Curso de Java

aula 10

Prof. Anderson Henrique

TECH
soluções em ti

I/O – Input/Output (Manipulação de Arquivo)

Agora vamos aprender como criar, ler e escrever arquivos utilizando a biblioteca I/O, mas, o que são arquivos? Vídeos, textos, planilhas, imagens, músicas são exemplos de arquivos que você tem no seu computador. E, com certeza tem também alguns programas capazes de manipular esses arquivos.

Esse processo de manipulação de arquivos é feita utilizando a biblioteca I/O.

O seu programa Java também pode ler dados de uma série de fontes: do computador, da rede, de outros computadores e até mesmo da internet.

E consegue manipular essas informações e gravá-las em outro destino de dados., que pode ser no mesmo computador ou em outros computadores.

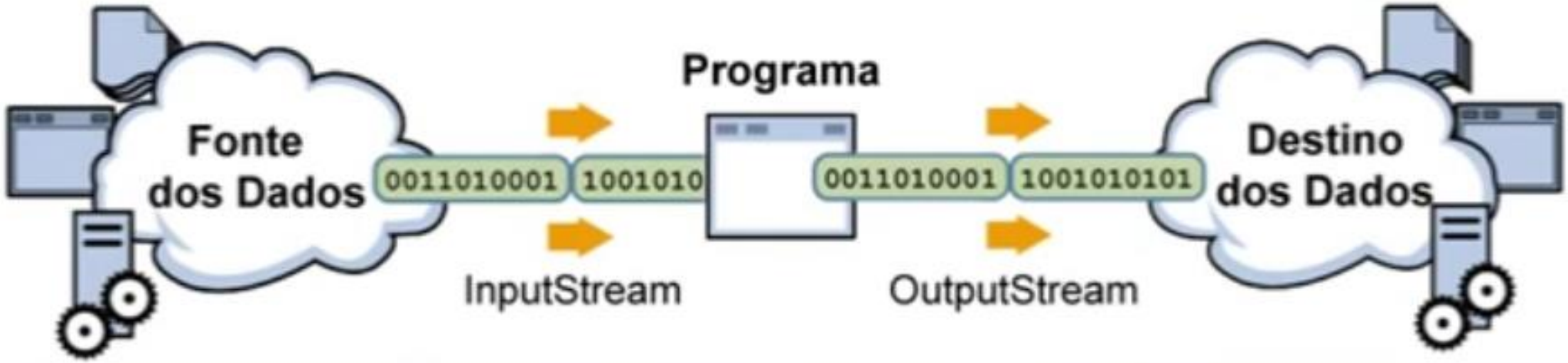
Esse processo de leitura de dados, é o que chamamos de input, porque está entrando com dados no seu programa. E o processo de escrever informações em um destino, é o que chamamos de output.

Nesta aula vamos manipular alguns arquivos no nosso computador. No nosso computador temos a raiz, representada pelo c:\ em um sistema operacional Windows.

I/O : Input / Output : Read / Write

Input : Read
Entrada de Dados no Programa
Programa Lê os Dados

Output : Write
Saída de Dados do Programa
Programa Escreve os Dados



E/S : Entrada / Saída : Ler / Escrever

Vamos criar um novo pacote: br.com.treinacom.java.arquivos, dentro de um novo projeto java chamado: aula 11.

Dentro desse pacote, vamos criar uma classe java principal chamada Arquivo.

Antes de criar arquivos, é preciso saber como tudo começa. Arquivos e diretórios tem um caminho (path), tem uma localização. E existe uma classe java que define esses caminhos (manipula esses caminhos), que é a classe Path. Então você pode criar um novo caminho, através de uma outra classe chamada Paths, que possui o método get( ).

O método get( ) ele pode receber uma string, que informará a localização do arquivo ou diretório que você deseja manipular.

Ex.:

Path caminho = Paths.get(“C:/files/texto.txt”);

Precisamos importar as classes que serão utilizadas, estas se encontram no pacote: java.nio.*. E a classe Path, foi introduzida no Java 7.

Temos diversos métodos na classe Path, vamos estuda-los:

toAbsolutePath( ) – retorna o caminho absoluto de um arquivo.

getParent( ) – mostra o arquivo pai de um arquivo, diretório.

getRoot( ) – mostra o diretório raiz de um arquivo.

getFileName( ) – mostra o nome do arquivo.

toUri( ) – mostra o identificador de recursos universal.

getFileSystem( ) – mostra qual sistema operacional está o arquivo.

