Lógica & Estruturas de Dados

Review Notebook

Victor Lopes Dos Santos

Contents

1 Conceitos Básicos **1**
1.1 O que é Programação? . . . 1
1.2 Compilação vs Interpretação . . . 1
1.2.1 Linguagens Compiladas . . . 1
1.2.2 Linguagens Interpretadas . . . 2
1.2.3 Abordagem Híbrida do Java . . . 3
1.3 Conceitos de Memória . . . 3
1.3.1 Organização da Memória . . . 3
1.3.2 Variáveis na Memória . . . 3
1.4 Sintaxe, Semântica e Pragmática . . . 4
1.4.1 Sintaxe . . . 4
1.4.2 Semântica . . . 4
1.4.3 Pragmática . . . 5
1.5 Estrutura Básica de um Programa . . . 5
1.5.1 Programa Java Mínimo . . . 5
1.5.2 Componentes Principais . . . 5
1.6 Ferramentas de Desenvolvimento . . . 6
1.6.1 Ambiente de Desenvolvimento Integrado (IDE) . . . 6
1.6.2 Interface de Linha de Comando (CLI) . . . 6
1.7 Tipos de Dados e Sistemas de Tipos . . . 7
1.7.1 Tipagem Estática vs Dinâmica . . . 7
1.8 Variáveis, Constantes e Escopo . . . 7
1.8.1 Escopo e Tempo de Vida . . . 7
1.9 Expressões e Operadores . . . 8
1.10 Estruturas de Controle . . . 8

1.10.1 Execução Sequencial 8
1.10.2 Instruções Condicionais 8
1.10.3 Estruturas de Repetição 9
1.11 Funções e Métodos 9
1.12 Entrada e Saída 10
1.13 Idiomas de Linguagem 10
1.13.1 Características dos Idiomas de Linguagem 10
1.13.2 Exemplos de Idiomas Java 10
1.14 Paradigmas de Programação 11
1.14.1 Programação Imperativa 11
1.14.2 Programação Procedural 12
1.14.3 Programação Orientada a Objetos (POO) 12
1.14.4 Programação Funcional 13
1.14.5 Linguagens Multi-Paradigma 13

2 Conceitos de Programação **15**
2.1 Variáveis e Tipos de Dados 15
2.1.1 Declaração de Variáveis e Escopo 15
2.1.2 Hierarquia de Tipos de Dados 15
2.1.3 Conversão e Casting de Tipos 16
2.2 Expressões e Declarações 16
2.2.1 Avaliação de Expressões 16
2.2.2 Tipos de Declarações 17

3 Estruturas de Decisão **19**
3.1 Lógica Condicional 19
3.1.1 Fundamentos da Álgebra Booleana 19
3.1.2 Expressões Condicionais Complexas 19
3.2 Estruturas If-Else 19
3.2.1 Cadeia If-Else-If 19
3.2.2 Detalhes da Declaração Switch 20
3.3 Pattern Matching 21
3.3.1 Padrões de Tipo (Java 16+) 21

4 Estruturas de Repetição **23**
4.1 Padrões de Laço 23

4.1.1 Padrão Acumulador 23
4.1.2 Padrão de Busca 24
4.1.3 Laços Controlados por Contagem vs Eventos 24
4.2 Laços Aninhados 25
4.2.1 Tabela de Multiplicação 25
4.2.2 Geração de Padrões 25
4.3 Otimização de Laços 26
4.3.1 Movimento de Código Invariante do Laço 26
4.3.2 Desenrolamento de Laço 27
4.4 Recursão vs Iteração 27
4.4.1 Comparação de Fatorial 27
4.4.2 Quando Usar Recursão 28

5 Estruturas de Dados Básicas **29**
5.1 Arrays 29
5.1.1 Variações de Declaração de Array 29
5.1.2 Algoritmos de Array 29
5.1.3 Utilitários de Array 31
5.2 ArrayLists (Introdução) 31
5.2.1 Operações Básicas 31
5.2.2 ArrayList vs Array 32
5.3 Problemas Comuns de Estruturas de Dados 33
5.3.1 Problema Two Sum 33
5.3.2 Subarray Máximo 33

6 Manipulação de Strings **35**
6.1 Fundamentos de String 35
6.1.1 Criação de String 35
6.1.2 String Pool 35
6.2 Métodos de String 36
6.2.1 Operações Básicas 36
6.2.2 Busca e Comparação 36
6.3 Construção de String 37
6.3.1 StringBuilder vs StringBuffer 37
6.3.2 Considerações de Eficiência 38

6.4 Formatação de String 38
6.4.1 Formatação estilo printf 38
6.5 Expressões Regulares 39
6.5.1 Padrões Básicos 39
6.6 Aplicações Práticas 40
6.6.1 Verificação de Palíndromo 40
6.6.2 Contagem de Palavras 41

7 Respostas das Questões de Revisão **43**
7.1 Respostas do Capítulo 1 43
7.2 Respostas do Capítulo 2 44
7.3 Respostas do Capítulo 3 45
7.4 Respostas do Capítulo 4 46
7.5 Respostas do Capítulo 5 48
7.6 Respostas do Capítulo 6 50

Chapter 1
Conceitos Básicos

Section 1.1: O que é Programação?

Programação é o processo de projetar e construir programas de computador executáveis para realizar tarefas específicas. Consiste em traduzir problemas do mundo real em instruções precisas que um computador possa entender e executar.

No seu cerne, a programação envolve:

- **Decomposição de problemas**: Dividir um problema complexo em partes menores e gerenciáveis.
- **Projeto de algoritmos**: Definir um procedimento passo a passo para resolver o problema.
- **Implementação**: Traduzir o algoritmo para uma linguagem de programação.
- **Teste e depuração**: Verificar a correção e corrigir erros.

A programação requer raciocínio lógico, abstração e atenção aos detalhes. Um programa bem escrito não deve apenas funcionar corretamente, mas também ser legível, eficiente e de fácil manutenção.

Section 1.2: Compilação vs Interpretação

As linguagens de programação diferem na forma como seu código-fonte é transformado em instruções executáveis. As duas principais abordagens são compilação e interpretação.

1.2.1 Linguagens Compiladas

Nas linguagens compiladas, todo o código-fonte é traduzido para código de máquina antes da execução.

- **Processo**: Código-fonte → Compilador → Código de máquina → Execução
- **Exemplos**: C, C++, Rust, Go, Java (parcialmente)
- **Vantagens**:
  - Execução mais rápida
  - Melhores otimizações de desempenho
  - Erros detectados antes da execução
- **Desvantagens**:
  - Binários dependentes de plataforma
  - Tempo de compilação mais longo

1.2.2 Linguagens Interpretadas

Linguagens interpretadas executam o código através de um interpretador em tempo de execução. O interpretador lê e executa o código-fonte linha por linha ou em pequenos blocos sem uma etapa de compilação separada.

- **Processo**: Código-fonte → Interpretador → Execução direta
- **Exemplos**: Python, JavaScript, Ruby, PHP
- **Vantagens**:
  - Independência de plataforma (executa em qualquer sistema com o interpretador)
  - Ciclo de desenvolvimento mais rápido (sem necessidade de compilação)
  - Recursos dinâmicos (modificação de código em tempo de execução, tipagem dinâmica)
  - Depuração mais fácil (feedback imediato de erros)
- **Desvantagens**:
  - Execução mais lenta (sobrecarga de interpretação)
  - Erros detectados apenas em tempo de execução
  - Menos oportunidades de otimização

Muitas linguagens modernas como Python e JavaScript usam abordagens híbridas com compilação Just-In-Time (JIT) para melhorar o desempenho.

1.2.3 Abordagem Híbrida do Java

Java adota um modelo de compilação híbrido, combinando vantagens de ambas as abordagens.

Código-fonte (.java) → Compilador (javac) → Bytecode (.class) → JVM (java) → Execução específica da máquina

O código-fonte é primeiro compilado em uma representação intermediária chamada *bytecode*. Este bytecode é então executado pela Java Virtual Machine (JVM), que o traduz para código de máquina específico do hardware subjacente.

Section 1.3: Conceitos de Memória

Compreender como a memória é organizada é fundamental para escrever programas eficientes e corretos.

1.3.1 Organização da Memória

Um programa em execução normalmente usa várias regiões de memória, cada uma servindo a um propósito específico:

- **Pilha (Stack)**: Armazena chamadas de funções, parâmetros e variáveis locais. Opera de maneira Last-In-First-Out (LIFO).
- **Heap**: Armazena memória alocada dinamicamente, como objetos e estruturas de dados criadas em tempo de execução.
- **Estática/Global**: Mantém variáveis globais e campos estáticos cujo tempo de vida abrange toda a execução do programa.
- **Código/Texto**: Contém as instruções compiladas do programa.

