



# Orientação a Objetos no Java 17 e Boas Práticas no Desenvolvimento de Software Escalável

A orientação a objetos (OO) continua sendo um dos paradigmas mais amplamente adotados para o desenvolvimento de sistemas de software, devido à sua capacidade de modularizar a lógica, facilitar a manutenção e promover a escalabilidade. O Java 17, com novas funcionalidades e aprimoramentos, oferece uma plataforma robusta para a construção de sistemas orientados a objetos escaláveis e eficientes. Para desenvolver software escalável de alta qualidade, é fundamental combinar os princípios da orientação a objetos com boas práticas de design e desenvolvimento.

## Princípios da Orientação a Objetos no Java 17

Os conceitos fundamentais da orientação a objetos são cruciais para a criação de sistemas robustos e flexíveis. Vamos explorar cada princípio com exemplos de código real e práticas recomendadas.

### 1. Encapsulamento

O encapsulamento protege os dados internos de uma classe ao expor apenas os métodos necessários. No Java 17, o uso de **Records** simplifica o processo de criar classes imutáveis e autoexplicativas.

Veja um exemplo de um Record utilizado para encapsular dados financeiros:

```
public record Transaction(String id, BigDecimal amount, LocalDate date) {  
    public Transaction {  
        if (amount.compareTo(BigDecimal.ZERO) <= 0) {  
            throw new IllegalArgumentException("Transaction amount must be positive");  
        }  
    }  
}
```

Neste exemplo, a classe `Transaction` encapsula os dados e garante que o valor da transação seja sempre positivo. A imutabilidade do record evita alterações indesejadas, sendo uma prática comum em sistemas críticos como os de transações bancárias, utilizados em plataformas como **PayPal** ou **Stripe**.



## 2. Herança

A herança permite que classes reutilizem código de outras classes, promovendo a reutilização. Entretanto, o abuso de herança pode levar a um sistema rígido. Um exemplo realista do uso de herança controlada está na criação de uma hierarquia de classes para um sistema de pedidos:

```
public abstract class Order {
    protected String orderId;
    protected LocalDate orderDate;

    public Order(String orderId, LocalDate orderDate) {
        this.orderId = orderId;
        this.orderDate = orderDate;
    }

    public abstract BigDecimal calculateTotal();
}

public class OnlineOrder extends Order {
    private BigDecimal shippingCost;

    public OnlineOrder(String orderId, LocalDate orderDate, BigDecimal shippingCost) {
        super(orderId, orderDate);
        this.shippingCost = shippingCost;
    }

    @Override
    public BigDecimal calculateTotal() {
        return shippingCost.add(new BigDecimal("100.00")); // Simulação de cálculo de valor
    }
}
```

A classe abstrata `Order` encapsula a lógica comum de pedidos, enquanto `OnlineOrder` especializa o cálculo de valores, permitindo flexibilidade para futuras expansões. Esse tipo de arquitetura é amplamente utilizado em sistemas como **Amazon** e **eBay** para lidar com diferentes tipos de pedidos (online, físicos, etc.).

## 3. Polimorfismo

O polimorfismo permite que classes diferentes implementem o mesmo método de maneiras distintas. Ele é fundamental para manter a flexibilidade no design. Um exemplo prático em sistemas de notificação seria:

```
public interface Notifier {
    void sendNotification(String message);
}

public class EmailNotifier implements Notifier {
    @Override
    public void sendNotification(String message) {
        System.out.println("Sending email: " + message);
    }
}
```



```
public class SmsNotifier implements Notifier {  
    @Override  
    public void sendNotification(String message) {  
        System.out.println("Sending SMS: " + message);  
    }  
}
```

Com o uso da interface `Notifier`, o sistema pode decidir dinamicamente se deve enviar notificações via e-mail ou SMS. Um exemplo desse padrão pode ser visto em plataformas como **Twilio**, que utiliza diferentes canais de comunicação para notificar usuários.

