Polymorphism

Java'da çok biçimlilik (polymorphism); method overloading, overload çözümleme kuralları, covariant return type, polymorphism vs inheritance ve composition ile Strategy Pattern.

Orta 45 dk

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

---

Polymorphism

Inheritance dersinin "Method Overriding" ve "Upcasting" bölümlerinde aslında polymorphism'in en yaygın biçimini — runtime polymorphism'i — zaten görmüştük: `Animal animal = new Dog(); animal.makeSound();` çağrısının hangi implementasyonu çalıştıracağına çalışma zamanında karar veriliyordu. Bu derste o gözlemi formel bir isimle taçlandırıp ("polymorphism"), onun daha az bilinen kardeşine — derleme zamanında çözülen **method overloading**'e — ve polymorphism'in Interface, Abstract Class ve composition ile nasıl bir araya geldiğine odaklanacağız.

Konu Nedir?

Polymorphism (çok biçimlilik), Yunanca "poly" (çok) ve "morph" (biçim) kelimelerinden gelir — aynı arayüzün (aynı metot çağrısının) farklı nesne tiplerinde farklı davranışlar sergileyebilmesi demektir: **"one interface, many implementations."** Java'da bunun iki farklı türü vardır:

- **Compile-time polymorphism (derleme zamanı):** Hangi metodun çalışacağına derleyici, metot çağrısındaki argümanlara bakarak karar verir. Method overloading budur.
- **Runtime polymorphism (çalışma zamanı):** Hangi implementasyonun çalışacağına, nesnenin gerçek tipine bakılarak çalışma zamanında karar verilir. Method overriding budur.

```
animal.makeSound(); // hangi makeSound() çalışır? -- nesnenin gerçek tipine bağlı (runtime)
print("merhaba");   // hangi print() çalışır? -- argümanın tipine bağlı (compile-time)
```

Neden Var?

Gerçek hayattan bir örnek: bir ödeme sistemi düşün — `CreditCard`, `PayPal`, `BankTransfer` gibi birden çok ödeme yöntemi var. Polymorphism olmadan, her ödeme yöntemini ayrı ayrı `if/else` ile kontrol etmen gerekir: `if (type.equals("CARD")) {...} else if (type.equals("PAYPAL")) {...}`. Yeni bir ödeme yöntemi eklendiğinde bu zincirin **her yerini** bulup güncellemen gerekir — kırılgan ve büyüdükçe yönetilemez hale gelen bir tasarım. Polymorphism, her ödeme yönteminin ortak bir arayüzü (`process()`) kendi şekliyle gerçeklemesine izin vererek çağıran kodun tek bir satırla (`payment.process()`) çalışmasını sağlar; kod, hangi ödeme yöntemiyle uğraştığını hiç bilmek zorunda kalmaz. Bu fikri ilk mini projede uçtan uca göreceğiz.

Tarihçe

Polymorphism kavramı da tıpkı inheritance gibi 1967'nin Simula 67'sine kadar uzanır, ama Java'nın miras aldığı iki biçimin kökeni farklı: method overriding (runtime polymorphism) doğrudan Simula/Smalltalk'ın nesne yönelimli mirasından gelirken, method overloading (compile-time polymorphism) çok daha eski, prosedürel dillerde de (Ada gibi) var olan bir fikir — "aynı isimli ama farklı imzalı birden fazla fonksiyon" kavramı. Java, 1996'daki ilk sürümünden beri ikisini birden destekler ve derleyici hangisinin hangi durumda geçerli olduğunu net biçimde ayırır: overloading her zaman derleme zamanında, imzaya bakılarak çözülür; overriding her zaman çalışma zamanında, nesnenin gerçek tipine bakılarak çözülür. Bu netlik, dinamik tipli dillerin (Python, Ruby gibi) benimsediği "duck typing" yaklaşımından bilinçli bir ayrılıktır — Java'da bir nesnenin hangi metotları çağırabileceği, çalışma zamanında değil, derleme zamanında statik tipe göre belirlenir.