1.3.2 Variáveis na Memória

Diferentes tipos de variáveis são armazenados em diferentes regiões de memória:

```
int x = 10;         // Pilha: armazena o valor 10
double y = 3.14;    // Pilha: armazena 3.14
String s = "Hello"; // Pilha: referência, Heap: objeto string
```

Tipos primitivos geralmente armazenam seus valores diretamente, enquanto tipos de referência armazenam endereços de memória apontando para objetos no heap.

Section 1.4: Sintaxe, Semântica e Pragmática

As linguagens de programação podem ser analisadas a partir de três perspectivas complementares.

1.4.1 Sintaxe

A sintaxe define as regras formais que determinam quais sequências de símbolos formam programas válidos.

```
// Sintaxe válida
int x = 10;
if (x > 5) { ... }

// Sintaxe inválida
int x = 10      // Ponto e vírgula faltando
if x > 5        // Parênteses faltando
```

Erros de sintaxe são detectados pelo compilador ou interpretador antes da execução.

1.4.2 Semântica

Semântica refere-se ao significado de programas sintaticamente válidos. Um programa pode estar sintaticamente correto mas semanticamente incorreto se não produzir o comportamento pretendido.

```
// Semanticamente correto
int x = 10;
int y = x + 5;  // y recebe o valor 15

// Exemplos semanticamente incorretos:
int a = 10;
int b = 0;
int c = a / b;  // Divisão por zero - erro em tempo de execução

String str = null;
int length = str.length();  // NullPointerException

boolean flag = true;
if (flag = false) {  // Atribuição em vez de comparação
    // Este bloco nunca será executado
}
```

1.4.3 Pragmática

A pragmática diz respeito a como os programas são escritos e usados na prática, focando em aspectos práticos além da correção formal.

- **Legibilidade e clareza**: Usar nomes significativos, formatação consistente
- **Manutenibilidade**: Escrever código modular e bem documentado
- **Desempenho**: Considerar eficiência de tempo e espaço
- **Segurança**: Proteger contra vulnerabilidades
- **Colaboração em equipe**: Seguir padrões e convenções de codificação

Section 1.5: Estrutura Básica de um Programa

1.5.1 Programa Java Mínimo

Toda aplicação Java deve definir uma classe contendo um método `main`, que serve como ponto de entrada do programa.

```
public class NomeDoPrograma {
    public static void main(String[] args) {
        // A execução do programa começa aqui
    }
}
```

1.5.2 Componentes Principais

- `class`: Define um modelo para objetos.
- `main`: Ponto de entrada do programa.
- `public`: Torna o método acessível à JVM.
- `static`: Permite execução sem criar um objeto.
- `void`: Indica nenhum valor de retorno.
- `String[] args`: Recebe argumentos da linha de comando.

Section 1.6: Ferramentas de Desenvolvimento

1.6.1 Ambiente de Desenvolvimento Integrado (IDE)

Um IDE fornece ferramentas abrangentes para auxiliar o desenvolvimento de software:

- Editor de código com realce de sintaxe e autocompletar
- Integração de compilador e interpretador
- Ferramentas de depuração com pontos de interrupção e inspeção de variáveis
- Suporte a controle de versão (integração Git)
- Gerenciamento de projeto e dependências
- Integração de frameworks de teste

IDEs Java populares incluem IntelliJ IDEA, Eclipse e NetBeans.

1.6.2 Interface de Linha de Comando (CLI)

A Interface de Linha de Comando (CLI) é uma interface baseada em texto onde os usuários digitam comandos para executar tarefas. É uma ferramenta poderosa para desenvolvedores porque permite controle preciso e automação de tarefas. O uso básico da CLI inclui:

- Comandos são digitados em um terminal ou prompt de comando
- Cada comando executa uma ação específica
- Comandos podem ser combinados e automatizados

Ferramentas CLI do Java

Java inclui várias ferramentas de linha de comando para desenvolvimento:

```
# Compilar código-fonte Java
javac Programa.java

# Executar programa Java compilado
java Programa

# Criar documentação
javadoc Programa.java

# Empacotar em arquivo JAR
jar cvf programa.jar *.class
```

Aprender a usar a CLI de forma eficaz pode torná-lo um desenvolvedor mais eficiente, especialmente ao trabalhar com servidores, automação ou processos de compilação complexos.

Section 1.7: Tipos de Dados e Sistemas de Tipos

Um **tipo de dados** especifica o tipo de valores que podem ser armazenados e as operações que podem ser realizadas sobre eles.

Categorias comuns de tipos de dados incluem:

- **Tipos primitivos**: inteiros, números de ponto flutuante, caracteres, booleanos
- **Tipos compostos**: arrays, registros, estruturas
- **Tipos de referência**: objetos, strings, coleções

Um **sistema de tipos** aplica regras que governam como diferentes tipos interagem, ajudando a detectar erros e melhorar a confiabilidade do programa.

1.7.1 Tipagem Estática vs Dinâmica

- **Tipagem estática**: Os tipos são verificados em tempo de compilação (ex: Java, C++).
- **Tipagem dinâmica**: Os tipos são verificados em tempo de execução (ex: Python, JavaScript).

Section 1.8: Variáveis, Constantes e Escopo

Uma **variável** é uma localização de memória nomeada cujo valor pode mudar durante a execução. Uma **constante** representa um valor fixo que não pode ser alterado após a inicialização.

```
final int MAX = 100;
int contador = 0;
```

1.8.1 Escopo e Tempo de Vida

O escopo define onde uma variável é acessível, enquanto o tempo de vida determina por quanto tempo ela existe na memória.

- **Escopo local**: Variáveis declaradas dentro de métodos ou blocos.
- **Escopo de classe ou global**: Variáveis acessíveis em toda uma classe ou programa.

Section 1.9: Expressões e Operadores

Uma **expressão** combina variáveis, literais e operadores para produzir um valor.

```
int resultado = (a + b) * c;
```

Os operadores são comumente classificados como:

- Operadores aritméticos: `+ - * / %`
- Operadores relacionais: `== != > < >= <=`
- Operadores lógicos: `&& || !`
- Operadores de atribuição: `= += -= *=`

A precedência e associatividade dos operadores determinam a ordem de avaliação nas expressões.

Section 1.10: Estruturas de Controle

Estruturas de controle determinam o fluxo de execução em um programa, permitindo execução seletiva e repetitiva de blocos de código.

1.10.1 Execução Sequencial

As instruções são executadas na ordem em que aparecem, a menos que alteradas por estruturas de controle.

1.10.2 Instruções Condicionais

Instruções condicionais permitem tomada de decisão com base em expressões lógicas.

```
if (x > 0) {
    // positivo
} else if (x < 0) {
```

```
    // negativo
} else {
    // zero
}
```

1.10.3 Estruturas de Repetição

Laços permitem execução repetida de blocos de código.

```
for (int i = 0; i < 10; i++) {
    // repetido 10 vezes
}

while (condição) {
    // repetido enquanto a condição for verdadeira
}

do {
    // executado pelo menos uma vez
} while (condição);
```

Section 1.11: Funções e Métodos

Funções (ou métodos) encapsulam unidades reutilizáveis de comportamento e promovem modularidade.

```
int soma(int a, int b) {
    return a + b;
}
```

Funções melhoram:

- Reutilização de código
- Abstração
- Organização do programa
- Testabilidade

Section 1.12: Entrada e Saída

Mecanismos de Entrada e Saída (E/S) permitem que os programas interajam com usuários e sistemas externos.

```
Scanner sc = new Scanner(System.in);
int valor = sc.nextInt();
System.out.println(valor);
```

Operações de E/S podem envolver:

- Entrada e saída padrão
- Arquivos
- Dispositivos externos ou redes

Section 1.13: Idiomas de Linguagem

Idiomas de programação são padrões recorrentes de código que representam a forma mais natural, eficiente e legível de realizar tarefas comuns dentro de uma linguagem de programação específica. Esses padrões emergem da filosofia de design da linguagem, características de sintaxe e convenções da comunidade ao longo do tempo.

