Laboratório 02 — I/O no terminal, `Scanner`, métodos e controlo de fluxo (120 min)

Unidade Curricular: Programação Orientada aos Objetos — Java
Modalidade: Laboratório em computador pessoal
Duração sugerida: 120 min

> **Nota de estrutura:** A **alínea 2)** apresenta o **enunciado** sob a forma de tarefas **a)**, **b)**, **c)**, … que os alunos devem realizar.
> A **alínea 3)** apresenta a **resolução guiada** de cada tarefa (**a)**, **b)**, **c)**, …) com explicações, justificação de decisões e ligações aos conceitos de POO.

---

1) Objetivos de aprendizagem

Ao terminar este laboratório, será capaz de:

- Ler dados do **terminal** de forma **robusta** com `Scanner`, evitando armadilhas (mistura de `nextInt`/`nextLine`, *locale*, etc.).
- Projetar **métodos pequenos e coesos**, com parâmetros e valor de retorno, reduzindo duplicações.
- Aplicar **controlo de fluxo** (`if/else`, `switch`, loops) e **validar entradas**, com mensagens de erro claras.
- Organizar um mini-projeto com **estrutura de pastas padronizada**, compilar e executar no terminal e no **VS Code**.
- Fazer **commits** passo-a-passo e explicar, no `README`, como compilar/testar e os **casos de teste** efetuados.

---

2) Enunciado — Tarefas a realizar (avaliação prática)

Realize as tarefas **a)** a **i)**. Cada tarefa tem **critérios de aceitação** e, quando indicado, **testes mínimos**.
Tenha em conta a explicação sobre as classes e métodos usados no apêndice final.
Se terminar cedo, avance para os desafios "Extensão".

a) Preparar ambiente e validar ferramentas

Objetivo: Garantir que o ambiente está funcional.
Validar versões no terminal do VS Code: `javac -version`, `java -version`, `git --version`.

Se os carateres não aparecerem corretamente acentuados, faça `chcp 65001` no terminal do Windows.

b) Estruturar o projeto e `.gitignore`

Objetivo: Criar a estrutura **padronizada** e o ignore file.
Estrutura esperada:

```
lab02/
  ├─ src/
  │   └─ main/
  │       └─ java/
  │           └─ pt/
```

```
│               └─ escnaval/
│                   └─ exercicios/
│                       ├─ LeituraRobusta.java
│                       ├─ CalcCLI.java
│                       ├─ ConversorUnidades.java
│                       ├─ EstatisticasSimples.java
│                       └─ UtilsIO.java
├─ README.md
├─ .gitignore
└─ out/
```

Aceitação: `.gitignore` contém `out/`, `*.class`, `.vscode/`, etc., e existe `README.md` inicial.

c) Iniciar repositório Git e primeiro commit

Objetivo: Controlar o progresso com commits atómicos.

1. `git init`, `git config user.name/email`.
2. Primeiro commit com a estrutura inicial e `.gitignore`.
   Aceitação: histórico com mensagem clara "Lab02: estrutura inicial".

d) Programa `LeituraRobusta` — leitura segura com `Scanner`

Objetivo: Ler **inteiro**, **double** e **linha** de forma **robusta**.
Requisitos:

- Usar **`Locale.US`**.
- Ler sempre via `nextLine()` e fazer `parse` (evitar `nextInt`/`nextDouble`).
- Mensagens de erro até a entrada ser válida.
- Imprimir no fim: `n`, `x` (3 casas), e `texto`.
  Testes mínimos:
- Inteiro inválido seguido de válido.
- Double inválido seguido de `3.14`.
- Linha de texto com espaços em ambas as extremidades.

e) Programa `CalcCLI` — operadores e `switch`

Objetivo: Calculadora simples que lê `a`, `b` e um operador em `{+,-,*,/,%}`.
Requisitos:

- Validar operador (pedir novamente se inválido).
- Impedir divisão/módulo por zero com **mensagem clara**.
- Calcular com `switch`.
  Testes mínimos: `10 / 3 → 3.3333`; operador inválido; `b=0` com `/` e `%`.

f) Programa `ConversorUnidades` — métodos "puros"

Objetivo: Converter **km↔mi** com métodos **sem I/O**.
Requisitos: menu `1) km→mi 2) mi→km`; validação da opção; `Locale.US`.
Testes mínimos: `10 km → 6.214 mi`; `1 mi → 1.609 km` (3 casas).

g) Programa `EstatisticasSimples` — loops e agregação

Objetivo: Ler **uma linha por valor** até **linha vazia**; ignorar não numéricos.
Requisitos: calcular `n`, `soma`, `média`, `min`, `max` num único percurso.
Testes mínimos: `1, 2, 3, "" → n=3, soma=6, média=2`; ignorar `x`.

