Transcrição

Caso você queira baixar o zip com o arquivo **lorem.txt**, clique aqui (https://s3.amazonaws.com/caelum-online-public/857-java-io/01/lorem.zip).

Nesta aula, daremos início ao nosso curso focado no pacote `java.io`.

Atualmente, nenhuma aplicação funciona isoladamente e não receba ou envie dados. Casos em que isto não aconteça são exceções raras.

Temos, em geral, um fluxo de dados de entrada e outro de saída. Por exemplo, aqueles que assistem aos vídeos a partir do navegador, conseguem fazê-lo porque eles vêm de um servidor, ou seja, um fluxo (ou *streaming*) de informações.

O mesmo é verdade para o mobile. Ainda que seja feito o download prévio, o aplicativo da Alura lê o arquivo no HD para que o usuário possa assisti-lo.

Ou seja, podemos concluir que sempre há uma entrada, e esta sempre pode variar. Pode ser um arquivo, a rede, ou ainda um teclado. Estes são os tipos de **entrada concreta**.

Para a aplicação, isto não tem grande relevância, já que de qualquer forma todos representam uma entrada.

O mesmo é válido para a saída, a aplicação mobile retornará dados para a Alura, por exemplo, no momento em que um usuário conclui um vídeo ou exercício. O fluxo de saída concreto é variável, pode ser que o usuário decida gravar um arquivo ou que haja um retorno por meio da rede, como é o caso da Alura, ou ainda, podemos ter um retorno no console que é aquele que vemos ao executar um programa no Eclipse, por exemplo.

Ainda que o tipo de fluxo varie, para aplicação, é importante que **haja uma saída**. Isto é válido para qualquer aplicação, ou pelo menos para a vasta maioria delas.

Passaremos a trabalhar com as classes do `java.io` para modelarmos a estrutura de entrada e saída que foi mencionada acima. Focaremos, primeiro, no fluxo de entrada, em particular, no **arquivo**. Estabeleceremos uma entrada a partir de um arquivo.

Abriremos o Eclipse. Estamos utilizando o Oxygen na Versão 2.

Os projetos que são exibidos no menu lateral esquerdo estão disponíveis para download, mas se preferir, este pode ser feito posteriormente já que trabalharemos ainda com o `bytebank-herdado-conta`.

Neste momento, para darmos continuidade, criaremos um novo projeto. Para isso, clicaremos com o botão direito sobre a barra lateral esquerda, onde temos o menu de exploração de arquivos, e selecionaremos a opção "New > Java Project".

Será um projeto Java padrão, onde utilizaremos a Java SE 10.0.0 - os recursos com os quais trabalhamos funcionam com versões anteriores do Java, portanto, não precisamos nos preocupar com esta questão. O nome do nosso projeto será `java-io`, e nele faremos os nossos testes com a entrada e saída.

Em seguida, podemos partir para a criação da nossa primeira classe.

Como queremos trabalhar com a entrada a partir de um arquivo, primeiro, temos que ter este arquivo. Como exemplo, utilizaremos um arquivo em formato `.txt`, no qual há um texto de exemplo em *Lorem Ipsum*, ele está disponível para download mas pode ser substituído por qualquer outro do mesmo formato, desde que tenha um conteúdo.

O arquivo deve ser inserido na raiz do projeto, ou seja, na própria pasta `java.io`, e não na pasta `src`.

Criaremos uma classe, clicando com o botão direito do mouse sobre a pasta `src`, selecionaremos a opção "New > Class". Ela será inserida no pacote `br.com.alura.java.io.teste` e terá o nome `TesteLeitura`, já com o método `main`:

```
package br.com.alura.java.io.teste;

public class TesteLeitura {

	public static void main(String[] args) {
		//TODO Auto-generated method stub
	}
}
```

COPIAR CÓDIGO

Apagaremos a linha de código gerada automaticamente pelo Java.

Nosso objetivo é estabelecer um fluxo de entrada com um arquivo. Já que em Java trabalhamos com a língua inglesa, precisamos traduzir certos termos, arquivo por exemplo, é "file", entrada é "input", e fluxo é "stream", resultado em um `FileInputStream` :

```
package br.com.alura.java.io.teste;

public class TesteLeitura {

	public static void main(String[] args) {

		//Fluxo de Entrada com Arquivo

		FileInputStream
	}
}
```

COPIAR CÓDIGO

Nossa variável se chamará `fis` , e criaremos um objeto do tipo `FileInputStream()` . Neste ponto, ele reconhecerá o pacote `java.io` . Ao confirmarmos, será a importação ocorrerá automaticamente:

```
package br.com.alura.java.io.teste;

import java.io.FileInputStream;

public class TesteLeitura {
```

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

                //Fluxo de Entrada com Arquivo

                FileInputStream fis = new FileInputStream(file)
        }
}
```

COPIAR CÓDIGO

A seguir, veremos os diferentes tipos de construtores. O primeiro que aparece na lista de sugestões apresentada pelo Eclipse, recebe um arquivo `File`, entretanto, não é esse que utilizaremos, uma vez que nos é disponibilizado algo ainda mais simples, que é o construtor `String name`, representando o nome do arquivo com o qual desejamos trabalhar.

Para o utilizarmos, basta escrevermos o nome do arquivo como uma `String`, no caso, nosso arquivo é o `lorem.txt`:

```
//Código omitido

public class TesteLeitura {

        public static void main(String[] args) {

                //Fluxo de Entrada com Arquivo

                FileInputStream fis = new FileInputStream("lorem.txt");
```

```
        }
}
```

COPIAR CÓDIGO

Contudo, o código ainda não compila. O Eclipse nos informa que, para que isso aconteça, precisamos fazer ainda um tratamento de exceção.

Como sabemos, há dois tipos de exceção, *checked* e *unchecked*, o `java.io` está repleto de exceções *checked* .

O Java não é capaz de garantir que o desenvolvedor realmente inseriu o arquivo na raiz do projeto, por isso, o código está passível de falhas. Precisamos alertar sobre esta falha, e o modo pelo qual fazemos isso é a exceção do tipo *checked*.

Neste caso, criaremos um `throws` de `FileNotFoundException` :

```
//Código omitido

public class TesteLeitura {

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

                //Fluxo de Entrada com Arquivo
                FileInputStream fis = new FileInputStream("lorem.txt");
        }
}
```

COPIAR CÓDIGO

Utilizando a variável `fis`, podemos utilizar uma série de métodos, dentre eles, está o `read()`. O seu retorno é do tipo `int`, ou seja, um número. Isso indica que ele é capaz de ler os bytes, o que não é interessante para nós, não queremos as informações de bytes e binários, mas sim os caracteres.

Entretanto, nos parece que o `FileInputStream` não é capaz de realizar isto que desejamos. Para isso, teremos de utilizar uma outra classe.

Há uma classe capaz de transformar um `int` em caracteres, que se chama `InputStreamReader`.

A ideia é que ela é capaz de ler um `FileInputStream`.

Criaremos uma variável `isr`, com um objeto do tipo `InputStramReader()`, que receberá em seu construtor um `fis`:

```
//Código omitido

public class TesteLeitura {

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

                //Fluxo de Entrada com Arquivo
                FileInputStream fis = new FileInputStream("lorem.txt");
                InputStreamReader isr = new InputStreamReader(fis);
        }
}
```

COPIAR CÓDIGO

A variável `isr` nos permite utilizar uma outra variedade de métodos, além do método `read()` citado acima, há um segundo, que recebe como parâmetro um array de caracteres. Ele também nos retorna um `int`, entretanto, neste caso ele corresponde ao número de caracteres que foram lidos.

Neste caso, conseguimos transformar bits e bytes em caracteres, mas ainda não é a melhor solução para nosso problema.

A ideia é que sejamos capazes de ler as linhas inteiras do arquivo de texto, para isso, temos que "guardar" cada um dos caracteres, até sermos capazes de completar uma linha, e assim por diante.

Para esta tarefa, há o que chamamos de `BufferedReader`. Criaremos um em nosso código:

```
//Código omitido

public class TesteLeitura {

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

                //Fluxo de Entrada com Arquivo
                FileInputStream fis = new FileInputStream("lorem.txt");
                InputStreamReader isr = new InputStreamReader(fis);
                BufferedReader br =  new BufferedReader(in);
        }
}
```

COPIAR CÓDIGO

Como parâmetro, ele receber um outro `reader`, no caso, nosso `InputStreamReader` se qualifica como tal, por isso, como passaremos o `isr`:

```
//Código omitido

public class TesteLeitura {

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

                //Fluxo de Entrada com Arquivo
                FileInputStream fis = new FileInputStream("lorem.txt");
                InputStreamReader isr = new InputStreamReader(fis);
                BufferedReader br = new BufferedReader(isr);
        }
}
```

COPIAR CÓDIGO

Primeiro, criamos o fluxo concreto com o arquivo, mas ainda binário, em seguida, conseguimos transforma-los em caracteres, mas apenas a contabilização, por fim, com o `BufferedReader`, podemos utilizar o método `readLine()`, que nos permite ler linha a linha.

Este método nos retorna uma `String`, que representa a `linha`:

```
//Código omitido

public class TesteLeitura {
```

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

                //Fluxo de Entrada com Arquivo
                FileInputStream fis = new FileInputStream("lorem.txt");
                InputStreamReader isr = new InputStreamReader(fis);
                BufferedReader br = new BufferedReader(isr);

                String linha = br.readLine();
        }
}
```

COPIAR CÓDIGO

O Eclipse sinaliza que o programa ainda não está funcionando, para isso, teremos que fazer um novo tratamento, ou `IOException`.

