



```
1 public class Main {  
2     public static void main(String[] args) {  
3  
4         int idade = 20;  
5  
6         if (idade >= 18) {  
7             System.out.println("Acesso permitido!");  
8         } else {  
9             System.out.println("Acesso negado!");  
10        }  
11    }  
12  
13    for (int i = 1; i <= 5; i++) {  
14        System.out.println("Contagem: " + i);  
15    }  
16 }
```

UMA ABORDAGEM OBJETIVA

INTRODUÇÃO A PROGRAMAÇÃO JAVA

GISELDO NEO

ALANA NEO

Índice

1	Introdução à Programação em Java	1
1.1	Para quem é este livro	1
1.2	Ambiente adotado	2
1.3	Como estudar	2
1.4	Organização	3
I	Parte I — Fundamentos aplicados ao chatbot	5
2	Primeiros passos com Java	7
2.1	Objetivos de aprendizagem	7
2.2	Java e sua plataforma	7
2.3	Primeiro programa	8
2.4	Convenções iniciais	9
2.5	Como interpretar erros	10
2.6	Evolução do projeto: versão 1	10
2.7	Prática verificável	11
3	Variáveis, tipos, texto e entrada de dados	13
3.1	Objetivos de aprendizagem	13
3.2	Variáveis e tipos	13
3.3	Precisão e conversões numéricas	14
3.4	Trabalhando com texto	15
3.5	Entrada com Scanner	16

3.6	Evolução do projeto: versão 2	17
3.7	Prática verificável	17
4	Condicionais	19
4.1	Objetivos de aprendizagem	19
4.2	Expressões booleanas	19
4.3	Decisões com if	20
4.4	Opções fechadas com switch	21
4.5	Erros frequentes	21
4.6	Evolução do projeto: versão 3	22
4.7	Prática verificável	22
5	Estruturas de repetição	25
5.1	Objetivos de aprendizagem	25
5.2	Escolhendo o laço	25
5.3	Chatbot em repetição	26
5.4	Contadores e acumuladores	27
5.5	Laços aninhados	28
5.6	Evolução do projeto: versão 4	28
5.7	Prática verificável	29
6	Métodos	31
6.1	Objetivos de aprendizagem	31
6.2	Contratos pequenos e claros	31
6.3	Refatorando o chatbot	32
6.4	Escopo e passagem por valor	33
6.5	Sobrecarga	34
6.6	Validação e exceções	35
6.7	Evolução do projeto: versão 5	35
6.8	Prática verificável	35
7	Arrays	37
7.1	Objetivos de aprendizagem	37

7.2	Tamanho fixo e índices	37
7.3	Inicialização e valores padrão	38
7.4	Percorrendo com segurança	39
7.5	Utilitários e cópia	40
7.6	Evolução do projeto: versão 6	40
7.7	Prática verificável	40
8	Listas e coleções	43
8.1	Objetivos de aprendizagem	43
8.2	List e ArrayList	43
8.3	Operações essenciais	44
8.4	Exemplo completo	45
8.5	Cópias e exposição de estado	45
8.6	Autoboxing	46
8.7	Evolução do projeto: versão 7	46
8.8	Prática verificável	46
9	Classes e objetos	49
9.1	Objetivos de aprendizagem	49
9.2	Estado e comportamento	49
9.3	Construtores e this	51
9.4	Classe do chatbot	52
9.5	Composição inicial	53
9.6	Evolução do projeto: versão 8	53
9.7	Prática verificável	54
10	Tratamento de erros e arquivos	55
10.1	Objetivos de aprendizagem	55
10.2	Exceções esperadas	56
10.3	Repetir até obter um valor válido	56
10.4	Lançar uma exceção	57
10.5	Persistência com try-with-resources	58
10.6	Exemplo completo	59

10.7	Evolução do projeto: versão 9	60
10.8	Prática verificável	60
11	Chatbot de console	61
11.1	Objetivos de aprendizagem	61
11.2	Contrato de comandos	61
11.3	Implementação integrada	62
11.4	Separação de responsabilidades	65
11.5	Evolução do projeto: versão 10	65
11.6	Roteiro de teste manual	65
11.7	Prática verificável	66
II	Parte II — Orientação a objetos e evolução do projeto	67
12	Design orientado a objetos	69
12.1	Objetivos de aprendizagem	69
12.2	Da classe grande às responsabilidades	69
12.3	Encapsulamento e acesso	70
12.4	Composição	71
12.5	static e final	72
12.6	Revisão de design	73
12.7	Prática verificável	73
13	Herança	75
13.1	Objetivos de aprendizagem	75
13.2	Relação é-um	75
13.3	Construtores e super	77
13.4	Sobrescrita	78
13.5	Classes abstratas e final	78
13.6	Quando preferir composição	78
13.7	Prática verificável	79

14 Interfaces e polimorfismo	81
14.1 Objetivos de aprendizagem	81
14.2 Contrato de comando	81
14.3 Polimorfismo em execução	83
14.4 Interface ou classe abstrata?	85
14.5 Evolução do projeto: versão 13	85
14.6 Prática verificável	85
15 Generics e coleções	87
15.1 Objetivos de aprendizagem	87
15.2 Escolha da coleção	87
15.3 Registro de comandos com Map	88
15.4 Classe genérica	89
15.5 Exemplo completo	90
15.6 Prática verificável	91
16 Testes e refatoração	93
16.1 Objetivos de aprendizagem	93
16.2 Teste como especificação	93
16.3 Casos de borda	95
16.4 Refatoração	95
16.5 Roteiro de regressão do chatbot	96
16.6 Evolução do projeto: versão 15	97
16.7 Prática verificável	97
17 Projeto final: TutorJavaBot	99
17.1 Objetivo	99
17.2 Requisitos obrigatórios	99
17.3 Arquitetura recomendada	100
17.4 Contrato dos comandos	101
17.5 Roteiro de implementação	101
17.5.1 Etapa 1 — domínio mínimo	101
17.5.2 Etapa 2 — comandos	102

17.5.3	Etapa 3 — coordenação	102
17.5.4	Etapa 4 — persistência	102
17.5.5	Etapa 5 — validação	102
17.6	Roteiro de aceitação	103
17.7	Crerios de avaliaão	103
17.8	Exemplo de referncia	103
17.9	Evoluões opcionais	104
III	Apêndices	105
18	Respostas e pistas	107
18.1	Primeiros passos	108
18.2	Variáveis, tipos e entrada	108
18.3	Condicionais	108
18.4	Repetição	109
18.5	Métodos	109
18.6	Arrays	109
18.7	Listas	109
18.8	Classes e objetos	110
18.9	Exceções e arquivos	110
18.10	Chatbot integrado	110
18.11	Design orientado a objetos	110
18.12	Herança	111
18.13	Interfaces e polimorfismo	111
18.14	Generics e coleções	111
18.15	Testes e refatoraão	111
18.16	Projeto final	112
19	Sobre os autores	113
	Giseldo da Silva Neo	113
	Alana Viana Borges da Silva Neo	114
	Contato	114

Aviso legal	114
Uso de inteligência artificial generativa	115

Capítulo 1

Introdução à Programação em Java

Aprendendo com chatbots

Bem-vindos!

[Baixar epub](#)

[Baixar PDF](#)

Este livro ensina programação em Java por meio da construção progressiva de um chatbot de console. Cada capítulo apresenta um conceito, aplica-o em um pequeno incremento do projeto e encerra com uma prática verificável.

1.1 Para quem é este livro

O conteúdo foi planejado para estudantes que estão iniciando em programação. Não é necessário conhecer Java, orientação a objetos ou ferra-

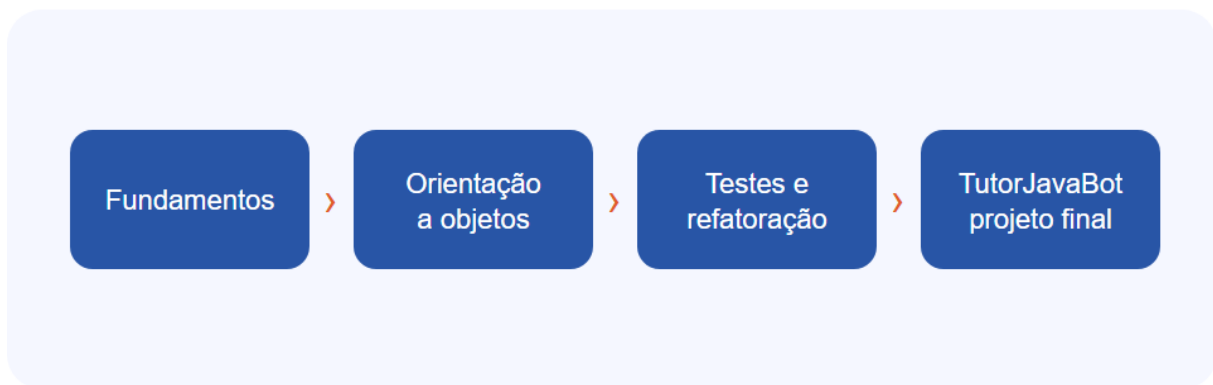


Figura 1.1: Percurso de aprendizagem: dos fundamentos ao TutorJavaBot.

mentas de desenvolvimento. É importante apenas saber criar arquivos e usar comandos básicos no terminal.

1.2 Ambiente adotado

Edição 0.2 — revisão de 29 de agosto de 2026.

Os exemplos têm como referência **Java 21 ou superior** e utilizam apenas recursos da biblioteca padrão. O código completo e compilável está disponível na pasta **examples/src**, enquanto os capítulos usam trechos menores para destacar cada conceito.

1.3 Como estudar

Em cada capítulo:

1. leia os objetivos e tente prever o comportamento dos exemplos;
2. execute o exemplo completo antes de modificá-lo;
3. implemente o incremento proposto para o chatbot;
4. valide o resultado com os casos de teste indicados;
5. consulte as respostas somente depois de tentar resolver a atividade.

O projeto começa com mensagens fixas, ganha entrada de dados, decisões, repetição, métodos e coleções, e depois é reorganizado com classes, encapsulamento, composição, herança, interfaces e polimorfismo.

1.4 Organização

- **Parte I — Fundamentos aplicados ao chatbot:** sintaxe, tipos, entrada, decisões, repetição, métodos, arrays, coleções, classes e tratamento de erros.
- **Parte II — Orientação a objetos e evolução do projeto:** design, encapsulamento, composição, herança, interfaces, generics, testes e refatoração.
- **Apêndices:** respostas comentadas, informações sobre os autores e aviso legal.

Ao terminar, você terá construído o TutorJavaBot, um programa de console com comandos, histórico, persistência em arquivo e uma estrutura preparada para receber novas funcionalidades.

Bom estudo!

Giseldo Neo e Alana Neo

Parte I

Parte I — Fundamentos aplicados ao chatbot

Capítulo 2

Primeiros passos com Java

Do código-fonte ao primeiro incremento do chatbot

2.1 Objetivos de aprendizagem

Ao concluir este capítulo, você deverá ser capaz de:

- explicar o papel do JDK, do compilador e da JVM;
- criar, compilar e executar uma classe Java;
- distinguir erros de compilação, execução e lógica;
- seguir as convenções básicas de nomes e arquivos.

2.2 Java e sua plataforma

Java combina uma linguagem de programação com uma plataforma de execução. O código escrito em um arquivo `.java` é transformado pelo compilador `javac` em bytecode, armazenado em um arquivo `.class`. A JVM executa esse bytecode no sistema operacional.

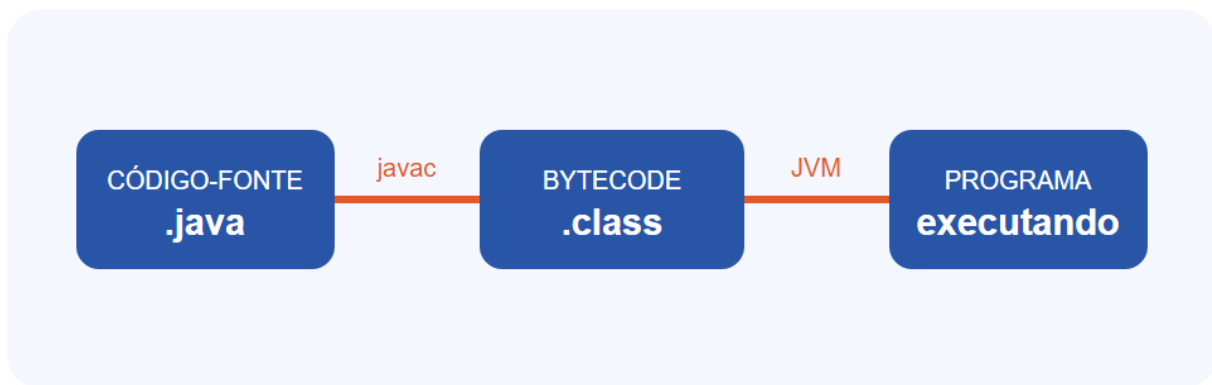


Figura 2.1: Do código-fonte à execução de um programa Java.

