Guia de Treinamento Java 6 para Programadores

1. Introdução ao Java 6

A plataforma Java, desde sua concepção pela Sun Microsystems (agora Oracle), tem sido um pilar fundamental no desenvolvimento de software, oferecendo uma abordagem robusta e portável para a criação de aplicações. O Java 6, lançado em 2006, codinome Mustang, representou um marco significativo, introduzindo melhorias de desempenho, novas APIs e aprimoramentos na experiência do desenvolvedor. Este guia se aprofundará nas capacidades do Java 6, fornecendo uma base sólida para programadores que desejam dominar a linguagem e suas aplicações.

1.1. Visão Geral da Plataforma Java e JVM

A **Plataforma Java** é um conjunto de programas que permite o desenvolvimento e a execução de aplicações Java. Ela é composta principalmente pela **Java Development Kit (JDK)**, que inclui as ferramentas para desenvolvimento, e pela **Java Runtime Environment (JRE)**, que contém a **Java Virtual Machine (JVM)** e as bibliotecas de classes necessárias para executar programas Java.

A **Java Virtual Machine (JVM)** é o coração da plataforma Java. Ela é uma máquina virtual que executa o bytecode Java, um formato intermediário gerado a partir do código-fonte Java compilado. A principal vantagem da JVM é a portabilidade: o mesmo bytecode pode ser executado em qualquer sistema operacional que possua uma JVM compatível, seguindo o princípio “Write Once, Run Anywhere” (Escreva uma vez, execute em qualquer lugar).

Componente	Descrição
JDK	Java Development Kit: Conjunto de ferramentas para desenvolver aplicações Java, incluindo compilador (javac), depurador (jdb) e a JRE.
JRE	Java Runtime Environment: Ambiente de execução para aplicações Java, contendo a JVM e as bibliotecas de classes.
JVM	Java Virtual Machine: Máquina virtual que executa o bytecode Java, garantindo a portabilidade da plataforma.

1.2. Configuração do Ambiente de Desenvolvimento (JDK 6)

Para começar a programar em Java 6, é essencial configurar corretamente o ambiente de desenvolvimento. Isso envolve a instalação do JDK 6 e a configuração das variáveis de ambiente necessárias.

1. **Download do JDK 6**: O JDK 6 pode ser baixado do site oficial da Oracle (embora versões mais antigas possam exigir uma conta Oracle ou serem encontradas em arquivos históricos).
2. **Instalação**: Siga as instruções do instalador para o seu sistema operacional. Geralmente, o JDK é instalado em um diretório como `C:\Program Files\Java\jdk1.6.0_xx` (Windows) ou `/usr/lib/jvm/java-6-sun` (Linux).
3. **Configuração da Variável de Ambiente `JAVA_HOME`**: Esta variável deve apontar para o diretório raiz da instalação do JDK. Por exemplo:
   - **Windows**: `JAVA_HOME=C:\Program Files\Java\jdk1.6.0_xx`
   - **Linux/macOS**: `JAVA_HOME=/usr/lib/jvm/java-6-sun`
4. **Configuração da Variável de Ambiente `PATH`**: Adicione o diretório `bin` do JDK à variável `PATH` do sistema para que os comandos `java` e `javac` possam ser executados a partir de qualquer diretório. Por exemplo:
   - **Windows**: Adicione `;%JAVA_HOME%\bin` ao `PATH`.
   - **Linux/macOS**: Adicione `:$JAVA_HOME/bin` ao `PATH`.

Após a configuração, você pode verificar a instalação abrindo um terminal e executando os seguintes comandos:

```
java -version
javac -version
```

Ambos os comandos devem exibir informações sobre a versão 1.6.x do Java.

1.3. Estrutura Básica de um Programa Java

Um programa Java básico consiste em uma ou mais classes. A execução de um programa Java começa no método `main` de uma classe pública. Abaixo está um exemplo simples de um programa "Olá, Mundo!":

```
public class OlaMundo {
    public static void main(String[] args) {
        System.out.println("Olá, Mundo!");
    }
}
```

Explicação:

- `public class OlaMundo`: Declara uma classe pública chamada `OlaMundo`. Em Java, o nome do arquivo-fonte (`.java`) deve ser o mesmo da classe pública contida nele.
- `public static void main(String[] args)`: Este é o método principal, o ponto de entrada para a execução do programa.
  - `public`: O método é acessível de qualquer lugar.
  - `static`: O método pertence à classe, não a uma instância específica do objeto, e pode ser chamado sem criar um objeto da classe.
  - `void`: O método não retorna nenhum valor.
  - `main`: O nome padrão do método principal.
  - `String[] args`: Um array de strings que pode receber argumentos de linha de comando.
- `System.out.println("Olá, Mundo!");`: Esta linha imprime a string "Olá, Mundo!" no console. `System` é uma classe, `out` é um objeto `PrintStream`

estático dentro de `System`, e `println` é um método que imprime uma linha de texto e adiciona uma nova linha.

2. Fundamentos da Linguagem Java

Os fundamentos da linguagem Java são os blocos de construção essenciais para qualquer programador. Compreender como os dados são armazenados, manipulados e como o fluxo de execução do programa é controlado é crucial para escrever código eficaz e robusto.

2.1. Tipos de Dados e Variáveis

Java é uma linguagem **fortemente tipada**, o que significa que todas as variáveis devem ter um tipo de dado declarado antes de serem usadas. Existem dois tipos principais de dados em Java: **tipos primitivos** e **tipos de referência**.

2.1.1. Tipos Primitivos

Os tipos primitivos representam valores únicos e são armazenados diretamente na memória. Java possui oito tipos primitivos:

Tipo de Dado	Descrição	Tamanho (bits)	Intervalo de Valores
`byte`	Inteiro	8	-128 a 127
`short`	Inteiro	16	-32.768 a 32.767
`int`	Inteiro	32	-2^31 a (2^{31})-1
`long`	Inteiro	64	-2^63 a (2^{63})-1
`float`	Ponto flutuante (precisão simples)	32	Aproximadamente ±3.40282347E+38F
`double`	Ponto flutuante (precisão dupla)	64	Aproximadamente ±1.79769313486231570E+308
`char`	Caractere Unicode	16	'\u0000' a '\uffff' (0 a 65.535)
`boolean`	Valor lógico (verdadeiro/falso)	1	`true` ou `false`

Exemplo de declaração e inicialização de variáveis primitivas:

```
int idade = 30;
double preco = 99.99;
char inicial = 'J';
boolean isAtivo = true;
byte nivel = 5;
long populacaoMundial = 7800000000L; // 'L' para indicar long
float temperatura = 25.5f; // 'f' para indicar float
```

2.1.2. Tipos de Referência

Tipos de referência, como `String`, arrays e objetos de classes definidas pelo usuário, armazenam o endereço de memória onde o objeto real está localizado. Eles são mais complexos e serão abordados em detalhes na seção de Programação Orientada a Objetos.

Exemplo:

```java
String nome = "Maria";
int[] numeros = {1, 2, 3};
```

2.2. Operadores

Operadores são símbolos que realizam operações em variáveis e valores. Java suporta vários tipos de operadores:

2.2.1. Operadores Aritméticos

Usados para realizar operações matemáticas básicas.

Operador	Descrição	Exemplo
`+`	Adição	`a + b`
`-`	Subtração	`a - b`
`*`	Multiplicação	`a * b`
`/`	Divisão	`a / b`
`%`	Módulo (resto da divisão)	`a % b`
`++`	Incremento	`a++` ou `++a`
`--`	Decremento	`a--` ou `--a`

Exemplo:

```java
int x = 10;
int y = 3;

System.out.println(x + y); // 13
System.out.println(x - y); // 7
System.out.println(x * y); // 30
System.out.println(x / y); // 3 (divisão inteira)
System.out.println(x % y); // 1

x++; // x agora é 11
y--; // y agora é 2
```

2.2.2. Operadores Relacionais (Comparação)

Usados para comparar dois valores e retornar um resultado booleano (`true` ou `false`).

Operador	Descrição	Exemplo
`==`	Igual a	`a == b`
`!=`	Diferente de	`a != b`
`>`	Maior que	`a > b`
`<`	Menor que	`a < b`
`>=`	Maior ou igual a	`a >= b`
`<=`	Menor ou igual a	`a <= b`

Exemplo:

```java
int a = 5;
int b = 10;

System.out.println(a == b); // false
System.out.println(a != b); // true
System.out.println(a > b);  // false
System.out.println(a < b);  // true
```

2.2.3. Operadores Lógicos

Usados para combinar expressões booleanas.

Operador	Descrição	Exemplo
`&&`	AND lógico (curto-circuito)	`(a > 0) && (b < 10)`
`||`	OR lógico (curto-circuito)	`(a > 0) || (b < 10)`
`!`	NOT lógico	`!(a > 0)`

Exemplo:

```
boolean condicao1 = true;
boolean condicao2 = false;

System.out.println(condicao1 && condicao2); // false
System.out.println(condicao1 || condicao2); // true
System.out.println(!condicao1);             // false
```

2.2.4. Operadores Bitwise (Bit a Bit)

Operam em bits individuais de números inteiros. São menos comuns em programação diária, mas essenciais em certas otimizações ou manipulações de baixo nível.

Operador	Descrição	Exemplo
`&`	AND bit a bit	`a & b`
`|`	OR bit a bit	`a | b`
`^`	XOR bit a bit	`a ^ b`
`~`	NOT bit a bit (complemento de um)	`~a`
`<<`	Deslocamento à esquerda	`a << 2`
`>>`	Deslocamento à direita com sinal	`a >> 2`
`>>>`	Deslocamento à direita sem sinal	`a >>> 2`

Exemplo:

```
int num1 = 5;  // 0101 em binário
int num2 = 3;  // 0011 em binário

System.out.println(num1 & num2); // 1 (0001)
System.out.println(num1 | num2); // 7 (0111)
System.out.println(num1 ^ num2); // 6 (0110)
System.out.println(~num1);       // -6 (complemento de um)
System.out.println(num1 << 1);   // 10 (1010)
System.out.println(num1 >> 1);   // 2 (0010)
```

2.3. Estruturas de Controle Condicionais

As estruturas condicionais permitem que o programa tome decisões e execute diferentes blocos de código com base em certas condições.

2.3.1. `if`, `else if`, `else`

A estrutura `if` é usada para executar um bloco de código se uma condição for verdadeira. O `else if` permite testar múltiplas condições em sequência, e o `else` fornece um bloco de código a ser executado se nenhuma das condições anteriores for verdadeira.

```
int nota = 75;

if (nota >= 90) {
    System.out.println("Conceito A");
} else if (nota >= 80) {
    System.out.println("Conceito B");
} else if (nota >= 70) {
    System.out.println("Conceito C");
} else {
    System.out.println("Conceito D");
}
```

2.3.2. `switch`

A estrutura `switch` é usada para selecionar um dos muitos blocos de código a serem executados. É uma alternativa mais limpa para múltiplas condições `if-else if` quando se compara uma única variável com vários valores possíveis.

```
int diaDaSemana = 3; // 1 = Domingo, 2 = Segunda, etc.
String nomeDoDia;

switch (diaDaSemana) {
    case 1:
        nomeDoDia = "Domingo";
        break;
    case 2:
        nomeDoDia = "Segunda-feira";
        break;
    case 3:
        nomeDoDia = "Terça-feira";
        break;
    case 4:
        nomeDoDia = "Quarta-feira";
        break;
    case 5:
        nomeDoDia = "Quinta-feira";
        break;
    case 6:
        nomeDoDia = "Sexta-feira";
        break;
    case 7:
        nomeDoDia = "Sábado";
        break;
    default:
        nomeDoDia = "Dia inválido";
        break;
}
System.out.println("Hoje é " + nomeDoDia);
```

Importante: O `break` é crucial no `switch` para evitar o "fall-through", onde a execução continua para os próximos `case`s se não houver um `break`.

2.4. Estruturas de Controle de Loop

As estruturas de loop permitem executar um bloco de código repetidamente.

2.4.1. `for` Loop

O `for` loop é ideal quando o número de iterações é conhecido.

```
for (int i = 0; i < 5; i++) {
    System.out.println("Iteração " + i);
}
// Saída:
// Iteração 0
// Iteração 1
// Iteração 2
// Iteração 3
// Iteração 4
```

2.4.2. `while` Loop

O `while` loop executa um bloco de código enquanto uma condição especificada for verdadeira. A condição é avaliada antes de cada iteração.

```
int contador = 0;
while (contador < 3) {
    System.out.println("Contador: " + contador);
    contador++;
}
// Saída:
// Contador: 0
// Contador: 1
// Contador: 2
```

2.4.3. `do-while` Loop

O `do-while` loop é semelhante ao `while`, mas garante que o bloco de código seja executado pelo menos uma vez, pois a condição é avaliada após a execução do bloco.

```java
int i = 0;
do {
    System.out.println("Valor de i: " + i);
    i++;
} while (i < 0);
// Saída:
// Valor de i: 0 (executa uma vez, mesmo com a condição falsa)
```

2.4.4. `for-each` Loop (Enhanced for Loop)

Introduzido no Java 5, o `for-each` loop simplifica a iteração sobre arrays e coleções. É mais legível e menos propenso a erros de índice.

```java
int[] numeros = {10, 20, 30, 40, 50};

for (int numero : numeros) {
    System.out.println("Número: " + numero);
}
// Saída:
// Número: 10
// Número: 20
// Número: 30
// Número: 40
// Número: 50
```

2.4.5. `break` e `continue`

- `break`: Usado para sair de um loop ou de uma instrução `switch` imediatamente.
- `continue`: Usado para pular a iteração atual de um loop e prosseguir para a próxima iteração.