Ao trabalharmos com `java.io` é necessário dominarmos dois tipos principais de exceção, a primeira é a `FileNotFoundException`, que já vimos, e a segunda é a `IOException`.

Com a tecla "Ctrl" pressionada, clicaremos sobre `FileNotFoundException` e abriremos esta classe. Veremos que ela estende a `IOException`:

```
//Código omitido

public class FileNotFoundException extends IOException {

//Código omitido
```

COPIAR CÓDIGO

Portanto, a `FileNotFoundException` é uma `IOException`, esta por sua vez, é uma exceção, já que estende `Exception`. Por isso, em vez de utilizarmos a exceção mais específica, utilizaremos o tipo mais genérico:

```
//Código omitido

public class TesteLeitura {

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

                //Fluxo de Entrada com Arquivo
                FileInputStream fis = new FileInputStream("lorem.txt");
                InputStreamReader isr = new InputStreamReader(fis);
                BufferedReader br =  new BufferedReader(isr);

                String linha = br.readLine();
        }
}
```

COPIAR CÓDIGO

Lembrando que, como estamos utilizando uma nova classe, precisamos importá-la.

Já sabemos ler uma linha, mas precisamos ler as demais. Por enquanto, imprimiremos apenas esta primeira linha, para criarmos uma saída, representada no caso por `out`:

```
//Código omitido

public class TesteLeitura {

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

                //Fluxo de Entrada com Arquivo
                FileInputStream fis = new FileInputStream("lorem.txt");
                InputStreamReader isr = new InputStreamReader(fis);
                BufferedReader br = new
BufferedReader(isr);

                String linha = br.readLine();

                System.out.println(linha);
        }
}
```

COPIAR CÓDIGO

O compilador indica que estabelecemos uma entrada em `BufferedReader`, mas não uma saída, assim, fecharemos com o `br.close()`:

```
//Código omitido

public class TesteLeitura {

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

```
                //Fluxo de Entrada com Arquivo
                FileInputStream fis = new FileInputStream("lorem.txt");
                InputStreamReader isr = new InputStreamReader(fis);
                BufferedReader br = new BufferedReader(isr);

                String linha = br.readLine();

                System.out.println(linha);

                br.close();
        }
}
```

COPIAR CÓDIGO

Isso faz com que tanto o `FileInputStream` quanto o `InputStreamReader` sejam fechados automaticamente. Por isso não há necessidade de os fecharmos individualmente.

Salvaremos todo o código. Executaremos, e temos o seguinte resultado no console:

```
Lorem ipsum dolor sit amet, consectetur elit, sed do eiusmod
```

COPIAR CÓDIGO

Funcionou, imprimimos a primeira linha do texto do nosso arquivo.

Adiante, veremos como podemos melhorar este código. Até lá!

Transcrição

Anteriormente, conseguimos estabelecer uma entrada e escrevemos código capaz de ler a primeira linha de nosso arquivo `lorem.txt` .

Nas seguintes linhas de código:

```
public class TesteLeitura {

//Código omitido

//Fluxo de Entrada com Arquivo
FileInputStream fis = new FileImputStream("lorem.txt");
InputStreamReader isr = new InputStreamReader(fis);
BufferedReader br = new BufferedReader(isr);

}
```

COPIAR CÓDIGO

Foi estabelecida a entrada com o arquivo e, além disso, melhoramos a leitura, já que nosso objetivo era traduzir a linha inteira. Para isso, foi necessário utilizarmos as classes `InputStreamReader` e `BufferedReader`. A primeira transforma bytes em caracteres, enquanto a segunda é responsável por unir os caracteres em uma linha e interpretá-los, linha a linha.

Temos a referência `fis`, que aponta para o objeto `FileInputStream("lorem.txt")`, e foi inserida como parâmetro no construtor `InputStreamReader(fis)`.

O mesmo foi feito com a referência `isr`, que aponta para o objeto `InputStreamReader(fis)`, e foi inserida como parâmetro no construtor `BufferedReader(isr)`.

Na prática, isso significa que o `FileInputStream` é administrado por meio do `InputStreamReader`, este por sua vez, é administrado pelo `BufferedReader`, pois é passado no construtor.

Ao utilizarmos o método `br.readLine()`, pedimos primeiro ao `BufferedLine`, ele por sua vez faz o pedido ao `InputStreamReader` que, seguindo a ordem, pede ao `FileStreamReader` que faça a leitura dos dados do arquivo, que no caso é `lorem.txt`. Visualmente, temos algo como o desenho a seguir:

```
BufferedReader > InputStreamReader > FileInputStream > lorem.txt
```

Isso que fizemos é um padrão de projeto chamado *decorator*, ou seja, um objeto está **decorando** a funcionalidade de outro, sucessivamente. Em geral, o `java.io` é repleto de padrões de projeto.

Nosso objetivo seguinte será ler linha a linha do arquivo, até sua totalidade.

O método `readLine()` nos dá um retorno `null` quando não há mais nenhum conteúdo, portanto, criaremos um `while`, indicando que, enquanto a linha não for nula (`null`), teremos a impressão desta e leremos a próxima:

```
//Código omitido

public class TesteLeitura {

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

                //Fluxo de Entrada com Arquivo
                FileInputStream fis = new FileInputStream("lorem.txt");
                InputStreamReader isr = new InputStreamReader(fis);
                BufferedReader br = BufferedReader(isr);

                String linha = br.readLine();

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

                System.out.println(linha);

                br.close();
        }
}
```

COPIAR CÓDIGO

Salvaremos e executaremos o código. Temos o seguinte resultado no console:

```
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod
tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam,
quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo
consequat. Duis aute irure dolor in reprehenderit in voluptate velit
esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat
cupidatat non proident, sunt in culpa qui officia deserunt mollit anim
id est laborum.
```

COPIAR CÓDIGO

Temos o texto impresso integralmente, indicando que o código funcionou!

Concluímos nosso primeiro objetivo, que era estabelecer uma entrada a partir de um arquivo, o **fluxo de entrada**, e uma saída para o console, representando o **fluxo de saída**.

Em seguida, trabalharemos dois conceitos que vemos muito presentes em nosso código, os termos `Stream` e `Reader`. Eles existem tanto para entrada quanto saída, mas por enquanto focaremos somente na entrada.

Primeiro, temos um a `Stream`, capaz de ler bits e bytes, um "*input stream of bytes*". Em contrapartida, há o `Reader`, que também faz uma leitura, só que esta é focada nos caracteres, "*reading character streams*".

Se precisamos ler uma imagem ou um PDF, por exemplo, utilizamos sempre o `Stream`, já se trabalhamos com um arquivo de texto, devemos utilizar o `Reader`.

Ademais, há algo ainda mais geral que o `FileInputStream`, um conceito que representa o fluxo de dados binários, que é a classe (abstrata) `InputStream`.

No mundo `Reader`, vimos duas classes, a `InputStreamReader` e `BufferedReader`. O que ambas têm em comum é que são `Readers`, ou seja, compete à elas a leitura de caracteres. Assim, o `Reader` também é um conceito, uma classe abstrata, que tem estas duas classes como filhos concretos.

É fundamental compreendermos a existência destes dois mundos, dos `Streams` e `Readers`, focados na leitura dos dados.

No Eclipse, podemos visualizar a classe `FileInputStream`:

```
//Código omitido

public class FileInputStream extends InputStream {

//Código omitido
```

COPIAR CÓDIGO

Vemos que ela estende `InputStream`. Ou seja, lembrando do conceito de polimorfismo, podemos utilizar este tipo mais genérico em nosso código, sem esquecer de importar esta classe:

```
//Código omitido

public class TesteLeitura {
```

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

                //Fluxo de Entrada com Arquivo
                InputStream fis = new FileInputStream("lorem.txt");
                InputStreamReader isr = new InputStreamReader(fis);
                BufferedReader br = BufferedReader(isr);

                String linha = br.readLine();

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

                System.out.println(linha);

                br.close();
        }
}
```

COPIAR CÓDIGO

O código continua funcionando. O próprio construtor do `InputStreamReader` funciona com um `InputStream`, não há necessidade de utilizarmos o tipo mais específico. Inclusive, ele é um `Reader`, e pode também ser representado pela classe mais genérica:

```
//Código omitido

public class TesteLeitura {

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

                //Fluxo de Entrada com Arquivo
                InputStream fis = new FileInputStream("lorem.txt");
                Reader isr = new InputStreamReader(fis);
                BufferedReader br = BufferedReader(isr);

                String linha = br.readLine();

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

                System.out.println(linha);

                br.close();
        }
}
```

COPIAR CÓDIGO

Na classe, vemos que ela estende a classe `Reader` :

```
//Código omitido

public class InputStreamReader extends Reader {

//Código omitido
```

COPIAR CÓDIGO

Que por sua vez, é uma classe abstrata:

```
//Código omitido

public abstract class Reader implements Readable, Closeable {

//Código omitido
```

COPIAR CÓDIGO

O mesmo é válido para a classe `InputStream`:

```
//Código omitido

public abstract class InputStream implements Closeable {

//Código omitido
```

COPIAR CÓDIGO

Retornando à classe `TesteLeitura`, vemos que o `BufferedReader` é capaz de receber um `Reader`, ou seja, não há necessidade de ser um tipo específico.

As classes `InputStream` e `Reader` são chamadas ***templates***, que são aquelas que pré-definem determinado conteúdo para as filhas.

Salvaremos e executaremos, o resultado no console permanece inalterado, indicando que nosso código continua funcionando.

