DOMINANDO JAVA EM 1 ANO

Caderno de Respostas

Gabarito comentado das 79 aulas, do #00 ao #78

Java 25 LTS · 9 fases e 3 trilhas · 189 questões · 7 desafios integradores

Autor Desconhecido · theknowledgeisle.com

Consulte este gabarito só depois de tentar.

As respostas são comentadas para ensinar, não só para conferir. Nos exercícios de código existem outras soluções igualmente válidas: o que importa é entender por que a sua funciona e o que ela troca em relação à mostrada aqui.

Todos os programas marcados com um quadro verde de **Saída** foram executados no Java 25 e a saída mostrada é a real.

Sumário

Módulo	Tema	Aulas	Questões
FASE 1	Primeiros passos na JVM	#00–#07	Q1–Q19
FASE 2	Objetos	#08–#15	Q20–Q38
FASE 3	A biblioteca padrão	#16–#22	Q39–Q58
FASE 4	Java funcional	#23–#27	Q59–Q73
FASE 5	Projetos e ferramentas	#28–#32	Q74–Q85
FASE 6	Por dentro da JVM	#33–#36	Q86–Q96
FASE 7	Concorrência	#37–#43	Q97–Q113
FASE 8	Qualidade e distribuição	#44–#46	Q114–Q121
FASE 9	O projeto final: o servidor HTTP Farol	#47–#52	Q122–Q134
TRILHA A	Web, API e bancos de dados: as Trilhas da Ilha	#53–#66	Q135–Q162
TRILHA B	Desktop e Android	#67–#70	Q163–Q170
TRILHA C	Jogos com libGDX	#71–#77	Q171–Q186
ENCERRAMENTO	O destino e o novo começo	#78	Q187–Q189
DESAFIOS	Projetos integradores que atravessam várias fases e trilhas	todas	D1–D7

Legenda: nas perguntas, `assim` indica código, nomes de classes e comandos. As linhas pautadas são para a sua resposta; nos exercícios de código, use também o computador: **todo programa deve ser compilado e executado**.

FASE 1

Primeiros passos na JVM

Artigos 00 a 07

#00 O Mapa da Jornada — o plano do curso
#01 Uma Linguagem para Torradeiras que Conquistou o Mundo — a História do Java
#02 Do Código ao Bytecode — JDK, javac, JVM e o Primeiro Programa
#03 Cada Valor em sua Caixa — Tipos Primitivos, Variáveis e Conversões
#04 Operadores, Texto e Conversa com o Usuário — Expressões, Strings e Entrada
#05 O Programa que Escolhe e Repete — if, switch e Laços
#06 Dividir para Resolver — Métodos, Parâmetros e Recursão
#07 Muitos Valores sob um Nome — Arrays

Perguntas conceituais

Q1 Origem do Java

> *Onde, quando e por quem o Java foi criado? Qual era o objetivo original do projeto, e por que o fracasso desse objetivo acabou sendo a sorte da linguagem?*

O Java nasceu em 1991 na Sun Microsystems, dentro do *Green Project*, um grupo à parte liderado por **James Gosling** (com Mike Sheridan e Patrick Naughton). O objetivo era programar aparelhos eletrônicos de consumo, como controles remotos e decodificadores de TV. A linguagem se chamava **Oak** e rodava no protótipo *7.

O mercado de TV interativa não decolou, mas a web explodiu em 1994 e 1995. As características pensadas para aparelhos (código pequeno, independente do processador, seguro para executar código recebido pela rede) eram exatamente o que a internet precisava. Rebatizada de **Java**, a linguagem foi lançada em 1995 com a promessa "*write once, run anywhere*".

Q2 JDK, JRE e JVM

> *Explique a diferença entre **JDK**, **JRE** e **JVM**. Qual deles você instala para programar, e por que hoje quase não se fala mais em "instalar o JRE"?*

- **JVM** (*Java Virtual Machine*): a máquina virtual que carrega e executa bytecode. Ela interpreta, compila com o JIT e gerencia a memória com o coletor de lixo.
- **JRE** (*Java Runtime Environment*): a JVM mais a biblioteca padrão. É o mínimo para **rodar** programas.
- **JDK** (*Java Development Kit*): o JRE mais as ferramentas de desenvolvimento (`javac`, `jar`, `javadoc`, `jshell`, `jlink`, `jpackage`, `jfr`...). É o que se instala para **programar**.

A partir do Java 11 a Oracle deixou de distribuir um JRE separado. Alguns fornecedores ainda publicam pacotes "JRE" (o Adoptium tem imagens `eclipse-temurin:25-jre`, usadas no Docker da Trilha A), mas a recomendação moderna é que cada aplicação leve o próprio runtime, montado com `jlink` só com os módulos de que precisa.

Q3 Bytecode

> *O que é **bytecode**? Por que ele permite que o mesmo `.class` rode em Windows, Linux e macOS? Que ferramenta do JDK mostra o bytecode de uma classe?*

Bytecode é o conjunto de instruções de uma máquina **virtual**, e não de um processador real. O `javac` traduz o código-fonte para bytecode (instruções como `iload`, `iadd`, `invokevirtual`) e o grava no arquivo `.class`. Quem executa esse arquivo é a JVM, e existe uma JVM para cada sistema e processador. O programa é compilado uma vez e cada JVM cuida da tradução para a máquina em que está.

O comando `javap -c NomeDaClasse` desmonta o `.class` e mostra as instruções.

Q4 Compilar ou lançar o código-fonte

> *Qual é a diferença entre rodar um programa com `javac Ola.java` seguido de `java Ola`, e rodar diretamente com `java Ola.java`? Em que situação cada forma é a mais adequada?*

Com `javac Ola.java` o compilador grava `Ola.class` no disco, e o `java Ola` executa essa classe já compilada. É o caminho de projetos com vários arquivos, bibliotecas e empacotamento.

Com `java Ola.java` (o "lançador de código-fonte", desde o Java 11) a JVM compila o arquivo **em memória** e executa na hora, sem gerar `.class`. É ideal para scripts, estudos e programas pequenos. Desde o Java 22 esse modo também aceita programas com vários arquivos-fonte na mesma pasta.

Q5 Aquecimento e JIT

> *O que significa dizer que um programa Java "aquece"? Relacione a resposta com o interpretador e o compilador JIT.*

A JVM começa **interpretando** o bytecode, instrução por instrução, e conta quantas vezes cada método e cada laço é executado. Quando um trecho fica "quente", o compilador **JIT** (*just-in-time*) o traduz para código de máquina otimizado, primeiro com o C1 (rápido de compilar) e depois com o C2 (mais agressivo). Por isso as primeiras execuções de um trecho são lentas e as seguintes ficam muito mais rápidas. Esse período inicial é o aquecimento, e é o motivo de *benchmarks* sérios descartarem as primeiras rodadas.

Q6 Os oito primitivos

> *Liste os oito tipos primitivos do Java com o tamanho de cada um. Qual deles não tem um tamanho fixo definido pela especificação da linguagem?*

- `byte`: 8 bits; `short`: 16 bits; `int`: 32 bits; `long`: 64 bits (inteiros com sinal, em complemento de dois);
- `float`: 32 bits; `double`: 64 bits (ponto flutuante IEEE 754);
- `char`: 16 bits (uma unidade de código UTF-16, sem sinal);
- `boolean`: `true` ou `false`.

O `boolean` é o único sem tamanho fixado pela especificação. Na prática, a HotSpot usa 1 byte para um campo `boolean` e para cada elemento de um `boolean[]`.

Q7 O uso de var

> *Em que situações a palavra `var` pode ser usada? Por que `var x;` não compila, e qual é o tipo inferido em `var lista = new ArrayList<>();`?*

`var` só vale para **variáveis locais** com inicializador: dentro de métodos, no `for`, no `for-each`, no `try` com recursos e nos parâmetros de *lambdas*. Não vale para campos, parâmetros de métodos nem tipos de retorno.

`var x;` não compila porque o tipo é inferido **do valor inicial**, e sem valor não há de onde inferir. `var x = null;` também não compila, pelo mesmo motivo.

Em `var lista = new ArrayList<>();` o operador diamante não tem nenhuma pista de tipo, e o resultado é `ArrayList<Object>`. Com `var`, escreva o tipo dos elementos do lado direito: `var lista = new ArrayList<String>();`.

Q8 Strings: == ou equals

> *Por que comparar duas `String`s com `==` é um erro, mesmo quando às vezes "funciona"? Dê um exemplo em que `==` devolve `false` para textos iguais.*

`==` compara **referências**: diz se as duas variáveis apontam para o mesmo objeto. `equals` compara o **conteúdo**. Literais iguais no código-fonte são guardados uma só vez no *pool* de strings, e por isso `"java" == "java"` dá `true`. É daí que vem a impressão de que funciona.

Uma string construída em tempo de execução é um objeto novo:

```
String a = "java";
String b = new Scanner(System.in).next();   // o usuário digita: java
IO.println(a == b);        // false
IO.println(a.equals(b));   // true
```

Regra: textos se comparam com `equals` (ou `equalsIgnoreCase`).

Exercícios de código — soluções

Q9 O que o programa imprime

> *Sem executar, diga o que o programa imprime. Depois execute e confira.*

```
void main() {
    int grande = Integer.MAX_VALUE;
    grande++;
    IO.println(grande);
    IO.println((byte) 200);
    IO.println(0.1 + 0.2);
    IO.println((int) 3.99);
    IO.println('a' + 1);
    IO.println((char) ('a' + 1));
    IO.println(7 / 2 + " " + 7 % 2 + " " + -7 / 2 + " " + -7 % 2);
}
```

Saída

```
-2147483648
-56
0.30000000000000004
3
98
b
3 1 -3 -1
```

- `int` que passa do máximo **dá a volta** (overflow) para o mínimo, sem erro nem aviso.
- 200 não cabe num `byte` (−128 a 127): sobram os 8 bits `11001000`, que em complemento de dois valem −56.
- 0,1 e 0,2 não têm representação exata em binário. Para dinheiro, use `BigDecimal`.
- A conversão de `double` para `int` **trunca** (não arredonda).
- `char` somado com `int` vira `int` (o código de 'a' é 97). Com o *cast*, volta a ser um caractere.
- A divisão inteira trunca em direção a zero, e o resto tem o sinal do dividendo. Para um resto sempre positivo, use `Math.floorMod(-7, 2)`, que dá 1.

Q10 Collatz de 27

Escreva um programa que conte quantos passos a sequência de Collatz leva para chegar a 1 a partir de 27. Regra: se `n` é par, vira `n / 2`; se é ímpar, vira `3n + 1`. Use um `long` e explique por quê.

```
void main() {
    long n = 27;              // long: os valores intermediários passam de 9 mil
    int passos = 0;           // (e, para outros inícios, podem estourar um int)
    long maior = n;
    while (n != 1) {
        n = (n % 2 == 0) ? n / 2 : 3 * n + 1;
        maior = Math.max(maior, n);
        passos++;
    }
    IO.println("passos: " + passos + ", maior valor: " + maior);
}
```

Saída

```
passos: 111, maior valor: 9232
```

Com 27 um `int` bastaria. Mas a sequência pode subir muito antes de descer, e para alguns inícios na casa dos bilhões os valores passam de `Integer.MAX_VALUE`. Com `long`, o laço está protegido para qualquer início razoável.

Q11 Switch expression

> *Reescreva o `switch` abaixo como uma **switch expression** (com `->`), sem `break`, e garantindo que o compilador reclame se um valor ficar de fora.*

```
enum Dia { SEG, TER, QUA, QUI, SEX, SAB, DOM }
```

```
String tipo(Dia dia) {
    return switch (dia) {
        case SAB, DOM -> "fim de semana";
        case SEG, TER, QUA, QUI, SEX -> "dia útil";
    };                                  // sem default: se surgir um novo Dia, não compila
}
```

```
void main() {
    IO.println(tipo(Dia.DOM) + " / " + tipo(Dia.QUA));
}
```

Saída

```
fim de semana / dia útil
```

Sem `default`, a expressão precisa cobrir todas as constantes do `enum`, e o compilador confere. Se alguém acrescentar um valor ao `enum`, o código deixa de compilar e aponta exatamente o lugar a corrigir. Com `default`, o valor novo cairia em "dia útil" em silêncio.

Q12 Break com rótulo

> *Use um `break` com **rótulo** para encontrar a primeira posição (linha, coluna) de um valor negativo numa matriz, saindo dos dois laços de uma vez.*

```
void main() {
    int[][] m = {{3, 1, 4}, {1, -5, 9}, {-2, 6, 5}};
    int linha = -1, coluna = -1;
    busca:
    for (int i = 0; i < m.length; i++) {
        for (int j = 0; j < m[i].length; j++) {
            if (m[i][j] < 0) {
                linha = i;
                coluna = j;
                break busca;            // sai dos DOIS laços
            }
        }
    }
    IO.println("primeiro negativo em [" + linha + "][" + coluna + "]");
}
```

Saída

```
primeiro negativo em [1][1]
```

Sem o rótulo, o `break` sairia só do laço de dentro, e o de fora continuaria até achar o −2 na linha 2. Muitas vezes a alternativa mais limpa é extrair a busca para um método e usar `return`.

Q13 Passagem por valor

> *O que o programa imprime? Explique com a ideia de que **o Java sempre passa parâmetros por valor**.*

```
void trocar(int[] v) { v = new int[]{9, 9, 9}; }
void mexer(int[] v) { v[0] = 42; }
void dobrar(int x) { x = x * 2; }
```

```
void main() {
    int[] a = {1, 2, 3};
    int n = 5;
    trocar(a);
    mexer(a);
    dobrar(n);
    IO.println(java.util.Arrays.toString(a) + " " + n);
}
```

Saída

```
[42, 2, 3] 5
```

O Java copia o **valor** de cada argumento. Para `int`, o valor é o número. Para um array (ou qualquer objeto), o valor é a **referência**.

- `trocar` recebe uma cópia da referência e aponta a cópia para outro array. A variável `a` do chamador não muda.
- `mexer` usa a cópia da referência para alterar o **objeto** compartilhado, e a mudança aparece para quem chamou.
- `dobrar` altera só a cópia local de `n`.

Q14 Resolução de sobrecarga

> *Qual versão de `f` é chamada em cada linha? Justifique pela ordem em que o compilador procura as sobrecargas.*

```
void f(long x)     { IO.println("long"); }
void f(Integer x)  { IO.println("Integer"); }
void f(int... x)   { IO.println("varargs"); }
```

```
void main() {
    f(5);
    f(5L);
    f(Integer.valueOf(5));
    f();
}
```

Saída
```
long
long
Integer
varargs
```

O compilador procura em três etapas e para na primeira que encontra candidato:

1. sem *boxing* e sem varargs, aceitando **alargamento** de primitivos (`int` → `long`);

2. com *boxing*/*unboxing*;

3. com varargs.

Por isso `f(5)` escolhe `f(long)`: o alargamento vence o *boxing* para `Integer`. O varargs é sempre o último recurso, e só `f()` precisou dele.

Q15 MDC recursivo

> *Escreva a versão **recursiva** do máximo divisor comum pelo algoritmo de Euclides (`mdc(a, b) = mdc(b, a % b)`, com `mdc(a, 0) = a`). Depois explique o que causa um `StackOverflowError` numa recursão.*

```
long mdc(long a, long b) {
    return b == 0 ? a : mdc(b, a % b);   // caso base: o resto zerou
}
```

```
void main() {
    IO.println(mdc(1071, 462) + " " + mdc(17, 5) + " " + mdc(0, 9));
}
```

Saída
```
21 1 9
```

Cada chamada ocupa um **quadro** na pilha da *thread* (parâmetros, variáveis locais e endereço de retorno), e o quadro só é liberado quando a chamada termina. Uma recursão sem caso base, ou com profundidade grande demais (centenas de milhares de níveis), esgota a pilha, que tem tamanho fixo (por volta de 1 MB por padrão), e a JVM lança `StackOverflowError`. O Euclides é seguro: o número de passos cresce com o logaritmo dos valores. O Java não faz otimização de chamada de cauda, então recursões profundas devem virar laços.

Q16 Inverter no lugar

*Escreva um método `void inverter(int[] v)` que inverta o array **no próprio lugar**, sem criar outro array. Mostre o resultado para `{1, 2, 3, 4, 5}` e para um array vazio.*

```
void inverter(int[] v) {
    for (int i = 0, j = v.length - 1; i < j; i++, j--) {   // dois índices se encontrando no meio
        int t = v[i];
        v[i] = v[j];
        v[j] = t;
    }
}
```

```
void main() {
    int[] a = {1, 2, 3, 4, 5};
    int[] vazio = {};
    inverter(a);
    inverter(vazio);                                        // i < j é falso de cara: nada
acontece
    IO.println(java.util.Arrays.toString(a) + " " + java.util.Arrays.toString(vazio));
}
```

Saída

```
[5, 4, 3, 2, 1] []
```

O laço vai até o meio, trocando as pontas. Num array de tamanho ímpar, o elemento central fica no lugar. O método muda o array do chamador porque recebe uma cópia da **referência**, como na questão sobre passagem por valor.

Q17 Imprimir e copiar arrays

Por que `IO.println(new int[]{1, 2})` imprime algo como `[I@1b6d3586` em vez dos números? Como imprimir um array corretamente? E como copiar um array de modo que alterar a cópia não altere o original?

Arrays não sobrescrevem `toString()`. Eles herdam o de `Object`, que mostra o tipo (`[I` = array de `int`), uma `@` e o *hash* do objeto em hexadecimal. Para os elementos, use `Arrays.toString(v)`, ou `Arrays.deepToString(m)` para matrizes.

`int[] b = a;` **não** copia: as duas variáveis passam a apontar para o mesmo array (*aliasing*). Para copiar, use `int[] b = a.clone();`, `Arrays.copyOf(a, a.length)` ou `Arrays.copyOfRange`. Numa matriz, essas cópias são **rasas**: copiam as referências das linhas, e é preciso copiar linha por linha para ter independência total.

Q18 Triângulo de Pascal

*Uma **matriz irregular** (jagged) tem linhas de tamanhos diferentes. Monte o triângulo de Pascal com 5 linhas usando `int[][]`, em que a linha `i` tem `i + 1` elementos, e imprima-o.*

```
void main() {
    int n = 5;
    int[][] t = new int[n][];                // só o "array de linhas"; cada linha é criada
depois
    for (int i = 0; i < n; i++) {
        t[i] = new int[i + 1];
        t[i][0] = t[i][i] = 1;               // as bordas valem 1
        for (int j = 1; j < i; j++) t[i][j] = t[i - 1][j - 1] + t[i - 1][j];
    }
    for (int[] linha : t) IO.println(java.util.Arrays.toString(linha));
}
```

Saída

```
[1]
[1, 1]
[1, 2, 1]
[1, 3, 3, 1]
[1, 4, 6, 4, 1]
```

Em Java, uma "matriz" é um array de arrays. `new int[n][]` cria só o array externo, com `n` referências nulas, e cada linha é alocada com o tamanho que precisar.

Q19 Leitura com Scanner

> *Escreva um programa que leia números inteiros digitados pelo usuário até ele digitar `fim`, e mostre a quantidade, a soma e a média. Trate a digitação de algo que não é número sem derrubar o programa.*

```
import java.util.Scanner;
```

```
void main() {
    var entrada = new Scanner(System.in);
    int quantos = 0;
    long soma = 0;
    while (true) {
        IO.print("número (ou fim): ");
        if (!entrada.hasNext()) break;              // fim da entrada (Ctrl+D / Ctrl+Z)
        String palavra = entrada.next();
        if (palavra.equalsIgnoreCase("fim")) break;
        try {
            soma += Integer.parseInt(palavra);
            quantos++;
        } catch (NumberFormatException e) {
            IO.println("\"" + palavra + "\" não é um número inteiro; ignorado.");
        }
    }
    if (quantos == 0) IO.println("nenhum número lido.");
    else IO.println("quantidade: %d, soma: %d, média: %.2f".formatted(quantos, soma, (double)
soma / quantos));
}
```

Ler cada item como texto e converter com `Integer.parseInt` dá controle sobre o erro. Com `nextInt()`, uma entrada inválida lança `InputMismatchException` e **não consome** o *token* ruim, e um laço ingênuo fica preso nele para sempre. O *cast* para `double` antes da divisão evita a divisão inteira.

FASE 2

Objetos

Artigos 08 a 15

#08 Moldes e Instâncias — Classes, Objetos e Construtores
#09 O que Fica Dentro, o que Fica Fora — Encapsulamento e Modificadores de Acesso
#10 Do Geral ao Específico — Herança e a Classe Object
#11 Contratos sem Implementação — Interfaces e Classes Abstratas
#12 Um Tipo, Várias Formas — Polimorfismo na Prática
#13 Iguais ou o Mesmo? — equals, hashCode e Identidade
#14 Dados sem Cerimônia — Records e Enums de Verdade
#15 Hierarquias Fechadas — Sealed Classes e Pattern Matching

Perguntas conceituais

Q20 Classe, objeto e memória

> *Qual é a diferença entre uma **classe** e um **objeto**? Onde ficam, na memória da JVM, o objeto criado com `new` e a variável local que aponta para ele?*

A classe é o **molde**: declara os campos que cada objeto terá e os métodos que operam sobre eles. O objeto é uma **instância** concreta, com valores próprios nos campos. De uma classe `ContaBancaria` saem quantas contas forem necessárias.

O objeto criado com `new` vive no **heap**, a área compartilhada que o coletor de lixo gerencia. A variável local que guarda a referência vive no **quadro da pilha** do método em execução, e some quando o método termina. O objeto continua no heap enquanto alguém o alcançar.

Q21 Modificadores de acesso

> *Quais são os quatro níveis de acesso do Java, do mais aberto ao mais fechado? Qual é o nível que se aplica quando nenhum modificador é escrito?*

1. `public`: visível em qualquer lugar (desde que o módulo exporte o pacote);
2. `protected`: visível no mesmo pacote e nas subclasses, mesmo em outros pacotes;
3. **sem modificador** (*package-private*): visível só dentro do mesmo pacote. É o padrão;
4. `private`: visível só dentro da própria classe (incluindo as classes aninhadas nela).

Uma boa regra de partida é tudo `private` e abrir só o necessário: cada item público é uma promessa que você terá de manter.

Q22 Por que encapsular

> *Por que **encapsular** um campo (deixá-lo `private` e expor métodos) é melhor do que deixá-lo `public`, se no fim o valor continua podendo ser lido? Use uma conta bancária como exemplo.*

Encapsular não é esconder o valor: é garantir as **invariantes**. Com `public double saldo`, qualquer código pode escrever `conta.saldo = -1000` e a regra "saque só com saldo suficiente" deixa de existir. Com `private` e um método `sacar(valor)`, que valida o valor e o saldo antes de mudar o campo, **a única porta de entrada** verifica a regra, e não existe outro jeito de deixar a conta num estado inválido.

O encapsulamento também permite mudar a representação interna (por exemplo, trocar `double` por `BigDecimal`, ou guardar o extrato e calcular o saldo) sem quebrar quem usa a classe.

Q23 Ordem de construção

> *Em que ordem acontecem as etapas quando se executa `new ContaCorrente(...)`, sendo `ContaCorrente` subclasse de `ContaBancaria`? Por que a chamada `super(...)` precisa ser a primeira instrução do construtor (até o Java 24)?*

1. A JVM reserva a memória do objeto inteiro e zera **todos** os campos (0, `false`, `null`).

2. O construtor de `ContaCorrente` chama `super(...)`, explícito ou implícito.

3. O construtor de `ContaBancaria` roda por inteiro: primeiro o `super()` de `Object`, depois os inicializadores de campo e os blocos de inicialização de `ContaBancaria`, e depois o corpo do construtor dela.

4. Voltando a `ContaCorrente`: rodam os inicializadores de campo dela e, por fim, o resto do corpo do construtor.

A regra existia para que a subclasse nunca usasse uma parte do objeto que a superclasse ainda não inicializou. O Java 25 finalizou os **corpos de construtor flexíveis** (JEP 513): agora é permitido escrever instruções **antes** do `super(...)`, como validar argumentos, desde que elas não usem `this`.

Q24 Método sobrescrito no construtor

> *Qual é o perigo de chamar um método que pode ser **sobrescrito** de dentro de um construtor? Dê um exemplo em que o resultado surpreende.*

O construtor da superclasse roda **antes** de os campos da subclasse serem inicializados. Se ele chamar um método que a subclasse sobrescreveu, é a versão da subclasse que executa, e ela enxerga os próprios campos ainda zerados.

```
class Base {
    Base() { descrever(); }                 // chama um método sobrescrevível
    void descrever() { IO.println("base"); }
}
```

```
class Filha extends Base {
    private String nome = "filha";          // inicializado na declaração...
    @Override void descrever() { IO.println("nome = " + nome); }
}
```

```
void main() {
    new Filha();
}
```

Saída
```
nome = null
```

Mesmo inicializado na declaração, o campo `nome` ainda vale `null` nesse momento, porque os inicializadores da subclasse só rodam **depois** que o construtor de `Base` termina. (Curiosidade: se o campo fosse `final` e recebesse um literal, o compilador o trataria como **constante** e embutiria "filha" no código. O programa imprimiria "filha", uma exceção à regra que só confunde mais.) A regra (item de *Effective Java*): construtores não devem chamar métodos sobrescrevíveis. Esses métodos devem ser `private`, `final` ou `static`.

Q25 Classe abstrata ou interface

> *Quando escolher uma **classe abstrata** e quando escolher uma **interface**? Cite duas coisas que só a classe abstrata pode ter.*

Interface é um **contrato**: diz o que um tipo sabe fazer. Uma classe pode implementar várias interfaces, e tipos sem nenhuma relação de herança podem compartilhar a mesma (`Comparable`, `AutoCloseable`). Desde o Java 8 ela pode ter métodos `default` e `static`, e desde o 9, `private`.

A classe abstrata é uma **implementação parcial** para uma família de tipos, e só pode ser herdada uma vez. Só ela pode ter:

- **estado de instância** (campos não estáticos);
- **construtores**, e com eles invariantes garantidas na criação.

Regra prática: comece pela interface. Use a classe abstrata quando as subclasses precisarem compartilhar estado ou um esqueleto de algoritmo (o padrão *template method*).

Q26 Conflito de métodos default

> *Uma classe implementa duas interfaces que têm o mesmo método `default void saudar()`. O que o compilador faz? Como resolver chamando a versão de uma delas?*

O compilador **recusa** a classe: há duas implementações herdadas com a mesma assinatura e nenhuma regra para escolher uma. A classe precisa sobrescrever o método, e pode delegar a uma das versões com a sintaxe `Interface.super.metodo()`:

```
interface Ingles { default String saudar() { return "hello"; } }
interface Portugues { default String saudar() { return "olá"; } }
```

```
class Bilingue implements Ingles, Portugues {
    @Override public String saudar() {
        return Portugues.super.saudar() + " / " + Ingles.super.saudar();
    }
}
```

```
void main() {
    IO.println(new Bilingue().saudar());
}
```

Saída
```
olá / hello
```

Q27 Polimorfismo

> *O que é **polimorfismo**? Explique a diferença entre o tipo **declarado** de uma variável e o tipo **real** do objeto, e qual deles decide o método executado.*

Polimorfismo é poder tratar objetos de tipos diferentes através de um tipo comum, com cada um respondendo do seu jeito. Em `Forma f = new Circulo(2);`, o tipo declarado é `Forma`, e o real é `Circulo`.

- O **tipo declarado** decide o que o **compilador** permite: só métodos de `Forma` podem ser chamados por `f`.
- O **tipo real** decide o que **executa**: `f.area()` roda o `area()` de `Circulo`. A escolha acontece em tempo de execução (*despacho dinâmico*).

Os campos e os métodos `static` **não** são polimórficos: são escolhidos pelo tipo declarado.

Q28 Contrato equals/hashCode

> *Quais são as regras do contrato entre `equals` e `hashCode`? O que acontece com um `HashSet` se uma classe sobrescreve `equals` e esquece o `hashCode`?*

`equals` precisa ser reflexivo, simétrico, transitivo, consistente e dar `false` para `null`. E a regra que liga os dois métodos: **objetos iguais por `equals` precisam ter o mesmo `hashCode`**. O inverso não é exigido, e colisões são permitidas.