Como podemos criar o diretório para armazenar esse arquivo? Utilizando outra classe Files, temos o método: createDirectories( ), podemos informar uma string com o caminho onde esse diretório deve ser criado.

Ex.:

```
Files.createDirectories(path.getParent( ));
```

Ao tentar executar esse método, ele irá lançar uma exceção, pode ser que não consiga criar o diretório.

Como podemos escrever e ler esse arquivo no nosso diretório? Utilizando o método write da classe Files, o método cria o arquivo (caso não exista, abre o arquivo, escreve no arquivo e fecha o arquivo. Método útil para manipular pequenos arquivos.

Ex.:

```
byte[ ] bytes = “Escrever o meu texto”.getBytes();

Files.write(path, bytes);
```

E, como podemos ler o conteúdo de um arquivo? Utilizando o método da classe Files, readAllBytes( ), recebe o caminho. O método retorna um conjunto de bytes.

Ex.:

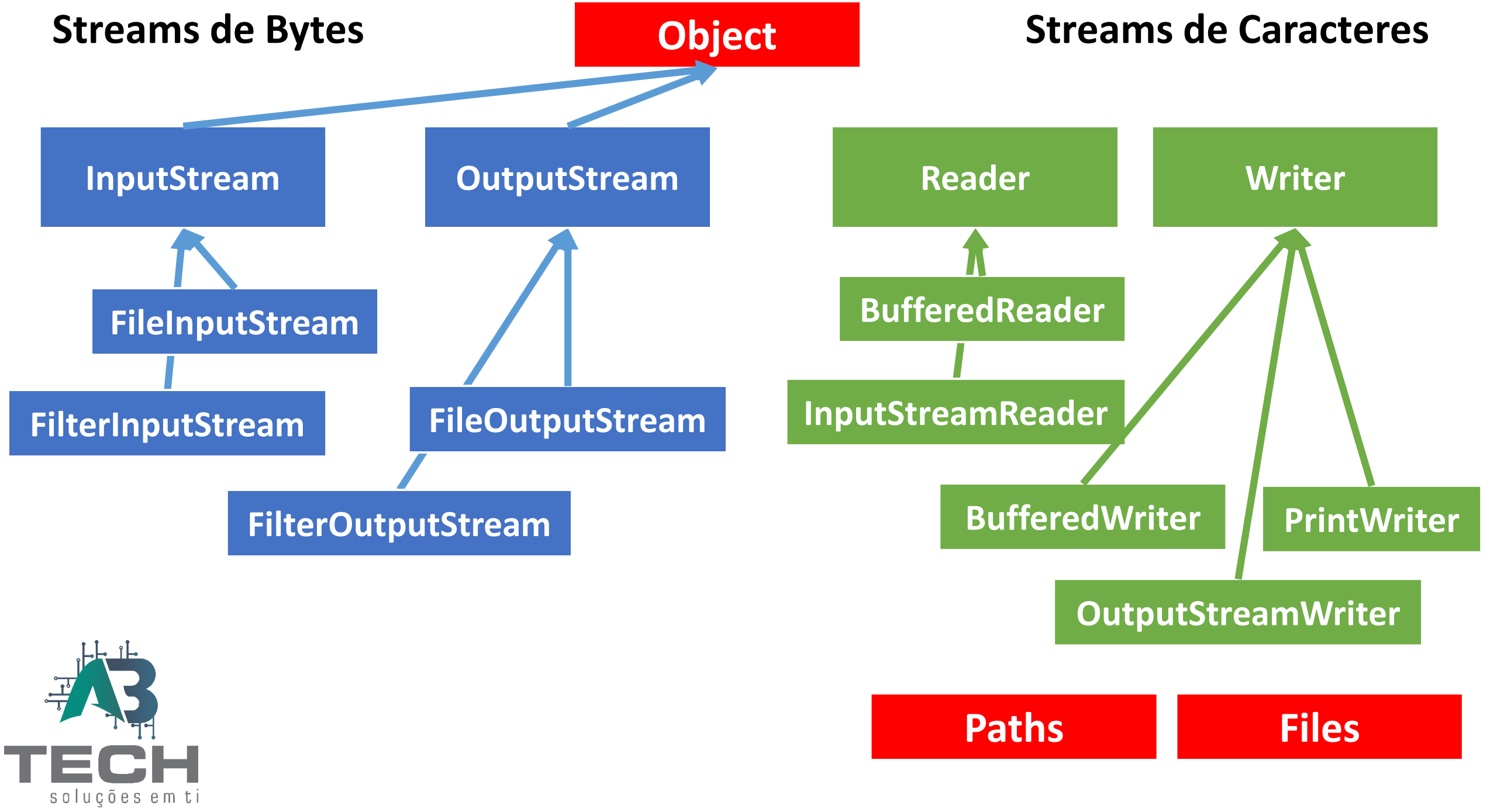
```
byte[ ] retorno = Files.readAllBytes(path);

System.out.println(new String(retorno));
```

I/O – Hierarquia buffer, try, Closable

Essa funcionalidade é importante porque permite ao seu programa java armazenar os dados com os quais está trabalhando nas unidades de armazenamento de longo prazo do seu computador, preservando essas informações mesmo depois que o computador for desligado.

