



**SENAI**



# FUNDAMENTOS DE JAVA

## TÓPICOS ABORDADOS



- Manipulação de Arquivos
  - Conceito: Fluxo de leitura e escrita.
- Classes usadas em Java para manipulação de arquivos.
  - File.
  - FileReader e FileWriter.
  - BufferedReader e BufferedWriter.
- Conceitos de Exceções em Java.
  - try/catch/finally.
- Conceito de manipulação de Strings.
  - Recursos usados em Java.



# FUNDAMENTOS DE JAVA: **CONCEITO DE MANIPULAÇÃO DE ARQUIVOS**



# FUNDAMENTOS DE JAVA



## CONCEITO DE MANIPULAÇÃO DE ARQUIVOS

- Manipulação de arquivos é o processo de **ler, escrever, criar, editar** ou **excluir** arquivos de um sistema de armazenamento.
- Os arquivos estão localizados na **memória de armazenamento permanente**, como um disco rígido, SSD ou outro meio de armazenamento.
- Para realizar operações de leitura ou escrita, é necessário que esses arquivos sejam movidos temporariamente para a **memória RAM**, onde o processamento pode ocorrer.

O sistema reserva espaço na RAM para o arquivo e mantém uma conexão aberta entre o programa e o arquivo no armazenamento.

- Se esse fluxo permanecer aberto indefinidamente, pode causar problemas como o **consumo excessivo de memória** e o **bloqueio de arquivos**, impedindo que outros programas acessem esses dados.
- **Muitas requisições** de abertura desse fluxo também causam **lentidão** no seu programa e **consumo de recursos desnecessários**.



# FUNDAMENTOS DE JAVA

## FLUXOS DE LEITURA/ESCRITA: **SOLUÇÕES**



- É uma boa prática **fechar o fluxo** de dados assim que a operação for concluída, liberando os recursos do sistema
- Programas que **preparam os dados** antes das requisições diminuem a quantidade de requisições e otimizam o uso de recursos deixando o programa mais rápido.



# FUNDAMENTOS DE JAVA: **MANIPULAÇÃO DE ARQUIVOS: CLASSES**



Agora que entendemos o motivo de gerenciar arquivos com eficiência, vamos explorar as classes do Java usadas para manipular arquivos.

- Classes para manipulação de arquivos em Java:
  - **File**
  - **FileReader e FileWriter**
  - **BufferedReader e BufferedWriter**



- A classe File representa **um caminho para um arquivo ou diretório** no sistema de arquivos.
- Essa classe é o ponto de partida para qualquer operação relacionada a arquivos, pois permite que você **identifique um arquivo** (ou diretório) e **verifique** informações como a **existência do arquivo, permissões e tipo de arquivo** (se é um diretório ou arquivo comum).

# FUNDAMENTOS DE JAVA

## CLASSES **FILE**



- A classe File não realiza a leitura ou escrita de dados; ela apenas representa o arquivo.

**Exemplo:** Verificação da existência do arquivo.

```
File arquivo = new File("caminho/do/arquivo.txt");  
if (arquivo.exists()) {  
    System.out.println("O arquivo existe!");  
}
```

# FUNDAMENTOS DE JAVA



## FILE: EXEMPLO

Criar arquivo com base no caminho especificado.

```
// Caminho do arquivo que será criado
File arquivo = new File("exemplo.txt");

// Verifica se o arquivo já existe
if (arquivo.exists()) {
    System.out.println("O arquivo já existe.");
} else {
    // Cria o arquivo
    if (arquivo.createNewFile()) {
        System.out.println("Arquivo criado com sucesso: " + arquivo.getName());
    } else {
        System.out.println("Falha ao criar o arquivo.");
    }
}
}
```



# FUNDAMENTOS DE JAVA

## CLASSES **FILEREADER** E **FILEWRITER**



Essas classes são usadas para **ler** e **escrever** em arquivos de texto. Elas são chamadas de "***streams***" de leitura e escrita e lidam **diretamente** com a transferência de dados entre a memória de **armazenamento** e a **memória RAM**.

- **FileReader** é uma classe para **ler arquivos** de texto. Ele lê os dados do arquivo como uma **sequência de caracteres**.
- **FileWriter** é usado para **gravar caracteres** em arquivos.

