Abstract Class

Java'da ortak durum ve davranışı paylaşan soyut sınıflar; constructor, Template Method Pattern ve abstract class vs interface karşılaştırması.

Orta 50 dk

Oluşturulma tarihi: 2 Ekim 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/abstract-class

---

Abstract Class

Java'da **abstract class** (soyut sınıf), doğrudan bir örneği oluşturulamayan, ama alt sınıfların (subclass) miras alıp tamamlayacağı ortak durum (state) ve davranışı bir arada tutan bir sınıf türüdür. Interface'in "yalnızca sözleşme" felsefesinin aksine, bir abstract class hem "şunu yapmalısın" (soyut metotlar) hem de "bunu senin için zaten yaptım" (somut metotlar, alanlar, constructor) diyebilir. Bu derste, en temel `abstract` anahtar kelimesinden başlayıp constructor'ların rolüne, Template Method deseninin gerçek dünya kullanımına ve Interface dersinde gördüğümüz `Interface vs Abstract Class` ayrımının tam karşı tarafına kadar uçtan uca ilerleyeceğiz.

Abstract Class Nedir?

Bir abstract class, `class` bildirimine `abstract` anahtar kelimesi eklenerek tanımlanır. Bu tek kelime iki şeyi birden garanti eder: bu sınıftan **asla doğrudan bir nesne oluşturulamaz** (`new AbstractClass()` derlenmez) ve bu sınıf, gövdesiz — yani **soyut** — metotlar içerebilir. Bir abstract class'ı somut (concrete) hale getirmenin tek yolu, onu `extends` eden ve tüm soyut metotlarını gerçekleyen bir alt sınıf yazmaktır.

Neden Var?

Gerçek bir problemle başlayalım: `Dog`, `Cat` ve `Bird` sınıflarını ayrı ayrı yazdığını düşün — üçünde de bir `name` alanı, bir `sleep()` davranışı (hepsi aynı şekilde uyur) ve bir `makeSound()` davranışı (her biri farklı ses çıkarır) var. `name` alanını ve `sleep()`'i üç sınıfa da tek tek kopyalamak hem tekrar (duplication) hem de bir hata düzeltmesi gerektiğinde üç yeri birden güncelleme riski demek. Ama `makeSound()`'u ortak bir yerde **tam olarak** tanımlamak da mümkün değil — her hayvan farklı ses çıkarıyor.

Abstract class tam olarak bu ara noktayı çözüyor: ortak olan `name` ve `sleep()`'i **bir kez**, üst sınıfta yazıyorsun; her alt sınıfın kendine özgü olması gereken `makeSound()`'u ise yalnızca bir

imza olarak bırakıyorsun, gövdesini her alt sınıfa bırakıyorsun. Sonuç: kod tekrarı yok, ama yine de her hayvan kendi sesini çıkarabiliyor.

Tarihçe

Abstract class, interface gibi Java'nın ilk gününden (JDK 1.0, 1996) beri var — nesne yönelimli programlamanın (OOP) temel taşlarından biri olarak, dilin başından beri kod paylaşımı ile soyutlamayı bir arada sunuyor. Yıllarca abstract class ile interface arasındaki çizgi çok netti: interface hiçbir gövde barındıramaz, abstract class hem soyut hem somut metot barındırabilirdi. Java 8'in (2014) interface'lere `default` metot eklemesi (Interface dersindeki "Tarihçe" bölümünde işledik) bu çizgiyi belirgin şekilde bulanıklaştırdı — artık bir interface de gövdeli bir metot sağlayabiliyordu. Ama "Abstract Class vs Interface" bölümünde göreceğimiz gibi, constructor'a ve instance alanlarına sahip olma gibi bazı farklar hâlâ kalıcı ve abstract class'ı vazgeçilmez kılıyor.

İlk Abstract Class'ını Yazmak

Bir abstract class, `class` yerine `abstract class` ile tanımlanır; içindeki gövdesiz bir metot de aynı şekilde `abstract` ile işaretlenir. Onu genişleten (`extends`) somut bir alt sınıf, tüm soyut metotları `@Override` ile gerçeklemek zorundadır:

```
abstract class Animal {
    protected String name;

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

    abstract void makeSound();

    void sleep() {
        System.out.println(name + " is sleeping... Zzz");
    }
}

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

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

class FirstAbstractClassDemo {
    public static void main(String[] args) {
        // Animal animal = new Animal("Generic"); // compile error: Animal is abstract;
cannot be instantiated

        Dog dog = new Dog("Rex");
        dog.makeSound(); // Rex says: Woof!
        dog.sleep();     // Rex is sleeping... Zzz -- inherited straight from Animal
    }
}
```

`Animal`'ı doğrudan `new Animal("Generic")` ile örneklemeye çalışsan derleme hatası alırsın — yorum satırındaki gibi. `Dog` ise `Animal`'ı genişletip `makeSound()`'u gerçekleştiği için sorunsuz örneklenebiliyor; üstelik `sleep()`'i hiç yazmadan, doğrudan `Animal`'dan miras alarak kullanabiliyor.

> Bir abstract class'ın en az bir soyut metoda sahip olması **şart değildir** — hiç soyut metodu olmayan bir sınıf bile, yalnızca `abstract` anahtar kelimesiyle işaretlenerek doğrudan örneklenmesi engellenebilir; bunu bir sonraki bölümde göreceğiz.

