☕ JAVA SENIOR MÜLAKAT REHBERİ

BÖLÜM 4

Design Patterns

2025 Edition

4. Design Patterns (Tasarım Desenleri)

4.1 Design Patterns Kategorileri

Design Patterns, yazılım geliştirmede karşılaşılan yaygın problemlere yeniden kullanılabilir çözümlerdir. Gang of Four (GoF) tarafından 3 ana kategoride sınıflandırılmıştır:

Kategori	Amaç	Örnekler
Creational (Yaratımsal)	Nesne oluşturma mekanizmaları	Singleton, Factory, Builder, Prototype
Structural (Yapısal)	Sınıfların ve nesnelerin kompozisyonu	Adapter, Decorator, Proxy, Facade
Behavioral (Davranışsal)	Nesneler arası iletişim ve sorumluluk	Strategy, Observer, Template Method, Command

4.2 Singleton Pattern

Bir sınıftan sadece **tek bir instance** oluşturulmasını garanti eder ve bu instance'a global erişim sağlar. Database connection, Logger, Configuration gibi durumlarda kullanılır.

> **Amaç:** Sadece bir nesne olması ve ona kolay erişim
> **Kullanım:** Database connection pool, Logger, Cache, Configuration

```
// 1. EAGER LOADING - Uygulama başında oluştur
public class EagerSingleton {
    // Class yüklendiğinde hemen oluşturulur
    private static final EagerSingleton instance = new EagerSingleton();

    private EagerSingleton() {
        System.out.println("Singleton oluşturuldu");
    }

    public static EagerSingleton getInstance() {
        return instance;
    }
}

// 2. LAZY LOADING - İlk kullanımda oluştur
public class LazySingleton {
    private static LazySingleton instance;

    private LazySingleton() {}

    // ⚠  Thread-safe DEĞİL
```

```
    public static LazySingleton getInstance() {
        if (instance == null) {
            instance = new LazySingleton();
        }
        return instance;
    }
}

// 3. THREAD-SAFE LAZY LOADING (Double-Checked Locking)
public class ThreadSafeSingleton {
    private static volatile ThreadSafeSingleton instance;

    private ThreadSafeSingleton() {}

    public static ThreadSafeSingleton getInstance() {
        if (instance == null) { // İlk kontrol (lock olmadan)
            synchronized (ThreadSafeSingleton.class) {
                if (instance == null) { // İkinci kontrol (lock içinde)
                    instance = new ThreadSafeSingleton();
                }
            }
        }
        return instance;
    }
}

// 4. ENUM SINGLETON (En iyi yöntem!)
public enum EnumSingleton {
    INSTANCE;

    private String value;

    public String getValue() {
        return value;
    }

    public void setValue(String value) {
        this.value = value;
    }

    public void doSomething() {
        System.out.println("Enum Singleton çalışıyor");
    }
}

// Kullanım
EnumSingleton.INSTANCE.setValue("Test");
EnumSingleton.INSTANCE.doSomething();

// 5. BILL PUGH SINGLETON (Inner Class)
public class BillPughSingleton {
    private BillPughSingleton() {}

    // Inner static class - lazy loading + thread-safe
    private static class SingletonHelper {
        private static final BillPughSingleton INSTANCE = new BillPughSingleton();
    }

    public static BillPughSingleton getInstance() {
        return SingletonHelper.INSTANCE;
    }
}

// GERÇEK DÜNYA ÖRNEĞİ: Database Connection
public class DatabaseConnection {
    private static volatile DatabaseConnection instance;
    private Connection connection;

    private DatabaseConnection() {
        try {
            connection = DriverManager.getConnection(
                "jdbc:mysql://localhost:3306/mydb", "user", "pass");
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    public static DatabaseConnection getInstance() {
        if (instance == null) {
```

```
            synchronized (DatabaseConnection.class) {
                if (instance == null) {
                    instance = new DatabaseConnection();
                }
            }
        }
        return instance;
    }

    public Connection getConnection() {
        return connection;
    }
}
```

Mülakat Sorusu: "Singleton anti-pattern midir?"
Cevap: Bazı durumlarda evet. Global state yaratır, testing zorlaşır, tight coupling oluşur. Dependency Injection ile alternatif olabilir. Ancak Logger, Configuration gibi durumlar için uygun.