# FUNDAMENTOS DE JAVA

## EXEMPLO: **FILEREADER** E **FILEWRITER**



Essas classes dependem de você passar diretamente um objeto **File** ou o caminho do arquivo como uma String.

```
// Escrita em um arquivo
FileWriter writer = new FileWriter("caminho/do/arquivo.txt");
writer.write("Texto a ser gravado no arquivo.");
writer.close(); // Sempre feche o stream após a escrita

// Leitura de um arquivo
FileReader reader = new FileReader("caminho/do/arquivo.txt");
int character;
while ((character = reader.read()) != -1) {
    System.out.print((char) character); // Lê cada caractere do arquivo
}
reader.close(); // Feche o stream após a leitura
```

# FUNDAMENTOS DE JAVA

## FILEWRITER: MODO APPEND



O construtor de FileWriter recebe dois parâmetros:

- O **caminho do arquivo** a ser escrito (caminhoArquivo).
- Um booleano true, que indica o **modo de append** (ou seja, adicionar conteúdo ao final do arquivo em vez de sobrescrevê-lo).
- Caso omita esse parâmetro ou coloque false, o arquivo será sobrescrito.

```
FileWriter fw = new FileWriter(caminhoArquivo, true);
```

Modo append



# FUNDAMENTOS DE JAVA

## FILEWRITER: EXEMPLO



Atualizando um arquivo usando a função **append** do `FileWriter`.

```
String caminhoArquivo = "meuArquivo.txt";
String conteudoNovo = "Este é o novo conteúdo que será adicionado ao arquivo.\n";

// Tenta abrir o arquivo em modo "append" para adicionar conteúdo sem apagar o existente
try (FileWriter fw = new FileWriter(caminhoArquivo, true)) {
    fw.write(conteudoNovo); // Adiciona o novo conteúdo ao final do arquivo
    System.out.println("Conteúdo adicionado com sucesso ao arquivo.");
} catch (IOException e) {
    System.out.println("Erro ao tentar atualizar o arquivo.");
    e.printStackTrace();
}
```

# FUNDAMENTOS DE JAVA

## CLASSES **BUFFEREDREADER** E **BUFFEREDWRITER**



- Enquanto **FileReader** e **FileWriter** manipulam dados diretamente, **BufferedReader** e **BufferedWriter** oferecem um **nível adicional de eficiência**.
- Essas classes utilizam um **buffer** (uma memória temporária) para otimizar as operações de leitura e escrita, permitindo que grandes blocos de dados sejam lidos ou escritos de **uma só vez**.
- Isso melhora a performance, especialmente ao trabalhar com **grandes volumes** de dados.

- **BufferedReader** usa um buffer para ler dados de maneira mais eficiente, permitindo a **leitura de linhas inteiras** de texto em vez de **ler caractere por caractere**.
- **BufferedWriter** permite gravar grandes blocos de dados de uma só vez, **acumulando os dados no buffer** antes de enviá-los para o arquivo, o que **reduz o número de acessos ao disco**.



# FUNDAMENTOS DE JAVA



## BUFFEREDREADER E BUFFEREDWRITER:

- Essas classes também dependem de **FileReader** e **FileWriter** como fluxo de base.
- Primeiro, você abre um **FileReader** ou **FileWriter** e o **passa para o construtor** de **BufferedReader** ou **BufferedWriter** para obter o benefício do buffer.
- Para adicionar informações a um arquivo com **BufferedWriter**, use um **FileWriter** com o modo append definido como **true**, permitindo escrever no final do arquivo sem sobrescrevê-lo.

```
BufferedWriter bufferWriter = new BufferedWriter(new FileWriter("caminho/do/arquivo.txt"));
```

