zaman üst sınıfın bir constructor'ını çağırır — bu, Java'nın esnek olmadığı kesin bir kuraldır. `super(...)` ifadesi bu çağrıyı **açıkça** yapmanın yoludur ve yalnızca alt sınıf constructor'ının **ilk satırı** olarak kullanılabilir:

```
class Vehicle {
    String brand;

    Vehicle(String brand) {
        this.brand = brand;
        System.out.println("Vehicle constructor running for " + brand);
    }
}

class Car extends Vehicle {
    int doors;

    Car(String brand, int doors) {
        super(brand); // must be the first statement
        this.doors = doors;
        System.out.println("Car constructor running, doors=" + doors);
    }
}

class ConstructorChainExample {
    public static void main(String[] args) {
        Car car = new Car("Toyota", 4);
        // Output order proves Vehicle's constructor runs FIRST:
        // Vehicle constructor running for Toyota
        // Car constructor running, doors=4
    }
}
```

Beklenen Çıktı

```
Vehicle constructor running for Toyota
Car constructor running, doors=4
```

Çıktıdaki sıralamaya dikkat et: `Vehicle`'ın constructor'ı, `Car`'ınkinden **önce** çalışıyor. Bu sıralama tesadüf değil, bir zorunluluk — üst sınıfın durumu (state), alt sınıf kendi ek işini yapmadan önce mutlaka kurulmuş olmalı; aksi halde `Car`'ın constructor'ı, henüz var olmayan bir `Vehicle` durumunun üzerine inşa etmeye çalışırdı.

> Alt sınıf constructor'ında `super(...)`'ı hiç yazmazsan, derleyici üst sınıfın **parametresiz** constructor'ını örtük olarak çağırmaya çalışır. Üst sınıfın parametresiz bir constructor'ı yoksa (bu bölümdeki `Vehicle` gibi), bu örtük çağrı başarısız olur ve `super(...)`'ı doğru parametrelerle açıkça çağırmak zorunlu hale gelir.

Method Overriding

Bir alt sınıf, üst sınıftan miras aldığı bir metodu **aynı imzayla yeniden yazarak** kendi davranışını sağlayabilir — buna **method overriding** denir. `@Override` annotation'ı zorunlu değildir ama şiddetle tavsiye edilir: imzayı yanlış yazdığında (parametre tipini değiştirmek gibi) derleyicinin bunu hemen yakalamasını sağlar.

```
class Shape {
    double area() {
        return 0.0;
    }
}

class Circle extends Shape {
    double radius;

    Circle(double radius) {
        this.radius = radius;
    }

    @Override
    double area() {
        return Math.PI * radius * radius;
    }
}

class Rectangle extends Shape {
    double width;
    double height;

    Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }

    @Override
    double area() {
        return width * height;
    }
}

class MethodOverridingExample {
    static void describe(Shape shape) {
        System.out.println("Area: " + shape.area());
    }

    public static void main(String[] args) {
        describe(new Circle(2));       // Area: 12.566370614359172
        describe(new Rectangle(3, 4)); // Area: 12.0
    }
}
```

`Circle` ve `Rectangle`, `Shape`'ten miras aldıkları `area()`'yı kendi formüllerine göre override ediyor. `describe()` metodundaki döngü ise elindeki her nesneyi yalnızca bir `Shape` olarak görüyor, hangi somut sınıf olduğunu hiç bilmeden `area()` çağırıyor — hangi implementasyonun çalışacağına, çalışma zamanında nesnenin **gerçek** sınıfına bakılarak karar veriliyor. Buna **dynamic dispatch** (dinamik gönderim) denir ve bu konuya "Upcasting" bölümünde tekrar döneceğiz.

super Anahtar Kelimesi

`super` anahtar kelimesinin üç ayrı kullanımı vardır ve hepsini tek bir örnekte görebiliriz: `super(...)` üst sınıfın constructor'ını çağırır ("Constructor'lar ve super()" bölümünde gördük), `super.metod()` üst sınıfın **override edilmiş** bir metodunu açıkça çağırır, `super.alan` ise üst sınıfın bir alanına doğrudan erişir:

```
class Employee {
    protected String name;
    protected double baseSalary;

    Employee(String name, double baseSalary) {
        this.name = name;
        this.baseSalary = baseSalary;
    }

    String describe() {
        return name + " earns " + baseSalary;
    }
}

class Manager extends Employee {
    double bonus;

    Manager(String name, double baseSalary, double bonus) {
        super(name, baseSalary); // super(...) -- calls Employee's constructor
        this.bonus = bonus;
    }

    @Override
    String describe() {
        // super.describe() -- calls Employee's original describe(), builds on top of it
        return super.describe() + " plus a bonus of " + bonus;
    }

    double totalCompensation() {
        // super.baseSalary -- direct access to the inherited field
        return super.baseSalary + bonus;
    }
}

class SuperKeywordExample {
    public static void main(String[] args) {
        Manager manager = new Manager("Ada", 5000, 1000);
        System.out.println(manager.describe());           // Ada earns 5000.0 plus a bonus
of 1000.0
        System.out.println(manager.totalCompensation());  // 6000.0
    }
}
```

`Manager`'ın `describe()`'u, kendi ek bilgisini eklemeden önce `super.describe()`'u çağırarak `Employee`'nin orijinal davranışını **tamamen atlamıyor**, üzerine inşa ediyor — bu, overriding'de çok yaygın bir kalıptır: üst sınıfın davranışını sıfırdan tekrar yazmak yerine, `super.metod()` ile çağırıp sonucuna ekleme yapmak.