O **JDK** reúne o compilador, a JVM e as demais ferramentas necessárias para desenvolver e executar programas. Em instalações modernas, basta instalar um JDK compatível; não é necessário procurar separadamente um “JRE para desenvolvimento”.

```
~{mermaid} flowchart LR A["Código-fonte (.java)"] -->|javac| B["Bytecode (.class)"] B -->|java / JVM| C["Programa em execução"]~
```

Este livro adota Java 21 ou superior. Confirme o ambiente no terminal:

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

2.3 Primeiro programa

Crie um arquivo chamado `Capitulo01PrimeiroPrograma.java`:

```
public class Capitulo01PrimeiroPrograma {
    public static void main(String[] args) {
        System.out.println("TutorJavaBot: Olá!");
        System.out.println("TutorJavaBot: Vamos estudar Java?")
    }
}
```

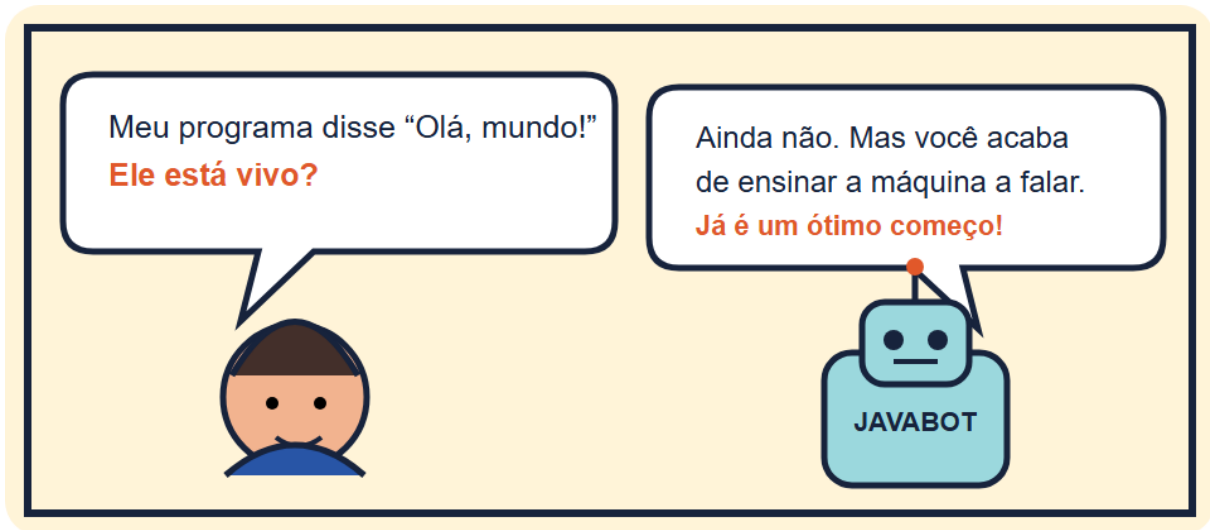


Figura 2.2: Quadrinho: o primeiro diálogo com Java.

A classe pública e o arquivo precisam ter o mesmo nome, inclusive no uso de letras maiúsculas e minúsculas.

Compile e execute:

```
javac Capitulo01PrimeiroPrograma.java  
java Capitulo01PrimeiroPrograma
```

Saída esperada:

```
TutorJavaBot: Olá!  
TutorJavaBot: Vamos estudar Java?
```

O método `main` é o ponto de entrada:

- `public`: pode ser chamado pela JVM;
- `static`: não exige a criação prévia de um objeto;
- `void`: não devolve um valor;
- `String[] args`: recebe argumentos da linha de comando.

2.4 Convenções iniciais

Em Java:

- classes usam `PascalCase`, como `TutorJavaBot`;
- variáveis e métodos usam `camelCase`, como `nomeUsuario`;
- constantes usam letras maiúsculas, como `LIMITE_TENTATIVAS`;
- instruções normalmente terminam com ponto e vírgula;
- chaves delimitam blocos de código.

Comentários devem explicar intenção ou contexto, não repetir uma instrução evidente:

```
// Esta mensagem identifica o personagem que acompanhará o livro
System.out.println("TutorJavaBot: Olá!");
```

2.5 Como interpretar erros

Um **erro de compilação** impede a criação do arquivo `.class`, por exemplo uma chave ausente. Uma **exceção em tempo de execução** acontece depois que o programa começa, como uma conversão numérica inválida. Um **erro de lógica** não interrompe o programa, mas produz resultado incorreto.

Leia sempre a primeira mensagem do compilador, identifique arquivo e linha e corrija uma causa por vez.

2.6 Evolução do projeto: versão 1

O chatbot ainda não recebe dados. Neste primeiro checkpoint, ele apenas:

1. apresenta seu nome;
2. explica o objetivo da conversa;
3. encerra normalmente.

O exemplo completo está em [examples/src/Capitulo01PrimeiroPrograma.j](#)

2.7 Prática verificável

1. Acrescente uma terceira mensagem com os comandos que serão implementados no livro.
2. Renomeie a classe e o arquivo de forma consistente.
3. Remova um ponto e vírgula, observe o erro e depois restaure o código.

Critérios de conclusão:

- o comando **javac** termina sem mensagens;
- a classe executada tem o mesmo nome do arquivo;
- as três mensagens aparecem na ordem prevista.

i Antes de avançar

Você deve conseguir explicar, sem consultar o texto, a sequência arquivo **.java** @ compilador @ arquivo **.class** @ JVM.

Capítulo 3

Variáveis, tipos, texto e entrada de dados

Representar informações antes de tomar decisões

3.1 Objetivos de aprendizagem

Ao concluir este capítulo, você deverá ser capaz de:

- declarar, inicializar e atualizar variáveis;
- escolher tipos coerentes com o problema;
- manipular e comparar textos corretamente;
- ler entradas com `Scanner` e converter texto em número;
- reconhecer conversões que podem perder informação.

3.2 Variáveis e tipos

Uma variável associa um nome a um valor de determinado tipo:

NOME	TIPO	VALOR
idade	int	19

Figura 3.1: Uma variável relaciona nome, tipo e valor.

```
String nomeUsuario = "Ana";  
int idade = 19;  
double media = 8.5;  
boolean estudanteAtivo = true;
```

Em Java, **declaração**, **inicialização** e **atribuição** são momentos diferentes:

```
int totalMensagens;      // declaração  
totalMensagens = 1;      // inicialização  
totalMensagens = 2;      // nova atribuição
```

Os oito tipos primitivos são `byte`, `short`, `int`, `long`, `float`, `double`, `char` e `boolean`. Para começar, `int`, `double`, `char` e `boolean` resolvem a maioria dos exercícios. `String` não é um tipo primitivo: é uma classe usada para representar texto.

Prefira nomes que expressem intenção, como `quantidadeTentativas` e `comandoValido`.

3.3 Precisão e conversões numéricas

Use tipos inteiros para contagens e tipos decimais para medidas e médias. Valores monetários reais exigem cuidados adicionais; em sistemas

financeiros, normalmente se usa `BigDecimal`, não `double`.

A divisão entre dois inteiros produz um resultado inteiro:

```
int acertos = 7;
int questoes = 10;

double taxaIncorreta = acertos / questoes;           // 0.0
double taxaCorreta = (double) acertos / questoes;    // 0.7
```

Converter de um tipo de maior capacidade para outro menor pode descartar informação:

```
double media = 8.75;
int parteInteira = (int) media; // resultado: 8
```

Não faça um casting apenas para “o código compilar”. Antes, verifique se a perda de precisão é aceitável.

3.4 Trabalhando com texto

Strings são imutáveis: métodos como `strip` e `toLowerCase` devolvem um novo texto.

```
String entrada = "  AJUDA  ";
String comando = entrada.strip().toLowerCase();

System.out.println(comando); // ajuda
```

Para comparar conteúdo, use `equals` ou `equalsIgnoreCase`. O operador `==` compara referências e não deve ser usado para decidir se dois textos têm o mesmo conteúdo.

```
if (comando.equals("ajuda")) {
    System.out.println("Comandos: ajuda, dica e sair.");
}
```

```
}
```

Em Java moderno, `isBlank()` verifica se um texto está vazio ou contém apenas espaços.

3.5 Entrada com Scanner

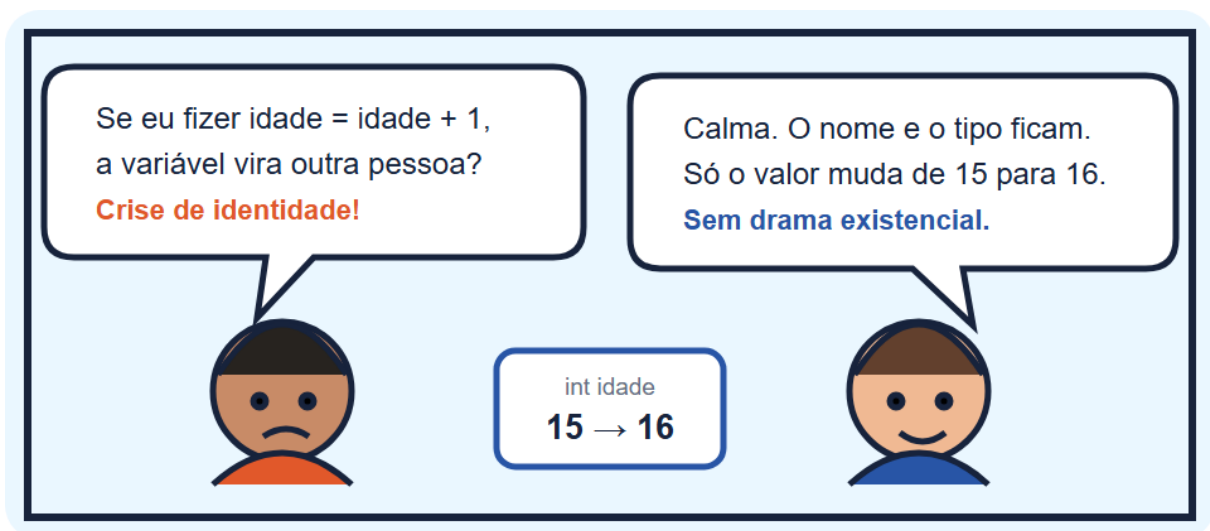


Figura 3.2: Quadrinho: variáveis não são caixas mágicas.

Uma estratégia previsível consiste em ler toda entrada com `nextLine()` e converter números quando necessário. Isso evita a quebra de linha pendente causada pela combinação de `nextInt()` e `nextLine()`.

```
import java.util.Scanner;

public class Capitulo02Entrada {
    public static void main(String[] args) {
        try (Scanner scanner = new Scanner(System.in)) {
            System.out.print("Seu nome: ");
            String nome = scanner.nextLine().strip();

            System.out.print("Sua idade: ");
```

```
        int idade = Integer.parseInt(scanner.nextLine());

        System.out.printf("Olá, %s! Idade registrada: %d.%n", nome, idade);
    }
}
```

Se a idade não estiver em formato numérico, `Integer.parseInt` lançará `NumberFormatException`. O tratamento desse caso será desenvolvido no capítulo de exceções.

3.6 Evolução do projeto: versão 2

O chatbot agora pergunta o nome da pessoa e apresenta uma saudação personalizada. Armazene:

- nome da pessoa, como `String`;
- total de mensagens, como `int`;
- estado da conversa, como `boolean`.

O exemplo completo está em [examples/src/Capitulo02Entrada.java](#).

3.7 Prática verificável

Crie um programa que leia nome, cidade e quantidade semanal de horas de estudo. Em seguida, mostre uma frase completa.

Casos de teste:

Entrada	Resultado esperado
Ana, Maceió, 6	a frase contém os três valores
nome com espaços nas bordas	os espaços externos não aparecem

Entrada	Resultado esperado
horas iguais a 0	o programa aceita o valor e o exibe

Desafio: calcule a carga de estudo em quatro semanas, mantendo o cálculo em uma variável separada.

 Limite deste capítulo

Ainda não trate texto não numérico. Primeiro observe a exceção; depois você implementará uma recuperação controlada.

Capítulo 4

Condicionais

Transformar regras em decisões previsíveis

4.1 Objetivos de aprendizagem

Ao concluir este capítulo, você deverá ser capaz de:

- escrever decisões com `if`, `else if` e `else`;
- combinar condições com operadores relacionais e lógicos;
- escolher entre `if` e `switch`;
- comparar comandos de texto sem depender de maiúsculas ou espaços.

4.2 Expressões booleanas

Uma condição produz `true` ou `false`. Os operadores relacionais são `==`, `!=`, `>`, `<`, `>=` e `<=`. Os operadores lógicos mais usados são `&&` (E), `||` (OU) e `!` (NÃO).

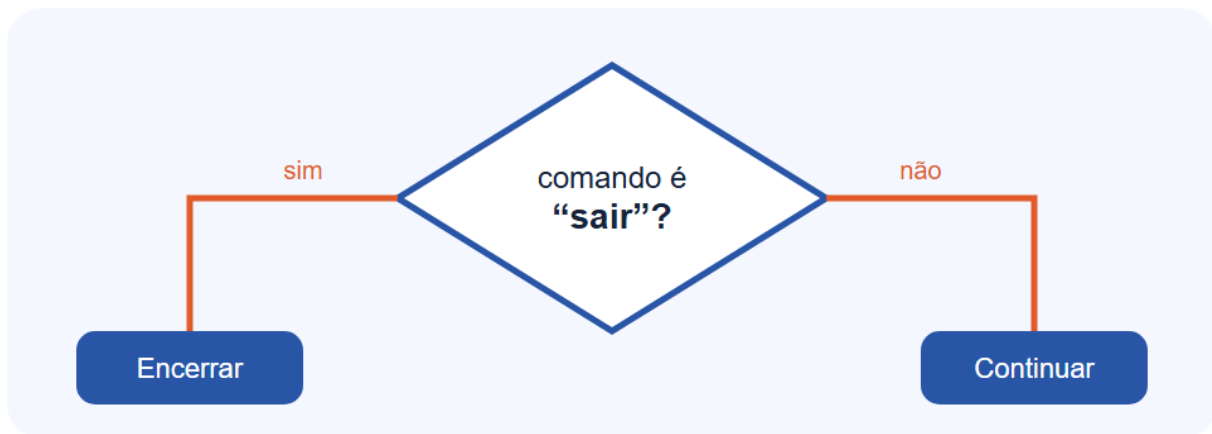


Figura 4.1: Uma condição escolhe apenas um dos caminhos possíveis.

```
double nota = 7.5;
int faltas = 2;

boolean aprovado = nota >= 7.0 && faltas <= 5;
```

Use parênteses quando eles tornarem a intenção mais clara, mesmo que a precedência da linguagem já produza o resultado correto.