Abstract Class vs Concrete Class

Buradaki en sık karışan nokta şu: bir sınıfın doğrudan örneklenip örneklenemeyeceğine karar veren şey, sınıfın soyut metoda **sahip olup olmadığı değil**, sınıfın kendisinin `abstract` olarak **işaretlenip işaretlenmediğidir**:

```
// Shape has ZERO abstract methods -- every method already has a body -- but
// the class itself is still marked `abstract`, so it still cannot be
// instantiated directly. It's the `abstract` keyword on the class, not the
// presence of an abstract method, that blocks `new`.
abstract class Shape {
    double area() {
        return 0;
    }
}

class Square extends Shape {
    private final double side;

    Square(double side) {
        this.side = side;
    }

    @Override
    double area() {
        return side * side;
    }
}

class AbstractVsConcreteExample {
    public static void main(String[] args) {
        // Shape shape = new Shape(); // compile error: Shape is abstract; cannot be
instantiated

        Shape shape = new Square(4);
        System.out.println(shape.area()); // 16.0
    }
}
```

`Shape`'in tek bir soyut metodu bile yok — `area()` tamamen gövdeli. Buna rağmen `Shape`, sınıf bildirimindeki `abstract` anahtar kelimesi yüzünden doğrudan örneklenemiyor. Bu, bir API tasarımcısının, "bu sınıf yalnızca bir taban sınıf (base class) olarak kullanılsın, kimse doğrudan örneğini oluşturmasın" demesinin bilinçli bir yolu — soyut metot olmasa bile.

Abstract Metotlar

Bir abstract class'taki soyut metotlar, tıpkı interface'teki gibi (Interface dersindeki "Soyut Metotlar" bölümünü hatırla) gövdesizdir — ama önemli bir fark var: **bir abstract class'ı genişleten başka bir abstract class, miras aldığı soyut metotları hemen gerçeklemek zorunda değildir**:

```
abstract class Vehicle {
    abstract void start();
}

// MotorVehicle does NOT implement start() -- and that's perfectly legal,
// because MotorVehicle itself is still declared `abstract`. Only a CONCRETE
// (non-abstract) subclass is forced to implement every inherited abstract
// method; an abstract subclass is free to defer some (or all) of them
// further down the hierarchy.
abstract class MotorVehicle extends Vehicle {
    abstract void refuel();
}

class Car extends MotorVehicle {
    @Override
    void start() {
        System.out.println("Car engine started");
    }

    @Override
    void refuel() {
        System.out.println("Car refueled with gasoline");
    }
}

class AbstractMethodExample {
    public static void main(String[] args) {
        Car car = new Car();
        car.start();
        car.refuel();
    }
}
```

`MotorVehicle`, `Vehicle`'dan miras aldığı `start()`'ı hiç implement etmiyor — ve bu sorunsuz derleniyor, çünkü `MotorVehicle`'ın kendisi de `abstract`. Soyut bir metodu gerçeklemek zorunda olan yalnızca **somut (concrete)** bir sınıf — yani `Car` — ki o da hem `start()`'ı hem kendi eklediği `refuel()`'i gerçeklemek zorunda.

> Bir alt sınıf, üst sınıftan miras aldığı bir soyut metodu gerçeklemezse ve kendisi de `abstract` olarak işaretlenmezse, derleme hatası alırsın. "Bu sınıfı şimdilik somut yapmıyorum" demenin tek yolu, sınıfı da `abstract` işaretlemektir — derleyici üçüncü bir seçeneğe izin vermez.

Concrete (Somut) Metotlar

Bir abstract class, soyut metotların yanı sıra tamamen gövdeli — yani **concrete** (somut) — metotlar da barındırabilir; bu, çoğu kişinin abstract class hakkında bilmediği ama en değerli özelliklerinden biridir:

```
abstract class Animal {
    protected String name;

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

    abstract void makeSound();

    // A CONCRETE method -- it has a body right here in the abstract class.
    // Every subclass gets this behavior for free and never has to write it
    // again, unlike makeSound(), which each subclass MUST provide itself.
    void sleep() {
        System.out.println(name + " is sleeping... Zzz");
    }
}

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

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

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

    @Override
    void makeSound() {
        System.out.println(name + " says: Meow!");
    }
}

class ConcreteMethodExample {
    public static void main(String[] args) {
        Dog dog = new Dog("Rex");
        Cat cat = new Cat("Whiskers");

        dog.makeSound(); // Rex says: Woof!
        cat.makeSound();  // Whiskers says: Meow!

        // Neither Dog nor Cat wrote a single line of sleep() -- both reuse
        // the exact same inherited implementation.
        dog.sleep(); // Rex is sleeping... Zzz
        cat.sleep(); // Whiskers is sleeping... Zzz
    }
}
```

`sleep()`, `Animal` içinde **tam bir gövdeyle** tanımlı — `Dog` da `Cat` da bunu tek satır bile yazmadan, doğrudan miras alarak kullanıyor. `makeSound()` ise soyut kaldığı için her ikisi de

kendi implementasyonunu sağlamak zorunda. Bu ikilik — bazı metotları paylaş, bazı metotları alt sınıfa bırak — abstract class'ın "Neden Var?" bölümünde tarif ettiğimiz temel değerinin ta kendisi.

