



PROGRAMAÇÃO DE COMPUTADORES 2

PROF. EDERSON / PROF^a. JULIANA

REVISÃO PROGRAMAÇÃO DE COMPUTADORES 1

Aula 01 – 29/07/2025



SINTAXE BÁSICA DO JAVA

- Estrutura de um programa
- Tipos primitivos
- Operadores
- Estruturas de controle

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



CLASSES E OBJETOS

- Classe → modelo
- Objeto → instância

```
class Conta {  
    double saldo;  
    void depositar(double valor) {  
        saldo += valor;  
    }  
}
```

```
public class Programa {  
    public static void main(String[] args) {  
        Conta contaDoJoao = new Conta();  
        Conta contaDaMaria = new Conta();  
        contaDoJoao.depositar(500.0);  
        contaDaMaria.depositar(300.0);  
    }  
}
```



ENCAPSULAMENTO E HERANÇA

- Modificadores de acesso (`private`, `protected`, `public`)
- Getters e setters
- `extends` para reutilização de código
- Polimorfismo simples com sobrescrita (`@Override`)

```

public class Pessoa {
    private String nome;
    private int idade;
    public Pessoa(String nome, int idade) {
        this.nome = nome;
        this.idade = idade;
    }
    public String getNome() {
        return nome;
    }
    public void setNome(String nome) {
        this.nome = nome;
    }
    public int getIdade() {
        return idade;
    }
    public void setIdade(int idade) {
        if (idade >= 0) {
            this.idade = idade;
        }
    }
    public void apresentar() {
        System.out.println("Olá! Meu nome é " + nome + " e tenho " + idade + " anos.");
    }
}

```

SUPERCLASSE

```

public class Aluno extends Pessoa {
    private String curso;
    public Aluno(String nome, int idade, String curso) {
        super(nome, idade);
        this.curso = curso;
    }
    public String getCurso() {
        return curso;
    }
    public void setCurso(String curso) {
        this.curso = curso;
    }
    @Override
    public void apresentar() {
        System.out.println("Sou o aluno " + getNome() + ", do curso de " + curso);
    }
}

```



INSTITUTO
FEDERAL
São Paulo

Campus
Caraguatatuba

SUBCLASSE



INSTITUTO
FEDERAL
São Paulo

Campus
Caraguatatuba

ENCAPSULAMENTO E HERANÇA

```
public class Programa {  
    public static void main(String[] args) {  
        Pessoa p1 = new Pessoa("Carlos", 30);  
        p1.apresentar();  
  
        Aluno a1 = new Aluno("Ana", 20, "ADS");  
        a1.apresentar();  
    }  
}
```



CLASSES ABSTRATAS E INTERFACE

Classe Abstrata: (`extends`)

- Não pode ser instanciada diretamente.
- Serve como **modelo** para subclasses.
- Pode ter **métodos concretos** e **métodos abstratos** (sem corpo).
- Subclasses precisam implementar os métodos abstratos.

Conceito de Interface: (`implements`)

- Um **contrato**: só contém métodos (assinaturas) e constantes.
- Uma classe pode implementar várias interfaces.
- Garante que diferentes classes tenham o mesmo conjunto de métodos.



INSTITUTO
FEDERAL
São Paulo

Campus
Caraguatatuba

```
public abstract class Forma {  
    public abstract double calcularArea();  
    public void imprimirTipo() {  
        System.out.println("Sou uma forma geométrica");  
    }  
}
```

```
public interface Colorido {  
    public abstract void aplicarCor(String cor);  
}
```

CLASSE ABSTRATA

INTERFACE



SUBCLASSE

```
public class Circulo extends Forma implements Colorido {  
    private double raio;  
  
    public Circulo(double raio) {  
        this.raio = raio;  
    }  
  
    @Override  
    public double calcularArea() {  
        return Math.PI * raio * raio;  
    }  
  
    @Override  
    public void aplicarCor(String cor) {  
        System.out.println("Cor aplicada ao círculo: " + cor);  
    }  
}
```



TRATAMENTO DE EXCEÇÕES

❑ Por que tratar exceções?