Compile-Time vs Runtime Polymorphism

Bu iki polymorphism türünü yan yana görmek, aralarındaki farkı netleştirir:

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

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

class PolymorphismOverviewExample {
    // Compile-time polymorphism: the compiler picks ONE of these based on the argument's
type
    static void print(String value) {
        System.out.println("String: " + value);
    }

    static void print(int value) {
        System.out.println("int: " + value);
    }

    public static void main(String[] args) {
        print("hello"); // resolved at COMPILE time -- calls print(String)
        print(42);      // resolved at COMPILE time -- calls print(int)

        Animal animal = new Dog();
        animal.makeSound(); // resolved at RUNTIME -- calls Dog's makeSound(), based on the
real object
    }
}
```

Beklenen Çıktı

```
String: hello
int: 42
Woof!
```

`print(...)` çağrılarından hangisinin çalışacağı, argümanın tipine bakılarak **derleme anında** belirleniyor — bu yüzden derleyici bu kararı `.class` dosyasına gömüyor, çalışma zamanında hiçbir belirsizlik yok. `animal.makeSound()` çağrısı ise tam tersi: derleyici yalnızca `Animal` tipinin `makeSound()` metoduna sahip olduğunu doğruluyor, hangi **implementasyonun** çalışacağına ise JVM çalışma zamanında, `animal`'ın gerçek tipine bakarak karar veriyor (Inheritance dersinin "Upcasting" bölümünde detaylı gördüğümüz dynamic dispatch).

Method Overloading

Aynı sınıf içinde, aynı isimli ama farklı **parametre listesine** (sayı, tip ya da sıra farkı) sahip birden fazla metot tanımlamaya **method overloading** denir. Dönüş tipi tek başına yeterli değildir — yalnızca dönüş tipi farklı olan iki metot overload sayılmaz, derleme hatası verir:

```
class OverloadingExample {
    static int add(int a, int b) {
        return a + b;
    }

    static double add(double a, double b) {
        return a + b;
    }

    static int add(int a, int b, int c) {
        return a + b + c;
    }

    // static double add(int a, int b) { return a + b; } // compile error: cannot overload
by return type alone

    public static void main(String[] args) {
        System.out.println(add(2, 3));    // 5    -- add(int, int)
        System.out.println(add(2.5, 3.5)); // 6.0  -- add(double, double)
        System.out.println(add(1, 2, 3)); // 6    -- add(int, int, int)
    }
}
```

`add(int, int)`, `add(double, double)` ve `add(int, int, int)` üçü de aynı isme sahip ama derleyici hangisinin çağrılacağına, çağrı sırasında verilen argümanların sayısına ve tipine bakarak karar veriyor. Bu, "Konu Nedir?" bölümünde tanıttığımız compile-time polymorphism'in ta kendisi — hiçbir çalışma zamanı maliyeti yok, karar tamamen derleme anında veriliyor.

> `@Override` yalnızca overriding için kullanılır, overloading için asla yazılmaz — üstüne `@Override` yazılmış bir overload derleme hatası verir, çünkü derleyici orada gerçek bir override beklemektedir. Bu, `@Override`'ın en sık yanlış anlaşılan yanlarından biridir.

Overload Çözümleme Kuralları

Derleyici, birden fazla overload adayı arasında hangisinin "en uygun" olduğuna belirli bir öncelik sırasıyla karar verir: önce **tam eşleşme**, sonra **widening** (küçük tipten büyük tipe örtük dönüşüm, örneğin `int` → `long`), sonra **autoboxing/unboxing** (`int` → `Integer`), en son da **varargs**. Bu sıralamayı bilmemek, hangi overload'un çağrılacağını tahmin edememene yol açar:

```
class OverloadResolutionExample {
    static void process(int value) {
        System.out.println("process(int): " + value);
    }

    static void process(long value) {
        System.out.println("process(long): " + value);
    }

    static void process(Integer value) {
        System.out.println("process(Integer): " + value);
    }

    static void process(int... values) {
        System.out.println("process(int...): " + values.length + " values");
    }

    public static void main(String[] args) {
        short s = 5;
        process(s); // no process(short) -- WIDENS to int -- process(int)

        process(5L); // exact match -- process(long)

        Integer boxed = 5;
        process(boxed); // exact match -- process(Integer)

        process(1, 2, 3); // no 3-arg overload -- falls back to VARARGS -- process(int...)
    }
}
```

Beklenen Çıktı