4.3 Factory Pattern

Nesne oluşturma mantığını gizler. Client hangi concrete sınıfın oluşturulduğunu bilmez, sadece interface ile çalışır. **Factory Method** ve **Abstract Factory** olmak üzere iki türü vardır.

Amaç: Nesne oluşturmayı kapsüllemek, loose coupling
Kullanım: Farklı tipte nesneler oluşturma, runtime'da karar verme

```
// Interface
interface Shape {
    void draw();
}

// Concrete sınıflar
class Circle implements Shape {
    @Override
    public void draw() {
        System.out.println("Circle çizildi");
    }
}

class Rectangle implements Shape {
    @Override
    public void draw() {
        System.out.println("Rectangle çizildi");
    }
}

class Triangle implements Shape {
    @Override
    public void draw() {
        System.out.println("Triangle çizildi");
    }
}

// FACTORY - Nesne oluşturma mantığı
class ShapeFactory {
    // Factory Method
    public Shape getShape(String shapeType) {
        if (shapeType == null) {
            return null;
        }

        switch (shapeType.toUpperCase()) {
            case "CIRCLE":
                return new Circle();
            case "RECTANGLE":
                return new Rectangle();
            case "TRIANGLE":
                return new Triangle();
            default:
                throw new IllegalArgumentException("Unknown shape: " + shapeType);
        }
```

```
    }
}

// Kullanım
public class FactoryPatternDemo {
    public static void main(String[] args) {
        ShapeFactory factory = new ShapeFactory();

        // Client concrete sınıfı bilmiyor
        Shape shape1 = factory.getShape("CIRCLE");
        shape1.draw(); // Circle çizildi

        Shape shape2 = factory.getShape("RECTANGLE");
        shape2.draw(); // Rectangle çizildi
    }
}

// GERÇEK DÜNYA ÖRNEĞİ: Notification System
interface Notification {
    void send(String message);
}

class EmailNotification implements Notification {
    @Override
    public void send(String message) {
        System.out.println("Email gönderildi: " + message);
    }
}

class SMSNotification implements Notification {
    @Override
    public void send(String message) {
        System.out.println("SMS gönderildi: " + message);
    }
}

class PushNotification implements Notification {
    @Override
    public void send(String message) {
        System.out.println("Push notification gönderildi: " + message);
    }
}

class NotificationFactory {
    public static Notification createNotification(String channel) {
        switch (channel.toUpperCase()) {
            case "EMAIL":
                return new EmailNotification();
            case "SMS":
                return new SMSNotification();
            case "PUSH":
                return new PushNotification();
            default:
                throw new IllegalArgumentException("Unknown channel: " + channel);
        }
    }
}

// Kullanım
Notification notification = NotificationFactory.createNotification("EMAIL");
notification.send("Hoşgeldiniz!");
```

4.4 Strategy Pattern

Algoritmaları kapsüller ve birbirinin yerine kullanılabilir hale getirir. **Runtime'da strateji değiştirilebilir.** If-else veya switch-case kalabalığını önler, Open/Closed prensibine uyar.