Um `HashSet` primeiro usa o `hashCode` para escolher o "balde" e só depois compara com `equals` dentro dele. Sem `hashCode` sobrescrito, dois pontos iguais têm *hashes* diferentes (o de `Object` vem da identidade), caem em baldes diferentes, e o conjunto aceita os dois:

```
class Ponto {
    final int x, y;
    Ponto(int x, int y) { this.x = x; this.y = y; }
    @Override public boolean equals(Object o) {
        return o instanceof Ponto p && p.x == x && p.y == y;
    }
    // falta hashCode!
}
```

```
void main() {
    var pontos = new HashSet<Ponto>();
    pontos.add(new Ponto(1, 2));
    pontos.add(new Ponto(1, 2));
    IO.println(pontos.size() + " " + pontos.contains(new Ponto(1, 2)));
}
```

Saída
```
2 false
```

(O resultado depende dos *hashes* de identidade. Numa execução muito azarada eles podem cair no mesmo balde, e é exatamente esse comportamento imprevisível que torna o *bug* traiçoeiro.)

Q29 Chave mutável

> *Por que é perigoso usar um objeto **mutável** como chave de um `HashMap` e depois alterar um campo que entra no `hashCode`?*

O mapa guarda a entrada no balde calculado pelo `hashCode` **no momento do** `put`. Se o campo mudar depois, o *hash* atual aponta para outro balde, e o `get` com a mesma chave (ou com uma igual) procura no lugar errado e devolve `null`. A entrada continua lá dentro, inacessível, ocupando memória. Chaves de mapas e elementos de conjuntos devem ser **imutáveis**, e é por isso que `String`, `Integer` e *records* com campos imutáveis são ótimas chaves.

Q30 O que um record gera

> *O que o compilador gera automaticamente num `record`? E como validar os dados na criação, por exemplo impedindo um `Intervalo` com início maior que o fim?*

Para `record Intervalo(int inicio, int fim) {}` o compilador gera: os campos `private final`, o construtor canônico, os acessores `inicio()` e `fim()` (sem o prefixo `get`), e `equals`, `hashCode` e `toString` baseados em **todos** os componentes. A classe é implicitamente `final`.

A validação vai no **construtor compacto**, que roda antes da atribuição dos campos:

```
record Intervalo(int inicio, int fim) {
    Intervalo {                                          // construtor compacto: sem lista de
parâmetros
        if (inicio > fim) throw new IllegalArgumentException("início %d > fim
%d".formatted(inicio, fim));
    }
    int tamanho() { return fim - inicio; }
}
```

```
void main() {
    IO.println(new Intervalo(2, 7) + " tamanho " + new Intervalo(2, 7).tamanho());
    try {
        new Intervalo(9, 1);
    } catch (IllegalArgumentException e) {
        IO.println("recusado: " + e.getMessage());
    }
}
```

Saída

```
Intervalo[inicio=2, fim=7] tamanho 5
recusado: início 9 > fim 1
```

Q31 Imutabilidade rasa

> *Um `record` com um componente `List<String> itens` é realmente imutável? O que precisa ser feito para que seja?*

Não. O `record` garante que o campo não será **reatribuído** (é `final`), mas a lista em si continua mutável: quem criou o *record* guarda a mesma referência e pode fazer `lista.add(...)` depois, e o acessor `itens()` devolve o objeto interno. A imutabilidade do *record* é **rasa**.

A solução é fazer uma cópia defensiva e imutável no construtor compacto: `itens = List.copyOf(itens);`. O `List.copyOf` também recusa elementos `null` e não copia de novo se a lista já for imutável.

Q32 Enums de verdade

> *Cite três coisas que um `enum` do Java pode ter e que as constantes inteiras (`static final int`) não podem. Para que serve o `EnumMap`?*

- **Segurança de tipo**: um método que recebe `StatusPedido` não aceita um inteiro qualquer.
- **Campos, construtores e métodos**, e até corpo próprio por constante (como `Op.SOMA` implementando `aplicar` do seu jeito).
- Nome legível no `toString()`, `values()` e `valueOf()`, e suporte a `switch` exaustivo.

O `EnumMap` é um mapa cujas chaves são constantes de um `enum`. Por dentro é só um array indexado pelo `ordinal()`, e por isso é mais rápido e compacto que um `HashMap` e percorre as chaves na ordem de declaração.

Q33 Tipos selados

> *O que uma interface `sealed` com `permits` garante? Por que ela combina tão bem com `switch` e record patterns?*

Uma interface (ou classe) `sealed` declara a **lista fechada** de subtipos permitidos. Nenhum outro código pode implementá-la, e cada subtipo precisa ser `final`, `sealed` ou `non-sealed`. Assim, o compilador conhece todas as variantes possíveis.

Isso permite um `switch` **exaustivo sem** `default`: se todos os subtipos têm um `case`, o compilador aceita, e se um subtipo novo for acrescentado, todo `switch` que o esquece deixa de compilar. Com *records* como subtipos, os *record patterns* desmontam os dados direto no `case`: `case Circulo(double r) -> Math.PI * r * r`.

Exercícios de código — soluções

Q34 Expressões com tipos selados

> *Modele expressões aritméticas com uma interface selada `Expr` e os records `Num`, `Soma` e `Mult`. Escreva `avaliar(Expr)` com um `switch` sobre record patterns, e `mostrar(Expr)` que produza texto com parênteses. Teste com (2 + 3) × 4.*

```
sealed interface Expr permits Num, Soma, Mult {}
record Num(int valor) implements Expr {}
record Soma(Expr esq, Expr dir) implements Expr {}
record Mult(Expr esq, Expr dir) implements Expr {}
```

```
int avaliar(Expr e) {
    return switch (e) {                                  // exaustivo: sem default
        case Num(int v) -> v;
        case Soma(Expr a, Expr b) -> avaliar(a) + avaliar(b);
        case Mult(Expr a, Expr b) -> avaliar(a) * avaliar(b);
    };
}
```

```
String mostrar(Expr e) {
    return switch (e) {
        case Num(int v) -> String.valueOf(v);
        case Soma(Expr a, Expr b) -> "(" + mostrar(a) + " + " + mostrar(b) + ")";
        case Mult(Expr a, Expr b) -> mostrar(a) + " × " + mostrar(b);
    };
}
```

```
void main() {
    Expr e = new Mult(new Soma(new Num(2), new Num(3)), new Num(4));
    IO.println(mostrar(e) + " = " + avaliar(e));
}
```

Saída

```
(2 + 3) × 4 = 20
```

Os dois métodos percorrem a árvore da expressão recursivamente. Acrescentar um *record* `Sub` à cláusula `permits` faz os dois `switch` pararem de compilar até ganharem o `case` novo, e é exatamente essa garantia que se quer.

Q35 Padrões com guarda

*Use um `switch` com **guarda** (`when`) para classificar um valor `Object`: `Integer` negativo, `Integer` não negativo, `String` vazia, `String` com texto, e "outro". O que acontece se o `case Integer i` sem guarda vier **antes** do `case Integer i when i < 0`?*

```
String classificar(Object o) {
    return switch (o) {
        case Integer i when i < 0 -> "inteiro negativo";
        case Integer i -> "inteiro não negativo";
        case String s when s.isBlank() -> "texto vazio";
        case String s -> "texto: " + s;
        case null -> "nulo";
        default -> "outro: " + o.getClass().getSimpleName();
    };
}
```

```
void main() {
    for (Object o : new Object[]{-3, 8, "", "java", 2.5, null}) IO.println(classificar(o));
}
```

Saída

```
inteiro negativo
inteiro não negativo
texto vazio
texto: java
outro: Double
nulo
```

Os `case` são testados em ordem. Se `case Integer i` viesse primeiro, ele capturaria todos os inteiros e o `case` com guarda nunca seria alcançado. O compilador detecta isso e recusa o código com o erro "this case label is dominated by a preceding case label". Sem o `case null`, um `switch` sobre `null` lança `NullPointerException`.

Q36 Enum com comportamento

Escreva um `enum Operacao` com as constantes `SOMA`, `SUB`, `MULT` e `DIV`, cada uma com um símbolo (campo) e implementando o método `double aplicar(double a, double b)`. Imprima uma tabela aplicando todas a `8` e `2`.

```
enum Operacao {
    SOMA("+") { double aplicar(double a, double b) { return a + b; } },
    SUB("-")  { double aplicar(double a, double b) { return a - b; } },
    MULT("*") { double aplicar(double a, double b) { return a * b; } },
    DIV("/")  { double aplicar(double a, double b) { return a / b; } };
```

```
    private final String simbolo;
    Operacao(String simbolo) { this.simbolo = simbolo; }
    String simbolo() { return simbolo; }
    abstract double aplicar(double a, double b);         // cada constante é obrigada a
implementar
}
```

```
void main() {
    for (Operacao op : Operacao.values())
        IO.println("8 %s 2 = %.1f".formatted(op.simbolo(), op.aplicar(8, 2)));
}
```

Saída

```
8 + 2 = 10.0
8 - 2 = 6.0
8 * 2 = 16.0
8 / 2 = 4.0
```

O método `abstract` obriga cada constante a trazer seu corpo. Acrescentar `POT("^")` sem implementar `aplicar` não compila. Uma alternativa igualmente boa é guardar um `DoubleBinaryOperator` como campo.

Q37 Simetria quebrada

*Encontre e explique o erro de projeto: a classe abaixo quebra a **simetria** do `equals`. Mostre um caso em que `a.equals(b)` e `b.equals(a)` discordam.*

```
class Ponto {
    final int x, y;
    Ponto(int x, int y) { this.x = x; this.y = y; }
    @Override public boolean equals(Object o) {
        return o instanceof Ponto p && p.x == x && p.y == y;
    }
    @Override public int hashCode() { return 31 * x + y; }
}
```

```
class PontoColorido extends Ponto {
    final String cor;
    PontoColorido(int x, int y, String cor) { super(x, y); this.cor = cor; }
    @Override public boolean equals(Object o) {
        return o instanceof PontoColorido p && super.equals(p) && p.cor.equals(cor);
    }
}
```

```
void main() {
    Ponto a = new Ponto(1, 2);
    Ponto b = new PontoColorido(1, 2, "azul");
    IO.println(a.equals(b) + " " + b.equals(a));
}
```

Saída

```
true false
```

Um `Ponto` aceita qualquer `Ponto` (inclusive o colorido) e ignora a cor. Um `PontoColorido` só aceita outro colorido. Não existe um jeito de estender uma classe instanciável, acrescentar um campo que participa do `equals` e manter o contrato (*Effective Java*, item 10). As saídas são **composição** (o `PontoColorido` *tem* um `Ponto`), tornar a classe `final`, ou usar *records*, que já são `final`.

Q38 Classe imutável

*Transforme a classe mutável abaixo numa classe **imutável** correta e explique cada mudança.*

```
import java.time.LocalDate;
```

```
record Periodo(LocalDate inicio, LocalDate fim) {
    Periodo {
        java.util.Objects.requireNonNull(inicio, "inicio");
        java.util.Objects.requireNonNull(fim, "fim");
        if (fim.isBefore(inicio)) throw new IllegalArgumentException("fim antes do início");
    }
    long dias() { return java.time.temporal.ChronoUnit.DAYS.between(inicio, fim); }
}
```

```
void main() {
    var p = new Periodo(LocalDate.of(2026, 1, 1), LocalDate.of(2026, 3, 1));
    IO.println(p + " -> " + p.dias() + " dias");
}
```

Saída

```
Periodo[inicio=2026-01-01, fim=2026-03-01] -> 59 dias
```

As mudanças:

- Campos `private final`, sem *setters*. O *record* já faz isso e já é `final`, então ninguém cria uma subclasse mutável.
- `java.util.Date` **é mutável**: mesmo com o campo `final`, quem guardasse a referência poderia mudar a data. Trocando por `LocalDate`, que é imutável, as cópias defensivas deixam de ser necessárias. Mantendo `Date`, seria preciso copiar na entrada (`new Date(d.getTime())`) e na saída.
- A **validação no construtor** garante que nenhum `Periodo` inválido chegue a existir.

FASE 3

A biblioteca padrão

Artigos 16 a 22

#16 Escreva uma Vez para Todo Tipo — Genéricos
#17 O Preço da Compatibilidade — Wildcards e Type Erasure
#18 Listas, Conjuntos e Mapas — o Collections Framework
#19 Ordem e Critério — Comparable, Comparator e Coleções Ordenadas
#20 Quando o Contrato se Quebra — Exceções Verificadas e Não Verificadas
#21 Recursos que se Fecham Sozinhos — try-with-resources, Arquivos e NIO.2
#22 Tempo, Dinheiro e Padrões — java.time, BigDecimal e Expressões Regulares

Perguntas conceituais

Q39 Por que genéricos

> *Que problema os genéricos resolveram? Compare uma `List` "crua" (raw type) com uma `List<String>` quanto ao momento em que um erro de tipo é descoberto.*

Antes do Java 5, as coleções guardavam `Object`: dava para pôr um `Integer` numa lista de nomes, e o erro só aparecia **em tempo de execução**, como `ClassCastException`, muitas vezes longe do lugar onde o dado errado entrou. Com `List<String>`, o **compilador** recusa o `add(42)` e dispensa os *casts* na leitura. O erro sai da produção e passa para a compilação.

Os *raw types* ainda compilam, por compatibilidade, mas geram avisos de *unchecked* e desligam essa proteção. Não devem ser usados em código novo.

Q40 Type erasure

> *O que é **type erasure**? Cite duas coisas que você **não** consegue fazer com um parâmetro de tipo `T` por causa dele.*

Os genéricos existem só no compilador: depois de checar os tipos, o `javac` **apaga** os parâmetros, trocando `T` pelo seu limite (normalmente `Object`) e inserindo *casts* onde preciso. Em tempo de execução, `List<String>` e `List<Integer>` são a mesma classe, `ArrayList`. Foi o preço da compatibilidade com o bytecode antigo.

Por isso não é possível:

- `new T()` nem `new T[10]` (a JVM não sabe o que é `T`);
- `obj instanceof List<String>` (só `List<?>`);
- ter duas sobrecargas `f(List<String>)` e `f(List<Integer>)`, porque depois do apagamento as assinaturas são iguais.

A saída clássica é receber um `Class<T>` ou uma fábrica (`Supplier<T>`, `IntFunction<T[]>`).

Q41 PECS

*Explique a regra **PECS** (Producer Extends, Consumer Super) usando a assinatura de `Collections.copy(List<? super T> destino, List<? extends T> origem)`.*

- Uma coleção de onde você **lê** elementos é um **produtor**: use `? extends T`. Dela sai algo que é pelo menos um `T`, mas não se pode inserir nada (exceto `null`), porque o tipo exato é desconhecido.
- Uma coleção onde você **escreve** é um **consumidor**: use `? super T`. Ela aceita um `T`, mas o que se lê dela só é garantidamente `Object`.

Em `Collections.copy`, a origem só produz (`? extends T`) e o destino só consome (`? super T`). Com isso dá para copiar uma `List<Integer>` para uma `List<Number>` ou `List<Object>`, algo impossível se os dois parâmetros fossem `List<T>`.

Q42 Invariância dos genéricos

*Por que `List<Integer>` **não** é subtipo de `List<Number>`, embora `Integer` seja subtipo de `Number`? Mostre o que daria errado se fosse.*

Se fosse subtipo, este código compilaria:

```
List<Integer> inteiros = new ArrayList<>();
List<Number> numeros = inteiros;   // (não compila — e ainda bem)
numeros.add(3.14);                 // um Double dentro de uma lista de Integer!
Integer i = inteiros.get(0);       // ClassCastException
```

Os genéricos são **invariantes** justamente para impedir isso. Os arrays, ao contrário, são covariantes (`Integer[]` é um `Number[]`), e por isso pagam uma verificação em tempo de execução: gravar um `Double` num `Integer[]` visto como `Number[]` lança `ArrayStoreException`. Quando só se quer ler, a forma segura de aceitar "lista de qualquer número" é `List<? extends Number>`.

Q43 ArrayList ou LinkedList

Compare `ArrayList` e `LinkedList`: qual é o custo de `get(i)`, de inserir no fim e de inserir no meio? Qual das duas deve ser a escolha padrão, e por quê?

- `get(i)`: O(1) no `ArrayList` (acesso direto ao array) contra O(n) na `LinkedList` (percorre nós).
- Inserir no fim: O(1) amortizado nas duas.
- Inserir no meio: O(n) nas duas. O `ArrayList` desloca elementos com uma cópia de memória muito rápida. A `LinkedList` precisa **caminhar** até a posição, a não ser que já esteja lá com um iterador.

O `ArrayList` é a escolha padrão: os elementos (referências) ficam contíguos na memória e aproveitam o cache do processador, enquanto cada nó da `LinkedList` é um objeto separado, com dois ponteiros extras. Para fila e pilha, a escolha certa é o `ArrayDeque`, e não a `LinkedList`.

Q44 ConcurrentModificationException

> *O que é uma `ConcurrentModificationException`? Por que ela aparece num programa **sem nenhuma thread**, e quais são as formas corretas de remover elementos enquanto se percorre uma lista?*

Os iteradores das coleções comuns são *fail-fast*: guardam um contador de modificações (`modCount`) e lançam a exceção se a coleção for alterada por fora do próprio iterador durante a iteração. O caso típico não tem nada de concorrência: é chamar `lista.remove(x)` dentro de um `for-each`.

Formas corretas:

- `lista.removeIf(x -> condição)`, a mais simples;
- um `Iterator` explícito com `it.remove()`;
- percorrer por índice **de trás para frente**;
- construir uma nova lista filtrada (com *streams*, por exemplo).

Q45 Coleções imutáveis e visões

> *Qual é a diferença entre `List.of(...)`, `Collections.unmodifiableList(lista)` e `List.copyOf(lista)`?*

- `List.of(a, b, c)` cria uma lista **imutável** nova. Não aceita `null`, e qualquer tentativa de mudança lança `UnsupportedOperationException`.
- `Collections.unmodifiableList(lista)` devolve uma **visão** somente leitura da lista original. Quem tiver a original ainda pode alterá-la, e a mudança aparece na visão.
- `List.copyOf(lista)` faz uma **cópia** imutável (ou devolve a própria lista, se ela já for uma lista imutável do `List.of`). Mudanças posteriores na original não afetam a cópia.

Para cópias defensivas em construtores, o certo é `List.copyOf`.

Q46 Sequenced Collections

> *Para que servem as **Sequenced Collections** (Java 21)? Dê dois métodos novos e duas classes que passaram a implementá-las.*

Até o Java 21 não havia um tipo comum para "coleção com ordem de encontro definida": pegar o último elemento era `lista.get(lista.size() - 1)` numa `List`, `deque.getLast()` num `Deque` e um laço inteiro num `LinkedHashSet`. As interfaces `SequencedCollection`, `SequencedSet` e `SequencedMap` unificaram isso com `getFirst()`, `getLast()`, `addFirst()`, `addLast()`, `removeFirst()`, `removeLast()` e `reversed()` (uma visão invertida). `List`, `Deque`, `LinkedHashSet`, `SortedSet` e `LinkedHashMap` passaram a implementá-las (no caso dos mapas, `firstEntry()`, `lastEntry()` e `pollFirstEntry()`).

Q47 Comparable, Comparator e o overflow

> *Qual é a diferença entre `Comparable` e `Comparator`? Por que escrever `return a.idade - b.idade;` num comparador é um erro, ainda que funcione quase sempre?*

`Comparable<T>` define a **ordem natural** dentro da própria classe (`compareTo`). Cada classe tem uma só (`String` em ordem alfabética, `Integer` numérica). `Comparator<T>` é um critério **externo**, e pode haver quantos se quiser (por nome, por idade, decrescente...).

A subtração estoura: com `a = Integer.MIN_VALUE` e `b = 1`, a diferença dá a volta e vira um número positivo enorme, dizendo que `a > b`. Use `Integer.compare(a.idade, b.idade)` ou, melhor ainda, `Comparator.comparingInt(Pessoa::idade)`.

Q48 Checked ou unchecked

> *Explique a diferença entre exceções **verificadas** e **não verificadas**, e dê a regra de bolso de quando usar cada uma.*

Verificadas (subclasses de `Exception` que não são `RuntimeException`, como `IOException` e `SQLException`) obrigam o chamador a tratar ou declarar com `throws`. **Não verificadas** (`RuntimeException` e subclasses, como `IllegalArgumentException` e `NullPointerException`, e os `Error`s) não obrigam nada.

Regra de bolso: exceção verificada para condições **esperadas e recuperáveis**, que o chamador pode tratar de um jeito razoável (um arquivo que sumiu, a rede que caiu). Não verificada para **erros de programação** (argumento inválido, estado impossível), que devem ser corrigidos no código, e não tratados. Na prática, muitas bibliotecas modernas preferem só exceções não verificadas, porque as verificadas combinam mal com *lambdas*.

Q49 Engolir exceções

> *O que há de errado neste tratamento de exceção? Liste pelo menos três problemas.*

```
try {
    processarPedido(pedido);
} catch (Exception e) {
    // TODO
}
```

- **Engole** o erro: o pedido falha em silêncio, e ninguém fica sabendo nem por quê. É o pior tratamento possível.
- Captura `Exception` inteira: pega também erros de programação (`NullPointerException`, `IllegalStateException`) que deveriam aparecer e ser corrigidos.
- Perde o *stack trace*, a informação mais valiosa para depurar.
- Deixa o sistema num estado incerto: o chamador acredita que o pedido foi processado.

O certo é capturar só o que se sabe tratar, registrar com contexto (`log.error("pedido {} falhou", id, e)`) ou relançar embrulhando **com a causa**: `throw new PedidoException("falha no pedido " + id, e)`.

Q50 Exceções suprimidas

> *O que são **exceções suprimidas** no `try` com recursos? Em que situação elas aparecem?*

Se o bloco `try` lança uma exceção e, ao fechar o recurso, o `close()` **também** lança, a exceção do `try` é a principal (é a que conta a causa real), e a do `close` é anexada a ela como **suprimida**. Pode ser recuperada

com `e.getSuppressed()` e aparece no *stack trace* como "Suppressed:". No `try`/`finally` antigo acontecia o contrário: a exceção do `finally` **substituía** a original, que se perdia.

Q51 Os tipos de tempo

> *Qual é a diferença entre `Instant`, `LocalDateTime` e `ZonedDateTime`? Qual deles usar para gravar **quando** um evento aconteceu?*

- `Instant`: um ponto exato na linha do tempo, em UTC, sem fuso nem calendário. É o que se grava para dizer "quando aconteceu".
- `LocalDateTime`: data e hora "de parede", **sem fuso**. "25/12/2026 às 20h" não diz em que lugar do mundo, e portanto não identifica um instante.
- `ZonedDateTime`: data e hora com fuso (`America/Sao_Paulo`), incluindo as regras de horário de verão. Serve para agendar e para mostrar o horário a uma pessoa.

Grave `Instant` (ou `OffsetDateTime` em UTC) e converta para `ZonedDateTime` só na hora de exibir, no fuso de quem vai ler.

Q52 Armadilhas do BigDecimal

> *Por que `new BigDecimal(0.1)` e `BigDecimal.valueOf(0.1)` dão resultados diferentes? E por que `new BigDecimal("2.0").equals(new BigDecimal("2.00"))` é `false`?*

`new BigDecimal(0.1)` recebe o `double` 0,1, que já é uma aproximação binária, e a guarda com todas as casas: `0.1000000000000000055511151231257827021181583404541015625`. `BigDecimal.valueOf(0.1)` usa a representação em texto do `double` (`"0.1"`) e dá exatamente 0,1. O mais seguro é sempre construir a partir de `String`: `new BigDecimal("0.10")`.

O `equals` do `BigDecimal` compara valor **e escala**: 2,0 (escala 1) e 2,00 (escala 2) são diferentes. Para comparar só o valor numérico, use `compareTo(...) == 0`. O mesmo cuidado vale para usar `BigDecimal` como chave de `HashMap`.

Exercícios de código — soluções

Q53 Máximo genérico

> *Escreva um método genérico `static <T extends Comparable<? super T>> T maximo(List<? extends T> lista)` que devolva o maior elemento e lance `NoSuchElementException` para lista vazia. Explique cada parte da assinatura.*

```
static <T extends Comparable<? super T>> T maximo(List<? extends T> lista) {
    if (lista.isEmpty()) throw new NoSuchElementException("lista vazia");
    T maior = lista.getFirst();
    for (T x : lista) if (x.compareTo(maior) > 0) maior = x;
    return maior;
}
```

```
void main() {
    IO.println(maximo(List.of(3, 9, 4)) + " " + maximo(List.of("pera", "uva", "maçã")));
    try { maximo(List.<Integer>of()); } catch (NoSuchElementException e) { IO.println("erro: " +
e.getMessage()); }
}
```

Saída
```
9 uva
erro: lista vazia
```

- `<T extends Comparable<? super T>>`: `T` sabe se comparar consigo mesmo **ou** herdou a comparação de um supertipo. Por exemplo, `java.sql.Timestamp` é `Comparable<Date>`, e não `Comparable<Timestamp>`.
- `List<? extends T>`: a lista só **produz** elementos (PECS), então aceita uma lista de qualquer subtipo de `T`.

Q54 Frequência de palavras

Conte a frequência das palavras de um texto com um `Map<String, Integer>` usando `merge`, e imprima as 3 mais frequentes, com empates desempatados por ordem alfabética.

```
void main() {
    String texto = "o rato roeu a roupa do rei de roma e o rei riu o rato fugiu";
    Map<String, Integer> freq = new HashMap<>();
    for (String p : texto.split("\\s+")) freq.merge(p, 1, Integer::sum);   // soma 1, ou cria com
1
```

```
    freq.entrySet().stream()
            .sorted(Map.Entry.<String, Integer>comparingByValue().reversed()
                    .thenComparing(Map.Entry.comparingByKey()))
            .limit(3)
            .forEach(e -> IO.println(e.getKey() + " = " + e.getValue()));
}
```

Saída
```
o = 3
rato = 2
rei = 2
```

O `merge(chave, 1, Integer::sum)` resolve numa chamada o "se não existe, põe 1; se existe, soma". O comparador ordena por contagem decrescente e, nos empates, pela palavra.

Q55 Exceção de domínio

Crie a exceção verificada `SaldoInsuficienteException`, que guarda o saldo e o valor pedido, e use-a num método `sacar`. Mostre o tratamento no chamador, imprimindo uma mensagem útil.

```
class SaldoInsuficienteException extends Exception {
    private final long saldo, pedido;
    SaldoInsuficienteException(long saldo, long pedido) {
        super("saldo %d insuficiente para sacar %d".formatted(saldo, pedido));
        this.saldo = saldo;
        this.pedido = pedido;
    }
    long falta() { return pedido - saldo; }
}
```

```
class Conta {
    private long saldo;                                    // em centavos
    Conta(long saldo) { this.saldo = saldo; }
    void sacar(long valor) throws SaldoInsuficienteException {
        if (valor <= 0) throw new IllegalArgumentException("valor deve ser positivo");   // erro
de programação
        if (valor > saldo) throw new SaldoInsuficienteException(saldo, valor);          //
situação de negócio
        saldo -= valor;
    }
}
```

```
void main() {
    var c = new Conta(10_000);
    try {
        c.sacar(3_000);
        c.sacar(9_000);
    } catch (SaldoInsuficienteException e) {
        IO.println(e.getMessage() + " (faltam " + e.falta() + ")");
    }
}
```

Saída

```
saldo 7000 insuficiente para sacar 9000 (faltam 2000)
```

Saldo insuficiente é uma situação de negócio esperada: vira exceção verificada, com dados que o chamador pode usar. Um valor negativo é erro de quem chamou: vira `IllegalArgumentException`. Guardar valores em centavos (`long`) evita os problemas do `double`.

Q56 Arquivos com NIO.2

Usando NIO.2 (`Files` e `Path`), escreva um programa que crie a pasta `saida`, grave nela um arquivo `diario.txt` com três linhas, depois leia o arquivo com `Files.lines` dentro de um `try` com recursos, contando as linhas que contêm "Java".

```
import java.nio.file.*;
```

```
void main() throws IOException {
    Path pasta = Files.createDirectories(Path.of("saida"));        // não falha se já existir
    Path arquivo = pasta.resolve("diario.txt");
    Files.write(arquivo, List.of("dia 1: aprendi Java", "dia 2: descansei", "dia 3: mais Java"));
```

```
    try (Stream<String> linhas = Files.lines(arquivo)) {           // Files.lines mantém o
arquivo aberto: feche
        long comJava = linhas.filter(l -> l.contains("Java")).count();
        IO.println(arquivo.getFileName() + ": " + comJava + " linha(s) com Java");
    }
}
```

Saída

```
diario.txt: 2 linha(s) com Java
```

`Files.write` e `Files.readAllLines` abrem e fecham o arquivo sozinhos. `Files.lines` é preguiçoso e mantém o arquivo aberto enquanto o *stream* existe, por isso vai num `try` com recursos. A codificação padrão das duas APIs é UTF-8.