Alanlar (Fields)

Bir abstract class, tıpkı normal bir sınıf gibi **instance alanları** tutabilir — bu, "yalnızca sabit tanımlayabilen" interface'lerden (Interface dersindeki "Constant Alanlar" bölümünü hatırla) en temel farklarından biri. Alt sınıflar bu alanlara genelde `protected` erişim belirleyicisiyle doğrudan ulaşır:

```
abstract class Employee {
    // `protected` -- visible to subclasses (even in other packages), but not
    // part of the public API. This is the access level abstract classes
    // typically use for state that subclasses are expected to read directly.
    protected String name;
    protected double baseSalary;

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

    abstract double calculateSalary();
}

class Manager extends Employee {
    private final double bonus;

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

    @Override
    double calculateSalary() {
        // Reads baseSalary directly -- it's inherited state, not something
        // Manager has to ask for through a getter.
        return baseSalary + bonus;
    }
}

class FieldsExample {
    public static void main(String[] args) {
        Manager manager = new Manager("Ada", 5000, 1500);
        System.out.println(manager.calculateSalary()); // 6500.0
    }
}
```

`Manager`, `calculateSalary()` içinde `baseSalary`'ye bir getter çağırmadan doğrudan erişiyor — çünkü `baseSalary`, `Employee`'den miras alınan gerçek bir **instance alanı**, salt bir sabit değil. Her `Employee` alt sınıfı örneği (her `Manager`, ileride yazılacak her başka rol) kendi `name`/

`baseSalary` **kopyasına** sahip olur — interface sabitlerinin aksine, burada paylaşılan tek bir değer yok, her nesnenin kendi durumu var.

> `protected` alanlar, alt sınıflara kolaylık sağlasa da encapsulation'ı (kapsülleme) zayıflatır — bir alt sınıf, üst sınıfın iç durumunu getter/setter'dan geçmeden doğrudan değiştirebilir. "Best Practices" bölümünde, ne zaman `protected` alan yerine `private` alan + `protected` getter/setter tercih etmen gerektiğine değineceğiz.

Constructor'lar

Bir abstract class'ın constructor'ı **olabilir** — ve olması çoğu zaman zorunludur, çünkü tuttuğu instance alanlarını başlatması gerekir. Tek fark, bu constructor'ın asla doğrudan `new` ile çağrılamaması; yalnızca bir alt sınıfın constructor'ı içindeki `super(...)` çağrısıyla dolaylı olarak çalışır:

```
abstract class Account {
    protected String owner;
    protected double balance;

    // An abstract class CAN have a constructor -- you just can never call it
    // with `new` directly. It only ever runs as part of building a concrete
    // subclass instance, to initialize the shared state the subclass inherits.
    Account(String owner, double balance) {
        this.owner = owner;
        this.balance = balance;
        System.out.println("Account constructor running for " + owner);
    }

    abstract double interestRate();
}

class SavingsAccount extends Account {
    SavingsAccount(String owner, double balance) {
        super(owner, balance); // must be the first statement in the subclass constructor
        System.out.println("SavingsAccount constructor running");
    }

    @Override
    double interestRate() {
        return 0.05;
    }
}

class ConstructorExample {
    public static void main(String[] args) {
        SavingsAccount account = new SavingsAccount("Ada", 1000);
        // Output order proves Account's constructor runs FIRST:
        // Account constructor running for Ada
        // SavingsAccount constructor running
        System.out.println(account.interestRate()); // 0.05
    }
}
```

`SavingsAccount`'un constructor'ındaki ilk satır `super(owner, balance)` — bu, Java'da **zorunlu bir kural**: bir alt sınıf constructor'ının ilk satırı (açıkça yazılmasa bile, örtük olarak) her zaman üst sınıfın bir constructor'ını çağırır. Çıktıdaki sıralamaya dikkat et: `Account`'un constructor'ı, `SavingsAccount`'unkinden **önce** çalışıyor — üst sınıfın durumu, alt sınıf kendi ek işini yapmadan önce mutlaka kurulmuş olmalı.

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

Kalıtım

Bir sınıf, `extends` ile yalnızca **bir** sınıftan (abstract olsun ya da olmasın) miras alabilir — bu, Interface dersinin "Neden Var?" bölümünde değindiğimiz Java'nın tek kalıtım kısıtlamasının ta kendisi. `Dog extends Animal` yazdığında, `Dog`'un başka bir sınıftan miras alma ihtimali tamamen kapanır — ama istersen `Dog`, dilediği kadar interface'i `implements` edebilir ("Abstract Class'ın Bir Interface'i Implement Etmesi" bölümünde tam olarak bunu yapacağız). Abstract class hiyerarşileri de, "Abstract Metotlar" bölümünde `Vehicle` → `MotorVehicle` → `Car` örneğinde gördüğümüz gibi, birden fazla seviyeye yayılabilir — her seviye, altındaki seviyeye bir kısım soyutlamayı devretmeyi seçebilir.

