Guia Completo de Orientação a Objetos em Java 17

Agosto 2025

Guia detalhado sobre Programação Orientada a Objetos em Java 17 com exemplos práticos e exercícios

Sumário

1 Introdução

A Programação Orientada a Objetos (POO) é um paradigma de programação que organiza o código em torno de objetos, que combinam dados e comportamentos. Java, uma linguagem fortemente orientada a objetos, implementa os conceitos de POO de forma robusta. Este guia explora os princípios de POO no contexto do Java 17, incluindo exemplos práticos, exercícios e melhores práticas.

1.1 Objetivo do Guia

Este documento tem como objetivo:

- Explicar os quatro pilares da POO: Encapsulamento, Herança, Polimorfismo e Abstração.
- Demonstrar a implementação desses conceitos em Java 17.
- Fornecer exemplos práticos e exercícios para reforçar o aprendizado.
- Apresentar recursos modernos do Java 17, como records e sealed classes.

2 Princípios de Orientação a Objetos

2.1 Encapsulamento

Encapsulamento protege os dados de uma classe, expondo apenas o necessário por meio de métodos públicos. Em Java, isso é feito com modificadores de acesso (`private`, `protected`, `public`) e getters/setters.

```java
public class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        setAge(age);
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        if (name != null && !name.isBlank()) {
            this.name = name;
        }
    }

    public int getAge() {
        return age;
    }

```

```
    public void setAge(int age) {
        if (age >= 0) {
            this.age = age;
        }
    }
}
```

Listing 1: Exemplo de Encapsulamento

2.2 Herança

Herança permite que uma classe herde atributos e métodos de outra, promovendo reuso de código. Em Java, usa-se a palavra-chave `extends`.

```
public class Vehicle {
    protected String brand;
    protected int year;

    public Vehicle(String brand, int year) {
        this.brand = brand;
        this.year = year;
    }

    public String getDescription() {
        return brand + " (" + year + ")";
    }
}

public class Car extends Vehicle {
    private int doors;

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

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

Listing 2: Exemplo de Herança

2.3 Polimorfismo

Polimorfismo permite que objetos de diferentes classes sejam tratados como instâncias de uma classe comum, geralmente por meio de interfaces ou herança.

```
public interface Animal {
    String makeSound();
}
```

```
4
5  public class Dog implements Animal {
6      @Override
7      public String makeSound() {
8          return "Woof!";
9      }
10 }
11
12 public class Cat implements Animal {
13     @Override
14     public String makeSound() {
15         return "Meow!";
16     }
17 }
18
19 public class Main {
20     public static void main(String[] args) {
21         Animal dog = new Dog();
22         Animal cat = new Cat();
23         System.out.println(dog.makeSound()); // Woof!
24         System.out.println(cat.makeSound()); // Meow!
25     }
26 }
```

Listing 3: Exemplo de Polimorfismo

2.4 Abstração

Abstração foca em esconder detalhes de implementação, expondo apenas a funcionalidade essencial. Em Java, isso é feito com classes abstratas ou interfaces.

```
1  public abstract class Shape {
2      public abstract double calculateArea();
3  }
4
5  public class Circle extends Shape {
6      private double radius;
7
8      public Circle(double radius) {
9          this.radius = radius;
10     }
11
12     @Override
13     public double calculateArea() {
14         return Math.PI * radius * radius;
15     }
16 }
17
18 public class Rectangle extends Shape {
19     private double width;
20     private double height;
21
```

```
22      public Rectangle(double width, double height) {
23          this.width = width;
24          this.height = height;
25      }
26
27      @Override
28      public double calculateArea() {
29          return width * height;
30      }
31  }
```

Listing 4: Exemplo de Abstração

3 Recursos do Java 17 para POO

Java 17 introduz funcionalidades que aprimoram a POO, como `record` e `sealed` classes.

3.1 Records

Records são classes imutáveis para modelar dados, reduzindo código boilerplate.

```
1  public record Point(int x, int y) {
2      public double distanceTo(Point other) {
3          return Math.sqrt(Math.pow(x - other.x, 2) + Math.pow(y -
               other.y, 2));
4      }
5  }
6
7  public class Main {
8      public static void main(String[] args) {
9          Point p1 = new Point(1, 2);
10         Point p2 = new Point(4, 6);
11         System.out.println(p1.distanceTo(p2)); // 5.0
12     }
13 }
```

Listing 5: Exemplo de Record

3.2 Sealed Classes

Classes seladas restringem quais classes podem estender ou implementar uma classe ou interface.

```
1  public sealed interface Vehicle permits Car, Motorcycle {
2      String getDescription();
3  }
4
5  public final class Car implements Vehicle {
6      private String brand;
7
8      public Car(String brand) {
```

```
        this.brand = brand;
    }

    @Override
    public String getDescription() {
        return "Car: " + brand;
    }
}

public final class Motorcycle implements Vehicle {
    private String brand;

    public Motorcycle(String brand) {
        this.brand = brand;
    }

    @Override
    public String getDescription() {
        return "Motorcycle: " + brand;
    }
}
```

Listing 6: Exemplo de Sealed Class

4 Exemplo Prático: Sistema de Biblioteca

Abaixo, um sistema completo de gerenciamento de biblioteca demonstrando POO.

4.1 Classe Book (Model)