3. Programação Orientada a Objetos (POO) em Java

A Programação Orientada a Objetos (POO) é um paradigma de programação que organiza o design de software em torno de **objetos**, em vez de funções e lógica. Em

Java, tudo é um objeto (com exceção dos tipos primitivos), tornando a POO um conceito central para o desenvolvimento eficaz e escalável.

3.1. Classes e Objetos

- **Classe**: Uma classe é um **modelo** ou **planta** para criar objetos. Ela define as características (atributos ou campos) e os comportamentos (métodos) que os objetos criados a partir dela terão. Uma classe não ocupa espaço na memória até que um objeto seja instanciado.

```
public class Carro {
    // Atributos (variáveis de instância)
    String marca;
    String modelo;
    int ano;
    String cor;


    // Construtor
    public Carro(String marca, String modelo, int ano, String cor) {
        this.marca = marca;
        this.modelo = modelo;
        this.ano = ano;
        this.cor = cor;
    }


    // Métodos (comportamentos)
    public void ligar() {
        System.out.println("O carro " + modelo + " está ligado.");
    }


    public void acelerar() {
        System.out.println("O carro " + modelo + " está acelerando.");
    }


    public void exibirDetalhes() {
        System.out.println("Marca: " + marca + ", Modelo: " + modelo +
", Ano: " + ano + ", Cor: " + cor);
    }
}
```

- **Objeto**: Um objeto é uma **instância** de uma classe. É uma entidade real que existe na memória e possui os atributos e comportamentos definidos pela sua classe. Criar um objeto é chamado de **instanciação**.

```java
public class TesteCarro {
    public static void main(String[] args) {
        // Criação de objetos (instâncias da classe Carro)
        Carro meuCarro = new Carro("Toyota", "Corolla", 2020,
"Prata");
        Carro seuCarro = new Carro("Honda", "Civic", 2022, "Preto");


        // Acessando atributos e chamando métodos dos objetos
        meuCarro.exibirDetalhes(); // Saída: Marca: Toyota, Modelo:
Corolla, Ano: 2020, Cor: Prata
        meuCarro.ligar();         // Saída: O carro Corolla está
ligado.


        seuCarro.exibirDetalhes(); // Saída: Marca: Honda, Modelo:
Civic, Ano: 2022, Cor: Preto
        seuCarro.acelerar();      // Saída: O carro Civic está
acelerando.
    }
}
```

3.2. Encapsulamento, Herança e Polimorfismo

Estes são os três pilares fundamentais da POO, que promovem a modularidade, reusabilidade e flexibilidade do código.

3.2.1. Encapsulamento

Encapsulamento é o mecanismo de agrupar dados (atributos) e os métodos que operam nesses dados em uma única unidade (a classe), e de restringir o acesso direto a alguns dos componentes do objeto. Isso é feito usando **modificadores de acesso** (`public`, `private`, `protected`, `default`). O objetivo é proteger a integridade dos dados e expor apenas o que é necessário.

- **`private`**: Membros (atributos e métodos) declarados como `private` são acessíveis apenas dentro da própria classe.
- **`public`**: Membros `public` são acessíveis de qualquer lugar.
- **`protected`**: Membros `protected` são acessíveis dentro da própria classe, por classes no mesmo pacote e por subclasses (mesmo que em pacotes diferentes).

- `default` **(sem modificador)**: Membros com acesso `default` são acessíveis apenas por classes no mesmo pacote.

Para acessar e modificar atributos `private`, são utilizados métodos **getters** (acessores) e **setters** (modificadores).

```java
public class ContaBancaria {
    private String numeroConta;
    private double saldo;

    public ContaBancaria(String numeroConta, double saldoInicial) {
        this.numeroConta = numeroConta;
        this.saldo = saldoInicial;
    }

    // Getter para numeroConta
    public String getNumeroConta() {
        return numeroConta;
    }

    // Getter para saldo
    public double getSaldo() {
        return saldo;
    }

    // Setter para saldo (com validação)
    public void depositar(double valor) {
        if (valor > 0) {
            this.saldo += valor;
            System.out.println("Depósito de " + valor + " realizado. Novo
saldo: " + this.saldo);
        } else {
            System.out.println("Valor de depósito inválido.");
        }
    }

    public void sacar(double valor) {
        if (valor > 0 && this.saldo >= valor) {
            this.saldo -= valor;
            System.out.println("Saque de " + valor + " realizado. Novo
saldo: " + this.saldo);
        } else {
            System.out.println("Saldo insuficiente ou valor de saque
inválido.");
        }
    }
}

public class TesteConta {
    public static void main(String[] args) {
        ContaBancaria conta = new ContaBancaria("12345-6", 1000.0);
```

```
        System.out.println("Número da Conta: " + conta.getNumeroConta());
        System.out.println("Saldo Inicial: " + conta.getSaldo());

        conta.depositar(500.0);
        conta.sacar(200.0);
        conta.sacar(2000.0); // Tentativa de saque com saldo insuficiente
    }
}
```

3.2.2. Herança

Herança é um mecanismo que permite que uma classe (subclasse ou classe filha) herde atributos e métodos de outra classe (superclasse ou classe pai). Isso promove a reusabilidade de código e estabelece uma relação “é um tipo de” (is-a relationship).

- A palavra-chave `extends` é usada para indicar herança.
- Uma subclasse pode adicionar novos atributos e métodos, ou sobrescrever (override) métodos da superclasse.
- Java não suporta herança múltipla de classes (uma classe só pode estender uma única superclasse), mas suporta herança múltipla de interfaces.

```java
// Superclasse
public class Animal {
    String nome;

    public Animal(String nome) {
        this.nome = nome;
    }

    public void comer() {
        System.out.println(nome + " está comendo.");
    }

    public void dormir() {
        System.out.println(nome + " está dormindo.");
    }
}

// Subclasse que herda de Animal
public class Cachorro extends Animal {
    String raca;

    public Cachorro(String nome, String raca) {
        super(nome); // Chama o construtor da superclasse
        this.raca = raca;
    }

    public void latir() {
        System.out.println(nome + " está latindo.");
    }

    // Sobrescrevendo o método comer da superclasse
    @Override
    public void comer() {
        System.out.println(nome + " (o cachorro) está comendo ração.");
    }
}

public class TesteHeranca {
    public static void main(String[] args) {
        Animal animalGenerico = new Animal("Bicho");
        animalGenerico.comer();

        Cachorro meuCachorro = new Cachorro("Rex", "Labrador");
        meuCachorro.comer(); // Chama o método sobrescrito do Cachorro
        meuCachorro.latir();
```

```
        meuCachorro.dormir();
    }
}
```

3.2.3. Polimorfismo

Polimorfismo significa “muitas formas”. Em POO, refere-se à capacidade de um objeto assumir muitas formas. Isso geralmente se manifesta de duas maneiras:

1. **Sobrescrita de Método (Method Overriding)**: Uma subclasse fornece uma implementação específica para um método que já é fornecido por uma de suas superclasses. Isso já foi demonstrado no exemplo de herança com o método `comer()`.
2. **Sobrecarga de Método (Method Overloading)**: Múltiplos métodos com o mesmo nome podem existir na mesma classe, desde que tenham diferentes listas de parâmetros (número, tipo ou ordem dos parâmetros). O compilador decide qual método chamar com base nos argumentos fornecidos.

```java
public class Calculadora {
    public int somar(int a, int b) {
        return a + b;
    }

    public double somar(double a, double b) {
        return a + b;
    }

    public int somar(int a, int b, int c) {
        return a + b + c;
    }
}

public class TestePolimorfismo {
    public static void main(String[] args) {
        Calculadora calc = new Calculadora();

        System.out.println(calc.somar(5, 10));         // Chama somar(int, int)
        System.out.println(calc.somar(5.5, 10.5));     // Chama somar(double, double)
        System.out.println(calc.somar(1, 2, 3));       // Chama somar(int, int, int)

        // Polimorfismo com Herança (Upcasting)
        Animal animal = new Cachorro("Buddy", "Golden"); // Um objeto Cachorro é tratado como um Animal
        animal.comer(); // Chama o comer() de Cachorro devido à sobrescrita
        // animal.latir(); // Erro de compilação: método latir não existe em Animal
    }
}
```

3.3. Interfaces e Classes Abstratas

Ambos são mecanismos para alcançar abstração em Java, mas com diferenças importantes.

3.3.1. Classes Abstratas

- Uma **classe abstrata** é uma classe que não pode ser instanciada diretamente. Ela é projetada para ser herdada por outras classes.
- Pode conter métodos abstratos (sem implementação) e métodos concretos (com implementação).
- É declarada com a palavra-chave `abstract`.
- Se uma classe contém pelo menos um método abstrato, a classe deve ser declarada como abstrata.
- Subclasses de uma classe abstrata devem implementar todos os métodos abstratos, a menos que também sejam declaradas como abstratas.

```java
public abstract class FormaGeometrica {
    // Método abstrato (sem implementação)
    public abstract double calcularArea();

    // Método concreto
    public void exibirMensagem() {
        System.out.println("Esta é uma forma geométrica.");
    }
}

public class Circulo extends FormaGeometrica {
    private double raio;

    public Circulo(double raio) {
        this.raio = raio;
    }

    @Override
    public double calcularArea() {
        return Math.PI * raio * raio;
    }
}

public class Retangulo extends FormaGeometrica {
    private double largura;
    private double altura;

    public Retangulo(double largura, double altura) {
        this.largura = largura;
        this.altura = altura;
    }

    @Override
    public double calcularArea() {
        return largura * altura;
    }
}

public class TesteFormas {
    public static void main(String[] args) {
        // FormaGeometrica forma = new FormaGeometrica(); // Erro: Não pode
instanciar classe abstrata

        Circulo circulo = new Circulo(5.0);
        System.out.println("Área do Círculo: " + circulo.calcularArea());
```

```
        circulo.exibirMensagem();

        Retangulo retangulo = new Retangulo(4.0, 6.0);
        System.out.println("Área do Retângulo: " +
retangulo.calcularArea());
    }
}
```

3.3.2. Interfaces

- Uma **interface** é um contrato que define um conjunto de métodos que uma classe deve implementar. Ela contém apenas declarações de métodos (sem implementação) e constantes.
- É declarada com a palavra-chave `interface`.
- Uma classe pode implementar múltiplas interfaces, permitindo alcançar um tipo de "herança múltipla" de comportamento.
- Todos os métodos em uma interface são implicitamente `public abstract` (em Java 6 e anteriores).
- Todos os campos em uma interface são implicitamente `public static final`.

```java
public interface Acao {
    void executar();
    void desfazer();
}

public class ComandoSalvar implements Acao {
    @Override
    public void executar() {
        System.out.println("Salvando arquivo...");
    }

    @Override
    public void desfazer() {
        System.out.println("Desfazendo salvamento...");
    }
}

public class ComandoAbrir implements Acao {
    @Override
    public void executar() {
        System.out.println("Abrindo arquivo...");
    }

    @Override
    public void desfazer() {
        System.out.println("Desfazendo abertura...");
    }
}

public class TesteInterfaces {
    public static void main(String[] args) {
        Acao salvar = new ComandoSalvar();
        salvar.executar();
        salvar.desfazer();

        Acao abrir = new ComandoAbrir();
        abrir.executar();
        abrir.desfazer();
    }
}
```

3.4. Tratamento de Exceções

Tratamento de exceções é um mecanismo para lidar com erros e condições anormais que podem ocorrer durante a execução de um programa, permitindo que o programa continue a ser executado ou termine de forma controlada. Em Java, as exceções são objetos que representam um evento que interrompe o fluxo normal do programa.

3.4.1. Hierarquia de Exceções

Todas as exceções em Java são subclasses da classe `java.lang.Throwable`. As duas subclasses principais são `Error` e `Exception`.

- **`Error`**: Representa problemas graves que geralmente não podem ser recuperados pelo programa (ex: `OutOfMemoryError`, `StackOverflowError`).
- **`Exception`**: Representa condições que um programa pode tentar recuperar. As `Exception`s são divididas em:
  - **Checked Exceptions**: Devem ser tratadas (com `try-catch`) ou declaradas (com `throws`) no método. O compilador as verifica em tempo de compilação (ex: `IOException`, `SQLException`).
  - **Unchecked Exceptions** (ou `RuntimeException`): Não precisam ser explicitamente tratadas ou declaradas. Geralmente indicam erros de programação (ex: `NullPointerException`, `ArrayIndexOutOfBoundsException`, `ArithmeticException`).