> **Amaç:** Algoritma ailesini tanımla, kapsülle ve değiştirilebilir yap
> **Kullanım:** Ödeme yöntemleri, sıralama algoritmaları, validasyon kuralları

```
// Strategy Interface
interface PaymentStrategy {
```

```
    void pay(int amount);
}

// Concrete Strategies
class CreditCardPayment implements PaymentStrategy {
    private String cardNumber;
    private String cvv;

    public CreditCardPayment(String cardNumber, String cvv) {
        this.cardNumber = cardNumber;
        this.cvv = cvv;
    }

    @Override
    public void pay(int amount) {
        System.out.println("Kredi kartı ile ödendi: " + amount + " TL");
        System.out.println("Kart: " + cardNumber);
    }
}

class PayPalPayment implements PaymentStrategy {
    private String email;

    public PayPalPayment(String email) {
        this.email = email;
    }

    @Override
    public void pay(int amount) {
        System.out.println("PayPal ile ödendi: " + amount + " TL");
        System.out.println("Email: " + email);
    }
}

class BitcoinPayment implements PaymentStrategy {
    private String walletAddress;

    public BitcoinPayment(String walletAddress) {
        this.walletAddress = walletAddress;
    }

    @Override
    public void pay(int amount) {
        System.out.println("Bitcoin ile ödendi: " + amount + " TL");
        System.out.println("Wallet: " + walletAddress);
    }
}

// Context - Stratejiyi kullanan sınıf
class ShoppingCart {
    private List items = new ArrayList<>();
    private PaymentStrategy paymentStrategy;

    public void addItem(Item item) {
        items.add(item);
    }

    public void setPaymentStrategy(PaymentStrategy strategy) {
        this.paymentStrategy = strategy;
    }

    public void checkout() {
        int total = items.stream()
            .mapToInt(Item::getPrice)
            .sum();

        paymentStrategy.pay(total);
    }
}

class Item {
    private String name;
    private int price;

    public Item(String name, int price) {
        this.name = name;
        this.price = price;
    }
```

```
    public int getPrice() { return price; }
}

// Kullanım
public class StrategyPatternDemo {
    public static void main(String[] args) {
        ShoppingCart cart = new ShoppingCart();
        cart.addItem(new Item("Laptop", 15000));
        cart.addItem(new Item("Mouse", 500));

        // Runtime'da strateji değiştirebiliriz
        cart.setPaymentStrategy(new CreditCardPayment("1234-5678", "123"));
        cart.checkout();

        // Farklı ödeme yöntemi
        cart.setPaymentStrategy(new PayPalPayment("user@example.com"));
        cart.checkout();
    }
}

// GERÇEK DÜNYA ÖRNEĞİ: Sorting Strategy
interface SortingStrategy {
    void sort(int[] array);
}

class BubbleSort implements SortingStrategy {
    @Override
    public void sort(int[] array) {
        System.out.println("Bubble Sort kullanıldı");
        // Bubble sort implementasyonu
    }
}

class QuickSort implements SortingStrategy {
    @Override
    public void sort(int[] array) {
        System.out.println("Quick Sort kullanıldı");
        // Quick sort implementasyonu
    }
}

class Sorter {
    private SortingStrategy strategy;

    public void setStrategy(SortingStrategy strategy) {
        this.strategy = strategy;
    }

    public void sort(int[] array) {
        strategy.sort(array);
    }
}
```

JAVA SENIOR MÜLAKAT REHBERİ

BÖLÜM 3

Multithreading & Concurrency

2025 Edition

3. Multithreading & Concurrency

3.1 Thread Oluşturma Yöntemleri

Java'da paralel işlem yapabilmek için thread'ler kullanılır. Thread oluşturmanın üç temel yolu vardır:

1. **Thread sınıfını extend etmek:** Eski yöntem, kalıtım kullanır.
2. **Runnable interface'ini implement etmek:** Tercih edilen, daha esnek.
3. **ExecutorService kullanmak:** Modern, profesyonel uygulamalar için önerilen.

```
// 1. YÖNTEM: Thread sınıfını extend etmek
class MyThread extends Thread {
    @Override
    public void run() {
        System.out.println("Thread çalışıyor: " + Thread.currentThread().getName());
        for (int i = 1; i <= 5; i++) {
            System.out.println(Thread.currentThread().getName() + " - " + i);
            try {
                Thread.sleep(1000); // 1 saniye bekle
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

// Kullanım
MyThread t1 = new MyThread();
t1.start(); // run() değil, start() çağrılır!

// 2. YÖNTEM: Runnable interface (tercih edilen)
class MyRunnable implements Runnable {
    @Override
    public void run() {
        System.out.println("Runnable çalışıyor: " + Thread.currentThread().getName());
    }
}

// Kullanım
Thread t2 = new Thread(new MyRunnable());
t2.start();

// Lambda ile kısa yazım
Thread t3 = new Thread(() -> {
    System.out.println("Lambda thread: " + Thread.currentThread().getName());
});
t3.start();

// 3. YÖNTEM: ExecutorService (en iyi pratik)
import java.util.concurrent.*;

public class ThreadDemo {
    public static void main(String[] args) {
        // Fixed thread pool - 3 thread oluştur
        ExecutorService executor = Executors.newFixedThreadPool(3);

        // 5 task gönder (3 thread arasında paylaşılır)
        for (int i = 1; i <= 5; i++) {
            int taskId = i;
            executor.submit(() -> {
                System.out.println("Task " + taskId + " - Thread: " +
                    Thread.currentThread().getName());
                try {
                    Thread.sleep(2000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            });
        }

        executor.shutdown(); // Yeni task kabul etme

        try {
```

```
            // Tüm task'lerin bitmesini bekle (max 1 dakika)
            if (!executor.awaitTermination(60, TimeUnit.SECONDS)) {
                executor.shutdownNow(); // Zorla kapat
            }
        } catch (InterruptedException e) {
            executor.shutdownNow();
        }

        System.out.println("Tüm işlemler tamamlandı");
    }
}
```