3) Resolução guiada — comentários e justificação (a → i)

a) Preparar ambiente e validar ferramentas

- Abra o **Terminal** no VS Code (`Ctrl` + `` ` ``) e valide:

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

- **Porque?** Evita bloqueios: sem JDK/Git ok, não há progresso.

b) Estruturar o projeto e `.gitignore`

`.gitignore` sugerido:

```
out/
bin/
target/
build/
.vscode/
*.class
*.log
.DS_Store
Thumbs.db
```

> Dica: para manter diretórios vazios no Git, use `.gitkeep` (ex.: `out/.gitkeep`).

c) Iniciar repositório Git e primeiro commit

```
git init
git config user.name  "Seu.Nome"
git config user.email "seu.email@example.com"
echo "# Lab 02 — I/O no terminal, Scanner e métodos" > README.md
git add .
git commit -m "Lab02: estrutura inicial, README e .gitignore"
```

Justificação: Commits atómicos criam pontos de restauração e comunicação técnica.

d) `LeituraRobusta` — implementação

`src/main/java/pt/escnaval/exercicios/LeituraRobusta.java`:

```
package pt.escnaval.exercicios;

import java.util.Locale;
import java.util.Scanner;

public class LeituraRobusta {
    public static void main(String[] args) {
        // Use Locale fixo para garantir ponto decimal (.) ao ler doubles
        Locale.setDefault(Locale.US);

        try (Scanner sc = new Scanner(System.in)) {
            System.out.print("Introduza um inteiro: ");
            int n = lerInt(sc);
```

```
            System.out.print("Introduza um número real (double): ");
            double x = lerDouble(sc);

            System.out.print("Introduza um texto (linha): ");
            String linha = sc.nextLine().trim();

            System.out.printf("OK: n=%d, x=%.3f, texto=\"%s\"%n", n, x, linha);
        }
    }

    // Ler inteiro de forma robusta (consome linha e tenta converter)
    static int lerInt(Scanner sc) {
        while (true) {
            String s = sc.nextLine();
            try {
                return Integer.parseInt(s.trim());
            } catch (NumberFormatException e) {
                System.out.print("Inteiro inválido. Tente novamente: ");
            }
        }
    }

    // Ler double de forma robusta
    static double lerDouble(Scanner sc) {
        while (true) {
            String s = sc.nextLine();
            try {
                return Double.parseDouble(s.trim());
            } catch (NumberFormatException e) {
                System.out.print("Double inválido. Tente novamente (ex.: 3.14): ");
            }
        }
    }
}
```

Porquê esta abordagem? Evita a armadilha `nextInt/nextDouble` + `nextLine` (newline pendente). Ler a **linha toda** e *parsear* é mais previsível e testável.

Compilar e executar:

```
# a partir de lab02/
javac -d out src/main/java/pt/escnaval/exercicios/LeituraRobusta.java
java -cp out pt.escnaval.exercicios.LeituraRobusta
```

Checkpoint: Testar entradas inválidas → mensagens e repetição até valor válido.

e) `CalcCLI` — implementação (switch + validação)

