# Curso Java backend

## Módulo II – Java Intermediário e avançado

## Sumário

# Introdução à Orientação a Objetos

**Orientação a Objetos (OO)** é um paradigma de programação que organiza o design de software em torno de "objetos" em vez de ações, e dados em vez de lógica. Os objetos são instâncias de "classes", que podem ser pensadas como moldes ou modelos que definem as características e comportamentos dos objetos.

## Principais Conceitos da Orientação a Objetos

### Classes e Objetos

**Classe:** Uma classe é uma estrutura que define um tipo de objeto, especificando os dados que ele contém (atributos) e as operações que ele pode realizar (métodos). Pense em uma classe como uma planta ou um projeto.

**Objeto:** Um objeto é uma instância de uma classe. Ele representa uma entidade específica que possui um estado e comportamento definidos pela classe. Por exemplo, se "Carro" é uma classe, então um "Carro específico" é um objeto.

### Encapsulamento

O encapsulamento é o princípio de esconder os detalhes internos de um objeto e expor apenas o que é necessário. Isso é feito usando modificadores de acesso (como private, protected e public) para controlar o acesso aos atributos e métodos de uma classe. O encapsulamento melhora a modularidade e facilita a manutenção do código.

### Herança

A herança permite que uma classe derive de outra classe, herdando seus atributos e métodos. A classe que herda é chamada de classe "derivada" ou "subclasse", e a classe da qual herda é chamada de classe "base" ou "superclasse". Isso promove a reutilização de código e estabelece uma hierarquia entre classes. Por exemplo, uma classe Veiculo pode ser a superclasse de Carro e Moto.

### Polimorfismo

O polimorfismo permite que objetos de diferentes classes sejam tratados como objetos da mesma classe base, particularmente quando eles compartilham uma interface comum. Existem dois tipos principais de polimorfismo: sobrecarga de métodos (métodos com o mesmo nome, mas parâmetros diferentes dentro da mesma classe) e sobrescrita de métodos (uma subclasse fornece uma implementação específica de um método que já está definido em sua superclasse).

### Abstração

A abstração é o processo de simplificar a complexidade através da modelagem de classes com foco apenas nos atributos e comportamentos essenciais. Ela permite que os desenvolvedores criem uma interface simples para interagir com objetos, escondendo os detalhes complexos da implementação.

## Modificadores de Acesso

Os modificadores de acesso em Java controlam a visibilidade das classes, métodos e atributos. Existem três modificadores principais: private, public, e protected.

**public**

- **Descrição**: O modificador public torna o membro de classe (atributo, método, ou a própria classe) acessível a partir de qualquer outra classe.
- **Uso**:
    - Geralmente usados para métodos que precisam ser chamados por outras classes.
    - Pode ser usado para atributos que precisam ser acessados por outras classes (mas é uma boa prática usar métodos getter e setter).

**private**

- **Descrição**: O modificador private torna o membro de classe acessível apenas dentro da própria classe.
- **Uso**:
    - Usado para encapsular dados.
    - Os atributos geralmente são private e acessados via métodos getter e setter.

**protected**

- **Descrição**: O modificador protected torna o membro de classe acessível dentro do mesmo pacote e por subclasses (mesmo em pacotes diferentes).
- **Uso**:
    - Usado para permitir que subclasses acessem membros da classe base.

**Sem Modificador (Package-Private)**

- **Descrição**: Se nenhum modificador for especificado, o membro é acessível apenas dentro do mesmo pacote.
- **Uso**:
    - Útil para encapsulamento de pacotes.

## Modificador static

O modificador static é usado para indicar que um membro (atributo ou método) pertence à própria classe, e não a uma instância específica da classe.

### Atributos static

- **Descrição**: Um atributo static é compartilhado entre todas as instâncias da classe.
- **Uso**:
    - Para definir constantes.
    - Para compartilhar dados entre todas as instâncias da classe.

```java
package classesStatic;

public class Utilidades {
    public static void exibirMensagem() {
        System.out.println("Mensagem estática da classe Utilidade");
    }
}
```

```java
package classesStatic;

public class ExemploAcessandoMetodoEstatico {
    public static void main(String[] args) {
        // Acessando o método estático exibirMensagem da classe
Utilidades
        Utilidades.exibirMensagem();
    }
}
```

## Record

Record é uma nova forma concisa de declarar classes imutáveis para representar dados. Um record encapsula dados de maneira semelhante a uma classe, mas com menos boilerplate, já que ele fornece automaticamente métodos como equals(), hashCode(), toString() e métodos de acesso aos campos.

```java
package records;
public record Point(int x, int y) { }
```

```java
package records;

public class Execucao {
    public static void main(String[] args) {
        Point p = new Point(10, 20);
        System.out.println(p.x()); // Saída: 10
        System.out.println(p.y()); // Saída: 20
        System.out.println(p);     // Saída: Point[x=10, y=20]
    }
}
```

## Vantagens da Orientação a Objetos

- **Reutilização de Código:** Classes e objetos podem ser reutilizados em diferentes programas.
- **Facilidade de Manutenção:** O encapsulamento e a modularidade facilitam a manutenção e atualização do código.
- **Modelagem Natural:** OO permite que os desenvolvedores modelem o software de acordo com o mundo real, tornando mais intuitivo o design e a compreensão do sistema.
- **Flexibilidade e Extensibilidade:** O polimorfismo e a herança permitem a criação de sistemas flexíveis e extensíveis.

# List

**List** é uma interface do pacote java.util que estende a interface Collection. Ela representa uma coleção ordenada de elementos, onde os elementos podem ser acessados por sua posição (índice) na lista e duplicatas são permitidas. List é uma interface genérica e permite a criação de listas que podem conter qualquer tipo de objeto.

**Principais Características:**

- Ordenação: Mantém a ordem de inserção dos elementos.
- Acesso por Índice: Permite acessar, adicionar, remover e manipular elementos pela posição.
- Duplicatas: Permite elementos duplicados.
- Flexibilidade: É uma interface, o que significa que pode ser implementada por várias classes (por exemplo, ArrayList, LinkedList).