3.4.2. Blocos `try-catch-finally`

- **`try`**: Contém o código que pode gerar uma exceção.
- **`catch`**: Bloco de código que é executado se uma exceção específica ocorrer no bloco `try`. Pode haver múltiplos blocos `catch` para diferentes tipos de exceções.
- **`finally`**: Bloco de código que é **sempre** executado, independentemente de uma exceção ter ocorrido ou não. É útil para liberar recursos (fechar arquivos, conexões de banco de dados).

```java
public class TratamentoExcecoes {
    public static void main(String[] args) {
        try {
            // Código que pode gerar uma exceção
            int resultado = 10 / 0; // ArithmeticException
            System.out.println("Resultado: " + resultado);
        } catch (ArithmeticException e) {
            // Bloco catch para ArithmeticException
            System.err.println("Erro: Divisão por zero não permitida.");
            System.err.println("Mensagem da exceção: " + e.getMessage());
        } catch (Exception e) {
            // Bloco catch genérico para outras exceções
            System.err.println("Ocorreu um erro inesperado: " +
e.getMessage());
        } finally {
            // Bloco finally sempre executado
            System.out.println("Execução do bloco finally.");
        }

        System.out.println("Programa continua após o tratamento da
exceção.");

        // Exemplo com Checked Exception (IOException)
        try {
            java.io.FileInputStream fis = new
java.io.FileInputStream("arquivo_inexistente.txt");
            fis.close();
        } catch (java.io.FileNotFoundException e) {
            System.err.println("Erro: Arquivo não encontrado: " +
e.getMessage());
        } catch (java.io.IOException e) {
            System.err.println("Erro de I/O: " + e.getMessage());
        }
    }
}
```

3.4.3. Palavra-chave `throws`

A palavra-chave `throws` é usada na assinatura de um método para declarar que o método pode lançar uma ou mais exceções verificadas. Isso delega a responsabilidade de tratar a exceção para o método chamador.

```java
import java.io.FileNotFoundException;
import java.io.FileReader;

public class ExemploThrows {

    public static void lerArquivo(String nomeArquivo) throws
FileNotFoundException, IOException { // Adicionado IOException
        FileReader fr = new FileReader(nomeArquivo);
        // ... lógica de leitura do arquivo ...
        fr.close(); // Fechar o FileReader também pode lançar IOException
        System.out.println("Arquivo lido com sucesso (ou tentado ler).");
    }

    public static void main(String[] args) {
        try {
            lerArquivo("meu_arquivo.txt");
        } catch (FileNotFoundException e) {
            System.err.println("Erro: O arquivo especificado não foi
encontrado. " + e.getMessage());
        } catch (IOException e) { // Captura IOException também
            System.err.println("Erro de I/O ao ler o arquivo. " +
e.getMessage());
        }
    }
}
```

3.4.4. Palavra-chave `throw`

A palavra-chave `throw` é usada para lançar explicitamente uma exceção a partir de um método ou bloco de código. Você pode lançar exceções existentes ou criar suas próprias exceções personalizadas.

```
public class ExemploThrow {

    public static void verificarIdade(int idade) {
        if (idade < 18) {
            throw new IllegalArgumentException("Idade deve ser maior ou
igual a 18.");
        }
        System.out.println("Idade válida: " + idade);
    }

    public static void main(String[] args) {
        try {
            verificarIdade(15);
        } catch (IllegalArgumentException e) {
            System.err.println("Erro de validação: " + e.getMessage());
        }

        try {
            verificarIdade(20);
        } catch (IllegalArgumentException e) {
            System.err.println("Erro de validação: " + e.getMessage());
        }
    }
}
```

4. Estruturas de Dados e Coleções

Em Java, as estruturas de dados são fundamentais para armazenar e organizar dados de forma eficiente, permitindo acesso e manipulação otimizados. Esta seção abordará os arrays e o Java Collections Framework, que são os principais mecanismos para trabalhar com coleções de objetos.

4.1. Arrays

Um **array** é uma estrutura de dados de tamanho fixo que armazena uma coleção de elementos do mesmo tipo. Os elementos de um array são acessados por um índice numérico, que começa em 0.

4.1.1. Declaração e Inicialização de Arrays

```
// Declaração de um array de inteiros
int[] numeros;

// Inicialização de um array com tamanho fixo (5 elementos)
numeros = new int[5];

// Declaração e inicialização em uma única linha
String[] nomes = new String[3];

// Inicialização com valores diretamente
double[] notas = {8.5, 9.0, 7.8, 9.2};

// Acessando e modificando elementos
nomes[0] = "Alice";
nomes[1] = "Bob";
nomes[2] = "Charlie";

System.out.println("Primeiro nome: " + nomes[0]); // Saída: Primeiro nome:
Alice
System.out.println("Número de notas: " + notas.length); // Saída: Número de
notas: 4

// Iterando sobre um array (loop for tradicional)
for (int i = 0; i < nomes.length; i++) {
    System.out.println("Nome na posição " + i + ": " + nomes[i]);
}

// Iterando sobre um array (for-each loop)
for (double nota : notas) {
    System.out.println("Nota: " + nota);
}
```

4.1.2. Arrays Multidimensionais

Arrays multidimensionais são arrays de arrays. O mais comum é o array bidimensional (matriz).

```java
// Declaração e inicialização de uma matriz 2x3
int[][] matriz = new int[2][3];

// Atribuindo valores
matriz[0][0] = 1;
matriz[0][1] = 2;
matriz[0][2] = 3;
matriz[1][0] = 4;
matriz[1][1] = 5;
matriz[1][2] = 6;

// Inicialização direta
int[][] outraMatriz = {{10, 20}, {30, 40, 50}};

// Iterando sobre uma matriz
for (int i = 0; i < outraMatriz.length; i++) {
    for (int j = 0; j < outraMatriz[i].length; j++) {
        System.out.print(outraMatriz[i][j] + " ");
    }
    System.out.println();
}
// Saída:
// 10 20
// 30 40 50
```

4.2. Java Collections Framework (JCF)

O **Java Collections Framework (JCF)** é um conjunto de interfaces e classes que fornecem uma arquitetura unificada para representar e manipular coleções de objetos. Ele oferece estruturas de dados mais flexíveis e poderosas que os arrays, pois seu tamanho pode ser dinamicamente ajustado e elas podem armazenar objetos de diferentes tipos (embora seja uma boa prática usar Generics para garantir a segurança de tipo).

As principais interfaces do JCF são `List`, `Set` e `Map`.

4.2.1. Interface `List`

Uma `List` é uma coleção ordenada de elementos que permite duplicatas. Os elementos são acessados por um índice (posição) na lista.

Principais implementações:

- `ArrayList`: Implementação baseada em array redimensionável. Boa para acesso rápido por índice, mas inserções/remoções no meio da lista podem ser lentas.
- `LinkedList`: Implementação baseada em lista duplamente encadeada. Boa para inserções/remoções rápidas no início ou meio da lista, mas acesso por índice pode ser lento.

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

public class ExemploList {
    public static void main(String[] args) {
        // Usando ArrayList
        List<String> frutas = new ArrayList<>();
        frutas.add("Maçã");
        frutas.add("Banana");
        frutas.add("Laranja");
        frutas.add("Maçã"); // Permite duplicatas

        System.out.println("ArrayList: " + frutas); // Saída: [Maçã, Banana, Laranja, Maçã]
        System.out.println("Elemento na posição 1: " + frutas.get(1)); // Saída: Elemento na posição 1: Banana

        frutas.remove("Banana");
        System.out.println("ArrayList após remover Banana: " + frutas); // Saída: [Maçã, Laranja, Maçã]

        // Usando LinkedList
        List<Integer> numeros = new LinkedList<>();
        numeros.add(10);
        numeros.add(20);
        numeros.add(30);

        System.out.println("LinkedList: " + numeros); // Saída: [10, 20, 30]
        numeros.add(1, 15); // Adiciona 15 na posição 1
        System.out.println("LinkedList após adicionar 15 na posição 1: " + numeros); // Saída: [10, 15, 20, 30]
    }
}
```

4.2.2. Interface `Set`

Um `Set` é uma coleção que não permite elementos duplicados. A ordem dos elementos não é garantida (exceto em algumas implementações específicas).

Principais implementações:

- **`HashSet`**: Implementação baseada em tabela hash. Oferece desempenho constante (O(1)) para operações básicas como `add`, `remove` e `contains`, mas não garante a ordem.
- **`LinkedHashSet`**: Mantém a ordem de inserção dos elementos.
- **`TreeSet`**: Armazena os elementos em ordem crescente (natural ou definida por um `Comparator`).

```
import java.util.HashSet;
import java.util.Set;
import java.util.TreeSet;

public class ExemploSet {
    public static void main(String[] args) {
        // Usando HashSet
        Set<String> cores = new HashSet<>();
        cores.add("Vermelho");
        cores.add("Verde");
        cores.add("Azul");
        cores.add("Vermelho"); // Não será adicionado, pois é duplicata

        System.out.println("HashSet: " + cores); // Saída: [Azul, Verde, Vermelho] (ordem pode variar)
        System.out.println("Contém Verde? " + cores.contains("Verde")); // Saída: Contém Verde? true

        // Usando TreeSet (mantém ordem natural)
        Set<Integer> numerosOrdenados = new TreeSet<>();
        numerosOrdenados.add(50);
        numerosOrdenados.add(10);
        numerosOrdenados.add(30);
        numerosOrdenados.add(10); // Não será adicionado

        System.out.println("TreeSet: " + numerosOrdenados); // Saída: [10, 30, 50]
    }
}
```

4.2.3. Interface `Map`

Um `Map` (também conhecido como dicionário ou array associativo) armazena pares de chave-valor. Cada chave é única e mapeia para um único valor. Não permite chaves

duplicadas, mas permite valores duplicados.

Principais implementações:

- **`HashMap`**: Implementação baseada em tabela hash. Oferece desempenho constante (O(1)) para operações básicas, mas não garante a ordem.
- **`LinkedHashMap`**: Mantém a ordem de inserção das chaves.
- **`TreeMap`**: Armazena as chaves em ordem crescente (natural ou definida por um `Comparator`).

```java
import java.util.HashMap;
import java.util.Map;
import java.util.TreeMap;

public class ExemploMap {
    public static void main(String[] args) {
        // Usando HashMap
        Map<String, String> capitais = new HashMap<>();
        capitais.put("Brasil", "Brasília");
        capitais.put("França", "Paris");
        capitais.put("Japão", "Tóquio");
        capitais.put("Brasil", "Rio de Janeiro"); // Sobrescreve o valor
para a chave "Brasil"

        System.out.println("HashMap: " + capitais); // Saída: {Japão=Tóquio,
Brasil=Rio de Janeiro, França=Paris} (ordem pode variar)
        System.out.println("Capital do Brasil: " + capitais.get("Brasil"));
// Saída: Capital do Brasil: Rio de Janeiro

        capitais.remove("França");
        System.out.println("HashMap após remover França: " + capitais); //
Saída: {Japão=Tóquio, Brasil=Rio de Janeiro}

        // Iterando sobre um Map
        for (Map.Entry<String, String> entry : capitais.entrySet()) {
            System.out.println("País: " + entry.getKey() + ", Capital: " +
entry.getValue());
        }

        // Usando TreeMap (chaves em ordem natural)
        Map<Integer, String> codigos = new TreeMap<>();
        codigos.put(3, "C");
        codigos.put(1, "A");
        codigos.put(2, "B");

        System.out.println("TreeMap: " + codigos); // Saída: {1=A, 2=B, 3=C}
    }
}
```

4.3. Iteradores

Um **Iterator** é um objeto que permite percorrer uma coleção e remover elementos durante a iteração. É a maneira universal de acessar elementos em qualquer coleção

do JCF, independentemente de sua implementação subjacente.

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

public class ExemploIterator {
    public static void main(String[] args) {
        List<String> linguagens = new ArrayList<>();
        linguagens.add("Java");
        linguagens.add("Python");
        linguagens.add("C++");
        linguagens.add("JavaScript");

        Iterator<String> iterator = linguagens.iterator();

        System.out.println("Elementos da lista:");
        while (iterator.hasNext()) {
            String linguagem = iterator.next();
            System.out.println(linguagem);

            // Exemplo de remoção segura durante a iteração
            if (linguagem.equals("C++")) {
                iterator.remove(); // Remove "C++" da lista
            }
        }
        System.out.println("Lista após remoção com Iterator: " +
linguagens);
        // Saída:
        // Elementos da lista:
        // Java
        // Python
        // C++
        // JavaScript
        // Lista após remoção com Iterator: [Java, Python, JavaScript]
    }
}
```

5. Algoritmos Essenciais

Algoritmos são sequências de instruções bem definidas para resolver um problema ou realizar uma tarefa. Nesta seção, exploraremos algoritmos fundamentais de busca,

ordenação e filtro, que são a base para muitas operações em ciência da computação e desenvolvimento de software.

5.1. Algoritmos de Busca

Algoritmos de busca são usados para encontrar um elemento específico dentro de uma coleção de dados.

5.1.1. Busca Linear (Sequential Search)

A busca linear é o algoritmo de busca mais simples. Ele verifica cada elemento da coleção sequencialmente até encontrar o elemento desejado ou percorrer toda a coleção. É adequado para coleções pequenas ou não ordenadas.

- **Complexidade de Tempo**: O(n) no pior caso (o elemento não está presente ou está no final) e O(1) no melhor caso (o elemento está no início).

```java
public class BuscaLinear {
    public static int buscar(int[] array, int elemento) {
        for (int i = 0; i < array.length; i++) {
            if (array[i] == elemento) {
                return i; // Elemento encontrado, retorna o índice
            }
        }
        return -1; // Elemento não encontrado
    }

    public static void main(String[] args) {
        int[] numeros = {5, 12, 3, 8, 15, 7};
        int elementoBuscado = 8;
        int indice = buscar(numeros, elementoBuscado);

        if (indice != -1) {
            System.out.println("Elemento " + elementoBuscado + " encontrado
no índice: " + indice);
        } else {
            System.out.println("Elemento " + elementoBuscado + " não
encontrado.");
        }

        elementoBuscado = 10;
        indice = buscar(numeros, elementoBuscado);
        if (indice != -1) {
            System.out.println("Elemento " + elementoBuscado + " encontrado
no índice: " + indice);
        } else {
            System.out.println("Elemento " + elementoBuscado + " não
encontrado.");
        }
    }
}
```

5.1.2. Busca Binária (Binary Search)

A busca binária é um algoritmo de busca muito mais eficiente, mas requer que a coleção de dados esteja **ordenada**. Ele funciona dividindo repetidamente a parte da lista que pode conter o item ao meio, eliminando metade dos elementos restantes a cada passo.