4.3 Decisões com if

```
if (nota >= 7.0) {
    System.out.println("Aprovado");
} else if (nota >= 5.0) {
    System.out.println("Recuperação");
} else {
    System.out.println("Reprovado");
}
```

A ordem importa: condições mais específicas devem aparecer antes de condições mais abrangentes.

4.4 Opções fechadas com switch

Use `switch` quando uma única expressão é comparada com um conjunto pequeno de valores fixos. Use `if` para intervalos, combinações e regras diferentes entre si.

```
import java.util.Locale;

public class Capitulo03Condicionais {
    public static String responder(String entrada) {
        String comando = entrada.strip().toLowerCase(Locale.ROOT);

        return switch (comando) {
            case "oi" -> "Olá! Como posso ajudar?";
            case "ajuda" -> "Comandos: oi, dica e sair.";
            case "dica" -> "Use nomes que revelem a intenção.";
            case "sair" -> "Até logo!";
            default -> "Não entendi. Digite ajuda.";
        };
    }

    public static void main(String[] args) {
        System.out.println(responder("  AJUDA  "));
    }
}
```

`Locale.ROOT` torna a normalização previsível, independentemente do idioma configurado no computador.

4.5 Erros frequentes

- comparar `String` com `==`;

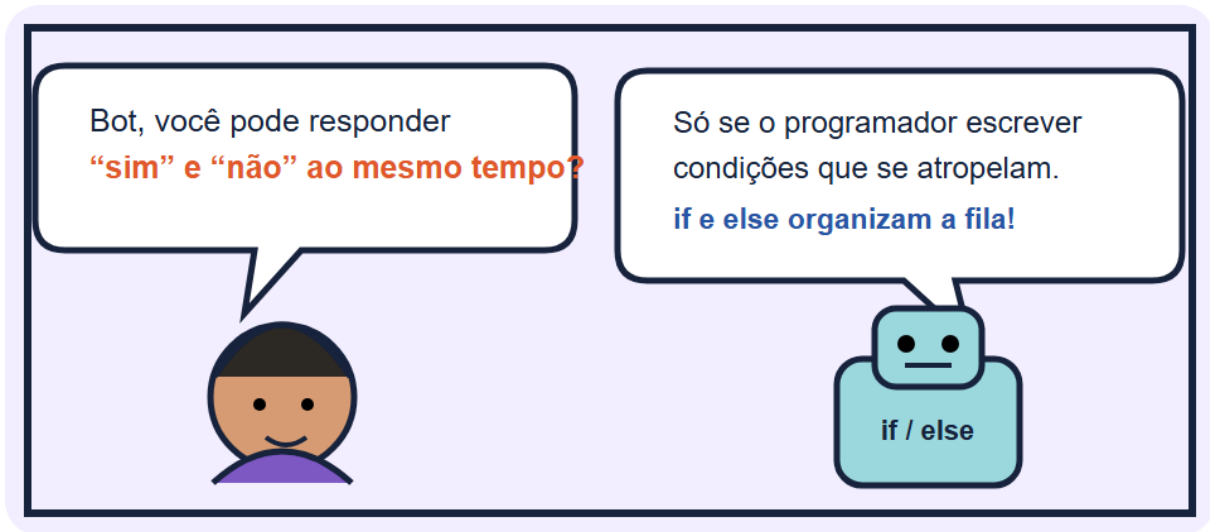


Figura 4.2: Quadrinho: o chatbot indeciso.

- repetir a mesma condição em vários pontos;
- escrever uma condição que nunca pode ser verdadeira;
- criar muitos níveis de `if` aninhados;
- não prever entrada vazia ou desconhecida.

Quando uma decisão crescer demais, extraia-a para um método com nome claro.

4.6 Evolução do projeto: versão 3

O chatbot agora reconhece `oi`, `ajuda`, `dica` e `sair`. A resposta padrão orienta a pessoa em vez de apenas informar erro.

O exemplo completo está em [examples/src/Capitulo03Condicionais.java](#).

4.7 Prática verificável

Adicione o comando `sobre` e escreva uma regra que classifique horas de estudo:

- menos de 3 horas: “ritmo inicial”;

- de 3 a 6 horas: “ritmo consistente”;
- acima de 6 horas: “ritmo intensivo”.

Casos de teste:

Entrada	Resultado esperado
AJUDA	lista de comandos
dica	dica de Java
desconhecido	orientação para digitar ajuda
3 horas	ritmo consistente

Critério de domínio

Explique por que `switch` foi usado para comandos e `if` para intervalos de horas.

Capítulo 5

Estruturas de repetição

Manter a conversa ativa com critérios de parada claros

5.1 Objetivos de aprendizagem

Ao concluir este capítulo, você deverá ser capaz de:

- escolher entre `while`, `do-while` e `for`;
- definir uma condição de parada;
- usar contadores e acumuladores;
- empregar `break` e `continue` sem esconder o fluxo.

5.2 Escolhendo o laço

Use:

- `while` quando a quantidade de repetições é desconhecida;
- `do-while` quando o bloco precisa executar ao menos uma vez;
- `for` quando existe um intervalo ou contador bem definido;

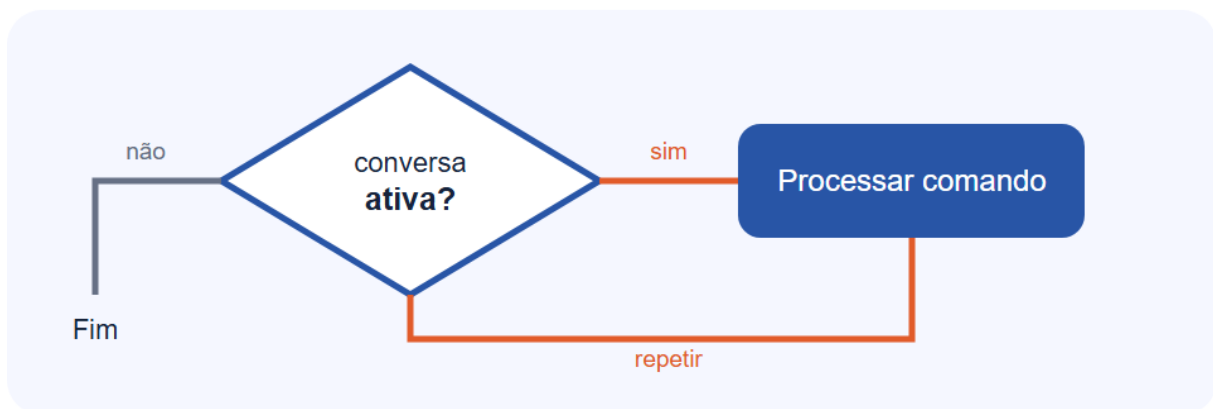


Figura 5.1: Anatomia de um laço com condição de parada.

- for-each para percorrer todos os elementos de uma estrutura.

```
for (int tentativa = 1; tentativa <= 3; tentativa++) {
    System.out.println("Tentativa " + tentativa);
}
```

5.3 Chatbot em repetição

```
import java.util.Scanner;

public class Capitulo04Repeticao {
    public static void main(String[] args) {
        int totalMensagens = 0;

        try (Scanner scanner = new Scanner(System.in)) {
            while (true) {
                System.out.print("Você: ");
                String mensagem = scanner.nextLine().strip();

                if (mensagem.equalsIgnoreCase("sair")) {
                    break;
                }
            }
        }
    }
}
```

```
    }

    if (mensagem.isBlank()) {
        System.out.println("Bot: Digite uma mensagem");
        continue;
    }

    totalMensagens++;
    System.out.println("Bot: Recebi -> " + mensagem);
}

System.out.println("Mensagens registradas: " + totalMensagens);
}
```

A condição de parada está explícita: o comando `sair` executa `break`. Entradas em branco usam `continue` e não são contabilizadas.

5.4 Contadores e acumuladores

Um contador registra quantas vezes algo aconteceu. Um acumulador agrega valores, como o total de caracteres:

```
int totalMensagens = 0;
int totalCaracteres = 0;

totalMensagens++;
totalCaracteres += mensagem.length();
```

Antes de escrever um laço, responda:

1. qual estado muda em cada iteração?

5.7 Prática verificável

Acrescente:

1. limite opcional de cinco mensagens;
2. contador de comandos desconhecidos;
3. total de caracteres digitados, sem contar **sair**.

Casos de teste:

Sequência	Resultado esperado
oi, sair	uma mensagem registrada
linha vazia, sair	zero mensagens
cinco mensagens	encerramento pelo limite
SAIR	encerramento normal

Erro clássico

Um **while** em que nenhuma variável altera a condição pode produzir repetição infinita. Verifique a condição de parada antes de executar.

Capítulo 6

Métodos

Separar decisões, cálculos e efeitos

6.1 Objetivos de aprendizagem

Ao concluir este capítulo, você deverá ser capaz de:

- identificar parâmetros, retorno e corpo de um método;
- extrair responsabilidades do método `main`;
- diferenciar cálculo de entrada e saída;
- compreender passagem de argumentos por valor;
- usar sobrecarga sem criar ambiguidade.

6.2 Contratos pequenos e claros

A assinatura comunica o que um método recebe e devolve:

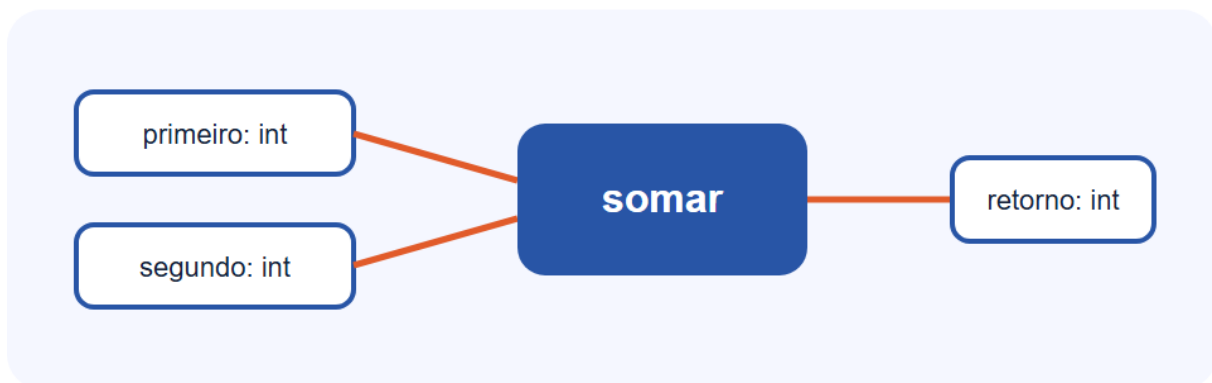


Figura 6.1: O contrato de um método transforma argumentos em um resultado.

```
public static int somar(int primeiro, int segundo) {  
    return primeiro + segundo;  
}
```

O nome deve expressar intenção. `normalizarComando` é mais informativo que `processar`.

6.3 Refatorando o chatbot

```
import java.util.Locale;  
  
public class Capitulo05Metodos {  
    public static String normalizarComando(String entrada) {  
        if (entrada == null) {  
            return "";  
        }  
        return entrada.strip().toLowerCase(Locale.ROOT);  
    }  
  
    public static String responder(String entrada) {
```

```
String comando = normalizarComando(entrada);

return switch (comando) {
    case "oi" -> "Olá!";
    case "ajuda" -> "Comandos: oi, ajuda, dica e sair.";
    case "dica" -> "Um método deve ter uma responsabili
    case "sair" -> "Até logo!";
    case "" -> "Digite uma mensagem.";
    default -> "Não entendi. Digite ajuda.";
};
}

public static boolean deveEncerrar(String entrada) {
    return normalizarComando(entrada).equals("sair");
}

public static void main(String[] args) {
    System.out.println(responder("  DICA  "));
}
}
```

`normalizarComando` e `responder` são determinísticos: para a mesma entrada, devolvem o mesmo resultado. Isso facilita testes. O `main` fica responsável apenas por coordenar a execução e mostrar resultados.

6.4 Escopo e passagem por valor

Variáveis locais existem somente no bloco em que foram declaradas. Java sempre passa argumentos **por valor**. Para tipos primitivos, o valor é copiado; para objetos, o valor copiado é a referência. O método pode alterar o objeto apontado por essa referência, mas não substituir a variável

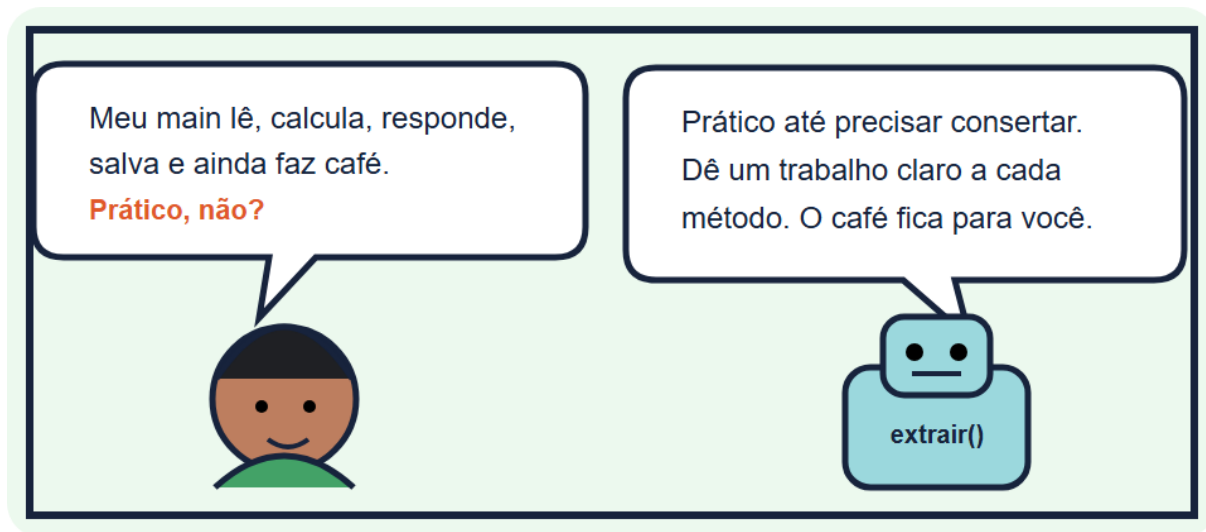


Figura 6.2: Quadrinho: o método faz-tudo.

original de quem fez a chamada.

```
public static void incrementar(int numero) {  
    numero++;  
}
```

Depois de chamar `incrementar(valor)`, a variável `valor` do chamador continua inalterada.