```
process(int): 5
process(long): 5
process(Integer): 5
process(int...): 3 values
```

`process(s)` çağrısında `short` tipinde bir `process` overload'u yok — derleyici bunu **widening** ile `int`'e genişletip `process(int)`'i çağırıyor. `process(5L)` ve `process(boxed)` ise sırasıyla `long` ve `Integer` parametreleriyle **tam eşleşiyor**, hiçbir dönüşüme gerek kalmıyor. `process(1, 2, 3)` ise üç `int` alan sabit parametreli bir overload olmadığı için, en son çare olarak **varargs**'a (`process(int...)`) düşüyor.

> Varargs (`int...`) her zaman **en son** çare olarak değerlendirilir — tam eşleşen ya da widening/autoboxing ile eşleşen başka bir overload varsa, derleyici varargs'ı hiç denemez bile. Bu yüzden bir metodu hem sabit parametreli hem varargs olarak overload etmek, beklenmedik çağrı sonuçlarına yol açabilir.

Covariant Return Type

Method overriding'in temel mekaniğini (imza eşleşmesi, `@Override`, dynamic dispatch) Inheritance dersinin "Method Overriding" bölümünde işledik — burada tekrarlamayacağız. Ama bir kural o derste hiç geçmedi: **covariant return type**. Bir override edilen metot, üst sınıftaki metodun döndürdüğü tipin **bir alt tipini** döndürebilir — dönüş tipinin birebir aynı olması şart değildir:

```
class Animal {
    Animal reproduce() {
        return new Animal();
    }
}

class Dog extends Animal {
    @Override
    Dog reproduce() { // covariant return type -- Dog is a subtype of Animal, this is legal
        return new Dog();
    }
}

class CovariantReturnTypeExample {
    public static void main(String[] args) {
        Dog dog = new Dog();
        Dog puppy = dog.reproduce(); // no cast needed -- reproduce() already returns Dog
here
        System.out.println(puppy.getClass().getSimpleName()); // Dog
    }
}
```

Beklenen Çıktı

```
Dog
```

`Animal.reproduce()` bir `Animal` döndürüyor, ama `Dog`'un override ettiği `reproduce()` daha spesifik bir tip olan `Dog` döndürüyor — bu, overriding kurallarına aykırı değil, çünkü her `Dog` zaten bir `Animal`'dır (Inheritance dersinin "Konu Nedir?" bölümündeki is-a ilişkisi). Bu sayede `Dog`'u çağıran kod, sonucu tekrar cast etmeden doğrudan `Dog` tipiyle kullanabiliyor.

Polymorphism vs Inheritance

Bu iki kavram sıkça birbirinin yerine kullanılıyor ama aynı şey değiller: **inheritance bir yapı ilişkisidir** (bir sınıfın başka bir sınıftan türetilmiş olması), **polymorphism ise bir çalışma zamanı davranışıdır** (aynı çağrının farklı implementasyonlar çalıştırabilmesi). Inheritance, polymorphism'i **mümkün kılar** ama onu **garanti etmez**:

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

class Cat extends Animal {
    // inherits makeSound() as-is -- no override, so no real polymorphism here
}

class Dog extends Animal {
    @Override
    void makeSound() {
        System.out.println("Woof!"); // overridden -- this IS polymorphism
    }
}

class PolymorphismVsInheritanceExample {
    public static void main(String[] args) {
        Animal cat = new Cat();
        cat.makeSound(); // Some generic animal sound -- inheritance without polymorphism

        Animal dog = new Dog();
        dog.makeSound(); // Woof! -- inheritance WITH polymorphism
    }
}
```

Beklenen Çıktı

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

`Cat`, `Animal`'dan miras alıyor (inheritance var) ama `makeSound()`'u hiç override etmiyor — bu yüzden `cat.makeSound()` her zaman `Animal`'ın davranışını çalıştırıyor, burada gerçek bir polymorphism yok. `Dog` ise hem miras alıyor hem override ediyor — asıl polymorphism burada, override edilmiş metodun çağrılmasında ortaya çıkıyor. Tersi de mümkün: Interface dersinde gördüğümüz gibi, hiç inheritance hiyerarşisi olmadan da (iki alakasız sınıf aynı interface'i implement ederek) polymorphism elde edebilirsin — yani polymorphism, inheritance'a **bağımlı değildir**, yalnızca onun en yaygın kullanıldığı yerlerden biridir.

Interface ve Abstract Class ile Polymorphism

Interface ve Abstract Class derslerinde gördüğümüz gibi (Abstract Class dersindeki "Abstract Class vs Interface" bölümünü hatırla), polymorphism ikisiyle de elde edilebilir — aradaki fark, paylaşılan implementasyonun olup olmadığıdır. JDK'nın `Comparable` interface'i, birbirine hiç akraba olmayan tiplerin bile aynı arayüzle polimorfik davranabileceğinin iyi bir örneği:

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

class Money implements Comparable<Money> {
    int cents;

    Money(int cents) {
        this.cents = cents;
    }

    @Override
    public int compareTo(Money other) {
        return Integer.compare(this.cents, other.cents);
    }