ExecutorService Türü	Açıklama	Kullanım Alanı
newFixedThreadPool(n)	Sabit sayıda thread havuzu	Belirli sayıda paralel işlem
newCachedThreadPool()	İhtiyaca göre thread oluşturur	Kısa süreli, çok sayıda task
newSingleThreadExecutor()	Tek thread, sıralı işlem	Task'lerin sırayla çalışması gerekiyorsa
newScheduledThreadPool(n)	Zamanlanmış görevler	Periyodik veya gecikmeli işlemler

3.2 Thread Yaşam Döngüsü & Thread States

Bir thread'in yaşam döngüsü boyunca farklı state'lerden geçer. Bu state'leri anlamak, thread davranışlarını kontrol etmek için önemlidir.

Thread Yaşam Döngüsü:

NEW → Thread oluşturuldu, henüz başlamadı
↓ `start()` çağrıldı
RUNNABLE → Çalışmaya hazır veya çalışıyor
↓ `sleep()`, `wait()`, I/O
BLOCKED/WAITING → Bekliyor (lock, notify)
↓ Bekleme sona erdi
RUNNABLE → Tekrar çalışabilir
↓ `run()` metodu bitti
TERMINATED → Thread sonlandı

```
public class ThreadStatesDemo {
    public static void main(String[] args) throws InterruptedException {
        Thread thread = new Thread(() -> {
            try {
                System.out.println("Thread başladı");
                Thread.sleep(2000); // TIMED_WAITING
                System.out.println("Thread uyandı");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });

        // NEW state
        System.out.println("State: " + thread.getState()); // NEW

        thread.start();
        Thread.sleep(100);

        // RUNNABLE veya TIMED_WAITING
        System.out.println("State: " + thread.getState()); // TIMED_WAITING

        thread.join(); // Ana thread bekler

        // TERMINATED
        System.out.println("State: " + thread.getState()); // TERMINATED
    }
}
```

JAVA SENIOR MÜLAKAT REHBERİ

BÖLÜM 2

OOP & SOLID Prensipleri

2025 Edition

2. OOP & SOLID Prensipleri

2.1 OOP'nin 4 Temel Prensibi

Object-Oriented Programming (OOP), yazılım geliştirmede kodun yeniden kullanılabilirliğini, sürdürülebilirliğini ve anlaşılabilirliğini artıran bir paradigmadır. Dört temel prensip üzerine kuruludur:

1. Encapsulation (Kapsülleme): Veriyi gizleme ve kontrollü erişim sağlama. Private değişkenler + public getter/setter metodları.

2. Inheritance (Kalıtım): Mevcut sınıflardan yeni sınıflar türetme. Kod tekrarını önler, "is-a" ilişkisi.

3. Polymorphism (Çok biçimlilik): Aynı arayüz, farklı uygulamalar. Method overloading (derleme zamanı) ve overriding (çalışma zamanı).

4. Abstraction (Soyutlama): Karmaşık detayları gizleme, sadece önemli özellikleri gösterme. Abstract class ve interface kullanımı.

```
// 1. ENCAPSULATION - Kapsülleme
class BankAccount {
    private double balance; // Private - dışarıdan erişilemez

    public double getBalance() {
        return balance;
    }

    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
        }
    }

    // Kontrollü erişim sağlanır
    public boolean withdraw(double amount) {
        if (amount > 0 && amount <= balance) {
            balance -= amount;
            return true;
        }
        return false;
    }
}

// 2. INHERITANCE - Kalıtım
class Animal {
    protected String name;

    public void eat() {
```

```
        System.out.println(name + " yemek yiyor");
    }
}

class Dog extends Animal {
    public void bark() {
        System.out.println(name + " havlıyor");
    }
}

// 3. POLYMORPHISM - Çok Biçimlilik
class Shape {
    public double calculateArea() {
        return 0;
    }
}

class Circle extends Shape {
    private double radius;

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

class Rectangle extends Shape {
    private double width, height;

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

// Kullanım - Polymorphism
Shape shape1 = new Circle();
Shape shape2 = new Rectangle();
System.out.println(shape1.calculateArea()); // Circle'ın metodu
System.out.println(shape2.calculateArea()); // Rectangle'ın metodu

// 4. ABSTRACTION - Soyutlama
abstract class Vehicle {
    abstract void start(); // Soyut metot

    public void stop() {    // Concrete metot
        System.out.println("Araç durdu");
    }
}

class Car extends Vehicle {
    @Override
    void start() {
        System.out.println("Araba kontak ile çalıştı");
    }
}
```