Q57 Datas com java.time

Calcule as datas de vencimento de 3 parcelas mensais a partir de 31/01/2026 e diga quantos dias faltam da primeira até a última. Explique o que o `plusMonths` faz com o dia 31.

```
import java.time.*;
import java.time.temporal.ChronoUnit;
```

```
void main() {
    LocalDate compra = LocalDate.of(2026, 1, 31);
    List<LocalDate> vencimentos = new ArrayList<>();
    for (int i = 1; i <= 3; i++) vencimentos.add(compra.plusMonths(i));   // sempre a partir da
data original
    IO.println(vencimentos);
    IO.println(ChronoUnit.DAYS.between(vencimentos.getFirst(), vencimentos.getLast()) + " dias");
}
```

Saída

```
[2026-02-28, 2026-03-31, 2026-04-30]
61 dias
```

Fevereiro de 2026 não tem dia 31, e o `plusMonths` **ajusta para o último dia válido** (28). Somar sempre a partir da data original (`compra.plusMonths(i)`) mantém o dia 31 nos meses que o têm. Encadear (`d = d.plusMonths(1)`) "grudaria" no dia 28 para sempre.

Q58 Regex com grupos nomeados

*Escreva uma expressão regular com **grupos nomeados** que extraia dia, mês e ano de datas no formato `dd/mm/aaaa` dentro de um texto, e imprima cada uma no formato ISO (`aaaa-mm-dd`).*

```
import java.util.regex.*;
```

```
void main() {
    String texto = "Entrega em 05/03/2026, pagamento em 10/04/2026; 99/99/99 não é data.";
    Pattern data = Pattern.compile("\\b(?<dia>\\d{2})/(?<mes>\\d{2})/(?<ano>\\d{4})\\b");
    Matcher m = data.matcher(texto);
    while (m.find()) IO.println(m.group("ano") + "-" + m.group("mes") + "-" + m.group("dia"));
}
```

Saída

```
2026-03-05
2026-04-10
```

O `Pattern` é compilado uma vez e reaproveitado. `\b` evita pegar pedaços de números maiores. A regex confere só o **formato**: "31/02/2026" passaria. Para validar a data de verdade, converta com `LocalDate.parse(texto, DateTimeFormatter.ofPattern("dd/MM/uuuu").withResolverStyle(ResolverStyle.STRICT))`.

FASE 4

Java funcional

Artigos 23 a 27

#23 Comportamento como Valor — Lambdas e Interfaces Funcionais
#24 Pare de Escrever Laços — a API de Streams
#25 Agrupar, Achatar, Reduzir — Collectors e Gatherers
#26 Um Valor que Pode Não Existir — Optional e a Guerra contra o null
#27 O Programa que se Examina — Anotações e Reflexão

Perguntas conceituais

Q59 Interfaces funcionais

*O que é uma **interface funcional**? Associe cada uma destas interfaces à forma da função que ela representa: `Supplier<T>`, `Consumer<T>`, `Function<T, R>`, `Predicate<T>`, `UnaryOperator<T>` e `BiFunction<T, U, R>`.*

É uma interface com **exatamente um método abstrato** (pode ter vários `default` e `static`). É o tipo que uma *lambda* ou uma referência de método assume. A anotação `@FunctionalInterface` pede ao compilador que confira a regra.

- `Supplier<T>`: `() -> T`, produz um valor sem receber nada;
- `Consumer<T>`: `T -> void`, consome e não devolve;
- `Function<T, R>`: `T -> R`, transforma;
- `Predicate<T>`: `T -> boolean`, testa;
- `UnaryOperator<T>`: `T -> T`, transforma sem mudar o tipo;
- `BiFunction<T, U, R>`: `(T, U) -> R`, dois argumentos.

Também existem as versões para primitivos (`IntPredicate`, `ToLongFunction`...), que evitam o *boxing*.

Q60 Variáveis efetivamente finais

*Por que uma lambda só pode usar variáveis locais que sejam **efetivamente finais**? Por que essa restrição não vale para campos?*

A *lambda* não enxerga a variável local: ela **captura uma cópia do valor** no momento em que é criada, porque pode ser executada depois que o método terminou e o quadro da pilha já foi desfeito, ou até em outra *thread*. Se a variável pudesse mudar, a cópia e o original divergiriam em silêncio. Proibir a mudança torna a cópia indistinguível do original.

Campos não são copiados: a *lambda* captura `this` e lê o campo pelo objeto, que vive no heap. Por isso eles podem mudar, e trazem de volta todos os cuidados de concorrência.

Q61 Referências de método

*Cite os quatro tipos de **referência de método** e dê um exemplo de cada.*

1. Método estático: `Integer::parseInt` equivale a `s -> Integer.parseInt(s)`;

2. Método de um objeto específico: `IO::println` equivale a `x -> IO.println(x)`, e `saudacao::concat` usa sempre o mesmo objeto;

3. Método de instância de um objeto qualquer do tipo: `String::toUpperCase` equivale a `s -> s.toUpperCase()`, e o objeto é o primeiro argumento;

4. Construtor: `ArrayList::new` equivale a `() -> new ArrayList<>()`, e `int[]::new` cria arrays.

Q62 Preguiça dos streams

> *Explique a **preguiça** (laziness) dos streams. Quantas vezes o `filter` é executado no código abaixo, e por quê?*

```
var r = Stream.of("ana", "bia", "caio", "davi", "eva")
        .filter(n -> { IO.println("filtrando " + n); return n.length() == 4; })
        .findFirst();
```

Três vezes: para "ana", "bia" e "caio". As operações intermediárias (`filter`, `map`...) **não fazem nada** quando são chamadas: só montam o *pipeline*. O trabalho começa na operação terminal (`findFirst`), e os elementos passam **um de cada vez** pelo *pipeline* inteiro. Assim que "caio" passa pelo filtro, o `findFirst` tem sua resposta e o *stream* para: "davi" e "eva" nunca são examinados. É o *short-circuit*, que também permite trabalhar com *streams* infinitos (`Stream.iterate(...).filter(...).limit(10)`).

Q63 Streams de uso único e efeitos colaterais

> *Por que um stream não pode ser consumido duas vezes? E por que não se deve modificar uma variável externa (por exemplo, somar num campo) de dentro de um `forEach` ou `map`?*

O *stream* não é uma coleção: é uma **sequência de passagem**, ligada a uma fonte que pode nem ser relida (um arquivo, a rede, um gerador). Uma segunda operação terminal lança `IllegalStateException: stream has already been operated upon or closed`. Para percorrer de novo, crie outro *stream* a partir da fonte.

Efeitos colaterais quebram as garantias da API: num *stream* paralelo, somar numa variável compartilhada é uma condição de corrida, e mesmo no sequencial o código fica difícil de ler e de mudar. O certo é **reduzir**: `sum()`, `reduce`, `collect`, `count`.

Q64 map e flatMap

> *Qual é a diferença entre `map` e `flatMap`? Dê um exemplo em que só `flatMap` resolve.*

`map` transforma cada elemento em **um** elemento: `Stream<Pedido>` → `Stream<Cliente>`. `flatMap` transforma cada elemento num **stream** e "achata" todos num só: `Stream<Pedido>` → `Stream<Item>`, em que cada pedido tem vários itens.

Com `map`, `pedidos.stream().map(p -> p.itens().stream())` daria um `Stream<Stream<Item>>`. Já `pedidos.stream().flatMap(p -> p.itens().stream())` dá o `Stream<Item>` com todos os itens de todos os

pedidos. O mesmo nome e a mesma ideia existem em `Optional.flatMap`. O Java 16 acrescentou o `mapMulti`, uma alternativa imperativa ao `flatMap`.

Q65 Gatherers

> *O que são os **Gatherers** (Java 24, JEP 485)? Que tipo de operação eles permitem que antes era difícil com streams?*

Gatherers são **operações intermediárias personalizadas**, usadas com `stream.gather(...)`. Até o Java 23 só era possível personalizar a operação **terminal** (com `Collector`). As intermediárias eram fixas (`map`, `filter`, `limit`...), e operações com estado entre elementos exigiam gambiarras.

A classe `Gatherers` traz prontas:

- `windowFixed(n)`: janelas de n elementos;
- `windowSliding(n)`: janelas deslizantes, útil para médias móveis;
- `fold` e `scan`: acumulações, sendo que o `scan` emite todos os valores parciais;
- `mapConcurrent(n, f)`: aplica `f` com até n execuções simultâneas em *virtual threads*.

Q66 Maus usos de Optional

> *Liste três usos **errados** de `Optional` e diga o que fazer no lugar de cada um.*

- **Campo** `Optional<String> apelido`: o `Optional` não é serializável e acrescenta um objeto por campo. No lugar, use um campo que pode ser nulo e faça o *getter* devolver `Optional.ofNullable(apelido)`.
- **Parâmetro** `void enviar(Optional<Anexo> anexo)`: obriga o chamador a embrulhar, e ainda pode receber `null`. No lugar, use duas sobrecargas, ou um parâmetro que pode ser nulo e está documentado.
- `opt.get()` sem checar, ou `if (opt.isPresent()) x = opt.get();`: é o mesmo `null` de antes, com mais cerimônia. No lugar, use `orElse`, `orElseGet`, `orElseThrow`, `map` e `ifPresentOrElse`.

Há outros: `Optional<List<X>>` (devolva uma lista vazia) e `Optional.of(x)` com `x` possivelmente nulo, que lança `NullPointerException` (use `ofNullable`).

Q67 orElse ou orElseGet

> *Qual é a diferença entre `orElse(criar())` e `orElseGet(() -> criar())`? Quando essa diferença importa?*

`orElse` recebe um **valor**, e o Java avalia os argumentos antes da chamada: `criar()` executa **sempre**, mesmo quando o `Optional` tem valor e o resultado é descartado. `orElseGet` recebe um `Supplier`, que só é chamado se o `Optional` estiver vazio. A diferença importa quando o valor alternativo é caro (consulta ao banco, chamada de rede, objeto grande) ou tem efeito colateral (gravar um registro, incrementar um contador). Para constantes, como `orElse("")`, tanto faz.

Q68 Retenção e alvo

*O que é **retenção** de uma anotação (`SOURCE`, `CLASS`, `RUNTIME`)? Que retenção é obrigatória para que um framework leia a anotação por reflexão, e em que tipos de elemento ela pode ser posta (`@Target`)?*

A retenção diz até onde a anotação sobrevive:

- `SOURCE`: só no código-fonte, e o compilador a descarta. Exemplos: `@Override`, `@SuppressWarnings`;
- `CLASS` (o padrão): gravada no `.class`, mas invisível à reflexão. Serve a ferramentas de bytecode;
- `RUNTIME`: disponível em tempo de execução via reflexão. É a que *frameworks* como JUnit, Spring e JPA usam.

O `@Target` restringe onde ela pode ser usada: `TYPE`, `METHOD`, `FIELD`, `PARAMETER`, `CONSTRUCTOR`, `RECORD_COMPONENT` etc.

Q69 Custos da reflexão

*Cite duas desvantagens da **reflexão** que justificam usá-la com moderação no código de aplicação.*

- **Perde a checagem do compilador**: nomes de métodos e campos viram *strings*, e um erro de digitação ou um renomeio só aparece em tempo de execução.
- **Quebra o encapsulamento**: `setAccessible(true)` acessa membros privados, e desde o Java 9 isso é barrado pelo sistema de módulos, a não ser que o pacote seja aberto com `opens`.
- É mais **lenta** que a chamada direta (o JIT otimiza menos) e atrapalha ferramentas como o GraalVM Native Image, que precisa saber de antemão o que será acessado por reflexão.

Ela é ótima para *frameworks* genéricos. Na aplicação, prefira interfaces e *lambdas*.

Exercícios de código — soluções

Q70 Collectors na prática

Dada uma lista de vendas (`record Venda(String vendedor, String regiao, double valor)`), use streams para obter: (a) o total por região, ordenado pelo nome da região; (b) o vendedor com o maior total; (c) as regiões separadas entre "acima" e "abaixo" de 1.000 em total.

```
record Venda(String vendedor, String regiao, double valor) {}
```

```
void main() {
    var vendas = List.of(
            new Venda("Ana", "Sul", 500), new Venda("Bia", "Norte", 300),
            new Venda("Ana", "Sul", 700), new Venda("Caio", "Norte", 900),
            new Venda("Bia", "Leste", 200));
```

```
    // (a) total por região, com as chaves em ordem (TreeMap)
    Map<String, Double> porRegiao = vendas.stream()
            .collect(Collectors.groupingBy(Venda::regiao, TreeMap::new,
Collectors.summingDouble(Venda::valor)));
    IO.println(porRegiao);
```

```
    // (b) vendedor com maior total
    String campeao = vendas.stream()
            .collect(Collectors.groupingBy(Venda::vendedor,
Collectors.summingDouble(Venda::valor)))
            .entrySet().stream()
            .max(Map.Entry.comparingByValue())
            .map(Map.Entry::getKey)
            .orElse("ninguém");
    IO.println("campeão: " + campeao);
```

```
    // (c) regiões acima/abaixo de 1000
    Map<Boolean, List<String>> faixas = porRegiao.entrySet().stream()
            .collect(Collectors.partitioningBy(e -> e.getValue() > 1000,
                    Collectors.mapping(Map.Entry::getKey, Collectors.toList())));
    IO.println("acima: " + faixas.get(true) + ", abaixo: " + faixas.get(false));
}
```

Saída

```
{Leste=200.0, Norte=1200.0, Sul=1200.0}
campeão: Ana
acima: [Norte, Sul], abaixo: [Leste]
```

O `groupingBy` com três argumentos escolhe o tipo do mapa (`TreeMap`, para ordenar) e o coletor de cada grupo. O `partitioningBy` sempre produz as duas chaves, `true` e `false`, mesmo que uma fique vazia. Para dinheiro de verdade, troque `double` por `BigDecimal` (com `Collectors.reducing(BigDecimal.ZERO, Venda::valor, BigDecimal::add)`).

Q71 Gatherers na prática

> *Use `Gatherers.windowSliding` para calcular a **média móvel** de 3 dias de uma série de temperaturas, e `Gatherers.scan` para o total acumulado de uma lista de depósitos.*

```
import java.util.stream.Gatherers;
```

```
void main() {
    var temperaturas = List.of(20, 22, 27, 25, 21, 19);
    var medias = temperaturas.stream()
            .gather(Gatherers.windowSliding(3))                        // [20,22,27], [22,27,25],
...
            .map(j -> j.stream().mapToInt(Integer::intValue).average().orElseThrow())
            .map(m -> "%.1f".formatted(m))
            .toList();
    IO.println(medias);
```

```
    var acumulado = Stream.of(100, 50, -30, 200)
            .gather(Gatherers.scan(() -> 0, Integer::sum))            // emite cada total
parcial
            .toList();
    IO.println(acumulado);
}
```

Saída

```
[23.0, 24.7, 24.3, 21.7]
[100, 150, 120, 320]
```

Com 6 valores e janelas de 3 saem 4 médias. O `scan` é um "`reduce` que mostra os passos", e isso antes exigia uma variável externa mutável dentro de um `map`, exatamente o efeito colateral que os *streams* desaconselham.

Q72 Optional encadeado

Reescreva o código abaixo usando `Optional` de forma idiomática, sem nenhum `if` e sem `get()`.

```
record Endereco(String cidade) {}
record Cliente(long id, Endereco endereco) {}
```

```
Map<Long, Cliente> banco = Map.of(
        1L, new Cliente(1, new Endereco("Recife")),
        2L, new Cliente(2, null));
```

```
Optional<Cliente> buscar(long id) { return Optional.ofNullable(banco.get(id)); }
```

```
String cidadeDoCliente(long id) {
    return buscar(id)
            .map(Cliente::endereco)                // map devolve Optional vazio se o resultado
for null
            .map(Endereco::cidade)
            .map(String::toUpperCase)
            .orElse("DESCONHECIDA");
}
```

```
void main() {
    IO.println(cidadeDoCliente(1) + " " + cidadeDoCliente(2) + " " + cidadeDoCliente(3));
}
```

Saída

```
RECIFE DESCONHECIDA DESCONHECIDA
```

O ponto-chave é que `Optional.map` transforma um resultado `null` num `Optional` vazio, e cada etapa só executa se a anterior teve valor. O ideal é o próprio repositório devolver `Optional<Cliente>`, e não `null`: o tipo de retorno avisa o chamador de que o cliente pode não existir.

Q73 Mini validador por reflexão

*Crie a anotação `@Positivo`, aplicável a campos, e um validador que use **reflexão** para conferir, em qualquer objeto, se os campos `int` anotados são maiores que zero, devolvendo a lista de violações.*

```
import java.lang.annotation.*;
import java.lang.reflect.Field;
```

```
@Retention(RetentionPolicy.RUNTIME)            // sem RUNTIME, a reflexão não enxerga a anotação
@Target(ElementType.FIELD)
@interface Positivo {}
```

```
static class Produto {
    @Positivo int estoque;
    @Positivo int preco;
    int desconto;                                  // sem anotação: não é conferido
    Produto(int estoque, int preco, int desconto) { this.estoque = estoque; this.preco = preco;
this.desconto = desconto; }
}
```

```
List<String> validar(Object alvo) throws IllegalAccessException {
    List<String> erros = new ArrayList<>();
    for (Field f : alvo.getClass().getDeclaredFields()) {
        if (f.isAnnotationPresent(Positivo.class) && f.getType() == int.class) {
            f.setAccessible(true);               // funciona: mesma classe/módulo sem nome
            int v = f.getInt(alvo);
            if (v <= 0) erros.add(f.getName() + " deve ser positivo (veio " + v + ")");
        }
    }
    return erros;
}
```

```
void main() throws Exception {
    IO.println(validar(new Produto(5, 10, -3)));
    IO.println(validar(new Produto(0, -1, 0)));
}
```

Saída

```
[]
[estoque deve ser positivo (veio 0), preco deve ser positivo (veio -1)]
```

É, em miniatura, o que o Bean Validation (`@Positive`, `@NotBlank`) faz nas Trilhas da Ilha. Uma ressalva: a ordem de `getDeclaredFields()` **não é garantida** pela especificação. Na HotSpot ela segue a declaração, mas um validador sério ordena ou não depende disso.

FASE 5

Projetos e ferramentas

Artigos 28 a 32

#28 Do Arquivo Solto ao Projeto — Pacotes, Classpath e JARs
#29 Construção Automatizada — Maven e Gradle
#30 Fronteiras Explícitas — o Sistema de Módulos
#31 Provando que Funciona — Testes com JUnit e AssertJ
#32 Dublês e Disciplina — Mockito e Testes que Não Mentem

Perguntas conceituais

Q74 Pacotes e classpath

> *Qual é a relação entre o nome do **pacote** e a estrutura de pastas? O que é o **classpath**, e o que acontece quando duas bibliotecas no classpath têm uma classe com o mesmo nome completo?*

O pacote `br.com.exemplo.farol.http` corresponde à pasta `br/com/exemplo/farol/http/`, tanto nos fontes quanto nos `.class` e dentro dos JARs, e o nome completo da classe (`br.com.exemplo.farol.http.Requisicao`) é o que a identifica. O classpath é a lista de pastas e JARs onde a JVM procura as classes, na ordem em que aparecem.

Se dois JARs trazem a mesma classe, vence **a primeira encontrada**, em silêncio. Com versões diferentes da mesma biblioteca, o resultado é o "inferno do classpath": `NoSuchMethodError` ou comportamento estranho, dependendo da ordem. O sistema de módulos detecta esse tipo de conflito (pacotes divididos entre módulos) já na inicialização.

Q75 JAR executável e JAR gordo

> *O que o arquivo `META-INF/MANIFEST.MF` de um JAR executável precisa ter? Qual é a diferença entre um JAR comum e um "JAR gordo" (fat jar)?*

Para `java -jar app.jar` funcionar, o manifesto precisa da linha `Main-Class: pacote.Principal`. Opcionalmente, pode ter um `Class-Path:` apontando para outros JARs, e desde o Java 22 também um `Enable-Native-Access: ALL-UNNAMED`.

Um JAR comum tem só as classes do projeto, e as dependências precisam estar no classpath. Um **JAR gordo** traz as dependências desempacotadas dentro dele (com o *maven-shade-plugin*, o `bootJar` do Spring Boot ou o `jar` do projeto libGDX) e roda sozinho, mas é maior, e pode sofrer colisões de arquivos com o mesmo nome (como os `META-INF/services`).

Q76 Ciclo de vida do Maven

> *Explique as **fases** do ciclo de vida padrão do Maven que você mais usa (`compile`, `test`, `package`, `verify`, `install`) e o que significa rodar `mvn package`.*

O ciclo é uma sequência ordenada, e pedir uma fase executa **todas as anteriores**:

- `compile`: compila `src/main/java`;
- `test`: compila e roda os testes de `src/test/java` com o Surefire. Uma falha interrompe o ciclo;
- `package`: empacota o resultado (JAR ou WAR) em `target/`;
- `verify`: roda verificações extras, como os testes de integração do Failsafe;
- `install`: copia o artefato para o repositório local (`~/.m2`), para outros projetos da máquina usarem.

`mvn package` faz, portanto, validação, compilação, testes e empacotamento. Com `-DskipTests` pula os testes, o que só deve ser um atalho consciente.

Q77 Escopos de dependência

> *Qual é a diferença entre os escopos de dependência `compile`, `test`, `provided` e `runtime` no Maven (ou `implementation`, `testImplementation`, `compileOnly` e `runtimeOnly` no Gradle)?*

- `compile` / `implementation`: necessária para compilar e rodar. É o padrão.
- `test` / `testImplementation`: só para compilar e rodar os testes (JUnit, AssertJ, Mockito). Não vai para o artefato final.
- `provided` / `compileOnly`: necessária para compilar, mas quem fornece em execução é o ambiente (a API de Servlet num servidor de aplicação, por exemplo).
- `runtime` / `runtimeOnly`: não é usada no código-fonte, só na execução (um *driver* JDBC, por exemplo, que o código acessa pela interface `java.sql`).

Escopos corretos deixam o artefato menor e impedem que o código dependa, por engano, de algo que só deveria existir nos testes.

Q78 Diretivas de módulo

> *No `module-info.java`, o que fazem as diretivas `requires`, `requires transitive`, `exports` e `opens`? Por que frameworks como JPA e Jackson precisam de `opens`?*

- `requires X`: meu módulo depende do módulo X e pode usar os pacotes que ele exporta.
- `requires transitive X`: além disso, quem depender de mim também "lê" X automaticamente. Serve quando os tipos de X aparecem na minha API pública.
- `exports p`: o pacote p fica acessível, **em tempo de compilação e execução**, para outros módulos. Só os tipos públicos, e sem reflexão profunda.
- `opens p`: o pacote p fica aberto à **reflexão profunda** em tempo de execução (acesso a membros privados), sem ser exportado para compilação.

JPA, Jackson e o Spring leem e escrevem campos privados por reflexão. Sem `opens`, o `setAccessible(true)` falha com `InaccessibleObjectException`. Pode-se abrir só para quem precisa: `opens br.com.exemplo.modelo to com.fasterxml.jackson.databind;`.

Q79 Encapsulamento forte

> *O que é o **encapsulamento forte** que o sistema de módulos trouxe? Por que `sun.misc.Unsafe` e as classes internas do JDK deixaram de ser acessíveis por padrão?*

Antes dos módulos, `public` significava "público para o mundo inteiro". Qualquer JAR usava classes internas do JDK (`sun.*`, `com.sun.*.internal`), e o JDK não conseguia mudá-las sem quebrar programas. Com os módulos, **só os pacotes exportados** são acessíveis, e desde o Java 16/17 o acesso reflexivo às partes internas é barrado por padrão (exige `--add-opens`). O `sun.misc.Unsafe` ficou no módulo `jdk.unsupported` por compatibilidade, e seus métodos de acesso à memória estão sendo removidos em etapas, com substitutos oficiais: `VarHandle` e a FFM API. Os avisos que aparecem nos jogos da Trilha C são desse processo.

Q80 Pirâmide de testes

> *O que é a pirâmide de testes? Por que testes **unitários** devem ser a maioria, e o que um teste de **integração** verifica que o unitário não verifica?*

A pirâmide recomenda muitos testes unitários na base, uma quantidade média de testes de integração e poucos testes de ponta a ponta no topo.

- **Unitários** testam uma unidade isolada, sem rede, banco ou disco. Rodam em milissegundos e apontam o defeito com precisão. Dá para ter milhares.
- **De integração** verificam que as peças funcionam **juntas e com o mundo real**: o SQL que o banco aceita de fato, o JSON que o Jackson produz, a configuração do Spring, o *driver*. Mocks não pegam esse tipo de erro.
- **De ponta a ponta** simulam o usuário. São lentos e frágeis, e ficam só para os fluxos críticos.

Q81 Testes que mentem

> *O que torna um teste **"mentiroso"**? Cite três sinais de que um teste passa sem provar nada.*

- **Sem asserção** relevante: o teste chama o código e só confere que "não explodiu".
- **Mock demais**: tudo é simulado, inclusive a lógica que deveria ser testada, e o teste só confirma que o *mock* devolve o que foi programado para devolver.
- **Verifica a implementação, não o comportamento**: confere que o método X foi chamado 3 vezes. Qualquer refatoração quebra o teste, e um *bug* real passa.
- **Nunca foi visto falhando**: se você não viu o teste ficar vermelho ao quebrar o código de propósito, não sabe se ele testa alguma coisa.
- Depende da ordem de execução, do relógio ou da rede, e passa ou falha ao acaso.

Q82 Dublês de teste

> *Qual é a diferença entre um stub, um mock e um spy no Mockito? Quando é preferível escrever um **fake** à mão?*

- **Stub**: devolve respostas prontas (`when(repo.buscar(1)).thenReturn(cliente)`). Serve para colocar o código testado num cenário.
- **Mock**: além disso, **registra as chamadas**, e o teste verifica a interação (`verify(email).enviar(...)`). Deve ser usado quando o efeito colateral é o próprio comportamento a testar.
- **Spy**: embrulha um objeto **real** e permite sobrescrever alguns métodos e observar as chamadas.

Um **fake** escrito à mão (um repositório em memória com `HashMap`, por exemplo) é melhor quando o dublê precisa de comportamento coerente ao longo de várias chamadas (salvar e depois buscar). Os testes ficam mais legíveis, e o mesmo fake serve a dezenas de testes.

Exercícios de código — soluções

Q83 Testes com JUnit 5

> *Escreva uma classe `Calculadora` com o método `static int dividir(int a, int b)`, e uma classe de testes JUnit 5 que verifique uma divisão normal, a divisão por zero (esperando `ArithmeticException`) e vários casos com `@ParameterizedTest`.*

```
import org.junit.jupiter.api.Test;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.CsvSource;
```

```
import static org.junit.jupiter.api.Assertions.*;
```

```
class Calculadora {
    static int dividir(int a, int b) { return a / b; }
}
```

```
class CalculadoraTest {
    @Test
    void divideInteirosTruncando() {
        assertEquals(3, Calculadora.dividir(7, 2));
    }
```

```
    @Test
    void divisaoPorZeroLancaExcecao() {
        var e = assertThrows(ArithmeticException.class, () -> Calculadora.dividir(1, 0));
        assertEquals("/ by zero", e.getMessage());
    }
```

```
    @ParameterizedTest(name = "{0} / {1} = {2}")
    @CsvSource({"10, 2, 5", "-7, 2, -3", "0, 5, 0", "2147483647, 1, 2147483647"})
    void variosCasos(int a, int b, int esperado) {
        assertEquals(esperado, Calculadora.dividir(a, b));
    }
}
```

O `assertThrows` devolve a exceção, o que permite conferir também a mensagem. O `@ParameterizedTest` com `@CsvSource` transforma cada linha num teste separado no relatório, com o nome definido em `name`. Com AssertJ, a asserção fica mais fluente: `assertThatThrownBy(() -> Calculadora.dividir(1, 0)).isInstanceOf(ArithmeticException.class)`.