    @Override
    public String toString() {
        return cents + " cents";
    }
}

class ComparableExample {
    public static void main(String[] args) {
        List<Money> wallet = new ArrayList<>();
        wallet.add(new Money(500));
        wallet.add(new Money(100));
        wallet.add(new Money(250));

        Collections.sort(wallet); // sort() only relies on compareTo() -- doesn't know
Money exists
        System.out.println(wallet); // [100 cents, 250 cents, 500 cents]
    }
}
```

Beklenen Çıktı

```
[100 cents, 250 cents, 500 cents]
```

`Money` ve hiçbir ortak üst sınıfı olmayan bambaşka bir tip bile `Comparable`'ı implement etseydi, `Collections.sort(...)` ikisini de aynı şekilde sıralayabilirdi — çünkü `sort`, elemanların **gerçek tipini hiç bilmeden**, yalnızca `compareTo()` sözleşmesine güveniyor. Bu, "Program to an interface" ilkesinin (Interface dersinin "Neden Var?" bölümünü hatırla) somut bir uygulaması.

Composition ile Polymorphism

Inheritance dersinin "Inheritance vs Composition" bölümünde composition'ı somut bir sınıfla (`Engine`) göstermiştik. Composition, bir **interface tipiyle** birleştiğinde çok daha güçlü bir desen ortaya çıkar: **Strategy Pattern** — bir sınıf, davranışını değiştirebileceği bir interface referansı **tutar**, o davranışı kendisi implement etmez:

```
interface TextFormatter {
    String format(String text);
}

class UpperCaseFormatter implements TextFormatter {
    @Override
    public String format(String text) {
        return text.toUpperCase();
    }
}

class ReverseFormatter implements TextFormatter {
    @Override
    public String format(String text) {
        return new StringBuilder(text).reverse().toString();
    }
}

class Document {
    private TextFormatter formatter; // composition -- holds a strategy, doesn't implement
one

    Document(TextFormatter formatter) {
        this.formatter = formatter;
    }

    void setFormatter(TextFormatter formatter) {
        this.formatter = formatter; // behavior can change at runtime
    }

    String render(String text) {
        return formatter.format(text); // delegates -- doesn't know WHICH formatter it's
using
    }
}

class TextFormatterStrategyExample {
    public static void main(String[] args) {
        Document doc = new Document(new UpperCaseFormatter());
        System.out.println(doc.render("hello")); // HELLO

        doc.setFormatter(new ReverseFormatter()); // swap strategy at runtime
        System.out.println(doc.render("hello")); // olleh
    }
}
```

`Document`, hangi `TextFormatter` implementasyonunu kullandığını hiç bilmiyor — yalnızca interface'in `format(String)` sözleşmesine güveniyor. `setFormatter(...)` ile çalışma zamanında **davranışı değiştirebiliyoruz**, ki bu inheritance ile asla mümkün olmazdı (bir nesnenin sınıfı çalışma zamanında değişemez). Bu, composition'ın polymorphism ile birleştiğinde neden bu kadar esnek olduğunu gösteriyor.

instanceof: Ne Zaman Kullanılmalı, Ne Zaman Kod Kokusu Sayılır

`instanceof`'un mekaniğini (pattern matching, `ClassCastException` riski) Inheritance dersinin "Downcasting ve instanceof" bölümünde işledik — burada asıl soru **ne zaman kullanmalısın**. Bir dizi `if (obj instanceof TypeA) {...} else if (obj instanceof TypeB) {...}` zinciri görüyorsan, bu genelde polymorphism'in **kullanılmadığının** bir işaretidir — çünkü doğru tasarımda çağıran kod hiçbir zaman "hangi tip bu?" diye sormaz, doğrudan polimorfik metodu çağırır:

```
class Animal {
    String name;

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

    String describe() {
        return name + " makes some sound";
    }
}

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

    @Override
    String describe() {
        return name + " barks";
    }
}

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

    @Override
    String describe() {
        return name + " meows";
    }
}

class InstanceofDesignExample {
    // Brittle: every new Animal subtype means editing this method again
    static String describeWithInstanceof(Animal animal) {
        if (animal instanceof Dog dog) {
            return dog.name + " barks";
        } else if (animal instanceof Cat cat) {
            return cat.name + " meows";
        } else {
            return animal.name + " makes some sound";
```