**Principais Métodos:**

- add(E e): Adiciona um elemento ao final da lista.
- add(int index, E element): Adiciona um elemento em uma posição específica.
- get(int index): Retorna o elemento na posição especificada.
- remove(int index): Remove o elemento na posição especificada.
- size(): Retorna o tamanho da lista.
- contains(Object o): Verifica se a lista contém o elemento especificado.

# ArrayList

ArrayList é uma classe que implementa a interface List, fornecendo uma implementação baseada em arrays dinâmicos. Ele é parte do pacote java.util e é amplamente utilizado devido à sua simplicidade e eficiência em operações de acesso e modificação.

```
package listas;

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

```java
public class ExemploList {
    public static void main(String[] args) {

        // Criando um ArrayList de Strings
        List<String> frutas = new ArrayList<>();

        // Adicionando elementos
        frutas.add("Maçã");
        frutas.add("Banana");
        frutas.add("Laranja");

        // Acessando elementos
        System.out.println("Primeira fruta: " + frutas.get(0));

        // Removendo um elemento
        frutas.remove("Banana");

        // Iterando sobre os elementos
        for (String fruta : frutas) {
            System.out.println("Fruta: " + fruta);
        }

        // Verificando o tamanho da lista
        System.out.println("Tamanho da lista: " + frutas.size());

    }
}
```

## Comparação entre List e ArrayList

- **Natureza:** List é uma interface; ArrayList é uma implementação específica dessa interface.
- **Implementações de** List**:** Além de ArrayList, outras classes como LinkedList e Vector também implementam List.
- **Uso de** ArrayList**:** Utilizado quando se deseja uma lista baseada em arrays dinâmicos, oferecendo rápido acesso aos elementos.

Em resumo, List fornece a estrutura geral para listas em Java, enquanto ArrayList fornece uma implementação específica eficiente para muitas operações comuns de listas.

# Enums

Enums em Java são um tipo especial de classe que representa um conjunto fixo de constantes. Eles são particularmente úteis quando você tem uma coleção de valores constantes que fazem sentido serem agrupados. As enums ajudam a tornar o código mais legível e menos propenso a erros, já que limitam os valores que uma variável pode assumir a um conjunto pré-definido.

Em Java, uma enum é declarada usando a palavra-chave enum e pode conter métodos, construtores e campos como uma classe regular. No entanto, os valores de uma enum são implicitamente public, static e final, o que significa que não podem ser alterados uma vez que são definidos.

```
package enums;

public enum NivelDeUrgencia {

   BAIXA(1), MEDIA(2), ALTA(3), CRITICA(4);

   private final int nivel;

   // Construtor
   private NivelDeUrgencia(int nivel) {
      this.nivel = nivel;
   }

   // Método getter
   public int getNivel() {
      return nivel;
   }
}
```

```
package enums;

public class ExemploEnum {
   public static void main(String[] args) {

      NivelDeUrgencia urgencia = NivelDeUrgencia.ALTA;

      switch (urgencia) {
         case BAIXA:
            System.out.println("Nível de urgência: BAIXA");
            break;
```

```
        case MEDIA:
            System.out.println("Nível de urgência: MEDIA");
            break;
        case ALTA:
            System.out.println("Nível de urgência: ALTA");
            break;
        case CRITICA:
            System.out.println("Nível de urgência: CRITICA");
            break;
    }

    System.out.println("Nível numérico: " + urgencia.getNivel());

  }
}
```

### Métodos Úteis

As enums em Java vêm com alguns métodos úteis:

- values(): Retorna um array contendo todas as constantes da enum na ordem em que foram declaradas.
- valueOf(String name): Retorna a constante da enum cujo nome é igual ao argumento name.

```
• package enums;

  public class ExemploEnum2 {
      public static void main(String[] args) {

          for (NivelDeUrgencia n : NivelDeUrgencia.values()) {
              System.out.println(n + " - " + n.getNivel());
          }

          NivelDeUrgencia urgencia =
  NivelDeUrgencia.valueOf("CRITICA");
          System.out.println("Urgência: " + urgencia);

      }
  }
•
```

## Set

Set é uma interface do pacote java.util que representa uma coleção que não permite elementos duplicados. Ela herda da interface Collection e adiciona restrições que impedem a inclusão de elementos repetidos. As implementações mais comuns de Set incluem HashSet, LinkedHashSet e TreeSet.

**Principais Características**:

- Elementos Únicos: Não permite elementos duplicados.
- Sem Ordem Garantida: A ordem dos elementos não é garantida (exceto em implementações específicas como LinkedHashSet e TreeSet).
- Implementações Comuns: HashSet, LinkedHashSet, TreeSet.

**Principais Implementações:**

1. HashSet:
    - Utiliza uma tabela hash para armazenar os elementos.
    - Não garante a ordem dos elementos.
    - Permite operações básicas (add, remove, contains) em tempo constante O(1).
2. LinkedHashSet:
    - Mantém a ordem de inserção dos elementos.
    - Utiliza uma tabela hash e uma lista vinculada (linked list) internamente.
    -
3. TreeSet:
    - Mantém os elementos ordenados de acordo com a ordem natural ou um comparador fornecido.
    - Baseado em uma árvore de busca binária (Red-Black tree), oferecendo operações em tempo O(log n).