2.2 SOLID Prensipleri

SOLID, Robert C. Martin (Uncle Bob) tarafından önerilen, nesne yönelimli tasarımda kodun sürdürülebilir, esnek ve anlaşılır olmasını sağlayan 5 temel prensiptir. Bu prensipler, yazılımın değişikliklere karşı dirençli ve test edilebilir olmasını hedefler.

Prensip	Açıklama	Örnek Senaryo
S Single Responsibility	Bir sınıf sadece tek bir işten sorumlu olmalı, değişmek için tek bir nedeni olmalı	User sınıfı sadece kullanıcı verilerini tutar. Kaydetme işi UserRepository'e, mail gönderme EmailService'e aittir
O Open/Closed	Sınıflar genişlemeye açık, değişikliğe kapalı olmalı	Yeni ödeme yöntemi eklemek için mevcut kodu değiştirmek yerine yeni bir PaymentMethod implementasyonu oluştur
L		

Liskov Substitution	Alt sınıflar, üst sınıfın yerine kullanıldığında program bozulmamalı	Bird sınıfından türeyen Penguin, fly() metodunu implement ederse prensip ihlal edilir (penguen uçamaz)
I Interface Segregation	Kullanılmayan metotları içeren büyük arayüzler yerine küçük, spesifik arayüzler tercih edilmeli	Worker interface'i yerine Workable, Eatable, Sleepable gibi küçük interface'ler
D Dependency Inversion	Yüksek seviye modüller, düşük seviye modüllere bağımlı olmamalı. Her ikisi de soyutlamalara bağımlı olmalı	OrderService, konkret MySQLRepository yerine Repository interface'ine bağımlı olmalı

2.3 Single Responsibility Principle (SRP)

Bir sınıf sadece bir işten sorumlu olmalıdır. Birden fazla sorumluluğa sahip sınıflar, değişiklik gerektiren birden fazla nedene sahip olur ve bu da bakımı zorlaştırır.

```
// ✕ YANLIŞ - Çok fazla sorumluluk
class User {
    private String name;
    private String email;

    // 1. Kullanıcı veri yönetimi
    public void setName(String name) { this.name = name; }

    // 2. Database işlemleri
    public void saveToDatabase() {
        // Database connection ve save logic
        System.out.println("User kaydedildi");
    }

    // 3. Email gönderme
    public void sendWelcomeEmail() {
        // SMTP configuration ve email logic
        System.out.println("Hoşgeldin emaili gönderildi");
    }

    // 4. Rapor oluşturma
    public String generateReport() {
        return "User Report: " + name;
    }
}

// ✓ DOĞRU - Her sınıf tek sorumluluk
class User {
    private String name;
    private String email;

    // Sadece kullanıcı verilerini tutar
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public String getEmail() { return email; }
    public void setEmail(String email) { this.email = email; }
}

class UserRepository {
    // Sadece database işlemlerinden sorumlu
    public void save(User user) {
        System.out.println("User kaydedildi: " + user.getName());
    }

    public User findById(Long id) {
        // Database'den user getir
        return new User();
    }
}

class EmailService {
    // Sadece email gönderiminden sorumlu
    public void sendWelcomeEmail(User user) {
        System.out.println("Email gönderildi: " + user.getEmail());
    }
}

class ReportGenerator {
    // Sadece rapor oluşturmaktan sorumlu
```

```
    public String generateUserReport(User user) {
        return "User Report: " + user.getName();
    }
}
```