6.5 Sobrecarga

Sobrecarga permite métodos com o mesmo nome e listas de parâmetros diferentes:

```
public static int calcularTotal(int a, int b) {  
    return a + b;  
}  
  
public static double calcularTotal(double a, double b) {  
    return a + b;  
}
```

O tipo de retorno sozinho não diferencia duas sobrecargas.

6.6 Validação e exceções

Um método deve rejeitar valores que violam seu contrato:

```
public static double calcularMedia(double soma, int quantidade)
    if (quantidade <= 0) {
        throw new IllegalArgumentException("A quantidade deve s
    }
    return soma / quantidade;
}
```

A exceção torna a falha explícita. Mais adiante, você decidirá onde capturá-la.

6.7 Evolução do projeto: versão 5

O fluxo do chatbot agora está dividido em três decisões reutilizáveis:

- normalizar entrada;
- produzir resposta;
- decidir se a conversa deve terminar.

O exemplo completo está em [examples/src/Capitulo05Metodos.java](#).

6.8 Prática verificável

1. Crie `ehComandoConhecido(String)`.
2. Crie `mensagemDeBoasVindas(String nome)`.
3. Reescreva o laço do capítulo anterior chamando esses métodos.

Casos de teste para `normalizarComando`:

Entrada	Retorno
" AJUDA "	"ajuda"
""	""
null	""

Critério de domínio

O método `main` deve ser legível como um roteiro. Se ele contiver todas as regras, ainda há responsabilidades a extrair.

Capítulo 7

Arrays

Coleções de tamanho fixo e acesso por índice

7.1 Objetivos de aprendizagem

Ao concluir este capítulo, você deverá ser capaz de:

- criar e inicializar arrays;
- respeitar os limites dos índices;
- escolher entre `for` tradicional e `for-each`;
- calcular soma, média, maior e menor valor;
- copiar um array sem compartilhar a mesma referência.

7.2 Tamanho fixo e índices

Um array guarda vários valores do mesmo tipo e tem tamanho fixo. Seus índices começam em zero. Em um array de quatro elementos, os índices válidos são 0, 1, 2 e 3.



Figura 7.1: Índices válidos de um array com quatro elementos.

```
String[] comandos = {"oi", "ajuda", "dica", "sair"};
System.out.println(comandos[0]); // oi
System.out.println(comandos.length); // 4
```

A linguagem garante acesso por índice e tamanho fixo; detalhes sobre a disposição física dos objetos na memória pertencem à implementação da JVM e não são necessários para usar arrays corretamente.

7.3 Inicialização e valores padrão

```
int[] contadores = new int[3]; // 0, 0, 0
boolean[] estados = new boolean[2]; // false, false
String[] respostas = new String[2]; // null, null
```

Arrays de tipos primitivos recebem valores padrão. Arrays de objetos recebem `null` até que cada posição seja preenchida.

7.4 Percorrendo com segurança

```
public class Capitulo06Arrays {  
    public static void main(String[] args) {  
        double[] notas = {7.0, 8.5, 10.0, 6.5};  
        double soma = 0.0;  
        double maior = notas[0];  
  
        for (double nota : notas) {  
            soma += nota;  
            if (nota > maior) {  
                maior = nota;  
            }  
        }  
  
        double media = soma / notas.length;  
        System.out.printf("Média: %.2f%n", media);  
        System.out.printf("Maior nota: %.2f%n", maior);  
    }  
}
```

Use **for-each** para ler todos os valores. Use o **for** com índice quando precisar alterar posições ou trabalhar com dois arrays em paralelo:

```
for (int indice = 0; indice < comandos.length; indice++) {  
    System.out.println(indice + " -> " + comandos[indice]);  
}
```

A condição correta é `indice < comandos.length`. Usar `<=` tentaria acessar uma posição inexistente.

7.5 Utilitários e cópia

A classe `Arrays` oferece ordenação, busca e representação textual:

```
import java.util.Arrays;

int[] valores = {9, 3, 7, 1};
Arrays.sort(valores);
System.out.println(Arrays.toString(valores));
```

A atribuição direta compartilha o mesmo array. Para obter uma cópia independente:

```
int[] copia = Arrays.copyOf(valores, valores.length);
```

7.6 Evolução do projeto: versão 6

Use um array para armazenar os comandos obrigatórios do chatbot. Percorra-o para montar a mensagem de ajuda. O tamanho ainda é fixo, o que torna visível a limitação que será resolvida com `List`.

O exemplo completo está em [examples/src/Capitulo06Arrays.java](#).

7.7 Prática verificável

Crie um array com cinco tempos de estudo, em minutos, e mostre total, média, maior e menor valor.

Casos de teste:

Entrada	Resultado esperado
{30, 40, 20, 50, 10}	total 150, média 30
um único elemento	maior, menor e média iguais ao elemento

Entrada	Resultado esperado
array vazio	deve ser rejeitado antes de acessar o índice zero

 Caso vazio

Antes de usar `valores[0]`, confirme que `valores.length > 0`.

Capítulo 8

Listas e coleções

Armazenar quantidades variáveis de dados

8.1 Objetivos de aprendizagem

Ao concluir este capítulo, você deverá ser capaz de:

- explicar a diferença entre array e lista;
- declarar uma variável pelo contrato `List`;
- adicionar, consultar, alterar e remover elementos;
- percorrer uma lista sem acessar índices inválidos;
- usar generics para obter segurança de tipos.

8.2 `List` e `ArrayList`

Arrays têm tamanho fixo. Uma `ArrayList` pode crescer ou diminuir durante a execução. Prefira declarar a variável com a interface `List`:



Figura 8.1: Uma lista pode crescer e diminuir durante a execução.

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

List<String> historico = new ArrayList<>();
historico.add("oi");
historico.add("ajuda");
```

O tipo entre < > informa quais elementos são aceitos. `List<String>` rejeita números e outros tipos em tempo de compilação.

8.3 Operações essenciais

```
historico.add("dica");
String primeira = historico.get(0);
historico.set(0, "OI");
historico.remove("dica");

System.out.println(historico.size());
System.out.println(historico.isEmpty());
System.out.println(historico.contains("ajuda"));
```

Índices válidos vão de zero até `size() - 1`. Se a posição vier da pessoa usuária, valide-a antes de chamar `get`, `set` ou `remove`.

8.4 Exemplo completo

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

public class Capitulo07Colecoes {
    public static void main(String[] args) {
        List<String> historico = new ArrayList<>();
        historico.add("oi");
        historico.add("ajuda");
        historico.add("dica");

        for (String mensagem : historico) {
            System.out.println("Usuário: " + mensagem);
        }

        String ultima = historico.get(historico.size() - 1);
        System.out.println("Última mensagem: " + ultima);
        System.out.println("Total: " + historico.size());
    }
}
```

8.5 Cópias e exposição de estado

Se uma classe devolver sua lista interna, código externo poderá alterá-la sem passar pelas regras da classe. Uma cópia somente para leitura evita esse problema:

```
public List<String> getHistorico() {  
    return List.copyOf(historico);  
}
```

8.6 Autoboxing

Coleções armazenam objetos. Para números inteiros, use `Integer`, não `int`:

```
List<Integer> pontuacoes = new ArrayList<>();  
pontuacoes.add(10); // int convertido para Integer
```

A conversão automática entre `int` e `Integer` é chamada autoboxing e unboxing.

8.7 Evolução do projeto: versão 7

Substitua o contador isolado por uma lista de mensagens válidas. O comando `historico` deve consultar a lista, mas não entrar nela; assim, “total de mensagens” significa apenas mensagens da pessoa usuária processadas antes da consulta.

O exemplo completo está em [examples/src/Capitulo07Colecoes.java](#).

8.8 Prática verificável

Implemente métodos para:

1. registrar mensagem não vazia;
2. devolver a última mensagem;
3. devolver, no máximo, as três mensagens mais recentes;
4. limitar o histórico a dez itens.

Casos de teste:

Estado	Resultado esperado
lista vazia	mensagem "Sem histórico"
dois itens	os dois aparecem, na ordem
doze inserções com limite dez	apenas as dez últimas permanecem

 Escolha da estrutura

Use array quando o tamanho fizer parte da regra. Use `List` quando a quantidade variar durante a execução.

Capítulo 9

Classes e objetos

Reunir estado, comportamento e invariantes

9.1 Objetivos de aprendizagem

Ao concluir este capítulo, você deverá ser capaz de:

- diferenciar classe, objeto, atributo e método;
- inicializar objetos com construtores;
- usar `this` para distinguir atributo e parâmetro;
- proteger o estado com `private`;
- representar regras por operações, não por setters genéricos.

9.2 Estado e comportamento

Uma classe define a estrutura e as operações disponíveis. Um objeto é uma instância concreta dessa classe. Seus atributos formam o estado; seus métodos definem comportamentos.

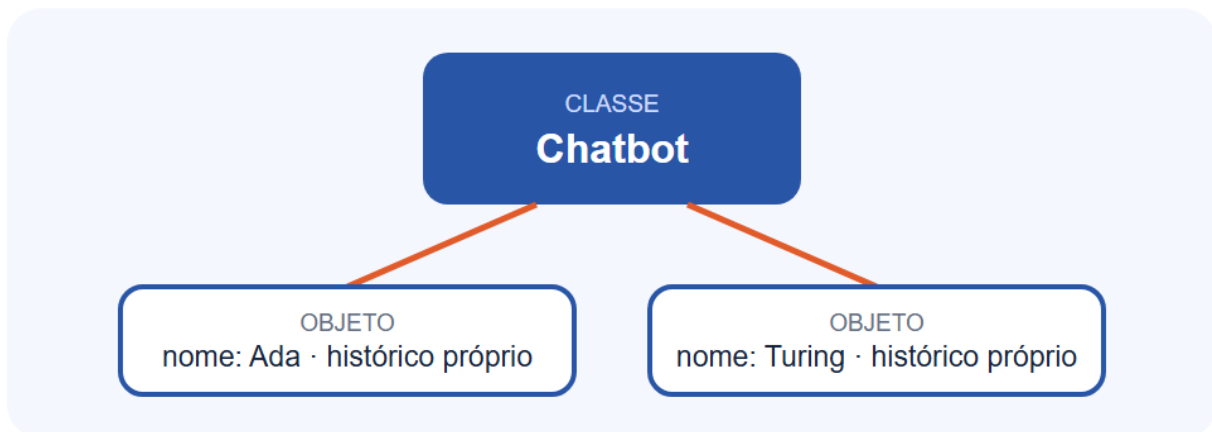


Figura 9.1: A classe define; cada objeto mantém seu próprio estado.

```
class ContaPontos {  
    private int saldo;  
  
    public ContaPontos(int saldoInicial) {  
        if (saldoInicial < 0) {  
            throw new IllegalArgumentException("Saldo inicial i  
        }  
        this.saldo = saldoInicial;  
    }  
  
    public void creditar(int valor) {  
        if (valor <= 0) {  
            throw new IllegalArgumentException("0 crédito deve  
        }  
        saldo += valor;  
    }  
  
    public boolean debitar(int valor) {  
        if (valor <= 0 || valor > saldo) {  
            return false;  
        }  
    }  
}
```

```
        saldo -= valor;  
        return true;  
    }  
  
    public int getSaldo() {  
        return saldo;  
    }  
}
```

A validação de `valor <= 0` é indispensável: sem ela, subtrair um valor negativo aumentaria o saldo. O atributo privado só pode mudar por operações que preservam as regras.

9.3 Construtores e this

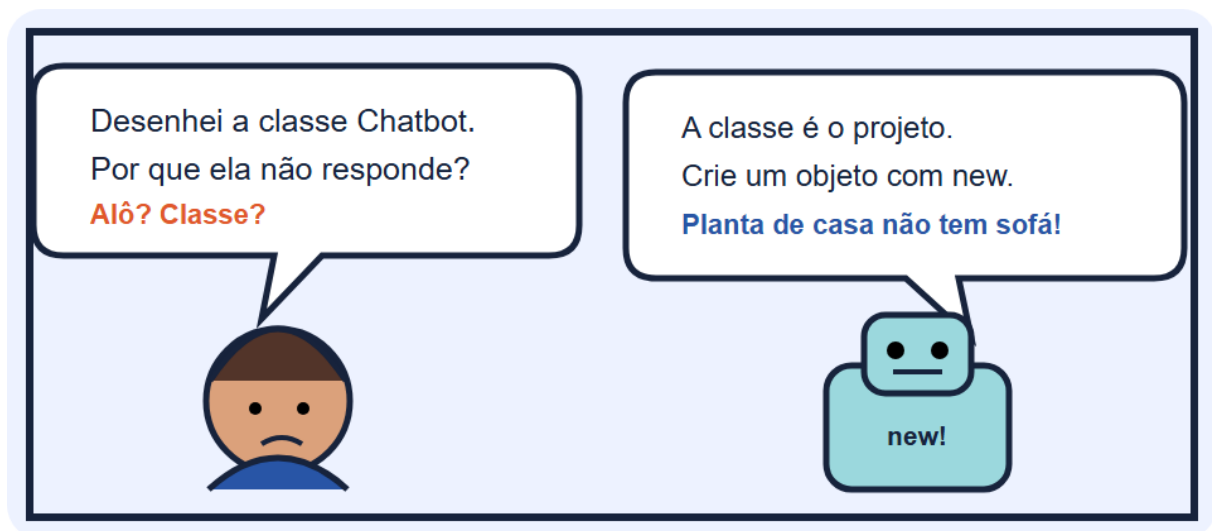


Figura 9.2: Quadrinho: classe não é objeto.