O que é que o seu programa lê e escreve? Streams de dados, são sequências ordenadas de dados, que possuem uma origem ou um destino. E na biblioteca I/O temos vários tipos de streams. Vamos ver uma hierarquia simples dessa biblioteca:



TECH
soluções em ti

I/O – Leitura de Arquivo

Utilizando as classes FileInputStream, InputStream (byte), InputStream Reader (char), BufferedReader (string) com o seu método readLine( ).

Ex.:

```
public class Leitura{
public static void main(String[ ] args) throws IOException{
        InputStream is = new FileInputStream("C:/files/texto.txt");
        InputStreamReader isr = new InputStreamReader(is);
        BufferedReader br = new BufferedReader(isr);
```

```
String linha = br.readLine( );
        while(linha != null){
                System.out.println(linha);
                linha = br.readLine( );
        }
}
}
```

I/O – Leitura de Arquivo (teclado)

Ex.:

```
public static void main(String[ ] args) throws IOException{
        InputStream is = System.in;
        InputStreamReader isr = new InputStreamReader(is);
        BufferedReader br = new BufferedReader(isr);
        System.out.println("Digite o seu nome:");
        String entrada = br.readLine( );
        System.out.println(entrada);
}
```

I/O – Escrita em Arquivo

Utilizando as classes FileOutputStream, OutputStream, OutputStreamWriter, BufferedWriter com o seu método write( ).

Ex.:

```
public static void main(String[ ] args) throws IOException{
        OutputStream os = new FileOutputStream("C:/files/saída.txt");
        OutputStreamWriter osw = new OutputStreamWriter(os);
        BufferedWriter bw = new BufferedWriter(osw);

        bw.write("Escrevendo linha no arquivo");
        bw.close( );
}
```

I/O – Facilitando Leitura/Escrita no Arquivo

Utilizando as classes Scanner e PrintStream. Método hasNextLine( ) para verificar se existem mais linhas a serem lidas.

Ex.:

```
public static void main(String[ ] args) {
        Scanner leitor = new Scanner(System.in);
        PrintStream ps = new PrintStream("C:/files/novasaida.txt");
        while(leitor.hasNextLine( )){
                String entrada = leitor.nextLine( );
                if(entrada.equals("sair"){
                        break;
                }
```

```
            ps.println(entrada);
            }
            ps.close( );
}
```

Praticando

Crie uma classe java principal chamada Arquivo2, dentro do mesmo pacote.

Crie um caminho, utilizando a classe Path

Ex.:

Path path = Paths.get(“C:/files/texto.txt”);

Vamos escrever no nosso arquivo utilizando o método BufferedWriter( )

Charset utf8 = StandardCharsets.UTF_8;

BufferedWriter w = Files.newBufferedWriter(path, utf8);

Esse método lançará exceções, retorna objeto do tipo BufferedWriter.

Através do objeto, podemos utilizar o método write, diversas vezes.

Ex.:

w.write(“Texto”);

w.write(“Texto”);

w.flush( );

w.close( );

Para gravar as informações transmitidas pelo método write( ), precisamos utilizar o método flush( ). A palavra buffer significa um repositório de dados., um depósito de dados na memória do computador, o método flush pega todos os dados e grava no disco rígido do computador, e o método close( ), fecha a ligação com o arquivo.

Vimos que os métodos da classe BufferedWriter lança exceções, neste ponto se torna interessante tratarmos as exceções com try/catch.

Ex.:

```
try{
        BufferedWriter w = Files.newBufferedWriter(path, utf8);
        w.write(“Texto”);
        w.write(“Texto”);
        w.flush( );
        w.close( );
}catch(IOException e){
        e.printStackTrace( );
}
```

Imagine que mesmo tratando as exceções, ao chamar o método write( ) ocorra algum erro, o programa deverá encerrar essa ligação mesmo assim.