```
package sets;

import java.util.HashSet;
import java.util.Set;

public class ExemploSet {
   public static void main(String[] args) {
```

```
    Set<String> frutas = new HashSet<>();

    // Adicionando elementos
    frutas.add("Maçã");
    frutas.add("Banana");
    frutas.add("Laranja");
    frutas.add("Maçã"); // Tentativa de adicionar um elemento duplicado

    // Exibindo elementos
    for (String fruta : frutas) {
      System.out.println("Fruta: " + fruta);
    }

    // Verificando se contém um elemento
    if (frutas.contains("Banana")) {
      System.out.println("A coleção contém Banana");
    }

    // Removendo um elemento
    frutas.remove("Laranja");

    // Verificando o tamanho da coleção
    System.out.println("Tamanho da coleção: " + frutas.size());
  }

}
```

```
package sets;

import java.util.LinkedHashSet;
import java.util.Set;

public class ExemploLinkedSet {
  public static void main(String[] args) {

    Set<String> frutas = new LinkedHashSet<>();

    // Adicionando elementos
    frutas.add("Maçã");
    frutas.add("Banana");
    frutas.add("Laranja");
```

```
    // Exibindo elementos (na ordem de inserção)
    for (String fruta : frutas) {
      System.out.println("Fruta: " + fruta);
    }

  }
}
```

```
package sets;

import java.util.Set;
import java.util.TreeSet;

public class ExemploTreeSet {
  public static void main(String[] args) {

    Set<String> frutas = new TreeSet<>();

    // Adicionando elementos
    frutas.add("Maçã");
    frutas.add("Banana");
    frutas.add("Laranja");

    // Exibindo elementos (em ordem natural)
    for (String fruta : frutas) {
      System.out.println("Fruta: " + fruta);
    }

  }
}
```

## Comparação entre Set e List

- **Duplicidade:** Set não permite elementos duplicados, enquanto List permite.
- **Ordenação:** A ordem dos elementos em Set não é garantida (exceto em implementações específicas), enquanto List mantém a ordem de inserção.
- **Uso de Casos:** Set é usado quando a unicidade dos elementos é necessária, e List é usado quando a ordem e duplicidade são importantes.

Em resumo, Set é uma interface que define uma coleção de elementos únicos, e suas implementações (HashSet, LinkedHashSet, TreeSet) fornecem diferentes comportamentos em termos de ordem e desempenho.

## Maps e HashMap

Mapas em Java são estruturas de dados que mapeiam chaves para valores únicos. O HashMap é uma implementação da interface Map que armazena os pares chave-valor sem garantir a ordem.

```
package maps;

import java.util.HashMap;
import java.util.Map;

public class ExemploMap {
   public static void main(String[] args) {

      Map<String, Integer> notas = new HashMap<>();
      notas.put("João", 7);
      notas.put("Maria", 8);
      notas.put("Pedro", 9);
      System.out.println("Nota de João: " + notas.get("João"));

   }
}
```

## Métodos

Métodos em Java são blocos de código que executam uma determinada tarefa e podem ser chamados em diferentes partes do programa.

```
package metodos;

public class Produto {

   private String codigo;
   private String nome;
   private Double preco;

   public Produto(String codigo, String nome, Double preco) {
```

```
        this.codigo = codigo;
        this.nome = nome;
        this.preco = preco;
    }

    public String getCodigo() {
        return codigo;
    }

    public void setCodigo(String codigo) {
        this.codigo = codigo;
    }

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public Double getPreco() {
        return preco;
    }

    public void setPreco(Double preco) {
        this.preco = preco;
    }

    @Override
    public String toString() {
        return "Produto{" +
                "codigo='" + codigo + '\'' +
                ", nome='" + nome + '\'' +
                ", preco=" + preco +
                '}';
    }
}
```

```
package metodos;

public class ExemploMetodos {
    public static void main(String[] args) {

        // Criando uma instancia de produto usando o construtor
```

```
        Produto produto = new Produto("123","Geladeira",3500.50);
        System.out.println(produto);

        // Exibindo somento o nome do produto chamando o método getNome()
        System.out.println(produto.getNome());

    }
}
```

## Trabalhando com Arquivos em Java

Em Java, a manipulação de arquivos é realizada usando as classes File, FileInputStream, FileOutputStream, BufferedReader e BufferedWriter, entre outras. Essas classes permitem ler e escrever arquivos no sistema de arquivos.

```
package arquivos;

import java.io.*;

public class ExemploArquivos {
    public static void main(String[] args) {

        // Lendo e exibindo as linhas do arquivo

        try (BufferedReader br = new BufferedReader(new
FileReader("./resources/arquivo.txt"))) {
            String linha;
            while ((linha = br.readLine()) != null) {
                System.out.println(linha);
            }
        } catch (IOException e) {
            System.out.println("Erro ao ler o arquivo: " + e.getMessage());
        }

        // Criando um novo arquivo
        try (BufferedWriter bw = new BufferedWriter(new
FileWriter("./resources/arquivo2.txt"))) {
            bw.write("Olá, mundo!");
            bw.write("\nEu estou aqui!");
        } catch (IOException e) {
            System.out.println("Erro ao escrever no arquivo: " + e.getMessage());
        }
```

```
    }

}
```

## Interfaces em Java

Interfaces em Java são utilizadas para definir um contrato que as classes devem seguir. Elas contêm apenas métodos abstratos e constantes.

```
package interfaces;
public interface Animal {
    void fazerSom();
}
package interfaces;
public class Cachorro implements Animal{
    @Override
    public void fazerSom() {
        System.out.println("Au au!");
    }
}
package interfaces;
public class Gato implements Animal{
    @Override
    public void fazerSom() {
        System.out.println("Mi Au!");
    }
}
package interfaces;
public class ExemploInterface {

    public static void main(String[] args) {

        Animal cachorro = new Cachorro();
        Animal gato = new Gato();

        cachorro.fazerSom(); // Deve imprimir "Au au!"
        gato.fazerSom(); // Deve imprimir "Miau!"
    }
}
```

# Programação Funcional

A programação funcional é um paradigma de programação que trata a computação como uma avaliação de funções matemáticas e evita alterar o estado e dados mutáveis. Isso significa que você escreve seu código de forma a definir funções que recebem valores de entrada e produzem um resultado, sem se preocupar com o estado interno do programa. Cada função é independente e retorna um resultado com base nos valores de entrada, sem modificar variáveis globais ou estados compartilhados. Isso torna o código mais previsível e fácil de entender, além de facilitar a paralelização e a execução em ambientes distribuídos.

Ela se baseia nos seguintes princípios fundamentais:

1. **Imutabilidade**: Os dados, uma vez criados, não podem ser modificados. Em vez disso, novos dados são criados a partir dos dados existentes.
2. **Funções como Cidadãos de Primeira Classe**: As funções podem ser atribuídas a variáveis, passadas como argumentos para outras funções e retornadas como resultado de outras funções.
3. **Sem Efeitos Colaterais**: As funções não devem causar efeitos colaterais, como modificar variáveis globais ou interagir com o ambiente externo.
4. **Recursão**: A recursão é frequentemente usada em programação funcional para iterar sobre estruturas de dados.

Exemplo
Neste exemplo, a função map é usada para dobrar cada número da lista e a função filter é usada para filtrar os números pares. Ambas as operações são realizadas de forma funcional, sem modificar o estado dos dados originais.