# FUNDAMENTOS DE JAVA

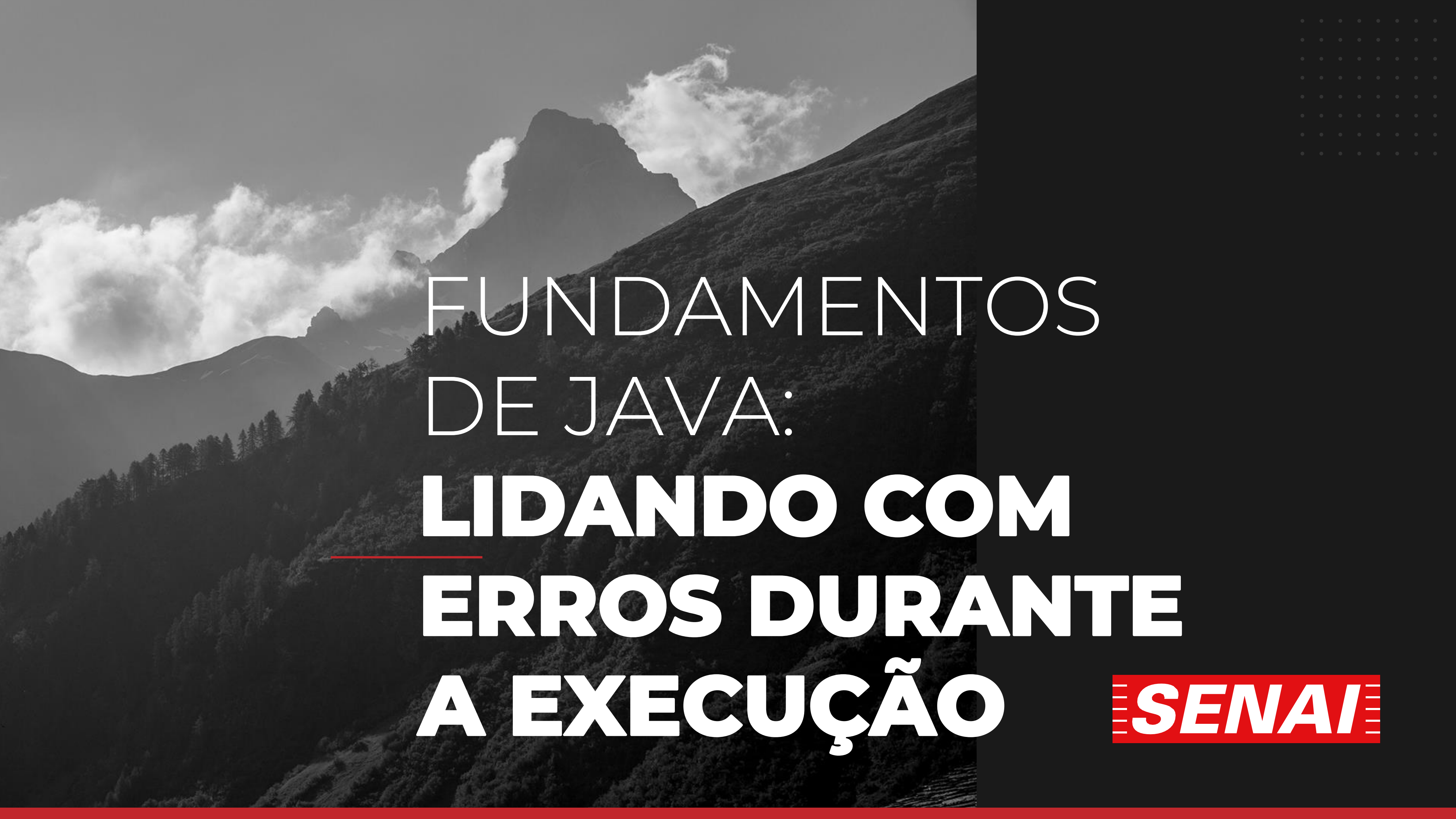
## BUFFEREDREADER E BUFFEREDWRITER:



### Exemplo de uso:

```
// Escrita otimizada usando BufferedWriter
BufferedWriter bufferWriter = new BufferedWriter(new FileWriter("caminho/do/arquivo.txt"));
bufferWriter.write("Texto a ser gravado no arquivo com buffer.");
bufferWriter.close();

// Leitura otimizada usando BufferedReader
BufferedReader bufferReader = new BufferedReader(new FileReader("caminho/do/arquivo.txt"));
String linha;
while ((linha = bufferReader.readLine()) != null) {
    System.out.println(linha); // Lê linha por linha do arquivo
}
bufferReader.close();
```



# FUNDAMENTOS DE JAVA: **LIDANDO COM ERROS DURANTE A EXECUÇÃO**





# FUNDAMENTOS DE JAVA



## O CONCEITO DE EXCEÇÕES EM JAVA:

- **Exceções** em Java são situações anormais ou **erros** que ocorrem **durante a execução** de um programa, interrompendo seu fluxo normal.
  - Quando um erro ocorre, Java lança uma "**exceção**" que **pode ser tratada** para evitar que o programa falhe de maneira inesperada.
  - Essas exceções são representadas por **objetos** que contêm informações sobre o erro, como o tipo de erro e onde ele ocorreu.