Ex.:

```
BufferedWriter w = null;
try{
        w = Files.newBufferedWriter(path, utf8);
        w.write(“Texto”);
        w.write(“Texto”);
        w.flush( );
}catch(IOException e){
        e.printStackTrace( );
}finally{
        if(w != null){
                w.close( );
        }
}
```

Essa forma de trabalhar com try/catch é uma prática dos antigos desenvolvedores java, a partir da versão do Java 7, podemos inicializar nossa variável dentro do bloco try( )

Ex.:

```
try(BufferedWriter w = Files.newBufferedWriter(path, ut8)){
}
```

E não precisamos mais finalizar a nossa ligação, podemos remover o nosso bloco finally, por quê? A partir da versão do Java 7, todos os objetos que implementam a interface Closable, são automaticamente fechados, o compilador cria um bloco finally. O método close( ) automaticamente chama o método flush( ), também podemos removê-lo.

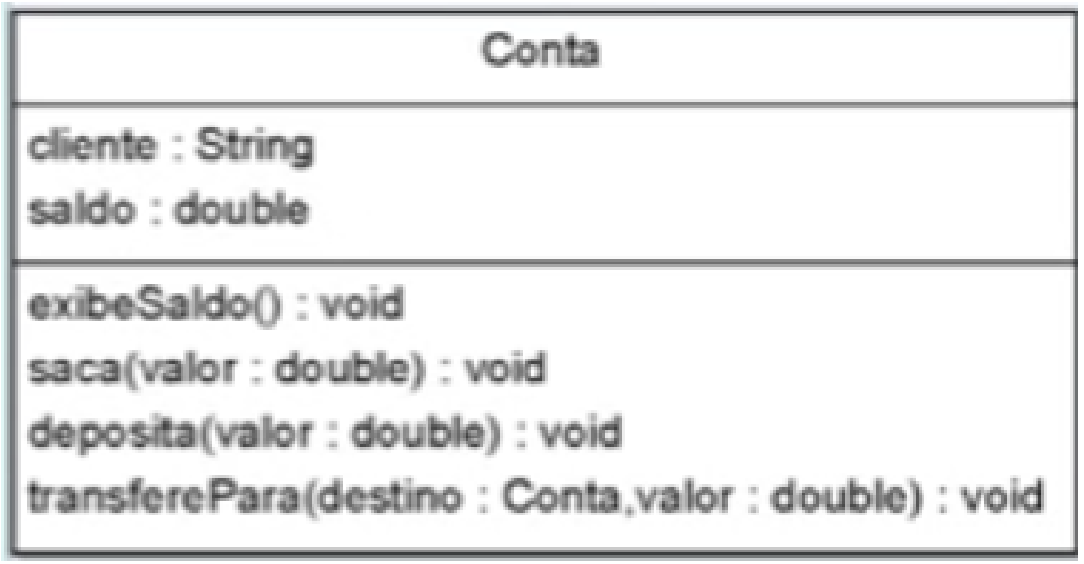
Como podemos trabalhar com a leitura desse arquivo, utilizando a classe BufferedReader?

Ex.:

```
try(BufferedReader r = Files.newBufferedReader(path, utf8)){
        String line = null
        while((line = r.readLine( )) != null){
                System.out.println(line);
        }
}catch(IOException e){
        e.printStackTrace( );
}
```

Desafio I/O

Criar um programa com 2 métodos:



(Dica: utilize um ArrayList<>( ) para armazenar os objetos do tipo Conta.

- Um que grava uma lista de contas, em arquivo sequencial;
- Outro que lê esse arquivo, carregando os objetos do tipo conta novamente;

TECH
soluções em ti

Ex.:

```
public class DesafioContas{
        Path path = Paths.get("C:/files/contas.txt");
        Charset utf8 = StandardCharsets.UTF_8;
        public void armazenarContas(ArrayList<Conta> contas) throws
IOException{
        try(BufferedWriter w = Files.newBufferedWriter(Path, utf8)){
                for(Conta conta : contas){
                        w.write(conta.getCliente( )+"; "+conta.getSaldo( ) +
"  \n");
                }
        }
}
```

Ex.:

```
public ArrayList<Conta> recuperarContas( ) throws IOException{
        ArrayList<Conta> contas = new ArrayList<>( );
        try(BufferedReader r = Files.newBufferedReader(path, utf8)){
                String line = null;
                while((line = r.readLine( )) != null){
                        String[ ] t = line.Split(“;”);
                        Conta conta = new Conta(t[0],
Double.parseDouble(t[1]));
                        contas.add(conta);
                }
        }
        return contas;
}
```