`src/main/java/pt/escnaval/exercicios/CalcCLI.java`:

```
package pt.escnaval.exercicios;

import java.util.Locale;
import java.util.Scanner;

public class CalcCLI {
    public static void main(String[] args) {
        Locale.setDefault(Locale.US);
        try (Scanner sc = new Scanner(System.in)) {
            System.out.println("=== CALC CLI ===");
            double a = lerDouble(sc, "a");
            double b = lerDouble(sc, "b");
```

```
            char op = lerOperador(sc, "+ - * / %");

            double res = calcular(a, b, op);
            System.out.printf("Resultado: %.4f%n", res);
        }
    }

    static double lerDouble(Scanner sc, String nome) {
        System.out.print("Introduza " + nome + ": ");
        while (true) {
            String s = sc.nextLine();
            try { return Double.parseDouble(s.trim()); }
            catch (NumberFormatException e) { System.out.print("Número inválido. Tente
        }
    }

    static char lerOperador(Scanner sc, String menu) {
        System.out.print("Operador (" + menu + "): ");
        while (true) {
            String s = sc.nextLine().trim();
            if (s.length() == 1 && "+-*/%".indexOf(s.charAt(0)) >= 0) return s.charAt(0
            System.out.print("Operador inválido. Tente: " + menu + " → ");
        }
    }

    static double calcular(double a, double b, char op) {
        switch (op) {
            case '+': return a + b;
            case '-': return a - b;
            case '*': return a * b;
            case '/':
                if (b == 0.0) throw new IllegalArgumentException("Divisão por zero");
                return a / b;
            case '%':
                if (b == 0.0) throw new IllegalArgumentException("Módulo por zero");
                return a % b;
            default: throw new IllegalArgumentException("Operador desconhecido: " + op)
        }
    }
}
```

Justificação de design: separação de responsabilidades (leitura vs. cálculo), `switch` claro, tratamento explícito de erros de domínio.

Compilar/Executar:

```
javac -d out src/main/java/pt/escnaval/exercicios/CalcCLI.java
java -cp out pt.escnaval.exercicios.CalcCLI
```

f) `ConversorUnidades` — implementação (métodos puros)

`src/main/java/pt/escnaval/exercicios/ConversorUnidades.java`:

```
package pt.escnaval.exercicios;

import java.util.Locale;
import java.util.Scanner;

public class ConversorUnidades {
    public static void main(String[] args) {
        Locale.setDefault(Locale.US);
        try (Scanner sc = new Scanner(System.in)) {
            System.out.println("=== Conversor ===");
```

```
            System.out.print("Escolha (1) km→mi, (2) mi→km: ");
            int modo = lerOpcao(sc, 1, 2);
            double v = lerDouble(sc, "valor");

            double out = (modo == 1) ? kmParaMilhas(v) : milhasParaKm(v);
            System.out.printf("Resultado: %.3f%n", out);
        }
    }

    static int lerOpcao(Scanner sc, int min, int max) {
        while (true) {
            String s = sc.nextLine();
            try {
                int op = Integer.parseInt(s.trim());
                if (op < min || op > max) throw new IllegalArgumentException();
                return op;
            } catch (Exception e) {
                System.out.print("Opção inválida. Tente novamente: ");
            }
        }
    }

    static double lerDouble(Scanner sc, String nome) {
        System.out.print(nome + ": ");
        while (true) {
            String s = sc.nextLine();
            try { return Double.parseDouble(s.trim()); }
            catch (NumberFormatException e) { System.out.print("Número inválido. Tente
        }
    }

    // Métodos "puros": não leem nem escrevem no terminal
    static double kmParaMilhas(double km)    { return km * 0.621371; }
    static double milhasParaKm(double mi)    { return mi / 0.621371; }
}
```

POO e testabilidade: "métodos puros" separam lógica de I/O → testes unitários mais simples.

g) `EstatisticasSimples` — implementação (loops + agregação)

`src/main/java/pt/escnaval/exercicios/EstatisticasSimples.java`:

```
package pt.escnaval.exercicios;

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

public class EstatisticasSimples {
    public static void main(String[] args) {
        Locale.setDefault(Locale.US);
        List<Double> valores = new ArrayList<>();

        try (Scanner sc = new Scanner(System.in)) {
            System.out.println("=== Estatísticas Simples ===");
            System.out.println("Introduza valores (ENTER por linha). Linha vazia termir
            while (true) {
                String s = sc.nextLine();
                if (s.isBlank()) break;
                try {
                    valores.add(Double.parseDouble(s.trim()));
                } catch (NumberFormatException e) {
                    System.out.println("Ignorado (não é número).");
```

```
                }
            }
        }

        if (valores.isEmpty()) {
            System.out.println("(sem dados)");
            return;
        }

        double soma = 0, min = Double.POSITIVE_INFINITY, max = Double.NEGATIVE_INFINITY
        for (double v : valores) {
            soma += v;
            if (v < min) min = v;
            if (v > max) max = v;
        }
        double media = soma / valores.size();

        System.out.printf("n=%d, soma=%.3f, média=%.3f, min=%.3f, max=%.3f%n",
                valores.size(), soma, media, min, max);
    }
}
```