Se observarmos a classe `BufferedReader` em detalhe:

```
//Código omitido

public class BufferedReader extends Reader {

//Código omitido
```

COPIAR CÓDIGO

Veremos que ela também é um `Reader`, assim, poderíamos pensar que assim como fizemos anteriormente, também será possível a substituição pelo tipo menos específico. Entretanto, a classe `Reader` não possui o método `readLine()`, necessário para a leitura do nosso arquivo, sendo assim, precisamos manter o `BufferedReader` neste casso.

Adiante, veremos como fazer a saída. Até a próxima!

Transcrição

Olá! Anteriormente, estabelecemos um fluxo de entrada concreto a partir de um arquivo. Nesta aula, nosso foco será o fluxo de saída.

Como estamos trabalhando com **saída**, em vez de utilizarmos o `InputStream`, faremos uso do `OutputStream`, e em vez do `Reader`, teremos o `Writer`. Apesar das diferentes nomenclaturas, os conceitos são os mesmos.

Assim, enquanto temos uma classe concreta `FileOutputStream`, temos acima dela a classe mãe, abstrata, `OutputStream`, que é análoga à `InputStream`. Ela é utilizada para manipular arquivos em formato PDF ou imagens, por exemplo.

Se quisermos ter uma entrada em texto, precisaremos das classes `OutputStreamWriter` e `BufferedWriter`, e são filhas da classe `Writer`.

No Eclipse, faremos uma cópia da classe `TesteLeitura`, que chamaremos de `TesteEscrita`. Com relação ao código da classe anterior, manteremos apenas o seguinte:

```
//Código omitido
```

```
public class TesteEscrita {

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

                //Fluxo de Entrada com Arquivo
                InputStream fis = new FileInputStream("lorem.txt");
                Reader isr = new InputStreamReader(fis);
                BufferedReader br = new BufferedReader(isr);

                br.close();

        }
}
```

COPIAR CÓDIGO

Onde há a palavra "*Input*", trocaremos para "*Output*", ou seja, trocaremos a entrada pela saída, e da mesma forma, a leitura se tornará escrita, trocaremos "*Reader*" por "*Writer*", onde for cabível.

Alteraremos também os nomes das variáveis, para maior clareza. Para não sobrescrevermos o arquivo, criaremos um `lorem2.txt` :

```
//Código omitido

public class TesteEscrita {

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

```
                //Fluxo de Entrada com Arquivo
                OutputStream fos = new FileOutputStream("lorem2.txt");
                Writer osw = new OutputStreamWriter(fos);
                BufferedWriter bw = new BufferedWriter(osw);

                bw.close();

        }
}
```

COPIAR CÓDIGO

Lembrando de importar as respectivas classes.

O próximo passo será escrevermos um conteúdo. Para isso, utilizamos o `BufferedWriter`, chamando pelo `bw`, o Eclipse nos apresenta uma série de métodos disponíveis, dentre eles temos o `write()`, que recebe uma `String`. É o que utilizaremos. Nele, escreveremos a mesma primeira linha do nosso arquivo `lorem.txt`.

Em seguida criaremos mais duas linhas em branco, e uma de texto:

```
//Código omitido

public class TesteEscrita {

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

                //Fluxo de Entrada com Arquivo
                OutputStream fos = new FileOutputStream("lorem2.txt");
```

```
                Writer osw = new OutputStreamWriter(fos);
                BufferedWriter bw = new BufferedWriter(osw);

                bw.write("Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do
                bw.newLine();
                bw.newLine();
                bw.write("asfasdfsafdas dfs sdf asf assdß");

                bw.close();

        }
}
```

COPIAR CÓDIGO

Podemos notar que aqui está presente o padrão de *decorator*, onde cada objeto decora a funcionalidade do anteriormente, da mesma forma como aconteceu no fluxo de entrada.

Salvaremos e executaremos, o resultado aparece como terminado mas nada é exibido no console. Isso acontece porque o Eclipse não percebeu que estamos trabalhando com um novo arquivo, para isso, teremos de atualizar. Clicaremos com o botão direito do mouse sobre a pasta `java-io` e selecionaremos a opção "Refresh".

No menu de arquivos, surgirá um novo, de nome `lorem2.txt` . Clicaremos nele e, ao abrirmos, veremos que ele contém o seguinte texto:

```
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod
```

```
asfasdfsafdas dfs sdf asf assdß
```

COPIAR CÓDIGO

Indicando que nosso programa funcionou.

Adiante, uniremos os dois códigos, ou seja, leremos e escreveremos em uma mesma oportunidade. Nos vemos lá.

Transcrição

Olá! Nesta aula, veremos como podemos copiar o conteúdo de um arquivo de texto para outro.

Utilizaremos o `TesteLeitura` como base. Faremos uma cópia desta classe, e nomearemos como `TesteCopiarArquivo`. Assim, temos o seguinte conteúdo:

```
//Código omitido
public class TesteCopiarArquivo {

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

                //Fluxo de Entrada com Arquivo
                InputStream fis = new FileInputStream("lorem.txt");
                Reader isr = new InputStreamReader(fis);
                BufferedReader br = new BufferedReader(isr);

                String linha = br.readLine();

                while(linha != null) {
                    System.out.println(linha);
```

```
            linha = br.readLine();
        }

        br.close();

    }
}
```

COPIAR CÓDIGO

Nela, há uma entrada já estabelecida. Seguindo isso, estabeleceremos a escrita. Para isso, podemos copiar o código da classe `TesteEscrita`, e teremos o seguinte resultado:

```
//Código omitido
public class TesteCopiarArquivo {

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

        //Fluxo de Entrada com Arquivo
        InputStream fis = new FileInputStream("lorem.txt");
        Reader isr = new InputStreamReader(fis);
        BufferedReader br = new BufferedReader(isr);

        OutputStream fos = new FileOutputStream("lorem2.txt");
        Writer osw = new OutputStreamWriter(fos);
        BufferedWriter bw = new BufferedWriter(osw);

        String linha = br.readLine();
```

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

                br.close()

        }
}
```

COPIAR CÓDIGO

Copiaremos o conteúdo do arquivo `lorem.txt` para o `lorem2.txt` .

Para tanto, não queremos mostrar isso no console, ou seja, queremos escrever para o `BufferedWriter` . Assim, em vez de utilizarmos o método `System.out.println` , faremos uso do `br.write()` , passando como parâmetro nossa `String linha` :

```
//Código omitido
public class TesteCopiarArquivo {

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

                //Fluxo de Entrada com Arquivo
                InputStream fis = new FileInputStream("lorem.txt");
                Reader isr = new InputStreamReader(fis);
                BufferedReader br = new BufferedReader(isr);
```

```
                OutputStream fos = new FileOutputStream("lorem2.txt");
                Writer osw = new OutputStreamWriter(fos);
                BufferedWriter bw = new BufferedWriter(osw);

                String linha = br.readLine();

                while(linha != null) {
                    bw.write(linha);
                    linha = br.readLine();
                }

                br.close()

        }
}
```

COPIAR CÓDIGO

Ao final, além de fecharmos o `BufferedReader` (`br.close()`), temos de fechar também o `BufferedWriter` :

```
//Código omitido
public class TesteCopiarArquivo {

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

                InputStream fis = new FileInputStream("lorem.txt");
                Reader isr = new InputStreamReader(fis);
                BufferedReader br = new BufferedReader(isr);
```

```
                OutputStream fos = new FileOutputStream("lorem2.txt");
                Writer osw = new OutputStreamWriter(fos);
                BufferedWriter bw = new BufferedWriter(osw);

                String linha = br.readLine();

                while(linha != null) {
                    bw.write(linha);
                    linha = br.readLine();
                }

                br.close();
                bw.close();

        }
}
```

COPIAR CÓDIGO

Salvaremos e executaremos o código. Atualizaremos a pasta `java-io` , e abriremos o arquivo `lorem2.txt` . Foi feita uma cópia do conteúdo escrito do arquivo `lorem.txt` , mas não houve uma quebra de linha. Criaremos esta quebra, adicionando uma `newLine()` dentro do laço `while` :

```
//Código omitido
public class TesteCopiarArquivo {

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

```
                InputStream fis = new FileInputStream("lorem.txt");
                Reader isr = new InputStreamReader(fis);
                BufferedReader br = new BufferedReader(isr);

                OutputStream fos = new FileOutputStream("lorem2.txt");
                Writer osw = new OutputStreamWriter(fos);
                BufferedWriter bw = new BufferedWriter(osw);

                String linha = br.readLine();

                while(linha != null) {

                bw.write(linha);
                    bw.newLine();
                    linha = br.readLine();
                }

                br.close();
                bw.close();

        }
}
```

COPIAR CÓDIGO

Executando mais uma vez, temos o seguinte resultado no console:

```
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod
tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam,
quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo
consequat. Duis aute irure dolor in reprehenderit in voluptate velit
esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat
cupidatat non proident, sunt in culpa qui officia deserunt mollit anim
id est laborum.
```

COPIAR CÓDIGO

Funcionou, temos uma cópia idêntica de `lorem.txt` .

As classes `InputStream` e `OutputStream` são bases do mundo `java.io` , há milhares de bibliotecas que as utilizam.

Recapitulando, nós partimos de um arquivo, estabelecendo um fluxo de entrada, e seguimos em direção a um novo arquivo, como fluxo de saída.

A entrada concreta pode acontecer de diversas formas, bem como a saída concreta, o importante é sabermos que a forma como elas ocorrem é menos importante que os fluxos de entrada e de saída.

Isto pode ser observado na classe `TesteCopiarArquivo` , onde utilizamos sempre o `InputStream` ou `OutputStream` , em tradução do inglês, "fluxo de entrada" e "fluxo de saída":