## TRATAMENTO DE EXCEÇÕES:

### Estrutura de um tratamento de exceções em Java:

- **try:** O código que pode gerar uma exceção é colocado dentro de um bloco try.
- **catch:** Se uma exceção for lançada no bloco try, ela pode ser capturada e tratada dentro de um bloco catch.
- **finally:** O bloco finally é opcional e contém código que será executado independentemente de uma exceção ter sido lançada ou não. É normalmente usado para liberar recursos, como fechar arquivos ou conexões de banco de dados.

# FUNDAMENTOS DE JAVA

TRY / CATCH / FINALLY:

## EXEMPLOS



### Exemplo Básico de Tratamento de Exceções em Java:

```
public class ExemploExcecoes {  
    public static void main(String[] args) {  
        try {  
            int resultado = 10 / 0; // Isso gera uma exceção do tipo ArithmeticException  
        } catch (ArithmeticException e) {  
            System.out.println("Erro: divisão por zero.");  
        } finally {  
            System.out.println("Este bloco sempre será executado.");  
        }  
    }  
}
```



# FUNDAMENTOS DE JAVA



## TRY-WITH-RESOURCES EM JAVA:

- O bloco **try-with-resources** é uma funcionalidade introduzida no **Java 7** que simplifica o gerenciamento de recursos, como fluxos de arquivos ou conexões de banco de dados.
- Esses recursos geralmente precisam **ser fechados** após o uso para **liberar memória** e outros **recursos do sistema**.
- Normalmente, isso seria feito no **bloco finally**, mas o **try-with-resources** garante que o recurso será **fechado automaticamente**, mesmo que ocorra uma **exceção** no código.

# FUNDAMENTOS DE JAVA

## TRY-WITH-RESOURCES EM JAVA:

### EXEMPLOS

- Você pode declarar **múltiplos recursos** em um bloco try, separados por ponto e vírgula.

### Exemplo usando um recurso

```
try (FileReader fr = new FileReader("arquivo.txt")) {  
    // código para ler o arquivo  
} catch (IOException e) {  
    e.printStackTrace();  
}
```

### Exemplo usando múltiplos recursos

```
try (FileReader fr = new FileReader("arquivo.txt");  
    BufferedReader br = new BufferedReader(fr)) {  
    String linha;  
    while ((linha = br.readLine()) != null) {  
        System.out.println(linha);  
    }  
} catch (IOException e) {  
    e.printStackTrace();  
}
```



# FUNDAMENTOS DE JAVA: **RECURSOS DE MANIPULAÇÃO DE STRING**



# FUNDAMENTOS DE JAVA

## MANIPULAÇÃO DE STRINGS



- Ao trabalhar com arquivos, o conteúdo geralmente é salvo como texto, seja em um formato simples como **.txt** ou algo mais estruturado como **.csv**.
- Esses textos precisam ser **convertidos em dados** que o programa possa interpretar e utilizar.

```
ID:1;Nome:João;Idade:25\n
```

```
ID:2;Nome:Maria;Idade:30\n
```



A manipulação de Strings é essencial para isso, pois permite:

**1. Extrair informações relevantes:**

- Um arquivo pode conter múltiplos dados (como **nome**, **e-mail**, **telefone**) que precisam ser **separados e processados** individualmente.

**2. Converter dados:**

- Muitas vezes, os dados no arquivo estão no formato de **String**, mas o programa precisa de outros **tipos de dados**, como números **inteiros** ou valores **booleanos**.

# FUNDAMENTOS DE JAVA

## MANIPULAÇÃO DE STRINGS



A manipulação de Strings é essencial para isso, pois permite:

### 3. Salvar dados estruturados:

- Ao salvar dados no arquivo, as **Strings** precisam ser manipuladas para serem organizadas em um **formato adequado** para armazenamento, como "chave" ou delimitadas por vírgulas.