```
        }
    }

    // Polymorphic: adding a new subtype never requires touching this method
    static String describeWithPolymorphism(Animal animal) {
        return animal.describe();
    }

    public static void main(String[] args) {
        Animal[] animals = { new Dog("Rex"), new Cat("Whiskers") };

        for (Animal a : animals) {
            System.out.println(describeWithInstanceof(a));
            System.out.println(describeWithPolymorphism(a));
        }
    }
}
```

Beklenen Çıktı

```
Rex barks
Rex barks
Whiskers meows
Whiskers meows
```

`describeWithInstanceof(...)`, her yeni hayvan tipi eklendiğinde büyümesi gereken bir `if /else` zinciri — yeni bir `Bird` eklediğinde bu metodu **bulup güncellemen** gerekir. `describeWithPolymorphism(...)` ise tek satır: her `Animal`, kendi `describe()`'unu nasıl üreteceğini zaten biliyor, çağıran kod hiçbir tip kontrolü yapmıyor. `instanceof`'un **meşru** kullanım alanları da var — örneğin bir koleksiyondaki karışık tiplerden yalnızca belirli birini filtrelemek, ya da bir API sınırında (`equals(Object)` içinde olduğu gibi) tip güvenliğini sağlamak; ama bir davranış farkını ifade etmek için kullanılıyorsa, neredeyse her zaman polymorphism daha iyi bir çözümdür.

> Yeni bir alt tip eklediğinde bir `instanceof` zincirinin **her dalını** güncellemen gerekiyorsa, bu tasarımın polymorphism'e ihtiyacı olduğunun güçlü bir işaretidir — "Yaygın Hatalar" bölümünde bu deseni tekrar ele alacağız.

Koleksiyonlarda Polymorphism

Java koleksiyonlarını neredeyse her zaman interface tipiyle bildirmen (`List<String> list = new ArrayList<>();`) tesadüf değil — bu, "Interface ve Abstract Class ile Polymorphism" bölümünde gördüğümüz "program to an interface" ilkesinin en sık karşılaşacağın uygulaması:

```
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;

class CollectionPolymorphismExample {
    // Declared with the INTERFACE type -- works with any List implementation
    static void printAll(List<String> list) {
        for (String s : list) {
            System.out.println(s);
        }
    }

    public static void main(String[] args) {
        List<String> arrayBacked = new ArrayList<>();
        arrayBacked.add("a");
        arrayBacked.add("b");

        List<String> linkedBacked = new LinkedList<>();
        linkedBacked.add("c");
        linkedBacked.add("d");

        printAll(arrayBacked);  // same method, different List implementation
        printAll(linkedBacked); // same method, different List implementation
    }
}
```

Beklenen Çıktı

```
a
b
c
d
```

`printAll(List<String> list)` metodu, kendisine bir `ArrayList` mi yoksa `LinkedList` mi geldiğini hiç bilmiyor ve bilmesine de gerek yok — yalnızca `List` sözleşmesine güveniyor. Bu sayede implementasyonu değiştirmek (`ArrayList` 'ten `LinkedList` 'e geçmek gibi) çağıran kodun **tek bir satırını bile** etkilemiyor; bu, "Composition ile Polymorphism" bölümünde gördüğümüz esnekliğin koleksiyonlar üzerindeki yansıması.

Gerçek Dünya Örnekleri

Polymorphism, JDK'nın neredeyse tüm temel API'lerinin bel kemiğidir. `java.io` paketindeki `InputStream` / `OutputStream` hiyerarşisi klasik bir örnek: bir dosyadan, bir ağ soketinden ya da bellekteki bir bayt dizisinden okuyan kod, hepsi aynı `InputStream` arayüzü üzerinden **aynı şekilde** çalışır. Aynı deseni küçük bir örnekte kurabiliriz:

```
interface DataSource {
    String readData();
}

class FileDataSource implements DataSource {
    @Override
    public String readData() {
        return "data read from a file";
    }
}

class MemoryDataSource implements DataSource {
    private String data;

    MemoryDataSource(String data) {
        this.data = data;
    }

    @Override
    public String readData() {
        return data;
    }
}

class RealWorldPolymorphismExample {
    // Mirrors how code that reads from an InputStream never cares about the real source
    static void readAll(DataSource source) {
        System.out.println(source.readData());
    }

    public static void main(String[] args) {
        readAll(new FileDataSource());             // data read from a file
        readAll(new MemoryDataSource("in-memory")); // in-memory
    }
}
```

Beklenen Çıktı