```
//Código omitido
```

```
InputStream fis = new FileInputStream("lorem.txt");
//Código omitido

OutputStream fos = new FileOutputStream("lorem2.txt");
//Código omitido
```

COPIAR CÓDIGO

O tipo concreto de entrada pode variar, por exemplo, pode passar a ser o teclado. Para isso, utilizamos o `System.in` :

```
//Código omitido
public class TesteCopiarArquivo {

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

                InputStream fis = System.in;
                Reader isr = new InputStreamReader(fis);
                BufferedReader br = new BufferedReader(isr);

                OutputStream fos = new FileOutputStream("lorem2.txt");
                Writer osw = new OutputStreamWriter(fos);
                BufferedWriter bw = new BufferedWriter(osw);

                String linha = br.readLine();

                while(linha != null) {
```

```
                bw.write(linha);
                    bw.newLine();
                    linha = br.readLine();
                }

                br.close();
                bw.close();

        }
}
```

COPIAR CÓDIGO

Executaremos. O console nunca encerra a execução, a JVM não para de rodar, isso acontece porque o programa está aguardando um input do usuário.

Assim, podemos digitar qualquer texto, e continuar digitando, de qualquer forma que o façamos, em momento algum a aplicação para de ser executada. Isso porque não há o gatilho para que ela pare, que é o momento em que a linha for `null`. Nós não somos capazes de simular manualmente a condição de saída do laço, `null`.

Ao abrirmos o arquivo `lorem2.txt`, vemos que o que escrevemos no console foi passado para o arquivo, mas não é garantido, já que estamos trabalhando com um `BufferedWriter`. Isso significa que ele guarda todos os caracteres e, em um momento posterior - ao fazer o `close()` -, escreve o que foi armazenado.

O programa está funcionando, mas precisamos fazer com que a execução pare sem a necessidade de o fazermos manualmente. Em nosso laço `while`, criaremos uma nova condição, além de ser diferente de `null`, ela não deve ser vazia. Para isso utilizamos a negativa do método `isEmpty()`:

```
//Código omitido
public class TesteCopiarArquivo {

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

                InputStream fis = System.in;
                Reader isr = new InputStreamReader(fis);
                BufferedReader br = new BufferedReader(isr);

                OutputStream fos = new FileOutputStream("lorem2.txt");
                Writer osw = new OutputStreamWriter(fos);
                BufferedWriter bw = new BufferedWriter(osw);

                String linha = br.readLine();

                while(linha != null && !linha.isEmpty()) {

                bw.write(linha);
                    bw.newLine();
                    linha = br.readLine();
                }

                br.close();
                bw.close();

        }
}
```

COPIAR CÓDIGO

Desta forma, o laço só funcionará quando a linha não for nula ou não estiver vazia. Executaremos, no console escreveremos:

```
mais coisas
outras coisas
oi
sadf
```

COPIAR CÓDIGO

Pressionaremos a tecla "Enter", deixando uma linha em branco. Desta forma, a máquina virtual encerrou a execução. No arquivo `lorem2.txt` lemos:

```
mais coisas
outras coisas
oi
sadf
```

COPIAR CÓDIGO

Portanto, nosso programa funcionou.

Se retornarmos para o `FileInputStream("lorem.txt")` o programa funcionará normalmente, e fará uma cópia do arquivo para o `lorem2.txt` , como havíamos feito anteriormente. Manteremos esta opção em comentários, por enquanto trabalharemos com o `System.in` .

A seguir, testaremos uma variação da saída concreta, o console. Neste caso, utilizamos o `System.out` :

```
//Código omitido
public class TesteCopiarArquivo {

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

                InputStream fis = System.in; //new FileInputStream("lorem.txt");
                Reader isr = new InputStreamReader(fis);
                BufferedReader br = new BufferedReader(isr);

                OutputStream fos = System.out; //new FileOutputStream("lorem2.txt");
                Writer osw = new OutputStreamWriter(fos);
                BufferedWriter bw = new BufferedWriter(osw);

                String linha = br.readLine();

                while(linha != null && !linha.isEmpty()) {

                bw.write(linha);
                    bw.newLine();
                    linha = br.readLine();
                }

                br.close();
                bw.close();

        }
}
```

COPIAR CÓDIGO

Executaremos o código. Primeiro, ele espera que haja uma entrada, digitaremos:

```
escrever algo
sfdasdfasdf
asdfasdfsa
```

COPIAR CÓDIGO

Pressionaremos a tecla "Enter", e internamente o `BufferedWriter` guarda esta informação, ao criarmos uma linha em branco, o programa imprime o que escrevemos e temos o seguinte resultado no console:

```
escrever algo
sfdasdfasdf
asdfasdfsa

escrever algo
sfdasdfasdf
asdfasdfsa
```

COPIAR CÓDIGO

Funcionou! Temos a entrada, e em seguida o resultado impresso no console, ou seja, a saída. Para que isso aconteça de forma imediata, sem a necessidade de uma linha em branco entre a entrada e a saída, utilizaremos o método `flush()`:

```
//Código omitido
public class TesteCopiarArquivo {
```

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

        InputStream fis = System.in; //new FileInputStream("lorem.txt");
        Reader isr = new InputStreamReader(fis);
        BufferedReader br = new BufferedReader(isr);

        OutputStream fos = System.out; //new FileOutputStream("lorem2.txt");
        Writer osw = new OutputStreamWriter(fos);
        BufferedWriter bw = new BufferedWriter(osw);

        String linha = br.readLine();

        while(linha != null && !linha.isEmpty()) {

            bw.write(linha);
            bw.newLine();
            bw.flush();
            linha = br.readLine();
        }

        br.close();
        bw.close();

    }
}
```

COPIAR CÓDIGO

Portanto, realizaremos um novo teste. Escreveremos um texto e pressionaremos a tecla "Enter". Temos o seguinte resultado no console:

```
escreve algo
escreve algo
```

COPIAR CÓDIGO

Funcionou, temos a entrada e a impressão. O programa continua rodando, até fazermos com que pare, gerando uma linha em branco.

Temos um código cada vez mais genérico e flexível.

Se quisermos ler o arquivo `lorem.txt` e imprimi-lo em seguida, é possível fazermos isso, retornando o código da linha `InputStream` para o que havíamos mantido em comentários:

```
//Código omitido
public class TesteCopiarArquivo {

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

                InputStream fis = new FileInputStream("lorem.txt");
                Reader isr = new InputStreamReader(fis);
                BufferedReader br = new BufferedReader(isr);

                OutputStream fos = System.out; //new FileOutputStream("lorem2.txt");
                Writer osw = new OutputStreamWriter(fos);
```

```
                BufferedWriter bw = new BufferedWriter(osw);

                String linha = br.readLine();

                while(linha != null && !linha.isEmpty()) {

                    bw.write(linha);
                    bw.newLine();
                    bw.flush();
                    linha = br.readLine();
                }

                br.close();
                bw.close();

        }
}
```

COPIAR CÓDIGO

Ao executarmos a classe, temos impresso no console todo o conteúdo do arquivo `lorem.txt` .

Se quisermos ler uma entrada do teclado, e imprimi-la no arquivo, basta mantermos o `System.in` , e retornarmos o `OutputStream` para o `FileOutputStream()` :

```
//Código omitido
public class TesteCopiarArquivo {
```

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

                InputStream fis = System.in; //new FileInputStream("lorem.txt");
                Reader isr = new InputStreamReader(fis);
                BufferedReader br = new BufferedReader(isr);

                OutputStream fos = new FileOutputStream("lorem2.txt");
                Writer osw = new OutputStreamWriter(fos);
                BufferedWriter bw = new BufferedWriter(osw);

                String linha = br.readLine();

                while(linha != null && !linha.isEmpty()) {

                    bw.write(linha);
                    bw.newLine();
                    bw.flush();
                    linha = br.readLine();
                }

                br.close();
                bw.close();

        }
}
```

COPIAR CÓDIGO

Executaremos, e digitaremos o seguinte texto no console:

```
oi oi oi
oi oi
oi
```

COPIAR CÓDIGO

Encerraremos a execução e, abrindo o arquivo `lorem2.txt`, vemos impresso exatamente isso que acabamos de digitar.

Percebemos que, sem fazermos grandes alterações ao código, é possível alterarmos a forma de entrada ou saída concreta. Funciona inclusive para a rede, contudo, é algo que não conseguimos simular por enquanto.

A comunicação via rede se dá de forma análoga ao telefone, na parte em que ouvimos, seria localizado o `OutputStream`, enquanto que a extremidade por onde falamos pode ser considerada como a `InputStream`. O telefone, no caso do nosso código, se chama `Socket`.

Em Java, temos uma classe com esse nome e, para utilizá-la, precisamos instanciá-la:

```
//Código omitido
public class TesteCopiarArquivo {

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

                new Socket();

                InputStream fis = System.in; //new FileInputStream("lorem.txt");
                Reader isr = new InputStreamReader(fis);
```

```
                BufferedReader br = new BufferedReader(isr);

                OutputStream fos = new FileOutputStream("lorem2.txt");
                Writer osw = new OutputStreamWriter(fos);
                BufferedWriter bw = new BufferedWriter(osw);

                String linha = br.readLine();

                while(linha != null && !linha.isEmpty()) {

                    bw.write(linha);
                    bw.newLine();
                    bw.flush();
                    linha = br.readLine();
                }

                br.close();
                bw.close();