1.13.1 Características dos Idiomas de Linguagem

- **Expressividade**: Idiomas permitem que operações complexas sejam escritas de forma concisa mantendo clareza
- **Desempenho**: Código idiomático frequentemente aproveita otimizações da linguagem e evita armadilhas comuns
- **Manutenibilidade**: Seguir idiomas estabelecidos torna o código mais previsível para outros desenvolvedores
- **Alinhamento Cultural**: Idiomas refletem o entendimento compartilhado e as melhores práticas da comunidade de programação

1.13.2 Exemplos de Idiomas Java

```
// Laço for aprimorado (for-each) para coleções
List<String> nomes = Arrays.asList("Alice", "Bob", "Charlie");
```

```
for (String nome : nomes) {
    System.out.println(nome);
}

// Try-with-resources para gerenciamento automático de recursos
try (BufferedReader reader = new BufferedReader(new FileReader("arquivo.txt"))) {
    String linha = reader.readLine();
}

// Padrão Builder para criação complexa de objetos
Pessoa pessoa = new Pessoa.Builder()
    .nome("João")
    .idade(30)
    .endereco("Rua Principal, 123")
    .build();

// Métodos de fábrica para coleções imutáveis
List<String> lista = List.of("a", "b", "c");
Set<Integer> conjunto = Set.of(1, 2, 3);

// Optional para manipulação segura de valores nulos
Optional<String> opcional = Optional.ofNullable(getString());
String resultado = opcional.orElse("padrão");
```

Section 1.14: Paradigmas de Programação

Um paradigma de programação é um estilo fundamental de programação de computadores que define como os programadores estruturam e organizam o código. Paradigmas fornecem diferentes abordagens para decomposição de problemas, organização de dados e gerenciamento de fluxo de controle.

1.14.1 Programação Imperativa

- **Filosofia**: Foca em *como* alcançar resultados através de comandos explícitos
- **Características**: Sequência de declarações que alteram o estado do programa
- **Conceitos-chave**: Variáveis, atribuição, laços, condicionais
- **Linguagens de Exemplo**: C, Pascal, versões antigas do BASIC

```
// Abordagem imperativa para somar um array
int soma = 0;
for (int i = 0; i < numeros.length; i++) {
```

```
    soma += numeros[i];
}
```

1.14.2 Programação Procedural

- **Filosofia**: Estende a programação imperativa com procedimentos/funções
- **Características**: Programas organizados em procedimentos reutilizáveis
- **Conceitos-chave**: Funções, parâmetros, variáveis locais, modularidade
- **Linguagens de Exemplo**: C, Pascal, Fortran

```
// Abordagem procedural com funções
public int calcularSoma(int[] numeros) {
    int soma = 0;
    for (int num : numeros) {
        soma += num;
    }
    return soma;
}

public double calcularMedia(int[] numeros) {
    return calcularSoma(numeros) / (double) numeros.length;
}
```

1.14.3 Programação Orientada a Objetos (POO)

- **Filosofia**: Modela entidades do mundo real como objetos com estado e comportamento
- **Características**: Encapsulamento, herança, polimorfismo, abstração
- **Conceitos-chave**: Classes, objetos, métodos, interfaces, herança
- **Linguagens de Exemplo**: Java, C++, Python, C#

```
// Abordagem POO com classes e objetos
public class ContaBancaria {
    private double saldo;

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

    public void depositar(double quantia) {
```

```
        if (quantia > 0) {
            saldo += quantia;
        }
    }

    public double getSaldo() {
        return saldo;
    }
}

// Uso
ContaBancaria conta = new ContaBancaria(1000);
conta.depositar(500);
```

1.14.4 Programação Funcional

- **Filosofia**: Trata a computação como avaliação de funções matemáticas
- **Características**: Dados imutáveis, funções de primeira classe, sem efeitos colaterais
- **Conceitos-chave**: Funções puras, funções de alta ordem, recursão
- **Linguagens de Exemplo**: Haskell, Lisp, Scala, JavaScript (suporta)

```
// Abordagem funcional em Java (usando Streams)
List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5);
int soma = numeros.stream()
                .filter(n -> n % 2 == 0)      // Manter números pares
                .map(n -> n * 2)              // Dobrá-los
                .reduce(0, Integer::sum);     // Somar todos os valores
```

1.14.5 Linguagens Multi-Paradigma

A maioria das linguagens de programação modernas suporta múltiplos paradigmas, permitindo que os desenvolvedores escolham a abordagem mais apropriada para cada problema:

- **Java**: Principalmente POO, mas suporta imperativo, procedural e funcional (via Streams)
- **Python**: Suporta POO, imperativo, procedural e paradigmas funcionais
- **JavaScript**: Suporta POO, imperativo, procedural e paradigmas funcionais
- **Scala**: Mistura POO e programação funcional de forma perfeita
- **C++**: Suporta POO, imperativo, procedural e programação genérica

Chapter 2
Conceitos de Programação

Section 2.1: Variáveis e Tipos de Dados

2.1.1 Declaração de Variáveis e Escopo

```
// Variáveis locais (escopo de método)
public void metodo() {
    int variavelLocal = 10;  // Acessível apenas dentro do método
}

// Variáveis de instância (escopo de objeto)
public class MinhaClasse {
    private int variavelInstancia;  // Cada objeto tem sua própria cópia
}

// Variáveis de classe (escopo de classe)
public class MinhaClasse {
    private static int variavelClasse;  // Compartilhada por todos os objetos
}
```

2.1.2 Hierarquia de Tipos de Dados

Os tipos de dados Java são organizados em uma estrutura hierárquica:

- **Tipos Primitivos**
  - **Tipos inteiros**: byte (8-bit), short (16-bit), int (32-bit), long (64-bit)
  - **Ponto flutuante**: float (32-bit), double (64-bit)
  - **Caractere**: char (16-bit Unicode)
  - **Booleano**: boolean (verdadeiro/falso)
- **Tipos de Referência**

- **Classes**: String, Object, classes personalizadas
- **Arrays**: Uni e multidimensionais
- **Interfaces**: Runnable, Comparable, List
- **Enumerações**: Tipos Enum

2.1.3 Conversão e Casting de Tipos

```
// Conversão implícita (alargamento) - sem perda de dados
int i = 100;
long l = i;           // OK: int para long
double d = i;         // OK: int para double

// Casting explícito (estreitamento) - potencial perda de dados
double preco = 19.99;
int precoInteiro = (int) preco;  // 19 (truncamento)
long grande = 10000000000L;
int pequeno = (int) grande;       // Possível overflow

// Casos especiais
char c = 'A';
int ascii = c;               // 65 (valor ASCII)
boolean flag = true;
// int num = (int) flag;     // ERRO: não pode converter boolean para int

// Conversões de String
int num = Integer.parseInt("123");
String str = String.valueOf(456);
```

Section 2.2: Expressões e Declarações

2.2.1 Avaliação de Expressões

```
// Expressão complexa com precedência
int resultado = (a + b) * c / d % e;

// Múltiplas atribuições
int x = 5, y = 10, z = 15;

// Expressão com efeitos colaterais
int contador = 0;
int valor = ++contador * 2;  // contador=1, valor=2
```

```
// Atribuição composta
x += 5;    // Equivalente a x = x + 5
x *= y;    // Equivalente a x = x * y
```

2.2.2 Tipos de Declarações

- **Declaração**: `int x;`
- **Atribuição**: `x = 10;`
- **Chamada de método**: `System.out.println("Olá");`
- **Fluxo de controle**: `if, for, while, switch, etc.`
- **Bloco**: `{ declarações }`
- **Retorno**: `return valor;`
- **Break/Continue**: Alteram a execução do laço

Questões de Revisão: Capítulo 2

1. Explique os diferentes escopos de variáveis em Java com exemplos. Quando você usaria cada um?
2. Compare tipos primitivos e tipos de referência. Quais são as implicações para memória e desempenho?
3. Dada a expressão: `int x = 5 + 3 * 2 / (4 - 2);` Qual é o valor de x? Mostre a avaliação passo a passo.
4. Escreva um programa que demonstre todos os operadores de atribuição composta e mostre os resultados.
5. O que é casting de tipos? Quando é permitido casting implícito e quando é necessário casting explícito?
6. Escreva código que converta entre: int e double, char e int, boolean e String.