> `super.alan` ile `this.alan` aynı isimli bir alanı ayırt etmek için kullanılır — ama bu yalnızca alt sınıf, üst sınıftaki alanla **aynı isimde yeni bir alan tanımlarsa** (field hiding) gerekli olur; bunu bir sonraki bölümde, "Field Hiding vs Method Overriding"'de detaylı işleyeceğiz.

Erişim Belirleyicilerin Inheritance Üzerindeki Etkisi

Bir üst sınıfın alanları ve metotları, erişim belirleyicisine (access modifier) göre alt sınıfa farklı derecede görünür olur:

- **`public`**: Her yerden, dolayısıyla her alt sınıftan erişilebilir.
- **`protected`**: Aynı paketten **ve** farklı paketteki alt sınıflardan erişilebilir — inheritance için özel olarak tasarlanmış bir görünürlük seviyesi.
- **package-private** (hiç modifier yazılmazsa): Yalnızca aynı paketteki sınıflardan erişilebilir; farklı paketteki bir alt sınıf bile göremez.
- **`private`**: Yalnızca tanımlandığı sınıfın kendisinden erişilebilir — alt sınıf, **miras alsa bile** doğrudan erişemez.

```
class Base {
    public String publicField = "public";
    protected String protectedField = "protected";
    String packagePrivateField = "package-private"; // no modifier
    private String privateField = "private";

    private String getPrivateField() {
        return privateField;
    }

    protected String privateFieldViaGetter() {
        // A subclass cannot see privateField directly, but it CAN call this
        // protected getter, which is defined inside Base and has access.
        return getPrivateField();
    }
}

class Derived extends Base {
    void printAccessible() {
        System.out.println(publicField);              // OK -- public
        System.out.println(protectedField);            // OK -- protected, visible to
subclasses
        System.out.println(packagePrivateField);       // OK -- same package (default
package here)
        // System.out.println(privateField);           // compile error: privateField has
private access in Base
        System.out.println(privateFieldViaGetter());   // OK -- goes through Base's own
protected getter
    }
}

class AccessModifiersExample {
    public static void main(String[] args) {
        new Derived().printAccessible();
        // Note: if Derived lived in a DIFFERENT package than Base, packagePrivateField
        // would no longer compile -- package-private members are invisible across
        // package boundaries even to subclasses. protectedField would still work,
        // because `protected` is specifically designed to stay visible to subclasses
        // regardless of package.
    }
}
```

Beklenen Çıktı

```
public
protected
package-private
private
```

`private` bir alan, alt sınıf tarafından teknik olarak **miras alınır** (bellekte alt sınıf nesnesinin bir parçasıdır) ama alt sınıf ona ismiyle doğrudan erişemez — yalnızca üst sınıfın sağladığı `public` / `protected` bir getter üzerinden ulaşabilir. "Miras almak" ile "erişebilmek" aynı şey değildir.

Field Hiding vs Method Overriding

Bu, çoğu kaynakta atlanan ama kafa karıştıran bir ayrım: bir alt sınıf, üst sınıftaki bir **metodu** override ettiğinde çalışma zamanındaki (dynamic dispatch, "Method Overriding" bölümünü hatırla) gerçek sınıfa bakılır — ama üst sınıftaki bir **alanla aynı isimde yeni bir alan tanımladığında** (buna **field hiding** denir), hangi alana erişileceğine çalışma zamanında değil, **derleme zamanındaki değişkenin statik tipine** bakılarak karar verilir:

```
class Animal {
    String label = "Animal";
}

class Dog extends Animal {
    String label = "Dog"; // hides Animal's label, does NOT override it
}

class FieldHidingExample {
    public static void main(String[] args) {
        Dog dog = new Dog();
        Animal animal = dog; // same object, viewed through a different static type

        System.out.println(animal.label); // Animal -- resolved by the STATIC type (Animal)
        System.out.println(dog.label);    // Dog    -- resolved by the STATIC type (Dog)
    }
}
```

Beklenen Çıktı

```
Animal
Dog
```

`animal.label` ve `dog.label` aynı nesneyi işaret etmelerine rağmen **farklı değerler** yazdırıyor — çünkü alanlara erişim polimorfik değildir, yalnızca metot çağrıları öyledir. Bu tutarsızlık tam olarak neden **alanları asla override edilecekmiş gibi tekrar tanımlamaman** gerektiğini gösteriyor; bunu "Yaygın Hatalar" bölümünde tekrar vurgulayacağız.