Ex.:

```
public static void main(String[] args) throws IOException{

        ArrayList<Conta> contas = new ArrayList<>( );
        contas.add(new Conta("Jéssica", 12000.23));
         contas.add(new Conta("Paulo", 11050.32));
         contas.add(new Conta("Amanda", 18000.21));
         contas.add(new Conta("Beatriz", 23200.09));

        DesafioContas operacao = new DesafioContas( );
        operacao.armazenarContas(contas);
        ArrayList<Conta> contas2 = operacao.retornarContas( );
        for(Conta conta : contas2){
                conta.exibeSaldo( );
        }
```

I/O – Check, Delete, Create, Copy e Move

Vamos explorar a biblioteca I/O um pouco mais. Utilizaremos os métodos Check (verifica atributos de arquivos e diretórios), Delete (excluir arquivos e diretórios), Create, Copy, Move (criar, copiar ou mover arquivos de um diretório para outro).

Crie uma nova classe java principal, chamada: Arquivo4, no mesmo pacote.

Crie um caminho, utilizando a classe Path e Paths, seguindo o modelo de acordo com o exemplo:

```
public class Arquivo4{
public static void main(String[ ] args){

        Path path = Paths.get("C:/files/saida.txt");
        Files.exists(path); //verifica se o caminho existe
        Files.isDirectory(path); //verifica se é um diretório
        Files.isRegularFile(path); //verifica se é um arquivo regular
        Files.isReadable(path); //verifica se pode ser lido pelo usuário
        Files.isExecutable(path); //verifica se é um arquivo executável
        Files.size(path); //lê o tamanho de um arquivo
        Files.getLastModifiedTime (path); //verifica a última vez que o
arquivo foi modificado
```

Files.**getOwner**(path); //retorna o dono do arquivo

Files.**probeContentType**(path); //retorna o tipo do arquivo

A classe Files possui dois métodos para exclusão: **delete**(path) pode lançar exceção e o **deleteIfExists**(path), não lança exceção.

Podemos criar arquivos com o método **createFile**(path), cria um arquivo sem conteúdo.

Podemos escrever conteúdo com o método **write**(path, “Meu texto”.getBytes( ));

Podemos criar uma cópia de um arquivo, na verdade é um novo destino (caminho) para o arquivo, Path copia = Paths.get(“C:/files/copia.txt”);

Files.**copy**(path, copia, StandardCopyOption.REPLACE_EXISTING);

Podemos mover um arquivo de um diretório para outro, também utilizamos outro caminho criando um novo diretório.

```
Path mover = Paths.get("C:/files/move/copia.text");
Files.createDirectories(mover.getParent( ));
Files.move(path, mover, StandardCopyOptions.REPLACE_EXISTING);
```

Arquivos I/O – Listar dir, Listar conteúdo, Listar partições

Para listar diretórios, conteúdos de diretórios e partições que existem no nosso sistema operacional, utilizamos a classe **FileSystems** que possui métodos com essas finalidades.

Métodos:

getDefault( ) – retorna o FileSystem padrão, ou seja, de cada sistema operacional específico.

getRootDirectories( ) – recupera os diretórios raízes do nosso sistema

Ex.:

```
Iterable<Path> dirs = FileSystems.getDefault( ).getRootDirectories( );
for(Path path : dirs){
        System.out.println(path);
}
```

Como poderia listar o conteúdo de cada diretório? Utilizando o objeto do tipo DirectoryStream( );

Ex.:

```
Path dir = Paths.get("C:/Arquivos de Programas");
try(DirectoryStream<Path> stream = Files.newDirectoryStream(dir)){
        for(Path path : stream){
                System.out.println(path.getFileName( ));
        }
}catch(IOException | DirectoryIteratorException e){
        e.getMessage( );
}
```

Como podemos listar o conteúdo de partições do nosso computador? Novamente utilizando o nosso **FileSystems**. Utilizamos o método **getFileStores**( ), retorna um **Iterable**<FileStore>. Objetos do tipo Iterable podem ser percorridos com o laço for.

Ex.:

```
FileSystem fs = FileSystems.getDefault( );
for(FileStore store : fs.getFileStores( )){
        System.out.println(store.toString); //exibe as unidades
        System.out.println(store.getTotalSpace); //exibe total em bytes
        System.out.println(store.getUsableSpace); //exibe total disponível
        System.out.println(store.getTotalSpace( ) - store.getUsableSpace( )); //exibe total utilizado
        System.out.println( );
}
```

Prosseguiremos no próximo slide...

Professor: Anderson Henrique

Programador nas Linguagens Java e PHP