Chapter 3
Estruturas de Decisão

Section 3.1: Lógica Condicional

3.1.1 Fundamentos da Álgebra Booleana

- **Leis de Identidade**: $A \wedge true = A$, $A \vee false = A$
- **Leis de Dominação**: $A \wedge false = false$, $A \vee true = true$
- **Leis de Idempotência**: $A \wedge A = A$, $A \vee A = A$
- **Dupla Negação**: $\neg(\neg A) = A$
- **Leis de De Morgan**: $\neg(A \wedge B) = \neg A \vee \neg B$, $\neg(A \vee B) = \neg A \wedge \neg B$

3.1.2 Expressões Condicionais Complexas

```
// Múltiplas condições com agrupamento adequado
if ((idade >= 18 && temHabilitacao) || (idade >= 16 && comPais)) {
    // Pode dirigir
}

// Condições aninhadas simplificadas
boolean podeVotar = idade >= 18 && eCidadao && !temCrime;
boolean podeBeber = idade >= 21 && !temSuspensaoSoberania;
```

Section 3.2: Estruturas If-Else

3.2.1 Cadeia If-Else-If

```
if (pontuacao >= 90) {
```

```
    nota = 'A';
} else if (pontuacao >= 80) {
    nota = 'B';
} else if (pontuacao >= 70) {
    nota = 'C';
} else if (pontuacao >= 60) {
    nota = 'D';
} else {
    nota = 'F';
}

// Equivalente usando ternário (menos legível)
nota = (pontuacao >= 90) ? 'A' :
        (pontuacao >= 80) ? 'B' :
        (pontuacao >= 70) ? 'C' :
        (pontuacao >= 60) ? 'D' : 'F';
```

3.2.2 Detalhes da Declaração Switch

```
// Switch tradicional (pré-Java 14)
switch (mes) {
    case 1: case 3: case 5: case 7: case 8: case 10: case 12:
        dias = 31;
        break;
    case 4: case 6: case 9: case 11:
        dias = 30;
        break;
    case 2:
        dias = ehAnoBissexto ? 29 : 28;
        break;
    default:
        throw new IllegalArgumentException("Mês inválido");
}

// Expressão switch (Java 14+)
String estacao = switch (mes) {
    case 12, 1, 2 -> "Inverno";
    case 3, 4, 5 -> "Primavera";
    case 6, 7, 8 -> "Verão";
    case 9, 10, 11 -> "Outono";
    default -> {
        System.err.println("Mês inválido: " + mes);
        yield "Inválido";
    }
};
```

```
// Switch com pattern matching (Java 21 preview)
String descricao = switch (obj) {
    case Integer i -> "Inteiro: " + i;
    case String s -> "String: " + s;
    case null -> "Objeto nulo";
    default -> "Tipo desconhecido";
};
```

Section 3.3: Pattern Matching

O pattern matching simplifica padrões comuns de codificação combinando verificação de tipo e atribuição de variável.

3.3.1 Padrões de Tipo (Java 16+)

```
Object obj = "Olá";

// Abordagem tradicional
if (obj instanceof String) {
    String s = (String) obj;
    System.out.println(s.length());
}

// Pattern matching
if (obj instanceof String s) {
    System.out.println(s.length());  // s é convertido automaticamente
    // s está no escopo aqui
}

// Pode ser usado em laços while
while (obj instanceof String s && s.length() > 0) {
    System.out.println(s.charAt(0));
    obj = s.substring(1);
}
```

Questões de Revisão: Capítulo 3

1. Simplifique a expressão booleana: `!(a && b) || (!a && !b) || (a && !b)`

2. Dados três números, escreva código para encontrar o máximo sem usar Math.max().
3. Explique a diferença entre igualdade profunda e superficial para objetos. Como isso se aplica a Strings?
4. Escreva um programa que valide uma data (dia, mês, ano) considerando anos bissextos.

Chapter 4
Estruturas de Repetição

Section 4.1: Padrões de Laço

Padrões comuns de laço fornecem modelos reutilizáveis para resolver problemas recorrentes.

4.1.1 Padrão Acumulador

Usado para acumular valores através da iteração:

```
// Soma de elementos de array
int[] numeros = {1, 2, 3, 4, 5};
int soma = 0;

for (int num : numeros) {
    soma += num;  // Acumula
}
System.out.println("Soma: " + soma);

// Acumulação de produto
double[] valores = {1.5, 2.0, 3.5, 4.0};
double produto = 1.0;
for (double val : valores) {
    produto *= val;
}

// Acumulador de concatenação de string
String[] palavras = {"Olá", " ", "Mundo", "!"};
StringBuilder mensagem = new StringBuilder();
for (String palavra : palavras) {
    mensagem.append(palavra);
}
```

4.1.2 Padrão de Busca

Usado para encontrar elementos que correspondam a critérios específicos:

```
// Busca linear
int[] array = {10, 20, 30, 40, 50};
int alvo = 30;
boolean encontrado = false;
int indice = -1;

for (int i = 0; i < array.length; i++) {
    if (array[i] == alvo) {
        encontrado = true;
        indice = i;
        break;  // Saída antecipada
    }
}

// Encontrar valor máximo
int max = Integer.MIN_VALUE;
for (int valor : array) {
    if (valor > max) {
        max = valor;
    }
}

// Encontrar todas as correspondências
List<Integer> correspondencias = new ArrayList<>();
for (int valor : array) {
    if (valor > 25) {
        correspondencias.add(valor);
    }
}
```

4.1.3 Laços Controlados por Contagem vs Eventos

```
// Controlado por contagem (iteração definida) - número conhecido de iterações
for (int i = 0; i < 10; i++) {
    System.out.println("Iteração: " + i);
}

// Controlado por evento (iteração indefinida) - depende de condições
Scanner scanner = new Scanner(System.in);
String entrada;
do {
```

```
    System.out.print("Digite 'sair' para sair: ");
    entrada = scanner.nextLine();
    // Processar entrada
} while (!entrada.equalsIgnoreCase("sair"));

// Laço while para leitura até valor sentinela
int soma = 0;
int valor;
while ((valor = scanner.nextInt()) != -1) {
    soma += valor;
}
```

Section 4.2: Laços Aninhados

4.2.1 Tabela de Multiplicação

```
for (int i = 1; i <= 10; i++) {
    for (int j = 1; j <= 10; j++) {
        System.out.printf("%4d", i * j);
    }
    System.out.println();  // Nova linha após cada linha
}

// Saída:
//   1   2   3   4   5   6   7   8   9  10
//   2   4   6   8  10  12  14  16  18  20
//   ... etc
```

4.2.2 Geração de Padrões

```
// Padrão de triângulo retângulo
int n = 5;
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= i; j++) {
        System.out.print("* ");
    }
    System.out.println();
}

// Saída:
// *
// * *
```

```
// * * *
// * * * *
// * * * * *

// Padrão de pirâmide
for (int i = 1; i <= n; i++) {
    // Imprimir espaços
    for (int j = 1; j <= n - i; j++) {
        System.out.print("  ");
    }
    // Imprimir estrelas
    for (int j = 1; j <= 2 * i - 1; j++) {
        System.out.print("* ");
    }
    System.out.println();
}
```

Section 4.3: Otimização de Laços

4.3.1 Movimento de Código Invariante do Laço

Mover cálculos que não mudam dentro do laço para fora:

```
// ANTES: Cálculo dentro do laço
for (int i = 0; i < array.length; i++) {
    double resultado = Math.PI * raio * array[i];  // Math.PI * raio é invariante
}

// DEPOIS: Mover invariante para fora do laço
double constante = Math.PI * raio;
for (int i = 0; i < array.length; i++) {
    double resultado = constante * array[i];  // Mais rápido
}

// ANTES: Chamada de método na condição do laço
for (int i = 0; i < lista.size(); i++) {  // size() chamado a cada iteração
    // ...
}

// DEPOIS: Armazenar tamanho em cache
int tamanho = lista.size();
for (int i = 0; i < tamanho; i++) {
    // ...
}
```

4.3.2 Desenrolamento de Laço

Reduzir sobrecarga do laço processando múltiplos elementos por iteração:

```
// Laço padrão
int soma = 0;
for (int i = 0; i < 100; i++) {
    soma += array[i];
}

// Parcialmente desenrolado (processa 4 elementos por iteração)
int soma = 0;
for (int i = 0; i < 100; i += 4) {
    soma += array[i] + array[i+1] + array[i+2] + array[i+3];
}
// Lidar com elementos restantes (se houver)
for (int i = 96; i < 100; i++) {
    soma += array[i];
}
```

Section 4.4: Recursão vs Iteração

4.4.1 Comparação de Fatorial

```
// Fatorial iterativo (preferido em Java)
int fatorialIterativo(int n) {
    if (n < 0) throw new IllegalArgumentException();
    int resultado = 1;
    for (int i = 2; i <= n; i++) {
        resultado *= i;
    }
    return resultado;
}

// Fatorial recursivo (mais simples mas menos eficiente)
int fatorialRecursivo(int n) {
    if (n < 0) throw new IllegalArgumentException();
    if (n <= 1) return 1;
    return n * fatorialRecursivo(n - 1);
}

// Fatorial recursivo de cauda (Java não otimiza chamadas de cauda)
int fatorialCauda(int n, int acumulador) {
```

```
    if (n <= 1) return acumulador;
    return fatorialCauda(n - 1, n * acumulador);
}
// Método wrapper
int fatorial(int n) {
    return fatorialCauda(n, 1);
}
```

4.4.2 Quando Usar Recursão

- **Use recursão para**:
  - Travessias de árvores/grafos
  - Algoritmos dividir para conquistar
  - Problemas com definições matemáticas recursivas
- **Use iteração para**:
  - Processamento linear simples
  - Código crítico para desempenho
  - Recursão profunda que pode causar estouro de pilha

Questões de Revisão: Capítulo 4

1. Compare e contraste laços for, while e do-while. Quando cada um deve ser usado?
2. Escreva um programa que imprima os primeiros 20 números de Fibonacci usando três tipos diferentes de laço.
3. Implemente o bubble sort e analise sua complexidade de tempo em termos de notação Big O.
4. Escreva um programa que encontre todos os números primos entre 1 e 100 usando o algoritmo Sieve of Eratosthenes.
5. Explique técnicas de otimização de laços com exemplos concretos.
6. Compare abordagens iterativas e recursivas para calcular fatorial. Qual é mais eficiente e por quê?
7. Escreva um programa que inverta um array in-place (sem criar um novo array).
8. Implemente busca binária e compare sua eficiência com busca linear.