Abstract Metotları Override Etmek ve Polimorfizm

Bir soyut metodu gerçeklerken kullandığın `@Override` annotation'ı zorunlu değildir, ama şiddetle tavsiye edilir — imzayı yanlış yazdığında (örneğin parametre tipini değiştirdiğinde) derleyicinin bunu hemen yakalamasını sağlar. Birden fazla alt sınıf aynı soyut metodu farklı şekillerde override ettiğinde, ortaya interface'lerde de gördüğümüz aynı **polimorfizm** çıkar:

```
abstract class Animal {
    protected String name;

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

    abstract void makeSound();
}

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

    @Override
    void makeSound() {
        System.out.println(name + " says: Woof!");
```

```
    }
}

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

    @Override
    void makeSound() {
        System.out.println(name + " says: Meow!");
    }
}

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

    @Override
    void makeSound() {
        System.out.println(name + " says: Tweet!");
    }
}

class OverridingAndPolymorphismExample {
    public static void main(String[] args) {
        // Polymorphism through the abstract type reference: the loop below
        // never needs to know whether each element is a Dog, a Cat, or a
        // Bird -- it only knows it's holding an Animal.
        Animal[] animals = { new Dog("Rex"), new Cat("Whiskers"), new Bird("Tweety") };

        for (Animal animal : animals) {
            animal.makeSound();
        }
        // Rex says: Woof!
        // Whiskers says: Meow!
        // Tweety says: Tweet!
    }
}
```

`Dog`, `Cat` ve `Bird`, `makeSound()`'u üçü de farklı şekilde override ediyor — `OverridingAndPolymorphismExample`'daki döngü ise elindeki her nesneyi yalnızca bir `Animal` olarak görüyor, hangi somut sınıf olduğunu hiç bilmeden `makeSound()` çağırıyor. Hangi implementasyonun çalışacağına, tıpkı Interface dersindeki `Shape` örneğinde olduğu gibi, çalışma zamanında nesnenin gerçek sınıfına bakılarak karar verilir (dinamik gönderim / dynamic dispatch).

Abstract Metotlarda Modifier ve Erişim Kuralları

Bir abstract metodun erişim belirleyicisi `public` ya da `protected` olabilir, ama **asla** `private` olamaz — ve `abstract`, birkaç anahtar kelimeyle bir arada asla kullanılamaz:

```
abstract class Shape {
    public abstract double area();          // legal: public abstract

    protected abstract double perimeter();  // legal: protected abstract

    // private abstract double helper();
    // ILLEGAL: a private method can never be overridden, which directly
    // contradicts what `abstract` requires (a subclass MUST override it).

    // static abstract double unit();
    // ILLEGAL: static methods aren't resolved polymorphically (no dynamic
    // dispatch), so they can't be "overridden" the way abstract requires.

    // abstract final double diagonal();
    // ILLEGAL: `final` forbids overriding, `abstract` requires it --
    // a direct contradiction.
}

// abstract final class Circle { }
// ILLEGAL for the exact same reason: a class can't be simultaneously
// "must be extended to be useful" (abstract) and "can never be extended"
// (final) at the same time.

class Square extends Shape {
    private final double side;

    Square(double side) {
        this.side = side;
    }

    @Override
    public double area() {
        return side * side;
    }

    @Override
    protected double perimeter() {
        return 4 * side;
    }
}

class ModifierRulesExample {
    public static void main(String[] args) {
        Shape shape = new Square(3);
        System.out.println(shape.area()); // 9.0

        // shape.perimeter();
        // Not accessible here: perimeter() is `protected`, and this call
        // site is outside Shape's package and outside any subclass of Shape.
    }
}
```

Buradaki her yasak, aslında `abstract`'ın kendi tanımıyla doğrudan çelişiyor: `abstract` bir metodun **override edilmesini zorunlu kılar**; `private` bir metot zaten hiçbir alt sınıfa görünmez (override edilemez), `static` bir metot polimorfik çözümlenmez (override değil,

gizleme/hiding olur), `final` bir metot ise override edilmesi açıkça yasaklanmış bir metottur. Üçü de "override edilebilirlik" ile taban tabana zıt olduğu için, `abstract` ile yan yana yazıldıklarında derleyici anında hata verir — aynı çelişki, sınıf seviyesinde `abstract final class` için de geçerlidir.

> `private abstract` ya da `static abstract` gibi bir kombinasyon yazmaya çalıştığında alacağın hata mesajı ("illegal combination of modifiers") ilk bakışta kafa karıştırıcı gelebilir — ama nedeni her zaman yukarıdaki çelişkilerden biridir. Hatayı gördüğünde, önce hangi modifier'ın override edilebilirliğe engel olduğunu sor.

Abstract Class'ın Bir Interface'i Implement Etmesi

Bir abstract class, `implements` ile bir (ya da birden fazla) interface'e bağlanabilir — ve Interface dersindeki kuralın aynısı burada da geçerli: **somut olması zorunlu olan yalnızca alt sınıflardır**, abstract class'ın kendisi interface'in metotlarını hemen gerçeklemek zorunda değildir:

```
interface Auditable {
    String auditLog();
}

// An abstract class that implements an interface is NOT required to provide
// bodies for that interface's methods -- it can leave auditLog() unimplemented
// and defer it to its own concrete subclasses, exactly the way it defers its
// own abstract methods like content().
abstract class Document implements Auditable {
    protected String title;

    Document(String title) {
        this.title = title;
    }

    abstract String content();
    // auditLog() from Auditable is inherited as still-abstract here --
    // Document compiles fine without implementing it.
}

class Report extends Document {
    Report(String title) {
        super(title);
    }

    @Override
    String content() {
        return "Q3 sales figures...";
    }

    @Override
    public String auditLog() {
        return "Report '" + title + "' was accessed";
    }
}

class AbstractImplementsInterfaceExample {
    public static void main(String[] args) {
        Document document = new Report("Q3 Report");
        System.out.println(document.content());  // Q3 sales figures...
        System.out.println(document.auditLog());  // Report 'Q3 Report' was accessed
    }
}
```