O construtor cria um estado inicial válido. `this.nome` indica o atributo do objeto atual, enquanto `nome` representa o parâmetro:

```
class Chatbot {  
    private final String nome;
```

```
public Chatbot(String nome) {  
    if (nome == null || nome.isBlank()) {  
        throw new IllegalArgumentException("Nome obrigatório");  
    }  
    this.nome = nome.strip();  
}  
}
```

`final` impede que a referência seja reatribuída depois da construção.

9.4 Classe do chatbot

```
import java.util.ArrayList;  
import java.util.List;  
  
class ChatbotBasico {  
    private final String nome;  
    private final List<String> historico = new ArrayList<>();  
  
    ChatbotBasico(String nome) {  
        if (nome == null || nome.isBlank()) {  
            throw new IllegalArgumentException("Nome obrigatório");  
        }  
        this.nome = nome.strip();  
    }  
  
    String responder(String mensagem) {  
        if (mensagem == null || mensagem.isBlank()) {  
            return "Digite uma mensagem.";  
        }  
    }  
}
```

```
        historico.add(mensagem.strip());
        return nome + ": recebi sua mensagem.";
    }

    List<String> consultarHistorico() {
        return List.copyOf(historico);
    }
}

public class Capitulo08Classes {
    public static void main(String[] args) {
        ChatbotBasico bot = new ChatbotBasico("TutorJavaBot");
        System.out.println(bot.responder("oi"));
        System.out.println(bot.consultarHistorico());
    }
}
```

A cópia devolvida por `consultarHistorico` impede alteração externa da lista interna.

9.5 Composição inicial

Composição representa uma relação “tem um”. Um chatbot pode ter um objeto responsável pelo histórico; uma turma pode ter uma lista de alunos. Quando a responsabilidade crescer, mover o histórico para uma classe própria evita que o chatbot concentre trabalho demais.

9.6 Evolução do projeto: versão 8

Transforme variáveis e métodos isolados em um objeto `ChatbotBasico`. O `main` cria o objeto e coordena entrada e saída; a classe protege nome,

histórico e regras de resposta.

O exemplo completo está em [examples/src/Capitulo08Classes.java](#).

9.7 Prática verificável

1. Acrescente o método `totalMensagens()`.
2. Rejeite nomes vazios no construtor.
3. Confirme que a lista devolvida não pode alterar o histórico interno.
4. Explique qual invariante cada validação protege.

i Invariante

Uma invariante é uma condição que deve permanecer verdadeira durante toda a vida do objeto, como “o nome do chatbot não é vazio”.

Capítulo 10

Tratamento de erros e arquivos

Recuperar entradas inválidas e persistir o histórico

10.1 Objetivos de aprendizagem

Ao concluir este capítulo, você deverá ser capaz de:

- distinguir erro de compilação, exceção e erro de lógica;
- capturar apenas exceções que podem ser tratadas naquele ponto;
- repetir a leitura até obter uma entrada válida;
- lançar `IllegalArgumentException` para contratos violados;
- salvar texto em arquivo com tratamento de `IOException`.



Figura 10.1: Recuperação de uma entrada inválida.

10.2 Exceções esperadas

Em programas interativos, entradas inválidas são previsíveis. `Integer.parseInt` lança `NumberFormatException` quando o texto não representa um inteiro:

```
try {  
    int idade = Integer.parseInt(entrada);  
    System.out.println("Idade: " + idade);  
} catch (NumberFormatException erro) {  
    System.out.println("Digite um número inteiro.");  
}
```

Não capture `Exception` sem necessidade. Um tipo específico documenta o cenário esperado e evita esconder defeitos diferentes.

10.3 Repetir até obter um valor válido

```
import java.util.Scanner;  
  
static int lerInteiro(Scanner scanner, String mensagem) {
```

```
while (true) {  
    System.out.print(mensagem);  
    String entrada = scanner.nextLine();  
  
    try {  
        return Integer.parseInt(entrada);  
    } catch (NumberFormatException erro) {  
        System.out.println("Valor inválido. Tente novamente");  
    }  
}
```

A leitura e a recuperação ficam centralizadas. O restante do programa recebe um inteiro válido.

10.4 Lançar uma exceção

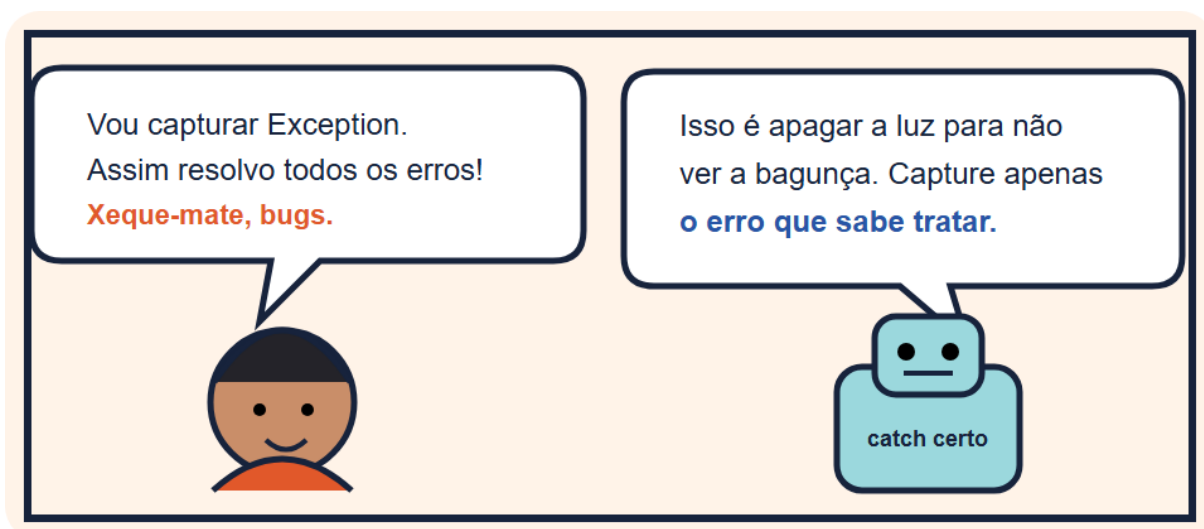


Figura 10.2: Quadrinho: capturar todas as exceções.

Use uma exceção quando o chamador viola o contrato de um método:

```
static void validarIdade(int idade) {  
    if (idade < 0 || idade > 130) {  
        throw new IllegalArgumentException("Idade fora do intervalo");  
    }  
}
```

Capturar ou propagar depende da responsabilidade. A interface de console pode capturar e mostrar uma orientação; uma classe de domínio normalmente lança a exceção e deixa a decisão para a camada externa.

10.5 Persistência com try-with-resources

A API Files simplifica a escrita de linhas e gerencia o fechamento do arquivo:

```
import java.io.IOException;  
import java.nio.file.Files;  
import java.nio.file.Path;  
import java.util.List;  
  
static void salvarHistorico(Path arquivo, List<String> linhas)  
    throws IOException {  
    Files.write(arquivo, linhas);  
}
```

No limite externo do programa, trate a falha:

```
try {  
    salvarHistorico(Path.of("historico.txt"), historico);  
    System.out.println("Histórico salvo.");  
} catch (IOException erro) {  
    System.out.println("Não foi possível salvar o histórico.");  
    System.err.println("Detalhe técnico: " + erro.getMessage());  
}
```

```
}
```

A mensagem para a pessoa usuária é curta; o detalhe técnico vai para o fluxo de erro.

10.6 Exemplo completo

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.List;

public class Capitulo09ErrosArquivos {
    public static void salvarHistorico(Path arquivo, List<String> historico)
        throws IOException {
        Files.write(arquivo, historico);
    }

    public static void main(String[] args) {
        List<String> historico = List.of("USUÁRIO: oi", "BOT: oi");

        try {
            salvarHistorico(Path.of("historico.txt"), historico);
            System.out.println("Histórico salvo.");
        } catch (IOException erro) {
            System.out.println("Não foi possível salvar o histórico.");
            System.err.println(erro.getMessage());
        }
    }
}
```

10.7 Evolução do projeto: versão 9

O chatbot recupera entradas numéricas inválidas e oferece a opção de salvar o histórico ao encerrar. A falha de escrita não destrói a conversa nem mostra uma pilha de exceção para a pessoa usuária.

O exemplo completo está em [examples/src/Capitulo09ErrosArquivos.java](#).

10.8 Prática verificável

1. Leia uma idade até receber valor numérico entre 0 e 130.
2. Salve três mensagens em `historico.txt`.
3. Teste um caminho de arquivo inválido e confirme a mensagem amigável.
4. Explique por que `NumberFormatException` e `IOException` são tratadas em lugares diferentes.

 Não use exceção como validação comum

Condições simples, como texto vazio, devem ser verificadas diretamente. Exceções são apropriadas quando uma operação não consegue cumprir seu contrato.

Capítulo 11

Chatbot de console

Integrar os fundamentos em uma aplicação completa

11.1 Objetivos de aprendizagem

Ao concluir este capítulo, você deverá ser capaz de:

- integrar entrada, decisão, repetição, métodos, listas e exceções;
- definir claramente o contrato de cada comando;
- separar interação de console das regras do chatbot;
- manter uma semântica consistente para o histórico.

11.2 Contrato de comandos

Nesta versão:

- `oi`: apresenta o bot;
- `ajuda`: lista comandos;
- `dica`: mostra uma dica de Java;



Figura 11.1: Fluxo principal do chatbot de console.

- **historico**: mostra interações anteriores;
- **sair**: encerra a conversa.

Mensagens vazias não entram no histórico. Os comandos **historico** e **sair** também não são registrados. Cada interação normal gera duas linhas: uma da pessoa usuária e uma do bot.

11.3 Implementação integrada

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

public class Capitulo10Chatbot {
    private final String nome;
    private final List<String> historico = new ArrayList<>();

    public Capitulo10Chatbot(String nome) {
        if (nome == null || nome.isBlank()) {
            throw new IllegalArgumentException("Nome obrigatório");
        }
    }
}
```

```
    }
    this.nome = nome.strip();
}

private static String normalizar(String texto) {
    if (texto == null) {
        return "";
    }
    return texto.strip().toLowerCase(Locale.ROOT);
}

public String responder(String entrada) {
    String comando = normalizar(entrada);

    if (comando.isBlank()) {
        return "Digite uma mensagem.";
    }
    if (comando.equals("historico")) {
        return historico.isEmpty()
            ? "Sem histórico."
            : String.join(System.lineSeparator(), historico);
    }

    String resposta = switch (comando) {
        case "oi" -> "Olá! Eu sou " + nome + ".";
        case "ajuda" -> "Comandos: oi, ajuda, dica, historico";
        case "dica" -> "Compile e teste depois de cada pequena alteração";
        default -> "Não entendi. Digite ajuda.";
    };

    historico.add("USUÁRIO: " + entrada.strip());
}
```

```
        historico.add("BOT: " + resposta);
        return resposta;
    }

    public int totalInteracoes() {
        return historico.size() / 2;
    }

    public static void main(String[] args) {
        Capitulo10Chatbot bot = new Capitulo10Chatbot("TutorJava");

        try (Scanner scanner = new Scanner(System.in)) {
            System.out.println("Digite ajuda para ver os comandos");

            while (true) {
                System.out.print("Você: ");
                String entrada = scanner.nextLine();

                if (normalizar(entrada).equals("sair")) {
                    break;
                }

                System.out.println("Bot: " + bot.responder(entrada));
            }

            System.out.println("Interações registradas: " + bot.totalInteracoes());
            System.out.println("Bot: Até logo!");
        }
    }
}
```

11.4 Separação de responsabilidades

A classe ainda contém a interface de console e as regras, mas a fronteira já está visível:

- **main**: lê e escreve no terminal;
- **normalizar**: padroniza comandos;
- **responder**: aplica regras;
- **historico**: representa o estado da conversa.

Na segunda parte do livro, essas responsabilidades serão distribuídas entre classes e contratos menores.

11.5 Evolução do projeto: versão 10

Esta versão conclui a trilha de fundamentos. O programa compila, executa até **sair**, orienta entradas inválidas e mantém histórico com significado definido.

O exemplo completo está em [examples/src/Capitulo10Chatbot.java](#).

11.6 Roteiro de teste manual

Cenário	Sequência	Resultado
fluxo normal	oi, dica, sair	duas interações
comando desconhecido	xyz	orientação para ajuda
entrada vazia	Enter	mensagem de correção, sem registro
histórico	oi, historico	duas linhas da interação anterior
caixa e espaços	AJUDA	lista de comandos

11.7 Prática verificável

1. Mostre somente as seis últimas linhas do histórico.
2. Acrescente o comando **sobre**.
3. Salve o histórico ao encerrar.
4. Evite duplicar a lista de comandos em diferentes métodos.

Checkpoint

Guarde esta versão funcionando. Os próximos capítulos mudarão a estrutura interna sem alterar o contrato observado pela pessoa usuária.

Parte II

Parte II — Orientação a objetos e evolução do projeto

Capítulo 12

Design orientado a objetos

Encapsulamento, coesão e composição no chatbot

12.1 Objetivos de aprendizagem

Ao concluir este capítulo, você deverá ser capaz de:

- distribuir responsabilidades entre classes coesas;
- proteger invariantes com encapsulamento;
- escolher modificadores de acesso;
- usar composição para colaborar sem herança;
- identificar acoplamento desnecessário.