Chapter 5
Estruturas de Dados Básicas

Section 5.1: Arrays

5.1.1 Variações de Declaração de Array

```
// Arrays unidimensionais
int[] arr1 = new int[5];           // Todos zeros [0,0,0,0,0]
int[] arr2 = {1, 2, 3, 4, 5};     // Inicializado
int[] arr3 = new int[]{1, 2, 3};  // Sintaxe alternativa

// Arrays multidimensionais
int[][] matriz1 = new int[3][4];   // Matriz 3x4 (todos zeros)
int[][] matriz2 = {{1,2}, {3,4}}; // Matriz 2x2
int[][] irregular = new int[3][];  // Array irregular (jagged)
irregular[0] = new int[2];         // Primeira linha tem 2 colunas
irregular[1] = new int[3];         // Segunda linha tem 3 colunas
irregular[2] = new int[1];         // Terceira linha tem 1 coluna

// Array de objetos
String[] nomes = new String[3];
nomes[0] = "Alice";
nomes[1] = "Bob";
nomes[2] = "Charlie";
```

5.1.2 Algoritmos de Array

Bubble Sort

```
void bubbleSort(int[] arr) {
    int n = arr.length;
    boolean trocado;
```

```
    for (int i = 0; i < n - 1; i++) {
        trocado = false;
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                // Trocar arr[j] e arr[j+1]
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
                trocado = true;
            }
        }
        // Se não houve trocas, o array está ordenado
        if (!trocado) break;
    }
}
```

Busca Binária

```
int buscaBinaria(int[] arr, int alvo) {
    int esquerda = 0;
    int direita = arr.length - 1;

    while (esquerda <= direita) {
        int meio = esquerda + (direita - esquerda) / 2;  // Evitar overflow

        if (arr[meio] == alvo) {
            return meio;  // Encontrado
        } else if (arr[meio] < alvo) {
            esquerda = meio + 1;  // Buscar metade direita
        } else {
            direita = meio - 1;   // Buscar metade esquerda
        }
    }
    return -1;  // Não encontrado
}

// Versão recursiva
int buscaBinariaRecursiva(int[] arr, int alvo, int esquerda, int direita) {
    if (esquerda > direita) return -1;

    int meio = esquerda + (direita - esquerda) / 2;
    if (arr[meio] == alvo) return meio;
    if (arr[meio] < alvo)
        return buscaBinariaRecursiva(arr, alvo, meio + 1, direita);
    else
```

```
        return buscaBinariaRecursiva(arr, alvo, esquerda, meio - 1);
}
```

5.1.3 Utilitários de Array

```
import java.util.Arrays;
import java.util.Collections;

int[] arr = {5, 2, 8, 1, 9};

// Ordenação
Arrays.sort(arr);                   // Ascendente: {1, 2, 5, 8, 9}
Arrays.sort(arr, 1, 4);            // Ordenar subconjunto [1,4)

// Busca (array deve estar ordenado)
int indice = Arrays.binarySearch(arr, 5);  // 2

// Preenchimento
Arrays.fill(arr, 0);               // {0, 0, 0, 0, 0}
Arrays.fill(arr, 1, 3, 99);        // Preencher posições 1-2 com 99

// Cópia
int[] copia = Arrays.copyOf(arr, arr.length);
int[] intervalo = Arrays.copyOfRange(arr, 1, 4);  // Elementos 1-3

// Comparação
boolean igual = Arrays.equals(arr1, arr2);
int comparacao = Arrays.compare(arr1, arr2);  // Lexicográfica

// Streaming
int soma = Arrays.stream(arr).sum();
double media = Arrays.stream(arr).average().orElse(0);
```

Section 5.2: ArrayLists (Introdução)

5.2.1 Operações Básicas

```
import java.util.ArrayList;
import java.util.Collections;

ArrayList<Integer> lista = new ArrayList<>();
```

```
// Adicionando elementos
lista.add(10);                      // [10]
lista.add(20);                      // [10, 20]
lista.add(1, 15);                   // [10, 15, 20] - inserir no índice 1
lista.addAll(Arrays.asList(30, 40)); // [10, 15, 20, 30, 40]

// Acessando elementos
int primeiro = lista.get(0);          // 10
int tamanho = lista.size();           // 5
boolean contem = lista.contains(20); // true
int indice = lista.indexOf(20);      // 2
int ultimoIndice = lista.lastIndexOf(20); // 2

// Modificando elementos
lista.set(1, 25);                   // [10, 25, 20, 30, 40]
lista.replaceAll(x -> x * 2);       // [20, 50, 40, 60, 80]

// Removendo elementos
lista.remove(0);                    // Remover primeiro: [50, 40, 60, 80]
lista.remove(Integer.valueOf(40)); // Remover por valor: [50, 60, 80]
lista.removeIf(x -> x > 70);        // Remover se >70: [50, 60]
lista.clear();                      // Lista vazia

// Iterando
for (int num : lista) {
    System.out.println(num);
}

for (int i = 0; i < lista.size(); i++) {
    System.out.println(lista.get(i));
}

lista.forEach(System.out::println);  // Referência de método
```

5.2.2 ArrayList vs Array

Característica	**Array**	**ArrayList**
Tamanho	Fixo na criação	Dinâmico (cresce/encolhe automaticamente)
Desempenho	Acesso mais rápido (indexação direta)	Um pouco mais lento (verificação de limites)
Segurança de tipo	Em tempo de compilação (para primitivos)	Tempo de execução (generics, type erasure)
Memória	Menos sobrecarga	Mais sobrecarga (wrapper de objeto)
Métodos	Limitado (campo length)	API rica (add, remove, sort, etc.)
Primitivos	Suporte direto	Requer classes wrapper (Integer, etc.)
Multidimensional	Suportado	Listas de Listas

Section 5.3: Problemas Comuns de Estruturas de Dados

5.3.1 Problema Two Sum

Encontre dois números que somam a um valor alvo:

```
// Abordagem de força bruta: O(n²) tempo, O(1) espaço
int[] twoSumForcaBruta(int[] nums, int alvo) {
    for (int i = 0; i < nums.length; i++) {
        for (int j = i + 1; j < nums.length; j++) {
            if (nums[i] + nums[j] == alvo) {
                return new int[]{i, j};
            }
        }
    }
    return new int[]{-1, -1};
}

// Otimizado usando HashMap: O(n) tempo, O(n) espaço
int[] twoSumHashMap(int[] nums, int alvo) {
    Map<Integer, Integer> mapa = new HashMap<>();
    for (int i = 0; i < nums.length; i++) {
        int complemento = alvo - nums[i];
        if (mapa.containsKey(complemento)) {
            return new int[]{mapa.get(complemento), i};
        }
        mapa.put(nums[i], i);
    }
    return new int[]{-1, -1};
}
```