> Field hiding, method overriding'in aksine gerçek bir polimorfizm değildir — yalnızca alt sınıftaki alanın, üst sınıftaki aynı isimli alanı **gizlemesi**dir. İki alan bellekte ayrı ayrı var olmaya devam eder; `super.alan` ile üst sınıfınkine hâlâ erişebilirsin ("super Anahtar Kelimesi" bölümünü hatırla).

final Sınıf ve final Metod

`final` anahtar kelimesi, inheritance'ı iki farklı seviyede engellemek için kullanılır: `final` **bir sınıf hiç extend edilemez**, `final` **bir metot override edilemez** (ama alt sınıf tarafından normal şekilde miras alınıp kullanılabilir):

```
final class ImmutablePoint {
    final int x;
    final int y;

    ImmutablePoint(int x, int y) {
        this.x = x;
        this.y = y;
    }
}

// class SubPoint extends ImmutablePoint { } // compile error: cannot inherit from final
ImmutablePoint

class Base {
    final void identify() {
        System.out.println("I am a Base, and this method cannot be overridden");
    }

    void greet() {
        System.out.println("Hello from Base");
    }
}

class Derived extends Base {
    // @Override
    // final void identify() { } // compile error: identify() cannot override final method
in Base

    @Override
    void greet() {
        System.out.println("Hello from Derived"); // greet() is NOT final, so overriding it
is fine
    }
}

class FinalClassAndMethodExample {
    public static void main(String[] args) {
        Derived derived = new Derived();
        derived.identify(); // inherited as-is, cannot be overridden
        derived.greet();    // overridden version runs
    }
}
```

Beklenen Çıktı

```
I am a Base, and this method cannot be overridden
Hello from Derived
```

Bu kısıtlamanın en tanıdık örneği, `String` sınıfının kendisi — `String` `final` olduğu için kimse `class MyString extends String` yazıp `String`'in davranışını değiştiremez. Bu bilinçli bir tasarım kararı: `String` değişmezliği (immutability) üzerine kurulu güvenlik ve performans varsayımlarının (örneğin string pool'un güvenle paylaşılabilmesi) hiçbir alt sınıf tarafından bozulamayacağını garanti eder.

> Bir metodu `final` yapmak, o metodun **davranışının sınıf hiyerarşisinin her yerinde aynı kalacağını** garanti etmenin yoludur — Abstract Class dersindeki Template Method Pattern'de gördüğümüz gibi, bir algoritmanın sabit iskeletini tanımlayan metot genellikle `final` yapılır.

Object Sınıfı

Java'da, `extends` yazmasan bile **her sınıf örtük olarak `java.lang.Object`'i extend eder** — bu yüzden `Object`, sınıf hiyerarşisinin köküdür, tüm sınıfların ortak atasıdır. `Object`, her nesnenin miras aldığı birkaç temel metot sağlar; en sık override edilen üçü `toString()`, `equals(Object)` ve `hashCode()`'dur:

```
import java.util.Objects;

class Point {
    int x;
    int y;

    Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    @Override
    public String toString() {
        return "Point(" + x + ", " + y + ")";
    }

    @Override
    public boolean equals(Object other) {
        if (this == other) return true;
        if (!(other instanceof Point)) return false;
        Point p = (Point) other;
        return x == p.x && y == p.y;
    }

    @Override
    public int hashCode() {
        return Objects.hash(x, y);
    }
}

class ObjectClassExample {
    public static void main(String[] args) {
        Point p1 = new Point(1, 2);
        Point p2 = new Point(1, 2);

        System.out.println(p1);                              // Point(1, 2) -- uses
overridden toString()
        System.out.println(p1.equals(p2));                    // true -- same x,y,
overridden equals()
        System.out.println(p1.hashCode() == p2.hashCode());   // true -- required when
equals() is true
    }
}
```

Override edilmemiş `toString()`'in varsayılan çıktısı (`ClassName@hashcode` gibi) neredeyse hiçbir zaman kullanışlı değildir — bu yüzden anlamlı bir metin sunmak istediğin her sınıfta override edilir. `equals()` ve `hashCode()` ise birlikte override edilmelidir: iki nesne `equals()`'a göre eşitse, `hashCode()`'ları da **mutlaka** eşit olmalıdır — aksi halde `HashMap` / `HashSet` gibi hash tabanlı koleksiyonlarda nesneler kayboluyormuş gibi davranır.

> `equals()`'ı override edip `hashCode()`'u override etmeden bırakmak, en sık yapılan Object metodu hatasıdır — bunu "Yaygın Hatalar" bölümünde tekrar ele alacağız.