        }
}
```

COPIAR CÓDIGO

Nosso `Socket` será representado pela variável `s` :

```
//Código omitido
public class TesteCopiarArquivo {
```

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

                Socket s = new Socket().getInputStream();

                InputStream fis = System.in; //new FileInputStream("lorem.txt");
                Reader isr = new InputStreamReader(fis);
                BufferedReader br = new BufferedReader(isr);

                OutputStream fos = new FileOutputStream("lorem2.txt");
                Writer osw = new OutputStreamWriter(fos);
                BufferedWriter bw = new BufferedWriter(osw);

                String linha = br.readLine();

                while(linha != null && !linha.isEmpty()) {

                    bw.write(linha);
                    bw.newLine();
                    bw.flush();
                    linha = br.readLine();
                }

                br.close();
                bw.close();

        }
}
```

COPIAR CÓDIGO

Em seguida, precisamos criar uma conexão. Por meio do `Socket`, podemos utilizar um `get()` e obter o `InputStream`:

```
//Código omitido
public class TesteCopiarArquivo {

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

                Socket s = new Socket().getInputStream();

                InputStream fis = s.getInputStream(); //System.in; //new FileInputStream(
                Reader isr = new InputStreamReader(fis);
                BufferedReader br = new BufferedReader(isr);

                OutputStream fos = new FileOutputStream("lorem2.txt");
                Writer osw = new OutputStreamWriter(fos);
                BufferedWriter bw = new BufferedWriter(osw);

                String linha = br.readLine();

                while(linha != null && !linha.isEmpty()) {

                    bw.write(linha);
                    bw.newLine();
                    bw.flush();
                    linha = br.readLine();
                }
```

```
                br.close();
                bw.close();

        }
}
```

COPIAR CÓDIGO

Da mesma forma, podemos obter o `OutputStream` :

```
//Código omitido
public class TesteCopiarArquivo {

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

                Socket s = new Socket();

                InputStream fis = s.getInputStream(); //System.in; //new FileInputStream(
                Reader isr = new InputStreamReader(fis);
                BufferedReader br = new BufferedReader(isr);

                OutputStream fos = s.getOutputStream(); //System.out; //new FileOutputStr
                Writer osw = new OutputStreamWriter(fos);
                BufferedWriter bw = new BufferedWriter(osw);

                String linha = br.readLine();

                while(linha != null && !linha.isEmpty()) {
```

```
                bw.write(linha);
                bw.newLine();
                bw.flush();
                linha = br.readLine();
            }

            br.close();
            bw.close();

        }
}
```

COPIAR CÓDIGO

Assim, temos três formas de entrada e saída concretas, a rede, console, e o arquivo.

Notamos que o programa é bastante flexível, com poucas alterações, alternamos entre estes tipos de entrada ou saída. Isto é importante, já que existem diversas bibliotecas que utilizam tanto o `InputStream` quanto o `OutputStream`.

No site da Caelum (https://www.caelum.com.br/apostila-java-web/), encontramos a apostila Java para Desenvolvimento Web. Nela, um dos tópicos, que também é uma das ferramentas fundamentais para o desenvolvimento web, trata dos *servlets*.

Servlet é um objeto Java que funciona como um mini servidor. Este, por sua vez, tem por função receber e devolver dados, ou seja, `java.io`.

Tomemos o seguinte código como exemplo:

```
public class OiMundo extends HttpServlet {

        protected void service (HttpServletRequest request, HttpServletResponse response)

                PrintWriter out = response.getWriter();

                //escreve o texto
                out.println("<html>");
                out.println("<body>");
                out.println("Primeira servlet");
                out.println("</body>");
                out.println("</html>");

        }
}
```

COPIAR CÓDIGO

Queremos retornar, ou seja, responder, e fazemos isso por meio de um `Writer()`, como é o caso em `response.getWriter()`. Para isso, utilizamos uma classe chamada `PrintWriter`, que assim como as classes `OutputStreamWriter` e `BufferedWriter`, também estende `Writer`.

Por isso, entendemos que ela tem por finalidade escrever caracteres.

Aqui temos um exemplo de outra biblioteca, um outro contexto, no qual utilizamos o *input* e *output*.

Atualmente, o mundo web é dominado por este fluxo de informações. Tudo isso funciona graças ao `java.io`.

Transcrição

Neste vídeo faremos uma breve revisão do que foi visto neste capítulo.

Vimos, de forma geral, os mundos da entrada `Input`, e da saída, `Output`. Estes mundos se subdividem em `InputStream` e `Reader` no primeiro caso, e `OutputStream` e `Writer` no segundo.

Além disso, temos a divisão entre *streams*, e *readers* e *writers*. `InputStream` e `OutputStream` lidam com dados binários, por exemplo imagens e PDFs, já se estivermos lidando com caracteres, utilizamos o `Reader` ou `Writer`.

Há ainda as classes que fazem a transição de um mundo para outro, como é o caso da `InputStreamReader`, que recebe um `InputStream` de bytes e o transforma em um `Reader`. Da mesma forma, temos o `OutputStreamWriter`, que faz o mesmo, só que para a escrita. Estas classes possuem padrões de projetos, próprios do `java.io`.

Com essa base, podemos partir para outras classes como `Scanner` e `PrintStream`. Até a próxima!

Transcrição

Caso queira, você pode fazer o download (https://s3.amazonaws.com/caelum-online-public/857-java-io/03/java7-aula3.zip) do projeto completo feito até a aula anterior.

Anteriormente, havíamos criado a classe `TesteEscrita`, onde estabelecemos uma entrada de texto, análoga à leitura de código.

Nosso objetivo nesta aula será estabelecer uma saída, só que de forma mais simples.

Criaremos uma cópia da classe `TesteEscrita`, que chamaremos de `TesteEscrita2`.

Trabalharemos com uma classe que é capaz de trabalhar diretamente com um arquivo que já contém caracteres, em vez de os escrevermos diretamente no código. Esta classe se chama `FileWriter`.

Comentaremos o fluxo de entrada de arquivo que havíamos criado, e importaremos esta nova classe:

```
//Código omitido
```

```
public class TesteEscrita2 {

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

        //Fluxo de Entrada com Arquivo
        //OutputStream fos = new FileOutputStream("lorem2.txt");
        //Writer osw = new OutputStreamWriter(fos);
        //BufferedWriter bw = new BufferedWriter(osw);

        FileWriter

        br.write("Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod
        br.newLine();
        br.newLine();
        br.write("asfasdfsafdas dfs sdf asf assdß");

        bw.close();

        }
}
```

COPIAR CÓDIGO

Chamaremos este `FileWriter` de `fw`, e utilizaremos um construtor que recebe uma `String` referente ao nome do arquivo com o qual desejamos trabalhar, no caso `lorem2.txt`:

```
//Código omitido
```

```
public class TesteEscrita2 {

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

        //Fluxo de Entrada com Arquivo
        //OutputStream fos = new FileOutputStream("lorem2.txt");
        //Writer osw = new OutputStreamWriter(fos);
        //Buff3eredWriter bw = new BufferedWriter(osw);

        FileWriter fw = new FileWriter("lorem2.txt");

        br.write("Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod
        br.newLine();
        br.newLine();
        br.write("asfasdfsafdas dfs sdf asf assdß");

        bw.close();

        }
}
```

COPIAR CÓDIGO

Estabelecemos assim uma saída com um arquivo.

A classe `FileWriter` possui um método que nos permite escrever uma `String` , ou seja, escrever uma linha. Testaremos com a primeira linha do nosso arquivo de texto, como havíamos feito anteriormente. Continuaremos utilizando o `fw` nos demais métodos:

```
//Código omitido

public class TesteEscrita2 {

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

        //Fluxo de Entrada com Arquivo
        //OutputStream fos = new FileOutputStream("lorem2.txt");
        //Writer osw = new OutputStreamWriter(fos);
        //Buff3eredWriter bw = new BufferedWriter(osw);

        FileWriter fw = new FileWriter("lorem2.txt");
        fw.write("Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod
        fw.newLine();
        fw.write("asfasdfsafdas dfs sdf asf assdß");

        fw.close();

        }
}
```

COPIAR CÓDIGO

Entretanto, percebemos que não existe o método `newLine()` na classe `FileWriter`. Para criarmos esta linha em branco, escreveremos um novo `String`, e indicaremos dentro dele que se trata de uma nova linha. Para isso, utilizamos certos caracteres especiais, como é o caso da barra invertida (`\`), seguida da letra "n", portanto `\n`:

```
//Código omitido

public class TesteEscrita2 {

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

        //Fluxo de Entrada com Arquivo
        //OutputStream fos = new FileOutputStream("lorem2.txt");
        //Writer osw = new OutputStreamWriter(fos);
        //Buff3eredWriter bw = new BufferedWriter(osw);

        FileWriter fw = new FileWriter("lorem2.txt");
        fw.write("Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod
        fw.write("\n");
        fw.write("asfasdfsafdas dfs sdf asf assdß");

        fw.close();

        }
}
```

COPIAR CÓDIGO

Isto pode variar entre sistemas operacionais, para os sistemas Linus e MacOSx basta fazermos como está escrito acima, contudo, para o Windows OS, devemos escrever `\r\n` , indicando o retorno de uma nova linha. Inseriremos duas linhas:

```
//Código omitido

public class TesteEscrita2 {

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

        //Fluxo de Entrada com Arquivo
        //OutputStream fos = new FileOutputStream("lorem2.txt");
        //Writer osw = new OutputStreamWriter(fos);
        //Buff3eredWriter bw = new BufferedWriter(osw);

        FileWriter fw = new FileWriter("lorem2.txt");
        fw.write("Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod
        fw.write("\r\n");
        fw.write("\r\n");
        fw.write("asfasdfsafdas dfs sdf asf assdß");

        fw.close();

        }
}
```

COPIAR CÓDIGO

Ao abrirmos o arquivo, o editor de texto saberá interpretar estes caracteres especiais.