2.4 Open/Closed Principle (OCP)

Sınıflar genişlemeye açık, değişikliğe kapalı olmalıdır. Yeni özellikler eklemek için mevcut kodu değiştirmek yerine, yeni sınıflar ekleyerek genişletmeliyiz.

```
// ✕ YANLIŞ - Her yeni ödeme tipi için if-else eklenir
class PaymentProcessor {
    public void processPayment(String type, double amount) {
        if (type.equals("CREDIT_CARD")) {
            System.out.println("Kredi kartı ile ödeme: " + amount);
        } else if (type.equals("PAYPAL")) {
            System.out.println("PayPal ile ödeme: " + amount);
        } else if (type.equals("BITCOIN")) {
            // Yeni eklendi - mevcut kod değişti!
            System.out.println("Bitcoin ile ödeme: " + amount);
        }
    }
}

// ✓ DOĞRU - Interface kullanarak genişlemeye açık
interface PaymentMethod {
    void pay(double amount);
}

class CreditCardPayment implements PaymentMethod {
    @Override
    public void pay(double amount) {
        System.out.println("Kredi kartı ile ödeme: " + amount);
    }
}

class PayPalPayment implements PaymentMethod {
    @Override
    public void pay(double amount) {
        System.out.println("PayPal ile ödeme: " + amount);
    }
}

// Yeni ödeme yöntemi - mevcut kod değişmedi!
class BitcoinPayment implements PaymentMethod {
    @Override
    public void pay(double amount) {
        System.out.println("Bitcoin ile ödeme: " + amount);
    }
}

class PaymentProcessor {
    public void processPayment(PaymentMethod method, double amount) {
        method.pay(amount); // Polymorphism
    }
}

// Kullanım
PaymentProcessor processor = new PaymentProcessor();
processor.processPayment(new CreditCardPayment(), 100.0);
processor.processPayment(new BitcoinPayment(), 200.0);
```

2.5 Liskov Substitution Principle (LSP)

Alt sınıflar, üst sınıfın yerine kullanıldığında programın davranışı bozulmamalıdır. Alt sınıf, üst sınıfın tüm özelliklerini doğru şekilde yerine getirmelidir.

```
// ✕ YANLIŞ - Penguen uçamaz, LSP ihlali
class Bird {
```

```
    public void fly() {
        System.out.println("Kuş uçuyor");
    }
}

class Sparrow extends Bird {
    @Override
    public void fly() {
        System.out.println("Serçe uçuyor");
    }
}

class Penguin extends Bird {
    @Override
    public void fly() {
        throw new UnsupportedOperationException("Penguenler uçamaz!");
    }
}

// Client kodu bozulur
void makeBirdFly(Bird bird) {
    bird.fly(); // Penguin için exception fırlatır!
}

// ✅ DOĞRU - Doğru soyutlama
abstract class Bird {
    public abstract void eat();
}

interface Flyable {
    void fly();
}

class Sparrow extends Bird implements Flyable {
    @Override
    public void eat() {
        System.out.println("Serçe yemek yiyor");
    }

    @Override
    public void fly() {
        System.out.println("Serçe uçuyor");
    }
}

class Penguin extends Bird {
    @Override
    public void eat() {
        System.out.println("Penguen yemek yiyor");
    }

    public void swim() {
        System.out.println("Penguen yüzüyor");
    }
}

// Client kodu güvenli
void makeFlyableFly(Flyable flyable) {
    flyable.fly(); // Sadece uçabilen kuşlar için çağrılır
}
```

2.6 Interface Segregation & Dependency Inversion

Interface Segregation (ISP): Büyük, monolitik interface'ler yerine küçük, spesifik interface'ler kullanılmalıdır.