Upcasting

Bir alt sınıf nesnesini, üst sınıf tipindeki bir değişkene atamaya **upcasting** denir — bu, Java'da her zaman güvenlidir ve örtük olarak (cast yazmadan) gerçekleşir, çünkü her `Dog` zaten bir `Animal`'dır ("Konu Nedir?" bölümündeki is-a ilişkisini hatırla):

```
class Animal {
    void makeSound() {
        System.out.println("Some generic animal sound");
    }
}

class Dog extends Animal {
    @Override
    void makeSound() {
        System.out.println("Woof!");
    }

    void bark() {
        System.out.println("Extra loud woof!");
    }
}

class UpcastingExample {
    public static void main(String[] args) {
        Animal animal = new Dog(); // upcasting -- implicit, always safe

        animal.makeSound(); // Woof! -- runtime type (Dog) decides, not the static type
(Animal)
        // animal.bark();   // compile error: bark() is not visible through an Animal
reference

        Animal[] animals = { new Animal(), new Dog() };
        for (Animal a : animals) {
            a.makeSound(); // each element uses ITS OWN real implementation
        }
    }
}
```

Beklenen Çıktı

```
Woof!
Some generic animal sound
Woof!
```

`Animal animal = new Dog();` satırında, değişkenin **statik tipi** `Animal`'dır ama işaret ettiği nesnenin **gerçek (runtime) tipi** hâlâ `Dog`'dur. `animal` üzerinden yalnızca `Animal`'ın tanımladığı metotları çağırabilirsin (`bark()` görünmez) — ama `makeSound()` gibi override edilmiş bir metot çağrıldığında, çalışan implementasyon her zaman **gerçek tipe** göre belirlenir ("Method Overriding" bölümünde gördüğümüz dynamic dispatch). Bu, polimorfik kod yazmanın temelidir: bir `List<Animal>` tek bir tiple çalışabilir, ama her eleman kendi gerçek davranışını sergiler.

Downcasting ve instanceof

Bir üst sınıf tipindeki değişkeni tekrar alt sınıf tipine döndürmeye **downcasting** denir — upcasting'in aksine bu **her zaman güvenli değildir** ve açık bir cast gerektirir. Değişkenin işaret ettiği nesne gerçekten o alt sınıftan değilse, çalışma zamanında `ClassCastException` fırlatılır. `instanceof` operatörü, cast'i denemeden önce nesnenin gerçek tipini kontrol etmenin güvenli yoludur:

```
class Animal {
}

class Dog extends Animal {
    void bark() {
        System.out.println("Woof!");
    }
}

class Cat extends Animal {
    void meow() {
        System.out.println("Meow!");
    }
}

class DowncastingExample {
    public static void main(String[] args) {
        Animal animal = new Dog(); // upcast first, as in the previous section

        if (animal instanceof Dog dog) {
            // pattern matching -- checks the type AND casts into `dog` in one step
            dog.bark(); // Woof!
        }

        Animal other = new Cat();
        if (other instanceof Dog otherDog) {
            otherDog.bark(); // never runs -- `other` is really a Cat
        } else {
            System.out.println("other is not a Dog"); // this branch runs instead
        }

        // Dog forced = (Dog) other; // compiles, but throws ClassCastException at runtime
    }
}
```

Beklenen Çıktı

```
Woof!
other is not a Dog
```

Modern Java (16 ve sonrası), `instanceof` ile **pattern matching** sayesinde kontrol ve cast'i tek satırda birleştirmene izin veriyor — `if (animal instanceof Dog dog)` hem tipi kontrol ediyor hem `dog` değişkenini otomatik olarak o tipte tanımlıyor, ayrı bir cast satırına gerek kalmıyor.

> `instanceof` kontrolü yapmadan doğrudan `(Dog) animal` gibi bir cast yazmak, nesne gerçekten bir `Dog` değilse programını `ClassCastException` ile çökertir. Downcasting'i yalnızca gerçekten zorunlu olduğun (örneğin bir API'nin sana yalnızca üst sınıf tipini verdiği) durumlarda, ve mutlaka bir `instanceof` kontrolüyle kullan.

Çoklu Kalıtımın Olmayışı

Java, sınıflar için neden yalnızca tek kalıtımı destekler? Cevap, "Tarihçe" bölümünde değindiğimiz **Diamond Problem**'de yatıyor. `B` ve `C`'nin ikisi de `A`'dan miras alsa, `D` de hem `B`'den hem `C`'den miras alsaydı (Java'da bu senaryo derlenmez) ve `A`'daki bir metodu hem `B` hem `C` kendi şekillerinde override etseydi, `D` hangisini miras almalıydı? Bu belirsizlik C++'ta gerçek bir sorundu. Java, sınıflar için bu senaryoyu tamamen yasaklayarak çözdü — ama interface'lerin `default` metotlarında (Interface dersinin "Default Metotlar" bölümünü hatırla) benzer bir çakışma yine de mümkün, ve Java bunu farklı bir kuralla çözüyor:

```
interface Flyer {
    default String move() {
        return "flying";
    }
}

interface Swimmer {
    default String move() {
        return "swimming";
    }
}

// A class implementing both would inherit two conflicting move() defaults.
// Java refuses to guess -- it forces the class to resolve the conflict itself.
class Multi implements Flyer, Swimmer {
    @Override
    public String move() {
        // Flyer.super.move() / Swimmer.super.move() let you reach either default explicitly
        return Flyer.super.move() + " and " + Swimmer.super.move();
    }
}

class DiamondProblemExample {
    public static void main(String[] args) {
        Multi multi = new Multi();
        System.out.println(multi.move()); // flying and swimming
    }
}
```

Beklenen Çıktı

```
flying and swimming
```

`Multi`, hem `Flyer` hem `Swimmer`'ı implement ettiği için ikisinden de `move()` default metodunu miras almaya çalışıyor — ve Java bunu **otomatik çözmüyor**, derleme hatası veriyor. Çözüm, `Multi`'nin `move()`'u **kendisinin** override etmesi, gerekirse `Flyer.super.move()` gibi bir sözdizimiyle her ikisine de erişebilmesi. Yani interface'lerdeki "diamond" durumu tamamen imkansız değil ama **asla sessizce, belirsiz bir şekilde çözülmüyor** — derleyici seni açıkça bir karar vermeye zorluyor. Sınıflar için ise Java bu belirsizliği kökünden kesiyor: bir sınıf asla iki sınıftan birden `extends` edemez.