```
data read from a file
in-memory
```

`readAll(DataSource source)`, verinin bir dosyadan mı bellekten mi geldiğini hiç bilmiyor — tıpkı gerçek `InputStream` API'sinin `read()` metodunu çağıran kodun, verinin kaynağını bilmesine gerek olmaması gibi. Spring framework'ünde de aynı fikir merkezi bir rol oynar: bir `@Service` sınıfı, ihtiyaç duyduğu bağımlılığı somut bir sınıf yerine bir **interface** tipiyle alır (constructor injection); Spring, çalışma zamanında hangi implementasyonu (`StripePaymentService`, `PaypalPaymentService` gibi) enjekte edeceğine karar verir. Bu, "Composition ile Polymorphism" bölümünde gördüğümüz Strategy deseninin, bir framework tarafından otomatikleştirilmiş hâlinden başka bir şey değil.

Best Practices

- Yeni bir tip eklediğinde büyüyecek bir `if/else` / `switch` zinciri görürsen, bunu bir interface + polymorphism ile değiştirmeyi düşün (bkz. "instanceof: Ne Zaman Kullanılmalı, Ne Zaman Kod Kokusu Sayılır").
- Metot parametrelerini ve koleksiyon değişkenlerini elinden geldiğince **interface tipiyle** bildir (`List` yerine `ArrayList` değil) — bu, implementasyonu değiştirmeni kolaylaştırır (bkz. "Koleksiyonlarda Polymorphism").
- Çalışma zamanında değişebilecek bir davranış tasarlıyorsan composition + interface (Strategy Pattern) kullan, inheritance ile sabitleme (bkz. "Composition ile Polymorphism").
- Bir overload seti tasarlarken, hangi argümanın hangi overload'u tetikleyeceğini derleyicinin bakış açısından test et — belirsiz overload'lar yazma (bkz. "Overload Çözümleme Kuralları").
- Bir override edilen metodun dönüş tipini, gerçekten daha spesifik bir tip döndürüyorsan covariant return type ile daraltmaktan çekinme (bkz. "Covariant Return Type").

Yaygın Hatalar

1. Aynı işi yapan bir dizi `instanceof` kontrolünü, her yeni tip eklendiğinde elle güncellemek. Bu, tam olarak polymorphism'in çözmesi gereken problem (bkz. "instanceof: Ne Zaman Kullanılmalı, Ne Zaman Kod Kokusu Sayılır").

2. Yalnızca dönüş tipini değiştirerek overload yazmaya çalışmak. Java, iki metodu yalnızca dönüş tipine bakarak ayırt edemez — parametre listesi de değişmeli (bkz. "Method Overloading").

3. Overload çözümleme sırasını bilmeden belirsiz overload'lar tasarlamak. Widening, autoboxing ve varargs'ın hangi sırayla denendiğini bilmemek, hangi metodun çağrılacağını yanlış tahmin etmene yol açar (bkz. "Overload Çözümleme Kuralları").

4. Inheritance kurmanın otomatik olarak polymorphism kazandırdığını sanmak. Bir alt sınıf hiçbir metodu override etmiyorsa, orada gerçek bir polymorphism yoktur — yalnızca miras alma vardır (bkz. "Polymorphism vs Inheritance").

5. Koleksiyon değişkenlerini somut implementasyon tipiyle (`ArrayList<String> list = new ArrayList<>();`) bildirmek. Bu, implementasyonu değiştirme esnekliğini baştan yok eder (bkz. "Koleksiyonlarda Polymorphism").

Özet, Cheat Sheet ve Terimler Sözlüğü

Polymorphism, "aynı arayüz, farklı implementasyonlar" fikrinin Java'daki iki biçimidir — derleme zamanında (overloading) ve çalışma zamanında (overriding) çözülür. Öne çıkan noktalar:

- **Compile-time polymorphism (method overloading):** Hangi metodun çalışacağına derleyici, argümanların sayı/tipine bakarak karar verir; dönüş tipi tek başına overload'u ayırt etmez
- Overload çözümleme sırası: **tam eşleşme → widening → autoboxing/unboxing → varargs**
- **Runtime polymorphism (method overriding):** Hangi implementasyonun çalışacağına JVM, nesnenin gerçek tipine bakarak çalışma zamanında karar verir (dynamic dispatch)
- Override edilen bir metot, üst sınıfın döndürdüğü tipin bir alt tipini döndürebilir (**covariant return type**)
- Inheritance bir yapı ilişkisidir, polymorphism bir çalışma zamanı davranışıdır — bir alt sınıf hiçbir şeyi override etmiyorsa, miras var ama polymorphism yok
- Composition, bir interface ile birleştiğinde **Strategy Pattern**'i doğurur — davranış çalışma zamanında değiştirilebilir hale gelir
- Büyüyen bir `instanceof` / `switch` zinciri, genelde eksik bir polymorphism tasarımının işaretidir
- Koleksiyonları interface tipiyle (`List`, `Set`, `Map`) bildirmek, implementasyonu değiştirme esnekliği sağlar
- JDK (`InputStream` / `OutputStream`, `Comparable`) ve Spring (interface tabanlı dependency injection) polymorphism'i yoğun şekilde kullanır

Hızlı referans:

```
// Compile-time polymorphism -- overloading
static void print(String value) { }
static void print(int value) { }
print("a"); // resolved at compile time by argument type
print(1);

// Runtime polymorphism -- overriding
class Animal {
    void makeSound() { }
}
class Dog extends Animal {
    @Override
    void makeSound() { }         // resolved at runtime by the object's real type
}
Animal a = new Dog();
a.makeSound();

// Covariant return type
class Animal2 {
    Animal2 reproduce() { return new Animal2(); }
}
class Dog2 extends Animal2 {
    @Override
    Dog2 reproduce() { return new Dog2(); } // narrower return type -- legal
}

// Strategy pattern -- composition + polymorphism
interface Formatter { String format(String s); }
class Document {
    private Formatter formatter; // held, not extended
    Document(Formatter formatter) { this.formatter = formatter; }
    String render(String s) { return formatter.format(s); }
}

// instanceof chain vs polymorphism
// Bad:
// if (obj instanceof Dog d) { ... } else if (obj instanceof Cat c) { ... }
// Good:
// obj.describe(); // let the object decide
```

Terimler Sözlüğü

Polymorphism (çok biçimlilik) — Aynı arayüzün (metot çağrısının) farklı nesne tiplerinde farklı davranışlar sergileyebilmesi.

Compile-time polymorphism (derleme zamanı çok biçimliliği) — Hangi metodun çalışacağına derleyicinin, argümanların tipine bakarak karar verdiği polymorphism türü; method overloading budur.

Runtime polymorphism (çalışma zamanı çok biçimliliği) — Hangi implementasyonun çalışacağına JVM'in, nesnenin gerçek tipine bakarak çalışma zamanında karar verdiği polymorphism türü; method overriding budur.

Method overloading — Aynı sınıfta, aynı isimli ama farklı parametre listesine sahip birden fazla metot tanımlamak.

Overload çözümleme sırası — Derleyicinin overload adayları arasında seçim yaparken izlediği öncelik: tam eşleşme, widening, autoboxing/unboxing, varargs.

Covariant return type — Bir override edilen metodun, üst sınıftaki metodun döndürdüğü tipin bir alt tipini döndürebilmesi.

Strategy Pattern — Bir sınıfın davranışını, extend etmek yerine bir interface referansı olarak tutup çalışma zamanında değiştirebildiği tasarım deseni; composition + polymorphism'in birleşimi.

Dynamic dispatch — Bir metot çağrısında hangi implementasyonun çalışacağına, nesnenin çalışma zamanındaki gerçek tipine bakılarak karar verilmesi.

Ek: Mini Proje — if/else Zincirinden Polymorphism'e: İndirim Hesaplama

Bu mini projede, "instanceof: Ne Zaman Kullanılmalı, Ne Zaman Kod Kokusu Sayılır" bölümünde tarif ettiğimiz refactor'ı uçtan uca yapıyoruz: müşteri tipine göre indirim hesaplayan kırılgan bir `if/else` zincirini, `DiscountStrategy` interface'i üzerinden polimorfik bir tasarıma çeviriyoruz:

```
class LegacyDiscountCalculator {
    // The brittle version: every new customer type means editing this method
    static double calculateDiscountWithIfElse(String customerType, double amount) {
        if (customerType.equals("VIP")) {
            return amount * 0.20;
        } else if (customerType.equals("REGULAR")) {
            return amount * 0.05;
        } else {
            return 0.0;
        }
    }
}

interface DiscountStrategy {
    double apply(double amount);
}

class PercentageDiscount implements DiscountStrategy {
    private double percentage;

    PercentageDiscount(double percentage) {
        this.percentage = percentage;
    }

    @Override
    public double apply(double amount) {
        return amount * percentage;
    }
}

class FlatDiscount implements DiscountStrategy {
    private double flatAmount;

    FlatDiscount(double flatAmount) {
        this.flatAmount = flatAmount;
    }

    @Override
    public double apply(double amount) {
        return Math.min(flatAmount, amount);
    }
}

class NoDiscount implements DiscountStrategy {
    @Override
    public double apply(double amount) {
        return 0.0;
    }
}
```