Q84 Testes com Mockito

O serviço abaixo envia um e-mail de boas-vindas ao cadastrar um cliente novo, mas não envia se o e-mail já existir. Escreva os testes com Mockito para os dois cenários.

```
import static org.mockito.Mockito.*;
import static org.junit.jupiter.api.Assertions.*;
```

```
class CadastroTest {
    Repositorio repo = mock(Repositorio.class);
    Correio correio = mock(Correio.class);
    Cadastro cadastro = new Cadastro(repo, correio);
```

```
    @Test
    void clienteNovoEhSalvoERecebeEmail() {
        when(repo.existe("ana@ex.com")).thenReturn(false);     // stub: monta o cenário
```

```
        assertTrue(cadastro.cadastrar("ana@ex.com"));
```

```
        verify(repo).salvar("ana@ex.com");                     // mock: o efeito é o
comportamento
        verify(correio).enviar("ana@ex.com", "Bem-vindo!");
    }
```

```
    @Test
    void emailRepetidoNaoSalvaNemEnvia() {
        when(repo.existe("ana@ex.com")).thenReturn(true);
```

```
        assertFalse(cadastro.cadastrar("ana@ex.com"));
```

```
        verify(repo, never()).salvar(anyString());
        verifyNoInteractions(correio);
    }
}
```

Aqui o *mock* se justifica: enviar o e-mail **é** o comportamento a verificar, e ninguém quer um e-mail de verdade saindo nos testes. Note que não se verifica a chamada a `repo.existe`, porque esse é um detalhe de implementação que o *stub* já exercita.

Q85 Dois módulos

Monte um projeto modular com dois módulos: `br.com.exemplo.tarefas` (o modelo, com o pacote `br.com.exemplo.tarefas`) e `br.com.exemplo.tarefas.app` (a aplicação que usa o modelo). Escreva os dois `module-info.java` e os comandos `javac` e `java` para compilar e executar.

```
// src/br.com.exemplo.tarefas/module-info.java
module br.com.exemplo.tarefas {
    exports br.com.exemplo.tarefas;            // só este pacote é visível para fora
}
```

```
// src/br.com.exemplo.tarefas.app/module-info.java
module br.com.exemplo.tarefas.app {
    requires br.com.exemplo.tarefas;
}
```

```
# compila todos os módulos de uma vez (modo multimódulo)
javac -d out --module-source-path src $(find src -name "*.java")
```

```
# executa: caminho de módulos + módulo/classe principal
java --module-path out -m br.com.exemplo.tarefas.app/br.com.exemplo.tarefas.app.Principal
```

Com `--module-source-path`, o `javac` espera uma pasta por módulo, com o mesmo nome do módulo. Se o `app` tentar usar um pacote interno do modelo que não foi exportado, o erro sai na **compilação**, e não em produção. `java --list-modules` e `jdeps` ajudam a inspecionar as dependências.

FASE 6

Por dentro da JVM

Artigos 33 a 36

#33 A Máquina por Trás da Máquina — Class Loading, Pilha, Heap e JIT
#34 Quem Recolhe o Lixo — os Coletores G1 e ZGC
#35 Medir Antes de Otimizar — JFR, Mission Control e JMH
#36 Ponte para o Nativo — a Foreign Function & Memory API

Perguntas conceituais

Q86 Class loading

*Descreva as três etapas do **carregamento de classes** (carregar, ligar e inicializar). Quando exatamente o bloco `static { ... }` de uma classe é executado?*

1. Carregar: um *class loader* encontra os bytes da classe (no JAR, no módulo, na rede) e cria o objeto `Class`. A delegação vai do *bootstrap* (o núcleo do JDK) ao *platform* e ao *application*.

2. Ligar: **verificar** o bytecode (tipos, pilha, saltos válidos), **preparar** os campos estáticos com valores padrão e, de forma preguiçosa, **resolver** as referências simbólicas a outras classes.

3. Inicializar: executar os inicializadores estáticos e os blocos `static`, na ordem do código.

A inicialização acontece **na primeira vez que a classe é usada de verdade**: a primeira instância criada, a primeira chamada a um método estático, ou o primeiro acesso a um campo estático que não seja uma constante de tempo de compilação. Por isso o idiom *holder* (`class Holder { static final X INSTANCIA = new X(); }`) cria um *singleton* preguiçoso e seguro entre *threads* sem nenhum `synchronized`.

Q87 Pilha e heap

*O que fica na **pilha** de uma thread e o que fica no **heap**? Por que uma variável local nunca tem problema de concorrência, mas o objeto para o qual ela aponta pode ter?*

Cada *thread* tem sua **pilha**, com um quadro por chamada em andamento. O quadro guarda os parâmetros, as variáveis locais (primitivos e referências) e a pilha de operandos. O **heap** é único e compartilhado: todos os objetos e arrays moram lá.

Uma variável local está no quadro de uma única *thread*, e ninguém mais a enxerga. Mas, se ela guarda uma referência, o **objeto** apontado está no heap, e outra *thread* que tenha a mesma referência pode mexer nele ao mesmo tempo. Daí a necessidade de sincronização ou de imutabilidade.

Q88 Análise de escape

*O que é **análise de escape** (escape analysis)? Dê um exemplo de objeto que o JIT pode eliminar completamente.*

O compilador C2 verifica se um objeto criado num método **escapa** dele: se é guardado num campo, devolvido, passado para um método não embutido (*inlined*) ou visto por outra *thread*. Se não escapa, o JIT pode aplicar a **substituição escalar**: em vez de alocar o objeto no heap, ele guarda os campos em registradores ou variáveis locais, e o objeto nunca chega a existir.

Exemplo: `double distancia(double x, double y) { var p = new Ponto(x, y); return Math.sqrt(p.x() * p.x() + p.y() * p.y()); }`. O `Ponto` morre ali mesmo, e depois de aquecido o método não aloca nada. Foi assim que o Asteroids da Trilha C alocava só 74 bytes por passo, apesar de todos os `new ArrayList` do código.

Q89 Hipótese geracional

*Explique a **hipótese geracional** e por que ela torna a coleta de lixo barata para objetos de vida curta.*

A hipótese observa que **a maioria dos objetos morre jovem**: iteradores, *strings* temporárias, DTOs de uma requisição. Por isso o heap é dividido em uma geração **jovem** e uma **velha**. A coleta da geração jovem copia só os **sobreviventes** (poucos) para outra área e descarta o resto de uma vez, sem visitar cada objeto morto. O custo é proporcional aos objetos **vivos**, e não ao lixo. Objetos que sobrevivem a várias coletas são promovidos para a geração velha, que é coletada com menos frequência. É por isso que alocar muito, em Java, costuma ser barato, e o problema aparece quando objetos vivem "um pouco demais" e são promovidos antes de morrer.

Q90 G1 e ZGC

*Compare o **G1** e o **ZGC**: como cada um trata as pausas? Em que tipo de aplicação você escolheria cada um?*

- **G1** (o padrão): divide o heap em regiões e coleta primeiro as que têm mais lixo (*garbage first*). As coletas da geração jovem são pausas *stop-the-world* curtas, com uma meta ajustável (`-XX:MaxGCPauseMillis`, 200 ms por padrão), e a marcação da geração velha é concorrente. É um bom equilíbrio entre vazão, latência e uso de memória, e serve à maioria das aplicações.
- **ZGC**: faz quase tudo **concorrentemente**, inclusive a relocação dos objetos, usando barreiras de carga e ponteiros coloridos. As pausas ficam abaixo de 1 ms, independentemente do tamanho do heap (de MB a TB). É geracional desde o JDK 21, e o padrão do ZGC desde o 23. Custa um pouco mais de CPU e de memória, e não usa *compressed oops*.

Escolha o ZGC quando a latência de cauda importa: APIs com metas de p99, jogos, *trading*, heaps enormes. Nos testes da Trilha C, a maior pausa do ZGC foi de 0,021 ms, contra 8,5 ms do G1.

Q91 Vazamentos em Java

*Um programa Java pode ter **vazamento de memória**, mesmo com coletor de lixo? Dê dois exemplos clássicos.*

Pode. O coletor libera o que é **inalcançável**, e um vazamento em Java é um objeto que continua alcançável sem ser mais útil.

- Um **cache estático** (`static Map`) que só cresce, sem limite nem expiração.

- **Ouvintes** (*listeners*) registrados e nunca removidos: o objeto que ouve continua preso à fonte de eventos.
- Um `ThreadLocal` em *pools* de *threads*: a *thread* nunca morre, e o valor fica para sempre.
- Chaves mutáveis em `HashMap`, que ficam "perdidas" e inacessíveis dentro do mapa.

O sintoma é o heap que cresce depois de cada coleta completa. O diagnóstico é um *heap dump* (`jcmd <pid> GC.heap_dump`) analisado no Eclipse MAT ou no JDK Mission Control.

Q92 Por que usar JMH

> *Por que um micro-benchmark escrito com `System.nanoTime()` num laço costuma mentir? Cite três armadilhas que o **JMH** resolve.*

- **Aquecimento**: as primeiras iterações são interpretadas, e as últimas, compiladas pelo C2. A média mistura dois programas diferentes. O JMH roda iterações de aquecimento separadas.
- **Eliminação de código morto**: se o resultado não é usado, o JIT pode apagar a conta inteira, e o *benchmark* mede o nada. O JMH usa o `Blackhole` e o retorno dos métodos.
- **Dobramento de constantes**: entradas constantes permitem que o JIT calcule o resultado durante a compilação. O JMH usa campos de estado (`@State`).
- Também há o *on-stack replacement* num laço longo, a interferência do coletor e a variação entre execuções. O JMH usa *forks* separados e reporta o erro estatístico.

Q93 JFR

> *O que o **JDK Flight Recorder** (JFR) registra, e por que ele pode ficar ligado em produção? Escreva o comando para gravar 60 segundos de um programa já em execução.*

O JFR grava **eventos** de dentro da JVM: amostras de CPU por método, alocações (com a pilha de onde vieram), pausas do coletor, bloqueios em *locks*, E/S de arquivo e rede, compilações do JIT, entre outros. Os eventos são gravados em *buffers* por *thread*, com custo em torno de 1% na configuração padrão, e por isso ele foi feito para ficar ligado em produção.

Num processo já em execução: `jcmd <pid> JFR.start duration=60s filename=gravacao.jfr`. Ao iniciar: `java -XX:StartFlightRecording=duration=60s,filename=gravacao.jfr ...`. O arquivo se abre no JDK Mission Control ou com `jfr print --events jdk.ExecutionSample gravacao.jfr`.

Q94 FFM API

> *Na **FFM API** (Foreign Function & Memory, final no Java 22), para que servem o `Arena`, o `MemorySegment` e o `Linker`? Por que ela é mais segura que o JNI?*

- `MemorySegment`: uma região de memória (dentro ou **fora** do heap) com **limites** e **tempo de vida** conhecidos. Um acesso fora dos limites lança exceção, em vez de corromper a memória.
- `Arena`: controla o tempo de vida dos segmentos. Ao fechar a arena (num `try` com recursos), toda a memória que ela alocou é liberada de uma vez, e qualquer acesso posterior dá erro, em vez de ler lixo.
- `Linker`: faz a ponte para funções nativas. Com um `FunctionDescriptor` (a assinatura em termos de C), ele gera um `MethodHandle` que chama a função C direto do Java.

Em relação ao JNI: não é preciso escrever nem compilar código C de "cola", os erros de tipo e de limite viram exceções Java, e a memória tem dono. O acesso nativo continua sendo um método "restrito", e por isso o Java 25 avisa, e pede `--enable-native-access`.

Exercícios de código — soluções

Q95 strlen via FFM

Use a FFM API para chamar a função `strlen` da biblioteca C padrão com a string "Olá, FFM!" e explique por que o resultado não é 9.

```
import java.lang.foreign.*;
import java.lang.invoke.MethodHandle;
```

```
void main() throws Throwable {
    Linker linker = Linker.nativeLinker();
    MethodHandle strlen = linker.downcallHandle(
            linker.defaultLookup().find("strlen").orElseThrow(),
            FunctionDescriptor.of(ValueLayout.JAVA_LONG, ValueLayout.ADDRESS));   // size_t
strlen(const char*)
```

```
    try (Arena arena = Arena.ofConfined()) {                  // a memória nativa vive até o fim
do try
        MemorySegment texto = arena.allocateFrom("Olá, FFM!");    // copia em UTF-8 e acrescenta
o \0
        long n = (long) strlen.invokeExact(texto);
        IO.println("strlen = " + n + " bytes; length() = " + "Olá, FFM!".length() + "
caracteres");
    }
}
```

Saída

```
strlen = 10 bytes; length() = 9 caracteres
```

O `strlen` conta **bytes** até o terminador nulo, e o `allocateFrom` codifica a *string* em UTF-8, em que o "á" ocupa 2 bytes. O `length()` do Java conta unidades UTF-16. Rode com `java --enable-native-access=ALL-UNNAMED Strlen.java` para evitar o aviso de método restrito.

Q96 Provocando e diagnosticando um OOM

Escreva um programa que provoque um `OutOfMemoryError` controlado, guardando arrays de 1 MB numa lista estática, e execute-o com `-Xmx64m`. Depois explique como o `-XX:+HeapDumpOnOutOfMemoryError` ajudaria a encontrar a causa num sistema real.

```
static final List<byte[]> CACHE = new ArrayList<>();       // o "cache" que só cresce
```

```
void main() {
    try {
        while (true) CACHE.add(new byte[1024 * 1024]);    // 1 MB por vez, sempre alcançável
    } catch (OutOfMemoryError e) {
        int mb = CACHE.size();
        CACHE.clear();                                     // libera para conseguir imprimir
        IO.println("OutOfMemoryError depois de " + (mb > 20 ? "mais de 20" : String.valueOf(mb))
+ " MB: " + e.getMessage());
    }
}
```

Saída

```
OutOfMemoryError depois de mais de 20 MB: Java heap space
```

(Execute com `java -Xmx64m Vazamento.java`. O número exato de MB varia com o coletor e a JVM. Pegar um `OutOfMemoryError`, como aqui, só faz sentido em demonstrações.)

Num sistema real, `-XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/tmp` grava uma fotografia do heap no instante da falha. Abrindo o arquivo no Eclipse MAT, o relatório de "maiores retentores" mostraria que quase todo o heap está preso à lista `CACHE`, e o caminho até as raízes do coletor mostra quem a mantém viva (aqui, o campo estático).

FASE 7

Concorrência

Artigos 37 a 43

#37 Fazendo Duas Coisas ao Mesmo Tempo — Threads
#38 O Cadeado e a Memória — synchronized, volatile e o Java Memory Model
#39 Não Crie Threads, Entregue Tarefas — ExecutorService e Futures
#40 Ferramentas de Precisão — Locks, Atomics e Coleções Concorrentes
#41 Encadeando o que Ainda Não Aconteceu — CompletableFuture
#42 Um Milhão de Threads — Virtual Threads
#43 Tarefas em Família — Structured Concurrency e Scoped Values

Perguntas conceituais

Q97 run, start e join

Qual é a diferença entre chamar `thread.run()` e `thread.start()`? E o que `join()` faz?

`start()` pede à JVM uma **nova** *thread* de execução, e é ela que chama o `run()`. O `start()` volta na hora, e as duas *threads* seguem em paralelo. Chamar `run()` diretamente é só uma chamada de método comum, na *thread* atual, sem concorrência nenhuma. É um erro clássico, porque "funciona", só que sequencialmente. Chamar `start()` duas vezes na mesma `Thread` lança `IllegalThreadStateException`.

`join()` faz a *thread* atual **esperar** até a outra terminar, e também garante que tudo o que a outra escreveu fique visível depois do retorno (uma relação *happens-before*).

Q98 Condição de corrida

*O que é uma **condição de corrida**? Por que `contador++` não é uma operação atômica, mesmo sendo uma linha só?*

Uma condição de corrida acontece quando o resultado depende da **ordem imprevisível** em que as *threads* intercalam suas operações sobre dados compartilhados. `contador++` são **três** passos: ler o valor, somar 1 e gravar. Duas *threads* podem ler o mesmo 41, somar e gravar 42 as duas, e um incremento se perde. Com milhões de incrementos, o total final sai sempre menor que o esperado, e diferente a cada execução. As soluções são `synchronized`, `AtomicInteger` / `LongAdder` ou não compartilhar o estado (cada *thread* conta o seu, e as contagens são somadas no fim).

Q99 synchronized e volatile

*Explique as duas garantias do `synchronized`: **exclusão mútua** e **visibilidade**. Por que só `volatile` não resolve um contador compartilhado?*

- **Exclusão mútua**: só uma *thread* por vez executa um bloco protegido pelo mesmo monitor (o mesmo objeto), e a sequência ler-somar-gravar vira indivisível.
- **Visibilidade**: tudo o que uma *thread* escreveu antes de **liberar** o monitor fica visível para a próxima que o **adquirir** (*happens-before*). Sem isso, a outra *thread* poderia ler valores velhos de cache ou

reordenados pelo compilador.

`volatile` dá **só a visibilidade** (e impede certas reordenações): a leitura sempre vê a última escrita. Mas `contador++` num campo `volatile` continua sendo ler-somar-gravar, e dois incrementos simultâneos ainda se perdem. O `volatile` serve para *flags* (`volatile boolean parar`) e para publicar objetos imutáveis.

Q100 Deadlock

> *Quais são as quatro condições necessárias para um **deadlock**? Qual delas é a mais prática de quebrar, e como?*

1. Exclusão mútua: o recurso só pode ter um dono.

2. Posse e espera: quem tem um recurso espera por outro sem soltar o primeiro.

3. Não preempção: ninguém tira o recurso à força.

4. Espera circular: A espera B, que espera A.

A mais prática de quebrar é a **espera circular**: definir uma **ordem global** de aquisição dos *locks*. Numa transferência entre contas, por exemplo, sempre travar primeiro a conta de menor id, qualquer que seja o sentido. Outra saída é o `tryLock` com prazo do `ReentrantLock`, que desiste e tenta de novo, em vez de esperar para sempre.

Q101 Por que executores

> *Por que criar uma `new Thread()` para cada tarefa é má ideia com threads de plataforma? O que um `ExecutorService` resolve, e por que ele precisa ser **desligado**?*

Cada *thread* de plataforma é uma *thread* do sistema operacional, com cerca de 1 MB de pilha reservado e custo de criação. Uma *thread* por tarefa, sob carga, esgota a memória ou o limite do sistema. O `ExecutorService` separa **o que fazer** (tarefas) de **como executar** (um *pool* de *threads* reaproveitadas, com fila, limite e política para quando a fila enche).

As *threads* do *pool* não morrem sozinhas: sem `shutdown()`, elas mantêm a JVM viva e o programa não termina. Desde o Java 19 o `ExecutorService` é `AutoCloseable`, e o `close()` desliga e espera as tarefas terminarem. Por isso, use `try (var executor = ...) { ... }`.

Q102 Fila cheia e rejeição

> *Um `ThreadPoolExecutor` foi configurado com 4 threads e uma fila **limitada** de 100 posições. O que acontece com a tarefa número 105, se todas estão ocupadas? Quais são as políticas de rejeição disponíveis?*

Com as 4 *threads* ocupadas e as 100 vagas da fila preenchidas (sem poder criar mais *threads*, porque `maximumPoolSize` = 4), a tarefa 105 é **rejeitada**, e a política de rejeição decide o destino:

- `AbortPolicy` (padrão): lança `RejectedExecutionException`;

- `CallerRunsPolicy`: a própria *thread* que submeteu executa a tarefa. Isso freia naturalmente o produtor (*backpressure*);
- `DiscardPolicy`: descarta em silêncio. Perigoso;
- `DiscardOldestPolicy`: descarta a tarefa mais antiga da fila e tenta de novo.

Uma fila **ilimitada** nunca rejeita, e sob sobrecarga cresce até esgotar a memória. É um dos motivos para evitar o `Executors.newFixedThreadPool` em servidores sem pensar.

Q103 Atomics e mapas concorrentes

> *Quando usar `AtomicLong` e quando usar `LongAdder`? E por que `ConcurrentHashMap.merge(chave, 1, Integer::sum)` é seguro, mas `mapa.put(chave, mapa.get(chave) + 1)` não é, mesmo num `ConcurrentHashMap`?*

`AtomicLong` faz operações atômicas com CAS (*compare-and-swap*) num único valor. Sob muita disputa, as *threads* ficam repetindo o CAS que falhou. `LongAdder` espalha a contagem em várias células, uma por *thread* disputando, e só soma tudo quando se lê o valor. É muito mais rápido para contadores muito disputados (métricas, estatísticas), mas não serve quando se precisa do valor exato a cada operação, como para gerar ids.

No `ConcurrentHashMap`, cada operação isolada é atômica, mas `get` seguido de `put` são **duas** operações: entre elas, outra *thread* pode ter mudado o valor, e um incremento se perde. `merge`, `compute` e `computeIfAbsent` fazem a leitura e a escrita **como uma única operação atômica** sobre aquela chave.

Q104 thenApply, thenCompose e erros

> *Qual é a diferença entre `thenApply` e `thenCompose` num `CompletableFuture`? E como tratar um erro que aconteceu em qualquer etapa da cadeia?*

`thenApply(f)` transforma o resultado com uma função **síncrona**: `T -> U`. `thenCompose(f)` encadeia uma função que devolve **outro** `CompletableFuture` (`T -> CompletableFuture<U>`) e o "achata". Com `thenApply`, o resultado seria um `CompletableFuture<CompletableFuture<U>>`. É o `flatMap` dos futuros.

Um erro em qualquer etapa pula as etapas seguintes até um tratador:

- `exceptionally(e -> valorPadrao)`: recupera com um valor;
- `handle((valor, erro) -> ...)`: trata os dois casos;
- `whenComplete(...)`: observa sem alterar o resultado.

O erro chega embrulhado numa `CompletionException`, e a causa real está em `getCause()`.

Q105 Virtual threads

> *O que é uma **virtual thread**? Por que ela permite um milhão de conexões simultâneas? Em que tipo de carga ela **não** traz ganho?*

É uma *thread* gerenciada pela JVM, e não pelo sistema operacional. Ela custa algumas centenas de bytes a poucos KB (a pilha cresce e encolhe no heap) e roda "montada" sobre um pequeno conjunto de *threads* de plataforma, as **portadoras**. Quando ela bloqueia em E/S (*socket*, arquivo, `sleep`, `lock`), a JVM a **desmonta**,

guarda a pilha no heap e libera a portadora para outra. Assim, um milhão de conexões esperando a rede ocupam memória, e não *threads* do sistema. O estilo de código continua o simples e bloqueante, uma *thread* por requisição.

Não há ganho em carga **CPU-bound**: os núcleos são os mesmos, e mil *threads* virtuais calculando não calculam mais rápido que 8 de plataforma. Também não faz sentido criar um *pool* de *threads* virtuais: elas são baratas, então crie uma por tarefa.

Q106 Pinning

> *O que é o pinning de uma virtual thread? O que mudou no Java 24 em relação ao `synchronized`?*

Pinning é quando a *virtual thread* **não pode ser desmontada** da portadora ao bloquear. A portadora fica presa, e com poucas portadoras (uma por núcleo) o sistema inteiro pode travar. Até o Java 23, bloquear **dentro de um bloco** `synchronized` causava *pinning*, e por isso a recomendação era trocar `synchronized` por `ReentrantLock` em código com E/S. O Java 24 (JEP 491) resolveu esse caso: agora as *virtual threads* podem ser desmontadas dentro de `synchronized`. Continua havendo *pinning* ao bloquear dentro de código nativo (JNI/FFM).

Q107 Concorrência estruturada

> *Que problemas a **concorrência estruturada** (`StructuredTaskScope`) resolve em relação a submeter duas tarefas num `ExecutorService` e chamar `get()` nelas?*

Com o `ExecutorService` solto:

- se a primeira tarefa falha, a segunda **continua rodando** à toa (vazamento de trabalho);
- se quem esperava é cancelado ou interrompido, as tarefas filhas não ficam sabendo;
- o *thread dump* não mostra quem é filha de quem.

A concorrência estruturada trata as subtarefas como um **bloco**, como um `try`: elas nascem e morrem dentro do escopo. Na política "todas precisam dar certo", a primeira falha **cancela as outras** automaticamente. O cancelamento de quem espera se propaga para as filhas, e o `join()` só volta quando todas terminaram. No Java 25 ela está em *preview* (JEP 505), com a API `StructuredTaskScope.open()` e os *joiners*, e no JDK 27 continua em *preview*.

Q108 Scoped values

> *O que é um `ScopedValue` e por que ele é preferível a um `ThreadLocal` com virtual threads?*

Um `ScopedValue` (final no Java 25, JEP 506) guarda um valor **imutável**, visível por todo o código chamado dentro de um escopo: `ScopedValue.where(USUARIO, u).run(() -> tratar())`. Quando o `run` termina, o valor some sozinho. Ele é herdado pelas subtarefas de um `StructuredTaskScope`.

Em relação ao `ThreadLocal`:

- **não é mutável**: não há `set()` perdido no meio do código mudando o valor escondido;
- **tempo de vida limitado**: não há vazamento por esquecer o `remove()`, algo comum em *pools*;

- **barato**: com milhões de *virtual threads*, cada uma com cópias de vários `ThreadLocal` herdáveis, a memória explode, e o `ScopedValue` é compartilhado sem cópia.

Exercícios de código — soluções

Q109 Corrida e correção

Demonstre a perda de incrementos: 4 threads somam 1 num `int` comum 250.000 vezes cada. Depois corrija com `AtomicInteger` e com `LongAdder`, e mostre os três totais.

```
import java.util.concurrent.atomic.*;
```

```
static int comum = 0;                                          // sem proteção nenhuma
```

```
void main() throws InterruptedException {
    var atomico = new AtomicInteger();
    var somador = new LongAdder();
    List<Thread> threads = new ArrayList<>();
    for (int t = 0; t < 4; t++) {
        threads.add(Thread.ofPlatform().start(() -> {
            for (int i = 0; i < 250_000; i++) {
                comum++;                                       // ler, somar, gravar: pode perder
                atomico.incrementAndGet();                     // CAS: nunca perde
                somador.increment();                           // células por thread: nunca perde
            }
        }));
    }
    for (Thread t : threads) t.join();                         // join também garante a
visibilidade
    IO.println("comum " + (comum < 1_000_000 ? "< 1000000 (perdeu incrementos)" : "= 1000000
(teve sorte)"));
    IO.println("atomico = " + atomico.get() + ", somador = " + somador.sum());
}
```

Saída

```
comum < 1000000 (perdeu incrementos)
atomico = 1000000, somador = 1000000
```

O total do `int` comum muda a cada execução e quase sempre fica abaixo de um milhão (numa máquina com um só núcleo, raramente pode acertar). Os outros dois são sempre exatos.

Q110 Dez mil tarefas bloqueantes

Com um `ExecutorService` de virtual threads, dispare 10.000 tarefas que "esperam a rede" por 100 ms (`Thread.sleep`) e devolvem o próprio número. Some os resultados e meça o tempo total. Explique por que leva perto de 100 ms, e não 1.000 segundos.

```
import java.util.concurrent.*;
```

```
void main() throws Exception {
    long inicio = System.nanoTime();
    long soma = 0;
    try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {   // close() espera todas
        List<Future<Integer>> futuros = new ArrayList<>();
        for (int i = 1; i <= 10_000; i++) {
            int n = i;
            futuros.add(executor.submit(() -> {
                Thread.sleep(100);                                       // "E/S": a portadora é
liberada
                return n;
            }));
        }
        for (Future<Integer> f : futuros) soma += f.get();
    }
    long ms = (System.nanoTime() - inicio) / 1_000_000;
    IO.println("soma = " + soma + (ms < 2_000 ? " em menos de 2 s" : " (lento: " + ms + " ms)"));
}
```

Saída

```
soma = 50005000 em menos de 2 s
```

Cada tarefa passa quase todo o tempo **bloqueada**. Ao bloquear, a *virtual thread* é desmontada e a portadora atende outra. As 10.000 esperas acontecem **ao mesmo tempo**, e o total fica em pouco mais de 100 ms, mais o custo de criar e agendar as tarefas. Sequencialmente seriam 1.000 s. Com um *pool* fixo de 8 *threads* de plataforma, seriam cerca de 125 s.

Q111 Combinando futuros

*Encadeie com `CompletableFuture`: buscar o preço de um produto (assíncrono), buscar a cotação do dólar (assíncrono, **em paralelo**), combinar os dois para obter o preço em dólares, e cair num valor padrão se qualquer busca falhar.*

```
import java.util.concurrent.*;
```

```
CompletableFuture<Double> precoEmReais(String produto) {
    return CompletableFuture.supplyAsync(() -> {
        if (produto.isBlank()) throw new IllegalArgumentException("produto vazio");
        return 250.0;
    });
}
```

```
CompletableFuture<Double> cotacaoDoDolar() {
    return CompletableFuture.supplyAsync(() -> 5.0);
}
```

```
String precoEmDolares(String produto) {
    return precoEmReais(produto)
            .thenCombine(cotacaoDoDolar(), (reais, cotacao) -> reais / cotacao)   // os dois
rodam em paralelo
            .thenApply(d -> "US$ %.2f".formatted(d))
            .exceptionally(e -> "indisponível (" + e.getCause().getMessage() + ")")
            .join();
}
```

```
void main() {
    IO.println(precoEmDolares("caderno"));
    IO.println(precoEmDolares(""));
}
```

Saída
```
US$ 50.00
indisponível (produto vazio)
```

`thenCombine` junta dois futuros **independentes**, que já estão rodando em paralelo. Se um dependesse do resultado do outro, seria `thenCompose`. O `exceptionally` recebe a `CompletionException`, e a mensagem original está na causa.