```
package programacaoFuncional;

import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class ExemploProgFuncional {
   public static void main(String[] args) {
      List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5);

      // Dobrar cada número usando map
      List<Integer> numerosDobrados = numeros.stream()
            .map(numero -> numero * 2)
            .collect(Collectors.toList());
      System.out.println("Números dobrados: " + numerosDobrados);

      // Filtrar números pares usando filter
      List<Integer> numerosPares = numeros.stream()
            .filter(numero -> numero % 2 == 0)
            .collect(Collectors.toList());
      System.out.println("Números pares: " + numerosPares);
   }
}
```

# Interface Funcional

Uma interface funcional é uma interface Java com apenas um método abstrato. Ela é usada para representar um contrato para qualquer classe que a implemente, indicando que essa classe deve fornecer uma implementação para o método abstrato da interface.

**Relação com Streams e Lambdas**

As interfaces funcionais são frequentemente usadas em conjunto com streams e lambdas em Java para permitir um código mais conciso e funcional. Aqui está como isso funciona:

1. **Lambdas**: Lambdas são expressões que podem ser usadas para criar instâncias de interfaces funcionais de forma mais concisa. Por

exemplo, a interface funcional Runnable pode ser implementada usando uma expressão lambda:

```
Runnable r = () -> System.out.println("Hello, world!");
```

Neste caso, a expressão lambda () -> System.out.println("Hello, world!") é uma implementação do método run() da interface Runnable.

2. **Streams**: Streams são sequências de elementos que suportam operações funcionais. As interfaces funcionais são comumente usadas como argumentos para as operações de stream, como map, filter, forEach, etc. Por exemplo:

```
List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5);
  numeros.stream()
  .map(numero -> numero * 2) // Usa uma lambda para dobrar cada número
    .forEach(System.out::println); // Imprime cada número resultante
```

Exemplo Completo

```java
package interfaceFuncional;

import java.util.Arrays;
import java.util.List;

public class ExemploCompleto {
    public static void main(String[] args) {

        List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5);

        // Usando uma interface funcional Predicate com uma expressão lambda
        numeros.stream()
            .filter(numero -> numero % 2 == 0) // Filtra os números pares
            .forEach(System.out::println); // Imprime os números pares

        // Usando uma interface funcional Function com uma expressão lambda
        numeros.stream()
            .map(numero -> numero * 2) // Dobra cada número
            .forEach(System.out::println); // Imprime os números dobrados
    }
}
```

## Lambda Expressions

Lambda Expressions em Java permitem definir funções anônimas de forma concisa. Elas são usadas principalmente para definir comportamentos que podem ser passados como argumentos ou armazenados em variáveis.

```
package lambda;

public class ExemploLambda {
    public static void main(String[] args) {
        // Expressão lambda para somar dois números
        OperacaoMatematica soma = (a, b) -> a + b;
        System.out.println("Soma: " + soma.operacao(5, 3));
    }
}

interface OperacaoMatematica {
    int operacao(int a, int b);
}
```

## Method reference

O method reference, ou referência a método em português, é uma forma de referenciar um método sem executá-lo. Em Java, ele é utilizado principalmente em expressões lambda e em situações em que se deseja passar um método como argumento para outro método. O method reference é uma forma mais concisa e legível de se utilizar expressões lambda quando se deseja apenas chamar um método existente, sem precisar escrever a implementação completa da expressão lambda. Ele é especialmente útil em situações em que o lambda tem apenas uma linha de código, que consiste em chamar um método.

Existem diferentes tipos de method references, dependendo do contexto e da assinatura do método que está sendo referenciado:

1. **Referência a um método estático**: NomeDaClasse::metodoEstatico Exemplo: Integer::parseInt
2. **Referência a um método de instância de um objeto particular**: nomeDoObjeto::metodoDeInstancia Exemplo: System.out::println
3. **Referência a um método de instância de um tipo específico**: NomeDaClasse::metodoDeInstancia Exemplo: String::length
4. **Referência a um construtor**: NomeDaClasse::new Exemplo: ArrayList::new

Referência a um método estático

```java
package methodReference;

public class ExemploEstatico {
    public static void main(String[] args) {
        // Method reference para o método estático Integer.parseInt
        // Equivalente a: (str) -> Integer.parseInt(str)
        Converter<String, Integer> converter = Integer::parseInt;

        Integer numero = converter.convert("123");
        System.out.println("Número convertido: " + numero);
    }

    interface Converter<F, T> {
        T convert(F from);
    }
}
```

Referência a um método de instância de um objeto particular

```java
package methodReference;

public class ExemplosPorInstancia {
    public static void main(String[] args) {
        String prefixo = "Prefixo: ";
        // Method reference para o método de instância concat da String prefixo
        // Equivalente a: (str) -> prefixo.concat(str)
        Concatenador concatenador = prefixo::concat;

        String resultado = concatenador.concat("Texto");
```

```
        System.out.println("Resultado: " + resultado);
    }

    interface Concatenador {
        String concat(String str);
    }
}
```

Referência a um método de instância de um tipo específico

```
package methodReference;

import java.util.function.Function;

public class ExemploTipoEspecifico {
    public static void main(String[] args) {
        // Cria uma instância de Function<String, Integer> usando uma expressão lambda
        Function<String, Integer> converter = str -> Integer.parseInt(str);

        // Usa a função para converter uma String em um Integer
        Integer numero = converter.apply("123");
        System.out.println("Número convertido: " + numero);
    }
}
```

# Streams

Streams em Java são uma maneira de processar coleções de elementos de forma declarativa. Eles permitem encadear operações como filtro, mapeamento e redução.

Filtro

```
package streams;

import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class ExemploStreamFiltro {
   public static void main(String[] args) {