5.3.2 Subarray Máximo

Encontre o subarray contíguo com a maior soma (Algoritmo de Kadane):

```
// Algoritmo de Kadane: O(n) tempo, O(1) espaço
int maxSubArray(int[] nums) {
    if (nums.length == 0) return 0;

    int maxAteAgora = nums[0];
    int maxTerminandoAqui = nums[0];
```

```
    for (int i = 1; i < nums.length; i++) {
        // Ou estende o subarray anterior ou inicia novo
        maxTerminandoAqui = Math.max(nums[i], maxTerminandoAqui + nums[i]);
        maxAteAgora = Math.max(maxAteAgora, maxTerminandoAqui);
    }

    return maxAteAgora;
}

// Versão que também retorna os índices do subarray
int[] maxSubArrayComIndices(int[] nums) {
    int maxAteAgora = nums[0], maxTerminandoAqui = nums[0];
    int inicio = 0, fim = 0, inicioTemp = 0;

    for (int i = 1; i < nums.length; i++) {
        if (nums[i] > maxTerminandoAqui + nums[i]) {
            maxTerminandoAqui = nums[i];
            inicioTemp = i;
        } else {
            maxTerminandoAqui = maxTerminandoAqui + nums[i];
        }

        if (maxTerminandoAqui > maxAteAgora) {
            maxAteAgora = maxTerminandoAqui;
            inicio = inicioTemp;
            fim = i;
        }
    }

    return new int[]{maxAteAgora, inicio, fim};
}
```

Questões de Revisão: Capítulo 5

1. Explique a diferença entre arrays e ArrayLists. Quando você escolheria um em vez do outro?
2. Implemente um método que rotacione um array por k posições para a direita.
3. Escreva um programa que encontre o elemento mais frequente em um array.
4. Implemente o algoritmo de Kadane para encontrar a soma máxima de subarray.
5. Escreva um programa que mescle dois arrays ordenados em um único array ordenado.
6. Explique como arrays multidimensionais são armazenados na memória.

Chapter 6
Manipulação de Strings

Section 6.1: Fundamentos de String

6.1.1 Criação de String

```
// Diferentes formas de criar strings
String s1 = "Olá";                    // Literal de string (do string pool)
String s2 = new String("Olá");        // Usando construtor (novo objeto)
char[] chars = {'O','l','á'};
String s3 = new String(chars);         // De array de caracteres
String s4 = String.valueOf(123);       // De outros tipos
String s5 = String.format("Valor: %d", 42);  // String formatada

// Concatenação de strings
String concat1 = "Olá" + " " + "Mundo";  // "Olá Mundo"
String concat2 = "Resultado: " + 42;          // "Resultado: 42"

// Importante: Imutabilidade de String
String s = "Olá";
s = s + " Mundo";  // Cria nova string "Olá Mundo", não modifica "Olá"
```

6.1.2 String Pool

Java mantém um string pool para reutilizar objetos de string imutáveis:

```
String s1 = "Olá";        // Vai para o string pool
String s2 = "Olá";        // Reutiliza do pool (s1 == s2 é verdadeiro)
String s3 = new String("Olá");  // Novo objeto (s1 == s3 é falso)
String s4 = s3.intern();  // Adiciona ao pool ou retorna existente

// == compara referências, equals() compara conteúdo
System.out.println(s1 == s2);        // true (mesma referência)
```

```
System.out.println(s1.equals(s2));   // true (mesmo conteúdo)
System.out.println(s1 == s3);        // false (referências diferentes)
System.out.println(s1.equals(s3));   // true (mesmo conteúdo)
System.out.println(s1 == s4);        // true (s4 é interned)
```

Section 6.2: Métodos de String

6.2.1 Operações Básicas

```
String str = "  Olá Mundo  ";

// Comprimento e vazio
int len = str.length();           // 15 (inclui espaços)
boolean vazio = str.isEmpty();    // false
boolean emBranco = str.isBlank(); // false (Java 11+) - verifica se vazio ou whitespace

// Acesso a caracteres
char primeiro = str.charAt(2);       // 'O' (índice 2, após espaços)
char ultimo = str.charAt(str.length() - 1);  // ' ' (último espaço)

// Substrings
String sub1 = str.substring(2);       // "Olá Mundo  "
String sub2 = str.substring(2, 7);    // "Olá"
String sub3 = str.substring(8, 13);   // "Mundo"

// Conversão de caixa
String maiuscula = str.toUpperCase();    // "  OLÁ MUNDO  "
String minuscula = str.toLowerCase();    // "  olá mundo  "

// Trimming
String aparado = str.trim();         // "Olá Mundo" (remove espaços iniciais/finais)
String strip = str.strip();          // "Olá Mundo" (Java 11+, Unicode-aware)
String stripInicial = str.stripLeading();  // "Olá Mundo  "
String stripFinal = str.stripTrailing();   // "  Olá Mundo"
```

6.2.2 Busca e Comparação

```
String str = "Olá Mundo";

// Busca
int indice1 = str.indexOf('á');        // 2 (primeiro 'á')
int indice2 = str.indexOf('o', 5);     // 7 (começar do índice 5)
```

```
int indice3 = str.indexOf("Mundo");    // 4
int ultimoIndice = str.lastIndexOf('o'); // 7
int ultimoIndice2 = str.lastIndexOf('o', 6); // 4 (buscar para trás do índice 6)

// Verificando conteúdo
boolean comeca = str.startsWith("Olá");   // true
boolean termina = str.endsWith("Mundo");  // true
boolean contem = str.contains("lá M");    // true

// Comparação
String s1 = "Olá";
String s2 = "olá";
boolean eq1 = s1.equals(s2);           // false (sensível a caixa)
boolean eq2 = s1.equalsIgnoreCase(s2); // true (insensível a caixa)
int cmp = s1.compareTo(s2);            // -32 (negativo, 'O' < 'o')
int cmpIgnore = s1.compareToIgnoreCase(s2); // 0

// Correspondência de padrões
boolean matches = str.matches("^Olá.*");  // true (regex)
```

Section 6.3: Construção de String

6.3.1 StringBuilder vs StringBuffer

```
// StringBuilder (não thread-safe, mais rápido para thread única)
StringBuilder sb = new StringBuilder();
sb.append("Olá");
sb.append(" ");
sb.append("Mundo");
sb.insert(5, ",");           // "Olá, Mundo"
sb.replace(5, 6, " ");       // "Olá Mundo"
sb.delete(5, 6);             // "OláMundo"
sb.reverse();                // "odnuM álO"
String resultado1 = sb.toString();

// StringBuffer (thread-safe, sincronizado, mais lento)
StringBuffer buffer = new StringBuffer();
buffer.append("Olá");
buffer.append(" ");
buffer.append("Mundo");
String resultado2 = buffer.toString();

// Encadeamento de métodos
```

```
String str = new StringBuilder()
    .append("Olá")
    .append(" ")
    .append("Mundo")
    .reverse()          // "odnuM álO"
    .toString();
```

6.3.2 Considerações de Eficiência

```
// INEFICIENTE: Cria muitas strings intermediárias (O(n²) tempo)
String resultado = "";
for (int i = 0; i < 1000; i++) {
    resultado += i;  // Cria nova string a cada iteração
}

// EFICIENTE: Usa StringBuilder (O(n) tempo)
StringBuilder sb = new StringBuilder();
for (int i = 0; i < 1000; i++) {
    sb.append(i);
}
String resultado = sb.toString();

// Pré-dimensionar StringBuilder para operações grandes
StringBuilder sb2 = new StringBuilder(10000);  // Capacidade inicial
for (int i = 0; i < 10000; i++) {
    sb2.append(i);
}
```

Section 6.4: Formatação de String

6.4.1 Formatação estilo printf

```
String nome = "João";
int idade = 25;
double salario = 50000.50;

// String formatada
String formatado = String.format(
    "Nome: %s, Idade: %d, Salário: $%,.2f",
    nome, idade, salario
);
// Resultado: "Nome: João, Idade: 25, Salário: $50.000,50"
```

```
// Impressão direta
System.out.printf("Nome: %s, Idade: %d%n", nome, idade);

// Especificadores de formato comuns:
// %s - String
// %d - Inteiro decimal
// %f - Ponto flutuante (%.2f para 2 casas decimais)
// %c - Caractere
// %b - Booleano
// %n - Nova linha (independente de plataforma)
// %% - Sinal de percentual
// %t - Data/hora
// %, - Separador de milhares

// Preenchimento e alinhamento
System.out.printf("%-10s %5d%n", "João", 25);   // Nome alinhado à esquerda
System.out.printf("%10s %5d%n", "Maria", 30);    // Alinhado à direita
```

Section 6.5: Expressões Regulares

6.5.1 Padrões Básicos

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;

String texto = "O preço é $19,99 e $29,50. Contato: info@exemplo.com";

// Correspondência de padrões
Pattern padraoPreco = Pattern.compile("\\$\\d+,\\d{2}");
Matcher matcherPreco = padraoPreco.matcher(texto);

while (matcherPreco.find()) {
    System.out.println("Preço encontrado: " + matcherPreco.group());
}
// Saída: Preço encontrado: $19,99
//        Preço encontrado: $29,50