- **Complexidade de Tempo**: O(log n).

```java
import java.util.Arrays;

public class BuscaBinaria {
    public static int buscar(int[] array, int elemento) {
        int inicio = 0;
        int fim = array.length - 1;

        while (inicio <= fim) {
            int meio = inicio + (fim - inicio) / 2; // Evita overflow para grandes valores de inicio + fim

            if (array[meio] == elemento) {
                return meio; // Elemento encontrado
            } else if (array[meio] < elemento) {
                inicio = meio + 1; // Busca na metade direita
            } else {
                fim = meio - 1; // Busca na metade esquerda
            }
        }
        return -1; // Elemento não encontrado
    }

    public static void main(String[] args) {
        int[] numeros = {3, 5, 7, 8, 12, 15}; // Array DEVE estar ordenado
        int elementoBuscado = 12;
        int indice = buscar(numeros, elementoBuscado);

        if (indice != -1) {
            System.out.println("Elemento " + elementoBuscado + " encontrado no índice: " + indice);
        } else {
            System.out.println("Elemento " + elementoBuscado + " não encontrado.");
        }

        elementoBuscado = 10;
        indice = buscar(numeros, elementoBuscado);
        if (indice != -1) {
            System.out.println("Elemento " + elementoBuscado + " encontrado no índice: " + indice);
        } else {
            System.out.println("Elemento " + elementoBuscado + " não encontrado.");
        }
```

```
    }
}
```

5.2. Algoritmos de Ordenação

Algoritmos de ordenação organizam os elementos de uma coleção em uma ordem específica (crescente ou decrescente).

5.2.1. Bubble Sort (Ordenação por Bolha)

O Bubble Sort é um algoritmo de ordenação simples que funciona repetidamente percorrendo a lista, comparando pares de elementos adjacentes e trocando-os se estiverem na ordem errada. O processo é repetido até que nenhuma troca seja necessária, indicando que a lista está ordenada.

- **Complexidade de Tempo**: O(n^2) no pior e caso médio, O(n) no melhor caso (já ordenado).

```java
import java.util.Arrays;

public class BubbleSort {
    public static void ordenar(int[] array) {
        int n = array.length;
        boolean trocou;
        for (int i = 0; i < n - 1; i++) {
            trocou = false;
            for (int j = 0; j < n - 1 - i; j++) {
                if (array[j] > array[j + 1]) {
                    // Troca array[j] e array[j+1]
                    int temp = array[j];
                    array[j] = array[j + 1];
                    array[j + 1] = temp;
                    trocou = true;
                }
            }
            // Se nenhuma troca ocorreu nesta passagem, o array está
ordenado
            if (!trocou) {
                break;
            }
        }
    }

    public static void main(String[] args) {
        int[] numeros = {64, 34, 25, 12, 22, 11, 90};
        System.out.println("Array original: " + Arrays.toString(numeros));
        ordenar(numeros);
        System.out.println("Array ordenado (Bubble Sort): " +
Arrays.toString(numeros));
    }
}
```

5.2.2. Selection Sort (Ordenação por Seleção)

O Selection Sort funciona dividindo a lista em duas partes: uma sublista de elementos já ordenados e uma sublista de elementos restantes não ordenados. Ele repetidamente encontra o menor elemento da sublista não ordenada e o move para o final da sublista ordenada.

- **Complexidade de Tempo**: O(n^2) em todos os casos (melhor, médio e pior).

```java
import java.util.Arrays;

public class SelectionSort {
    public static void ordenar(int[] array) {
        int n = array.length;
        for (int i = 0; i < n - 1; i++) {
            int indiceMinimo = i;
            for (int j = i + 1; j < n; j++) {
                if (array[j] < array[indiceMinimo]) {
                    indiceMinimo = j;
                }
            }
            // Troca o elemento mínimo encontrado com o primeiro elemento
não ordenado
            int temp = array[indiceMinimo];
            array[indiceMinimo] = array[i];
            array[i] = temp;
        }
    }

    public static void main(String[] args) {
        int[] numeros = {64, 25, 12, 22, 11};
        System.out.println("Array original: " + Arrays.toString(numeros));
        ordenar(numeros);
        System.out.println("Array ordenado (Selection Sort): " +
Arrays.toString(numeros));
    }
}
```

5.2.3. Insertion Sort (Ordenação por Inserção)

O Insertion Sort constrói a lista final ordenada um item por vez. Ele itera sobre a lista, pegando cada elemento e inserindo-o em sua posição correta na parte já ordenada da lista.

- **Complexidade de Tempo**: O(n^2) no pior e caso médio, O(n) no melhor caso (já ordenado).

```java
import java.util.Arrays;

public class InsertionSort {
    public static void ordenar(int[] array) {
        int n = array.length;
        for (int i = 1; i < n; i++) {
            int chave = array[i];
            int j = i - 1;

            // Move os elementos de array[0..i-1], que são maiores que a
chave,
            // para uma posição à frente de sua posição atual
            while (j >= 0 && array[j] > chave) {
                array[j + 1] = array[j];
                j = j - 1;
            }
            array[j + 1] = chave;
        }
    }

    public static void main(String[] args) {
        int[] numeros = {12, 11, 13, 5, 6};
        System.out.println("Array original: " + Arrays.toString(numeros));
        ordenar(numeros);
        System.out.println("Array ordenado (Insertion Sort): " +
Arrays.toString(numeros));
    }
}
```

5.2.4. Merge Sort (Ordenação por Intercalação)

O Merge Sort é um algoritmo de ordenação eficiente, baseado no paradigma “dividir para conquistar”. Ele divide recursivamente o array em duas metades até que cada sub-array tenha apenas um elemento (que é considerado ordenado). Em seguida, ele mescla (merge) essas sub-arrays de forma ordenada.

- **Complexidade de Tempo**: O(n log n) em todos os casos (melhor, médio e pior).

```java
import java.util.Arrays;

public class MergeSort {

    public static void ordenar(int[] array) {
        if (array == null || array.length <= 1) {
            return; // Array já ordenado ou vazio
        }
        int[] temp = new int[array.length];
        mergeSort(array, temp, 0, array.length - 1);
    }

    private static void mergeSort(int[] array, int[] temp, int esquerda, int
direita) {
        if (esquerda < direita) {
            int meio = esquerda + (direita - esquerda) / 2;
            mergeSort(array, temp, esquerda, meio);
            mergeSort(array, temp, meio + 1, direita);
            merge(array, temp, esquerda, meio, direita);
        }
    }

    private static void merge(int[] array, int[] temp, int esquerda, int
meio, int direita) {
        // Copia ambas as metades para o array temporário
        for (int i = esquerda; i <= direita; i++) {
            temp[i] = array[i];
        }

        int i = esquerda; // Índice inicial da primeira metade
        int j = meio + 1; // Índice inicial da segunda metade
        int k = esquerda; // Índice inicial do array principal

        // Mescla as duas metades de volta no array original
        while (i <= meio && j <= direita) {
            if (temp[i] <= temp[j]) {
                array[k] = temp[i];
                i++;
            } else {
                array[k] = temp[j];
                j++;
            }
            k++;
        }
```

```
        // Copia os elementos restantes da primeira metade, se houver
        while (i <= meio) {
            array[k] = temp[i];
            k++;
            i++;
        }

        // Copia os elementos restantes da segunda metade, se houver
        while (j <= direita) {
            array[k] = temp[j];
            k++;
            j++;
        }
    }

    public static void main(String[] args) {
        int[] numeros = {38, 27, 43, 3, 9, 82, 10};
        System.out.println("Array original: " + Arrays.toString(numeros));
        ordenar(numeros);
        System.out.println("Array ordenado (Merge Sort): " +
Arrays.toString(numeros));
    }
}
```

5.2.5. Quick Sort (Ordenação Rápida)

O Quick Sort é outro algoritmo de ordenação eficiente baseado em “dividir para conquistar”. Ele seleciona um elemento como pivô e particiona o array em torno do pivô, de modo que todos os elementos menores que o pivô fiquem antes dele e todos os elementos maiores fiquem depois. Em seguida, ele aplica recursivamente o mesmo processo às sub-arrays.

- **Complexidade de Tempo**: O(n log n) no caso médio, O(n^2) no pior caso (escolha de pivô ruim).

```java
import java.util.Arrays;

public class QuickSort {

    public static void ordenar(int[] array) {
        if (array == null || array.length <= 1) {
            return;
        }
        quickSort(array, 0, array.length - 1);
    }

    private static void quickSort(int[] array, int baixo, int alto) {
        if (baixo < alto) {
            // pi é o índice de particionamento, array[pi] está agora no
lugar certo
            int pi = particionar(array, baixo, alto);

            // Ordena recursivamente os elementos antes e depois do
particionamento
            quickSort(array, baixo, pi - 1);
            quickSort(array, pi + 1, alto);
        }
    }

    private static int particionar(int[] array, int baixo, int alto) {
        int pivo = array[alto]; // Escolhe o último elemento como pivô
        int i = (baixo - 1); // Índice do menor elemento

        for (int j = baixo; j < alto; j++) {
            // Se o elemento atual é menor ou igual ao pivô
            if (array[j] <= pivo) {
                i++;

                // Troca array[i] e array[j]
                int temp = array[i];
                array[i] = array[j];
                array[j] = temp;
            }
        }

        // Troca array[i+1] e array[alto] (o pivô)
        int temp = array[i + 1];
        array[i + 1] = array[alto];
        array[alto] = temp;
```

```
        return i + 1;
    }

    public static void main(String[] args) {
        int[] numeros = {10, 7, 8, 9, 1, 5};
        System.out.println("Array original: " + Arrays.toString(numeros));
        ordenar(numeros);
        System.out.println("Array ordenado (Quick Sort): " +
Arrays.toString(numeros));
    }
}
```

5.3. Algoritmos de Filtro

Algoritmos de filtro são usados para criar uma nova coleção de dados contendo apenas os elementos que satisfazem uma ou mais condições específicas.

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

public class AlgoritmoFiltro {

    /**
     * Filtra um array de inteiros, retornando uma nova lista contendo
apenas
     * os números pares.
     *
     * @param array O array de inteiros a ser filtrado.
     * @return Uma lista de inteiros contendo apenas os números pares.
     */
    public static List<Integer> filtrarNumerosPares(int[] array) {
        List<Integer> pares = new ArrayList<>();
        for (int numero : array) {
            if (numero % 2 == 0) {
                pares.add(numero);
            }
        }
        return pares;
    }

    /**
     * Filtra uma lista de strings, retornando uma nova lista contendo
apenas
     * as strings que começam com uma letra específica.
     *
     * @param lista A lista de strings a ser filtrada.
     * @param letraInicial A letra inicial para o filtro.
     * @return Uma lista de strings que começam com a letra especificada.
     */
    public static List<String> filtrarPorLetraInicial(List<String> lista,
char letraInicial) {
        List<String> filtrados = new ArrayList<>();
        for (String item : lista) {
            if (item != null && !item.isEmpty() && item.charAt(0) ==
letraInicial) {
                filtrados.add(item);
            }
        }
        return filtrados;
    }

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

```
        int[] numeros = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
        List<Integer> numerosPares = filtrarNumerosPares(numeros);
        System.out.println("Números pares: " + numerosPares); // Saída: [2,
4, 6, 8, 10]

        List<String> nomes = new ArrayList<>();
        nomes.add("Ana");
        nomes.add("Bruno");
        nomes.add("Carla");
        nomes.add("Daniel");
        nomes.add("Amanda");

        List<String> nomesComA = filtrarPorLetraInicial(nomes, 'A');
        System.out.println("Nomes que começam com 'A': " + nomesComA); //
Saída: [Ana, Amanda]

        List<String> nomesComB = filtrarPorLetraInicial(nomes, 'B');
        System.out.println("Nomes que começam com 'B': " + nomesComB); //
Saída: [Bruno]
    }
}
```

6. Programação Concorrente (Threads e Sincronização)

A programação concorrente é a capacidade de um programa executar várias tarefas ou processos simultaneamente. Em Java, a concorrência é alcançada principalmente através de **threads**, que são unidades de execução leves dentro de um processo. A programação concorrente é essencial para aproveitar ao máximo os processadores multi-core modernos, melhorar a responsividade de aplicações com interface gráfica e lidar com operações de I/O de forma eficiente.

6.1. Conceitos de Concorrência

- **Processo**: Um processo é uma instância de um programa em execução. Cada processo tem seu próprio espaço de memória e recursos.
- **Thread**: Uma thread é uma unidade de execução dentro de um processo. Múltiplas threads podem existir dentro do mesmo processo e compartilhar

recursos como memória e arquivos abertos. Isso torna a comunicação entre threads mais rápida e eficiente do que a comunicação entre processos.

- **Multithreading**: É a execução de múltiplas threads dentro de um único processo. O sistema operacional gerencia a alternância entre as threads (context switching) para dar a ilusão de execução paralela, mesmo em um processador de núcleo único. Em processadores multi-core, as threads podem ser executadas verdadeiramente em paralelo.