Inheritance vs Composition

Inheritance güçlü bir araç ama her "ortak kod" ihtiyacının cevabı değil. **Composition** (bileşim), bir sınıfın başka bir sınıfı extend etmek yerine, onu bir **alan olarak içinde tutmasıdır** — "is-a" değil **"has-a"** ilişkisi. Joshua Bloch'un *Effective Java*'daki ünlü tavsiyesi tam olarak bunu söylüyor: **"favor composition over inheritance"** (inheritance yerine composition'ı tercih et):

```
class Engine {
    void start() {
        System.out.println("Engine starting...");
    }

    void stop() {
        System.out.println("Engine stopping...");
    }
}

// Inheritance approach: Car "is-a" Engine -- semantically wrong, and Car
// exposes every one of Engine's public methods whether it makes sense or not.
class CarWithInheritance extends Engine {
}

// Composition approach: Car "has-a" Engine -- Car controls exactly which
// Engine behavior it exposes, and can swap the engine implementation freely.
class CarWithComposition {
    private final Engine engine;

    CarWithComposition(Engine engine) {
        this.engine = engine;
    }

    void drive() {
        engine.start(); // delegation -- Car forwards to its Engine
        System.out.println("Car driving...");
    }
}

class CompositionVsInheritanceExample {
    public static void main(String[] args) {
        CarWithInheritance carA = new CarWithInheritance();
        carA.start(); // works, but semantically odd -- a Car IS an Engine?

        CarWithComposition carB = new CarWithComposition(new Engine());
        carB.drive(); // Engine starting... / Car driving...
    }
}
```

`CarWithInheritance`, `Engine`'i extend ederek onun `start()`'ını miras alıyor — ama bu tuhaf: bir `Car` gerçekten bir `Engine` **değildir**, bir `Engine`'e **sahiptir**. `CarWithComposition` ise `Engine`'i bir alan olarak tutuyor ve `start()`'ını kendi `start()`'ı içinde **çağırıyor** (delegasyon). İkinci yaklaşım, motoru çalışma zamanında değiştirebilmeni (`ElectricEngine` gibi başka bir `Engine` implementasyonu geçirebilirsin), `Car`'ın `Engine`'in **tüm** public metotlarına yanlışlıkla maruz kalmamasını (encapsulation daha güçlü) ve `Engine` değişse bile `Car`'ın davranışının öngörülebilir kalmasını sağlıyor.

> Kendine şunu sor: "Alt sınıf, üst sınıfın **her** public metodunu anlamlı şekilde miras almalı mı?" Cevap hayırsa (bir `Car`'ın bir `Engine`'in her metodunu doğrudan sergilemesi garip kaçar) composition'a yönel. "Gerçek Dünya Örnekleri" bölümünde, JDK'nın da bazı yerlerde inheritance'ı kötüye kullandığını göreceğiz.

Gerçek Dünya Örnekleri

Inheritance, JDK'nın kendi tasarımında hem doğru hem yanlış kullanımıyla öğretici örnekler sunar. `java.io` paketindeki istisna (exception) hiyerarşisi klasik ve doğru bir kullanım: `IOException`, `Exception`'ı extend eder; `FileNotFoundException` ve `EOFException` gibi daha özel istisnalar da `IOException`'ı extend eder. Bu sayede bir `catch (IOException e)` bloğu, hangi özel alt tip fırlatılırsa fırlatılsın hepsini yakalayabilir. Aynı deseni kendi kod tabanımızda da kurabiliriz:

```
class AppException extends Exception {
    AppException(String message) {
        super(message);
    }
}

class ValidationException extends AppException {
    ValidationException(String message) {
        super(message);
    }
}

class NotFoundException extends AppException {
    NotFoundException(String message) {
        super(message);
    }
}

class RealWorldHierarchyExample {
    static void handle(AppException e) {
        // One catch site handles every subtype, exactly like catching java.io.IOException
        // handles FileNotFoundException, EOFException, and every other subtype.
        System.out.println("Handled: " + e.getMessage());
    }

    public static void main(String[] args) {
        handle(new ValidationException("email is invalid"));
        handle(new NotFoundException("user not found"));
    }
}
```

`ValidationException` ve `NotFoundException`, `AppException`'ı extend ediyor — tıpkı JDK'daki `IOException` alt tiplerinin `IOException`'ı extend etmesi gibi. `handle()` metodu tek bir `catch (AppException e)` ile ikisini de yakalayabiliyor, ama `getMessage()` her istisnanın kendi mesajını döndürüyor.