Salvaremos e executaremos, temos o segundo resultado, no arquivo `lorem2.txt` :

```
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod

asfasdfsafdas dfs sdf asf assdß
```

COPIAR CÓDIGO

Funcionou.

Contudo, utilizar estes caracteres especiais para criar estas linhas em branco não é uma solução elegante, principalmente considerando estas disparidades entre sistemas operacionais. Para deixarmos nosso código mais robusto, podemos contar com um método do Java que nos devolve estes caracteres com base no sistema operacional que estamos utilizando.

Começamos chamando-o com a palavra `System` , e em seguida selecionaremos o método `lineSeparator()` :

```
//Código omitido

public class TesteEscrita2 {

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

        //Fluxo de Entrada com Arquivo
        //OutputStream fos = new FileOutputStream("lorem2.txt");
        //Writer osw = new OutputStreamWriter(fos);
        //Buff3eredWriter bw = new BufferedWriter(osw);

        FileWriter fw = new FileWriter("lorem2.txt");
```

```
        fw.write("Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod
        fw.write(System.lineSeparator());
        fw.write(System.lineSeparator());
        fw.write("asfasdfsafdas dfs sdf asf assdß");

        fw.close();

        }
}
```

COPIAR CÓDIGO

Do ponto de vista semântico, isto facilita a compreensão do código, já que explicita nossa intenção de criar uma separação em linha. Executaremos novamente o programa e veremos que tudo continuará funcionando.

Por mais que o `FileWriter` atenda às nossas necessidades, ainda assim, é recomendável continuarmos utilizando o `BufferedWriter` e apenas passarmos o `fw` no seu construtor:

```
//Código omitido

public class TesteEscrita2 {

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

        //Fluxo de Entrada com Arquivo
        //OutputStream fos = new FileOutputStream("lorem2.txt");
        //Writer osw = new OutputStreamWriter(fos);
```

```
        //Buff3eredWriter bw = new BufferedWriter(osw);

        FileWriter fw = new FileWriter("lorem2.txt");
        BufferedWriter bw = new BufferedWriter(fw);
        fw.write("Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod
        fw.write(System.lineSeparator());
        fw.write(System.lineSeparator());
        fw.write(System.lineSeparator());
        fw.write("asfasdfsafdas dfs sdf asf assdß");

        fw.close();

        }
}
```

COPIAR CÓDIGO

Assim, criamos a saída, e estamos a "embrulhado", em um `BufferedWriter`. Portanto, utilizamos o `bw`, trabalhando sempre com este fluxo. Para nos organizarmos, manteremos duas linhas entre os textos e, como voltamos a utilizar o `bw`, podemos fazer uso do método `newLine()`:

```
//Código omitido

public class TesteEscrita2 {

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

        //Fluxo de Entrada com Arquivo
```

```
        //OutputStream fos = new FileOutputStream("lorem2.txt");
        //Writer osw = new OutputStreamWriter(fos);
        //Buff3eredWriter bw = new BufferedWriter(osw);

        FileWriter fw = new FileWriter("lorem2.txt");
        BufferedWriter bw = new BufferedWriter(fw);
        bw.write("Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod
        bw.newLine();
        bw.newLine();
        bw.write("asfasdfsafdas dfs sdf asf assdß");

        bw.close();

        }
}
```

COPIAR CÓDIGO

Para simplificarmos o código, podemos mover a criação do `FileWriter` diretamente para o `BufferedWriter` :

```
//Código omitido

public class TesteEscrita2 {

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

        //Fluxo de Entrada com Arquivo
        //OutputStream fos = new FileOutputStream("lorem2.txt");
```

```
        //Writer osw = new OutputStreamWriter(fos);
        //Buff3eredWriter bw = new BufferedWriter(osw);

        BufferedWriter bw = new BufferedWriter(FileWriter fw = new FileWriter("lorem2.txt
        bw.write("Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod
        bw.newLine();
        bw.newLine();
        bw.write("asfasdfsafdas dfs sdf asf assdß");

        bw.close();

        }
}
```

COPIAR CÓDIGO

Adiante, veremos ainda outra forma de fazermos isso.

Transcrição

Anteriormente havíamos estabelecido uma nova entrada, usando diretamente um `Writer`. Contudo, é possível atingirmos este mesmo resultado com um código ainda mais simples.

De início, alteraremos o nome da nossa classe, para que evidencia o tipo de teste que estamos fazendo. Ela passará a se chamar `TesteEscritaFileWriter`:

```
//Código omitido

public class TesteEscritaFileWriter {

        //Restante do código omitido
```

COPIAR CÓDIGO

Como a classe é pública, o compilador indica que há um erro, pois alteramos o nome da classe, e assim ele não corresponde mais ao nome do arquivo. Para resolver este problema, basta renomearmos o arquivo para que tenha o mesmo nome da classe.

Faremos uma cópia da classe `TesteEscritaFileWriter`, e daremos o nome de `TesteEscrita3`, e comentaremos a linha em que temos o `BufferedWriter`:

```
//Código omitido

public class TesteEscrita3 {

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

        //Fluxo de Entrada com Arquivo
        //OutputStream fos = new FileOutputStream("lorem2.txt");
        //Writer osw = new OutputStreamWriter(fos);
        //Buff3eredWriter bw = new BufferedWriter(osw);

        //BufferedWriter bw = new BufferedWriter( new FileWriter("lorem2.txt"));

                bw.write("Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do
        bw.newLine();
        bw.newLine();
        bw.write("asfasdfsafdas dfs sdf asf assdß");

        bw.close();

        }
}
```

COPIAR CÓDIGO

Aqui, utilizaremos uma classe chamada `PrintStream`. Por meio dela é possível fazermos uma impressão para um fluxo binário.

Não podemos esquecer de importar a classe `PrintStream`.

Ela será representada pela variável `ps`, e terá um construtor que receberá uma `String` com o nome do arquivo:

```
//Código omitido

public class TesteEscrita3 {

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

        //Fluxo de Entrada com Arquivo
        //OutputStream fos = new FileOutputStream("lorem2.txt");
        //Writer osw = new OutputStreamWriter(fos);
        //Buff3eredWriter bw = new BufferedWriter(osw);

        //BufferedWriter bw = new BufferedWriter(new FileWriter("lorem2.txt"));

                PrintStream ps = new PrintStream("lorem2.txt");

                bw.write("Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do
        bw.newLine();
        bw.newLine();
        bw.write("asfasdfsafdas dfs sdf asf assdß");
```

```
        bw.close();

        }
}
```

COPIAR CÓDIGO

Abrindo a classe `PrintStream`, vemos que ela existe desde a versão 1.0 do Java, enquanto as `FileWriter` e `BufferedWriter` entraram somente na versão 1.1. Ou seja, aqueles que desejavam trabalhar com caracteres desde o Java 1.0 utilizavam, necessariamente, a classe `PrintStream`. A partir disso, foram criadas ferramentas mais especializadas.

Agora que temos o `ps`, podemos utiliza-lo para imprimir caracteres, e para isso há o método `println()`. Ele é sobrecarregado, ou seja, possui várias versões e, dentre elas, a que recebe uma `String`, e que utilizaremos para imprimir a primeira linha do nosso arquivo de texto:

```
//Código omitido

public class TesteEscrita3 {

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

        //Fluxo de Entrada com Arquivo
        //OutputStream fos = new FileOutputStream("lorem2.txt");
        //Writer osw = new OutputStreamWriter(fos);
        //Buff3eredWriter bw = new BufferedWriter(osw);
```

```
        //BufferedWriter bw = new BufferedWriter( new FileWriter("lorem2.txt"));

                PrintStream ps = new PrintStream("lorem2.txt");

                ps.println("Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed

        bw.newLine();
        bw.newLine();
        bw.write("asfasdfsafdas dfs sdf asf assdß");

        bw.close();

        }
}
```

COPIAR CÓDIGO

Para criarmos uma quebra de linha, podemos simplesmente imprimir uma nova, utilizando o `println()` , vazio:

```
//Código omitido

public class TesteEscrita3 {

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

        //Fluxo de Entrada com Arquivo
        //OutputStream fos = new FileOutputStream("lorem2.txt");
```

```
        //Writer osw = new OutputStreamWriter(fos);
        //Buff3eredWriter bw = new BufferedWriter(osw);

        //BufferedWriter bw = new BufferedWriter(new FileWriter("lorem2.txt"));

                PrintStream ps = new PrintStream("lorem2.txt");

                ps.println("Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed
        ps.println();

        bw.newLine();
        bw.newLine();
        bw.write("asfasdfsafdas dfs sdf asf assdß");

        bw.close();

        }
}
```

COPIAR CÓDIGO

Por fim, passaremos a segunda linha de texto que queremos imprimir, e fecharemos com a referência `ps` :

```
//Código omitido

public class TesteEscrita3 {

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

```
        //Fluxo de Entrada com Arquivo
        //OutputStream fos = new FileOutputStream("lorem2.txt");
        //Writer osw = new OutputStreamWriter(fos);
        //Buff3eredWriter bw = new BufferedWriter(osw);

        //BufferedWriter bw = new BufferedWriter(new FileWriter("lorem2.txt"));

                PrintStream ps = new PrintStream("lorem2.txt");

                ps.println("Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed
    ps.println();
                ps.println("asfasdfsafdas dfs sdf asf assdß");

        ps.close();