Q112 Transferência sem deadlock

> *Escreva uma transferência entre contas que **não** entre em deadlock quando duas threads transferem em sentidos opostos ao mesmo tempo (A→B e B→A), usando ordem global de locks.*

```
static final class Conta {
    final int id;
    long saldo;
    Conta(int id, long saldo) { this.id = id; this.saldo = saldo; }
}
```

```
void transferir(Conta de, Conta para, long valor) {
    Conta primeira = de.id < para.id ? de : para;              // ordem global: sempre o menor id
primeiro
    Conta segunda = de.id < para.id ? para : de;
    synchronized (primeira) {
        synchronized (segunda) {
            if (de.saldo >= valor) {
                de.saldo -= valor;
                para.saldo += valor;
            }
        }
    }
}
```

```
void main() throws InterruptedException {
    Conta a = new Conta(1, 1_000_000), b = new Conta(2, 1_000_000);
    Thread t1 = Thread.ofPlatform().start(() -> { for (int i = 0; i < 100_000; i++) transferir(a,
b, 1); });
    Thread t2 = Thread.ofPlatform().start(() -> { for (int i = 0; i < 100_000; i++) transferir(b,
a, 1); });
    t1.join();
    t2.join();
    IO.println("a = " + a.saldo + ", b = " + b.saldo + ", total = " + (a.saldo + b.saldo));
}
```

Saída

```
a = 1000000, b = 1000000, total = 2000000
```

Se cada *thread* travasse primeiro a conta de origem, `t1` pegaria `a` e esperaria `b`, enquanto `t2` pegaria `b` e esperaria `a`: espera circular e *deadlock*. Com a ordem por id, as duas travam `a` primeiro, e uma simplesmente espera a outra. O total conservado mostra que nenhuma transferência foi corrompida.

Q113 StructuredTaskScope

> *Com `StructuredTaskScope` (preview no Java 25), busque ao mesmo tempo os dados de um usuário e os pedidos dele. Se uma das buscas falhar, a outra deve ser cancelada. Mostre o código e o comando para executá-lo.*

```
import java.util.concurrent.StructuredTaskScope;
```

```
record Usuario(long id, String nome) {}
record Painel(Usuario usuario, List<String> pedidos) {}
```

```
Usuario buscarUsuario(long id) throws InterruptedException {
    Thread.sleep(50);
    return new Usuario(id, "Ana");
}
```

```
List<String> buscarPedidos(long id) throws InterruptedException {
    Thread.sleep(80);
    if (id < 0) throw new IllegalStateException("serviço de pedidos fora do ar");
    return List.of("pedido-1", "pedido-2");
}
```

```
Painel painel(long id) throws InterruptedException {
    try (var escopo = StructuredTaskScope.open()) {                 // política padrão: todas
precisam dar certo
        var usuario = escopo.fork(() -> buscarUsuario(id));
        var pedidos = escopo.fork(() -> buscarPedidos(id));
        escopo.join();                                              // espera as duas; falha de
uma cancela a outra
        return new Painel(usuario.get(), pedidos.get());
    }
}
```

```
void main() throws InterruptedException {
    IO.println(painel(7));
    try {
        painel(-1);
    } catch (StructuredTaskScope.FailedException e) {
        IO.println("falhou: " + e.getCause().getMessage());
    }
}
```

Saída

```
Painel[usuario=Usuario[id=7, nome=Ana], pedidos=[pedido-1, pedido-2]]
falhou: serviço de pedidos fora do ar
```

Execute com `java --enable-preview Estruturada.java` (o lançador de código-fonte aceita `--enable-preview` sem `--source`). As subtarefas rodam em *virtual threads* e não "escapam" do bloco `try`: quando ele termina, todas terminaram. A API mudou entre as versões de *preview*, e esta é a do Java 25.

FASE 8

Qualidade e distribuição

Artigos 44 a 46

#44 Lições de Quem Veio Antes — Effective Java, Código Limpo e Padrões
#45 Contratos à Vista — Imutabilidade, Programação Defensiva e JSpecify
#46 Do JAR ao Executável — jlink, jpackage e GraalVM Native Image

Perguntas conceituais

Q114 Composição em vez de herança

Por que Effective Java recomenda "prefira composição a herança"? Dê um exemplo em que herdar de uma classe da biblioteca quebra de forma sutil.

A herança acopla a subclasse aos **detalhes de implementação** da superclasse, que podem mudar em qualquer versão. O exemplo clássico do livro: um `HashSet` que conta quantos elementos foram adicionados, sobrescrevendo `add` e `addAll` para somar. Como o `addAll` do `HashSet` chama `add` internamente, cada elemento é contado **duas vezes**, e basta uma mudança interna para o comportamento mudar de novo.

Com composição, a classe **tem** um `Set` e repassa as chamadas (um *wrapper*, ou decorador). Ela só depende da interface pública e funciona com qualquer implementação de `Set`. Herança fica para relações "é um" de verdade, dentro de código que você controla, ou em classes projetadas e documentadas para serem estendidas.

Q115 Builder

*Quando um **builder** é melhor que um construtor? E quando um record com um construtor comum já basta?*

O builder vale a pena quando há **muitos parâmetros**, principalmente opcionais ou do mesmo tipo, e a chamada `new Pizza("G", true, false, true, 2, null)` fica ilegível e fácil de errar. `Pizza.builder().tamanho("G").comBorda().queijos(2).build()` se lê como uma frase, e o `build()` pode validar tudo de uma vez antes de criar um objeto imutável.

Com poucos componentes, todos obrigatórios, um *record* com construtor compacto é mais simples e igualmente seguro. Um meio-termo são métodos *wither* (`comPrazo(...)`) no próprio *record*, como os do `Limites` do Farol.

Q116 Programação defensiva

*O que é **programação defensiva**? Explique as cópias defensivas "na entrada" e "na saída" e o princípio fail-fast.*

É escrever cada classe supondo que quem a usa pode errar, ou agir de má-fé, sem que isso quebre as invariantes dela.

- **Cópia na entrada**: o construtor copia coleções e objetos mutáveis recebidos (`List.copyOf(itens)`), para que quem os passou não possa alterá-los depois.
- **Cópia na saída**: os *getters* não devolvem a referência interna mutável. Devolvem uma cópia ou uma visão imutável.
- **Fail-fast**: validar os argumentos **na entrada** (`Objects.requireNonNull`, faixas de valores) e falhar na hora, com uma mensagem clara, em vez de guardar um estado inválido que explode muito longe da causa.

Q117 JSpecify

> *O que as anotações do **JSpecify** (`@NullMarked`, `@Nullable`) acrescentam ao código? Elas mudam o comportamento em tempo de execução?*

O JSpecify é um padrão comum de anotações de nulidade, adotado por Spring, Guava e outros. Com `@NullMarked` num pacote ou módulo, **tudo passa a ser não nulo por padrão**, e só o que tiver `@Nullable` pode receber ou devolver `null`. O contrato fica visível na assinatura, e não enterrado no Javadoc.

Elas **não mudam nada em tempo de execução**: não geram verificações. Quem usa as anotações são as ferramentas de análise (NullAway, o IntelliJ, o compilador do Kotlin), que apontam em tempo de compilação o lugar onde um `null` pode vazar.

Q118 jlink, jpackage e Native Image

> *Qual é a diferença entre o resultado do `jlink`, do `jpackage` e do **GraalVM Native Image**? Quando cada um é a escolha certa?*

- **jlink**: monta um **runtime Java sob medida**, só com os módulos de que a aplicação precisa (dezenas de MB em vez de centenas). Ainda é a JVM normal, com JIT e coletor. Serve para imagens Docker enxutas e para distribuir servidores, como o Farol.
- **jpackage**: empacota a aplicação com um runtime (de preferência feito com `jlink`) num **instalador ou executável nativo do sistema** (`.deb`, `.rpm`, `.msi`, `.exe`, `.dmg` ou uma pasta *app-image*). Serve para aplicações de desktop e jogos, que o usuário instala com um clique.
- **Native Image**: compila **antecipadamente** (AOT) a aplicação inteira num binário nativo, sem JVM. Inicia em milissegundos e usa pouca memória, mas exige declarar a reflexão usada, não tem JIT (o desempenho de pico pode ser menor) e compila devagar. Serve para funções *serverless*, ferramentas de linha de comando e microsserviços que escalam do zero.

Q119 CDS e Leyden

> *O que a opção `--generate-cds-archive` do `jlink` faz, e que problema ela ataca? Como o Projeto Leyden continua essa ideia?*

O **CDS** (*Class Data Sharing*) grava, num arquivo, as classes já carregadas, verificadas e pré-processadas. Na próxima execução, a JVM mapeia esse arquivo na memória em vez de carregar e verificar tudo de novo, e a **inicialização** fica mais rápida (e, com vários processos, a memória é compartilhada). O `jlink --generate-cds-archive` gera esse arquivo para o runtime sob medida.

O Projeto Leyden estende a ideia de guardar trabalho feito numa execução de treino: o **cache AOT** (JEP 483, Java 24) também guarda classes já ligadas, e o Java 25 acrescentou os perfis de métodos (JEP 515), para o JIT aquecer mais cedo. Tudo isso sem abrir mão da JVM completa, ao contrário do Native Image.

Exercícios de código — soluções

Q120 Builder de um record

> *Implemente um builder para `record Pedido(String cliente, List<String> itens, boolean urgente, String observacao)`, com cliente obrigatório, pelo menos um item, `urgente` falso e `observacao` vazia por padrão, validando tudo no `build()`.*

```
record Pedido(String cliente, List<String> itens, boolean urgente, String observacao) {
    Pedido {
        Objects.requireNonNull(cliente, "cliente");
        itens = List.copyOf(itens);                                  // cópia defensiva e
imutável
        if (itens.isEmpty()) throw new IllegalArgumentException("pedido sem itens");
    }
```

```
    static Construtor para(String cliente) { return new Construtor(cliente); }
```

```
    static final class Construtor {
        private final String cliente;
        private final List<String> itens = new ArrayList<>();
        private boolean urgente;
        private String observacao = "";
```

```
        private Construtor(String cliente) { this.cliente = cliente; }
        Construtor item(String item) { itens.add(item); return this; }
        Construtor urgente() { urgente = true; return this; }
        Construtor observacao(String texto) { observacao = texto; return this; }
        Pedido build() { return new Pedido(cliente, itens, urgente, observacao); }   // o record
valida
    }
}
```

```
void main() {
    IO.println(Pedido.para("Ana").item("café").item("pão").urgente().build());
    try {
        Pedido.para("Bia").build();
    } catch (IllegalArgumentException e) {
        IO.println("recusado: " + e.getMessage());
    }
}
```

Saída

```
Pedido[cliente=Ana, itens=[café, pão], urgente=true, observacao=]
recusado: pedido sem itens
```

O construtor compacto do *record* é a **única** porta de criação e concentra as validações: tanto o builder quanto quem chamar `new Pedido(...)` diretamente passam por elas. O builder é mutável, mas vive pouco e nunca é compartilhado.

Q121 Da aplicação ao instalador

> *Escreva os comandos para: (1) descobrir os módulos de que o JAR `app.jar` precisa; (2) gerar com `jlink` um runtime mínimo, sem cabeçalhos, páginas de manual e informações de depuração; (3) criar com `jpackage` um instalador `.deb` usando esse runtime.*

```
# 1) quais módulos do JDK o JAR usa
jdeps --ignore-missing-deps --print-module-deps --multi-release 25 app.jar
# saída de exemplo: java.base,java.net.http,java.sql
```

```
# 2) runtime sob medida
jlink --add-modules java.base,java.net.http,java.sql \
      --strip-debug --no-header-files --no-man-pages \
      --compress zip-6 --generate-cds-archive \
      --output runtime
```

```
# 3) instalador .deb (rode no Linux; para .msi, no Windows; para .dmg, no macOS)
jpackage --type deb --name minha-app --app-version 1.0.0 \
         --input pasta-com-o-jar --main-jar app.jar \
         --runtime-image runtime --dest instaladores
```

O `jpackage` só gera pacotes para o sistema em que está rodando: não há empacotamento cruzado. Para os três sistemas, rode num CI com três máquinas, ou use uma ferramenta como o Construo (visto na Trilha C).

FASE 9

O projeto final: o servidor HTTP Farol

Artigos 47 a 52

#47 O Projeto Final Começa — Arquitetando um Servidor HTTP sobre Sockets
#48 A Primeira Pedra — ServerSocket e a Leitura do HTTP/1.1
#49 A Segunda Pedra — Requisições e Respostas como Tipos: Records, Sealed e Rotas
#50 A Terceira Pedra — Mil Conexões ao Mesmo Tempo: Virtual Threads e Keep-Alive
#51 A Pedra de Acabamento — Arquivos Estáticos, Testes e Empacotamento
#52 A Jornada Completa — Retrospectiva de Um Ano de Java

Perguntas conceituais

Q122 Anatomia de uma requisição

> *Descreva a estrutura de uma requisição HTTP/1.1 como ela chega pelo socket: quais são as partes, como elas se separam, e como o servidor sabe onde o corpo termina?*

1. Linha de pedido: `MÉTODO alvo versão`, por exemplo `GET /tarefas?pagina=2 HTTP/1.1`, terminada em `CRLF` (`\r\n`).

2. Cabeçalhos: linhas `Nome: valor`, cada uma terminada em `CRLF`. O `Host` é obrigatório no HTTP/1.1.

3. Uma **linha vazia** (`CRLF` sozinho), que marca o fim dos cabeçalhos.

4. O **corpo**, opcional. O fim dele é dado pelo cabeçalho `Content-Length` (número exato de bytes) **ou** por `Transfer-Encoding: chunked` (pedaços precedidos pelo tamanho em hexadecimal, até um pedaço de tamanho 0). Sem nenhum dos dois, uma requisição não tem corpo.

O texto dos cabeçalhos é ASCII, e o corpo é feito de bytes. Por isso o Farol lê bytes e decodifica só o que precisa.

Q123 Request smuggling

> *O que é request smuggling, e por que o Farol recusa com 400 uma requisição que traz **ao mesmo tempo** `Content-Length` e `Transfer-Encoding`?*

Quando há dois servidores no caminho (um *proxy* como o nginx e o servidor da aplicação), eles precisam concordar sobre **onde termina cada requisição**. Se um usa o `Content-Length` e o outro usa o `chunked`, um atacante pode montar bytes que, para o *proxy*, são uma requisição só e, para o servidor, são duas. A segunda, "contrabandeada", passa por fora dos controles do *proxy* e pode até ser atribuída à conexão de outro usuário.

A RFC 9112 diz que, com os dois cabeçalhos presentes, a mensagem é ambígua. O jeito mais seguro, que é o do Farol, é **recusá-la** e fechar a conexão, em vez de adivinhar.

Q124 Response splitting

O que é response splitting? Que validação o `EscritorDeResposta` precisa fazer nos valores de cabeçalho, e por quê?

Se um valor de cabeçalho vindo do usuário (um nome de arquivo, um parâmetro usado num `Location` de redirecionamento) contém `\r\n`, o atacante consegue **encerrar o cabeçalho e escrever os seus próprios**, ou até uma resposta inteira nova: `Location: /x\r\nSet-Cookie: sessao=do-atacante`. O servidor precisa **recusar** (ou nunca aceitar) CR, LF e outros caracteres de controle em nomes e valores de cabeçalho antes de escrevê-los. É o mesmo princípio da injeção de SQL: dado nunca pode virar estrutura.

Q125 Limites contra abuso

Como o Farol se protege de clientes lentos ou maliciosos? Cite os limites do record `Limites` e o código de status de cada um.

- Linha de pedido maior que o permitido (8 KB): **414 URI Too Long**.
- Cabeçalhos grandes demais no total (16 KB) ou em número excessivo (100): **431 Request Header Fields Too Large**.
- Corpo maior que o permitido (1 MB): **413 Content Too Large**.
- Cliente que fica calado no meio da requisição além do **prazo de leitura** (10 s): **408 Request Timeout**. Esta é a defesa contra o ataque *Slowloris*, que abre conexões e manda um byte de vez em quando.

Além disso, `Configuracao` limita as **conexões simultâneas** com um `Semaphore` (acima disso responde **503**) e define o **prazo ocioso** do *keep-alive*. Sem limites, um único cliente consegue esgotar a memória ou as conexões do servidor.

Q126 Dois prazos

*Por que o Farol tem **dois** prazos diferentes, o prazo de leitura e o prazo ocioso? O que aconteceria se houvesse um só?*

- O **prazo ocioso** vale **entre** requisições numa conexão *keep-alive*: o cliente pode ficar um tempo sem pedir nada (um navegador faz isso), e a conexão é fechada em silêncio quando o prazo acaba.
- O **prazo de leitura** vale **durante** uma requisição, depois que o primeiro byte chegou. Um cliente que começou a falar e parou está travando recursos, e recebe 408.

Com um prazo único e longo, um atacante *Slowloris* ganharia muito tempo para cada byte. Com um prazo único e curto, conexões legítimas em *keep-alive* seriam derrubadas a toda hora, e cada requisição pagaria de novo o custo de abrir a conexão (e o do TLS, num servidor real).

Q127 Especificidade e 405

*No roteador do Farol, as rotas `/tarefas/{id}` e `/tarefas/novas` podem casar com o mesmo caminho. Como o `Roteador` escolhe? E quando ele devolve **405** em vez de **404**?*

Cada segmento do padrão tem um peso: **literal** vale 2, **parâmetro** (`{id}`) vale 1 e **resto** (`{*caminho}`) vale 0. Entre as rotas que casam com o caminho, vence a de maior especificidade, comparando segmento a

segmento. Para `/tarefas/novas`, a rota literal `/tarefas/novas` (2, 2) vence `/tarefas/{id}` (2, 1), qualquer que seja a ordem de registro.

A `Busca` é uma interface selada com três resultados:

- `Encontrada`: rota e parâmetros;
- `MetodoNaoPermitido`: **algum** padrão casa com o caminho, mas nenhum com aquele método. Vira **405**, com o cabeçalho `Allow` listando os métodos permitidos;
- `NaoEncontrada`: nenhum padrão casa. Vira **404**.

Q128 Path traversal

> *Explique as **duas** defesas do Farol contra path traversal em `ArquivosEstaticos`. Por que só verificar se o caminho contém ".." não basta?*

1. Filtro de nomes: cada segmento do caminho pedido é recusado se for `..`, `.`, vazio, ou contiver barras, `\` ou caracteres de controle. Isso é feito **depois** de decodificar o `%2e%2e`, que é ".." codificado.

2. Confinamento pela realidade: o arquivo candidato é resolvido com `toRealPath()` (absoluto, normalizado e **seguindo ligações simbólicas**), e o resultado precisa começar com a raiz também resolvida: `real.startsWith(raiz)`.

Só procurar ".." no texto não basta porque há codificações (`%2e%2e`, `..%2f`, barras invertidas no Windows) e, principalmente, **ligações simbólicas**: um *link* `fotos -> /etc` dentro da pasta pública passa em qualquer filtro de texto. A segunda defesa pega esse caso.

Q129 Cache condicional

> *Como funcionam `ETag` e `If-None-Match`? O que o servidor ganha ao responder `304 Not Modified`?*

O servidor envia com o arquivo um `ETag`, uma "impressão digital" da versão (no Farol, derivada do tamanho e da data de modificação), e um `Last-Modified`. Na próxima vez, o navegador pergunta com `If-None-Match: "<etag>"` (ou `If-Modified-Since`). Se o arquivo não mudou, o servidor responde **304 Not Modified sem corpo**, e o navegador usa a cópia que já tem. Economiza banda e tempo: para uma imagem de 2 MB, a resposta cai para algumas dezenas de bytes de cabeçalho.

Q130 Desligamento gracioso

> *Como o Farol faz o ***desligamento gracioso***? Por que simplesmente encerrar o processo é ruim para os clientes?*

Um *shutdown hook* (`Runtime.getRuntime().addShutdownHook`), acionado por Ctrl+C ou `SIGTERM` (o sinal que Docker e systemd enviam), para de aceitar conexões fechando o `ServerSocket`, deixa as requisições **em andamento** terminarem dentro de um prazo de desligamento, e só depois fecha o que sobrou. Encerrar na hora cortaria respostas pela metade: o cliente receberia um arquivo truncado ou um erro de conexão numa operação que talvez já tivesse sido gravada, sem saber se ela deu certo.

Q131 Cada pedra, uma fase

> *Na retrospectiva, o Farol usou praticamente todas as fases do curso. Associe cada item a uma fase: `record`s e `sealed` para requisição e resposta; `Semaphore` e virtual threads; `toRealPath` e `Files`; testes com `HttpClient`; `jlink` com CDS.*

- `record`s, `sealed` e *pattern matching* para `Requisicao`, `Conteudo` e `Busca`: **Fase 2** (Objetos).
- `Semaphore`, *virtual threads* e *keep-alive* concorrente: **Fase 7** (Concorrência).
- `Files`, `Path.toRealPath`, `try` com recursos e *streams* de bytes: **Fase 3** (Biblioteca padrão).
- Testes unitários do leitor e do roteador, e de integração com `java.net.http.HttpClient`: **Fase 5** (Projetos e ferramentas).
- `jlink` com `--generate-cds-archive` e o `module-info` do servidor: **Fases 5 e 8** (módulos e distribuição).

Exercícios de código — soluções

Q132 Parser da linha de pedido

> *Escreva um método que interprete a **linha de pedido** HTTP (`"GET /tarefas?pagina=2 HTTP/1.1"`) num record `LinhaDePedido(String metodo, String caminho, String consulta, String versao)`, recusando linhas malformadas com `IllegalArgumentException`.*

```
record LinhaDePedido(String metodo, String caminho, String consulta, String versao) {
```

```
    static LinhaDePedido ler(String linha) {
        String[] partes = linha.split(" ", -1);                    // -1: não descarta vazios no
fim
        if (partes.length != 3) throw new IllegalArgumentException("esperado 3 partes separadas
por UM espaço");
        String metodo = partes[0], alvo = partes[1], versao = partes[2];
        if (!metodo.matches("[A-Z]{1,16}")) throw new IllegalArgumentException("método inválido:
" + metodo);
        if (!alvo.startsWith("/")) throw new IllegalArgumentException("alvo deve começar com /");
        if (!versao.equals("HTTP/1.1") && !versao.equals("HTTP/1.0"))
            throw new IllegalArgumentException("versão não suportada: " + versao);
        int interrogacao = alvo.indexOf('?');
        String caminho = interrogacao < 0 ? alvo : alvo.substring(0, interrogacao);
        String consulta = interrogacao < 0 ? "" : alvo.substring(interrogacao + 1);
        return new LinhaDePedido(metodo, caminho, consulta, versao);
    }
}
```

```
void main() {
    IO.println(LinhaDePedido.ler("GET /tarefas?pagina=2 HTTP/1.1"));
    for (String ruim : List.of("GET  /duplo-espaco HTTP/1.1", "get / HTTP/1.1", "GET tarefas
HTTP/1.1", "GET / HTTP/2")) {
        try {
            LinhaDePedido.ler(ruim);
        } catch (IllegalArgumentException e) {
            IO.println("400: " + e.getMessage());
        }
    }
}
```

Saída

```
LinhaDePedido[metodo=GET, caminho=/tarefas, consulta=pagina=2, versao=HTTP/1.1]
400: esperado 3 partes separadas por UM espaço
400: método inválido: get
400: alvo deve começar com /
400: versão não suportada: HTTP/2
```

A RFC exige **um** espaço entre as partes, e servidores tolerantes demais são justamente os que abrem brechas de *smuggling*. No Farol, esses erros viram `ErroDeProtocolo`, que o servidor transforma em resposta 400. O limite de tamanho da linha (414) é aplicado **antes**, durante a leitura dos bytes.

Q133 Leitor de chunked

> *Implemente a decodificação de um corpo `Transfer-Encoding: chunked` a partir de uma `String`, respeitando um tamanho máximo de corpo. Cuidado com tamanhos em hexadecimal como `ffffffff`.*

```
String decodificar(String bruto, long maximo) {
    var corpo = new StringBuilder();
    int pos = 0;
    while (true) {
        int fimDaLinha = bruto.indexOf("\r\n", pos);
        if (fimDaLinha < 0) throw new IllegalArgumentException("linha de tamanho sem CRLF");
        String linha = bruto.substring(pos, fimDaLinha);
        int ext = linha.indexOf(';');                              // ignora extensões:
"1a;nome=valor"
        String hexa = (ext < 0 ? linha : linha.substring(0, ext)).strip();
        if (hexa.isEmpty() || hexa.length() > 16 || !hexa.matches("[0-9a-fA-F]+"))
            throw new IllegalArgumentException("tamanho inválido: " + linha);
        long tamanho = Long.parseLong(hexa, 16);                   // long: "ffffffff" estoura
um int
        pos = fimDaLinha + 2;
        if (tamanho == 0) return corpo.toString();                 // último pedaço (trailers
ignorados aqui)
        if (corpo.length() + tamanho > maximo) throw new IllegalArgumentException("413: corpo
grande demais");
        corpo.append(bruto, pos, pos + (int) tamanho);
        pos += (int) tamanho;
        if (!bruto.startsWith("\r\n", pos)) throw new IllegalArgumentException("pedaço sem CRLF
no fim");
        pos += 2;
    }
}
```

```
void main() {
    IO.println(decodificar("4\r\nWiki\r\n6\r\npedia \r\n3\r\nfaz\r\n0\r\n\r\n", 1024));
    try {
        decodificar("ffffffff\r\n", 1024);
    } catch (IllegalArgumentException e) {
        IO.println(e.getMessage());
    }
}
```

Saída

```
Wikipedia faz
413: corpo grande demais
```

Com `Integer.parseInt(hexa, 16)`, o `ffffffff` (4.294.967.295) lançaria `NumberFormatException`. Pior: sem o limite, um valor enorme faria o servidor tentar reservar gigabytes. A regra de ouro é conferir o limite **antes** de ler ou alocar, porque o tamanho vem de quem pode estar mentindo. O Farol faz o mesmo lendo bytes do *socket*, e não de uma `String`.

Q134 Confinamento com toRealPath

> *Escreva a verificação de path traversal com `toRealPath`: um método `Optional<Path> resolver(Path raiz, String pedido)` que devolve o arquivo só se ele existir e estiver **dentro** da raiz. Teste com `index.html`, `../segredo.txt` e um link simbólico que aponta para fora.*

```
import java.nio.file.*;
```

```
Optional<Path> resolver(Path raiz, String pedido) {
    try {
        Path raizReal = raiz.toRealPath();
        Path real = raizReal.resolve(pedido).toRealPath();           // normaliza e segue links
simbólicos
        if (!real.startsWith(raizReal) || !Files.isRegularFile(real)) return Optional.empty();
        return Optional.of(real);
    } catch (IOException | InvalidPathException e) {
        return Optional.empty();                                     // não existe ou nome
inválido: 404
    }
}
```

```
void main() throws IOException {
    Path base = Files.createTempDirectory("farol");
    Path publica = Files.createDirectories(base.resolve("publica"));
    Files.writeString(publica.resolve("index.html"), "<h1>oi</h1>");
    Files.writeString(base.resolve("segredo.txt"), "senha");
    Files.createSymbolicLink(publica.resolve("atalho"), base.resolve("segredo.txt"));
```

```
    for (String pedido : List.of("index.html", "../segredo.txt", "atalho", "nao-existe.html"))
        IO.println(pedido + " -> " + (resolver(publica, pedido).isPresent() ? "200" : "404"));
}
```

Saída

```
index.html -> 200
../segredo.txt -> 404
atalho -> 404
nao-existe.html -> 404
```

O "atalho" não tem nenhum ".." no nome e passaria por qualquer filtro de texto, mas o `toRealPath` revela que ele leva a `segredo.txt`, fora da raiz. Responder 404, e não 403, a quem tenta sair da pasta não confirma a existência do arquivo. (A criação de *links* simbólicos pode exigir permissão especial no Windows.)