6.2. Criação e Gerenciamento de Threads

Existem duas maneiras principais de criar uma thread em Java:

1. **Estendendo a classe `Thread`**
2. **Implementando a interface `Runnable`**

A abordagem de implementar `Runnable` é geralmente preferida, pois permite que a classe que contém a lógica da thread herde de outra classe (Java não suporta herança múltipla de classes) e promove uma melhor separação entre a lógica da tarefa e o mecanismo de execução da thread.

6.2.1. Estendendo a Classe `Thread`

```
public class MinhaThread extends Thread {
    private String nomeThread;

    public MinhaThread(String nome) {
        this.nomeThread = nome;
    }

    @Override
    public void run() {
        try {
            for (int i = 0; i < 5; i++) {
                System.out.println(nomeThread + ": Contagem " + i);
                Thread.sleep(500); // Pausa a execução da thread por 500ms
            }
        } catch (InterruptedException e) {
            System.out.println("Thread " + nomeThread + " interrompida.");
        }
        System.out.println("Thread " + nomeThread + " finalizada.");
    }

    public static void main(String[] args) {
        MinhaThread t1 = new MinhaThread("Thread-1");
        MinhaThread t2 = new MinhaThread("Thread-2");

        t1.start(); // Inicia a execução da thread (chama o método run())
        t2.start();
    }
}
```

6.2.2. Implementando a Interface `Runnable`

```
public class MinhaTarefa implements Runnable {
    private String nomeTarefa;

    public MinhaTarefa(String nome) {
        this.nomeTarefa = nome;
    }

    @Override
    public void run() {
        try {
            for (int i = 0; i < 5; i++) {
                System.out.println(nomeTarefa + ": Contagem " + i);
                Thread.sleep(500);
            }
        } catch (InterruptedException e) {
            System.out.println("Tarefa " + nomeTarefa + " interrompida.");
        }
        System.out.println("Tarefa " + nomeTarefa + " finalizada.");
    }

    public static void main(String[] args) {
        // Cria instâncias da tarefa
        MinhaTarefa tarefa1 = new MinhaTarefa("Tarefa-A");
        MinhaTarefa tarefa2 = new MinhaTarefa("Tarefa-B");

        // Cria threads e passa as tarefas para elas
        Thread t1 = new Thread(tarefa1);
        Thread t2 = new Thread(tarefa2);

        // Inicia as threads
        t1.start();
        t2.start();
    }
}
```

6.3. Sincronização

Quando múltiplas threads acessam e modificam recursos compartilhados (como variáveis de instância ou objetos), podem ocorrer problemas de **condição de corrida** (race condition), onde o resultado da operação depende da ordem em que as threads

são executadas. A **sincronização** é o mecanismo para evitar esses problemas, garantindo que apenas uma thread possa acessar o recurso compartilhado por vez.

6.3.1. Palavra-chave `synchronized`

A maneira mais simples de garantir a exclusão mútua (mutex) é usando a palavra-chave `synchronized`. Ela pode ser aplicada a métodos ou a blocos de código.

- **Método Sincronizado**: Quando um método é declarado como `synchronized`, a thread que o chama adquire um bloqueio (lock) no objeto ao qual o método pertence. Nenhuma outra thread pode chamar qualquer método `synchronized` no mesmo objeto até que a primeira thread libere o bloqueio (ao sair do método).

```
public class Contador {
    private int contagem = 0;


    // Apenas uma thread pode executar este método por vez em uma
instância de Contador
    public synchronized void incrementar() {
        contagem++;
    }


    public int getContagem() {
        return contagem;
    }
}
```

- **Bloco Sincronizado**: Permite sincronizar apenas uma parte de um método, em vez do método inteiro. Isso pode melhorar o desempenho, pois o bloqueio é mantido por menos tempo. O bloco `synchronized` deve especificar em qual objeto o bloqueio será adquirido.

```java
public class ExemploBlocoSincronizado {
    private Object lock1 = new Object();
    private Object lock2 = new Object();


    public void metodo1() {
        synchronized (lock1) {
            // Seção crítica 1
            System.out.println("Dentro do bloco sincronizado 1");
        }
        // Código não sincronizado
    }


    public void metodo2() {
        synchronized (lock2) {
            // Seção crítica 2
            System.out.println("Dentro do bloco sincronizado 2");
        }
    }
}
```

Exemplo de Condição de Corrida e Sincronização:

```java
class ContadorNaoSincronizado {
    private int c = 0;

    public void incrementar() { c++; }
    public int valor() { return c; }
}

class ContadorSincronizado {
    private int c = 0;

    public synchronized void incrementar() { c++; }
    public int valor() { return c; }
}

public class TesteContador {
    public static void main(String[] args) throws InterruptedException {
        // Teste sem sincronização
        ContadorNaoSincronizado contador1 = new ContadorNaoSincronizado();
        Runnable r1 = () -> {
            for (int i = 0; i < 10000; i++) {
                contador1.incrementar();
            }
        };
        Thread t1 = new Thread(r1);
        Thread t2 = new Thread(r1);
        t1.start();
        t2.start();
        t1.join(); // Espera a thread t1 terminar
        t2.join(); // Espera a thread t2 terminar
        System.out.println("Valor final (não sincronizado): " +
contador1.valor()); // Provavelmente < 20000

        // Teste com sincronização
        ContadorSincronizado contador2 = new ContadorSincronizado();
        Runnable r2 = () -> {
            for (int i = 0; i < 10000; i++) {
                contador2.incrementar();
            }
        };
        Thread t3 = new Thread(r2);
        Thread t4 = new Thread(r2);
        t3.start();
        t4.start();
        t3.join();
        t4.join();
```

```
        System.out.println("Valor final (sincronizado): " +
contador2.valor()); // Exatamente 20000
    }
}
```

6.3.2. `wait()`, `notify()` e `notifyAll()`

Esses métodos da classe `Object` são usados para coordenar a interação entre threads. Eles permitem que uma thread espere por uma condição específica e que outra thread a notifique quando essa condição for atendida. **Esses métodos devem ser chamados apenas de dentro de um bloco ou método `synchronized`.**

- **`wait()`**: Faz com que a thread atual libere o bloqueio do objeto e entre em um estado de espera. A thread permanecerá em espera até que outra thread chame `notify()` ou `notifyAll()` no mesmo objeto.
- **`notify()`**: Acorda uma única thread que está esperando no bloqueio do objeto. A escolha de qual thread acordar é arbitrária.
- **`notifyAll()`**: Acorda todas as threads que estão esperando no bloqueio do objeto.

Exemplo: Problema do Produtor-Consumidor

```java
import java.util.LinkedList;
import java.util.Queue;

// Buffer compartilhado entre produtor e consumidor
class Buffer {
    private Queue<Integer> fila = new LinkedList<>();
    private int capacidade;

    public Buffer(int capacidade) {
        this.capacidade = capacidade;
    }

    public synchronized void produzir(int item) throws InterruptedException
{
        // Espera enquanto a fila estiver cheia
        while (fila.size() == capacidade) {
            System.out.println("Buffer cheio, produtor esperando...");
            wait();
        }

        fila.add(item);
        System.out.println("Produzido: " + item);
        notifyAll(); // Notifica os consumidores que um item foi produzido
    }

    public synchronized int consumir() throws InterruptedException {
        // Espera enquanto a fila estiver vazia
        while (fila.isEmpty()) {
            System.out.println("Buffer vazio, consumidor esperando...");
            wait();
        }

        int item = fila.poll();
        System.out.println("Consumido: " + item);
        notifyAll(); // Notifica os produtores que há espaço no buffer
        return item;
    }
}

class Produtor implements Runnable {
    private Buffer buffer;

    public Produtor(Buffer buffer) {
        this.buffer = buffer;
    }
```

```
    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            try {
                buffer.produzir(i);
                Thread.sleep((long) (Math.random() * 100));
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
    }
}

class Consumidor implements Runnable {
    private Buffer buffer;

    public Consumidor(Buffer buffer) {
        this.buffer = buffer;
    }

    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            try {
                buffer.consumir();
                Thread.sleep((long) (Math.random() * 500));
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
    }
}

public class TesteProdutorConsumidor {
    public static void main(String[] args) {
        Buffer buffer = new Buffer(5);
        Thread produtor = new Thread(new Produtor(buffer));
        Thread consumidor = new Thread(new Consumidor(buffer));

        produtor.start();
        consumidor.start();
    }
}
```

6.4. Problemas Comuns de Concorrência

- **Deadlock**: Ocorre quando duas ou more threads estão bloqueadas para sempre, cada uma esperando pela outra. Isso geralmente acontece quando múltiplas threads tentam adquirir bloqueios em objetos diferentes em ordens diferentes.

```java
// Exemplo de Deadlock
public class ExemploDeadlock {
    public static final Object lock1 = new Object();
    public static final Object lock2 = new Object();


    public static void main(String[] args) {
        Thread t1 = new Thread(() -> {
            synchronized (lock1) {
                System.out.println("Thread 1: Segurando lock 1...");
                try { Thread.sleep(100); } catch (InterruptedException
e) {}
                System.out.println("Thread 1: Esperando por lock
2...");
                synchronized (lock2) {
                    System.out.println("Thread 1: Segurando lock 1 &
2...");
                }
            }
        });


        Thread t2 = new Thread(() -> {
            synchronized (lock2) {
                System.out.println("Thread 2: Segurando lock 2...");
                try { Thread.sleep(100); } catch (InterruptedException
e) {}
                System.out.println("Thread 2: Esperando por lock
1...");
                synchronized (lock1) {
                    System.out.println("Thread 2: Segurando lock 1 &
2...");
                }
            }
        });


        t1.start();
        t2.start();
    }
}
```

Para evitar deadlocks: Adquira os bloqueios sempre na mesma ordem em todas as threads.

- **Starvation (Inanição)**: Ocorre quando uma thread é perpetuamente negada o acesso a um recurso compartilhado porque outras threads (geralmente de maior prioridade) estão constantemente usando o recurso. Isso pode ser causado por um agendador de threads mal projetado ou pelo uso inadequado de prioridades de thread.
- **Livelock**: Ocorre quando duas ou mais threads estão ativamente tentando resolver um conflito, mas acabam presas em um ciclo de ações que não leva a nenhum progresso. As threads não estão bloqueadas, mas estão ocupadas respondendo umas às outras de uma maneira que as impede de fazer qualquer trabalho útil. É como duas pessoas tentando passar uma pela outra em um corredor e ambas se movem para o mesmo lado repetidamente.

7. Interação com o Sistema de Arquivos e I/O

A interação com o sistema de arquivos e operações de Entrada/Saída (I/O) são tarefas comuns em muitas aplicações Java. O Java 6 oferece um conjunto robusto de classes para ler e escrever dados em arquivos, manipular diretórios e até mesmo serializar objetos.

7.1. Classes `File`, `FileReader`, `FileWriter`, `BufferedReader`, `BufferedWriter`

7.1.1. Classe `File`

A classe `java.io.File` é usada para representar arquivos e diretórios no sistema de arquivos. Ela não lida com o conteúdo dos arquivos, mas sim com suas propriedades (nome, caminho, tamanho, permissões) e operações de gerenciamento (criar, excluir, renomear, listar conteúdo de diretório).

```java
import java.io.File;
import java.io.IOException;

public class ExemploFile {
    public static void main(String[] args) {
        // Criando um objeto File para um arquivo
        File arquivo = new File("meu_arquivo.txt");

        try {
            // Criar o arquivo se ele não existir
            if (arquivo.createNewFile()) {
                System.out.println("Arquivo criado: " + arquivo.getName());
            } else {
                System.out.println("Arquivo já existe.");
            }

            // Obtendo informações do arquivo
            System.out.println("Caminho absoluto: " +
arquivo.getAbsolutePath());
            System.out.println("Nome do arquivo: " + arquivo.getName());
            System.out.println("É um diretório? " + arquivo.isDirectory());
            System.out.println("É um arquivo? " + arquivo.isFile());
            System.out.println("Tamanho do arquivo (bytes): " +
arquivo.length());
            System.out.println("Pode ser lido? " + arquivo.canRead());
            System.out.println("Pode ser escrito? " + arquivo.canWrite());

            // Criando um diretório
            File diretorio = new File("novo_diretorio");
            if (diretorio.mkdir()) {
                System.out.println("Diretório criado: " +
diretorio.getName());
            } else {
                System.out.println("Diretório já existe ou não pôde ser
criado.");
            }

            // Listando arquivos em um diretório
            File pastaAtual = new File("."); // Diretório atual
            System.out.println("Conteúdo do diretório atual:");
            String[] arquivosNaPasta = pastaAtual.list();
            if (arquivosNaPasta != null) {
                for (String nome : arquivosNaPasta) {
                    System.out.println("- " + nome);
                }
```

```
            }

            // Excluindo o arquivo e o diretório (descomente para testar)
            // if (arquivo.delete()) {
            //     System.out.println("Arquivo excluído: " +
arquivo.getName());
            // }
            // if (diretorio.delete()) {
            //     System.out.println("Diretório excluído: " +
diretorio.getName());
            // }

        } catch (IOException e) {
            System.err.println("Ocorreu um erro de I/O: " + e.getMessage());
        }
    }
}
```