        }
}
```

COPIAR CÓDIGO

Salvaremos e executaremos a classe. Abrindo o arquivo `lorem2.txt` vemos que os textos foram impressos sem problemas, indicando que tudo funcionou corretamente.

O `PrintStream` é uma classe de mais alto nível, que aceita uma grande variedade de construtores, como é o caso do `new File()`.

Além disso, temos o `println()` que já inclui um pulo de linha sempre que o utilizamos. Já o conhecíamos do `System.out.println()`, onde `out` nada mais é que um `PrintStream`, só que este não está vinculado a um arquivo específico, e sim ao console. Por isso essa impressão é feita no console.

Temos outra classe, que funciona de forma análoga a essa, e se chama `PrintWriter`. Comentaremos o que havíamos feito, e criaremos um novo `ps`:

```
//Código omitido

public class TesteEscrita3 {

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

        //Fluxo de Entrada com Arquivo
        //OutputStream fos = new FileOutputStream("lorem2.txt");
        //Writer osw = new OutputStreamWriter(fos);
        //Buff3eredWriter bw = new BufferedWriter(osw);

        //BufferedWriter bw = new BufferedWriter( new FileWriter("lorem2.txt"));
                //PrintStream ps = new PrintStream(new File("lorem2.txt"));

                PrintWriter ps = new PrintWriter();

                ps.println("Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed
    ps.println();
                ps.println("asfasdfsafdas dfs sdf asf assdß");
```

```
        ps.close();

        }
}
```

COPIAR CÓDIGO

Ele receberá o nome do arquivo:

```
//Código omitido

public class TesteEscrita3 {

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

        //Fluxo de Entrada com Arquivo
        //OutputStream fos = new FileOutputStream("lorem2.txt");
        //Writer osw = new OutputStreamWriter(fos);
        //Buff3eredWriter bw = new BufferedWriter(osw);

        //BufferedWriter bw = new BufferedWriter(new FileWriter("lorem2.txt"));
                //PrintStream ps = new PrintStream(new File("lorem2.txt"));

                PrintWriter ps = new PrintWriter("lorem2.txt");

                ps.println("Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed
        ps.println();
                ps.println("asfasdfsafdas dfs sdf asf assdß");
```

```
            ps.close();

            }
    }
```

COPIAR CÓDIGO

Salvaremos e executaremos, temos o mesmo resultado. Continua funcionando perfeitamente.

Como os `Writers` e `Readers` foram criados após o `Stream`, eles têm funções muito similares. Inicialmente existia somente o `PrintStream`, mas como depois surgiu o mundo de `Writers`, viu-se a necessidade de criar um `PrintWriter`, este que não precisa utilizar um `Stream` internamente.

Falamos aqui sobre a saída e, adiante, falaremos sobre a entrada. Até lá!

Transcrição

*O arquivo **contas.csv** pode ser baixado aqui (https://s3.amazonaws.com/caelum-online-public/857-java-io/04/contas.csv). E caso queira, você pode fazer o download (https://s3.amazonaws.com/caelum-online-public/857-java-io/04/java7-aula4.zip) do projeto completo feito até a aula anterior.*

Já falamos sobre alternativas de **saída** e ,nesta aula, nosso foco será na **entrada**.

Renomearemos a classe `TesteEscrita3` para `TesteEscritaPrintStreamPrintWriter` , lembrando de renomear o arquivo para que ela continue a compilar.

Trabalharemos com um arquivo de extensão `.csv` . O "c" é para a palavra "*comma*", "s" para "*separated*", e "v" para "*values*", o que significa "valores separados por vírgulas". Dentro deste arquivo `contas.csv` temos os seguintes dados:

```
CC,22,33,Nico Steppat,210.1
CP,11,55,Luan Silva,1300.98
CC,22,44,Ana Garcias,350.40
```

COPIAR CÓDIGO

Este tipo de arquivo é comumente utilizado na área de ciência de dados.

Primeiro temos o tipo de conta, `CC` para conta corrente, e `CP` para conta poupança. Em seguida temos o número da agência, conta, nome do titular, e por fim o saldo.

Inseriremos este arquivo na pasta raiz do nosso projeto, `java.io`.

Criaremos uma nova classe chamada `TesteLeitura2`. Nela, utilizaremos a classe `Scanner`, que advém do pacote `java.util`. No seu construtor, passaremos o `new File`, com cuidado para não confundirmos com o `String source` com o `String fileName`, que utilizávamos anteriormente:

```
package br.com.alura.java.io.teste;

import java.util.Scanner;

public class TesteLeitura2; {

        public static void main(String[] args) {

                Scanner scanner = new Scanner(new File);

        }
}
```

COPIAR CÓDIGO

O `new File` serve para representarmos o arquivo que desejamos ler, por meio de um novo objeto. O `File` receberá o nome do arquivo `contas.csv` :

```
package br.com.alura.java.io.teste;

import java.util.Scanner;

public class TesteLeitura2; {

        public static void main(String[] args) {

                Scanner scanner = new Scanner(new File("contas.csv"));

        }
}
```

COPIAR CÓDIGO

Precisamos criar o `throws` para o `Exception` , especificamente. Assim, evitamos a confusão com as demais exceções:

```
package br.com.alura.java.io.teste;

import java.util.Scanner;

public class TesteLeitura2; throws FileNotFoundException {
```

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

                Scanner scanner = new Scanner(new File("contas.csv"));

        }
}
```

COPIAR CÓDIGO

A classe `Scanner` conta com uma série de métodos de alto nível. Utilizaremos o `nextLine()`, que nos permitirá acessar a próxima linha em nosso aquivo, assim, ela nos retornará uma `String`, que imprimiremos em seguida:

```
package br.com.alura.java.io.teste;

import java.util.Scanner;

public class TesteLeitura2; throws FileNotFoundException {

        public static void main(String[] args) {

                Scanner scanner = new Scanner(new File("contas.csv"));

                String linha = scanner.nextLine();
                System.out.println(linha);

        }
}
```

COPIAR CÓDIGO

Por fim, precisamos fechar o `scanner` :

```
package br.com.alura.java.io.teste;

import java.util.Scanner;

public class TesteLeitura2; throws FileNotFoundException {

        public static void main(String[] args) {

                Scanner scanner = new Scanner(new File("contas.csv"));

                String linha = scanner.nextLine();
                System.out.println(linha);

                scanner.close();

        }
}
```

COPIAR CÓDIGO

Executaremos a classe, e temos o seguinte resultado no console:

```
CC,22,33,Nico Steppat,210.1
```

COPIAR CÓDIGO

Funcionou, imprimimos a primeira linha com sucesso.

Nosso próximo passo será criar um laço, para imprimirmos as linhas sucessivamente. A `Scanner` conta com métodos próprios, dentre eles, um específico para saber se é possível obter uma próxima linha, chamado `hasNextLine()`. Seu retorno será do tipo `boolean`:

```
package br.com.alura.java.io.teste;

import java.util.Scanner;

public class TesteLeitura2; throws FileNotFoundException {

        public static void main(String[] args) {

                Scanner scanner = new Scanner(new File("contas.csv"));

                boolean tem = scanner.hasNextLine();
                System.out.println(tem);
                String linha = scanner.nextLine();
                System.out.println(linha);

                scanner.close();
```

```
        }
}
```

COPIAR CÓDIGO

Executaremos, e teremos o seguinte resultado no console:

```
true
22,33,Nico Steppat,210.1
```

COPIAR CÓDIGO

Isso significa duas coisas, primeiro, que nosso programa funcionou, já que o console exibiu a palavra `true`, e segundo, que há outras linhas a serem linhas, já que a resposta foi verdadeira.

Este método será utilizado, portanto, em nosso laço, para criar um mecanismo que funcione sempre que houver uma próxima linha a ser lida. O tipo de laço mais indicado, neste caso, é o `while`:

```
package br.com.alura.java.io.teste;

import java.util.Scanner;

public class TesteLeitura2; throws FileNotFoundException {

        public static void main(String[] args) {

                Scanner scanner = new Scanner(new File("contas.csv"));
                while(scanner.hasNextLine()) {
```

```
                String linha = scanner.nextLine();
                System.out.println(linha);
            }

            scanner.close();

        }
}
```

COPIAR CÓDIGO

Ao executar a classe, teremos o seguinte resultado no console:

```
CC,22,33,Nico Steppat,210.1
CP,11,55,Luan Silva,1300.98
CC,22,44,Ana Garcias,350.40
```

COPIAR CÓDIGO

Funcionou.

Em comparação com o código da classe `TesteLeitura`, este é mais sucinto, e eficaz da mesma forma.

Tudo foi facilitado graças à classe `Scanner`, mas ela possui ainda muitas funcionalidades que não exploramos. A seguir, veremos que, além de ler linha a linha, é possível lermos valor a valor, separadamente.

Transcrição

Nas aulas anteriores, conseguimos estabelecer um laço para a leitura de um arquivo, linha a linha. Nesta aula, aprenderemos a ler as informações contidas em cada uma destas linhas, individualmente.

Primeiro faremos isso sem utilizar a classe `Scanner`, em seguida, veremos como é o funcionamento com o uso desta.

Como sabemos, a `linha` é do tipo `String`, sendo assim, utilizaremos o método `split()` contido nesta classe, cuja função é separar uma `String` grande em pedaços menores.

O método `split()` recebe como parâmetro uma `String` do tipo `regex`, que é um conjunto de caracteres que define regras de como analisar, ou separar, uma `String` maior. É um tópico complexo, que inclusive possui um curso dedicado exclusivamente ao seu estudo.