- O programa pode usar **tokenização** com os delimitadores “;” e “:” para separar cada campo (**ID**, **Nome**, **Idade**) e seus respectivos valores. Assim, a linha será convertida em tokens como:

```
ID:1;Nome:João;Idade:25\nID:2;Nome:Maria;Idade:30\n
```

- **Tokenização** é o processo de dividir uma String em partes menores, chamadas de tokens, com base em **delimitadores específicos**, como **espaços**, **vírgulas**, ou outros caracteres definidos.
- No contexto de manipulação de arquivos, a tokenização é fundamental para **separar e interpretar** os **dados** armazenados em formato de texto.
- Isso permite transformar o **conteúdo bruto** de um arquivo em **informações estruturadas** que podem ser processadas pelo programa.

# FUNDAMENTOS DE JAVA

## TOKENIZAÇÃO EM JAVA



Em Java, o principal recurso usado para **tokenização** está disponível na classe **String**.

- **split():**

- O método `split()`, da classe `String`, permite que você defina um delimitador (ou expressão regular) para dividir a `String`.

### Exemplo:

```
String linha = "ID:1;Nome:João;Idade:25";
String[] tokens = linha.split(";");

for (String token : tokens) {
    System.out.println(token);
}
```

- O método **recebe** um delimitador (token) para dividir a `String` em várias partes.
- O **retorno** desse processo é um **array de Strings**, onde cada elemento representa uma parte da `String` original separada pelo delimitador fornecido.



# FUNDAMENTOS DE JAVA

## TOKENIZAÇÃO EM JAVA



- **trim():**

- O método trim() é uma função da classe String que remove os espaços em branco do início e do final de uma String.

### Exemplo:

```
String textoComEspacos = "  Olá, Mundo!  ";  
String textoSemEspacos = textoComEspacos.trim();  
System.out.println(textoSemEspacos); // Saída: "Olá, Mundo!"
```

- Esse método é extremamente útil para tratar dados extraídos de Strings.

### Métodos de Conversão de String para Valores Numéricos

- Quando precisamos trabalhar com números que estão armazenados como texto (Strings), podemos utilizar métodos de conversão da classe **Integer**, **Double**, **Float**, entre outras, para transformar essas Strings em seus tipos numéricos correspondentes.
- **Exemplo:** Conversão de String para int = **Integer.parseInt()**

```
String numeroTexto = "123";  
int numero = Integer.parseInt(numeroTexto);  
System.out.println(numero); // Saída: 123
```



# FUNDAMENTOS DE JAVA

## EXERCÍCIOS:

---

MANIPULAÇÃO DE **ARQUIVOS:**

- FILE
- FILEREADER E FILEWRITER
- BUFFEREDREADER E  
BUFFEREDWRITER
- TRY/CATCH





# MÃOS A **OBRA**

---

Use tudo o que você aprendeu para resolver os exercícios. Concentre-se no objetivo principal de cada um.

Se a sua solução funcionar como o exercício pede, que tal deixar tudo ainda mais legal? Use a sua criatividade para deixar o código mais intuitivo e fácil de entender. Organize as informações e melhore os textos que aparecem para o usuário.

**Lembre-se:** Faça com calma, peça ajuda quando precisar entender melhor alguma coisa. O importante é aprender e treinar a sua lógica de programação. Não vale copiar a resposta pronta! Esses exercícios são a sua chance de praticar e se tornar um programador ainda melhor. "





# MANIPULAÇÃO DE ARQUIVOS:

## EXERCÍCIOS



### **Exercício 1: Criar e Escrever em um Arquivo de Texto**

Implemente um programa que crie um arquivo chamado alunos.txt e permita que o usuário insira o nome de cinco alunos. O programa deve escrever esses nomes no arquivo, um em cada linha.

- Use a classe File para representar o arquivo.
- Utilize a classe FileWriter para escrever os dados.
- **Lembre-se de fechar o fluxo de escrita** após a operação e tratar os erros que podem ocorrer, como problemas ao criar ou abrir o arquivo.