`Document`, `Auditable`'ı implement ediyor ama `auditLog()`'u hiç yazmıyor — kendi soyut metodu `content()` gibi, `auditLog()`'u da alt sınıfına (`Report`'a) devrediyor. `Report`, ikisini birden gerçeklemek zorunda: hem `Document`'ın kendi soyut metodu `content()`'i, hem `Auditable`'dan miras aldığı `auditLog()`'u — ikisi arasında derleyici için hiçbir fark yok, ikisi de "gerçeklenmemiş soyut metot" olarak aynı listede birikiyor.

Abstract Class vs Interface

Java 8'den beri default metotlarla arayüzler abstract class'a epey yaklaşsa da (Interface dersindeki "Tarihçe" bölümünde bahsettik), aralarındaki kalıcı farklar şöyle özetlenebilir:

- **Constructor:** Abstract class'ta var; interface'te hiçbir zaman olamaz.
- **Instance alanları (mutable state):** Abstract class'ta var; interface yalnızca `public static final` sabit tanımlayabilir.
- **Çoklu kalıtım:** Bir sınıf yalnızca bir abstract class'ı `extends` edebilir; ama istediği kadar interface'i `implements` edebilir.
- **Metotların erişim belirleyicisi:** Abstract class'ın somut metotları `public` / `protected` / `private` olabilir; interface metotları örtük olarak her zaman `public`'tir.
- **Amaç:** Abstract class, yakından ilişkili tiplerin ortak implementasyonunu paylaşmak içindir ("is-a" ilişkisi); interface, birbirine hiç ilişkisi olmayan tiplerin bile uyabileceği bir sözleşme sunmak içindir ("can-do" ilişkisi).

Pratik karar kuralı şu: paylaşacağın şey **durum (state) ve/veya ortak implementasyon** ise (gerçekten akraba tipler — bir `Animal` hiyerarşisi gibi) abstract class'ı seç; paylaşacağın şey yalnızca bir **yetenek sözleşmesi**yse (birbirine hiç akraba olmayan tipler bile uysun istiyorsan — `Comparable`, `Auditable` gibi) interface'i seç. İkisini **bir arada** kullanmak da tamamen normaldir — bir önceki bölümde `Document implements Auditable` örneğinde tam olarak bunu yaptık, ve "Ek: Mini Proje — Abstract Class + Interface Birlikte Kullanımı (Ödeme İşleyici)" bölümünde bunu daha büyük bir örnekle pekiştireceğiz.

> Bir sınıf hiyerarşisi tasarlarken kendine şunu sor: "Yarın bambaşka, hiç ilişkisi olmayan bir tip de bu sözleşmeye uymak isteyebilir mi?" Cevap evetse (`Serializable`, `Comparable` gibi) interface'e yönel; cevap "hayır, bu yalnızca benim `Animal` ailemin bir üyesi için anlamlı" ise abstract class'a yönel.

Template Method Pattern

Abstract class'ın en klasik ve en çok kullanılan tasarım deseni **Template Method**'dur: bir üst sınıf, bir algoritmanın **sabit iskeletini** (adımların sırasını) tanımlar, adımların bazılarını (ya da hepsini) soyut bırakarak alt sınıflara devreder:

```
abstract class DataProcessor {
    // The "template method" -- it defines the FIXED skeleton of the
    // algorithm, marked `final` so no subclass can ever change the order
    // (or skip a step); only the individual steps below are customizable.
    final void process() {
        validate();
        transform();
        save();
    }

    abstract void validate();

    abstract void transform();

    // A default step with a body -- subclasses may override it if they need
    // to, but most won't have to.
    void save() {
        System.out.println("Saved.");
    }
}

class CsvProcessor extends DataProcessor {
    @Override
    void validate() {
        System.out.println("Validating CSV structure...");
    }

    @Override
    void transform() {
        System.out.println("Transforming CSV rows...");
    }
}

class TemplateMethodExample {
    public static void main(String[] args) {
        DataProcessor processor = new CsvProcessor();
        processor.process();
        // Validating CSV structure...
        // Transforming CSV rows...
        // Saved.
    }
}
```

`process()` metodunun `final` işaretlendiğine dikkat et — bu bilinçli bir tasarım kararı: `CsvProcessor` (ya da gelecekte yazılacak herhangi bir `DataProcessor` alt sınıfı) adımların **sırasını** asla değiştiremez, yalnızca `validate()` ve `transform()`'un **içeriğini** kendine göre doldurabilir. `save()` ise soyut değil, gövdeli bir varsayılan adım (`default` bir interface metoduna çok benzer bir rolde) — alt sınıf isterse override eder, istemezse hiç dokunmaz.

> Template Method, "Hollywood Prensibi" diye anılan bir fikrin somut örneğidir: "Bizi arama, biz seni ararız" — akışın kontrolü (`process()`'in ne zaman, hangi sırada çalışacağı) üst sınıfta kalır; alt sınıf yalnızca üst sınıfın çağırdığı belirli noktalarda (`validate()`, `transform()`) devreye girer, akışı kendisi yönetmez.