12.2 Da classe grande às responsabilidades

A versão anterior concentra entrada, regras e histórico. Uma primeira decomposição pode criar:

- **Histórico:** registra e consulta interações;



Figura 12.1: Responsabilidades separadas colaboram por composição.

- `ProcessadorComandos`: escolhe respostas;
- `AplicacaoConsole`: coordena terminal e encerramento.

Uma classe é coesa quando seus dados e métodos trabalham para o mesmo objetivo. Acoplamento é a dependência entre partes; o objetivo não é eliminá-lo, mas depender de contratos pequenos e estáveis.

12.3 Encapsulamento e acesso

Modificador	Acesso principal
<code>private</code>	somente a própria classe
sem modificador	classes do mesmo pacote
<code>protected</code>	mesmo pacote e subclasses
<code>public</code>	qualquer classe com acesso ao tipo

Comece com a menor visibilidade necessária. Tornar um atributo `private` não basta: a classe também deve oferecer operações que preservem seu estado válido.

12.4 Composição

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

final class Historico {
    private final List<String> linhas = new ArrayList<>();

    void registrarInteracao(String entrada, String resposta) {
        if (entrada == null || entrada.isBlank()) {
            throw new IllegalArgumentException("Entrada obrigat
        }
        linhas.add("USUÁRIO: " + entrada.strip());
        linhas.add("BOT: " + resposta);
    }

    List<String> consultar() {
        return List.copyOf(linhas);
    }

    int totalInteracoes() {
        return linhas.size() / 2;
    }
}

final class ProcessadorComandos {
    String responder(String comando) {
        return switch (comando) {
            case "oi" -> "Olá!";
            case "ajuda" -> "Comandos: oi, ajuda e dica.";
            case "dica" -> "Prefira composição para relações do
```

```
        default -> "Não entendi.";
    };
}
}

public class Capitulo11Design {
    public static void main(String[] args) {
        Historico historico = new Historico();
        ProcessadorComandos processador = new ProcessadorComand

        String entrada = "dica";
        String resposta = processador.responder(entrada);
        historico.registrarInteracao(entrada, resposta);

        System.out.println(resposta);
        System.out.println(historico.consultar());
    }
}
```

`Historico` não decide respostas. `ProcessadorComandos` não escreve no console nem conhece a estrutura do histórico. A aplicação compõe os objetos.

12.5 static e final

Use `static` para comportamento ou estado que pertence à classe, não a uma instância específica. Evite transformá-lo em atalho para estado global mutável.

Use `final` para comunicar que:

- uma referência não será reatribuída;

- um método não poderá ser sobrescrito;
- uma classe não foi projetada para herança.

12.6 Revisão de design

Pergunte:

1. cada classe muda por um motivo principal?
2. uma regra aparece em apenas um lugar?
3. as coleções internas estão protegidas?
4. seria possível trocar a interface de console sem reescrever o domínio?
5. os nomes usam a linguagem do problema?

O exemplo completo está em [examples/src/Capitulo11Design.java](#).

12.7 Prática verificável

Extraia da versão 10 uma classe `Historico`. A aplicação deve continuar produzindo as mesmas respostas.

Casos de teste:

- histórico vazio devolve uma lista vazia;
- uma interação gera exatamente duas linhas;
- alterar a lista devolvida não altera o estado interno;
- entrada vazia é rejeitada.

Regra de modelagem

Use composição para relações “tem um”: o chatbot **tem um** histórico. Herança será reservada a relações legítimas de especialização.

Capítulo 13

Herança

Especializar comportamento sem criar hierarquias frágeis

13.1 Objetivos de aprendizagem

Ao concluir este capítulo, você deverá ser capaz de:

- reconhecer uma relação legítima de especialização;
- inicializar a superclasse com `super`;
- sobrescrever métodos com `@Override`;
- usar classes abstratas;
- justificar quando composição é preferível.

13.2 Relação é-um

Herança representa uma relação “é um”. Antes de usar `extends`, teste a frase “todo X é um Y” e confirme que a subclasse pode respeitar o contrato da superclasse.

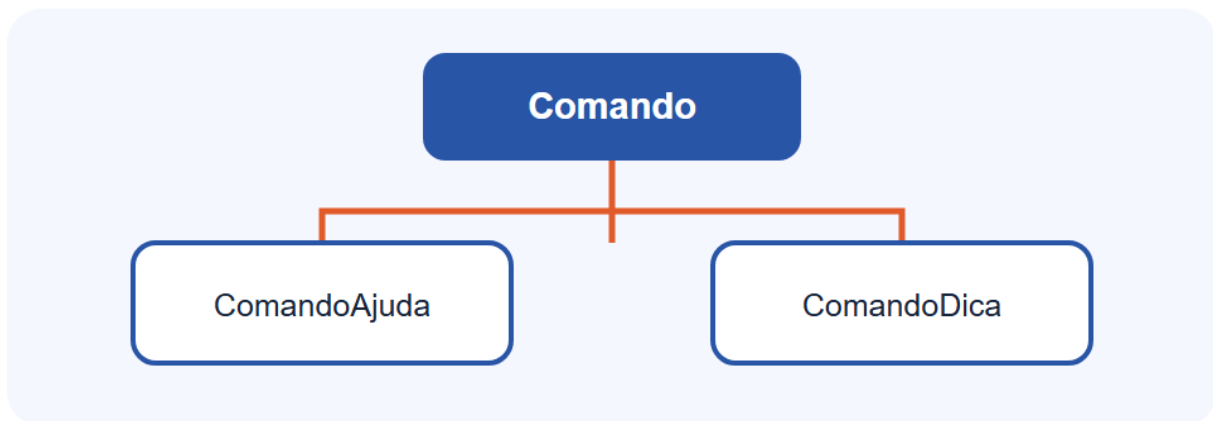


Figura 13.1: Uma hierarquia simples baseada em uma relação é-um.

```
abstract class MensagemBot {
    private final String categoria;

    protected MensagemBot(String categoria) {
        this.categoria = categoria;
    }

    public String getCategoria() {
        return categoria;
    }

    public abstract String texto();
}

final class Saudacao extends MensagemBot {
    Saudacao() {
        super("saudação");
    }

    @Override
    public String texto() {
```

```
        return "Olá! Vamos estudar Java?";
    }
}

final class Dica extends MensagemBot {
    Dica() {
        super("dica");
    }

    @Override
    public String texto() {
        return "Mantenha métodos pequenos e testáveis.";
    }
}

public class Capitulo12Heranca {
    public static void main(String[] args) {
        MensagemBot mensagem = new Dica();
        System.out.println(mensagem.getCategoria());
        System.out.println(mensagem.texto());
    }
}
```

A referência tem o tipo `MensagemBot`, mas o método executado pertence a `Dica`. Esse comportamento prepara o estudo de polimorfismo.

13.3 Construtores e super

Construtores não são herdados. A inicialização da superclasse ocorre antes da subclasse. Se a superclasse não tiver construtor sem parâmetros, a subclasse precisa chamar `super(...)`.

Membros `private` continuam presentes no objeto, mas só são acessados pelos métodos da própria superclasse. Evite transformar atributos em `protected` apenas para facilitar acesso; prefira operações protegidas ou públicas que mantenham as invariantes.

13.4 Sobrescrita

`@Override` permite que o compilador confirme a intenção de sobrescrever. A assinatura deve ser compatível e a subclasse não deve enfraquecer o contrato esperado.

Sobrecarga e sobrescrita são diferentes:

- **sobrecarga:** mesmo nome, parâmetros diferentes, decisão em compilação;
- **sobrescrita:** mesma assinatura em uma subclasse, decisão em execução.

13.5 Classes abstratas e final

Uma classe abstrata representa um conceito incompleto e pode combinar estado, métodos concretos e métodos abstratos. Uma classe `final` bloqueia herança quando a extensão não foi planejada.

Não crie hierarquias profundas. Duas ou três camadas já exigem atenção; cadeias maiores tornam alterações e testes mais difíceis.

13.6 Quando preferir composição

- Carro **tem um** motor: composição.
- Chatbot **tem um** histórico: composição.
- Dica **é uma** mensagem do bot: herança pode ser adequada.

Se o objetivo é apenas reutilizar algumas linhas, herança provavelmente é a ferramenta errada.

O exemplo completo está em [examples/src/Capitulo12Heranca.java](#).

13.7 Prática verificável

Crie uma nova subclasse **Aviso** que devolva uma mensagem de atenção. Depois, responda:

1. qual estado pertence à superclasse?
2. qual comportamento foi sobrescrito?
3. a substituição de **Dica** por **Aviso** mantém o programa válido?
4. seria melhor modelar esse caso com composição?

 Herança não é cópia de código

Use herança para representar substituição comportamental, não como mecanismo automático de reuso.

Capítulo 14

Interfaces e polimorfismo

Adicionar comandos sem alterar o processador

14.1 Objetivos de aprendizagem

Ao concluir este capítulo, você deverá ser capaz de:

- definir um contrato com `interface`;
- implementar o mesmo contrato em classes diferentes;
- usar referências do tipo da interface;
- explicar despacho dinâmico;
- adicionar comportamento sem editar uma cadeia de condicionais.

14.2 Contrato de comando

Uma interface descreve o que um objeto oferece sem impor uma hierarquia de estado:

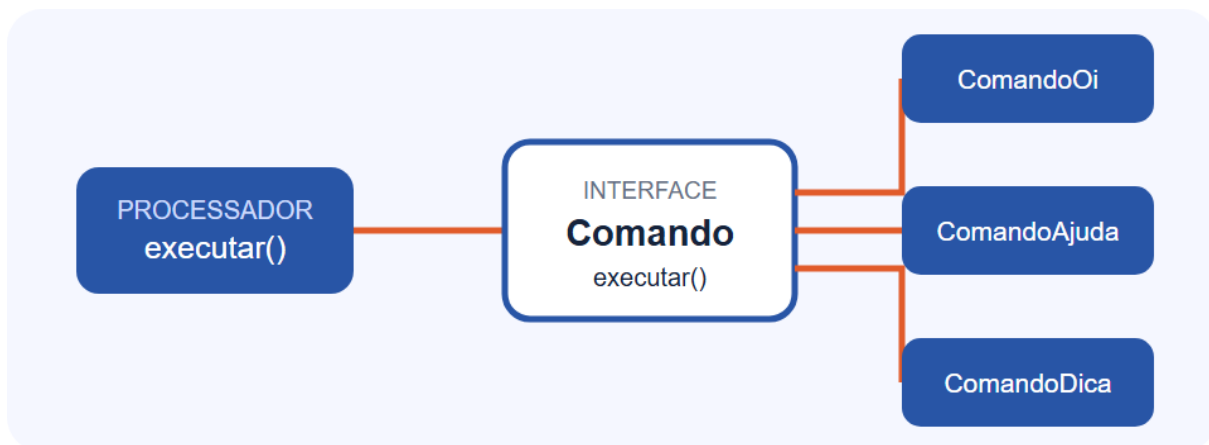


Figura 14.1: Implementações diferentes atendem ao mesmo contrato.

```
interface Comando {  
    String nome();  
    String executar();  
}
```

Diferentes classes podem implementar o contrato:

```
final class ComandoAjuda implements Comando {  
    @Override  
    public String nome() {  
        return "ajuda";  
    }  
  
    @Override  
    public String executar() {  
        return "Comandos: ajuda e dica.";  
    }  
}  
  
final class ComandoDica implements Comando {  
    @Override  
    public String nome() {
```

```
        return "dica";
    }

    @Override
    public String executar() {
        return "Programe contra contratos pequenos.";
    }
}
```

14.3 Polimorfismo em execução

```
import java.util.List;

interface Comando {
    String nome();
    String executar();
}

final class ComandoAjuda implements Comando {
    public String nome() {
        return "ajuda";
    }

    public String executar() {
        return "Comandos: ajuda e dica.";
    }
}

final class ComandoDica implements Comando {
    public String nome() {
```

```
        return "dica";
    }

    public String executar() {
        return "Programe contra contratos pequenos.";
    }
}

public class Capitulo13Polimorfismo {
    static String processar(String entrada, List<Comando> comandos) {
        for (Comando comando : comandos) {
            if (comando.nome().equalsIgnoreCase(entrada)) {
                return comando.executar();
            }
        }
        return "Comando desconhecido.";
    }

    public static void main(String[] args) {
        List<Comando> comandos =
            List.of(new ComandoAjuda(), new ComandoDica());

        System.out.println(processar("dica", comandos));
    }
}
```

A lista armazena objetos diferentes por meio do mesmo contrato. Quando `executar()` é chamado, a JVM seleciona a implementação do objeto concreto. Isso é despacho dinâmico.

14.4 Interface ou classe abstrata?

Prefira interface quando o foco é um papel que classes diferentes podem desempenhar. Use classe abstrata quando as subclasses realmente compartilham estado e implementação.

Uma classe pode implementar várias interfaces, mas só pode estender uma classe.

14.5 Evolução do projeto: versão 13

Cada comando passa a ser um objeto. Para adicionar `ComandoSobre`, crie uma classe e registre uma instância; a lógica central de processamento permanece inalterada.

O exemplo completo está em [examples/src/Capitulo13Polimorfismo.java](#).

14.6 Prática verificável

1. Implemente `ComandoSobre`.
2. Adicione-o à lista.
3. Confirme que `processar` não foi alterado.
4. Crie uma implementação de teste que sempre devolva "OK".

Casos de teste:

Entrada	Implementação chamada
ajuda	<code>ComandoAjuda</code>
DICA	<code>ComandoDica</code>
xyz	nenhuma; resposta padrão

 Sinal de polimorfismo útil

O código consumidor conhece o contrato **Comando**, não uma sequência de verificações para cada classe concreta.

Capítulo 15

Generics e coleções

List, Set e Map com contratos de tipo

15.1 Objetivos de aprendizagem

Ao concluir este capítulo, você deverá ser capaz de:

- escolher entre `List`, `Set` e `Map`;
- explicar como generics evitam casts inseguros;
- criar uma classe genérica simples;
- registrar comandos por nome;
- reconhecer coleções mutáveis e imutáveis.

15.2 Escolha da coleção

Estrutura	Use quando
<code>List<E></code>	ordem e repetição importam

Estrutura	Use quando
<code>Set<E></code>	elementos não podem se repetir
<code>Map<K, V></code>	um valor é localizado por uma chave

A interface descreve o contrato; a implementação define detalhes de desempenho e ordenação. Por exemplo, `HashSet` costuma oferecer busca eficiente, mas a interface `Set` sozinha não promete uma implementação específica.

15.3 Registro de comandos com Map