JDK'da inheritance'ın **kötüye** kullanıldığı ünlü bir örnek de var: `java.util.Stack`, `java.util.Vector`'ı extend eder — bu, `Stack`'in `Vector`'ın `add(int, E)` gibi rastgele konuma ekleme yapan tüm metotlarını da (yanlışlıkla) miras almasına yol açar, oysa bir yığın (stack) yalnızca `push` / `pop` ile en üstten çalışmalıdır. Bu, "Inheritance vs Composition" bölümünde tartıştığımız tam olarak yanlış kullanım örneği — modern JDK koleksiyonları (`ArrayDeque` gibi) bu yüzden `Stack` yerine tercih edilir.

Best Practices

- Bir metodu override ederken her zaman `@Override` annotation'ını kullan — imza hatalarını derleme zamanında yakalar (bkz. "Method Overriding").
- Üst sınıftaki bir alanla aynı isimde yeni bir alan **asla** tanımlama — field hiding, çoğu geliştiricinin beklemediği, statik tipe bağlı bir davranıştır (bkz. "Field Hiding vs Method Overriding").
- `equals()`'ı override ediyorsan `hashCode()`'u da mutlaka birlikte override et — aksi halde hash tabanlı koleksiyonlarda nesneler kaybolur (bkz. "Object Sınıfı").
- Downcasting yapmadan önce her zaman `instanceof` ile kontrol et, mümkünse pattern matching sözdizimini kullan (bkz. "Downcasting ve instanceof").
- Bir alt sınıfın üst sınıfın **tüm** public davranışını anlamlı şekilde sergilemediği her durumda, inheritance yerine composition'ı tercih et — "favor composition over inheritance" (bkz. "Inheritance vs Composition").
- Bir sınıfın asla extend edilmesini istemiyorsan (örneğin değişmez bir değer taşıyıcısıysa) onu `final` işaretle — `String`'in kendisi bu prensibin en tanıdık örneği (bkz. "final Sınıf ve final Metod").

Yaygın Hatalar

1. Üst sınıftaki bir alanı, override edilecekmiş gibi aynı isimle yeniden tanımlamak. Bu, method overriding değil field hiding'dir ve statik tipe göre farklı sonuç verir — çoğu zaman istenmeyen, kafa karıştırıcı bir davranıştır (bkz. "Field Hiding vs Method Overriding").

2. `equals()`'ı override edip `hashCode()`'u unutmak. İki eşit nesnenin farklı hash kodu üretmesi, `HashMap` / `HashSet` gibi koleksiyonlarda nesnelerin "kayboluyormuş" gibi davranmasına yol açar (bkz. "Object Sınıfı").

3. `instanceof` kontrolü yapmadan doğrudan downcast yazmak. Nesne gerçekten o tipte değilse, bu çalışma zamanında `ClassCastException` fırlatır (bkz. "Downcasting ve instanceof" bölümündeki uyarı).

4. Yalnızca kod paylaşmak için, "is-a" ilişkisi hiç anlamlı olmadığı halde inheritance kurmak. `Stack extends Vector` gibi JDK'nın kendi hatası bunun klasik örneği — bu durumlarda composition çok daha doğru bir tercihtir (bkz. "Gerçek Dünya Örnekleri").

5. Alt sınıf constructor'ında `super(...)`'ı unutup, üst sınıfın parametresiz bir constructor'ı olduğunu varsaymak. Üst sınıfın yalnızca parametreli bir constructor'ı varsa, `super(...)`'ı doğru parametrelerle açıkça çağırmazsan derleme hatası alırsın (bkz. "Constructor'lar ve super()" bölümündeki tip).

6. Çok derin inheritance zincirleri kurmak (örneğin `A → B → C → D → E`). Her seviye bir sonrakini anlamayı zorlaştırır, bir üst seviyedeki değişikliğin etkisini tüm zinciri okumadan tahmin etmek imkansızlaşır — genel kural, üç seviyeden fazla derinleşmemeye çalışmaktır.

Özet, Cheat Sheet ve Terimler Sözlüğü

Inheritance, Java'nın 1996'daki ilk sürümünden beri sınıflar arasında kod paylaşımını sağlayan temel OOP mekanizmasıdır. Öne çıkan noktalar:

- `extends` ile kurulan ilişki bir **"is-a"** ilişkisidir — alt sınıf, üst sınıfın alanlarını ve metotlarını miras alır
- Alt sınıf constructor'ının ilk satırı her zaman (açık ya da örtük) üst sınıfın constructor'ını çağırır — üst sınıfın state'i, alt sınıfınkinden **önce** kurulur
- Method overriding polimorfiktir (çalışma zamanındaki gerçek tipe bakılır); field hiding **değildir** (değişkenin statik tipine bakılır) — bu ikisi sıkça karıştırılır
- `final` bir sınıf extend edilemez, `final` bir metot override edilemez
- Her sınıf örtük olarak `Object`'i extend eder — `toString()`, `equals()`, `hashCode()` en sık override edilen `Object` metotlarıdır
- Upcasting örtük ve her zaman güvenlidir; downcasting açık bir cast gerektirir ve `instanceof` ile korunmalıdır
- Java, sınıflar için çoklu kalıtımı Diamond Problem yüzünden desteklemez; interface'lerin `default` metotlarında benzer bir çakışma yaşanırsa, derleyici geliştiriciyi açıkça çözmeye zorlar
- "Is-a" ilişkisi gerçekten anlamlı değilse, inheritance yerine composition'ı ("has-a") tercih et