```
public class Book {
    private String isbn;
    private String title;
    private String author;
    private boolean isAvailable;

    public Book(String isbn, String title, String author) {
        this.isbn = isbn;
        this.title = title;
        this.author = author;
        this.isAvailable = true;
    }

    public String getIsbn() { return isbn; }
    public String getTitle() { return title; }
    public String getAuthor() { return author; }
    public boolean isAvailable() { return isAvailable; }
    public void setAvailable(boolean available) {
       this.isAvailable = available; }

```

```
20      @Override
21      public String toString() {
22          return title + " by " + author + " (ISBN: " + isbn + ")";
23      }
24  }
```

Listing 7: Classe Book.java

4.2 Interface Library

```
1  public interface Library {
2      void addBook(Book book);
3      Book findBookByIsbn(String isbn);
4      boolean borrowBook(String isbn);
5      boolean returnBook(String isbn);
6  }
```

Listing 8: Interface Library.java

4.3 Classe LibraryImpl

```
1  import java.util.HashMap;
2  import java.util.Map;
3
4  public class LibraryImpl implements Library {
5      private Map<String, Book> books;
6
7      public LibraryImpl() {
8          this.books = new HashMap<>();
9      }
10
11     @Override
12     public void addBook(Book book) {
13         if (book != null && !books.containsKey(book.getIsbn())) {
14             books.put(book.getIsbn(), book);
15         }
16     }
17
18     @Override
19     public Book findBookByIsbn(String isbn) {
20         return books.get(isbn);
21     }
22
23     @Override
24     public boolean borrowBook(String isbn) {
25         Book book = books.get(isbn);
26         if (book != null && book.isAvailable()) {
27             book.setAvailable(false);
28             return true;
29         }
30         return false;
```

```
    }

    @Override
    public boolean returnBook(String isbn) {
        Book book = books.get(isbn);
        if (book != null && !book.isAvailable()) {
            book.setAvailable(true);
            return true;
        }
        return false;
    }
}
```

Listing 9: Classe LibraryImpl.java

4.4 Classe Main

```
public class Main {
    public static void main(String[] args) {
        Library library = new LibraryImpl();
        Book book1 = new Book("12345", "Java Programming", "John
            Doe");
        Book book2 = new Book("67890", "OOP Design", "Jane
            Smith");

        library.addBook(book1);
        library.addBook(book2);

        System.out.println("Borrowing: " + book1.getTitle());
        library.borrowBook("12345");
        System.out.println("Book available: " +
            book1.isAvailable()); // false

        System.out.println("Returning: " + book1.getTitle());
        library.returnBook("12345");
        System.out.println("Book available: " +
            book1.isAvailable()); // true
    }
}
```

Listing 10: Classe Main.java

5 Exercícios

5.1 Exercício 1: Sistema de Banco

Crie um sistema bancário com as seguintes especificações:

- Classe `Account` com atributos `accountNumber`, `balance` e `owner`.
- Métodos para depósito, saque e consulta de saldo.

- Encapsulamento para proteger o saldo.
- Classe `Bank` para gerenciar múltiplas contas.

Solução Parcial:

```java
public class Account {
    private String accountNumber;
    private double balance;
    private String owner;

    public Account(String accountNumber, String owner, double
       initialBalance) {
        this.accountNumber = accountNumber;
        this.owner = owner;
        this.balance = initialBalance >= 0 ? initialBalance : 0;
    }

    public void deposit(double amount) {
        if (amount > 0) {
            balance26 balance += amount;
        }
    }

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

    // Getters
    public String getAccountNumber() { return accountNumber; }
    public double getBalance() { return balance; }
    public String getOwner() { return owner; }
}
```

Listing 11: Solução Parcial do Exercício 1

5.2 Exercício 2: Sistema de Veículos

Crie uma hierarquia de classes para veículos:

- Interface `Vehicle` com método `startEngine()`.
- Classes `Car` e `Truck` implementando a interface.
- Use polimorfismo para chamar `startEngine()` em uma lista de veículos.

5.3 Exercício 3: Sistema de Pagamento

Crie um sistema de pagamento com:

- Classe abstrata `Payment` com método abstrato `processPayment()`.
- Classes `CreditCardPayment` e `BankTransferPayment`.
- Use records para armazenar dados de transação.

6 Melhores Práticas

- **Nomenclatura Clara**: Use nomes descritivos para classes, métodos e variáveis.
- **Encapsulamento Forte**: Sempre use modificadores de acesso apropriados.
- **Single Responsibility Principle**: Cada classe deve ter uma única responsabilidade.
- **Use Interfaces**: Prefira interfaces para definir contratos e promover polimorfismo.
- **Imutabilidade**: Considere usar `record` para dados imutáveis.
- **Documentação**: Use JavaDoc para documentar classes e métodos.

7 Conclusão

A Programação Orientada a Objetos em Java 17 oferece ferramentas poderosas para criar sistemas robustos e escaláveis. Este guia cobriu os pilares da POO, recursos modernos como records e sealed classes, exemplos práticos e exercícios para consolidar o aprendizado. A prática contínua e a aplicação de boas práticas são essenciais para dominar esses conceitos.

8 Referências

- https://www.oracle.com/java/technologies/javase/17/
- Bloch, Joshua. *Effective Java*, 3rd Edition, 2018.
- https://docs.oracle.com/en/java/javase/17/docs/api/