```
import java.util.LinkedHashMap;
import java.util.Locale;
import java.util.Map;

interface Comando {
    String nome();
    String executar();
}

final class RegistroComandos {
    private final Map<String, Comando> comandos = new LinkedHashMap<>();

    void registrar(Comando comando) {
        String chave = comando.nome().toLowerCase(Locale.ROOT);
        if (comandos.putIfAbsent(chave, comando) != null) {
            throw new IllegalArgumentException("Comando duplicado");
        }
    }
}
```

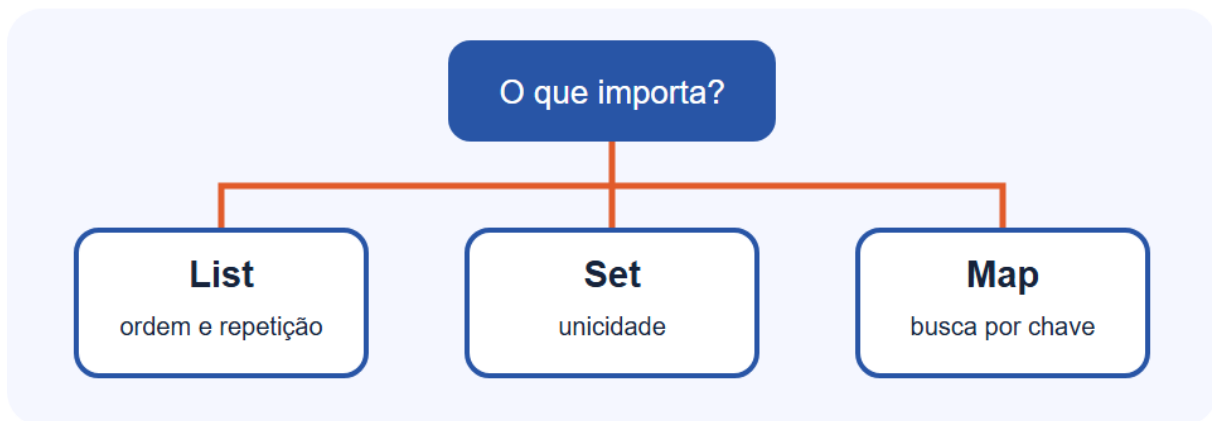


Figura 15.1: A necessidade do problema orienta a escolha da coleção.

```
String executar(String entrada) {
    String chave = entrada.strip().toLowerCase(Locale.ROOT);
    Comando comando = comandos.get(chave);
    return comando == null
        ? "Comando desconhecido."
        : comando.executar();
}
```

O mapa substitui a busca linear: o nome normalizado funciona como chave e o objeto polimórfico funciona como valor. Não é necessário introduzir `Runnable`; o contrato `Comando` expressa melhor o domínio.

15.4 Classe genérica

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

final class Repositorio<T> {
    private final List<T> itens = new ArrayList<>();
}
```

```
void adicionar(T item) {  
    itens.add(item);  
}  
  
List<T> listar() {  
    return List.copyOf(itens);  
}  
}
```

T representa um tipo escolhido por quem usa a classe. `Repositorio<String>` aceita textos; `Repositorio<Comando>` aceita comandos. O compilador impede misturas indevidas.

15.5 Exemplo completo

```
import java.util.LinkedHashMap;  
import java.util.Locale;  
import java.util.Map;  
  
public class Capitulo14Generics {  
    interface Comando {  
        String executar();  
    }  
  
    private final Map<String, Comando> comandos = new LinkedHashMap<>();  
  
    void registrar(String nome, Comando comando) {  
        String chave = nome.toLowerCase(Locale.ROOT);  
        if (comandos.putIfAbsent(chave, comando) != null) {  
            throw new IllegalArgumentException("Comando duplicado");  
        }  
    }  
}
```

```
}

String processar(String entrada) {
    Comando comando =
        comandos.get(entrada.strip().toLowerCase(Locale.ROOT));
    return comando == null ? "Comando desconhecido." : comando.getDescricao();
}

public static void main(String[] args) {
    Capitulo14Generics registro = new Capitulo14Generics();
    registro.registrar("ajuda", () -> "Comandos disponíveis");
    registro.registrar("dica", () -> "Use Map para localização");

    System.out.println(registro.processar("DICA"));
}
}
```

A expressão lambda implementa a única operação de **Comando**. Ela aparece aqui somente depois de o contrato ter sido estudado; uma classe nomeada continua válida quando o comportamento exigir estado ou explicação adicional.

List.of e **Map.of** criam coleções que não aceitam alterações. Use-as para dados fixos. Para registros que crescem durante a execução, escolha uma implementação mutável, como **ArrayList** ou **LinkedHashMap**.

O exemplo completo está em [examples/src/Capitulo14Generics.java](#).

15.6 Prática verificável

1. Rejeite nomes de comando vazios.
2. Impeça duplicação de comandos.

3. Crie um `Set<String>` com categorias de dicas.
4. Use `Repositorio<String>` para armazenar perguntas de quiz.

Casos de teste:

- registrar a mesma chave duas vezes lança exceção;
- " DICA " encontra o comando `dica`;
- comando desconhecido devolve a mensagem padrão;
- a lista retornada pelo repositório não altera o estado interno.

Capítulo 16

Testes e refatoração

Mudar a estrutura sem perder comportamento

16.1 Objetivos de aprendizagem

Ao concluir este capítulo, você deverá ser capaz de:

- transformar requisitos em casos de teste;
- separar teste de unidade e teste de fluxo;
- aplicar o ciclo vermelho-verde-refatorar;
- reconhecer duplicação e responsabilidades excessivas;
- refatorar preservando resultados observáveis.

16.2 Teste como especificação

Um teste contém preparação, ação e verificação. Antes de adotar uma biblioteca externa, é possível praticar a ideia com um verificador simples:



Figura 16.1: Ciclo curto de desenvolvimento orientado por testes.

```
import java.util.Locale;

public class Capitulo15Testes {
    static String normalizar(String entrada) {
        if (entrada == null) {
            return "";
        }
        return entrada.strip().toLowerCase(Locale.ROOT);
    }

    static void verificar(String descricao, Object esperado, Object obtido) {
        if (!esperado.equals(obtido)) {
            throw new AssertionError(
                descricao + " | esperado=" + esperado + ", obtido=" + obtido);
        }
    }

    public static void main(String[] args) {
        verificar("remove espaços", "ajuda", normalizar(" ajuda"));
        verificar("ignora maiúsculas", "dica", normalizar("DICA"));
        verificar("trata null", "", normalizar(null));
    }
}
```

```
        System.out.println("3 testes passaram.");  
    }  
}
```

Esse recurso didático não substitui JUnit em projetos profissionais, mas torna explícita a estrutura de um teste sem adicionar dependências ao livro.

16.3 Casos de borda

Para cada método, cubra:

- caso comum;
- entrada vazia;
- limites mínimo e máximo;
- valor desconhecido;
- erro esperado;
- estado antes e depois da operação.

Um teste deve falhar por um motivo claro. Evite reunir muitas regras em uma única verificação.

16.4 Refatoração

Refatorar é alterar a estrutura interna sem modificar o comportamento externo. Um ciclo seguro é:

1. escrever ou executar testes;
2. fazer uma mudança pequena;
3. executar os testes novamente;
4. registrar a decisão importante.

Sinais de refatoração:

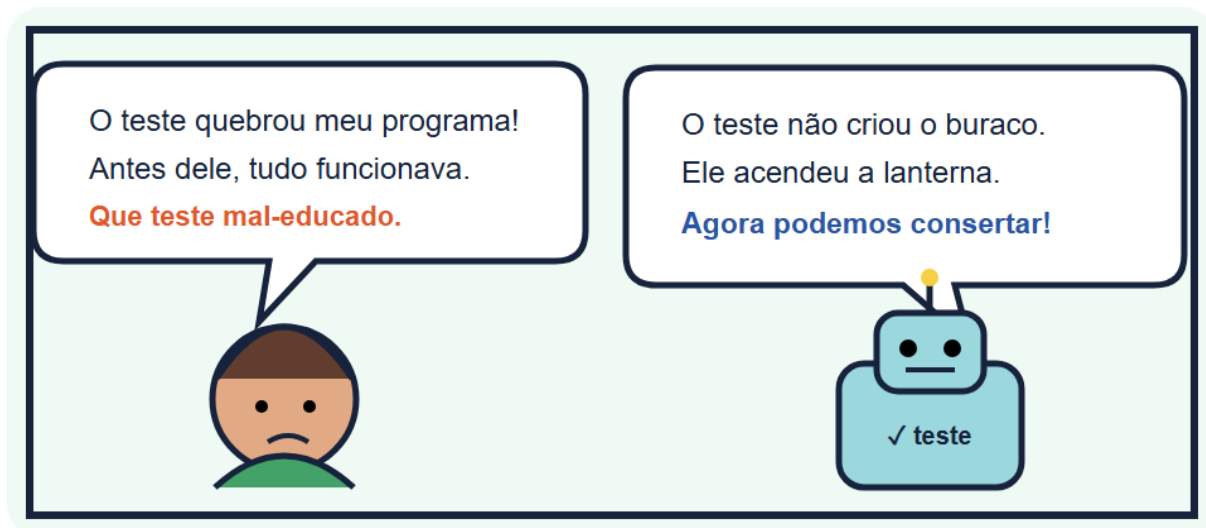


Figura 16.2: Quadrinho: o teste que encontrou um bug.

- método longo;
- cadeia extensa de `if` ou `switch`;
- regra repetida;
- classe com responsabilidades não relacionadas;
- coleção interna exposta;
- dependência direta de console em regras de negócio.

16.5 Roteiro de regressão do chatbot

Além dos testes de métodos, execute um fluxo completo:

Cenário	Entrada	Resultado esperado
ajuda	ajuda	todos os comandos aparecem
normalização	DICA	comando encontrado
vazio	linha em branco	orientação, sem histórico
histórico	oi, historico	interação anterior exibida

Cenário	Entrada	Resultado esperado
persistência	encerrar sessão	arquivo contém as linhas registradas
duplicação	registrar dois comandos iguais	exceção clara

16.6 Evolução do projeto: versão 15

Crie testes para normalização, registro de comandos e histórico antes de trocar condicionais por objetos polimórficos. Se os mesmos testes continuarem passando, a refatoração preservou o contrato.

O exemplo completo está em [examples/src/Capitulo15Testes.java](#).

16.7 Prática verificável

1. Faça um teste falhar de propósito e leia a mensagem.
2. Teste uma exceção esperada com `try/catch`.
3. Extraia uma regra duplicada e confirme que os testes permanecem verdes.
4. Escreva um roteiro manual com entrada e saída exatas para o projeto final.

Próximo passo profissional

Depois de dominar a estrutura, migre esses casos para JUnit em um projeto Maven ou Gradle. A biblioteca muda; a lógica de preparação, ação e verificação permanece.

Capítulo 17

Projeto final: TutorJavaBot

Uma entrega integrada, testável e evolutiva

17.1 Objetivo

Construir um chatbot de console que aplique os fundamentos e as decisões de design estudadas. A entrega deve funcionar, comunicar erros de forma clara e permitir a adição de comandos sem alterar o núcleo do processador.

17.2 Requisitos obrigatórios

O programa deve:

1. executar em Java 21 ou superior;
2. permanecer ativo até o comando `sair`;
3. reconhecer `oi`, `ajuda`, `dica`, `quiz`, `historico` e `sair`;
4. normalizar espaços e diferenças entre maiúsculas e minúsculas;

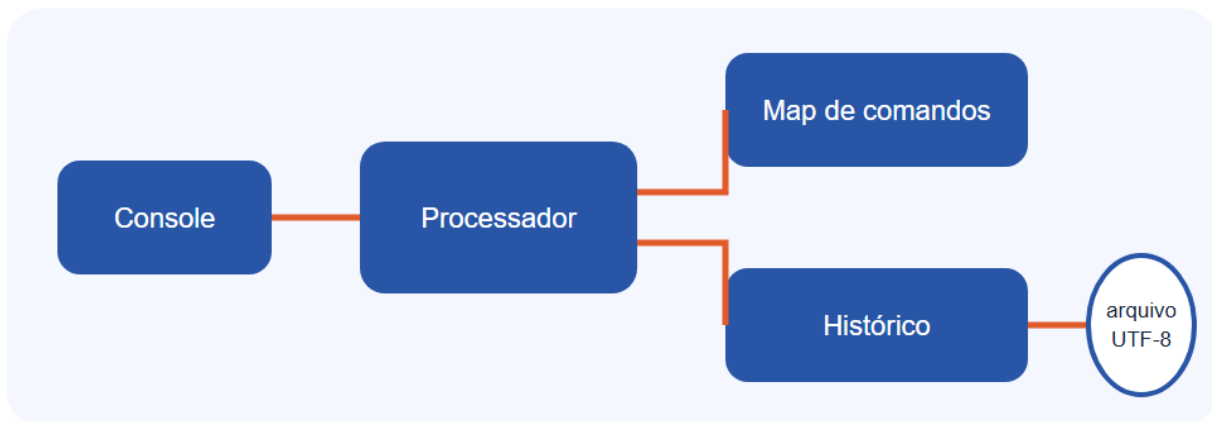


Figura 17.1: Arquitetura de referência do TutorJavaBot.