Dependency Inversion (DIP): Yüksek seviye modüller, düşük seviye modüllere doğrudan bağımlı olmamalı. Her ikisi de abstraction'lara bağımlı olmalıdır.

```
// ❌ ISP İHLALİ - Gereksiz metotlar
interface Worker {
    void work();
    void eat();
    void sleep();
```

```
}

class Robot implements Worker {
    public void work() { System.out.println("Robot çalışıyor"); }
    public void eat() { /* Robot yemek yemez! */ }
    public void sleep() { /* Robot uyumaz! */ }
}

// ✅ ISP DOĞRU - Küçük interface'ler
interface Workable {
    void work();
}

interface Eatable {
    void eat();
}

interface Sleepable {
    void sleep();
}

class Human implements Workable, Eatable, Sleepable {
    public void work() { System.out.println("İnsan çalışıyor"); }
    public void eat() { System.out.println("İnsan yemek yiyor"); }
    public void sleep() { System.out.println("İnsan uyuyor"); }
}

class Robot implements Workable {
    public void work() { System.out.println("Robot çalışıyor"); }
}

// ❌ DIP İHLALİ - Concrete sınıfa bağımlılık
class OrderService {
    private MySQLDatabase database = new MySQLDatabase();

    public void saveOrder(Order order) {
        database.save(order); // Sıkı bağımlılık!
    }
}

// ✅ DIP DOĞRU - Abstraction'a bağımlılık
interface Database {
    void save(Object entity);
    Object findById(Long id);
}

class MySQLDatabase implements Database {
    public void save(Object entity) {
        System.out.println("MySQL'e kaydedildi");
    }

    public Object findById(Long id) {
        return new Object();
    }
}

class MongoDatabase implements Database {
    public void save(Object entity) {
        System.out.println("MongoDB'ye kaydedildi");
    }

    public Object findById(Long id) {
        return new Object();
    }
}

class OrderService {
    private Database database; // Interface'e bağımlı

    // Constructor Injection
    public OrderService(Database database) {
        this.database = database;
    }

    public void saveOrder(Order order) {
        database.save(order); // Hangi DB olduğu önemli değil
    }
}
```

```
// Kullanım - Esneklik
OrderService service1 = new OrderService(new MySQLDatabase());
OrderService service2 = new OrderService(new MongoDatabase());
```


☕ JAVA SENIOR MÜLAKAT REHBERİ

BÖLÜM 1

Java Core

2025 Edition

1. Java Core

1.1 JVM, JDK, JRE Farkı

JVM (Java Virtual Machine): Java bytecode'unu çalıştıran sanal makinadır. Platform bağımsızlığını sağlar ve farklı işletim sistemlerinde aynı bytecode'un çalışmasını mümkün kılar.

JRE (Java Runtime Environment): JVM + Kütüphaneler içerir. Java uygulamalarını çalıştırmak için minimum gerekli ortamdır. Sadece kullanıcılar için yeterlidir.

JDK (Java Development Kit): JRE + Geliştirme araçları (javac, debugger, javadoc) içerir. Java uygulaması geliştirmek için gereklidir. Geliştiriciler için zorunludur.

Hiyerarşi İlişkisi:

JDK ⊃ JRE ⊃ JVM

• JDK → Geliştirme + Çalıştırma
• JRE → Sadece Çalıştırma
• JVM → Bytecode Yorumlayıcı

1.2 Memory Management & Garbage Collection

Java'da bellek yönetimi otomatik olarak yapılır. Heap alanında oluşturulan nesneler, referansları kalmadığında Garbage Collector (GC) tarafından otomatik olarak temizlenir.

Heap: Nesnelerin ve dizilerin saklandığı dinamik bellek alanı. Tüm thread'ler tarafından paylaşılır.
Stack: Metodların çağrı bilgileri ve yerel değişkenlerin saklandığı alan. Her thread'in kendi stack'i vardır.

GC Algoritmaları: Serial GC, Parallel GC, CMS (Concurrent Mark Sweep), G1GC (Garbage First), ZGC (Z Garbage Collector), Shenandoah GC.

```
public class GCExample {
    public static void main(String[] args) {
        // Heap'te String nesnesi oluştur
        String str = new String("Java Senior Interview");

        // Referansı kopar - nesne artık erişilemez
        str = null;

        // GC'yi manuel olarak önerebiliriz (garanti değil!)
        System.gc();

        // Veya Runtime üzerinden
        Runtime.getRuntime().gc();

        System.out.println("GC çağrıldı - sistem belleği optimize edebilir");
    }
}

// finalize() metodu (deprecated - kullanılmamalı)
class Resource {
    @Override
    protected void finalize() throws Throwable {
        // GC nesneyi temizlemeden önce çağrılır
        System.out.println("Kaynak temizleniyor");
        super.finalize();
    }
}
```

Bellek Alanı	İçerik	Yaşam Süresi
Heap	Nesneler, diziler	GC tarafından yönetilir
Stack	Metot çağrıları, yerel değişkenler	Metot bitince silinir
Method Area	Class bilgileri, static değişkenler	Uygulama boyunca

1.3 Collections Framework

Java Collections Framework, veri yapılarını standart bir şekilde yönetmek için güçlü bir altyapı sunar. List, Set, Map, Queue gibi temel arayüzler ve bunların implementasyonlarını içerir.