- ❑ Para lidar com erros em tempo de execução sem quebrar o programa.
- ❑ Permite reagir a problemas de forma controlada.
- ❑ Evita mensagens de erro inesperadas para o usuário final.

❑ Blocos principais

- ❑ `try` → onde você coloca o código que pode gerar exceção.
- ❑ `catch` → onde você trata a exceção específica.
- ❑ `finally` → (opcional) executa sempre, mesmo com exceção



try / catch / finally

```
public class ExemploExcecao {  
    public static void main(String[] args) {  
        try {  
            int[] numeros = {10, 20, 30};  
            System.out.println(numeros[5]); // erro: índice inválido  
        } catch (ArrayIndexOutOfBoundsException e) {  
            System.out.println("⚠ Erro: índice fora do limite do array!");  
        } finally {  
            System.out.println("Bloco finally executado.");  
        }  
    }  
}
```



PROPAGANDO EXCEÇÕES

- ❑ **throws**

- ❑ Usado no **método** para dizer que ele pode lançar uma exceção.
 - ❑ Obriga quem chama o método a tratar (`try /catch`) ou também propagar



throws

```
public void lerArquivo(String caminho) throws IOException {
    BufferedReader br = new BufferedReader(new FileReader(caminho));
    System.out.println(br.readLine());
    br.close();
}
```

```
public static void main(String[] args) {
    try {
        new Exemplo().lerArquivo("dados.txt");
    } catch (IOException e) {
        System.out.println("Erro ao ler o arquivo: " + e.getMessage());
    }
}
```



EXCEÇÕES PERSONALIZADAS

- Podemos criar uma classe que estende `Exception` (checked) ou `RuntimeExpectation` (unchecked).
- Para situações específicas do nosso sistema.



INSTITUTO
FEDERAL
São Paulo

Campus
Caraguatatuba

```
public class SaldoInsuficienteException extends Exception {  
    public SaldoInsuficienteException(String mensagem) {  
        super(mensagem);  
    }  
}
```

```
public void sacar(double valor) throws SaldoInsuficienteException {  
    if (valor > saldo) {  
        throw new SaldoInsuficienteException("Saldo insuficiente para saque.");  
    }  
    saldo -= valor;  
}
```

```
try {  
    conta.sacar(1000.0);  
} catch (SaldoInsuficienteException e) {  
    System.out.println("Erro: " + e.getMessage());  
}
```



MANIPULAÇÃO DE ARQUIVOS

□ Por que trabalhar com arquivos?

- Salvar e recuperar dados persistentes.
- Ler configurações, registros ou resultados de programas.
- Interagir com dados externos sem depender apenas de banco de dados.

□ `java.io` (API clássica)

- `File` – representação do caminho do arquivo/diretório.
- `FileReader` / `FileWriter` – leitura/escrita de caracteres.
- `BufferedReader` / `BufferedWriter` – leitura/escrita com buffer (mais eficiente).



```
try (BufferedReader br = new BufferedReader(new FileReader("dados.txt"))) {  
    String linha;  
    while ((linha = br.readLine()) != null) {  
        System.out.println(linha);  
    }  
} catch (IOException e) {  
    System.out.println("Erro ao ler arquivo: " + e.getMessage());  
}
```

```
try (BufferedWriter bw = new BufferedWriter(new FileWriter("saída.txt"))) {  
    bw.write("Primeira linha");  
    bw.newLine();  
    bw.write("Segunda linha");  
} catch (IOException e) {  
    System.out.println("Erro ao escrever arquivo: " + e.getMessage());  
}
```



MANIPULAÇÃO DE ARQUIVOS

❑ Pacote `java.nio.file`:

- ❑ Substitui `java.io.File` com mais segurança e flexibilidade
- ❑ Principais classes e interfaces:

Classe	Descrição
Path	Representa um caminho de arquivo ou diretório (abstrato)
Paths	Fábrica de objetos Path
Files	Operações utilitárias (ler, escrever, copiar, mover, deletar)
StandardOpenOption	Controla como arquivos são abertos
DirectoryStream	Para iterar sobre diretórios



EXEMPLO

```
import java.nio.file.*;  
  
public class ExemploBasico {  
    public static void main(String[] args) throws Exception {  
        // Caminho relativo  
        Path caminho = Paths.get("dados.txt");  
  
        // Escrevendo conteúdo  
        Files.writeString(caminho, "Olá, mundo!");  
  
        // Lendo conteúdo  
        String conteudo = Files.readString(caminho);  
        System.out.println("Conteúdo do arquivo: " + conteudo);  
  
        // Verificando existência  
        if (Files.exists(caminho)) {  
            System.out.println("Arquivo encontrado!");  
        }  
  
        // obtendo tamanho do arquivo  
        System.out.println("Tamanho: " + Files.size(caminho) + " bytes");  
    }  
}
```



FORMAS DE LEITURA

- Linha por linha:

```
Path caminho = Paths.get("dados.txt");
List<String> linhas = Files.readAllLines(caminho);
for (String linha : linhas) {
    System.out.println(linha);
}
```

- Com Stream<String>:

```
Files.lines(Paths.get("dados.txt")).forEach(System.out::println);
```



FORMAS DE LEITURA

- Caractere por caractere:

```
string conteudo = Files.readString(Paths.get("dados.txt"));
for (char c : conteudo.toCharArray()) {
    System.out.println(c);
}
```

- Palavra por palavra:

```
List<String> linhas = Files.readAllLines(Paths.get("dados.txt"));
for (String linha : linhas) {
    String[] palavras = linha.split("\\s+");
    for (String palavra : palavras) {
        System.out.println(palavra);
    }
}
```



FORMAS DE ESCRITA

❑ BufferedWriter:

```
try (BufferedWriter writer = Files.newBufferedWriter(Paths.get("saída.txt"))) {  
    writer.write("Olá, mundo!");  
    writer.newLine();  
    writer.write("Outra linha.");  
}
```

❑ PrintWriter:

```
try (PrintWriter writer = new PrintWriter(Files.newBufferedWriter(Paths.get("saída.txt")))) {  
    writer.println("Escrevendo com PrintWriter");  
    writer.printf("Valor: %.2f%n", 12.34);  
}
```



INSTITUTO
FEDERAL
São Paulo

Campus
Caraguatatuba

□ Linhas:

```
import java.nio.file.*;
import java.io.IOException;
import java.util.List;

public class Escritasimples {
    public static void main(String[] args) throws IOException {
        Path caminho = Paths.get("saída.txt");
        List<String> linhas = List.of("Linha 1", "Linha 2", "Linha 3");
        Files.write(caminho, linhas); // escreve todas as linhas no arquivo
    }
}
```

FORMAS DE ESCRITA

□ Bytes:

```
byte[] dados = { 10, 20, 30, 40 };
Files.write(Paths.get("dados.bin"), dados);
```



OUTROS EXEMPLOS ÚTEIS

```
// Copiar um arquivo
Files.copy(Paths.get("origem.txt"), Paths.get("copia.txt"), StandardCopyOption.REPLACE_EXISTING);

// Mover um arquivo
Files.move(Paths.get("copia.txt"), Paths.get("arquivos/copia.txt"));

// Apagar
Files.deleteIfExists(Paths.get("dados.txt"));
```



ORGANIZAÇÃO DE PROJETOS EM JAVA

- ❑ Estrutura Básica de um Projeto Java:
 - ❑ src/ — pasta onde ficam os arquivos-fonte (.java).
 - ❑ bin/ ou build/ — pasta onde são gerados os arquivos compilados (.class).
 - ❑ lib/ — pasta para bibliotecas externas (.jar).
 - ❑ doc/ — documentação (opcional).
 - ❑ outros arquivos: README.md, MANIFEST.MF, arquivos de configuração.