Gerçek Dünya Kullanım Alanları

Abstract class, JDK'nın kendisinde de yaygın bir tasarım aracı — özellikle koleksiyon framework'ünde art arda kullanılıyor:

```
import java.util.AbstractList;

// java.util.AbstractList is a real, JDK-shipped abstract class: implement
// just get(int) and size(), and it gives you a fully working, read-only
// List implementation -- iterator(), contains(), indexOf(), toString(),
// even the for-each loop -- all built on top of those two methods.
class ReadOnlyRange extends AbstractList<Integer> {
    private final int start;
    private final int end;

    ReadOnlyRange(int start, int end) {
        this.start = start;
        this.end = end;
    }

    @Override
    public Integer get(int index) {
        return start + index;
    }

    @Override
    public int size() {
        return end - start;
    }
}

class ReadOnlyListExample {
    public static void main(String[] args) {
        ReadOnlyRange range = new ReadOnlyRange(5, 10);

        System.out.println(range);              // [5, 6, 7, 8, 9] -- toString() came for
free
        System.out.println(range.contains(7));   // true -- contains() came for free too

        for (int n : range) { // the for-each loop came for free as well
            System.out.print(n + " ");
        }
        System.out.println();
    }
}
```

`java.util.AbstractList`, gerçek bir JDK abstract class'ı: yalnızca `get(int)` ve `size()`'ı gerçeklersin, `iterator()`, `contains()`, `indexOf()`, `toString()`, hatta for-each döngüsü desteği **bedava** gelir — hepsi bu iki metodun üzerine, `AbstractList` içinde bir kez yazılmış. `java.util.AbstractMap` ve `java.util.AbstractQueue` de aynı felsefeyi izler: JDK, koleksiyon arayüzlerinin (`List`, `Map`, `Queue`) karmaşık kısmını bir kez abstract class olarak yazıp, sana yalnızca birkaç temel metodu gerçeklemeni bırakıyor.

Spring framework'ünde de aynı desen sıkça karşımıza çıkar — `AbstractController` gibi taban sınıflar, HTTP isteği işleme akışının (Template Method'daki gibi) sabit kısmını (loglama, hata yönetimi, response yazma) kendileri üstlenir; sen yalnızca gerçek iş mantığını içeren tek bir metodu doldurursun.

Best Practices

- Bir sınıfın yalnızca **taban sınıf** olarak kullanılmasını istiyorsan, hiç soyut metodu olmasa bile onu `abstract` işaretle (bkz. "Abstract Class vs Concrete Class").
- Template Method deseninde, algoritmanın sabit iskeletini tanımlayan metodu `final` yap — alt sınıfların yalnızca adımları doldurmasına izin ver, sırayı değiştirmesine değil (bkz. "Template Method Pattern").
- `protected` alan yerine, mümkünse `private` alan + `protected` getter/setter tercih et — bu, alt sınıfların üst sınıfın iç durumunu beklenmedik şekilde bozmasını önler (bkz. "Alanlar (Fields)" bölümündeki uyarı).
- Paylaşacağın şey yalnızca bir sözleşmeyse (state yok, implementasyon paylaşımı yok) abstract class yerine interface'i tercih et (bkz. "Abstract Class vs Interface").
- Bir abstract class'ın constructor'ını, yalnızca alt sınıflardan çağrılacağını bilerek tasarla — `public` yapmak yerine paket içi ya da `protected` bir constructor genelde niyeti daha iyi anlatır.

Yaygın Hatalar

1. Bir sınıfın soyut metodu olmadığı için `abstract` işaretlemeye gerek olmadığını sanmak. Doğrudan örneklenmesini engellemek istiyorsan, soyut metodun olup olmaması önemli değil — `abstract` anahtar kelimesinin kendisi yeterli (bkz. "Abstract Class vs Concrete Class").

2. Bir orta seviye abstract class'ın, miras aldığı tüm soyut metotları hemen gerçeklemesi gerektiğini sanmak. Yalnızca somut (concrete) bir sınıf bu zorunluluğa tabidir; bir abstract class, soyut metotları bir alt seviyeye devretmekte serbesttir (bkz. "Abstract Metotlar").

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

4. `abstract` metodu `private`, `static` ya da `final` ile birlikte kullanmaya çalışmak. Üçü de "override edilebilirlik" ile doğrudan çelişir, hiçbiri `abstract` ile bir arada yazılamaz (bkz. "Abstract Metotlarda Modifier ve Erişim Kuralları").

5. Abstract class'ı, tek bir somut alt sınıfı olacağını bile bile "ihtiyaten" kullanmak. Gerçekten paylaşılan durum/implementasyon yoksa ve gelecekte ikinci bir alt sınıf da beklenmiyorsa, gereksiz bir soyutlama katmanı eklemiş olursun — bu YAGNI ("ihtiyacın olmayacak") prensibinin klasik bir ihlalidir.

6. Template Method'daki iskelet metodunu `final` yapmayı unutup, bir alt sınıfın adımların sırasını (yanlışlıkla ya da kasıtlı olarak) değiştirmesine izin vermek. Bu, deseni kırılgan hale getirir — sıralamayı garanti eden tek şey `final`'dır (bkz. "Template Method Pattern").