TRILHA A

Web, API e bancos de dados: as Trilhas da Ilha

Artigos 53 a 66

#53 Extensão — Java no Servidor: Servlets, Jakarta EE, Spring e Quarkus
#54 Extensão — Dados que Sobrevivem: SQL Essencial e JDBC com SQLite
#55 Extensão — Spring Boot pela Primeira Vez: Injeção de Dependência
#56 Extensão — Objetos no Banco: JPA e Hibernate
#57 Extensão — O Banco que Evolui: Spring Data e Migrations com Flyway
#58 Extensão — PostgreSQL: o Banco Servidor, JSONB e Arrays
#59 Extensão — MySQL: Driver, Dialeto e as Diferenças que Importam
#60 Extensão — MongoDB: Documentos, Agregações e Quando Escolher Cada Modelo
#61 Extensão — A API das Trilhas: REST, Validação, Erros e OpenAPI
#62 Extensão — Quem É Você? Spring Security e JWT
#63 Extensão — Bancos de Verdade nos Testes: Testcontainers
#64 Extensão — Páginas Dinâmicas: Thymeleaf e Formulários
#65 Extensão — O Site Completo: htmx, Login, Área Administrativa e Upload
#66 Extensão — Do Laptop ao Servidor: Docker, nginx e Deploy

Perguntas conceituais

Q135 O mapa do Java no servidor

> *Qual é a relação entre a API de **Servlets**, o **Jakarta EE**, o **Spring** e o **Quarkus**? Por que o pacote mudou de `javax.*` para `jakarta.*`?*

- A **API de Servlets** é a base: um contrato em que um contêiner (Tomcat, Jetty) recebe a requisição HTTP e chama o seu código. Ela faz, pronta, a parte que escrevemos à mão no Farol.
- **Jakarta EE** é o conjunto de especificações padronizadas por cima disso: Servlets, JPA, Bean Validation, CDI, JAX-RS...
- **Spring** (e o Spring Boot) é um ecossistema independente que **usa** várias dessas especificações (Servlets, JPA, Validation), com o seu próprio contêiner de injeção de dependência e muita configuração automática.
- **Quarkus** implementa especificações do Jakarta EE/MicroProfile com foco em inicialização rápida e compilação nativa com GraalVM.

A troca do pacote aconteceu porque a Oracle doou o Java EE à Eclipse Foundation, mas não o direito de usar a marca "java" nos pacotes. A partir do Jakarta EE 9 tudo virou `jakarta.*`, e o Spring Boot 3 fez a migração.

Q136 INNER e LEFT JOIN

> *Explique a diferença entre `INNER JOIN` e `LEFT JOIN` com as tabelas `trilha` e `ponto_de_interesse`. Qual delas lista também as trilhas sem nenhum ponto?*

O `INNER JOIN` devolve só as combinações que **casam** dos dois lados: trilhas sem nenhum ponto de interesse não aparecem. O `LEFT JOIN` devolve **todas** as linhas da tabela da esquerda (as trilhas) e, quando não há correspondente à direita, preenche as colunas do ponto com `NULL`.

```
SELECT t.nome, COUNT(p.id) AS pontos
FROM trilha t
LEFT JOIN ponto_de_interesse p ON p.trilha_id = t.id
GROUP BY t.id, t.nome
ORDER BY pontos DESC, t.nome;
```

Note o `COUNT(p.id)`, e não `COUNT(*)`: `COUNT(coluna)` ignora os `NULL`, então as trilhas sem pontos aparecem com 0, e não com 1.

Q137 SQL injection

> *Por que concatenar texto do usuário numa consulta SQL é perigoso? Mostre o ataque com a entrada `' OR '1'='1` e explique por que o `PreparedStatement` o neutraliza.*

Com `"SELECT * FROM usuario WHERE email = '" + email + "'"` e a entrada `' OR '1'='1`, a consulta vira `... WHERE email = '' OR '1'='1'`, que é verdadeira para todas as linhas: o atacante vê (ou entra como) qualquer usuário. Com `'; DROP TABLE usuario; --` e um *driver* que aceite vários comandos, pior ainda.

O `PreparedStatement` envia a **estrutura** da consulta (`WHERE email = ?`) separada dos **valores**. O banco compila a consulta antes de ver os dados, e o valor é tratado só como um literal de texto, por mais aspas que contenha. Dado nunca vira código. Isso vale também para `JdbcClient`, JPA com parâmetros (`:email`) e Spring Data. Nomes de coluna em `ORDER BY` não podem ser parâmetros: esses precisam vir de uma **lista fixa** de valores permitidos.

Q138 ACID

> *O que é uma **transação**? Explique o que significa cada letra de **ACID** com o exemplo de uma inscrição numa trilha que decrementa as vagas e grava a inscrição.*

Uma transação é um conjunto de operações que o banco trata como **uma unidade**.

- **Atomicidade**: ou as duas coisas acontecem (vaga decrementada **e** inscrição gravada), ou nenhuma. Uma falha no meio desfaz tudo (*rollback*).
- **Consistência**: as regras do banco (chaves estrangeiras, `CHECK vagas >= 0`, `UNIQUE`) valem antes e depois.
- **Isolamento**: duas inscrições simultâneas não enxergam os passos intermediários uma da outra, no grau definido pelo nível de isolamento.
- **Durabilidade**: depois do `COMMIT`, os dados sobrevivem a uma queda de energia (graças ao log de escrita antecipada).

No JDBC: `conexao.setAutoCommit(false)`, as duas operações, `commit()`, e `rollback()` no `catch`. No Spring: `@Transactional`.

Q139 Injeção de dependência

> *O que é **injeção de dependência**? Por que o Spring prefere a injeção pelo **construtor** à injeção em campos com `@Autowired`?*

Em vez de a classe criar as próprias dependências (`new RepositorioJdbc(...)` dentro do serviço), ela as **recebe** de fora, e um contêiner monta o grafo de objetos. As classes ficam desacopladas das implementações (o serviço depende da interface `RepositorioDeTrilhas`) e testáveis (no teste, passa-se um *fake*). Foi o que a miniatura de contêiner com reflexão mostrou.

A injeção pelo construtor:

- deixa as dependências **explícitas** e obrigatórias: o objeto nunca existe pela metade;
- permite campos `final`, e portanto imutáveis;
- permite testar sem Spring, com um simples `new Servico(fake)`;
- revela dependências demais: um construtor com 9 parâmetros é um sinal de que a classe faz coisas demais.

Com um único construtor, o `@Autowired` nem é necessário.

Q140 A armadilha do proxy

> *Por que um método anotado com `@Transactional` que é chamado **por outro método da mesma classe** não abre transação?*

O Spring implementa `@Transactional` com um **proxy**: quem recebe o *bean* injetado recebe, na verdade, um objeto intermediário que abre a transação, chama o método real e faz *commit* ou *rollback*. Uma chamada `this.outroMetodo()` de dentro da própria classe **não passa pelo proxy**: vai direto ao objeto real, e a anotação é ignorada em silêncio. O mesmo vale para `@Async` e `@Cacheable`, e também para métodos `private`. As soluções são mover o método para outro *bean* ou anotar o método público que é chamado de fora.

Q141 Ciclo de vida da entidade

> *Quais são os estados de uma entidade JPA (**transient**, **managed**, **detached**, **removed**)? O que é o dirty checking?*

- **Transient**: criada com `new`, o JPA não sabe que ela existe.
- **Managed** (gerenciada): associada ao **contexto de persistência** aberto, depois de um `persist`, de um `find` ou do resultado de uma consulta.
- **Detached** (desanexada): o contexto foi fechado (a transação terminou) e o objeto continua existindo, mas as mudanças nele não vão mais para o banco sozinhas.
- **Removed**: marcada para exclusão no próximo *flush*.

Dirty checking: o contexto guarda uma **cópia** (*snapshot*) dos valores de cada entidade gerenciada. No *flush* (antes do *commit*), compara cada entidade com a cópia e gera os `UPDATE`s das que mudaram, sem nenhuma chamada a `save`. Foi o que a miniatura de contexto de persistência mostrou.

Q142 N+1

> *O que é o **problema N+1**? Mostre como ele aparece ao listar trilhas com os seus pontos de interesse, e duas formas de resolver.*

Com o relacionamento `@OneToMany(fetch = LAZY)`, a consulta `SELECT t FROM Trilha t` traz as N trilhas numa consulta, e cada acesso a `trilha.getPontos()` dispara **mais uma** consulta. Com 500 trilhas, são 501 idas ao banco, e a página que era instantânea na máquina do desenvolvedor, com 3 trilhas, fica lenta em produção.

Soluções:

- `JOIN FETCH` na consulta: `SELECT DISTINCT t FROM Trilha t LEFT JOIN FETCH t.pontos`;
- `@EntityGraph(attributePaths = "pontos")` no repositório do Spring Data;
- `@BatchSize` ou `hibernate.default_batch_fetch_size`, que carrega os pontos em lotes com `IN (...)`;
- para listagens, uma **projeção** (um DTO ou *record*) com só as colunas necessárias.

Para detectar, ligue o log de SQL nos testes, ou conte as consultas.

Q143 Migrações imutáveis

> *Por que **nunca** se deve editar um script de migração do Flyway que já foi aplicado? Como se faz, então, para mudar uma coluna?*

O Flyway grava, na tabela `flyway_schema_history`, a versão e um **checksum** de cada *script* aplicado. Se o arquivo `V3__...sql` for editado, o *checksum* não bate, e a aplicação **recusa-se a iniciar** (`Validate failed`). E, mesmo que não recusasse, os bancos que já rodaram a versão antiga e os bancos novos teriam esquemas diferentes.

Toda mudança vira uma **nova** migração (`V5__renomeia_coluna.sql`). Para mudanças sem parar o sistema, o padrão é **expandir e contrair**: primeiro acrescentar a coluna nova e copiar os dados (a aplicação velha e a nova convivem), e só numa migração posterior remover a coluna antiga.

Q144 O que o PostgreSQL trouxe

> *Cite três recursos do **PostgreSQL** usados nas Trilhas da Ilha que o SQLite não tinha (ou tinha de forma limitada), e para que cada um serviu.*

- `TEXT[]` (arrays) com índice **GIN**: as etiquetas de cada trilha (`'{cachoeira,familia}'`) com busca rápida por `etiquetas @> '{cachoeira}'`.
- `JSONB` com índice GIN (`jsonb_path_ops`): os detalhes variáveis de cada trilha, com consultas por conteúdo (`detalhes @> '{"pet": true}'`).
- **Busca textual em português**: uma coluna `tsvector` gerada a partir do nome e da descrição, com `to_tsvector('portuguese', ...)`, que entende radicais ("cachoeiras" encontra "cachoeira"), e índice GIN.
- Também: `IDENTITY` para as chaves, `RETURNING`, o código SQLState `23505` para violação de `UNIQUE`, e `SELECT ... FOR UPDATE` para travar linhas.

Q145 O que muda no MySQL

> *Cite três diferenças práticas entre **MySQL** e **PostgreSQL** que obrigaram a ajustar o esquema ou o código das Trilhas.*

- **Sem** `RETURNING`: o id gerado vem por `getGeneratedKeys()` (ou `LAST_INSERT_ID()`).
- **Codificação**: `utf8` no MySQL é o `utf8mb3`, que não guarda *emojis*. É preciso `utf8mb4` em tudo, inclusive na conexão do cliente. A *collation* `utf8mb4_0900_ai_ci` compara sem distinguir acentos e maiúsculas, e por isso "São Paulo" e "sao paulo" colidem num `UNIQUE`.
- **Sem arrays**: as etiquetas viram uma coluna `JSON` com índice *multi-valued* e consulta com `MEMBER OF`.
- **DDL não é transacional**: um `CREATE TABLE` faz um *commit* implícito, e uma migração que falha no meio não é desfeita.
- *Upsert* com `ON DUPLICATE KEY UPDATE`, no lugar de `ON CONFLICT`.

Q146 Documentos ou tabelas

> *Por que os **relatos** dos caminhantes foram para o **MongoDB**, enquanto o cadastro das trilhas ficou no PostgreSQL? Quando o modelo de documentos é a escolha certa?*

Os relatos são **documentos autocontidos**: texto, fotos, condições do dia, avaliações. Cada relato tem uma estrutura um pouco diferente, é quase sempre lido e escrito **inteiro**, e raramente precisa de *joins* ou de transações envolvendo outros dados. É o caso ideal do modelo de documentos: o esquema flexível e a leitura de uma entidade inteira num único acesso.

O cadastro das trilhas tem **relações** (trilha, pontos de interesse, inscrições, usuários), **integridade** (chaves estrangeiras, vagas nunca negativas) e **transações** que atravessam tabelas. Isso é o forte do modelo relacional.

Regra: documentos quando o dado é naturalmente um agregado lido junto. Relacional quando as relações e a consistência entre entidades importam. E o PostgreSQL com `JSONB` cobre muitos casos "no meio" sem um segundo banco.

Q147 Status HTTP da API

> *Quais códigos de status HTTP a API das Trilhas devolve para: criar uma trilha com sucesso; buscar uma trilha que não existe; enviar JSON com um campo obrigatório vazio; fazer login com a senha errada; um usuário comum tentar apagar uma trilha; tentar criar uma trilha com um nome que já existe?*

- Criada com sucesso: **201 Created**, com o cabeçalho `Location: /api/trilhas/{id}`.
- Não existe: **404 Not Found**.
- Dado inválido (Bean Validation): **400 Bad Request**, com a lista de campos no corpo (*Problem Details*, RFC 9457).
- Senha errada ou *token* ausente ou inválido: **401 Unauthorized** (o nome engana: significa "não autenticado").
- Autenticado, mas sem permissão: **403 Forbidden**.
- Nome duplicado (violação de `UNIQUE`): **409 Conflict**.

As respostas de erro seguem o formato `application/problem+json`, com `type`, `title`, `status`, `detail` e, no caso de validação, a lista de erros.

Q148 Anatomia do JWT

> *Descreva as três partes de um **JWT**. O conteúdo de um JWT é secreto? O que impede um usuário de trocar `"papel": "USUARIO"` por `"ADMIN"` no próprio token?*

Um JWT é `cabeçalho.carga.assinatura`, as três partes em Base64URL:

- **cabeçalho**: o algoritmo (`{"alg":"HS256","typ":"JWT"}`);
- **carga** (*claims*): `sub` (o usuário), `exp` (a expiração), `iat` (a emissão) e dados como os papéis;
- **assinatura**: HMAC-SHA256 do cabeçalho e da carga com a chave secreta do servidor (ou uma assinatura RSA/EC).

O conteúdo **não é secreto**: é só Base64, qualquer um lê. Nunca coloque senhas ou dados sensíveis na carga. O que impede a adulteração é a **assinatura**: ao trocar o papel, a assinatura deixa de conferir, e só quem tem a chave consegue gerar uma válida. O servidor também precisa **fixar** o algoritmo esperado, e recusar `"alg": "none"` e troca de algoritmo, e verificar o `exp`.

Q149 Hash de senha

> *Por que senhas são guardadas com um hash **lento** e com **sal** (BCrypt, PBKDF2, Argon2), e não com SHA-256?*

Se o banco vazar, o atacante tenta adivinhar as senhas calculando *hashes* de milhões de candidatas. O SHA-256 foi feito para ser **rápido**: uma GPU calcula bilhões por segundo. Os algoritmos de senha são **lentos de propósito** e ajustáveis (o fator de custo do BCrypt, as iterações do PBKDF2, a memória do Argon2), e cada tentativa passa a custar milissegundos.

O **sal** é um valor aleatório por usuário, guardado junto com o *hash*. Ele faz duas senhas iguais terem *hashes* diferentes e inutiliza tabelas pré-calculadas (*rainbow tables*): o atacante precisa atacar cada usuário separadamente. No Spring, o `PasswordEncoder` (`DelegatingPasswordEncoder`, com BCrypt por padrão) cuida de tudo isso.

Q150 Testcontainers

> *O que o **Testcontainers** resolve que um banco em memória (H2) não resolve? Por que o `@ServiceConnection` simplifica os testes?*

O H2 "imitando" o PostgreSQL não tem `JSONB`, arrays, `tsvector`, os mesmos códigos de erro nem o mesmo comportamento de *collation* e concorrência. Um teste verde no H2 pode esconder um SQL que o PostgreSQL recusa. O Testcontainers sobe o **banco de verdade**, na versão de produção, num contêiner Docker descartável durante os testes, e as migrações do Flyway rodam nele exatamente como em produção.

O `@ServiceConnection` (Spring Boot 3.1+) liga o contêiner à aplicação automaticamente: a URL, o usuário e a senha do banco do contêiner viram a configuração do `DataSource`, sem `@DynamicPropertySource` nem propriedades escritas à mão.

Q151 Escapar por padrão

> *Por que o **Thymeleaf** escapa o HTML por padrão em `th:text`? Quando `th:utext` seria aceitável, e que cuidado exige?*

Qualquer texto que veio do usuário (o nome de uma trilha, um comentário) pode conter `<script>...</script>`. Se for inserido na página sem escape, o *script* roda no navegador de quem lê: é o **XSS** (*cross-site scripting*), com roubo de sessão e ações em nome da vítima. O `th:text` converte `<` em `<` e afins, e o texto aparece como texto.

`th:utext` insere HTML cru. Só é aceitável para conteúdo **confiável** ou **sanitizado** por uma biblioteca de lista permitida (como o OWASP Java HTML Sanitizer), por exemplo o HTML de uma descrição escrita pela equipe num editor. Nunca para texto vindo diretamente do usuário.

Q152 CSRF

> *O que é **CSRF**? Por que ele afeta o site com login por sessão (cookie) mas não a API com JWT no cabeçalho `Authorization`?*

No *cross-site request forgery*, um site malicioso faz o navegador da vítima enviar uma requisição (um formulário oculto com POST) para o **seu** site. O navegador anexa **automaticamente** o *cookie* de sessão, e a ação acontece em nome da vítima logada. A defesa é um *token* CSRF secreto em cada formulário (o Spring Security e o Thymeleaf inserem sozinhos), que o site malicioso não consegue ler, e *cookies* `SameSite`.

Na API com JWT no cabeçalho `Authorization: Bearer ...`, o navegador **não** anexa nada automaticamente: é o JavaScript da própria aplicação que coloca o cabeçalho, e o site malicioso não tem acesso ao *token*. Por isso a API desliga a proteção CSRF, e o site não.

Q153 htmx

> *O que o **htmx** muda na forma de fazer páginas interativas? Como ele se encaixa com o Thymeleaf no back-end?*

Com o htmx, **atributos HTML** (`hx-get`, `hx-post`, `hx-target`, `hx-swap`) fazem requisições e trocam **pedaços da página** pelo HTML que o servidor devolve, sem escrever JavaScript nem ter um *front-end* separado consumindo JSON. O estado e a lógica ficam no servidor.

Com o Thymeleaf, o controlador devolve um **fragmento** de *template* (`"trilhas :: lista"`) em vez da página inteira: a busca "enquanto digita" e a paginação trocam só a lista, e o botão de excluir remove só a linha. A mesma validação do servidor funciona para páginas completas e para fragmentos.

Q154 ENTRYPOINT na forma exec

> *Na imagem Docker das Trilhas, por que o `ENTRYPOINT` chama o `java` diretamente (forma exec, `["java", "-jar", "app.jar"]`) em vez de passar por um shell (`sh -c "java -jar app.jar"`)?*

O `docker stop` envia `SIGTERM` ao **processo 1** do contêiner. Na forma *exec*, o processo 1 é a própria JVM: ela recebe o sinal, roda os *shutdown hooks* e o Spring faz o desligamento gracioso (termina as requisições em

andamento e fecha o *pool* do banco). Na forma *shell*, o processo 1 é o `sh`, que **não repassa** o sinal. A JVM nunca fica sabendo, e depois de 10 s o Docker envia `SIGKILL`: as requisições são cortadas no meio. O Farol precisou do mesmo cuidado.

Q155 Proxy reverso

> *O que o **nginx** faz na frente da aplicação Spring no servidor? Cite três responsabilidades.*

- **TLS/HTTPS**: termina a criptografia com o certificado (do Let's Encrypt) e fala HTTP simples com a aplicação na rede interna.
- **Arquivos estáticos** e fotos enviadas: serve direto do disco, com cache e compressão, sem ocupar a JVM.
- **Proteções**: limite de tamanho de corpo (`client_max_body_size`), limite de taxa de requisições, *timeouts*, cabeçalhos de segurança.
- Também repassa `X-Forwarded-For` e `X-Forwarded-Proto`, para a aplicação saber o IP e o protocolo originais (com `server.forward-headers-strategy` no Spring), e permite trocar a versão da aplicação sem derrubar as conexões.

Q156 Segredos

> *Onde devem ficar os **segredos** (senha do banco, chave do JWT) de uma aplicação em produção? Cite duas práticas erradas e uma certa.*

Erradas:

- escritos no `application.properties` versionado no Git (ficam no histórico para sempre, mesmo depois de apagados);
- embutidos na imagem Docker (qualquer um com a imagem os extrai com `docker history` ou abrindo as camadas).

Certas: **variáveis de ambiente** ou **arquivos de segredo** montados no contêiner em tempo de execução (Docker/Compose *secrets*, um `.env` fora do repositório e com permissões restritas) ou um cofre de segredos (Vault, os serviços dos provedores de nuvem). O Spring lê `SPRING_DATASOURCE_PASSWORD` do ambiente sem nenhuma mudança no código. E cada ambiente (desenvolvimento, homologação, produção) tem os seus próprios segredos.

Exercícios de código — soluções

Q157 JDBC com PreparedStatement

> *Com JDBC puro e SQLite, crie a tabela `trilha (id, nome, km)`, insira três trilhas com `PreparedStatement` e mostre que a entrada maliciosa `' OR '1'='1` **não** retorna nada quando usada como parâmetro de busca por nome.*

```
import java.sql.*;
```

```
void main() throws SQLException {
    try (Connection c = DriverManager.getConnection("jdbc:sqlite::memory:")) {
        try (Statement s = c.createStatement()) {
            s.execute("CREATE TABLE trilha (id INTEGER PRIMARY KEY, nome TEXT NOT NULL UNIQUE, km
REAL NOT NULL CHECK (km > 0)) STRICT");
        }
        try (PreparedStatement ins = c.prepareStatement("INSERT INTO trilha (nome, km) VALUES (?,
?)")) {
            for (var t : List.of(Map.entry("Pedra Bonita", 1.8), Map.entry("Pico da Tijuca",
3.2), Map.entry("Trilha do Morro", 5.5))) {
                ins.setString(1, t.getKey());
                ins.setDouble(2, t.getValue());
                ins.addBatch();                                    // envia as três de uma vez
            }
            ins.executeBatch();
        }
        for (String busca : List.of("Pico da Tijuca", "' OR '1'='1")) {
            try (PreparedStatement q = c.prepareStatement("SELECT nome, km FROM trilha WHERE nome
= ?")) {
                q.setString(1, busca);                             // o valor nunca vira SQL
                try (ResultSet rs = q.executeQuery()) {
                    int achadas = 0;
                    while (rs.next()) achadas++;
                    IO.println("busca [" + busca + "]: " + achadas + " trilha(s)");
                }
            }
        }
    }
}
```

Saída

```
busca [Pico da Tijuca]: 1 trilha(s)
busca [' OR '1'='1]: 0 trilha(s)
```

Com parâmetro, o banco procura uma trilha **chamada literalmente** `' OR '1'='1`, que não existe. Todos os recursos JDBC (`Connection`, `Statement`, `ResultSet`) são fechados pelo `try` com recursos. Execute com o *driver* no classpath: `java --enable-native-access=ALL-UNNAMED -cp sqlite-jdbc.jar TrilhasJdbc.java` (sem a opção, o Java 25 imprime um aviso quando o *driver* carrega a sua biblioteca nativa).

Q158 GROUP BY e HAVING

> *Escreva a consulta SQL que devolve, para cada dificuldade, a quantidade de trilhas e a média de quilômetros, mostrando só as dificuldades com pelo menos 2 trilhas. Qual é a diferença entre `WHERE` e `HAVING` nessa consulta?*

```
SELECT dificuldade,
       COUNT(*)           AS quantidade,
       ROUND(AVG(km), 1)  AS media_km
FROM trilha
WHERE ativa = 1                -- filtra LINHAS, antes de agrupar
GROUP BY dificuldade
HAVING COUNT(*) >= 2           -- filtra GRUPOS, depois de agrupar
ORDER BY media_km DESC;
```

O `WHERE` atua em cada linha, antes do agrupamento, e não pode usar funções de agregação. O `HAVING` atua nos grupos já formados, e é onde entram condições sobre `COUNT`, `AVG` e `SUM`. Filtrar no `WHERE` o que for possível é também mais eficiente: menos linhas chegam ao agrupamento.

Q159 Controlador REST

> *Escreva um controlador REST do Spring para `GET /api/trilhas/{id}` e `POST /api/trilhas`, com um record de entrada validado (`nome` obrigatório, `km` positivo), devolvendo 201 com `Location` na criação e 404 quando a trilha não existe.*

```
@RestController
@RequestMapping("/api/trilhas")
class TrilhaController {
```

```
    record NovaTrilha(@NotBlank @Size(max = 100) String nome,
                      @Positive double km,
                      @NotNull Dificuldade dificuldade) {}
```

```
    record TrilhaResposta(long id, String nome, double km, Dificuldade dificuldade) {}
```

```
    private final ServicoDeTrilhas servico;
```

```
    TrilhaController(ServicoDeTrilhas servico) { this.servico = servico; }   // injeção pelo
construtor
```

```
    @GetMapping("/{id}")
    ResponseEntity<TrilhaResposta> buscar(@PathVariable long id) {
        return servico.buscar(id)                                  // Optional<TrilhaResposta>
                .map(ResponseEntity::ok)
                .orElse(ResponseEntity.notFound().build());           // 404
    }
```

```
    @PostMapping
    ResponseEntity<TrilhaResposta> criar(@Valid @RequestBody NovaTrilha nova,
UriComponentsBuilder uri) {
        TrilhaResposta criada = servico.criar(nova);
        URI local = uri.path("/api/trilhas/{id}").build(criada.id());
        return ResponseEntity.created(local).body(criada);          // 201 + Location
    }
}
```

O `@Valid` dispara o Bean Validation. Se falhar, o Spring lança `MethodArgumentNotValidException`, que um `@RestControllerAdvice` (ou o suporte embutido a *Problem Details*, com `spring.mvc.problemdetails.enabled=true`) transforma em 400 com a lista de campos. *Records* separados para entrada e saída evitam expor a entidade JPA e impedem que o cliente defina campos como `id`.

Q160 PostgreSQL: arrays, JSONB e busca textual

> *Escreva a consulta PostgreSQL que encontra as trilhas com a etiqueta `cachoeira` **e** que aceitam animais (`detalhes->>'pet' = 'true'` no JSONB), ordenadas pela relevância de uma busca textual por "mata atlântica". Quais índices ajudam essa consulta?*

```
SELECT id, nome,
       ts_rank(busca, consulta) AS relevancia
FROM trilha,
     websearch_to_tsquery('portuguese', 'mata atlântica') AS consulta
WHERE etiquetas @> ARRAY['cachoeira']          -- contém a etiqueta (índice GIN no array)
  AND detalhes @> '{"pet": true}'               -- contém o par no JSONB (índice GIN
jsonb_path_ops)
  AND busca @@ consulta                         -- casa com a busca textual (índice GIN no
tsvector)
ORDER BY relevancia DESC, nome;
```

Os índices:

```
CREATE INDEX ON trilha USING GIN (etiquetas);
CREATE INDEX ON trilha USING GIN (detalhes jsonb_path_ops);
```

- `CREATE INDEX ON trilha USING GIN (busca);`, em que `busca` é a coluna `tsvector` **gerada** (`GENERATED ALWAYS AS (to_tsvector('portuguese', nome || ' ' || descricao)) STORED`).

Usar `@>` (contém) em vez de `detalhes->>'pet' = 'true'` é o que permite o índice `jsonb_path_ops` ser aproveitado. Confira com `EXPLAIN ANALYZE`.

Q161 Agregação no MongoDB

> *Escreva uma agregação do MongoDB (no `mongosh`) que calcule, para cada trilha, a nota média e a quantidade de relatos, só com os relatos de 2026, ordenando pela média e mostrando as 5 melhores.*

```
db.relatos.aggregate([
  { $match: { data: { $gte: ISODate("2026-01-01"), $lt: ISODate("2027-01-01") } } },  // filtra
primeiro (usa índice)
  { $group: {
      _id: "$trilhaId",
      media: { $avg: "$nota" },
      relatos: { $sum: 1 }
  } },
  { $sort: { media: -1, relatos: -1 } },
  { $limit: 5 },
  { $project: { _id: 0, trilhaId: "$_id", media: { $round: ["$media", 1] }, relatos: 1 } }
])
```

O `$match` vem **primeiro** para reduzir os documentos que passam pelo resto do *pipeline* e aproveitar um índice em `data`. O `$group` é o `GROUP BY`, e o `$project` dá o formato final. Para mostrar o nome da trilha, que está no PostgreSQL, a aplicação junta os dois resultados, ou o nome é duplicado no relato, uma escolha típica do modelo de documentos.