```
class DiscountStrategyDemo {
    public static void main(String[] args) {
        System.out.println(LegacyDiscountCalculator.calculateDiscountWithIfElse("VIP",
200)); // 40.0

        DiscountStrategy[] strategies = {
            new PercentageDiscount(0.20), // VIP-style
            new FlatDiscount(10),         // coupon-style
            new NoDiscount()              // no discount
        };

        for (DiscountStrategy strategy : strategies) {
            System.out.println(strategy.apply(200)); // 40.0 / 10.0 / 0.0
        }
    }
}
```

`calculateDiscountWithIfElse(...)` yeni bir müşteri tipi eklendiğinde büyümesi gereken bir zincir; `PercentageDiscount`, `FlatDiscount` ve `NoDiscount` ise her biri `DiscountStrategy`'nin `apply(double)` sözleşmesini kendi formülüyle gerçekliyor. `DiscountStrategyDemo`'daki döngü, hangi strateji ile karşılaştığını hiç bilmeden `apply(...)` çağırıyor — yeni bir indirim türü eklemek, artık yalnızca yeni bir sınıf yazmak demek, var olan hiçbir kodu değiştirmeye gerek yok.

> Bu ilke "Open/Closed Principle" (açık/kapalı ilkesi) olarak bilinir: kod, **yeni davranış eklemeye açık**, ama **var olan kodu değiştirmeye kapalı** olmalı. Polymorphism, bu ilkeyi gerçekleştirmenin en yaygın yoludur.

Ek: Mini Proje — Composition + Strategy: Bildirim Gönderme Sistemi

Son mini proje, "Composition ile Polymorphism" bölümündeki fikri daha gerçekçi bir senaryoya taşıyor: bir `NotificationService`, hangi kanaldan (e-posta, SMS, push) bildirim göndereceğini kendisi bilmiyor — bunu bir `NotificationSender` interface'i üzerinden **composition** ile alıyor:

```
interface NotificationSender {
    void send(String message);
}

class EmailSender implements NotificationSender {
    @Override
    public void send(String message) {
        System.out.println("Email sent: " + message);
    }
}

class SmsSender implements NotificationSender {
    @Override
    public void send(String message) {
        System.out.println("SMS sent: " + message);
    }
}

class PushSender implements NotificationSender {
    @Override
    public void send(String message) {
        System.out.println("Push notification sent: " + message);
    }
}

class NotificationService {
    private NotificationSender sender; // composition -- holds a channel, doesn't become one

    NotificationService(NotificationSender sender) {
        this.sender = sender;
    }

    void setSender(NotificationSender sender) {
        this.sender = sender; // channel can be swapped at runtime
    }

    void notifyUser(String message) {
        sender.send(message); // delegates -- doesn't know which channel it's using
    }
}
```

```
class NotificationSenderDemo {
    public static void main(String[] args) {
        NotificationService service = new NotificationService(new EmailSender());
        service.notifyUser("Your order has shipped"); // Email sent: ...

        service.setSender(new SmsSender()); // swap channel at runtime
        service.notifyUser("Your order has shipped"); // SMS sent: ...

        service.setSender(new PushSender());
        service.notifyUser("Your order has shipped"); // Push notification sent: ...
    }
}
```

Beklenen Çıktı

```
Email sent: Your order has shipped
SMS sent: Your order has shipped
Push notification sent: Your order has shipped
```

`NotificationService`, constructor'ında bir `NotificationSender` **alıyor** (composition), onu extend etmiyor. `EmailSender`, `SmsSender` ve `PushSender`'ın her biri aynı `send(String)` sözleşmesini kendi şekliyle gerçekliyor. `NotificationSenderDemo`'daki gibi, aynı `NotificationService` nesnesinin `setSender(...)` ile **çalışma zamanında** farklı bir kanala geçebilmesi, inheritance ile asla elde edemeyeceğin bir esneklik — bu, "Composition ile Polymorphism" bölümünde vurguladığımız temel avantaj.

> `NotificationService`'i `EmailSender`'ı extend edecek şekilde tasarlasaydık ("is-a" ilişkisi hiç anlamlı olmadığı halde), her yeni kanal için ya yeni bir `NotificationService` alt sınıfı yazman ya da tek bir sınıfı tüm kanalları destekleyecek şekilde şişirmen gerekirdi — tam olarak Inheritance dersinin "Inheritance vs Composition" bölümünde uyardığımız tuzak.