Hızlı referans:

```
// Temel tanım ve terminoloji
class Animal {                    // superclass / parent / base class
    String name;
}

class Dog extends Animal {        // subclass / child / derived class
    Dog(String name) {
        super(name);              // constructor chaining -- always the first line
    }
}

// Method overriding vs field hiding
class Base {
    String label = "Base";
    String describe() { return "Base"; }
}

class Sub extends Base {
    String label = "Sub";               // field hiding -- resolved by STATIC type
    @Override
    String describe() { return "Sub"; } // overriding -- resolved by RUNTIME type
}

// super keyword -- three forms
class Child extends Base {
    Child() {
        super();                  // 1) call superclass constructor
    }
    @Override
    String describe() {
        return super.describe();  // 2) call superclass method
    }
    void show() {
        System.out.println(super.label); // 3) access superclass field
    }
}

// Upcasting / downcasting
Animal a = new Dog("Rex");        // upcasting -- implicit, always safe
if (a instanceof Dog d) {         // downcasting -- explicit, needs instanceof
    // use d as a Dog
}

// final
final class CannotBeExtended { }
class HasFinalMethod {
    final void fixedBehavior() { }
}
```

Terimler Sözlüğü

Inheritance (kalıtım) — Bir sınıfın (alt sınıf) başka bir sınıfın (üst sınıf) alanlarını ve metotlarını devralmasını sağlayan mekanizma.

Superclass / parent class / base class — `extends` edilen, alanları ve davranışı devreden üst sınıf.

Subclass / child class / derived class — `extends` eden, üst sınıfın üyelerini devralan alt sınıf.

"is-a" ilişkisi — Inheritance'ın temsil ettiği ilişki türü ("Dog is an Animal"); "has-a" ilişkisinden (composition) farklıdır.

Method overriding — Bir alt sınıfın, üst sınıftan miras aldığı bir metodu aynı imzayla yeniden tanımlaması; çalışma zamanında gerçek tipe göre çözülür (dynamic dispatch).

Field hiding — Bir alt sınıfın, üst sınıftaki bir alanla aynı isimde yeni bir alan tanımlaması; derleme zamanındaki statik tipe göre çözülür, polimorfik değildir.

`super` — Üst sınıfın constructor'ını (`super(...)`), override edilmiş bir metodunu (`super.metod()`) ya da bir alanını (`super.alan`) çağırmak/erişmek için kullanılan anahtar kelime.

Upcasting — Bir alt sınıf nesnesini üst sınıf tipindeki bir değişkene atamak; örtük ve her zaman güvenlidir.

Downcasting — Bir üst sınıf tipindeki değişkeni alt sınıf tipine geri döndürmek; açık cast gerektirir, `instanceof` ile korunmalıdır, aksi halde `ClassCastException` riski taşır.

Diamond Problem — Çoklu kalıtımda, aynı üst sınıftan/interface'ten miras alınan çakışan üyelerin hangisinin kullanılacağının belirsiz kalması sorunu; Java, sınıflar için çoklu kalıtımı tamamen yasaklayarak bunu önler.

Composition (bileşim) — Bir sınıfın başka bir sınıfı extend etmek yerine onu bir alan olarak içinde tutması ("has-a" ilişkisi); Joshua Bloch'un "favor composition over inheritance" tavsiyesinin konusu.

Ek: Mini Proje — Employee Hiyerarşisi (Manager ve Developer)

Öğrendiklerimizi gerçek bir senaryoya taşıyalım: bir şirketteki `Employee`'lerin ortak alanları (`name` , `baseSalary`) ve davranışı (`describe()`) var, ama maaş hesaplama mantığı role göre değişiyor — `Manager`'ın bir ekip bonusu var, `Developer`'ın ise mesai saatine göre ek ödemesi var:

```
class Employee {
    String name;
    double baseSalary;

    Employee(String name, double baseSalary) {
        this.name = name;
        this.baseSalary = baseSalary;
    }

    double calculateSalary() {
        return baseSalary;
    }
```

```
    String describe() {
        return name + ": base salary " + baseSalary;
    }
}

class Manager extends Employee {
    double teamBonus;

    Manager(String name, double baseSalary, double teamBonus) {
        super(name, baseSalary);
        this.teamBonus = teamBonus;
    }

    @Override
    double calculateSalary() {
        return baseSalary + teamBonus;
    }

    @Override
    String describe() {
        return super.describe() + ", team bonus " + teamBonus + ", total " +
calculateSalary();
    }
}

class Developer extends Employee {
    int overtimeHours;
    static final double OVERTIME_RATE = 50.0;

    Developer(String name, double baseSalary, int overtimeHours) {
        super(name, baseSalary);
        this.overtimeHours = overtimeHours;
    }

    @Override
    double calculateSalary() {
        return baseSalary + (overtimeHours * OVERTIME_RATE);
    }

    @Override
    String describe() {
        return super.describe() + ", overtime " + overtimeHours + "h, total " +
calculateSalary();
    }
}
```

```
class EmployeeHierarchyDemo {
    public static void main(String[] args) {
        Employee[] employees = {
            new Employee("Ada", 4000),
            new Manager("Grace", 6000, 1500),
            new Developer("Linus", 5000, 10)
        };

        for (Employee e : employees) {
            System.out.println(e.describe());
        }
        // Ada: base salary 4000.0
        // Grace: base salary 6000.0, team bonus 1500.0, total 7500.0
        // Linus: base salary 5000.0, overtime 10h, total 5500.0
    }
}
```