      List<String> frutas = Arrays.asList("Maçã", "Banana", "Laranja", "Morango",
"Banana");

      // Filtrando frutas que começam com 'M'
      List<String> frutasComM = frutas.stream()
            .filter(f -> f.startsWith("M"))
            .collect(Collectors.toList());

      System.out.println("Frutas que começam com 'M': " + frutasComM);
   }
}
```

Mapeamento de Dados

```
package streams;

import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class ExemploMapeamento {

   public static void main(String[] args) {

      List<String> frutas = Arrays.asList("Maçã", "Banana", "Laranja", "Morango",
"Banana");

      // Convertendo todos os nomes de frutas para maiúsculas
      List<String> frutasMaiusculas = frutas.stream()
            .map(String::toUpperCase)
```

```
            .collect(Collectors.toList());

        System.out.println("Frutas em maiúsculas: " + frutasMaiusculas);

    }
}
```

Removendo Duplicatas

```
package streams;

import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class RemoverDuplicatas {
    public static void main(String[] args) {

        List<String> frutas = Arrays.asList("Maçã", "Banana", "Laranja", "Morango",
"Banana");

        // Removendo duplicatas
        List<String> frutasUnicas = frutas.stream()
                .distinct()
                .collect(Collectors.toList());

        System.out.println("Frutas únicas: " + frutasUnicas);
    }
}
```

Ordenação

```
package streams;

import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class ExemploOrdenacao {
    public static void main(String[] args) {
        List<String> frutas = Arrays.asList("Maçã", "Banana", "Laranja", "Morango",
"Banana");

        // Ordenando frutas em ordem alfabética
        List<String> frutasOrdenadas = frutas.stream()
                .sorted()
                .collect(Collectors.toList());
```

```
    System.out.println("Frutas ordenadas: " + frutasOrdenadas);
  }

}
```

Redução

```
package streams;

import java.util.Arrays;
import java.util.List;

public class ExemploReducao {

  public static void main(String[] args) {
    List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5);

    // Somando todos os números
    int soma = numeros.stream()
        .reduce(0, Integer::sum);

    System.out.println("Soma dos números: " + soma);
  }
}
```

## Collect

**Coletando em uma Map**

```
package streams;

import java.util.Arrays;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

public class ExemploCollectParaMap {
   public static void main(String[] args) {

      // Coletando em um map
      List<String> frutas = Arrays.asList("Maçã", "Banana", "Laranja", "Morango");

      // Coletando em um map com o nome da fruta como chave e o comprimento como valor
      Map<String, Integer> frutasMap = frutas.stream()
            .collect(Collectors.toMap(f -> f, String::length));

      System.out.println("Map de frutas: " + frutasMap);

   }
}
```

Coletando em um Set

```
package streams;

import java.util.Arrays;
import java.util.List;
import java.util.Set;
import java.util.stream.Collectors;

public class ExemploCollectParaSet {
   public static void main(String[] args) {

      List<String> frutas = Arrays.asList("Maçã", "Banana", "Laranja", "Morango", "Banana");

      // Coletando em um set (removendo duplicatas)
      Set<String> frutasSet = frutas.stream()
            .collect(Collectors.toSet());

      System.out.println("Set de frutas: " + frutasSet);
   }
}
```

# Threads

Uma **thread** é uma unidade básica de execução que pode ser gerenciada pelo sistema operacional. Em Java, uma thread permite a execução de várias partes de um programa simultaneamente. Cada thread tem seu próprio caminho de execução, mas todas compartilham os mesmos recursos, como memória.

Threads são usadas para realizar tarefas de maneira assíncrona, permitindo que programas façam múltiplas operações ao mesmo tempo, o que pode melhorar significativamente a performance de aplicações que realizam operações intensivas ou bloqueantes.

**Criando Threads em Java**

Existem duas maneiras principais de criar um thread em Java:

1. **Extender a classe** Thread
2. **Implementar a interface** Runnable

```
package usandothreads;

public class ExemploThreads {
  public static void main(String[] args) {

    MinhaThread thread1 = new MinhaThread();
    System.out.println();
    MinhaThread thread2 = new MinhaThread();

    thread1.start();
    thread2.start();

    Thread thread3 = new Thread(new MinhaRunnable());
    Thread thread4 = new Thread(new MinhaRunnable());
    System.out.println();
    thread3.start();
    thread4.start();

  }
}

class MinhaThread extends Thread {
  public void run() {
    for (int i = 0; i < 5; i++) {
```

```
        System.out.println(Thread.currentThread().getName() + " - Valor: " + i);
        try {
          Thread.sleep(1000); // Pausa por 1 segundo
        } catch (InterruptedException e) {
          e.printStackTrace();
        }
      }
    }
}

class MinhaRunnable implements Runnable {
  public void run() {
    for (int i = 0; i < 5; i++) {
      System.out.println(Thread.currentThread().getName() +
          " - Valor:  " + i);
      try {
        Thread.sleep(1000); // Pausa por 1 segundo
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
  }
}
```

### Sincronização de Threads

A sincronização é usada para controlar o acesso a recursos compartilhados por múltiplas threads. Isso previne problemas como condições de corrida (race conditions), onde a saída do programa pode variar dependendo de como as threads interagem.

Exemplo: Sincronização com synchronized

```
package usandothreads;

public class Sincronizacao{
  public static void main(String[] args) {

    Contador contador = new Contador();
    Thread thread1 = new Thread(new MinhaRunnableSync(contador));
    Thread thread2 = new Thread(new MinhaRunnableSync(contador));

    thread1.start();
    thread2.start();
```

```
        try {
            thread1.join();
            thread2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Contador final: " + contador.getCount());

    }
}

class Contador {
    private int count = 0;
    public synchronized void increment() {
        count++;
    }
    public synchronized int getCount() {
        return count;
    }
}
class MinhaRunnableSync implements Runnable {
    private Contador contador;

    public MinhaRunnableSync(Contador contador) {
        this.contador = contador;
    }

    @Override
    public void run() {

        for (int i = 0; i < 1000; i++) {
            contador.increment();
        }
    }
}
```

### Comunicação Entre Threads: wait e notify

As threads podem se comunicar entre si usando os métodos wait(), notify(), e notifyAll(). Estes métodos devem ser usados dentro de um bloco sincronizado.