Özet, Cheat Sheet ve Terimler Sözlüğü

Abstract class, Java'nın JDK 1.0'dan beri sahip olduğu, ortak durum ve davranışı paylaşmakla soyutlamayı bir arada sunan temel OOP aracıdır. Öne çıkan noktalar:

- `abstract` anahtar kelimesi bir sınıfa eklendiğinde, o sınıf soyut metodu olsun ya da olmasın **doğrudan örneklenemez**
- Bir abstract class hem soyut (gövdesiz) hem concrete (gövdeli) metot barındırabilir; somut olması zorunlu olan yalnızca **alt sınıflardır**, ara seviyedeki abstract sınıflar değil
- Interface'in aksine, bir abstract class **instance alanları** ve bir **constructor** tutabilir — bu, ikisi arasındaki en kalıcı fark
- 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
- `abstract`, `private` / `static` / `final` ile asla bir arada kullanılamaz — üçü de override edilebilirlikle çelişir
- Bir abstract class bir interface'i implement edebilir ve onun metotlarını da alt sınıflara devredebilir
- Template Method Pattern: sabit adım sırasını `final` bir metotla tanımla, adımların içeriğini soyut/override edilebilir metotlara bırak
- JDK'nın kendisi (`AbstractList`, `AbstractMap`, `AbstractQueue`) ve Spring (`AbstractController`) bu deseni yoğun şekilde kullanır

Hızlı referans:

```
// Temel tanım
abstract class Animal {
    protected String name;                  // instance alanı -- interface'te olamaz

    Animal(String name) {                    // constructor -- interface'te olamaz
        this.name = name;
    }

    abstract void makeSound();                // soyut metot -- gövde yok

    void sleep() {                            // concrete metot -- gövdeli
        System.out.println(name + " sleeping");
    }
}

class Dog extends Animal {
    Dog(String name) { super(name); }         // super() her zaman ilk satır

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

// Template Method
abstract class Pipeline {
    final void run() {                        // sıra sabit -- final
        step1();
        step2();
    }
    abstract void step1();
    abstract void step2();
}

// Abstract class + interface birlikte
interface Auditable { String auditLog(); }
abstract class Document implements Auditable {
    // auditLog()'u gerçeklemeye gerek yok -- alt sınıfa devredilir
    abstract String content();
}
```

Terimler Sözlüğü

Abstract class (soyut sınıf) — Doğrudan örneklenemeyen, ortak durum/davranışı paylaşan ve soyut metotlarla alt sınıflara sözleşme dayatan sınıf türü.

Soyut (abstract) metot — Gövdesi olmayan, `abstract` ile işaretlenen ve somut bir alt sınıf tarafından mutlaka gerçeklenmesi gereken metot.

Concrete (somut) metot/sınıf — Tam bir gövdesi olan metot; ya da tüm soyut metotlarını gerçeklemiş, doğrudan örneklenebilen sınıf.

`super(...)` — Bir alt sınıf constructor'ından üst sınıfın constructor'ını çağıran, her alt sınıf constructor'ının (açık ya da örtük) ilk satırı olması zorunlu çağrı.

Template Method Pattern — Bir algoritmanın sabit adım sırasını `final` bir metotla tanımlayıp, adımların içeriğini soyut/override edilebilir metotlara bırakan tasarım deseni.

Dinamik gönderim (dynamic dispatch) — Bir metot çağrısında, hangi implementasyonun çalışacağına derleme zamanında değil, nesnenin çalışma zamanındaki gerçek sınıfına bakılarak karar verilmesi.

`java.util.AbstractList` / `AbstractMap` / `AbstractQueue` — JDK'nın koleksiyon framework'ünde, yalnızca birkaç temel metodu gerçekleterek tam bir implementasyon kazandıran gerçek dünya abstract class örnekleri.

Ek: Mini Proje — Template Method ile Rapor Oluşturma Pipeline'ı

"Template Method Pattern" bölümünde öğrendiğimizi, gerçek bir senaryoya taşıyalım: farklı rapor türlerinin (satış, envanter) hepsi aynı dört adımdan (`validate` → `process` → `save` → `log`) geçiyor, ama her adımın içeriği rapor türüne göre değişiyor:

```
// A reusable Template Method skeleton: every report goes through the same
// four fixed steps, in the same fixed order -- only validate() and
// process() are mandatory for a subclass to fill in; save() and log() are
// optional hooks with sensible defaults.
abstract class ReportPipeline {
    final void run() {
        validate();
        process();
        save();
        log();
    }

    abstract void validate();

    abstract void process();

    void save() {
        System.out.println("[save] report written to disk");
    }

    void log() {
        System.out.println("[log] pipeline finished for " + getClass().getSimpleName());
    }
}
```

```
class SalesReportPipeline extends ReportPipeline {
    @Override
    void validate() {
        System.out.println("[validate] checking sales data...");
    }

    @Override
    void process() {
        System.out.println("[process] aggregating sales totals...");
    }
    // save() and log() are not overridden -- both use ReportPipeline's defaults.
}

class InventoryReportPipeline extends ReportPipeline {
    @Override
    void validate() {
        System.out.println("[validate] checking inventory counts...");
    }

    @Override
    void process() {
        System.out.println("[process] computing stock levels...");
    }

    @Override
    void log() {
        // This pipeline needs a different log step -- and since log() was a
        // concrete (not abstract) hook, overriding it is optional, not required.
        System.out.println("[log] inventory pipeline audited and archived");
    }
}

class ReportPipelineDemo {
    public static void main(String[] args) {
        new SalesReportPipeline().run();
        // [validate] checking sales data...
        // [process] aggregating sales totals...
        // [save] report written to disk
        // [log] pipeline finished for SalesReportPipeline

        System.out.println("---");

        new InventoryReportPipeline().run();
        // [validate] checking inventory counts...
        // [process] computing stock levels...
        // [save] report written to disk
        // [log] inventory pipeline audited and archived
    }
}
```