`Manager` ve `Developer`, `Employee`'nin `calculateSalary()`'sini kendi formüllerine göre override ediyor, `describe()`'da ise `super.describe()`'u çağırarak ("super Anahtar Kelimesi" bölümünü hatırla) `Employee`'nin ortak metnini tekrar yazmadan üzerine ekliyor. `EmployeeHierarchyDemo`'daki dizi, tüm elemanları `Employee` tipiyle tutuyor ("Upcasting" bölümü) ama her eleman kendi `calculateSalary()` implementasyonunu çalıştırıyor.

> `Employee`'yi burada bilinçli olarak `abstract` yapmadık — Abstract Class dersindeki gibi soyut bir taban değil, somut bir üst sınıf da inheritance için gayet geçerli bir tasarımdır. Bir üst sınıfı `abstract` yapmak, yalnızca doğrudan örneklenmesini engellemek istediğinde gerekli bir karardır.

Ek: Mini Proje — Vehicle Hiyerarşisi (Çok Seviyeli Kalıtım)

Son mini proje, tek seviyeli değil **çok seviyeli** (multi-level) bir inheritance zinciri kuruyor — `Vehicle → MotorVehicle → Car` / `Motorcycle` — ve bu dersin birçok fikrini (constructor zinciri, `Object` metotlarının override edilmesi, upcasting/downcasting) tek bir örnekte birleştiriyor:

```
class Vehicle {
    String brand;

    Vehicle(String brand) {
        this.brand = brand;
    }

    @Override
    public String toString() {
        return brand;
    }
}

class MotorVehicle extends Vehicle {
    int horsePower;
```

```
    MotorVehicle(String brand, int horsePower) {
        super(brand);
        this.horsePower = horsePower;
    }

    void revEngine() {
        System.out.println(brand + " engine revving at " + horsePower + " hp");
    }

    @Override
    public String toString() {
        return super.toString() + " (" + horsePower + " hp)";
    }
}

class Car extends MotorVehicle {
    int doors;

    Car(String brand, int horsePower, int doors) {
        super(brand, horsePower);
        this.doors = doors;
    }

    @Override
    public String toString() {
        return super.toString() + ", " + doors + " doors";
    }
}

class Motorcycle extends MotorVehicle {
    boolean hasSidecar;

    Motorcycle(String brand, int horsePower, boolean hasSidecar) {
        super(brand, horsePower);
        this.hasSidecar = hasSidecar;
    }

    @Override
    public String toString() {
        return super.toString() + (hasSidecar ? " with sidecar" : " no sidecar");
    }
}
```

```
class VehicleHierarchyDemo {
    public static void main(String[] args) {
        Vehicle[] vehicles = {
            new Car("Toyota", 150, 4),
            new Motorcycle("Harley-Davidson", 90, false)
        };

        for (Vehicle v : vehicles) {
            System.out.println(v); // uses each subclass's overridden toString()

            if (v instanceof MotorVehicle motorVehicle) {
                motorVehicle.revEngine(); // safe downcast, all vehicles here are motorized
            }

            if (v instanceof Car car) {
                System.out.println("It's a car with " + car.doors + " doors");
            } else if (v instanceof Motorcycle motorcycle) {
                System.out.println("It's a motorcycle, sidecar: " + motorcycle.hasSidecar);
            }
        }
    }
}
```

`Car`'ın constructor'ı `super(...)` ile `MotorVehicle`'ı çağırıyor, o da kendi `super(...)`'ıyla `Vehicle`'ı çağırıyor — üç seviyeli bir constructor zinciri, her zaman en tepeden (`Vehicle`) aşağı doğru çalışıyor ("Constructor'lar ve super()" bölümünü hatırla). `VehicleHierarchyDemo`, hem upcasting'i (`Vehicle` dizisi) hem `instanceof` ile güvenli downcasting'i ("Downcasting ve instanceof" bölümü) bir arada gösteriyor.

> Üç seviyeli bir zincir (`Vehicle → MotorVehicle → Car`) bu örnekte okunabilir kalıyor, ama "Yaygın Hatalar" bölümünde uyardığımız gibi her ek seviye anlaşılırlığı zorlaştırır — gerçek bir kod tabanında dört-beş seviyeyi geçen bir hiyerarşi kurmadan önce composition'ın daha iyi bir çözüm olup olmadığını mutlaka sorgula.