5. rejeitar entrada vazia sem registrá-la;
6. guardar interações em memória;
7. salvar o histórico em UTF-8 ao encerrar;
8. tratar falha de entrada e saída sem encerrar abruptamente;
9. impedir comandos duplicados;
10. passar pelos testes e pelo roteiro manual do capítulo anterior.

17.3 Arquitetura recomendada

```
AplicacaoConsole
├── TutorJavaBot
│   ├── RegistroComandos
│   │   └── Map<String, Comando>
│   └── Historico
│       └── List<String>
```

Responsabilidades:

- `AplicacaoConsole`: entrada, saída e ciclo de vida;
- `TutorJavaBot`: coordena processamento e histórico;
- `RegistroComandos`: registra e localiza comandos;
- `Comando`: contrato polimórfico;

- **Histórico:** registra, consulta e salva interações.

Não crie classes apenas para aumentar a quantidade de arquivos. Cada separação deve reduzir uma responsabilidade real.

17.4 Contrato dos comandos

Comando	Resposta ou efeito
oi	apresentação do bot
ajuda	lista obtida do próprio registro
dica	dica de Java
quiz	pergunta e validação de resposta
historico	últimas interações, sem registrar a consulta
sair	encerra e tenta salvar o histórico
desconhecido	orienta a usar ajuda

Defina “interação” como um par formado pela entrada processada e pela resposta. Essa definição deve ser usada no contador, no histórico e nos testes.

17.5 Roteiro de implementação

17.5.1 Etapa 1 — domínio mínimo

Implemente `Historico` e seus testes:

- rejeitar entrada vazia;
- registrar duas linhas por interação;
- devolver cópia somente para leitura;
- calcular total corretamente.

17.5.2 Etapa 2 — comandos

Implemente a interface:

```
interface Comando {  
    String nome();  
    String executar();  
}
```

Crie uma classe por comando quando houver regra própria. Registre as implementações em `Map<String, Comando>`.

17.5.3 Etapa 3 — coordenação

TutorJavaBot normaliza a entrada, consulta o registro e grava o resultado. Console e arquivos não devem aparecer dentro das implementações de comandos simples.

17.5.4 Etapa 4 — persistência

Use `Files.write` ou `Files.writeString` e propague `IOException` até a fronteira da aplicação. O console decide qual mensagem mostrar.

17.5.5 Etapa 5 — validação

Execute:

```
./scripts/validate-examples.ps1  
quarto render --to html
```

Depois, percorra os casos manuais abaixo.

17.6 Roteiro de aceitação

1. Inicie o programa e digite AJUDA.
2. Execute `oi`, `dica` e um comando desconhecido.
3. Pressione Enter sem texto.
4. Consulte `historico`.
5. Execute `quiz` com resposta correta e incorreta.
6. Digite SAIR.
7. Abra o arquivo gerado e compare-o com as interações mostradas.

A entrada vazia, a consulta ao histórico e o comando de saída não devem aumentar o total de interações.

17.7 Critérios de avaliação

Critério	Peso
comportamento e contrato dos comandos	30%
organização e encapsulamento	20%
tratamento de erros e persistência	15%
testes automatizados	15%
legibilidade e nomes	10%
experiência no console	10%

17.8 Exemplo de referência

Uma implementação completa, sem bibliotecas externas, está em [examples/src/Capitulo16Projeto.java](#). Leia-a depois de concluir sua primeira versão e compare decisões, não apenas linhas.

17.9 Evoluções opcionais

- carregar perguntas de quiz de um arquivo;
- persistir data e hora de cada interação;
- migrar os testes para JUnit;
- criar interface gráfica com JavaFX;
- expor o processador por uma API;
- integrar um modelo de linguagem atrás de uma nova implementação de Comando.

Cada evolução deve começar com um contrato e um teste. Recursos externos não devem comprometer o funcionamento básico do chatbot.

i Definição de pronto

A entrega está pronta quando compila do zero, passa pelos testes, completa o roteiro de aceitação e pode receber um novo comando sem editar o processador central.

Parte III

Apêndices

Capítulo 18

Respostas e pistas

Consulte depois de tentar resolver as atividades

Este apêndice apresenta critérios de conferência e caminhos possíveis. A solução pode usar nomes ou estruturas diferentes e ainda estar correta se respeitar o contrato proposto. Os programas completos estão em [examples/src](#).

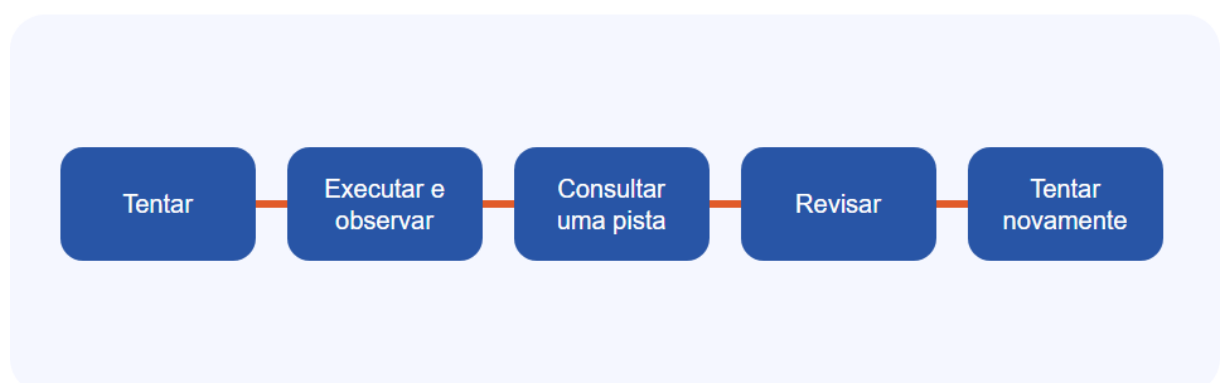


Figura 18.1: Como usar as respostas sem perder a oportunidade de praticar.

18.1 Primeiros passos

A classe pública e o arquivo devem ter o mesmo nome. A sequência correta é:

```
javac NomeDaClasse.java  
java NomeDaClasse
```

Se o compilador apontar ' ; ' expected, revise a instrução indicada e também a linha anterior.

18.2 Variáveis, tipos e entrada

Uma solução possível usa `String` para nome e cidade e `int` para horas. A carga de quatro semanas é `horasSemanais * 4`. Aplique `strip()` aos textos e leia números com `Integer.parseInt(scanner.nextLine())`.

18.3 Condicionais

A classificação deve testar os limites em ordem:

```
if (horas < 3) {  
    return "ritmo inicial";  
} else if (horas <= 6) {  
    return "ritmo consistente";  
}  
return "ritmo intensivo";
```

O valor 3 pertence ao intervalo consistente; o valor 6 também.

18.4 Repetição

Mantenha contadores separados para mensagens válidas, desconhecidas e caracteres. Atualize-os somente depois de rejeitar entrada vazia e antes de verificar o limite máximo.

18.5 Métodos

Uma implementação possível de comando conhecido:

```
static boolean ehComandoConhecido(String entrada) {  
    return switch (normalizarComando(entrada)) {  
        case "oi", "ajuda", "dica", "sair" -> true;  
        default -> false;  
    };  
}
```

O método não deve imprimir; ele apenas devolve uma decisão.

18.6 Arrays

Para o array {30, 40, 20, 50, 10}, o total é 150, a média é 30.0, o maior é 50 e o menor é 10. Antes de inicializar maior e menor com o primeiro elemento, rejeite `length == 0`.

18.7 Listas

Para manter somente os dez últimos itens:

```
historico.add(mensagem);  
if (historico.size() > 10) {
```

```
historico.remove(0);  
}
```

Para consultar as três últimas, calcule `Math.max(0, size - 3)` e devolva uma cópia do subintervalo.

18.8 Classes e objetos

`totalMensagens()` deve consultar a lista, não manter um segundo contador que possa ficar dessincronizado. O construtor rejeita `null` e `isBlank()`. `List.copyOf` protege a estrutura interna.

18.9 Exceções e arquivos

Capture `NumberFormatException` dentro do laço de leitura. Propague `IOException` no método de persistência e capture-a na aplicação de console. Assim, cada camada decide apenas o erro que sabe apresentar.

18.10 Chatbot integrado

A consulta ao histórico deve ocorrer antes do registro da interação. O comando `sair` é tratado pelo laço externo. Entradas vazias não passam para o método que registra.

18.11 Design orientado a objetos

A extração está correta quando `Historico` pode ser testado sem `Scanner` e quando `ProcessadorComandos` não conhece arquivo

ou console. O comportamento observado pela pessoa usuária deve permanecer igual.

18.12 Herança

`Aviso` extends `MensagemBot` chama `super("aviso")` e sobreescreve `texto()`. Se avisos precisarem apenas conter uma mensagem variável, composição pode ser mais simples que criar muitas subclasses.

18.13 Interfaces e polimorfismo

`ComandoSobre` implementa `Comando` e é adicionado à coleção. O método `processar` não deve ganhar outro `if` ou `case`.

18.14 Generics e coleções

Use `putIfAbsent` para detectar chave duplicada. Normaliza a chave no registro e na consulta. `List.copyOf` e `Map.copyOf` são adequados para expor visões imutáveis.

18.15 Testes e refatoração

Faça um teste falhar alterando temporariamente o valor esperado. A mensagem deve identificar descrição, esperado e obtido. Restaure o teste antes de continuar. Uma refatoração só termina quando todos os casos voltam a passar.

18.16 Projeto final

Antes de consultar a referência, confirme:

- ☐ todos os comandos obrigatórios funcionam;
- ☐ entrada vazia não entra no histórico;
- ☐ histórico e contador usam a mesma definição de interação;
- ☐ comandos duplicados são rejeitados;
- ☐ falha de arquivo é tratada;
- ☐ exemplos compilam com `--release 21`;
- ☐ um comando novo não exige alteração do processador central.

Compare sua solução com `Capitulo16Projeto.java` e registre ao menos uma decisão que você manteria e uma que mudaria.

Capítulo 19

Sobre os autores

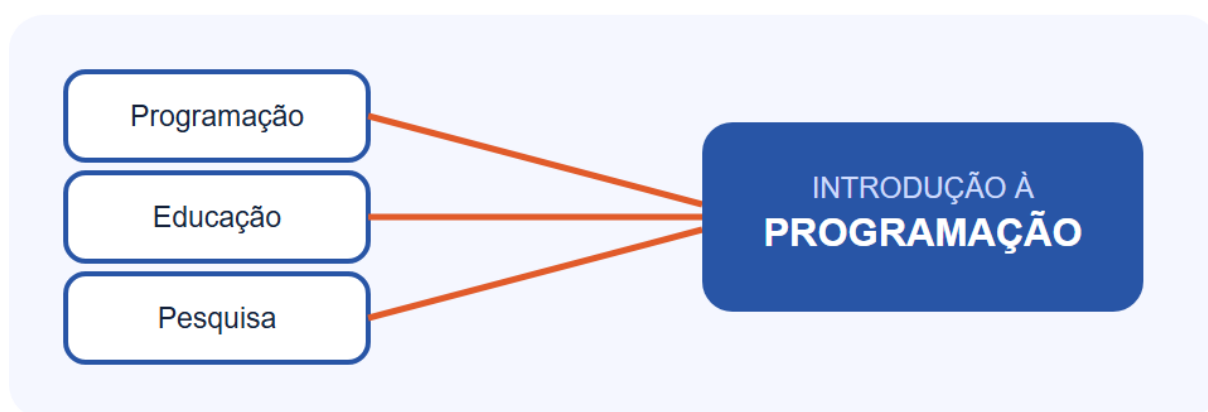


Figura 19.1: Conhecimentos reunidos na criação do livro.

Giseldo da Silva Neo

Giseldo da Silva Neo é professor de Informática no Instituto Federal de Alagoas (IFAL) e pesquisador na área de inteligência artificial. É doutor em Ciência da Computação pela Universidade Federal de Campina Grande (UFCG), mestre em Modelagem Computacional do Conhecimento pela Universidade Federal de Alagoas (UFAL) e mestre em Contabilidade pela FUCAPE.

Possui formação complementar em gestão, projetos, arquitetura e engenharia de software, além de graduações em Análise e Desenvolvimento de Sistemas e Processos Gerenciais.

Alana Viana Borges da Silva Neo

Alana Viana Borges da Silva Neo é professora de Informática no Instituto Federal de Mato Grosso do Sul (IFMS) e pesquisadora na área de Informática na Educação. É doutoranda em Ciência da Computação pela Universidade Federal de Campina Grande (UFCG) e mestra em Modelagem Computacional do Conhecimento pela Universidade Federal de Alagoas (UFAL).

Possui formação em desenvolvimento de software, segurança da informação, estratégias didáticas com tecnologias digitais, Análise e Desenvolvimento de Sistemas, Sistemas de Informação e Licenciatura em Computação.

Contato

Para comunicar erros, críticas ou sugestões, escreva para giseldo@gmail.com ou acesse <https://giseldo.github.io/>.

Aviso legal

As informações deste trabalho destinam-se a fins educacionais e informativos. Embora os autores tenham revisado o conteúdo com atenção, não oferecem garantia absoluta sobre sua aplicação em todos os ambientes ou contextos.

Os autores e editores não poderão ser responsabilizados por danos ou

perdas decorrentes do uso do material. As opiniões apresentadas são de responsabilidade dos autores e não representam necessariamente as instituições às quais estão vinculados.

Uso de inteligência artificial generativa

Ferramentas de inteligência artificial generativa auxiliaram a criação ou a revisão de partes textuais e visuais. Os autores são responsáveis pela seleção, verificação técnica e versão final do conteúdo.