`SalesReportPipeline` yalnızca zorunlu iki adımı (`validate` , `process`) dolduruyor, `save()` / `log()` 'un varsayılanını kullanıyor. `InventoryReportPipeline` ise aynı iki zorunlu adımı doldurmanın yanında, `log()` 'u da kendi ihtiyacına göre override ediyor — `save()` soyut

olmadığı için bu override tamamen opsiyonel. İkisi de `run()`'ı hiç değiştiremiyor: adımların sırası, "Template Method Pattern" bölümünde vurguladığımız gibi, `final` sayesinde her zaman aynı.

> Gerçek bir rapor pipeline'ında `save()` muhtemelen bir veritabanına ya da dosya sistemine yazacaktır — bu mini projede bilinçli olarak `System.out.println` ile sadeleştirdik ki asıl konu (adım sırasının sabitlenmesi, hangi adımların zorunlu/opsiyonel olduğu) net kalsın.

Ek: Mini Proje — Abstract Class + Interface Birlikte Kullanımı (Ödeme İşleyici)

Son mini proje, "Abstract Class'ın Bir Interface'i Implement Etmesi" ve "Abstract Class vs Interface" bölümlerinde gördüğümüz fikri birleştiriyor: `PaymentProcessor` abstract class'ı hem paylaşılan durumu/algoritmayı (abstract class'ın işi) hem de `Auditable` sözleşmesini (interface'in işi) bir arada taşıyor:

```
interface Auditable {
    String auditTrail();
}

// Combines an abstract class (shared state + a fixed charging algorithm)
// with an interface (an unrelated "can-do" contract, Auditable) --
// PaymentProcessor implements Auditable but, like Document in
// AbstractImplementsInterfaceExample.java, leaves it to its own concrete
// subclasses to actually provide auditTrail().
abstract class PaymentProcessor implements Auditable {
    protected double totalProcessed = 0;

    // final: every processor charges the exact same way -- amount plus
    // whatever fee its own calculateFee() decides on -- only the fee
    // formula itself varies per processor.
    final double charge(double amount) {
        double fee = calculateFee(amount);
        totalProcessed += amount;
        return amount + fee;
    }

    abstract double calculateFee(double amount);
}

class CreditCardProcessor extends PaymentProcessor {
    @Override
    double calculateFee(double amount) {
        return amount * 0.029;
    }

    @Override
    public String auditTrail() {
        return "CreditCard: total processed = " + totalProcessed;
    }
}

class BankTransferProcessor extends PaymentProcessor {
    @Override
    double calculateFee(double amount) {
        return 1.50; // flat fee, regardless of amount
    }

    @Override
    public String auditTrail() {
        return "BankTransfer: total processed = " + totalProcessed;
    }
}
```

```
class PaymentProcessorDemo {
    public static void main(String[] args) {
        PaymentProcessor creditCard = new CreditCardProcessor();
        System.out.println(creditCard.charge(100)); // 102.9
        System.out.println(creditCard.auditTrail()); // CreditCard: total processed = 100.0

        PaymentProcessor bankTransfer = new BankTransferProcessor();
        System.out.println(bankTransfer.charge(500)); // 501.5
        System.out.println(bankTransfer.auditTrail()); // BankTransfer: total processed =
500.0

        // Both are used purely through the PaymentProcessor/Auditable
        // contracts here -- neither variable's declared type ever mentions
        // CreditCardProcessor or BankTransferProcessor directly.
    }
}
```

`charge(...)` metodunun `final` olması tesadüf değil — her ödeme işleyicisi tutarı `calculateFee(...)`'nin döndürdüğü ücretle **aynı şekilde** toplar, yalnızca ücret hesaplama **formülü** işleyiciden işleyiciye değişir (Template Method Pattern'in bir başka görünümü). `auditTrail()` ise `Auditable`'dan geliyor ve `PaymentProcessor` tarafından hiç gerçeklenmiyor — tıpkı `Document`/`auditLog()` örneğinde olduğu gibi, bu sorumluluk doğrudan somut alt sınıflara (`CreditCardProcessor`, `BankTransferProcessor`) devrediliyor.

> `PaymentProcessorDemo`'daki her iki değişkenin de tipi `PaymentProcessor` — hiçbiri `CreditCardProcessor` ya da `BankTransferProcessor` yazmıyor. Bu, Interface dersinin "Neden Var?" bölümünde tanıttığımız "implementasyona değil, sözleşmeye göre programla" ilkesinin, abstract class ve interface'in **aynı anda** kullanıldığı bir örnekte de geçerli olduğunu gösteriyor.