List: Sıralı, indeksli, tekrarlı elemanlara izin verir.
Set: Tekrarsız elemanlar içerir, sıralama garanti edilmez (TreeSet hariç).
Map: Key-Value çiftlerini saklar, her key benzersiz olmalıdır.
Queue: FIFO (First In First Out) yapısıyla çalışır.

Arayüz	Implementasyon	Özellik	Zaman Karmaşıklığı
List	ArrayList	Dinamik dizi, hızlı erişim	O(1) erişim, O(n) ekleme
List	LinkedList	Çift yönlü liste, hızlı ekleme/silme	O(1) ekleme, O(n) erişim
Set	HashSet	Hash tabanlı, benzersiz elemanlar	O(1) ortalama
Set	TreeSet	Sıralı, Red-Black Tree	O(log n)
Map	HashMap	Key-Value, hash tabanlı	O(1) ortalama
Map	TreeMap	Key'e göre sıralı	O(log n)

```
import java.util.*;

public class CollectionsDemo {
    public static void main(String[] args) {
        // List - Sıralı, tekrarlı
        List<String> list = new ArrayList<>();
        list.add("Java");
        list.add("Spring");
```

```
        list.add("Java"); // Tekrar eklenebilir
        System.out.println("List: " + list); // [Java, Spring, Java]

        // Set - Tekrarsız
        Set<Integer> set = new HashSet<>();
        set.add(1);
        set.add(2);
        set.add(1); // Eklenmez
        System.out.println("Set: " + set); // [1, 2]

        // Map - Key-Value
        Map<String, Integer> map = new HashMap<>();
        map.put("Java", 17);
        map.put("Spring", 6);
        map.put("Hibernate", 5);
        System.out.println("Map: " + map);

        // Iteration
        for (Map.Entry<String, Integer> entry : map.entrySet()) {
            System.out.println(entry.getKey() + " -> " + entry.getValue());
        }

        // Stream API ile
        list.stream()
            .filter(s -> s.startsWith("J"))
            .forEach(System.out::println);
    }
}
```

1.4 Exception Handling

Java'da hatalar **Checked Exceptions** (derleme zamanı kontrol edilir) ve **Unchecked Exceptions** (çalışma zamanı) olarak ikiye ayrılır.

Checked Exceptions: IOException, SQLException gibi derleme zamanında yakalanması zorunlu hatalar.
Unchecked Exceptions: NullPointerException, ArrayIndexOutOfBoundsException gibi RuntimeException'dan türeyen hatalar.

Try-catch-finally blokları ile hata yönetimi yapılır. Java 7'den itibaren try-with-resources kullanılabilir.

```
import java.io.*;

public class ExceptionDemo {
    public static void main(String[] args) {
        // Try-catch-finally
        try {
            int result = divide(10, 0);
            System.out.println("Sonuç: " + result);
        } catch (ArithmeticException e) {
            System.out.println("Hata: Sıfıra bölme - " + e.getMessage());
        } catch (Exception e) {
            System.out.println("Genel hata: " + e.getMessage());
        } finally {
            System.out.println("Finally bloğu her zaman çalışır");
        }

        // Try-with-resources (AutoCloseable)
        try (BufferedReader br = new BufferedReader(new FileReader("test.txt"))) {
            String line = br.readLine();
            System.out.println(line);
        } catch (IOException e) {
            e.printStackTrace();
        }
        // br otomatik olarak kapanır

        // Custom Exception
        try {
            validateAge(15);
        } catch (InvalidAgeException e) {
            System.out.println(e.getMessage());
        }
    }
```

```
    public static int divide(int a, int b) {
        return a / b;
    }

    public static void validateAge(int age) throws InvalidAgeException {
        if (age < 18) {
            throw new InvalidAgeException("Yaş 18'den küçük olamaz!");
        }
    }
}

// Custom Exception
class InvalidAgeException extends Exception {
    public InvalidAgeException(String message) {
        super(message);
    }
}
```

Exception Türü	Örnek	Yakalanması
Checked	IOException, SQLException	Zorunlu (throws veya try-catch)
Unchecked	NullPointerException, ArithmeticException	Opsiyonel
Error	OutOfMemoryError, StackOverflowError	Yakalanmamalı