Notas: Finalização por linha vazia; validação "tolerante" (ignora entradas inválidas); cálculo de métricas num único percurso por eficiência.

Compilar tudo (Linux/macOS):

```
find src -name "*.java" -print0 | xargs -0 javac -d out
```

Compilar tudo (PowerShell – Windows):

```
Get-ChildItem -Recurse src -Filter *.java | % { $_.FullName } | % { & javac -d out $_ }
```

Executar (exemplos):

```
java -cp out pt.escnaval.exercicios.LeituraRobusta
java -cp out pt.escnaval.exercicios.CalcCLI
java -cp out pt.escnaval.exercicios.ConversorUnidades
java -cp out pt.escnaval.exercicios.EstatisticasSimples
```

Run/Debug no VS Code: instalar "Extension Pack for Java", abrir a pasta `exercicios` e usar os botões de execução nas classes `main`.

---

4) Resolução de problemas (FAQ)

- **Leituras "saltadas"** → não misture `nextInt`/`nextDouble` com `nextLine`; leia a **linha** e faça `parse`.
- **Decimal com vírgula** não aceite → com `Locale.US`, use `3.14`. Para aceitar vírgulas, normalize `,` → `.` antes do `parse`.
- **`Could not find or load main class`** → verifique `package`, `-cp out` e nome **total** da classe (`pt.escnaval.exercicios.Nome`).
- **Problemas de codificação** → forçar UTF-8 no terminal/VS Code.

---

5) Cronograma sugerido (120 min)

- **0–10 min** → a) Ambiente.
- **10–20 min** → b) Estrutura + `.gitignore`.
- **20–25 min** → c) Git init.
- **25–50 min** → d) LeituraRobusta.
- **50–80 min** → e) CalcCLI.
- **80–95 min** → f) ConversorUnidades.
- **95–110 min** → g) EstatisticasSimples.

---

6) Apêndice — Comandos úteis (colagem rápida)

Git "em 1 minuto"

```
git add .
git commit -m "Progresso do Lab 02"
git branch -M main
git remote add origin https://github.com/<utilizador>/lab02.git
git push -u origin main
```

Explicação das Classes e Métodos Usados

O Lab 02 foca em entrada/saída (I/O) robusta no terminal, validação de dados e controle de fluxo em Java. As classes mencionadas (`java.util.Locale`, `java.util.Scanner`, `java.util.ArrayList` e `java.util.List`) são da API padrão do Java (pacote `java.util`), usadas para internacionalização, leitura de entrada, e manipulação de coleções. Elas são essenciais para programas interativos e processamento de dados. Abaixo, explico cada uma com base no uso no lab, incluindo métodos principais e funcionalidades.

1. `java.util.Locale`

- **Funcionalidade geral:** Representa uma configuração regional (locale), que define regras para formatação de números, datas, moedas e textos com base em idioma e país. No Java, afeta como o `Scanner` ou `printf` interpretam decimais (ex.: ponto vs. vírgula).
- **Uso no Lab 02:** Definido como `Locale.US` para garantir que números decimais usem ponto (ex.: `3.14` em vez de `3,14`), evitando erros de parsing em entradas. Exemplo: `Locale.setDefault(Locale.US);` no início dos programas.
- **Métodos principais usados ou relacionados:**
  - `setDefault(Locale)`: Define o locale padrão para a JVM (usado para forçar consistência).
  - Outros: `getDefault()`, `Locale.US` (constante para inglês americano).
- **Por que usar:** Evita problemas de internacionalização; sem isso, o sistema pode usar o locale do SO (ex.: português com vírgula), causando falhas em `Double.parseDouble`.