#### 4. Abstração

A abstração permite que apenas os detalhes essenciais de uma classe sejam expostos, mantendo o restante da implementação escondido. No desenvolvimento de sistemas modulares, a abstração é crucial para separar os contratos de comportamento da implementação real:

```
public interface PaymentProcessor {  
    void processPayment(BigDecimal amount);  
}  
  
public class StripePaymentProcessor implements PaymentProcessor {  
    @Override  
    public void processPayment(BigDecimal amount) {  
        // Código de integração com a API da Stripe  
        System.out.println("Processing payment of " + amount + " via Stripe");  
    }  
}
```

Aqui, `PaymentProcessor` define um contrato para o processamento de pagamentos, enquanto a classe `StripePaymentProcessor` implementa a lógica específica para a Stripe. Essa abstração é fundamental em sistemas de pagamento como o **Shopify**, que integra múltiplos provedores de pagamento através de APIs.

## Boas Práticas para Desenvolvimento de Software Escalável

Além de entender os pilares da orientação a objetos, é vital aplicar práticas de desenvolvimento que garantam a escalabilidade e a eficiência de sistemas complexos.

### 1. Design Modular e Clean Code

A modularidade é uma característica essencial de sistemas escaláveis. O **Java Platform Module System (JPMS)**, introduzido no Java 9 e aperfeiçoado no Java 17, permite criar sistemas altamente modulares, controlando dependências entre módulos. Um exemplo simples de criação de módulos:



```
module payment.module {  
    exports com.example.payment;  
}
```

Aqui, o módulo `payment.module` expõe o pacote `com.example.payment` para outros módulos, garantindo controle sobre o que é visível e evitando dependências indesejadas. Este conceito é amplamente utilizado em sistemas como o **Netflix**, que depende de uma arquitetura de microservices modular.

## 2. Uso de Padrões de Projeto (Design Patterns)

A adoção de padrões de projeto melhora a manutenibilidade e flexibilidade do software. Um exemplo comum é o **Factory Method**, frequentemente utilizado para criar instâncias de classes sem acoplar a lógica do cliente à implementação concreta:

```
public interface DatabaseConnection {  
    void connect();  
}  
  
public class MySQLConnection implements DatabaseConnection {  
    @Override  
    public void connect() {  
        System.out.println("Connecting to MySQL");  
    }  
}  
  
public class DatabaseConnectionFactory {  
    public static DatabaseConnection getConnection(String type) {  
        if (type.equals("MySQL")) {  
            return new MySQLConnection();  
        }  
        throw new IllegalArgumentException("Unknown database type");  
    }  
}
```

Esse padrão facilita a expansão, permitindo que novos tipos de conexões de banco de dados sejam adicionados sem alterar o código existente. É amplamente usado em frameworks como o **Spring**, que cria beans dinamicamente.

## 3. Gerenciamento Eficiente de Recursos

Gerenciar corretamente recursos como memória e threads é essencial para escalabilidade. No Java 17, o **ZGC (Z Garbage Collector)** é um aprimoramento crucial, garantindo baixa latência para aplicações que demandam alta disponibilidade, como as utilizadas pelo **Uber**. Exemplo de configuração de ZGC:

```
bash  
java -XX:+UseZGC -Xmx10g -jar myapp.jar
```



Aqui, o ZGC é ativado para uma aplicação Java, proporcionando coleta de lixo com baixa interrupção, fundamental para sistemas de tempo real.

#### 4. Concorrência e Paralelismo

O uso eficiente de threads é crítico para sistemas escaláveis. O Java oferece o pacote `java.util.concurrent` para lidar com concorrência, e o **Project Loom** promete simplificar ainda mais esse cenário com **fibers**. Exemplo de uso de `ExecutorService` para executar tarefas em paralelo:

```
ExecutorService executor = Executors.newFixedThreadPool(10);  
for (int i = 0; i < 100; i++) {  
    executor.submit(() -> {  
        System.out.println("Processing task in thread: " + Thread.currentThread().getName());  
    });  
}  
executor.shutdown();
```

Plataformas como o **LinkedIn** usam intensivamente essas técnicas para gerenciar milhões de solicitações simultâneas.

### Conclusão

O Java 17 oferece um ecossistema maduro e otimizado para a construção de sistemas orientados a objetos escaláveis. A adoção de boas práticas como modularidade, padrões de projeto e gerenciamento eficiente de recursos garante que o software seja sustentável, eficiente e preparado para crescer com as demandas do mercado. Exemplos de empresas como **Netflix**, **Stripe** e **Uber** demonstram na prática como a combinação desses princípios resulta em sistemas resilientes e escaláveis.

***EducaCiência FastCode para a comunidade***