7.1.2. `FileReader` e `FileWriter`

`FileReader` e `FileWriter` são classes para ler e escrever dados de caracteres em arquivos. Eles são úteis para lidar com arquivos de texto simples.

```java
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class ExemploFileReaderWriter {
    public static void main(String[] args) {
        String nomeArquivo = "exemplo.txt";
        String conteudo = "Olá, mundo!\nEste é um exemplo de escrita e
leitura de arquivo em Java.";

        // Escrevendo no arquivo
        try (FileWriter writer = new FileWriter(nomeArquivo)) {
            writer.write(conteudo);
            System.out.println("Conteúdo escrito no arquivo: " +
nomeArquivo);
        } catch (IOException e) {
            System.err.println("Erro ao escrever no arquivo: " +
e.getMessage());
        }

        // Lendo do arquivo
        try (FileReader reader = new FileReader(nomeArquivo)) {
            int caractere;
            System.out.println("Conteúdo lido do arquivo: " + nomeArquivo);
            while ((caractere = reader.read()) != -1) {
                System.out.print((char) caractere);
            }
            System.out.println(); // Nova linha após a leitura
        } catch (IOException e) {
            System.err.println("Erro ao ler do arquivo: " + e.getMessage());
        }
    }
}
```

7.1.3. `BufferedReader` e `BufferedWriter`

`BufferedReader` e `BufferedWriter` são classes de buffer que envolvem `FileReader` e `FileWriter` (ou outros `Reader`/`Writer`) para melhorar o desempenho de I/O, lendo ou escrevendo blocos de dados de uma vez, em vez de caractere por caractere. Eles também fornecem métodos convenientes como `readLine()`.

```java
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class ExemploBufferedReaderWriter {
    public static void main(String[] args) {
        String nomeArquivo = "buffered_exemplo.txt";
        String[] linhas = {"Primeira linha.", "Segunda linha com buffer.",
"Terceira e última linha."};

        // Escrevendo no arquivo com BufferedWriter
        try (BufferedWriter writer = new BufferedWriter(new
FileWriter(nomeArquivo))) {
            for (String linha : linhas) {
                writer.write(linha);
                writer.newLine(); // Adiciona uma nova linha
            }
            System.out.println("Conteúdo escrito no arquivo com buffer: " +
nomeArquivo);
        } catch (IOException e) {
            System.err.println("Erro ao escrever no arquivo com buffer: " +
e.getMessage());
        }

        // Lendo do arquivo com BufferedReader
        try (BufferedReader reader = new BufferedReader(new
FileReader(nomeArquivo))) {
            String linha;
            System.out.println("Conteúdo lido do arquivo com buffer: " +
nomeArquivo);
            while ((linha = reader.readLine()) != null) {
                System.out.println(linha);
            }
        } catch (IOException e) {
            System.err.println("Erro ao ler do arquivo com buffer: " +
e.getMessage());
        }
    }
}
```

7.2. Serialização de Objetos

Serialização é o processo de converter um objeto em um fluxo de bytes para armazená-lo em um arquivo, transmiti-lo pela rede ou armazená-lo em um banco de dados. **Desserialização** é o processo inverso, reconstruindo o objeto a partir do fluxo de bytes. Em Java, um objeto pode ser serializado se sua classe implementar a interface `java.io.Serializable`.

- A interface `Serializable` é uma interface de marcação (marker interface), o que significa que ela não possui métodos para implementar. Ela apenas indica que a classe pode ser serializada.
- Todos os campos não `static` e não `transient` de um objeto serializável são serializados.
- Para serializar e desserializar, usamos `ObjectOutputStream` e `ObjectInputStream`.

```java
import java.io.*;

// A classe Pessoa deve implementar Serializable para ser serializada
class Pessoa implements Serializable {
    private static final long serialVersionUID = 1L; // Recomendado para
controle de versão
    private String nome;
    private int idade;
    private transient String senha; // 'transient' impede que este campo
seja serializado

    public Pessoa(String nome, int idade, String senha) {
        this.nome = nome;
        this.idade = idade;
        this.senha = senha;
    }

    @Override
    public String toString() {
        return "Pessoa{nome=\'" + nome + "\', idade=" + idade + ", senha=\'"
+ senha + "\'}";
    }
}

public class ExemploSerializacao {
    public static void main(String[] args) {
        Pessoa pessoaOriginal = new Pessoa("Alice", 30,
"minhaSenhaSecreta");
        String nomeArquivo = "pessoa.ser";

        // Serialização (Escrevendo o objeto em um arquivo)
        try (FileOutputStream fileOut = new FileOutputStream(nomeArquivo);
             ObjectOutputStream out = new ObjectOutputStream(fileOut)) {
            out.writeObject(pessoaOriginal);
            System.out.println("Objeto Pessoa serializado e salvo em " +
nomeArquivo);
        } catch (IOException i) {
            i.printStackTrace();
        }

        Pessoa pessoaDesserializada = null;

        // Desserialização (Lendo o objeto do arquivo)
        try (FileInputStream fileIn = new FileInputStream(nomeArquivo);
             ObjectInputStream in = new ObjectInputStream(fileIn)) {
```

```
            pessoaDesserializada = (Pessoa) in.readObject();
            System.out.println("Objeto Pessoa desserializado do arquivo.");
        } catch (IOException i) {
            i.printStackTrace();
            return;
        } catch (ClassNotFoundException c) {
            System.out.println("Classe Pessoa não encontrada.");
            c.printStackTrace();
            return;
        }

        System.out.println("Pessoa Original: " + pessoaOriginal);
        System.out.println("Pessoa Desserializada: " +
pessoaDesserializada);
        // Note que a senha será null na desserialização devido ao
'transient'
    }
}
```

8. Networking Básico

Networking em Java permite que aplicações se comuniquem através de uma rede, seja ela local ou a internet. O Java 6 oferece um conjunto de classes robustas no pacote `java.net` para facilitar a programação de rede, com foco principal em **sockets** para comunicação cliente-servidor.

8.1. Sockets (`ServerSocket`, `Socket`)

Um **socket** é um endpoint de comunicação. Ele permite que um programa envie e receba dados através de uma rede. Em Java, existem dois tipos principais de sockets para comunicação TCP/IP:

- **`ServerSocket`**: Usado no lado do servidor para "escutar" por conexões de clientes. Quando um cliente tenta se conectar, o `ServerSocket` aceita a conexão e retorna um objeto `Socket` para lidar com a comunicação com aquele cliente específico.
- **`Socket`**: Usado no lado do cliente para iniciar uma conexão com um servidor e no lado do servidor para se comunicar com um cliente após a conexão ser estabelecida.

8.1.1. Comunicação Cliente-Servidor Básica

Vamos criar um exemplo simples de um servidor que escuta em uma porta e um cliente que se conecta a ele, envia uma mensagem e recebe uma resposta.

Exemplo de Servidor (Echo Server):

Este servidor simplesmente recebe uma mensagem do cliente e a envia de volta (echo).

```java
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;

public class ServidorEcho {
    private static final int PORTA = 12345;

    public static void main(String[] args) {
        System.out.println("Servidor Echo iniciado na porta " + PORTA);
        try (ServerSocket serverSocket = new ServerSocket(PORTA)) {
            while (true) {
                // Espera por uma conexão de cliente
                Socket clientSocket = serverSocket.accept();
                System.out.println("Cliente conectado: " +
clientSocket.getInetAddress().getHostAddress());

                // Cria um novo thread para lidar com o cliente
                new Thread(new ClientHandler(clientSocket)).start();
            }
        } catch (IOException e) {
            System.err.println("Erro no servidor: " + e.getMessage());
        }
    }

    private static class ClientHandler implements Runnable {
        private Socket clientSocket;

        public ClientHandler(Socket socket) {
            this.clientSocket = socket;
        }

        @Override
        public void run() {
            try (
                PrintWriter out = new
PrintWriter(clientSocket.getOutputStream(), true);
                BufferedReader in = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()))
            ) {
                String inputLine;
                while ((inputLine = in.readLine()) != null) {
                    System.out.println("Recebido do cliente " +
```

```
clientSocket.getInetAddress().getHostAddress() + ": " + inputLine);
                    out.println("Echo: " + inputLine); // Envia a mensagem
de volta ao cliente
                    if (inputLine.equals("sair")) {
                        break;
                    }
                }
            } catch (IOException e) {
                System.err.println("Erro ao lidar com o cliente " +
clientSocket.getInetAddress().getHostAddress() + ": " + e.getMessage());
            } finally {
                try {
                    clientSocket.close();
                    System.out.println("Cliente desconectado: " +
clientSocket.getInetAddress().getHostAddress());
                } catch (IOException e) {
                    System.err.println("Erro ao fechar socket do cliente: "
+ e.getMessage());
                }
            }
        }
    }
}
```

Exemplo de Cliente:

```java
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;
import java.util.Scanner;

public class ClienteEcho {
    private static final String HOST = "localhost"; // Ou o IP do servidor
    private static final int PORTA = 12345;

    public static void main(String[] args) {
        System.out.println("Cliente Echo iniciado.");
        try (
            Socket socket = new Socket(HOST, PORTA);
            PrintWriter out = new PrintWriter(socket.getOutputStream(),
true);
            BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
            Scanner scanner = new Scanner(System.in)
        ) {
            String userInput;
            System.out.println("Conectado ao servidor. Digite 'sair' para
encerrar.");
            while (true) {
                System.out.print("Você: ");
                userInput = scanner.nextLine();
                out.println(userInput); // Envia a mensagem para o servidor

                String serverResponse = in.readLine(); // Recebe a resposta
do servidor
                System.out.println("Servidor: " + serverResponse);

                if (userInput.equals("sair")) {
                    break;
                }
            }
        } catch (IOException e) {
            System.err.println("Erro no cliente: " + e.getMessage());
        }
        System.out.println("Cliente encerrado.");
    }
}
```

Como executar:

1. Compile ambos os arquivos Java (`ServidorEcho.java` e `ClienteEcho.java`).
2. Primeiro, execute o `ServidorEcho` em um terminal.
3. Em outro terminal, execute o `ClienteEcho`.
4. Digite mensagens no cliente e veja o servidor respondê-las.

9. JDBC (Java Database Connectivity)

JDBC (Java Database Connectivity) é uma API (Application Programming Interface) que permite que aplicações Java interajam com bancos de dados relacionais. Ela fornece um conjunto padrão de interfaces e classes para conectar-se a um banco de dados, executar comandos SQL e processar os resultados. O JDBC é a base para muitas tecnologias de persistência em Java, como ORMs (Object-Relational Mappers).

9.1. Conexão com Bancos de Dados

Para conectar-se a um banco de dados usando JDBC, os seguintes passos são geralmente necessários:

1. **Carregar o Driver JDBC**: O driver é um software específico do banco de dados que implementa a API JDBC. Ele é carregado na memória usando `Class.forName()`.
2. **Estabelecer a Conexão**: A classe `DriverManager` é usada para obter uma conexão com o banco de dados, fornecendo a URL de conexão, nome de usuário e senha.

Exemplo de Conexão (usando H2 Database em modo de arquivo para simplicidade):

Para este exemplo, você precisará do driver JDBC do H2. Baixe o JAR do H2 e adicione-o ao classpath do seu projeto. Se estiver usando Maven, adicione a dependência:

```xml
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <version>1.4.200</version> <!-- Ou uma versão compatível com Java 6 -->
</dependency>
```

```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class ExemploConexaoJDBC {
    // URL de conexão para um banco de dados H2 em arquivo
    private static final String DB_URL = "jdbc:h2:~/testdb";
    private static final String USER = "sa";
    private static final String PASS = "";

    public static void main(String[] args) {
        Connection conn = null;
        try {
            // 1. Carregar o driver JDBC (para H2, geralmente não é
estritamente necessário em versões mais recentes, mas é boa prática)
            Class.forName("org.h2.Driver");
            System.out.println("Driver JDBC carregado com sucesso.");

            // 2. Estabelecer a conexão
            conn = DriverManager.getConnection(DB_URL, USER, PASS);
            System.out.println("Conexão estabelecida com o banco de dados
H2.");

        } catch (ClassNotFoundException e) {
            System.err.println("Driver JDBC não encontrado: " +
e.getMessage());
        } catch (SQLException e) {
            System.err.println("Erro de conexão com o banco de dados: " +
e.getMessage());
        } finally {
            // Fechar a conexão no bloco finally para garantir que seja
fechada mesmo em caso de erro
            try {
                if (conn != null) {
                    conn.close();
                    System.out.println("Conexão fechada.");
                }
            } catch (SQLException e) {
                System.err.println("Erro ao fechar a conexão: " +
e.getMessage());
            }
        }
    }
}
```

9.2. Execução de Consultas (`SELECT`, `INSERT`, `UPDATE`, `DELETE`)

Após estabelecer uma conexão, você pode executar comandos SQL usando objetos `Statement` ou `PreparedStatement`.

- **`Statement`**: Usado para executar comandos SQL estáticos (sem parâmetros).
- **`PreparedStatement`**: Mais eficiente e seguro (previne SQL Injection) para executar comandos SQL pré-compilados com parâmetros.