```
package usandothreads;

public class Comunicacao {
  public static void main(String[] args) throws InterruptedException {

    Processador processador = new Processador();

    Thread threadProdutor = new Thread(() -> {
      try {
        processador.produzir();
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    });

    Thread threadConsumidor = new Thread(() -> {
      try {
        processador.consumir();
      }
      catch (InterruptedException e) {
        e.printStackTrace();
      }
    });

    threadProdutor.start();
    threadConsumidor.start();
    threadProdutor.join();
    threadConsumidor.join();
  }
}


class Processador {

  public void produzir() throws InterruptedException {
    synchronized (this) {
      System.out.println("Produtor está produzindo...");
      wait();
      System.out.println("Produtor retomou.");
    }
```

```
    }

    public void consumir() throws InterruptedException {
        Thread.sleep(2000);
        synchronized (this) {
            System.out.println("Consumidor está consumindo...");
            notify();
            Thread.sleep(1000);
        }
    }
}
```

## Annotations

Annotations em Java fornecem metadados adicionais sobre classes, métodos, variáveis e outros elementos do código. Elas são usadas para fornecer informações ao compilador ou ao tempo de execução.

Criando anotação

```
package anotacoes;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface AnnotacaoTeste {
    String descricao();
}
```

Classe que utiliza a anotação

```
package anotacoes;
public class UsandoAnotacaoTeste {
    @AnnotacaoTeste(descricao = "Este é o primeiro método de teste.")
    public void metodoTeste1() {
        System.out.println("Executando metodoTeste1");
    }
    @AnnotacaoTeste(descricao = "Este é o segundo método de teste.")
    public void metodoTeste2() {
        System.out.println("Executando metodoTeste2");
    }
    public void metodoNormal() {
```

```
    System.out.println("Executando metodoNormal");
  }
}
```

Executando usando reflexão

```
package anotacoes;

import java.lang.reflect.Method;
public class Execucao {
  public static void main(String[] args) {

    UsandoAnotacaoTeste minhaClasse = new UsandoAnotacaoTeste();
    Class<?> classe = minhaClasse.getClass();

    for (Method metodo : classe.getDeclaredMethods()) {
      if (metodo.isAnnotationPresent(AnnotacaoTeste.class)) {
        AnnotacaoTeste anotacao = metodo.getAnnotation(AnnotacaoTeste.class);
        System.out.println("Descrição do método " + metodo.getName() + ": " +
anotacao.descricao());
        try {
          metodo.invoke(minhaClasse); // Invoca o método anotado
        } catch (Exception e) {
          e.printStackTrace();
        }
      }
    }
  }
}
```

## Generics

Generics em Java permitem que classes e métodos operem em tipos especificados como parâmetros. Eles fornecem maior flexibilidade e segurança de tipos.

Classe

```
package generics;
public class Caixa<T> {
  private T conteudo;

  public void setConteudo(T conteudo) {
    this.conteudo = conteudo;
  }
  public T getConteudo() {
```

```
        return conteudo;
    }
}
```

Execução

```
package generics;

public class Execucao {
    public static void main(String[] args) {

        Caixa<Integer> caixaInteiro = new Caixa<>();
        caixaInteiro.setConteudo(10);
        System.out.println("Conteúdo da caixa: " + caixaInteiro.getConteudo());

    }
}
```

# API REST

## O que é uma API?

API (Application Programming Interface) é um conjunto de definições e protocolos que permitem que diferentes aplicações se comuniquem entre si. Uma API define a maneira como as funcionalidades de um software podem ser utilizadas por outras aplicações, através de uma série de comandos e respostas padronizadas.

## O que é REST?

REST (Representational State Transfer) é um estilo de arquitetura para sistemas distribuídos, particularmente para a web. Foi introduzido por Roy Fielding em sua tese de doutorado em 2000. REST define um conjunto de restrições e propriedades baseadas em HTTP, que são usadas para criar APIs leves e escaláveis.

**Princípios de uma API RESTful**

Para que uma API seja considerada RESTful, ela deve aderir a certos princípios:

1. **Cliente-Servidor**: A arquitetura deve separar a interface do cliente da lógica de servidor. Isso permite que ambos evoluam de forma independente.
2. **Stateless**: Cada chamada de API deve ser independente e não deve depender do estado mantido no servidor. Todas as informações necessárias para processar a solicitação devem ser incluídas na própria solicitação.
3. **Cacheable**: As respostas da API devem ser explicitamente rotuladas como cacheáveis ou não-cacheáveis. Isso melhora a escalabilidade e desempenho, permitindo que os clientes armazenem respostas reutilizáveis.
4. **Interface Uniforme**: A interface deve ser consistente e padronizada, utilizando os métodos HTTP (GET, POST, PUT, DELETE) de maneira convencional.
5. **Sistema em Camadas**: A arquitetura deve ser composta por camadas hierárquicas, onde cada camada é responsável por uma funcionalidade específica, sem que as camadas adjacentes saibam da implementação das outras.
6. **Código Sob Demanda (Opcional)**: A funcionalidade pode ser estendida enviando código executável do servidor para o cliente, quando necessário.

**Componentes de uma API RESTful**

1. **Recursos**: Tudo em REST é considerado um recurso. Recursos são identificados por URIs (Uniform Resource Identifiers).
2. **Representações**: Um recurso pode ter múltiplas representações, como JSON, XML, HTML, etc. A representação é o formato específico em que o recurso é transferido entre cliente e servidor.
3. **Verbos HTTP**: REST usa métodos HTTP para realizar operações sobre os recursos:
    - **GET**: Recupera a representação de um recurso.
    - **POST**: Cria um novo recurso.
    - **PUT**: Atualiza um recurso existente.
    - **DELETE**: Remove um recurso.
4. **Status HTTP**: Respostas de uma API RESTful incluem códigos de status HTTP que indicam o resultado da solicitação (ex: 200 OK, 404 Not Found, 500 Internal Server Error).

**Documentação de API com OpenAPI**

O OpenAPI é uma especificação que permite descrever APIs REST de forma detalhada, incluindo endpoints, parâmetros, respostas e exemplos. Ele é usado para criar documentação automatizada e ferramentas de geração de código para várias linguagens de programação.

Observações:
Iremos criar uma API REST completa e documentada com OpenAPI no nosso projeto no Módulo III.

# Testes Unitários

## O que são Testes Unitários?

Testes unitários são testes automatizados que verificam o funcionamento de partes específicas do código, geralmente unidades individuais, como métodos ou classes. Eles são usados para garantir que cada unidade de código funcione corretamente de forma isolada.

**JUnit**

JUnit é um framework de teste unitário para Java. Ele fornece anotações e métodos para facilitar a escrita e execução de testes unitários.

Exemplo de teste unitário com JUnit:

```
package testes;

public class MinhaClasse {
   public int multiplicar(int a, int b) {
      return a * b;
   }
}
package testes;

import org.junit.Test;

import static org.junit.Assert.assertEquals;
public class MinhaClasseTest {
   @Test
   public void testarMetodo() {
      MinhaClasse minhaClasse = new MinhaClasse();
      int resultado = minhaClasse.multiplicar(2, 3);
      assertEquals(6, resultado);
   }
}
```

### Mockito

Mockito é um framework de mock para Java que permite criar objetos simulados para testar o comportamento de classes dependentes.

Observação:
Veremos mais a fundo testes unitários no módulo III.

## Design Patterns

Design Patterns são soluções para problemas comuns de design de software. Eles fornecem um modelo para resolver problemas de forma eficiente e reutilizável.

### Singleton

O Singleton é um padrão de design que garante que uma classe tenha apenas uma instância e fornece um ponto global de acesso a essa instância.

```
package designPattern.singleton;
public class Singleton {

  // Instância única da classe Singleton
  private static Singleton instanciaUnica;

  // Construtor privado para evitar instâncias diretas
  private Singleton() {}

  // Método público estático para obter a instância única
  public static Singleton getInstancia() {
    if (instanciaUnica == null) {
      instanciaUnica = new Singleton();
    }
    return instanciaUnica;
  }

  // Método de exemplo para demonstrar o funcionamento do Singleton
  public void mostrarMensagem() {
    System.out.println("Instância única do Singleton.");
  }
}
package designPattern.singleton;
public class Execucao {
  public static void main(String[] args) {
```

```
        // Obter a instância única do Singleton
        Singleton singleton1 = Singleton.getInstancia();
        Singleton singleton2 = Singleton.getInstancia();

        // Chamar um método da instância Singleton
        singleton1.mostrarMensagem();

        // Verificar se as duas referências apontam para a mesma instância
        System.out.println("singleton1 == singleton2? " + (singleton1 == singleton2));
    }
}
```

### Factory Method

O Factory Method é um padrão de design que define uma interface para criar um objeto, mas permite que as subclasses decidam qual classe instanciar. Ele é útil quando você precisa encapsular a lógica de criação de objetos.

```
package designPattern.factotyMethod;
public interface Produto {
    void usar();
}
package designPattern.factotyMethod;
public class ProdutoConcretoA implements Produto{
    @Override
    public void usar() {
        System.out.println("Usando ProdutoConcretoA.");
    }
}
package designPattern.factotyMethod;
public class ProdutoConcretoB implements Produto{
    @Override
    public void usar() {
        System.out.println("Usando ProdutoConcretoB.");
    }
}
package designPattern.factotyMethod;
public abstract class Criador {
    public abstract Produto factoryMethod();
    public void algumMetodo() {
        Produto produto = factoryMethod();
        produto.usar();
```

```
    }
}
package designPattern.factotyMethod;
public class CriadorConcretoA extends Criador{
    @Override
    public Produto factoryMethod() {
        return new ProdutoConcretoA();
    }
}
package designPattern.factotyMethod;
public class CriadorConcretoB extends Criador{
    @Override
    public Produto factoryMethod() {
        return new ProdutoConcretoB();
    }
}
```

Execução

```
package designPattern.factotyMethod;
public class Execucao {
    public static void main(String[] args) {
        Criador criadorA = new CriadorConcretoA();
        Criador criadorB = new CriadorConcretoB();

        criadorA.algumMetodo();  // Deve usar ProdutoConcretoA
        criadorB.algumMetodo();  // Deve usar ProdutoConcretoB
    }
}
```

**Observer**

O Observer é um padrão de design que define uma dependência um-para-muitos entre objetos, de modo que quando um objeto muda de estado, todos os seus dependentes são notificados e atualizados automaticamente.

```
package designPattern.observer;
public interface Observer {
    void update(String message);
}
package designPattern.observer;
public interface Subject {
    void registerObserver(Observer observer);
```

```
    void removeObserver(Observer observer);
    void notifyObservers();
}
package designPattern.observer;

import java.util.ArrayList;
import java.util.List;

public class ConcreteSubject implements Subject {
    private List<Observer> observers;
    private String state;
    public ConcreteSubject() {
        observers = new ArrayList<>();
    }
    public void setState(String state) {
        this.state = state;
        notifyObservers();
    }
    public String getState() {
        return state;
    }
    @Override
    public void registerObserver(Observer observer) {
        observers.add(observer);
    }
    @Override
    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }
    @Override
    public void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(state);
        }
    }
}
package designPattern.observer;
public class ConcreteObserver implements Observer {
    private String name;
    public ConcreteObserver(String name) {
        this.name = name;
    }
    @Override
    public void update(String message) {
        System.out.println("Observador " + name + " recebeu a mensagem: " + message);
    }
}
```

Execução

```
package designPattern.observer;

public class Execucao {
   public static void main(String[] args) {
      ConcreteSubject subject = new ConcreteSubject();