// Padrões de validação comuns
Pattern email = Pattern.compile("^[\\w.-]+@[\\w.-]+\\.[a-zA-Z]{2,}$");
Pattern telefone = Pattern.compile("^\\+?[\\d\\s-]{10,}$");
Pattern data = Pattern.compile("^\\d{4}-\\d{2}-\\d{2}$");
Pattern url = Pattern.compile("^(https?://)?([\\w-]+\\.)+[a-zA-Z]{2,}(/\\S*)?$");
```

```
// Usando métodos String com regex
String[] palavras = texto.split("\\s+");  // Dividir por whitespace
String limpo = texto.replaceAll("\\$", "");  // Remover sinais de dólar
boolean temDigitos = texto.matches(".*\\d+.*");  // Verificar se contém dígitos
```

Section 6.6: Aplicações Práticas

6.6.1 Verificação de Palíndromo

```
public boolean ehPalindromo(String str) {
    // Remover não alfanuméricos e converter para minúsculas
    String limpo = str.replaceAll("[^a-zA-Z0-9]", "").toLowerCase();

    // Abordagem de dois ponteiros
    int esquerda = 0;
    int direita = limpo.length() - 1;

    while (esquerda < direita) {
        if (limpo.charAt(esquerda) != limpo.charAt(direita)) {
            return false;
        }
        esquerda++;
        direita--;
    }
    return true;
}

// Alternativa usando StringBuilder
public boolean ehPalindromo2(String str) {
    String limpo = str.replaceAll("[^a-zA-Z0-9]", "").toLowerCase();
    String invertido = new StringBuilder(limpo).reverse().toString();
    return limpo.equals(invertido);
}

// Casos de teste
ehPalindromo("A man, a plan, a canal: Panama");  // true
ehPalindromo("race a car");                       // false
ehPalindromo(" ");                                // true (vazio ou whitespace)
```

6.6.2 Contagem de Palavras

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

public Map<String, Integer> contagemPalavras(String texto) {
    Map<String, Integer> mapaContagem = new HashMap<>();

    if (texto == null || texto.isBlank()) {
        return mapaContagem;
    }

    // Dividir em palavras (lidar com pontuação e múltiplos espaços)
    String[] palavras = texto.toLowerCase()
                        .replaceAll("[^a-z\\s]", " ")  // Substituir não letras por espac
                        .trim()                         // Remover espaços iniciais/fina
                        .split("\\s+");                 // Dividir por um ou mais espaços

    // Contar ocorrências
    for (String palavra : palavras) {
        if (!palavra.isEmpty()) {
            mapaContagem.put(palavra, mapaContagem.getOrDefault(palavra, 0) + 1);
        }
    }

    return mapaContagem;
}

// Versão aprimorada com filtro de stop words
public Map<String, Integer> contagemPalavrasComFiltro(String texto, Set<String> stopWords)
    Map<String, Integer> mapaContagem = new HashMap<>();

    String[] palavras = texto.toLowerCase()
                        .replaceAll("[^a-z\\s]", " ")
                        .trim()
                        .split("\\s+");

    for (String palavra : palavras) {
        if (!stopWords.contains(palavra) && !palavra.isEmpty()) {
            mapaContagem.put(palavra, mapaContagem.getOrDefault(palavra, 0) + 1);
        }
    }

    return mapaContagem;
}
```

Questões de Revisão: Capítulo 6

1. Explique a imutabilidade de strings e suas implicações para desempenho. Como StringBuilder pode ajudar?
2. Escreva um programa que verifique se duas strings são anagramas uma da outra.
3. Implemente um método que comprima uma string (ex: ”aaabbbccc” → ”a3b3c3”) mas retorne a original se a compressão não ajudar.
4. Escreva um programa que valide e analise um endereço de email usando expressões regulares.
5. Implemente um mecanismo de template simples que substitua placeholders como {{nome}} com valores de um mapa.
6. Escreva um programa que conte a frequência de palavras em um documento de texto.
7. Explique a diferença entre String, StringBuilder e StringBuffer.
8. Escreva um programa que formate um número de telefone de vários formatos de entrada para um formato padrão.

Chapter 7
Respostas das Questões de Revisão

Section 7.1: Respostas do Capítulo 1

1. **Compilação vs Interpretação**: Linguagens compiladas são traduzidas para código de máquina antes da execução (mais rápido, específico da plataforma). Linguagens interpretadas são executadas linha por linha (mais lento, independente de plataforma). Java usa ambas: compilado para bytecode, então interpretado pela JVM.
2. **Processo Java**: Fonte (.java) → Compilador (javac) → Bytecode (.class) → JVM → Execução. Híbrido porque o bytecode é compilado mas precisa de interpretação da JVM.
3. **Áreas de Memória**:
   - Pilha: Variáveis locais, chamadas de métodos
   - Heap: Objetos, alocação dinâmica
   - Estática: Variáveis de classe, constantes
   - Código: Instruções do programa
4. **Sintaxe, Semântica, Pragmática**:
   - Sintaxe: `if (x > 0) {}` - estrutura correta
   - Semântica: `x > 0` significa "x é positivo"
   - Pragmática: Usar nomes de variáveis significativos, formatação adequada
5. **Propósito da JVM**: Executa bytecode, fornece gerenciamento de memória, segurança, abstração de plataforma, coleta de lixo.
6. **Bytecode**: Representação intermediária entre fonte e código de máquina, independente de plataforma, executado pela JVM.
7. **Componentes do Programa Java**:
   - `class`: Define um modelo para objetos
   - `main`: Ponto de entrada do programa

- `public`: Modificador de acesso (JVM pode chamá-lo)
- `static`: Método de nível de classe (nenhum objeto necessário)
- `void`: Nenhum valor de retorno
- `String[] args`: Array de argumentos da linha de comando

Section 7.2: Respostas do Capítulo 2

1. **Escopos de Variáveis**:
   - Local: Dentro de método/bloco, acessível apenas dentro
   - Instância: Campos não estáticos, cada objeto tem sua própria cópia
   - Classe: Campos estáticos, compartilhado entre todos os objetos
   - Use instância para estado do objeto, classe para dados compartilhados, local para valores temporários
2. **Primitivos vs Referência**:
   - Primitivos: Armazenam valores diretamente (pilha), 8 tipos, mais rápido
   - Referências: Armazenam endereços de memória (heap), apontam para objetos, podem ser null
   - Implicações: Primitivos usam menos memória, referências permitem estruturas complexas
3. **Avaliação de Expressão**: `5 + 3 * 2 / (4 - 2)` = `5 + 3 * 2 / 2` = `5 + 6 / 2` = `5 + 3` = `8`
4. **Atribuição Composta**:

```
int x = 10;
x += 5;   // x = 15
x -= 3;   // x = 12
x *= 2;   // x = 24
x /= 4;   // x = 6
x %= 5;   // x = 1
x <<= 2;  // x = 4 (deslocamento bitwise à esquerda)
x >>= 1;  // x = 2 (deslocamento bitwise à direita)
```

5. **Casting de Tipos**:
   - Implícito: Conversões de alargamento (int para double, byte para int)
   - Explícito: Conversões de estreitamento (double para int, long para short)
   - Necessário quando possível perda de informação ou tipos incompatíveis

6. **Conversões de Tipo**:

```
// int para double
int i = 42;
double d = i;           // Implícita

// double para int
double preco = 19.99;
int precoInt = (int) preco;  // Explícita, trunca para 19

// char para int
char c = 'A';
int ascii = c;               // 65

// boolean para String
boolean flag = true;
String s = String.valueOf(flag);  // "true"
```

Section 7.3: Respostas do Capítulo 3

1. **Simplificação Booleana**: `!(a && b) || (!a && !b) || (a && !b)` Usando De Morgan: `(!a || !b) || (!a && !b) || (a && !b)` Distributiva: `!a || !b || (!a && !b) || (a && !b)` Simplifica para: `!a || !b`

2. **Encontrar Máximo**:

```
int max = a;
if (b > max) max = b;
if (c > max) max = c;
return max;

// Alternativa usando ternário
int max = (a > b) ? (a > c ? a : c) : (b > c ? b : c);
```

3. **Tipos de Igualdade**:
   - Igualdade superficial: Compara referências (==)
   - Igualdade profunda: Compara conteúdo (método equals())
   - Strings: Use equals() para comparação de conteúdo, == para comparação de referência

4. **Validação de Data**:

```
boolean dataValida(int dia, int mes, int ano) {
    if (ano < 1 || mes < 1 || mes > 12) return false;

    int[] diasNoMes = {31,28,31,30,31,30,31,31,30,31,30,31};

    // Verificação de ano bissexto
    boolean ehBissexto = (ano % 400 == 0) ||
                         (ano % 4 == 0 && ano % 100 != 0);
    if (ehBissexto) diasNoMes[1] = 29;

    return dia >= 1 && dia <= diasNoMes[mes-1];
}
```