9.2.1. Criando uma Tabela

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;

public class CriarTabela {
    private static final String DB_URL = "jdbc:h2:~/testdb";
    private static final String USER = "sa";
    private static final String PASS = "";

    public static void main(String[] args) {
        try (Connection conn = DriverManager.getConnection(DB_URL, USER,
PASS);
             Statement stmt = conn.createStatement()) {

            String sql = "CREATE TABLE IF NOT EXISTS USUARIOS (" +
                         "id INT PRIMARY KEY AUTO_INCREMENT," +
                         "nome VARCHAR(255) NOT NULL," +
                         "email VARCHAR(255) UNIQUE NOT NULL)";
            stmt.execute(sql);
            System.out.println("Tabela USUARIOS criada ou já existente.");

        } catch (SQLException e) {
            System.err.println("Erro ao criar tabela: " + e.getMessage());
        }
    }
}
```

9.2.2. `INSERT` (Inserir Dados)

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;

public class InserirDados {
    private static final String DB_URL = "jdbc:h2:~/testdb";
    private static final String USER = "sa";
    private static final String PASS = "";

    public static void main(String[] args) {
        String sql = "INSERT INTO USUARIOS (nome, email) VALUES (?, ?)";

        try (Connection conn = DriverManager.getConnection(DB_URL, USER,
PASS);
             PreparedStatement pstmt = conn.prepareStatement(sql)) {

            pstmt.setString(1, "Alice Silva");
            pstmt.setString(2, "alice@example.com");
            int linhasAfetadas = pstmt.executeUpdate();
            System.out.println(linhasAfetadas + " linha(s) inserida(s).");

            pstmt.setString(1, "Bob Santos");
            pstmt.setString(2, "bob@example.com");
            linhasAfetadas = pstmt.executeUpdate();
            System.out.println(linhasAfetadas + " linha(s) inserida(s).");

        } catch (SQLException e) {
            System.err.println("Erro ao inserir dados: " + e.getMessage());
        }
    }
}
```

9.2.3. `SELECT` (Consultar Dados)

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class ConsultarDados {
    private static final String DB_URL = "jdbc:h2:~/testdb";
    private static final String USER = "sa";
    private static final String PASS = "";

    public static void main(String[] args) {
        String sql = "SELECT id, nome, email FROM USUARIOS";

        try (Connection conn = DriverManager.getConnection(DB_URL, USER,
PASS);
             Statement stmt = conn.createStatement();
             ResultSet rs = stmt.executeQuery(sql)) {

            System.out.println("\n--- Usuários Cadastrados ---");
            while (rs.next()) {
                int id = rs.getInt("id");
                String nome = rs.getString("nome");
                String email = rs.getString("email");
                System.out.println("ID: " + id + ", Nome: " + nome + ",
Email: " + email);
            }
            System.out.println("----------------------------");

        } catch (SQLException e) {
            System.err.println("Erro ao consultar dados: " +
e.getMessage());
        }
    }
}
```

9.2.4. `UPDATE` (Atualizar Dados)

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;

public class AtualizarDados {
    private static final String DB_URL = "jdbc:h2:~/testdb";
    private static final String USER = "sa";
    private static final String PASS = "";

    public static void main(String[] args) {
        String sql = "UPDATE USUARIOS SET email = ? WHERE nome = ?";

        try (Connection conn = DriverManager.getConnection(DB_URL, USER,
PASS);
             PreparedStatement pstmt = conn.prepareStatement(sql)) {

            pstmt.setString(1, "alice.nova@example.com");
            pstmt.setString(2, "Alice Silva");
            int linhasAfetadas = pstmt.executeUpdate();
            System.out.println(linhasAfetadas + " linha(s) atualizada(s).");

        } catch (SQLException e) {
            System.err.println("Erro ao atualizar dados: " +
e.getMessage());
        }
    }
}
```

9.2.5. `DELETE` (Excluir Dados)

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;

public class ExcluirDados {
    private static final String DB_URL = "jdbc:h2:~/testdb";
    private static final String USER = "sa";
    private static final String PASS = "";

    public static void main(String[] args) {
        String sql = "DELETE FROM USUARIOS WHERE nome = ?";

        try (Connection conn = DriverManager.getConnection(DB_URL, USER,
PASS);
             PreparedStatement pstmt = conn.prepareStatement(sql)) {

            pstmt.setString(1, "Bob Santos");
            int linhasAfetadas = pstmt.executeUpdate();
            System.out.println(linhasAfetadas + " linha(s) excluída(s).");

        } catch (SQLException e) {
            System.err.println("Erro ao excluir dados: " + e.getMessage());
        }
    }
}
```

9.3. Transações

Uma **transação** é uma sequência de operações de banco de dados que são executadas como uma única unidade lógica de trabalho. As transações garantem a **atomicidade**, **consistência**, **isolamento** e **durabilidade (ACID)** dos dados. Em JDBC, as transações são gerenciadas desativando o autocommit e usando `commit()` e `rollback()`.

```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;

public class ExemploTransacao {
    private static final String DB_URL = "jdbc:h2:~/testdb";
    private static final String USER = "sa";
    private static final String PASS = "";

    public static void main(String[] args) {
        Connection conn = null;
        try {
            conn = DriverManager.getConnection(DB_URL, USER, PASS);
            conn.setAutoCommit(false); // Desativa o autocommit

            // Operação 1: Inserir um novo usuário
            String sqlInsert = "INSERT INTO USUARIOS (nome, email) VALUES
(?, ?)";
            try (PreparedStatement pstmtInsert =
conn.prepareStatement(sqlInsert)) {
                pstmtInsert.setString(1, "Carlos Lima");
                pstmtInsert.setString(2, "carlos@example.com");
                pstmtInsert.executeUpdate();
                System.out.println("Usuário Carlos Lima inserido.");
            }

            // Operação 2: Tentar inserir um usuário com email duplicado
(causará erro)
            String sqlInsertDuplicado = "INSERT INTO USUARIOS (nome, email)
VALUES (?, ?)";
            try (PreparedStatement pstmtInsertDuplicado =
conn.prepareStatement(sqlInsertDuplicado)) {
                pstmtInsertDuplicado.setString(1, "Daniel Costa");
                pstmtInsertDuplicado.setString(2, "alice.nova@example.com");
// Email duplicado
                pstmtInsertDuplicado.executeUpdate();
                System.out.println("Usuário Daniel Costa inserido.");
            }

            conn.commit(); // Confirma todas as operações se tudo deu certo
            System.out.println("Transação concluída com sucesso.");

        } catch (SQLException e) {
            System.err.println("Erro na transação: " + e.getMessage());
```

```
            try {
                if (conn != null) {
                    conn.rollback(); // Reverte todas as operações em caso
de erro
                    System.out.println("Transação revertida (rollback).");
                }
            } catch (SQLException rollbackEx) {
                System.err.println("Erro ao fazer rollback: " +
rollbackEx.getMessage());
            }
        } finally {
            try {
                if (conn != null) {
                    conn.setAutoCommit(true); // Reativa o autocommit
                    conn.close();
                    System.out.println("Conexão fechada.");
                }
            } catch (SQLException closeEx) {
                System.err.println("Erro ao fechar a conexão: " +
closeEx.getMessage());
            }
        }
    }
}
```

10. Reflexão e Anotações

Reflexão e **Anotações** são recursos poderosos em Java que permitem inspecionar e manipular classes, métodos e campos em tempo de execução, além de adicionar metadados ao código. Embora sejam ferramentas avançadas, são fundamentais para frameworks, bibliotecas e ferramentas de desenvolvimento.

10.1. Uso de Reflexão para inspecionar e manipular classes e objetos em tempo de execução

A **Reflexão** em Java é a capacidade de um programa examinar ou modificar seu próprio comportamento em tempo de execução. Isso significa que você pode, por exemplo, descobrir o nome de uma classe, seus métodos, seus campos, e até mesmo invocar métodos ou acessar campos privados, tudo isso dinamicamente, sem saber os nomes em tempo de compilação.

Casos de uso comuns para Reflexão:

- **Frameworks e Bibliotecas**: Muitos frameworks (como Spring, Hibernate) usam reflexão para injetar dependências, mapear objetos para bancos de dados, etc.
- **Ferramentas de Desenvolvimento**: Depuradores, IDEs e ferramentas de teste usam reflexão para inspecionar o estado dos objetos.
- **Serialização/Desserialização**: Para converter objetos em formatos de dados e vice-versa.

Principais classes da API de Reflexão:

- `java.lang.Class` : Representa classes e interfaces.
- `java.lang.reflect.Field` : Representa um campo (variável de instância) de uma classe.
- `java.lang.reflect.Method` : Representa um método de uma classe.
- `java.lang.reflect.Constructor` : Representa um construtor de uma classe.

Exemplo de Reflexão:

```java
import java.lang.reflect.Field;
import java.lang.reflect.Method;
import java.lang.reflect.Constructor;

class MinhaClasse {
    private String nome;
    public int valorPublico;

    public MinhaClasse() {
        this.nome = "Default";
        this.valorPublico = 0;
    }

    public MinhaClasse(String nome, int valorPublico) {
        this.nome = nome;
        this.valorPublico = valorPublico;
    }

    public String getNome() {
        return nome;
    }

    private void metodoPrivado() {
        System.out.println("Método privado invocado!");
    }

    public void metodoPublico(String mensagem) {
        System.out.println("Método público: " + mensagem);
    }
}

public class ExemploReflexao {
    public static void main(String[] args) throws Exception {
        // 1. Obtendo o objeto Class
        Class<?> classe = MinhaClasse.class;
        System.out.println("Nome da Classe: " + classe.getName());

        // 2. Acessando Construtores
        System.out.println("\n--- Construtores ---");
        Constructor<?>[] construtores = classe.getConstructors();
        for (Constructor<?> c : construtores) {
            System.out.println("Construtor: " + c.getName() + " com " +
c.getParameterCount() + " parâmetros.");
        }
```

```
        // Criando uma instância usando um construtor específico
        Constructor<?> construtorComParametros =
classe.getConstructor(String.class, int.class);
        MinhaClasse obj = (MinhaClasse)
construtorComParametros.newInstance("Reflexão", 123);
        System.out.println("Instância criada via reflexão: " + obj.getNome()
+ ", " + obj.valorPublico);

        // 3. Acessando Campos (Fields)
        System.out.println("\n--- Campos ---");
        Field[] campos = classe.getDeclaredFields(); // getDeclaredFields()
inclui campos privados
        for (Field f : campos) {
            System.out.println("Campo: " + f.getName() + ", Tipo: " +
f.getType().getName());
        }

        // Acessando e modificando um campo privado
        Field campoNome = classe.getDeclaredField("nome");
        campoNome.setAccessible(true); // Permite acesso a campos privados
        System.out.println("Valor original do campo 'nome': " +
campoNome.get(obj));
        campoNome.set(obj, "Nome Alterado");
        System.out.println("Novo valor do campo 'nome': " +
campoNome.get(obj));

        // Acessando e modificando um campo público
        Field campoValorPublico = classe.getField("valorPublico"); //
getField() apenas campos públicos
        System.out.println("Valor original do campo 'valorPublico': " +
campoValorPublico.get(obj));
        campoValorPublico.set(obj, 456);
        System.out.println("Novo valor do campo 'valorPublico': " +
campoValorPublico.get(obj));

        // 4. Acessando Métodos
        System.out.println("\n--- Métodos ---");
        Method[] metodos = classe.getDeclaredMethods(); //
getDeclaredMethods() inclui métodos privados
        for (Method m : metodos) {
            System.out.println("Método: " + m.getName() + ", Retorno: " +
m.getReturnType().getName());
        }

        // Invocando um método público
        Method metodoPublico = classe.getMethod("metodoPublico",
```

```
String.class);
        metodoPublico.invoke(obj, "Olá da Reflexão!");

        // Invocando um método privado
        Method metodoPrivado = classe.getDeclaredMethod("metodoPrivado");
        metodoPrivado.setAccessible(true); // Permite acesso a métodos
privados
        metodoPrivado.invoke(obj);
    }
}
```

10.2. Criação e uso de Anotações personalizadas

Anotações (Annotations) são uma forma de adicionar metadados ao código-fonte Java. Elas não afetam diretamente a execução do programa, mas podem ser processadas por ferramentas de compilação, frameworks ou em tempo de execução via reflexão para adicionar funcionalidades ou impor regras.

Java 6 introduziu a capacidade de criar anotações personalizadas, que são definidas como interfaces com a palavra-chave `@interface`.

Principais anotações meta-anotações (para definir anotações):

- `@Target`: Indica onde a anotação pode ser aplicada (classe, método, campo, etc.).
- `@Retention`: Indica por quanto tempo a anotação deve ser retida (em tempo de compilação, em tempo de execução, etc.).
- `@Documented`: Indica que a anotação deve ser incluída na documentação Javadoc.
- `@Inherited`: Indica que a anotação é herdada por subclasses.

Exemplo de Anotação Personalizada:

```java
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import java.lang.reflect.Method;

// 1. Definindo uma anotação personalizada
@Retention(RetentionPolicy.RUNTIME) // A anotação estará disponível em tempo de execução via reflexão
@Target(ElementType.METHOD)      // A anotação pode ser aplicada apenas a métodos
@interface ExecutarAoIniciar {
    int ordem() default 0; // Um elemento com valor padrão
}

class TarefasDeInicializacao {

    @ExecutarAoIniciar(ordem = 2)
    public void carregarConfiguracoes() {
        System.out.println("Carregando configurações do sistema...");
    }

    @ExecutarAoIniciar(ordem = 1)
    public void inicializarBancoDeDados() {
        System.out.println("Inicializando conexão com o banco de dados...");
    }

    @ExecutarAoIniciar(ordem = 3)
    public void iniciarServicos() {
        System.out.println("Iniciando serviços da aplicação...");
    }

    public void metodoNormal() {
        System.out.println("Este é um método normal, não anotado para execução.");
    }
}

public class ExemploAnotacoes {
    public static void main(String[] args) throws Exception {
        TarefasDeInicializacao app = new TarefasDeInicializacao();
        Class<?> classe = app.getClass();