      Observer observer1 = new ConcreteObserver("Observer 1");
      Observer observer2 = new ConcreteObserver("Observer 2");

      subject.registerObserver(observer1);
      subject.registerObserver(observer2);

      subject.setState("Estado 1");
      subject.setState("Estado 2");

      subject.removeObserver(observer1);

      subject.setState("Estado 3");
   }
}
```

**DTO (Data Transfer Object)**

O padrão DTO é usado para transferir dados entre camadas de um aplicativo. Ele é especialmente útil para encapsular dados que precisam ser transferidos entre a camada de apresentação e a camada de serviço ou entre diferentes serviços.

**Repository**

O padrão Repository é usado para encapsular a lógica de acesso aos dados e fornecer uma interface para a persistência e recuperação de objetos de domínio. Ele separa a lógica de persistência do restante do aplicativo, facilitando o teste e a manutenção do código.

Observação:
Veremos exemplos dos padrões DTO e REPOSITORY durante o desenvolvimento do nosso projeto no módulo III.

# Padrões de Projetos

**DDD (Domain-Driven Design)**

O Domain-Driven Design (DDD) é uma abordagem de design de software que enfatiza a colaboração entre especialistas de domínio e desenvolvedores para criar um modelo de domínio rico e expressivo. Ele se concentra em identificar conceitos-chave do domínio e expressá-los no código de forma clara e concisa.

Principais conceitos do DDD:

- **Entidades**: Objetos que possuem identidade e são distinguíveis por essa identidade ao longo do tempo.
- **Value Objects**: Objetos que representam um valor, não uma identidade, e são comparados com base em seus atributos.
- **Agregados**: Grupos de entidades e value objects que são tratados como uma única unidade transacional.
- **Repositórios**: Abstrações que permitem recuperar e armazenar entidades.

**TDD (Test-Driven Development)**

O Test-Driven Development (TDD) é uma prática de desenvolvimento de software que se baseia em um ciclo curto de repetições: escrever um teste, fazer o teste falhar, escrever o código mínimo necessário para passar no teste e refatorar o código para melhorá-lo.

Benefícios do TDD:

- Código mais limpo e simples.
- Maior confiabilidade do software.
- Menor acoplamento entre os componentes.
- Melhor design de software.

**SOLID**

Os princípios SOLID são um conjunto de cinco princípios de design de software que visam tornar o software mais fácil de entender, manter e estender. Eles foram introduzidos por Robert C. Martin (Uncle Bob) e são os seguintes:

- **S - Single Responsibility Principle (Princípio da Responsabilidade Única)**: Uma classe deve ter apenas uma razão para mudar.
- **O - Open/Closed Principle (Princípio Aberto/Fechado)**: Uma classe deve estar aberta para extensão, mas fechada para modificação.
- **L - Liskov Substitution Principle (Princípio da Substituição de Liskov)**: Objetos de um subtipo devem ser substituíveis por objetos de um supertipo sem alterar a integridade do programa.
- **I - Interface Segregation Principle (Princípio da Segregação de Interfaces)**: Clientes não devem ser forçados a depender de interfaces que não utilizam.
- **D - Dependency Inversion Principle (Princípio da Inversão de Dependência)**: Módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações.

**CQRS (Command Query Responsibility Segregation)**

O CQRS é um padrão arquitetural que sugere separar a operação de leitura (query) da operação de escrita (command) em um sistema. Ele propõe que os modelos de leitura e escrita sejam diferentes para otimizar o desempenho e a escalabilidade.

Principais conceitos do CQRS:

- **Comandos (Commands)**: Solicitações para alterar o estado do sistema.
- **Consultas (Queries)**: Solicitações para obter informações do sistema.
- **Modelo de Domínio de Escrita (Write Model)**: Modelo otimizado para operações de escrita.
- **Modelo de Leitura (Read Model)**: Modelo otimizado para operações de leitura.

## Conclusão

Os padrões de projetos como DDD, TDD, SOLID e CQRS são fundamentais para o desenvolvimento de software de alta qualidade. Eles ajudam a criar sistemas robustos, flexíveis e fáceis de manter, e são amplamente utilizados na indústria de desenvolvimento de software.

## Arquitetura Hexagonal (Ports and Adapters)

A Arquitetura Hexagonal, também conhecida como Arquitetura de Ports and Adapters (ou Ports and Adapters Architecture), é um estilo arquitetural que enfatiza a separação de preocupações e a independência das camadas em um sistema. Essa arquitetura é projetada para facilitar a testabilidade, manutenção e evolução do software.

### Princípios da Arquitetura Hexagonal

1. **Independência da Interface do Usuário**: A interface do usuário é tratada como um adaptador externo, separado da lógica de negócio.
2. **Independência de Frameworks**: O núcleo da aplicação (domínio) não depende de frameworks externos, facilitando a troca de frameworks sem modificar a lógica da aplicação.
3. **Testabilidade**: A arquitetura é altamente testável, pois a lógica de negócio é isolada e pode ser testada independentemente das interfaces de entrada e saída.
4. **Flexibilidade e Manutenção**: A separação de camadas permite que cada uma possa ser modificada ou substituída sem afetar as outras, facilitando a evolução e manutenção do sistema.

**Componentes da Arquitetura Hexagonal**

1. **Domínio**: Contém as regras de negócio e os objetos de domínio da aplicação. É a parte central e mais importante da arquitetura.
2. **Camada de Aplicação**: Responsável por orquestrar as operações do domínio e interagir com as interfaces de entrada e saída.
3. **Interfaces de Entrada e Saída (Adapters)**: São as interfaces com o mundo externo. As interfaces de entrada recebem as solicitações do usuário, enquanto as interfaces de saída lidam com a persistência e comunicação externa.
4. **Testes**: A arquitetura Hexagonal facilita a escrita de testes, pois permite que os testes se concentrem apenas na lógica de negócio, sem se preocupar com detalhes de implementação ou infraestrutura.

Observação:
O nosso projeto do módulo III, será estrutura utilizando arquitetura hexagonal.

**Conclusão**

A Arquitetura Hexagonal é uma abordagem flexível e eficaz para projetar sistemas de software, permitindo a separação clara de responsabilidades e facilitando a evolução e manutenção do sistema. Ao adotar essa arquitetura, os desenvolvedores podem criar sistemas mais testáveis, flexíveis e fáceis de manter.