Q162 Dockerfile multiestágio

> *Escreva um `Dockerfile` em **dois estágios** para a aplicação Spring Boot das Trilhas: um estágio que compila com Maven e outro, mínimo, que só roda. O contêiner não deve rodar como root.*

```
# estágio 1: compila (tem JDK e Maven; fica de fora da imagem final)
FROM eclipse-temurin:25-jdk AS construcao
WORKDIR /src
COPY .mvn .mvn
COPY mvnw pom.xml ./
RUN ./mvnw -q dependency:go-offline          # camada de dependências: cache enquanto o pom não
muda
COPY src src
RUN ./mvnw -q package -DskipTests
```

```
# estágio 2: só roda (JRE, sem compilador nem código-fonte)
FROM eclipse-temurin:25-jre
RUN useradd --system --uid 10001 trilhas
WORKDIR /app
COPY --from=construcao /src/target/trilhas-*.jar app.jar
USER trilhas
EXPOSE 8080
ENTRYPOINT ["java", "-XX:MaxRAMPercentage=75", "-jar", "app.jar"]
```

- **Dois estágios**: a imagem final não carrega JDK, Maven, código-fonte nem o repositório `.m2`, e fica menor e com menos superfície de ataque.
- **Cópia do `pom.xml` antes do `src`**: a camada de dependências só é refeita quando o `pom` muda, e o *build* fica muito mais rápido.
- **Usuário sem privilégios**: uma falha na aplicação não dá *root* no contêiner.
- `ENTRYPOINT` **na forma exec**: repassa o `SIGTERM` para a JVM.
- `MaxRAMPercentage`: dimensiona o heap pelo limite de memória do contêiner.

Os testes rodam no CI (com Testcontainers), antes do *build* da imagem, e é por isso que o `-DskipTests` aqui é aceitável.

TRILHA B

Desktop e Android

Artigos 67 a 70

#67 Extensão — Janelas em Java: AWT, Swing e JavaFX
#68 Extensão — JavaFX: Cenas, Controles e FXML
#69 Extensão — Um App Desktop que Consome a API (MVVM)
#70 Extensão — Java no Android: o que Mudou com o Kotlin e um App que Consome a API

Perguntas conceituais

Q163 A thread da interface

> *O que é a **thread de eventos** de uma interface gráfica (a EDT no Swing, a JavaFX Application Thread no JavaFX)? Quais são as duas regras de ouro para usá-la?*

Os *toolkits* gráficos não são seguros entre *threads*. Toda a interface é desenhada e todos os eventos (cliques, teclas) são tratados por **uma única thread**, a *thread* de eventos.

1. Nunca bloqueie a thread de eventos: uma chamada à API das Trilhas, uma consulta ao banco ou um `sleep` ali congela a janela inteira ("Não está respondendo"). Trabalho lento vai para outra *thread* (um `Task` do JavaFX, um `SwingWorker`, uma *virtual thread*).

2. Só mexa na interface pela thread de eventos: de outra *thread*, o resultado volta com `Platform.runLater(...)` (JavaFX) ou `SwingUtilities.invokeLater(...)` (Swing). Mexer por fora causa exceções (`IllegalStateException: Not on FX application thread`) ou corrupções aleatórias.

Q164 JavaFX fora do JDK

> *Por que o JavaFX deixou de vir no JDK a partir do Java 11, e como ele é usado hoje num projeto?*

Ele foi separado para evoluir no seu próprio ritmo, como o projeto **OpenJFX**, sem depender do ciclo do JDK, e para deixar o JDK menor. Hoje ele é uma dependência comum: artefatos `org.openjfx:javafx-controls` (e `-fxml`) no Maven ou no Gradle, com os *plugins* `javafx-maven-plugin` e `org.openjfx.javafxplugin`, ou baixando o SDK. Algumas distribuições do JDK (Liberica Full, Zulu FX) ainda o trazem embutido. Por ser modular, a aplicação declara `requires javafx.controls;` e precisa abrir (`opens`) o pacote dos controladores para o `javafx.fxml`.

Q165 Propriedades e bindings

> *O que são **propriedades** e **bindings** no JavaFX? Dê um exemplo em que um binding substitui um ouvinte (listener) escrito à mão.*

Uma propriedade (`StringProperty`, `BooleanProperty`, `ObjectProperty<T>`) é um valor **observável**: avisa quem estiver interessado quando muda. Um *binding* declara que um valor **é sempre** função de outros, e o JavaFX mantém a relação sozinho.

Exemplo: o botão "Salvar" só deve ficar habilitado quando o campo de nome não estiver vazio. Com *listener*, seria preciso ouvir o campo e atualizar o botão à mão, com o risco de esquecer o estado inicial. Com *binding*: `botaoSalvar.disableProperty().bind(campoNome.textProperty().isEmpty());`. No MVVM, a *View* faz *bind* nas propriedades do *ViewModel*.

Q166 MVVM

> *Explique o padrão **MVVM** usado no Painel das Trilhas: o que fica no Model, na View e no ViewModel? Por que o ViewModel pode ser testado sem abrir nenhuma janela?*

- **Model**: os dados e o acesso a eles, como o `ClienteHttpDaApi`, que conversa com a API das Trilhas, e os *records* `ResumoDeTrilha`, `Pagina` e `Problema`.
- **View**: o FXML e o controlador fino, que só faz *bindings* e repassa eventos, sem lógica.
- **ViewModel**: o **estado da tela** em propriedades (a lista de trilhas, "carregando", a mensagem de erro, a página atual) e as **ações** (buscar, próxima página), que chamam o *Model* e atualizam as propriedades.

O *ViewModel* não conhece nenhum controle visual (nada de `Button` ou `TableView`), só propriedades e o *Model*. Num teste, cria-se o *ViewModel* com um cliente *fake*, chama-se `buscar("cachoeira")` e verificam-se as propriedades. Nenhuma janela é aberta.

Q167 localhost no Android

> *Por que um app Android **não** pode chamar a API das Trilhas em `http://localhost:8080`, e qual endereço usar no emulador?*

No celular (ou no emulador), `localhost` é **o próprio aparelho**, e não o computador onde a API está rodando. No emulador padrão do Android, o endereço especial `10.0.2.2` aponta para o `localhost` da máquina hospedeira. Num aparelho físico, usa-se o IP do computador na rede local. Além disso, o Android bloqueia HTTP sem TLS por padrão (desde a API 28): em desenvolvimento, é preciso liberar o domínio na `network_security_config` ou usar HTTPS. Em produção, sempre HTTPS.

Q168 Java e Kotlin no Android

> *Por que, em 2026, a recomendação do Google para apps Android novos é Kotlin, e o que continua valendo do Java nesse ecossistema?*

Desde 2019 o Google adota a abordagem "Kotlin primeiro": as bibliotecas novas (Jetpack Compose, corrotinas, as extensões do Jetpack) são projetadas para Kotlin, a documentação e os exemplos vêm em Kotlin, e o Compose, a forma atual de fazer interfaces, exige Kotlin. Além disso, o Android não roda o Java 25 completo: o ART suporta um subconjunto da API, via *desugaring*.

O que continua valendo: Kotlin roda na mesma lógica de bytecode, usa **as mesmas bibliotecas Java** (OkHttp, Retrofit, Jackson) e interopera nos dois sentidos. Todo o conhecimento de orientação a objetos, coleções, concorrência e *design* transfere diretamente, e o código Java continua compilando e rodando em apps existentes.

Exercícios de código — soluções

Q169 ViewModel de busca

*Escreva um **ViewModel** de busca de trilhas, com as propriedades `termo`, `carregando` e `mensagem`, e a lista `resultados`, que chama um cliente assíncrono e atualiza tudo na thread da interface. Mostre como seria o teste dele com um cliente fake.*

```
public class BuscaViewModel {
    private final ClienteDaApi cliente;                          // interface: a real usa
HttpClient
    private final Executor naInterface;                          // em produção:
Platform::runLater
```

```
    public final StringProperty termo = new SimpleStringProperty("");
    public final BooleanProperty carregando = new SimpleBooleanProperty(false);
    public final StringProperty mensagem = new SimpleStringProperty("");
    public final ObservableList<ResumoDeTrilha> resultados = FXCollections.observableArrayList();
```

```
    public BuscaViewModel(ClienteDaApi cliente, Executor naInterface) {
        this.cliente = cliente;
        this.naInterface = naInterface;
    }
```

```
    public void buscar() {
        carregando.set(true);
        mensagem.set("");
        cliente.buscar(termo.get())                              //
CompletableFuture<List<ResumoDeTrilha>>
                .whenCompleteAsync((lista, erro) -> {
                    carregando.set(false);
                    if (erro != null) mensagem.set("Falha ao buscar: " +
erro.getCause().getMessage());
                    else if (lista.isEmpty()) mensagem.set("Nenhuma trilha encontrada.");
                    else resultados.setAll(lista);
                }, naInterface);                                 // volta para a thread da
interface
    }
}
```

```
@Test
void buscaPreencheResultados() {
    var fake = (ClienteDaApi) termo -> CompletableFuture.completedFuture(
            List.of(new ResumoDeTrilha(1, "Pedra Bonita", 1.8)));
    var vm = new BuscaViewModel(fake, Runnable::run);             // no teste: executa na mesma
thread
    vm.termo.set("pedra");
    vm.buscar();
    assertFalse(vm.carregando.get());
    assertEquals("Pedra Bonita", vm.resultados.getFirst().nome());
}
```

Injetar o `Executor` da interface é o truque que torna o *ViewModel* testável: em produção é `Platform::runLater`, e no teste é `Runnable::run`, que executa na hora, sem precisar do *toolkit* do JavaFX.

Q170 Cliente da API

> *Escreva o cliente HTTP que busca uma página de trilhas na API com o `java.net.http.HttpClient`, de forma assíncrona e com timeout, e converte o JSON em records com Jackson. O mesmo código funcionaria no Android?*

```
public final class ClienteHttpDaApi implements ClienteDaApi {
    private final HttpClient http = HttpClient.newBuilder()
            .connectTimeout(Duration.ofSeconds(5))
            .build();
    private final ObjectMapper json = new ObjectMapper();
    private final URI base;
```

```
    public ClienteHttpDaApi(URI base) { this.base = base; }
```

```
    @Override
    public CompletableFuture<List<ResumoDeTrilha>> buscar(String termo) {
        String q = URLEncoder.encode(termo, StandardCharsets.UTF_8);          // espaços e
acentos na URL
        var pedido = HttpRequest.newBuilder(base.resolve("/api/trilhas?busca=" + q))
                .timeout(Duration.ofSeconds(10))
                .header("Accept", "application/json")
                .GET()
                .build();
        return http.sendAsync(pedido, HttpResponse.BodyHandlers.ofString())
                .thenApply(resp -> {
                    if (resp.statusCode() != 200) throw new ErroDaApi(resp.statusCode(),
resp.body());
                    try {
                        return json.readValue(resp.body(), new
TypeReference<Pagina<ResumoDeTrilha>>() {}).conteudo();
                    } catch (IOException e) {
                        throw new UncheckedIOException(e);
                    }
                });
    }
}
```

No desktop, sim. No Android, o `java.net.http` **não está disponível** (não faz parte da API do Android), e por isso a versão Android da Trilha B usa OkHttp ou `HttpURLConnection`. Os *records* e o Jackson funcionam nos dois. É por isso que vale isolar o transporte atrás da interface `ClienteDaApi`.

TRILHA C

Jogos com libGDX

Artigos 71 a 77

#71 Um Novo Território — Java nos Jogos, o Game Loop e o Caso Minecraft
#72 O Primeiro Sprite — Janela, Desenho e Input com libGDX
#73 Quando as Coisas se Tocam — Colisão e o Primeiro Jogo: Asteroids
#74 Gravidade de Verdade — Física com Box2D
#75 Mundos em Blocos — Mapas de Tiles e um Jogo de Plataforma
#76 Sem Lixo no Loop — Pools, ECS e o Coletor nos Jogos
#77 O Capstone dos Jogos — o Plataforma Completo e Exportado

Perguntas conceituais

Q171 Game loop

> *O que é o **game loop**? Por que um jogo não pode simplesmente esperar por eventos, como um servidor ou uma janela de formulário?*

É o laço que roda o jogo inteiro: a cada volta ele **lê a entrada**, **atualiza o mundo** (movimento, colisões, regras) e **desenha** o resultado, dezenas de vezes por segundo (60 vezes, num monitor de 60 Hz, o que dá 16,7 ms por quadro). O jogo não espera por eventos porque o mundo **muda sozinho**: os inimigos andam, a gravidade puxa e o tempo corre mesmo quando o jogador não aperta nada. Um servidor dorme até chegar uma requisição, e um jogo precisa produzir um quadro novo a cada 16,7 ms, aconteça o que acontecer.

Q172 Passo fixo e interpolação

> *Por que a física do jogo deve avançar em **passo fixo** (1/60 s), com um acumulador, em vez de usar o tempo do quadro diretamente? Para que serve a **interpolação** na hora de desenhar?*

Com o passo variável (`posicao += velocidade * delta`), o resultado depende da taxa de quadros: num computador lento, os passos grandes fazem objetos atravessarem paredes (*tunneling*), e integradores como o de Euler dão alturas de pulo diferentes em máquinas diferentes. Com o **passo fixo**, a simulação é **determinística**: o mesmo número de passos de 1/60 s dá sempre o mesmo resultado, em qualquer máquina, e fica testável.

O acumulador soma o tempo real de cada quadro e executa quantos passos fixos couberem. A **interpolação** resolve o que sobra: o desenho acontece "entre" dois passos, e desenhar a posição `lerp(anterior, atual, sobra/passo)` evita tremidas quando a taxa do monitor (144 Hz, por exemplo) não bate com a da física.

Q173 dispose

> *Por que o libGDX exige que `Texture`, `SpriteBatch`, `BitmapFont` e o `World` do Box2D recebam `dispose()`, se o Java tem coletor de lixo?*

Esses objetos Java são "alças" para recursos que **não estão no heap**: texturas na **memória da GPU** (via OpenGL) e o mundo do Box2D na **memória nativa** (C++). O coletor só enxerga o pequeno objeto Java. Ele

não sabe que por trás há megabytes de memória de vídeo, e não tem pressa em coletar. Sem `dispose()`, cada troca de tela vaza texturas até a GPU esgotar. É a mesma regra do `close()`: recurso externo tem dono, e o dono libera. E o contrário também é perigoso: usar um recurso já liberado, como a junta de um corpo destruído no Box2D, derruba o processo com falha de segmentação, e não com uma exceção.

Q174 Unidades do mundo e viewport

> *Por que se trabalha em **metros** (unidades do mundo) e não em pixels, e o que o `FitViewport` faz com uma janela de proporção diferente?*

Em pixels, a velocidade, o tamanho dos objetos e a física mudam com a resolução da tela, e o Box2D é ajustado para objetos entre 0,1 e 10 **metros**: com números de pixel (100, 500), a simulação fica lenta e esquisita. Em metros, a lógica do jogo é a mesma em qualquer tela, e só a câmera converte para pixels.

O `FitViewport(20, 11.25)` garante que **sempre** aparece a mesma área do mundo, na mesma proporção (16:9). Numa janela mais larga ou mais alta, ele acrescenta faixas pretas (*letterbox*) em vez de esticar a imagem ou mostrar mais mundo a quem tem monitor maior.

Q175 Colisão por varredura

> *Por que o teste de colisão de um tiro rápido com um asteroide pequeno usou **segmento contra círculo** (o caminho do passo inteiro), e não só o ponto final do tiro?*

Um tiro rápido anda uma distância grande em cada passo. Se o asteroide é menor que esse deslocamento, o tiro pode estar **antes** dele num passo e **depois** no seguinte, sem nunca ter um ponto "dentro" do círculo: ele atravessa sem colidir (*tunneling*). Testar o **segmento** entre a posição anterior e a atual contra o círculo (`Intersector.intersectSegmentCircle`) pega qualquer cruzamento durante o passo. É a mesma ideia da colisão contínua do Box2D (`bullet = true`).

Q176 Tipos de corpo

> *Qual é a diferença entre corpos **estáticos**, **cinemáticos** e **dinâmicos** no Box2D? Dê um exemplo de cada no jogo do cais.*

- **Estático**: não se move e tem massa infinita, como o chão, as paredes e o cais. É o mais barato: não é simulado.
- **Cinemático**: move-se com a velocidade que **você** define, sem sofrer forças nem colisões, mas empurra os dinâmicos. Exemplo: uma plataforma que vai e volta, ou um guindaste com trajetória fixa.
- **Dinâmico**: sofre gravidade, forças, impulsos e colisões, como as caixas de carga, o projétil e as tábuas da ponte.

Q177 Coyote time e jump buffer

> *Explique o **coyote time** e o **jump buffer** do jogo de plataforma. Por que eles são regras de game design, e não de física?*

- **Coyote time** (0,10 s): o jogador ainda pode pular por alguns instantes **depois** de sair da borda da plataforma.

- **Jump buffer** (0,12 s): um pulo apertado um pouco **antes** de tocar o chão fica guardado e acontece no instante do pouso.

Os dois compensam a percepção humana: o jogador sente que apertou "na hora", mas o reflexo e a taxa de quadros o fazem apertar alguns milissegundos fora da janela exata. Fisicamente, pular do ar não faz sentido nenhum, mas o jogo fica **justo**, e o controle, "gostoso". Por serem regras, e não física, ganharam testes próprios (4 passos no ar ainda pula, 9 não).

Q178 Altura discreta do pulo

> *Por que a altura do pulo no jogo foi 2,3333 m e não os 2,45 m da fórmula $v_0^2/2g$? Que consequência isso tem para o level design?*

O jogo integra com **Euler semi-implícito** em passo fixo: a cada passo, primeiro a gravidade reduz a velocidade e depois a posição anda com a velocidade já reduzida. A soma exata desses passos dá $v_0^2/2g - v_0 \cdot dt/2 = 2{,}45 - 14 \times (1/60) / 2 = 2{,}3333$ m. A fórmula contínua supõe passos infinitamente pequenos. Consequência: as plataformas precisam ser projetadas com a **altura discreta**. Uma plataforma a 2,4 m parece alcançável no papel e é impossível no jogo. Por isso o teste confere a altura discreta, e o piloto automático confirma que cada fase pode ser vencida.

Q179 type e class no Tiled

> *Por que o carregador TMX do libGDX não encontrava objetos marcados no campo **"Class"** do Tiled 1.9? Como o jogo resolveu?*

O Tiled grava o tipo do objeto no atributo XML `type`, menos na versão 1.9, que passou a gravar `class`; a 1.10 voltou ao `type`, para ser compatível com as anteriores. O carregador do libGDX só lê `type`: num mapa salvo pelo Tiled 1.9, os objetos marcados pelo campo "Class" chegam sem tipo, e sem erro nenhum. O jogo identifica os objetos pelo **nome** (`inicio`, `moeda`, `bandeira`), que funciona em qualquer versão. A alternativa é uma propriedade personalizada (`tipo = moeda`), lida com `obj.getProperties().get("tipo", String.class)`. Atualizar o Tiled para a 1.10 ou mais nova também resolve, mas as duas primeiras saídas funcionam com qualquer versão.

Q180 O pior quadro

> *No teste com 30 mil partículas, a versão com objetos novos teve média de 0,37 ms por quadro, mas um pior quadro de quase 6 ms, caído numa pausa do coletor. Por que o **pior quadro** importa mais que a média num jogo, e o que as versões com `Pool` e com arrays mudaram?*

O jogador não percebe a média: percebe o **engasgo**. Um quadro de 6 ms gasta mais de um terço do orçamento de 16,7 ms de uma vez. Somado ao desenho e a uma cena mais pesada, estoura o prazo e o jogo "pula" um quadro. Esse pior quadro caiu dentro de uma pausa do G1 nas três execuções medidas: o coletor entrava para limpar os 113.739 bytes alocados por quadro.

- **Pool**: reaproveita os objetos. Zero bytes por quadro e nenhuma coleta. A média quase não muda (0,35 ms), mas o pior quadro cai para 1,5 ms: o que o Pool elimina é o engasgo.
- **Arrays por atributo** (o estilo ECS): zero alocação e os dados contíguos na memória, aproveitando o cache do processador. Média de 0,14 ms e pior quadro de 0,23 ms.

Os tempos variam com a máquina; os bytes e as coletas se repetem iguais. Trocar para o **ZGC** derrubou as pausas para 0,02 a 0,03 ms, mas o pior quadro da versão com objetos ficou na mesma faixa (3,9 a 6,5 ms): os engasgos que sobram ali não vêm do coletor.

Q181 Armadilhas do Pool

> *Cite as três armadilhas de um `Pool` que o curso demonstrou e a regra única que evita todas.*

1. Liberar duas vezes: o `Pool` do libGDX não confere, e o mesmo objeto é entregue a dois donos (dois inimigos dividindo a mesma bala).

2. Guardar referência depois do `free`: o objeto renasce com outro dono, e quem guardou a referência passa a mexer no objeto errado.

3. Esquecer o `reset`: o objeto "novo" nasce com o estado do anterior (a faísca que já nasce vermelha).

A regra: **cada objeto do Pool tem exatamente um dono**, e quem o devolve apaga toda referência que tinha a ele. Além disso, implemente `Pool.Poolable` com um `reset()` completo.

Q182 Troca de telas

> *No capstone, por que `setScreen` do `Game` **não** chama `dispose()` na tela antiga, e por que a troca de tela precisava ser a **última** linha do `render`?*

O `Game.setScreen` chama `hide()` na tela antiga e `show()` na nova, e **não** libera a antiga, porque o jogo pode querer voltar a ela (um menu que se reaproveita, por exemplo). Quem decide descartar é o programador, e o jogo centralizou isso em `trocarPara`, que chama `dispose()` logo depois de `setScreen`.

Por isso a troca é a última coisa do `render`: depois dela, a tela atual **já foi descartada** (o desenhista de mapas e o *pool* de partículas foram liberados), e qualquer código que continuasse rodando usaria recursos mortos, com o risco de derrubar o processo, no caso de recursos nativos.

Q183 Distribuir o jogo

> *Quais são os três degraus para distribuir o jogo, do `.jar` ao programa que roda sem Java instalado? O que cada um resolve?*

1. JAR gordo (`./gradlew lwjgl3:jar`): um arquivo com o jogo, o libGDX, o LWJGL, as bibliotecas nativas e os *assets*. Roda com `java -jar`, mas **exige Java 25** na máquina do jogador.

2. jlink: um runtime com só os módulos necessários (no teste, `java.base`, `java.desktop` e `jdk.unsupported`, 61 MB, contra algumas centenas de um JDK completo).

3. jpackage (ou o **Construo**, que empacota para Windows, macOS e Linux a partir de um único sistema): junta o JAR e o runtime num executável ou instalador nativo (a pasta de 76 MB, ou 50 MB compactada em `.tar.gz`). O jogador abre com dois cliques, sem saber que existe Java por trás.

Exercícios de código — soluções

Q184 Passo fixo com acumulador

*Escreva um game loop de **passo fixo com acumulador** em Java puro (sem libGDX), que simule uma bola caindo sob gravidade de −9,8 m/s² durante 1 segundo "real", recebido em quadros de tamanhos irregulares (0,016 s, 0,033 s, 0,005 s...). Mostre que o resultado depende só do número de passos, e não do tamanho dos quadros.*

```
static final float PASSO = 1f / 60f;
```

```
/** Simula a bola com os quadros dados; devolve a posição final. */
float simular(float[] quadros) {
    float y = 10f, vy = 0f, acumulado = 0f;
    int passos = 0;
    for (float dt : quadros) {
        acumulado += Math.min(dt, 0.25f);                 // trava contra a "espiral da morte"
        while (acumulado >= PASSO) {                      // quantos passos fixos couberem
            vy += -9.8f * PASSO;                          // Euler semi-implícito: velocidade
primeiro
            y += vy * PASSO;
            acumulado -= PASSO;
            passos++;
        }
    }
    IO.println("passos: " + passos + ", y = %.5f".formatted(y));
    return y;
}
```

```
void main() {
    float[] regulares = new float[61];
    java.util.Arrays.fill(regulares, 1f / 60f);
    regulares[60] = 0.001f;                               // arredondamentos de float: um
empurrãozinho
    float[] irregulares = {0.016f, 0.033f, 0.005f, 0.1f, 0.2f, 0.25f, 0.2f, 0.1f, 0.09f, 0.007f};
    float a = simular(regulares), b = simular(irregulares);
    IO.println(a == b ? "mesmo resultado" : "diferente");
}
```

Saída

```
passos: 60, y = 5.01833
passos: 60, y = 5.01833
mesmo resultado
```

Os dois lados executam exatamente 60 passos de 1/60 s, e por isso chegam ao **mesmo** `y`, bit a bit, embora os quadros tenham tamanhos completamente diferentes. É o mesmo `y = 5,018333` do Box2D numa aula anterior (1 segundo de queda livre a partir de 10 m, com a mesma integração). A trava de 0,25 s impede que um quadro muito longo (o jogo ficou parado, ou o computador engasgou) gere centenas de passos de uma vez.

Q185 AABB contra a grade

*Implemente a resolução de colisão de uma caixa (AABB) contra uma **grade de tiles**, um eixo de cada vez: mova em X, corrija se entrou em bloco sólido, e depois faça o mesmo em Y, marcando `noChao` quando a correção for para cima.*

```
static final float LARG = 0.7f, ALT = 0.9f;
static final String[] MAPA = {      // primeira linha = topo
        "......",
        "....#.",
        "######"};
```

```
boolean solido(int col, int lin) {
    int linhas = MAPA.length;
    if (col < 0 || col >= MAPA[0].length()) return true;           // laterais: paredes
    if (lin < 0 || lin >= linhas) return false;                    // abaixo e acima do mapa:
vazio
    return MAPA[linhas - 1 - lin].charAt(col) == '#';              // o texto cresce para baixo;
o mundo, para cima
}
```

```
float x = 1f, y = 1.5f;
boolean noChao;
```

```
void mover(float dx, float dy) {
    x += dx;                                                       // 1) eixo X
    int lb = (int) Math.floor(y + 0.001f), lc = (int) Math.floor(y + ALT - 0.001f);
    if (dx > 0) { int c = (int) Math.floor(x + LARG); for (int l = lb; l <= lc; l++) if
(solido(c, l)) { x = c - LARG; break; } }
    if (dx < 0) { int c = (int) Math.floor(x);        for (int l = lb; l <= lc; l++) if
(solido(c, l)) { x = c + 1;    break; } }
```

```
    y += dy;                                                       // 2) eixo Y
    noChao = false;
    int ce = (int) Math.floor(x + 0.001f), cd = (int) Math.floor(x + LARG - 0.001f);
    if (dy < 0) { int l = (int) Math.floor(y);       for (int c = ce; c <= cd; c++) if (solido(c,
l)) { y = l + 1; noChao = true; break; } }
    if (dy > 0) { int l = (int) Math.floor(y + ALT); for (int c = ce; c <= cd; c++) if (solido(c,
l)) { y = l - ALT; break; } }
}
```

```
void main() {
    for (int i = 0; i < 10; i++) mover(0f, -0.2f);                 // cai até o chão
    IO.println("depois de cair: y=%.2f noChao=%b".formatted(y, noChao));
    for (int i = 0; i < 30; i++) mover(0.2f, -0.2f);               // anda até a parede na coluna
4
    IO.println("encostado na parede: x=%.2f".formatted(x));
}
```

Saída

```
depois de cair: y=1.00 noChao=true
encostado na parede: x=3.30
```

O chão é a linha 0, e o topo dela fica em y = 1. A parede está na coluna 4, e o herói encosta em x = 4 − 0,7 = 3,3. O `0.001` encolhe a caixa ao calcular as linhas e colunas ocupadas: sem ele, um herói apoiado exatamente em y = 1 "ocuparia" a linha do chão e trataria o próprio chão como parede ao andar.