        System.out.println("Executando métodos anotados com @ExecutarAoIniciar:");
```

```
        // Percorre todos os métodos da classe
        for (Method metodo : classe.getDeclaredMethods()) {
            // Verifica se o método possui a anotação @ExecutarAoIniciar
            if (metodo.isAnnotationPresent(ExecutarAoIniciar.class)) {
                ExecutarAoIniciar anotacao =
metodo.getAnnotation(ExecutarAoIniciar.class);
                System.out.println("  -> Encontrado método: " +
metodo.getName() + " (Ordem: " + anotacao.ordem() + ")");
                metodo.invoke(app); // Invoca o método anotado
            }
        }

        // Note que a ordem de execução acima pode não ser a desejada (pela
'ordem' da anotação)
        // Para ordenar, seria necessário coletar os métodos e ordená-los
antes de invocar.
        // Isso demonstra como frameworks usam reflexão para encontrar e
processar anotações.
    }
}
```

Neste exemplo, a anotação `@ExecutarAoIniciar` é usada para marcar métodos que devem ser executados durante a inicialização. Em tempo de execução, a reflexão é usada para encontrar esses métodos e invocá-los. Isso é um padrão comum em frameworks que usam anotações para configurar e estender funcionalidades.

11. Boas Práticas e Padrões de Projeto

Desenvolver software de qualidade vai além de apenas escrever código que funciona. Envolve a aplicação de **boas práticas** e o uso de **padrões de projeto** para criar sistemas que sejam legíveis, manuteníveis, escaláveis e testáveis. Esta seção aborda conceitos essenciais para elevar a qualidade do seu código Java.

11.1. Código Limpo (Clean Code)

Código Limpo é um código que é fácil de ler, entender e modificar. É um código que faz o que deveria fazer, sem surpresas, e que é escrito de forma clara e concisa. Seguir os princípios de código limpo reduz a complexidade, melhora a colaboração e diminui a probabilidade de bugs.

Princípios e Práticas Chave:

- **Nomes Significativos**: Use nomes que revelem a intenção. Variáveis, métodos e classes devem ter nomes que descrevam claramente seu propósito.
  - **Ruim**: `int d;` (duração ambígua)
  - **Bom**: `int diasDesdeUltimaModificacao;`
- **Funções Pequenas e Focadas**: Funções (métodos) devem ser pequenas e fazer apenas uma coisa, e fazê-la bem. Isso melhora a legibilidade e a testabilidade.
- **Comentários Apenas Quando Necessário**: O código deve ser autoexplicativo. Comentários devem ser usados para explicar o "porquê" de uma decisão, não o "o quê" (que o código já deveria expressar).
- **Formatação Consistente**: Siga um estilo de formatação consistente (indentação, espaçamento, quebras de linha). Isso melhora a legibilidade e a familiaridade com o código.
- **Tratamento de Erros**: Lide com erros de forma graciosa e consistente. Use exceções para condições excepcionais e retorne valores de erro para condições esperadas.
- **Não Repita Seu Código (DRY - Don't Repeat Yourself)**: Evite duplicação de código. Se você se encontra escrevendo o mesmo código várias vezes, considere refatorá-lo em um método ou classe reutilizável.
- **Princípio da Responsabilidade Única (SRP - Single Responsibility Principle)**: Uma classe deve ter apenas uma razão para mudar. Isso significa que ela deve ter apenas uma responsabilidade.

11.2. Padrões de Projeto (Design Patterns)

Padrões de Projeto são soluções reutilizáveis para problemas comuns de design de software. Eles são descrições ou modelos de como resolver um problema que pode ser usado em muitas situações diferentes. Os padrões de projeto não são classes ou bibliotecas prontas para uso, mas sim diretrizes que podem ser aplicadas para resolver problemas específicos.

Os padrões são categorizados em três tipos:

- **Criacionais**: Lidam com a criação de objetos de forma flexível e controlada.

- **Estruturais**: Lidam com a composição de classes e objetos para formar estruturas maiores.
- **Comportamentais**: Lidam com a comunicação e atribuição de responsabilidades entre objetos.

11.2.1. Padrão Singleton (Criacional)

O padrão Singleton garante que uma classe tenha apenas uma instância e fornece um ponto de acesso global a ela. É útil para recursos que devem ser únicos no sistema, como um gerenciador de configurações ou um pool de conexões de banco de dados.

```
public class Singleton {
    // A única instância da classe
    private static Singleton instancia;

    // Construtor privado para evitar instanciação externa
    private Singleton() {
        System.out.println("Instância Singleton criada.");
    }

    // Método estático para obter a única instância
    public static Singleton getInstancia() {
        if (instancia == null) {
            // Sincronização para garantir thread-safety em ambientes multi-
thread
            synchronized (Singleton.class) {
                if (instancia == null) { // Double-checked locking
                    instancia = new Singleton();
                }
            }
        }
        return instancia;
    }

    public void mostrarMensagem() {
        System.out.println("Olá do Singleton!");
    }

    public static void main(String[] args) {
        Singleton s1 = Singleton.getInstancia();
        Singleton s2 = Singleton.getInstancia();

        System.out.println("s1 e s2 são a mesma instância? " + (s1 == s2));
// true
        s1.mostrarMensagem();
    }
}
```

11.2.2. Padrão Factory Method (Criacional)

O padrão Factory Method define uma interface para criar um objeto, mas permite que as subclasses decidam qual classe instanciar. Isso delega a responsabilidade de instanciação para as subclasses, tornando o sistema mais flexível e extensível.

```java
// Interface comum para os produtos
interface Produto {
    void exibirInfo();
}

// Implementações concretas dos produtos
class ProdutoA implements Produto {
    @Override
    public void exibirInfo() {
        System.out.println("Este é o Produto A.");
    }
}

class ProdutoB implements Produto {
    @Override
    public void exibirInfo() {
        System.out.println("Este é o Produto B.");
    }
}

// Interface do criador (Factory)
abstract class CriadorProduto {
    public abstract Produto criarProduto();

    public void operacao() {
        Produto produto = criarProduto();
        produto.exibirInfo();
    }
}

// Criadores concretos que implementam o Factory Method
class CriadorProdutoA extends CriadorProduto {
    @Override
    public Produto criarProduto() {
        return new ProdutoA();
    }
}

class CriadorProdutoB extends CriadorProduto {
    @Override
    public Produto criarProduto() {
        return new ProdutoB();
    }
}
```

```
public class ExemploFactoryMethod {
    public static void main(String[] args) {
        CriadorProduto criadorA = new CriadorProdutoA();
        criadorA.operacao(); // Saída: Este é o Produto A.

        CriadorProduto criadorB = new CriadorProdutoB();
        criadorB.operacao(); // Saída: Este é o Produto B.
    }
}
```

11.2.3. Padrão Observer (Comportamental)

O padrão Observer 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. É comumente usado em sistemas de eventos e interfaces gráficas.

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

// Interface do Sujeito (Subject)
interface Sujeito {
    void adicionarObservador(Observador o);
    void removerObservador(Observador o);
    void notificarObservadores();
}

// Interface do Observador (Observer)
interface Observador {
    void atualizar(String mensagem);
}

// Implementação concreta do Sujeito
class EstacaoMeteorologica implements Sujeito {
    private List<Observador> observadores = new ArrayList<>();
    private String dadosMeteorologicos;

    public void setDadosMeteorologicos(String dados) {
        this.dadosMeteorologicos = dados;
        notificarObservadores();
    }

    @Override
    public void adicionarObservador(Observador o) {
        observadores.add(o);
    }

    @Override
    public void removerObservador(Observador o) {
        observadores.remove(o);
    }

    @Override
    public void notificarObservadores() {
        for (Observador o : observadores) {
            o.atualizar(dadosMeteorologicos);
        }
    }
}

// Implementações concretas dos Observadores
class DisplayTemperatura implements Observador {
```

```
    private String nomeDisplay;

    public DisplayTemperatura(String nome) {
        this.nomeDisplay = nome;
    }

    @Override
    public void atualizar(String mensagem) {
        System.out.println(nomeDisplay + ": Nova temperatura - " +
mensagem);
    }
}

class DisplayUmidade implements Observador {
    private String nomeDisplay;

    public DisplayUmidade(String nome) {
        this.nomeDisplay = nome;
    }

    @Override
    public void atualizar(String mensagem) {
        System.out.println(nomeDisplay + ": Nova umidade - " + mensagem);
    }
}

public class ExemploObserver {
    public static void main(String[] args) {
        EstacaoMeteorologica estacao = new EstacaoMeteorologica();

        DisplayTemperatura display1 = new DisplayTemperatura("Display da
Sala");
        DisplayUmidade display2 = new DisplayUmidade("Display do Jardim");

        estacao.adicionarObservador(display1);
        estacao.adicionarObservador(display2);

        estacao.setDadosMeteorologicos("25°C, 60% umidade");
        estacao.setDadosMeteorologicos("28°C, 55% umidade");

        estacao.removerObservador(display1);
        estacao.setDadosMeteorologicos("22°C, 70% umidade"); // Apenas
display2 será notificado
    }
}
```

11.3. Testes Unitários (JUnit)

Testes Unitários são uma parte crucial do desenvolvimento de software de qualidade. Eles envolvem testar pequenas unidades de código (geralmente métodos ou classes) isoladamente para garantir que funcionem conforme o esperado. **JUnit** é o framework de teste unitário mais popular para Java.

Benefícios dos Testes Unitários:

- **Detecção Precoce de Bugs**: Ajuda a encontrar erros no início do ciclo de desenvolvimento.
- **Melhora a Qualidade do Código**: Força o desenvolvedor a escrever código mais modular e testável.
- **Facilita a Refatoração**: Permite fazer alterações no código com confiança, sabendo que os testes irão alertar sobre regressões.
- **Documentação Viva**: Os testes servem como exemplos de como o código deve ser usado.

Configuração do JUnit (para Java 6, geralmente JUnit 4):

Adicione a dependência do JUnit ao seu projeto. Se estiver usando Maven:

```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.13.2</version> <!-- Ou uma versão compatível com Java 6, como
4.x -->
    <scope>test</scope>
</dependency>
```

Exemplo de Teste Unitário com JUnit:

Considere uma classe simples para realizar operações matemáticas:

```java
// src/main/java/com/example/Calculadora.java
package com.example;

public class Calculadora {
    public int somar(int a, int b) {
        return a + b;
    }

    public int subtrair(int a, int b) {
        return a - b;
    }

    public int multiplicar(int a, int b) {
        return a * b;
    }

    public double dividir(int a, int b) {
        if (b == 0) {
            throw new IllegalArgumentException("Divisão por zero não
permitida.");
        }
        return (double) a / b;
    }
}
```

Agora, o teste unitário para esta classe:

```java
// src/test/java/com/example/CalculadoraTest.java
package com.example;

import org.junit.Test;
import static org.junit.Assert.*;

public class CalculadoraTest {

    @Test
    public void testSomar() {
        Calculadora calc = new Calculadora();
        assertEquals("A soma de 2 e 3 deve ser 5", 5, calc.somar(2, 3));
        assertEquals("A soma de -1 e 1 deve ser 0", 0, calc.somar(-1, 1));
        assertEquals("A soma de 0 e 0 deve ser 0", 0, calc.somar(0, 0));
    }

    @Test
    public void testSubtrair() {
        Calculadora calc = new Calculadora();
        assertEquals("A subtração de 5 e 2 deve ser 3", 3, calc.subtrair(5,
2));
        assertEquals("A subtração de 2 e 5 deve ser -3", -3,
calc.subtrair(2, 5));
    }

    @Test
    public void testMultiplicar() {
        Calculadora calc = new Calculadora();
        assertEquals("A multiplicação de 2 e 3 deve ser 6", 6,
calc.multiplicar(2, 3));
        assertEquals("A multiplicação de -2 e 3 deve ser -6", -6,
calc.multiplicar(-2, 3));
        assertEquals("A multiplicação de 0 e 5 deve ser 0", 0,
calc.multiplicar(0, 5));
    }

    @Test
    public void testDividir() {
        Calculadora calc = new Calculadora();
        assertEquals("A divisão de 6 por 2 deve ser 3.0", 3.0,
calc.dividir(6, 2), 0.001); // O.001 é a margem de erro para doubles
        assertEquals("A divisão de 7 por 2 deve ser 3.5", 3.5,
calc.dividir(7, 2), 0.001);
    }
```

```
    @Test(expected = IllegalArgumentException.class)
    public void testDividirPorZero() {
        Calculadora calc = new Calculadora();
        calc.dividir(10, 0); // Espera-se que lance IllegalArgumentException
    }
}
```

Executando Testes:

- Em IDEs como Eclipse ou IntelliJ IDEA, você pode clicar com o botão direito no arquivo de teste e selecionar “Run as JUnit Test”.
- Via Maven, execute `mvn test` no terminal.

Ao aplicar essas boas práticas e padrões de projeto, você estará apto a desenvolver aplicações Java 6 mais robustas, manuteníveis e de alta qualidade.

12. Referências

12. Referências

[1] Oracle. *Java SE 6 Documentation*. Disponível em: https://docs.oracle.com/javase/6/docs/ [2] Bloch, Joshua. *Effective Java*. 2nd ed. Addison-Wesley, 2008. [3] Gamma, Erich, et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994. [4] Martin, Robert C. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008. [5] Goetz, Brian, et al. *Java Concurrency in Practice*. Addison-Wesley, 2006. [6] H2 Database. *H2 Database Engine Documentation*. Disponível em: http://www.h2database.com/html/main.html [7] JUnit. *JUnit 4 Documentation*. Disponível em: https://junit.org/junit4/