PACOTES EM JAVA

- ❑ Usados para organizar classes de forma hierárquica.
- ❑ Evitam conflitos de nomes.
- ❑ Convenção de nomeação: geralmente br.com.empresaprojeto.

```
package br.com.meuprojeto.util;  
  
public class MinhaClasse {  
    // código  
}
```



ORGANIZAÇÃO NO NETBEANS IDE

- ❑ O NetBeans cria automaticamente a estrutura de pastas:
 - ❑ Source Packages — pasta src com pacotes.
 - ❑ Libraries — gerencia bibliotecas (.jar) adicionadas ao projeto.
 - ❑ Test Packages — para testes unitários, separados dos códigos fonte.

- ❑ Dicas práticas para organização:
 - ❑ Sempre utilize pacotes para separar funcionalidades (ex: model, view, controller).
 - ❑ Nomeie pacotes com convenção para evitar conflitos.
 - ❑ Separe arquivos fonte, testes e libs para facilitar manutenção.
 - ❑ Use o NetBeans para automatizar compilações, execução e gerenciamento de bibliotecas.



```
package br.com.exemplo.model;

public class Pessoa {
    private String nome;
    private int idade;

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

    public String getNome() {
        return nome;
    }

    public int getIdade() {
        return idade;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public void setIdade(int idade) {
        this.idade = idade;
    }

    public void imprimirInfo() {
        System.out.println("Nome: " + nome + ", Idade: " + idade);
    }
}
```

EXEMPLO

MeuProjeto/

- └ src/
 - └ br/com/exemplo/model/ (contém classe Pessoa)
 - └ br/com/exemplo/service/ (contém classe PessoaService)
 - └ br/com/exemplo/app/ (contém classe MainApp)



EXEMPLO

```
package br.com.exemplo.service;

import br.com.exemplo.model.Pessoa;
import java.util.ArrayList;
import java.util.List;

public class PessoaService {
    private List<Pessoa> pessoas = new ArrayList<>();

    public void adicionarPessoa(Pessoa p) {
        pessoas.add(p);
    }

    public void listarPessoas() {
        for (Pessoa p : pessoas) {
            p.imprimirInfo();
        }
    }
}
```

```
package br.com.exemplo.app;

import br.com.exemplo.model.Pessoa;
import br.com.exemplo.service.PessoaService;

public class MainApp {
    public static void main(String[] args) {
        PessoaService service = new PessoaService();

        Pessoa p1 = new Pessoa("Ana", 30);
        Pessoa p2 = new Pessoa("Bruno", 25);

        service.adicionarPessoa(p1);
        service.adicionarPessoa(p2);

        service.listarPessoas();
    }
}
```

EXERCÍCIO PRÁTICO

GERENCIADOR BÁSICO DE CONTA BANCÁRIA

1. Crie a classe abstrata `Conta` com:
 - atributos: `numero` (int), `titular` (String), `saldo` (double)
 - método abstrato `void sacar(double valor) throws SaldoInsuficienteException`
 - método `depositar(double valor)`
 - método concreto `imprimirDados()` para mostrar os dados da conta
2. Crie a exceção personalizada `SaldoInsuficienteException`.
3. Crie a subclasse `ContaCorrente` implementando o método `sacar` com a exceção.
4. No método `main`:
 - Leia os dados de uma conta do arquivo `conta.txt` (com 1 linha: número, titular e saldo separados por vírgula)
 - Crie o objeto `ContaCorrente` com esses dados
 - Solicite ao usuário um valor para saque
 - Tente realizar o saque, tratando exceção e exibindo mensagem adequada
 - Grave os dados atualizados no arquivo `conta_atualizada.txt`



**INSTITUTO
FEDERAL**

São Paulo

Campus
Caraguatatuba