2. `java.util.Scanner`

- **Funcionalidade geral:** Classe para ler entrada de dados de fontes como o terminal (`System.in`), arquivos ou strings. É "token-based", dividindo a entrada em partes (tokens) separadas por espaços ou quebras de linha.
- **Uso no Lab 02:** Usada para ler entradas do usuário de forma robusta (ex.: inteiros, doubles, linhas de texto). Sempre com `try-with-resources` para fechar automaticamente. Exemplo: `Scanner sc = new Scanner(System.in);` seguido de `sc.nextLine()` para ler linhas inteiras e evitar armadilhas como newline pendente.
- **Métodos principais usados:**
  - `nextLine()`: Lê a linha inteira (até ENTER), ideal para entrada robusta (usado em `lerInt`, `lerDouble`).
  - `close()`: Fecha o scanner (feito automaticamente com `try-with-resources`).

    - Relacionados: Embora o lab evite `nextInt()`/`nextDouble()` (por causa de bugs com `nextLine()`), eles leem tokens específicos.
- **Por que usar:** Permite leitura interativa e validação; mais flexível que `BufferedReader` para entradas simples.

3. `java.util.ArrayList`

- **Funcionalidade geral:** Implementação concreta de uma lista dinâmica (array redimensionável). Permite adicionar/remover elementos sem tamanho fixo, mantendo ordem de inserção. Parte da Collections Framework.
- **Uso no Lab 02:** Em `EstatisticasSimples`, armazena valores numéricos lidos (`List<Double> valores = new ArrayList<>();`), permitindo agregação (soma, média, min/max) em um loop.
- **Métodos principais usados:**
    - `add(E element)`: Adiciona um elemento à lista (ex.: `valores.add(v);`).
    - `size()`: Retorna o número de elementos (usado para `n` e média).
    - Outros relacionados: `isEmpty()` (verifica se vazia), `get(int index)` (acessa por posição).
- **Por que usar:** Flexível para coleções de tamanho variável; mais eficiente que arrays fixos para entradas dinâmicas.

4. `java.util.List`

- **Funcionalidade geral:** Interface que define o contrato para listas ordenadas (coleções sequenciais). Não é uma classe concreta, mas uma abstração implementada por classes como `ArrayList` ou `LinkedList`. Permite operações como adicionar, remover e acessar por índice.
- **Uso no Lab 02:** Declarada como tipo (`List<Double> valores`), mas instanciada como `ArrayList` (`new ArrayList<>()`). Promove polimorfismo e foco na interface (boa prática POO).
- **Métodos principais (da interface, usados indiretamente via ArrayList):**
    - `add(E)`, `size()`, `isEmpty()` (mesmos de ArrayList).
    - Outros: `remove(int)`, `clear()` (não usados no lab, mas comuns).
- **Por que usar:** Separa a abstração da implementação; facilita testes e mudanças futuras (ex.: trocar para `LinkedList` sem alterar código).

Como Consultar a Documentação

A documentação oficial dessas classes está na **Java API Documentation** (Javadoc), mantida pela Oracle. É gratuita e detalhada, com exemplos, métodos e versões.

- **Acesso online:**
    - Vá para docs.oracle.com/javase/17/docs/api/ (ou versão atual, ex.: 21).
    - Navegue por `java.util` → clique na classe (ex.: `Locale`, `Scanner`).
    - Busque por métodos específicos (ex.: "nextLine" em Scanner).
- **Via IDE (VS Code):**
    - Com a "Extension Pack for Java" instalada, passe o mouse sobre a classe/método para ver tooltips com resumo.
    - Pressione `Ctrl+Espaço` ou `F12` para ir à definição (abre Javadoc inline).
    - Para documentação completa, use `Ctrl+Shift+P` → "Java: Open Java Language Server Log" ou pesquise online.
- **Dicas:** Sempre consulte a versão do JDK em uso (ex.: 17 ou 21). Para exemplos práticos, veja tutoriais no Oracle ou Stack Overflow. No lab, evite misturar `Scanner` com outras leituras para robustez. Se precisar de código de exemplo, posso gerar snippets!