**Dica:** Use **try/catch** para tratar exceções e garanta que o arquivo será fechado, mesmo que ocorra um erro.

# MANIPULAÇÃO DE ARQUIVOS:

## EXERCÍCIOS



### **Exercício 2: Leitura de Arquivo de Texto**

Desenvolva um programa que leia o conteúdo do arquivo alunos.txt (criado no exercício anterior) e exiba os nomes dos alunos no console.

- Utilize a classe FileReader para ler o arquivo.
- Utilize a classe BufferedReader para melhorar a eficiência da leitura.
- **Lembre-se de fechar o fluxo de leitura e tratar os possíveis erros** (por exemplo, caso o arquivo não exista).

**Dica:** Utilize um loop para ler todas as linhas do arquivo até que não haja mais conteúdo (ou seja, quando o método `readLine()` retornar `null`).

# MANIPULAÇÃO DE ARQUIVOS:

## EXERCÍCIOS

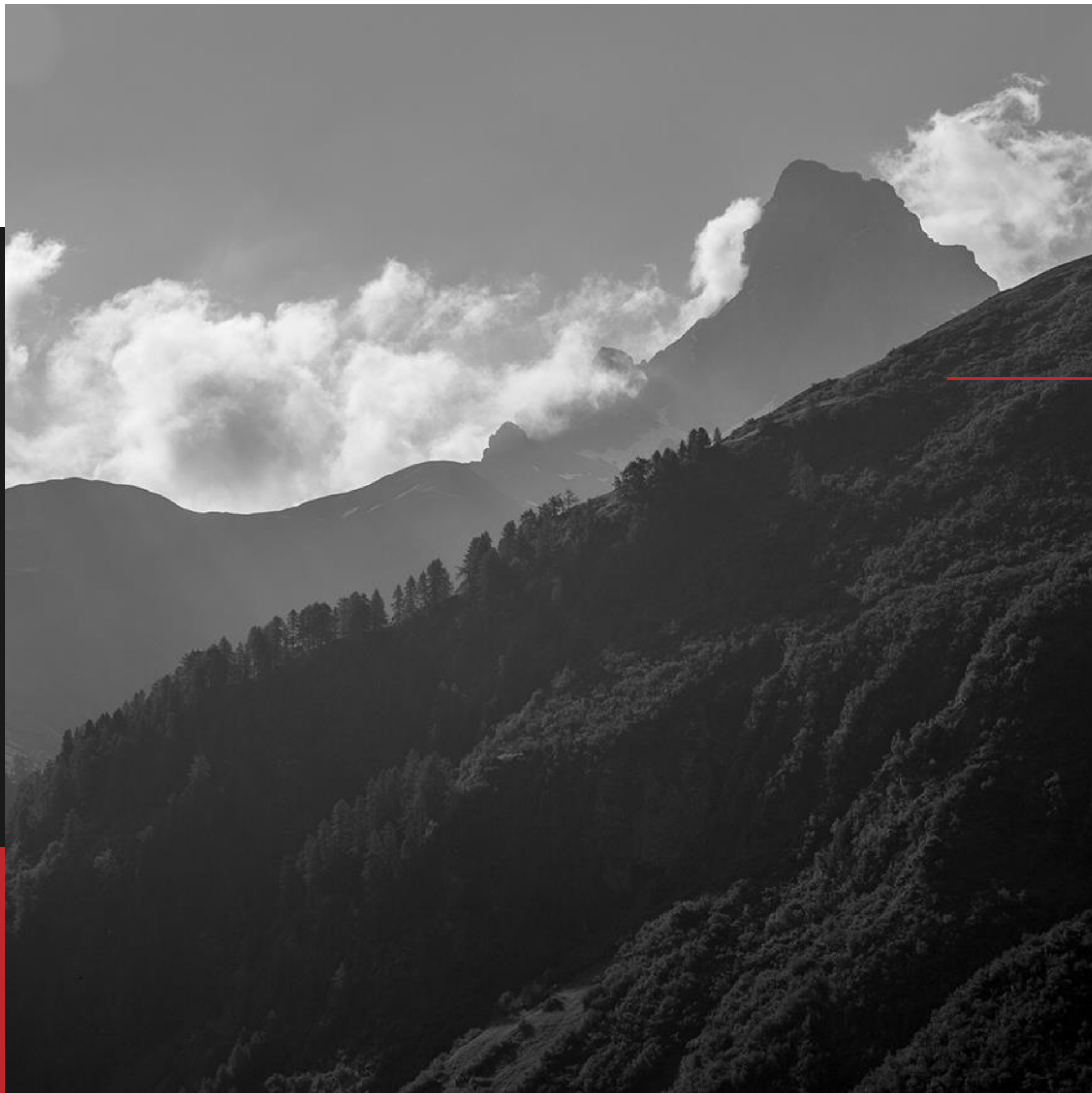
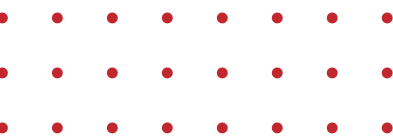