Section 7.4: Respostas do Capítulo 4

1. **Comparação de Laços**:
    - for: Iterações conhecidas, baseado em contador
    - while: Baseado em condição, pode executar zero vezes
    - do-while: Baseado em condição, executa pelo menos uma vez
2. **Fibonacci**:

```
// Usando laço for
int a=0, b=1;
System.out.print(a + " " + b + " ");
for(int i=2; i<20; i++) {
    int c = a + b;
    System.out.print(c + " ");
    a = b;
    b = c;
}

// Usando while
int a=0, b=1, contador=2;
System.out.print(a + " " + b + " ");
while(contador < 20) {
    int c = a + b;
    System.out.print(c + " ");
    a = b;
    b = c;
    contador++;
}
```

3. **Análise do Bubble Sort**:
    - Complexidade de tempo: $O(n^2)$ pior e caso médio, O(n) melhor caso (já ordenado)
    - Complexidade de espaço: O(1) in-place
    - Estável: Elementos iguais mantêm ordem relativa
4. **Sieve of Eratosthenes**:

```
boolean[] ehPrimo = new boolean[101];
Arrays.fill(ehPrimo, true);
ehPrimo[0] = ehPrimo[1] = false;

for(int i=2; i*i<=100; i++) {
    if(ehPrimo[i]) {
        for(int j=i*i; j<=100; j+=i) {
            ehPrimo[j] = false;
        }
    }
}

for(int i=2; i<=100; i++) {
    if(ehPrimo[i]) System.out.print(i + " ");
}
```

5. **Otimização de Laços**:
    - Movimento de código invariante do laço: Mover cálculos para fora do laço
    - Desenrolamento de laço: Processar múltiplos elementos por iteração
    - Redução de força: Substituir operações caras por mais baratas
    - Fusão de laço: Combinar laços adjacentes
6. **Comparação de Fatorial**:
    - Iterativo: O(n) tempo, O(1) espaço, mais eficiente
    - Recursivo: O(n) tempo, O(n) espaço (pilha de chamadas), código mais simples
    - Iterativo preferido para desempenho e prevenção de estouro de pilha
7. **Inversão de Array**:

```
void inverterArray(int[] arr) {
    int esquerda = 0, direita = arr.length-1;
    while(esquerda < direita) {
        int temp = arr[esquerda];
```

```
            arr[esquerda] = arr[direita];
            arr[direita] = temp;
            esquerda++;
            direita--;
        }
    }
```

8. **Busca Binária vs Linear**:
    - Busca linear: O(n) tempo, funciona em arrays não ordenados
    - Busca binária: O(log n) tempo, requer array ordenado
    - Busca binária 100x mais rápida para 1.000.000 elementos

Section 7.5: Respostas do Capítulo 5

1. **Arrays vs ArrayLists**:
    - Arrays: Tamanho fixo, melhor desempenho, suporte direto a primitivos
    - ArrayLists: Tamanho dinâmico, API rica, requer wrappers de objeto para primitivos
    - Escolha arrays para código crítico de desempenho, ArrayLists quando o tamanho muda frequentemente
2. **Rotação de Array**:

```
void rotacionarDireita(int[] arr, int k) {
    k = k % arr.length;
    inverter(arr, 0, arr.length-1);
    inverter(arr, 0, k-1);
    inverter(arr, k, arr.length-1);
}

void inverter(int[] arr, int inicio, int fim) {
    while(inicio < fim) {
        int temp = arr[inicio];
        arr[inicio] = arr[fim];
        arr[fim] = temp;
        inicio++;
        fim--;
    }
}
```

3. **Elemento Mais Frequente**:

```
int maisFrequente(int[] arr) {
    Map<Integer, Integer> freq = new HashMap<>();
    int maxFreq = 0, maisFreq = arr[0];

    for(int num : arr) {
        freq.put(num, freq.getOrDefault(num, 0) + 1);
        if(freq.get(num) > maxFreq) {
            maxFreq = freq.get(num);
            maisFreq = num;
        }
    }
    return maisFreq;
}
```

4. **Algoritmo de Kadane**:

```
int maxSubArray(int[] nums) {
    int maxAteAgora = nums[0];
    int maxTerminandoAqui = nums[0];

    for(int i=1; i<nums.length; i++) {
        maxTerminandoAqui = Math.max(nums[i], maxTerminandoAqui + nums[i]);
        maxAteAgora = Math.max(maxAteAgora, maxTerminandoAqui);
    }
    return maxAteAgora;
}
```

5. **Mesclar Arrays Ordenados**:

```
int[] mesclar(int[] a, int[] b) {
    int[] resultado = new int[a.length + b.length];
    int i=0, j=0, k=0;

    while(i < a.length && j < b.length) {
        resultado[k++] = (a[i] <= b[j]) ? a[i++] : b[j++];
    }

    while(i < a.length) resultado[k++] = a[i++];
    while(j < b.length) resultado[k++] = b[j++];

    return resultado;
}
```

6. **Armazenamento Multidimensional**:
    - Java usa ordem row-major: elementos de cada linha armazenados contiguamente
    - Array 2D é array de arrays: cada linha pode ter comprimento diferente (jagged)
    - Cálculo de endereço de memória: base + (linha * cols + col) * tamanhoElemento

Section 7.6: Respostas do Capítulo 6

1. **Imutabilidade de String**:
    - Strings não podem ser modificadas após criação
    - Operações criam novas strings, levando a problemas de desempenho
    - StringBuilder fornece alternativa mutável para código com muita concatenação
2. **Verificação de Anagrama**:

```
boolean ehAnagrama(String s1, String s2) {
    if(s1.length() != s2.length()) return false;

    char[] c1 = s1.toLowerCase().toCharArray();
    char[] c2 = s2.toLowerCase().toCharArray();

    Arrays.sort(c1);
    Arrays.sort(c2);

    return Arrays.equals(c1, c2);
}
```

3. **Compressão de String**:

```
String comprimir(String str) {
    if(str.length() <= 2) return str;

    StringBuilder comprimido = new StringBuilder();
    int contador = 1;

    for(int i=1; i<str.length(); i++) {
        if(str.charAt(i) == str.charAt(i-1)) {
            contador++;
        } else {
            comprimido.append(str.charAt(i-1)).append(contador);
            contador = 1;
```

```
            }
        }
        comprimido.append(str.charAt(str.length()-1)).append(contador);

        return comprimido.length() < str.length() ?
               comprimido.toString() : str;
    }
```

4. **Validação de Email**:

```
    boolean emailValido(String email) {
        return email != null &&
               email.matches("^[\\w.-]+@[\\w.-]+\\.[a-zA-Z]{2,}$");
    }
```

5. **Mecanismo de Template**:

```
    String renderizarTemplate(String template, Map<String, String> dados) {
        String resultado = template;
        for(Map.Entry<String, String> entrada : dados.entrySet()) {
            String placeholder = "{{" + entrada.getKey() + "}}";
            resultado = resultado.replace(placeholder, entrada.getValue());
        }
        return resultado;
    }
```

6. **Frequência de Palavras**:

```
    Map<String, Integer> frequenciaPalavras(String texto) {
        Map<String, Integer> freq = new HashMap<>();
        String[] palavras = texto.toLowerCase()
                            .replaceAll("[^a-z\\s]", " ")
                            .trim()
                            .split("\\s+");

        for(String palavra : palavras) {
            if(!palavra.isEmpty()) {
                freq.put(palavra, freq.getOrDefault(palavra, 0) + 1);
            }
        }
        return freq;
    }
```

7. **String, StringBuilder, StringBuffer**:
    - String: Imutável, thread-safe, armazenado no string pool
    - StringBuilder: Mutável, não thread-safe, mais rápido para thread única
    - StringBuffer: Mutável, thread-safe (sincronizado), um pouco mais lento
8. **Formatador de Telefone**:

```
String formatarTelefone(String telefone) {
    String digitos = telefone.replaceAll("\\D", "");

    if(digitos.length() == 10) {
        return String.format("(%s) %s-%s",
            digitos.substring(0, 3),
            digitos.substring(3, 6),
            digitos.substring(6));
    } else if(digitos.length() == 11 && digitos.startsWith("1")) {
        return String.format("+1 (%s) %s-%s",
            digitos.substring(1, 4),
            digitos.substring(4, 7),
            digitos.substring(7));
    }
    return telefone; // Retornar original se formato não reconhecido
}
```