Q186 Pool mínimo

> *Escreva um `Pool` genérico mínimo, com `obtain()`, `free()` e um contador de objetos criados. Use-o para simular 1.000 quadros em que 50 balas nascem e 50 morrem por quadro, e mostre quantos objetos foram criados de verdade.*

```
import java.util.function.Supplier;
```

```
static final class Pool<T> {
    private final ArrayDeque<T> livres = new ArrayDeque<>();
    private final Supplier<T> fabrica;
    private final java.util.function.Consumer<T> reset;
    int criados;
```

```
    Pool(Supplier<T> fabrica, java.util.function.Consumer<T> reset) { this.fabrica = fabrica;
this.reset = reset; }
```

```
    T obtain() {
        T t = livres.poll();
        if (t == null) { criados++; return fabrica.get(); }
        return t;
    }
```

```
    void free(T t) {
        reset.accept(t);                                    // limpa o estado ANTES de guardar
        livres.push(t);
    }
}
```

```
static final class Bala { float x, y, vida; }
```

```
void main() {
    var pool = new Pool<Bala>(Bala::new, b -> { b.x = b.y = b.vida = 0; });
    List<Bala> vivas = new ArrayList<>();
    for (int quadro = 0; quadro < 1_000; quadro++) {
        for (int i = 0; i < 50; i++) { Bala b = pool.obtain(); b.vida = 1; vivas.add(b); }
        for (Bala b : vivas) pool.free(b);                  // todas morrem no fim do quadro
        vivas.clear();                                      // nenhuma referência sobra depois do
free
    }
    IO.println("balas usadas: " + 1_000 * 50 + ", objetos criados: " + pool.criados);
}
```

Saída

```
balas usadas: 50000, objetos criados: 50
```

Das 50 mil balas usadas, só 50 objetos foram criados: os do primeiro quadro, reaproveitados nos outros 999. O `vivas.clear()` logo depois dos `free` cumpre a regra do dono único. Um `Pool` de produção também teria um limite máximo de livres, como o do libGDX.

ENCERRAMENTO

O destino e o novo começo

Artigo 78

#78 O Destino e o Novo Começo — Encerrando a Jornada Completa

Perguntas de revisão

Q187 Princípios que atravessam o curso

> *A última aula listou seis princípios que atravessaram o curso inteiro. Escolha **três** e dê, para cada um, um exemplo concreto de **duas trilhas diferentes** em que ele apareceu.*

Exemplos possíveis (há outras boas respostas):

- **Todo recurso tem dono**: o `try` com recursos que fecha o *socket* no Farol; a conexão devolvida ao *pool* na Trilha A; o `dispose()` das texturas e o `free()` do `Pool` na Trilha C.
- **Desconfie da entrada**: o `Content-Length` com `Transfer-Encoding` recusado e o *path traversal* no Farol; o `PreparedStatement` e o escape do Thymeleaf na Trilha A; o arquivo de preferências editado à mão no capstone dos jogos.
- **Meça antes de mudar**: o JMH e o JFR na Fase 6; as 501 consultas do N+1 na Trilha A; os 74 bytes por passo do Asteroids e o pior quadro na Trilha C.
- **Regras longe da entrada e da saída**: o `Roteador` testado sem *socket*; o *ViewModel* testado sem janela; a `Fase` do plataforma testada sem GPU.

Q188 LTS e ritmo de lançamentos

> *Por que o Java 25 é a escolha certa para um projeto que vai para produção hoje, mesmo com o JDK 27 já lançado? Qual é o ritmo de lançamentos do Java?*

Desde o Java 10 (2018) sai uma versão a cada **seis meses** (março e setembro), e a cada **dois anos** uma delas é **LTS** (suporte longo): 17, 21, 25 e, se o ritmo se mantiver, 29, em setembro de 2027. As versões intermediárias, como o JDK 27 (setembro de 2026), recebem atualizações só até a versão seguinte sair. Em produção, a LTS dá anos de correções de segurança e o ecossistema inteiro (Spring, Hibernate, drivers, imagens Docker) validado. As versões novas entram no CI, numa matriz de testes, para descobrir incompatibilidades cedo e medir ganhos, como os cabeçalhos de objeto compactos, padrão no JDK 27.

Q189 Um plano possível

> *Monte um plano de estudos pessoal para os próximos 90 dias com uma meta concreta por mês, escolhendo **um** destino (back-end e nuvem, JVM por dentro, Android, jogos ou código aberto). Para cada mês, diga como você vai saber que cumpriu a meta.*

Uma resposta pessoal não tem gabarito único. Um bom plano tem metas **verificáveis**. Exemplo para o destino *back-end*:

- **Mês 1**: publicar o Farol, as Trilhas da Ilha e o jogo no GitHub, com README (o que é, como rodar, capturas, a seção "Decisões") e CI verde. *Cumprido quando* um amigo clonar um dos projetos e o rodar seguindo só o README.
- **Mês 2**: fazer a API das Trilhas publicar um evento "inscrição criada" no Kafka e escrever um consumidor que envia o e-mail de confirmação, com teste de integração usando Testcontainers. *Cumprido quando* o teste passar no CI.
- **Mês 3**: uma contribuição aceita num projeto de código aberto usado no curso (uma correção de documentação já conta) e um texto publicado explicando algo aprendido. *Cumprido quando* o *pull request* for aceito e o texto estiver no ar.

PARA IR ALÉM

Desafios integradores

Projetos maiores, que juntam várias fases e trilhas. Não há uma única resposta certa: o gabarito mostra um caminho.

D1

O Farol com HTTPS e compressão

Estenda o servidor Farol (Fase 9) para: (a) aceitar conexões TLS com `SSLServerSocket` e um certificado autoassinado gerado com `keytool`; (b) comprimir com gzip as respostas de texto quando o cliente enviar `Accept-Encoding: gzip`, acrescentando `Content-Encoding` e `Vary`; (c) não comprimir respostas pequenas (menos de 1 KB) nem imagens; (d) manter os testes existentes passando e acrescentar testes com `HttpClient` para os dois recursos.

Caminho da solução

- **TLS**: gere o certificado com `keytool -genkeypair -alias farol -keyalg EC -keystore farol.p12 -storetype PKCS12`. Carregue-o num `KeyStore`, crie um `KeyManagerFactory` e um `SSLContext.getInstance("TLS")`, e troque o `new ServerSocket(porta)` por `contexto.getServerSocketFactory().createServerSocket(porta)`. O resto do servidor não muda: `SSLSocket` é um `Socket`. Restrinja os protocolos a `TLSv1.3` e `TLSv1.2`.
- **gzip**: no `EscritorDeResposta`, antes de escrever um `Conteudo.Bytes` de tipo `text/*`, `application/json` ou `application/javascript` com mais de 1.024 bytes, comprima com `GZIPOutputStream` num `ByteArrayOutputStream` e ajuste o `Content-Length` para o tamanho comprimido. `Vary: Accept-Encoding` avisa os *caches* de que a resposta muda conforme esse cabeçalho.
- **Decisão importante**: imagens (PNG, JPEG) e arquivos já comprimidos ficam de fora, porque comprimir de novo gasta CPU e pode até aumentar o tamanho.
- **Testes**: o `HttpClient` precisa de um `SSLContext` que confie no certificado de teste. Ao pedir com `Accept-Encoding: gzip`, confira o cabeçalho e descomprima o corpo com `GZIPInputStream`.

D2

Inscrições concorrentes sem vender vaga a mais

*Nas Trilhas da Ilha, uma trilha guiada tem 10 vagas. Escreva o serviço de inscrição (Spring + PostgreSQL) e um teste de integração com Testcontainers que dispare 50 inscrições **simultâneas** (com virtual threads) e prove que exatamente 10 são aceitas e 40 recebem **409 Conflict**. Compare três estratégias: `SELECT ... FOR UPDATE`, um `UPDATE` atômico condicional e o bloqueio otimista com `@Version`.*

Caminho da solução

- **Atômica (a mais simples e rápida)**: `UPDATE trilha SET vagas = vagas - 1 WHERE id = ? AND vagas > 0`. Se `linhas afetadas == 0`, não há vaga: 409. Depois, na mesma transação, `INSERT` da inscrição. Um `CHECK (vagas >= 0)` na tabela é a rede de segurança.

- **FOR UPDATE**: `SELECT vagas FROM trilha WHERE id = ? FOR UPDATE` trava a linha até o fim da transação. As outras 49 esperam na fila. É correto, porém mais lento sob disputa.
- **Otimista** (`@Version`): quem grava por último recebe `OptimisticLockException` e precisa **tentar de novo**. Com 50 disputando 10 vagas, há muitas repetições, e o método é bom quando o conflito é raro, e não aqui.
- **Teste**: `Executors.newVirtualThreadPerTaskExecutor()`, 50 tarefas chamando o serviço, e a contagem de sucessos e de 409. Confira no banco que `vagas = 0` e que há exatamente 10 inscrições. Rode o teste várias vezes: as condições de corrida aparecem de forma intermitente. Uma versão ingênua ("ler vagas, se > 0 gravar vagas − 1") deve **falhar** o teste. Escreva-a primeiro para ver o vermelho.

D3

Painel desktop com atualização ao vivo

*Estenda o Painel JavaFX (Trilha B) para mostrar as inscrições de uma trilha **em tempo real**: a API passa a expor um endpoint de Server-Sent Events (`text/event-stream`) que emite um evento a cada inscrição, e o Painel consome o fluxo com `HttpClient` (`BodyHandlers.ofLines()`), atualizando uma lista e um contador de vagas sem travar a interface. Trate a queda de conexão com reconexão e espera exponencial.*

Caminho da solução

- **API**: no Spring MVC, um método que devolve `SseEmitter`. O serviço de inscrições publica um evento (`ApplicationEventPublisher`), e um componente mantém os *emitters* abertos e envia `event: inscricao` / `data: {json}`. Remova os *emitters* em `onCompletion` e `onTimeout`.
- **Painel**: `http.sendAsync(pedido, BodyHandlers.ofLines())` devolve um `Stream<String>`. Consuma-o numa *virtual thread*, acumulando as linhas `data:` até a linha vazia que fecha cada evento, converta com Jackson e publique no *ViewModel* com o `Executor` da interface (`Platform::runLater`).
- **Reconexão**: ao terminar ou falhar o fluxo, espere 1 s, 2 s, 4 s... (limite de 30 s, com um pouco de aleatoriedade) e reconecte, enviando `Last-Event-ID` para não perder eventos. Uma propriedade `conectado` no *ViewModel* mostra o estado na tela.
- **Teste**: com um cliente *fake* que emite eventos numa `SubmissionPublisher`, verifique o *ViewModel* sem abrir janela.

D4

Uma terceira fase com inimigos

*Acrescente ao jogo A Ilha dos Blocos (Trilha C) uma terceira fase com **inimigos** que andam de um lado para o outro nas plataformas e viram na borda ou numa parede. Encostar num inimigo pela lateral faz o herói voltar ao início, e pular em cima dele o elimina e dá um pequeno impulso para cima. A regra precisa estar na `Fase`, testada sem GPU, e o piloto automático precisa continuar vencendo **todas** as fases.*

Caminho da solução

- **Modelo**: `Inimigo` com posição, velocidade e direção, que usa a mesma `Grade` e a mesma colisão por eixos do herói. Na borda, se a célula abaixo da frente não for sólida, ou numa parede, ele inverte a direção.

- **Colisão herói × inimigo**: AABB contra AABB. Se o herói está **descendo** (`vy < 0`) e os pés dele estavam acima do meio do inimigo no passo anterior, é um pisão: o inimigo é removido, o herói recebe `vy = PULO * 0,6` e um evento `derrotouInimigo` é emitido para as partículas. Caso contrário, é uma queda: volta ao início.
- **Mapa**: um caractere novo no `Mapas` (por exemplo, `'i'`) e um objeto `inimigo` no TMX, lido pelo nome no `FaseDoTiled`.
- **Testes**: um inimigo vira na borda; o pisão elimina; o encontro lateral reinicia; e o piloto (que agora também pula quando há inimigo à frente) vence as três fases sem quedas.
- **Cuidado de projeto**: os inimigos nascem de novo quando o herói cai? Decida e teste.

D5

Métricas do Farol com JFR e um painel

*Instrumente o Farol com **eventos JFR personalizados** (`jdk.jfr.Event`): um evento por requisição com método, rota, status e duração. Grave uma sessão de carga (gerada com um programa de virtual threads fazendo 100 mil requisições) e escreva um analisador, com a API `jdk.jfr.consumer.RecordingFile`, que produza a tabela de latência p50/p95/p99 por rota e a taxa de erros. Compare os números com o servidor usando o G1 e o ZGC.*

Caminho da solução

- **Evento**: `@Name("farol.Requisicao") @Label("Requisição HTTP") class EventoDeRequisicao extends Event { @Label("Rota") String rota; String metodo; int status; }`. Na `Conexao`: `var e = new EventoDeRequisicao(); e.begin();`, e depois de escrever a resposta, `e.rota = ...; e.status = ...; e.commit();`. A duração é medida pelo próprio JFR, e o custo é quase zero quando a gravação está desligada.
- **Carga**: `newVirtualThreadPerTaskExecutor` com `HttpClient` e *keep-alive*, e uma parte das requisições para rotas inexistentes (404) e com corpos inválidos (400).
- **Gravação**: `java -XX:StartFlightRecording=filename=carga.jfr,settings=profile -jar farol.jar`.
- **Análise**: `RecordingFile.readAllEvents(Path)`, filtrar por `getEventType().getName()`, agrupar por rota com `Collectors.groupingBy`, ordenar as durações e pegar os percentis por índice. Imprima uma tabela em Markdown, como o inventário da aula 78.
- **Conclusão esperada**: o p50 quase não muda entre os coletores, e o p99 costuma cair com o ZGC. Meça, e não suponha: o resultado depende da alocação por requisição.

D6

A jornada completa em um só sistema

Integre as três trilhas: o jogo envia a pontuação de cada fase vencida para a API das Trilhas (autenticado com JWT); a API valida a pontuação contra trapaças (tempo mínimo possível para a fase, moedas no máximo o total da fase, assinatura do jogo), grava no PostgreSQL e mantém um ranking; o site mostra o ranking ao vivo com htmx; e o Painel desktop permite a um administrador anular pontuações suspeitas. Escreva um documento de arquitetura de uma página e implemente pelo menos o caminho jogo → API → site.

Caminho da solução

- **Contrato**: `POST /api/pontuacoes` com `{fase, tempo, moedas, versaoDoJogo}` e `Authorization: Bearer <jwt>`. Resposta 201, ou 422 se a validação anti-trapaça recusar.
- **Anti-trapaça** (sem ilusões: o cliente está na mão do jogador): (1) regras de plausibilidade do lado do servidor, como o tempo não ser menor que o do piloto automático menos uma margem e as moedas não passarem do total do mapa; (2) limite de envios por usuário; (3) para quem quiser ir além, o jogo grava a sequência de comandos, e o servidor **reexecuta a fase**, com a mesma `Fase` determinística de passo fixo (o mesmo código, num módulo compartilhado), e confere o resultado. É aqui que o passo fixo da Trilha C paga o investimento.
- **Ranking**: uma tabela `pontuacao`, um índice por `(fase, tempo)` e uma *view* com `ROW_NUMBER() OVER (PARTITION BY fase ORDER BY tempo)` para o melhor tempo de cada jogador.
- **Site**: um fragmento Thymeleaf com `hx-get="/ranking/1" hx-trigger="every 5s"`, ou SSE, como no D3.
- **Painel**: uma tela de administração que lista as pontuações com a menor margem de plausibilidade e chama `DELETE /api/pontuacoes/{id}` (papel ADMIN, 403 para os outros).
- **Documento**: um diagrama com os componentes, o fluxo de uma pontuação, as decisões (e o que foi trocado por elas) e os riscos conhecidos.

D7

Uma calculadora, duas interfaces

*Escreva uma calculadora com sete operações: soma, subtração, multiplicação, divisão, potência (expoente inteiro), porcentagem (`15 % 200` é 15% de 200) e raiz quadrada. Ela deve: (a) fazer as contas com `BigDecimal`, para que `0,1 + 0,2` dê exatamente `0,3`, aceitando vírgula ou ponto e limitando o resultado a 16 dígitos significativos; (b) guardar o último resultado em `ans`, que pode ser usado na conta seguinte; (c) recusar divisão por zero, raiz de negativo, expoente fracionário e texto que não é número com uma mensagem clara, **sem encerrar o programa**. Escreva o núcleo numa classe que não sabe nada de entrada e saída, e reaproveite-o em duas interfaces: uma de **terminal** (com os comandos `hist` e `sair`) e uma **gráfica** em JavaFX, com teclado na tela, visor e histórico.*

Caminho da solução

- **Núcleo**: um `enum Operacao` com o símbolo de cada operação e uma classe `Calculadora` que só calcula. O `switch` sobre o `enum` é exaustivo: se alguém acrescentar uma oitava operação e esquecer de tratá-la, o código não compila.
- **Precisão**: `MathContext.DECIMAL64` (16 dígitos) na divisão, na potência negativa e na raiz, que são as operações capazes de gerar dízimas. O resultado guardado em `ans` já sai arredondado, como no visor de uma calculadora de mesa.
- **Erros**: o núcleo lança `ArithmeticException` (conta impossível) ou `IllegalArgumentException` (entrada mal escrita). As duas interfaces capturam essas exceções, mostram a mensagem e seguem funcionando.
- **Interfaces**: o terminal lê linha a linha com `IO.readln`, e a janela JavaFX monta o mesmo texto ("12,5 + 3") com os botões e entrega a `avaliar`. Nenhuma regra de conta aparece fora da `Calculadora`.

`Calculadora.java`, o núcleo:

```
import java.math.BigDecimal;
import java.math.MathContext;

enum Operacao {
    SOMA("+"), SUBTRACAO("-"), MULTIPLICACAO("*"), DIVISAO("/"),
    POTENCIA("^"), PORCENTAGEM("%"), RAIZ("raiz");

    final String simbolo;

    Operacao(String simbolo) { this.simbolo = simbolo; }

    static Operacao de(String simbolo) {
        for (var op : values()) if (op.simbolo.equals(simbolo)) return op;
        throw new IllegalArgumentException("operação desconhecida: " + simbolo);
    }
}

/** O núcleo: não sabe nada de terminal nem de janela, e por isso as duas interfaces o
reaproveitam. */
final class Calculadora {
    private static final MathContext PRECISAO = MathContext.DECIMAL64;   // 16 dígitos
significativos
    private static final BigDecimal CEM = BigDecimal.valueOf(100);

    private BigDecimal ultimo = BigDecimal.ZERO;                          // o "ans" das
calculadoras

    BigDecimal calcular(Operacao op, BigDecimal a, BigDecimal b) {
        return switch (op) {
            case SOMA -> a.add(b);
            case SUBTRACAO -> a.subtract(b);
            case MULTIPLICACAO -> a.multiply(b);
            case DIVISAO -> {
                if (b.signum() == 0) throw new ArithmeticException("divisão por zero");
                yield a.divide(b, PRECISAO);
            }
            case POTENCIA -> {
                if (b.stripTrailingZeros().scale() > 0) throw new ArithmeticException("o expoente
precisa ser inteiro");
                int expoente = b.intValue();
                if (Math.abs(expoente) > 999) throw new ArithmeticException("expoente fora de
-999 a 999");
                if (a.signum() == 0 && expoente < 0) throw new ArithmeticException("divisão por
zero");
                yield a.pow(expoente, PRECISAO);
            }
            case PORCENTAGEM -> a.multiply(b).divide(CEM, PRECISAO);       // "15 % 200" = 15% de
200
            case RAIZ -> {
                if (a.signum() < 0) throw new ArithmeticException("raiz de número negativo");
                yield a.sqrt(PRECISAO);
```

```
            }
        };
    }

    /** Avalia "12,5 + 3", "2 ^ 10", "15 % 200" ou "raiz 2"; "ans" é o último resultado. */
    BigDecimal avaliar(String linha) {
        String[] t = linha.trim().split("\\s+");
        BigDecimal resultado = switch (t.length) {
            case 2 -> calcular(Operacao.de(t[0]), numero(t[1]), BigDecimal.ZERO);
            case 3 -> calcular(Operacao.de(t[1]), numero(t[0]), numero(t[2]));
            default -> throw new IllegalArgumentException("use: número operação número, ou raiz
número");
        };
        if (t.length == 2 && Operacao.de(t[0]) != Operacao.RAIZ)
            throw new IllegalArgumentException("só a raiz tem um operando");
        ultimo = resultado.round(PRECISAO);                                  // como no visor: 16
dígitos
        return ultimo;
    }

    private BigDecimal numero(String texto) {
        if (texto.equalsIgnoreCase("ans")) return ultimo;
        try {
            return new BigDecimal(texto.replace(',', '.'));
        } catch (NumberFormatException e) {
            throw new IllegalArgumentException("não é um número: " + texto);
        }
    }

    static String formatar(BigDecimal valor) {
        BigDecimal v = valor.stripTrailingZeros();
        boolean enorme = v.precision() - v.scale() > 16;                  // mais de 16 dígitos
inteiros
        return (enorme ? v.toString() : v.toPlainString()).replace('.', ',');
    }
}
```

`CalculadoraTerminal.java`:

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

public class CalculadoraTerminal {
    public static void main(String[] args) {
        var calc = new Calculadora();
        List<String> historico = new ArrayList<>();
        IO.println("Calculadora: + - * / ^ % raiz  |  ans, hist, sair");
        String linha;
        while ((linha = IO.readln("> ")) != null) {            // null: fim da entrada (Ctrl+D)
            linha = linha.trim();
            if (linha.isEmpty()) continue;
            if (linha.equals("sair")) break;
            if (linha.equals("hist")) {
                historico.forEach(IO::println);
                continue;
            }
            try {
                BigDecimal r = calc.avaliar(linha);
                String conta = linha + " = " + Calculadora.formatar(r);
                historico.add(conta);
                IO.println(conta);
            } catch (ArithmeticException | IllegalArgumentException e) {
                IO.println("erro: " + e.getMessage());          // a conta errada não derruba o
programa
            }
        }
    }
}
```

Com a entrada redirecionada, o terminal não ecoa o que foi digitado e cada resultado aparece logo depois do `>`:

```
printf '0,1 + 0,2\n10 / 3\nans * 3\n1 / 0\n2 ^ 100\nraiz 2\nans * ans\n15 %% 200\n2 ^ 0,5\nraiz
-4\nabc + 1\nhist\nsair\n' | java CalculadoraTerminal.java
```

Saída

```
Calculadora: + - * / ^ % raiz  |  ans, hist, sair
> 0,1 + 0,2 = 0,3
> 10 / 3 = 3,333333333333333
> ans * 3 = 9,999999999999999
> erro: divisão por zero
> 2 ^ 100 = 1,267650600228229E+30
> raiz 2 = 1,414213562373095
> ans * ans = 2
> 15 % 200 = 30
> erro: o expoente precisa ser inteiro
> erro: raiz de número negativo
> erro: não é um número: abc
> 0,1 + 0,2 = 0,3
10 / 3 = 3,333333333333333
ans * 3 = 9,999999999999999
2 ^ 100 = 1,267650600228229E+30
raiz 2 = 1,414213562373095
ans * ans = 2
15 % 200 = 30
>
```

Repare em `10 / 3` seguido de `ans * 3`: o resultado é `9,999999999999999`, e não `10`, porque `ans` guardou a dízima cortada em 16 dígitos. As calculadoras de bolso escondem isso trabalhando internamente com alguns dígitos a mais do que mostram no visor. Fazer o mesmo aqui é uma boa extensão.

`CalculadoraFX.java`, a versão gráfica. O JavaFX não vem com o JDK: baixe os módulos `javafx-base`, `javafx-`

`graphics` e `javafx-controls` (ou o SDK do openjfx.io) e aponte o `--module-path` para eles:

```
import java.math.BigDecimal;
import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.ListView;
import javafx.scene.control.TextField;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class CalculadoraFX extends Application {
    private final Calculadora calc = new Calculadora();           // o mesmo núcleo do terminal
    private final TextField visor = new TextField();
    private final Label erro = new Label();
    private final ListView<String> historico = new ListView<>();

    // Cada botão só escreve no visor; quem entende a conta é o Calculadora.avaliar.
    private static final String[][] TECLAS = {
        {"7", "8", "9", " / ", "C"},
        {"4", "5", "6", " * ", "raiz "},
        {"1", "2", "3", " - ", " ^ "},
        {"0", ",", "ans", " + ", " % "},
    };

    @Override
    public void start(Stage palco) {
        visor.setPromptText("12,5 + 3   ou   raiz 2");
        visor.setStyle("-fx-font-size: 20px; -fx-alignment: center-right;");
        visor.setOnAction(e -> calcular());                        // Enter também calcula
        erro.setStyle("-fx-text-fill: firebrick;");

        var grade = new GridPane(6, 6);
        for (int lin = 0; lin < TECLAS.length; lin++)
            for (int col = 0; col < TECLAS[lin].length; col++)
                grade.add(botao(TECLAS[lin][col]), col, lin);
        var igual = new Button("=");
        igual.setMaxWidth(Double.MAX_VALUE);
        igual.setOnAction(e -> calcular());
        grade.add(igual, 0, TECLAS.length, TECLAS[0].length, 1);

        var raiz = new BorderPane(new VBox(8, visor, erro, grade), null, null, historico, null);
        historico.setPrefHeight(140);
        BorderPane.setMargin(historico, new Insets(8, 0, 0, 0));
        raiz.setPadding(new Insets(12));

        palco.setTitle("Calculadora");
        palco.setScene(new Scene(raiz));
        palco.show();
```

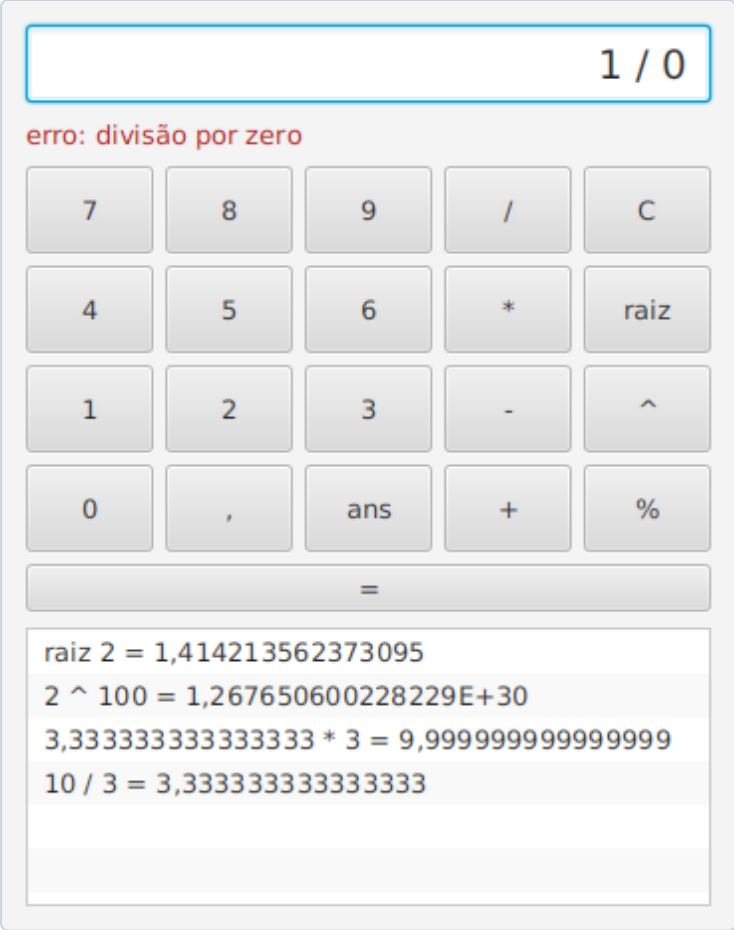
```
    }

    private Button botao(String tecla) {
        var b = new Button(tecla.trim());
        b.setPrefSize(64, 44);
        b.setOnAction(e -> {
            if (tecla.equals("C")) visor.clear();
            else visor.appendText(tecla);
            visor.requestFocus();
            visor.end();
        });
        return b;
    }

    private void calcular() {
        String conta = visor.getText().trim();
        if (conta.isEmpty()) return;
        try {
            BigDecimal r = calc.avaliar(conta);
            historico.getItems().addFirst(conta + " = " + Calculadora.formatar(r));
            visor.setText(Calculadora.formatar(r));  // a próxima conta continua dele
            erro.setText("");
        } catch (ArithmeticException | IllegalArgumentException e) {
            erro.setText("erro: " + e.getMessage());  // a janela continua aberta
        }
        visor.end();
    }

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

```
javac --module-path javafx/lib --add-modules javafx.controls Calculadora.java CalculadoraFX.java
java --module-path javafx/lib --add-modules javafx.controls CalculadoraFX
```



A janela depois de `10 / 3`, `* 3`, `2 ^ 100`, `raiz 2` e uma tentativa de `1 / 0`.

Fim do gabarito.

Dominando Java em 1 Ano · theknowledgeisle.com