### **Exercício 3: Copiar Conteúdo de um Arquivo para Outro**

Crie um programa que copie o conteúdo do arquivo alunos.txt para um novo arquivo chamado backup\_alunos.txt.

- Utilize a classe FileReader para ler o arquivo de origem.
- Utilize a classe BufferedReader para otimizar a leitura.
- Utilize a classe FileWriter para escrever no arquivo de destino.
- Utilize a classe BufferedWriter para otimizar a escrita.
- Garanta que os fluxos de leitura e escrita sejam fechados corretamente e trate os erros que podem ocorrer durante o processo de leitura ou escrita.

**Dica:** Leia o arquivo linha por linha e escreva cada linha no novo arquivo.



# FUNDAMENTOS DE JAVA

## DESAFIO:

MANIPULAÇÃO DE **ARQUIVOS:**





# FUNDAMENTOS DE JAVA



## DESAFIO: MANIPULAÇÃO DE ARQUIVOS

### Sistema de Cadastro: Integração com Arquivos.

- No desafio anterior, vocês criaram um sistema de cadastro de pessoas utilizando uma matriz para armazenar dados como "ID", "Nome", "E-mail", e "Telefone".
- Agora, vamos evoluir o programa para usar um arquivo de texto como banco de dados, o que permitirá **salvar** e **carregar** os dados de maneira **persistente**.
- Isso significa que, mesmo que o programa seja fechado, os dados estarão disponíveis para serem carregados na próxima execução.

## DESAFIO: MANIPULAÇÃO DE ARQUIVOS

### Entendimento da Nova Função: **Carregar Dados do Arquivo**

- Agora, o programa deverá ler um arquivo de texto que contém os dados dos usuários e carregar essas informações na matriz que já foi usada no desafio anterior. O formato do arquivo seguirá o padrão:
  - Cada linha do arquivo conterá os dados de um usuário no formato:
    - cabeçalho = **"ID:Nome:E-mail:Telefone"**
    - 1º cadastro = **"01:Rafael:rafael@email.com:119000000000"**
  - Cada conjunto de informações estará separado por `\n` (quebra de linha), de modo que cada linha represente um cadastro.

## DESAFIO: MANIPULAÇÃO DE ARQUIVOS

### Como Ler e Preencher a Matriz:

- Quando o programa iniciar, ele deverá ler o arquivo bancoDeDados.txt e preencher a matriz que já está sendo usada no sistema com as informações contidas no arquivo.

### Vocês deverão:

1. **Ler** cada linha do arquivo.
2. **Separar** os dados com base no token ":" usando o método split().
3. **Preencher** a matriz com os dados separados.
4. **Certificar-se** de que a matriz tem o tamanho correto para armazenar todos os dados.

# FUNDAMENTOS DE JAVA



## DESAFIO: MANIPULAÇÃO DE ARQUIVOS

### Como Salvar as Alterações no Arquivo

- Depois que os dados forem carregados na matriz, o programa deverá permitir que o usuário faça alterações por meio do menu.
- Qualquer alteração feita (cadastro, atualização, exclusão) deverá ser refletida no arquivo para garantir que o banco de dados esteja sempre atualizado.

**Dica:** Sobrescreva o arquivo .txt a cada alteração feita na matriz.

Dessa forma, a matriz serve como referência, evitando múltiplas operações no arquivo para modificar item por item.





**SENAI**

DEPARTAMENTO REGIONAL  
**DE SÃO PAULO**

[www.sp.senai.br](http://www.sp.senai.br)