No nosso caso, a regra de divisão é simplesmente a vírgula:

```
//Código omitido

public class TesteLeitura2 {
```

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

                Scanner scanner = new Scanner(new File("contas.csv"));
                while(scanner.hasNextLine()) {
                        String linha = scanner.nextLine();
                        System.out.println(linha);

                        linha.split(",");
                }
                scanner.close();
        }
}
```

COPIAR CÓDIGO

O retorno será uma `String` , com os valore contidos na linha:

```
//Código omitido

public class TesteLeitura2 {

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

                Scanner scanner = new Scanner(new File("contas.csv"));
                while(scanner.hasNextLine()) {
                        String linha = scanner.nextLine();
                        System.out.println(linha);
```

```
                String[] valores = linha.split(",");
            }
            scanner.close();
        }
}
```

COPIAR CÓDIGO

Imprimiremos, para checarmos se nosso programa funciona:

```
//Código omitido

public class TesteLeitura2 {

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

            Scanner scanner = new Scanner(new File("contas.csv"));
            while(scanner.hasNextLine()) {
                String linha = scanner.nextLine();
                System.out.println(linha);

                String[] valores = linha.split(",");
                System.out.println(valores);
            }
            scanner.close();
        }
}
```

COPIAR CÓDIGO

Executaremos, e temos o seguinte resultado no console:

```
CC,22,33,Nico Steppat,210.1
[Ljava.lang.String;@47f6473
CP,11,55,Luan Silva,1300.98
[Ljava.lang.String;@15975490
CC,22,44,Ana Garcias,350.40
[Ljava.langString;@6b143ee9
```

COPIAR CÓDIGO

Vemos que ele imprimiu a primeira linha, com as informações corretas, entretanto, após cada linha temos a impressão de uma saída que não faz muito sentido a primeira vista.

Para fazermos esta impressão sem a necessidade da criação de um laço, podemos utilizar a classe `Arrays`. Assim como temos a classe `Collections`, com uma porção de métodos auxiliares, temos a `Arrays`, que funciona da mesma forma, só que para o mundo dos arrays.

Desta classe, utilizaremos o método `toString()` e passaremos o array `valores` como parâmetro:

```
//Código omitido

public class TesteLeitura2 {

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

                Scanner scanner = new Scanner(new File("contas.csv"));
```

```
                while(scanner.hasNextLine()) {
                        String linha = scanner.nextLine();
                        System.out.println(linha);

                        String[] valores = linha.split(",");
                        System.out.println(Arrays.toString(valores));
                }
                scanner.close();
        }
}
```

COPIAR CÓDIGO

Executaremos e temos o seguinte resultado no console:

```
CC,22,33,Nico Steppat,210.1
[CC,22,33,Nico Steppat,210.1]
CP,11,55,Luan Silva,1300.98
[CP,11,55,Luan Silva,1300.98]
CC,22,44,Ana Garcias,350.40
[CC,22,44,Ana Garcias,350.40]
```

COPIAR CÓDIGO

Funcionou, temos todos os valores exibidos.

Podemos, inclusive, acessar uma posição específica no array, por exemplo `3` (o quarto elemento):

```
//Código omitido

public class TesteLeitura2 {

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

		Scanner scanner = new Scanner(new File("contas.csv"));
		while(scanner.hasNextLine()) {
			String linha = scanner.nextLine();
			System.out.println(linha);

			String[] valores = linha.split(",");
			System.out.println(valores[3]);
		}
		scanner.close();
	}
}
```

COPIAR CÓDIGO

Executando, temos o seguinte resultado no console:

```
CC,22,33,Nico Steppat,210.1
Nico Steppat
CP,11,55,Luan Silva,1300.98
Luan Silva
CC,22,44,Ana Garcias,350.40
Ana Garcias
```

COPIAR CÓDIGO

Conseguimos acessar diretamente o quarto elemento, ou seja, o titular da conta.

o `split()` nos ajuda, mas podemos fazer isso de forma ainda mais eficiente com o `Scanner`. Comentaremos as linhas de código com o método e a seguinte, em que imprimimos:

```
//Código omitido

public class TesteLeitura2 {

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

		Scanner scanner = new Scanner(new File("contas.csv"));
		while(scanner.hasNextLine()) {
			String linha = scanner.nextLine();
			System.out.println(linha);

//				String[] valores = linha.split(",");
//				System.out.println(valores[1]);
		}
		scanner.close();
	}
}
```

COPIAR CÓDIGO

Apesar de utilizarmos a mesma classe `Scanner`, não podemos utilizar o mesmo objeto `scanner`. Ele foi criado para ler linha a linha, nosso objetivo agora é setorizar cada uma destas linhas, e para isso deveremos criar um novo objeto.

Este novo `Scanner` se chamará `linhaScanner`, e receberá uma `String source`, ou seja, não é o nome do arquivo, mas sim a fonte que gostaríamos de analisar:

```
//Código omitido

public class TesteLeitura2 {

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

		Scanner scanner = new Scanner(new File("contas.csv"));
		while(scanner.hasNextLine()) {
			String linha = scanner.nextLine();
			System.out.println(linha);
			Scanner linhaScanner = new Scanner(linha);

//				String[] valores = linha.split(",");
//				System.out.println(valores[1]);
		}
		scanner.close();
	}
}
```

COPIAR CÓDIGO

Em seguida, precisamos indicar para o `linhaScanner` que a análise da linha deve ser feita respeitando determinando critério de separação das informações, que em nosso caso é a vírgula (,). Em programação, chamamos este critério de delimitador, mais especificamente `Delimiter` .

Sendo assim, utilizaremos o método `useDelimiter()` , que receberá nosso padrão, que como já foi discutido, é a vírgula (,):

```
//Código omitido

public class TesteLeitura2 {

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

                Scanner scanner = new Scanner(new File("contas.csv"));
                while(scanner.hasNextLine()) {
                        String linha = scanner.nextLine();
                        System.out.println(linha);

                        Scanner linhaScanner = new Scanner(linha);
                        linhaScanner.useDelimiter(",");

//                        String[] valores = linha.split(",");
//                        System.out.println(valores[1]);
                }
                scanner.close();
        }
}
```

COPIAR CÓDIGO

A seguir, utilizaremos o `Scanner` para que nos retorne o próximo item do arquivo. O método utilizado para este fim é o `next()`, ele nos retornará uma `String`. Faremos este processo por seis vezes, para englobarmos todas as informações dos clientes.

Imprimiremos estes valores e, por fim, fecharemos o `linhaScanner`:

```
//Código omitido

public class TesteLeitura2 {

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

                Scanner scanner = new Scanner(new File("contas.csv"));
                while(scanner.hasNextLine()) {
                        String linha = scanner.nextLine();
                        System.out.println(linha);

                        Scanner linhaScanner = new Scanner(linha);
                        linhaScanner.useDelimiter(",");

                        String valor1 = linhaScanner.next();
                        String valor2 = linhaScanner.next();
                        String valor3 = linhaScanner.next();
                        String valor4 = linhaScanner.next();
```

```
                        String valor5 = linhaScanner.next();
                        String valor6 = linhaScanner.next();

                        System.out.println(valor1 + valor2 + valor3 + valor4 + valor5 + v

                        linhaScanner.close();

//                          String[] valores = linha.split(",");
//                          System.out.println(valores[1]);
                }
                scanner.close();
        }
}
```

COPIAR CÓDIGO

Executamos e temos um erro, isso aconteceu pois na verdade só há cinco valores, assim, quando foi solicitado o próximo elemento da lista, este não existe e o programa apresentou o erro. Basta removermos o `valor6` :

```
//Código omitido

public class TesteLeitura2 {

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

                Scanner scanner = new Scanner(new File("contas.csv"));
                while(scanner.hasNextLine()) {
                        String linha = scanner.nextLine();
```

```
                        System.out.println(linha);

                        Scanner linhaScanner = new Scanner(linha);
                        linhaScanner.useDelimiter(",");

                        String valor1 = linhaScanner.next();
                        String valor2 = linhaScanner.next();
                        String valor3 = linhaScanner.next();
                        String valor4 = linhaScanner.next();
                        String valor5 = linhaScanner.next();

                        System.out.println(valor1 + valor2 + valor3 + valor4 + valor5);

                        linhaScanner.close();

//                          String[] valores = linha.split(",");
//                          System.out.println(valores[1]);
                }
                scanner.close();
        }
}
```

COPIAR CÓDIGO

Salvaremos e executaremos, temos o seguinte resultado:

```
CC,22,33,Nico Steppat,210.1
CC2233Nico Steppat210.1
CP,11,55,Luan Silva,1300.98
```

```
CP1155Luan Silva1300.98
CC,22,44,Ana Garcias,350.40
CC2244Ana Garcias350.40
```

COPIAR CÓDIGO

Funcionou. Temos a linha impressa diretamente do texto, e logo em seguida os valores extraídos, sem as vírgulas que os separam.

As informações de agência e conta são do tipo `int`, e como os transformamos em `Strings`, precisaremos realizar um `parseInt()` - transformando o `String` em um tipo mais específico, no caso, um inteiro.

Contudo, para evitar procedimentos complicados, podemos contar com o método `nextInt()`, para trabalhar com as informações de agência e número da conta, e `nextDouble()`, para o saldo. Assim já estamos preservando o tipo específico:

```
//Código omitido

public class TesteLeitura2 {

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

                Scanner scanner = new Scanner(new File("contas.csv"));
                while(scanner.hasNextLine()) {
                        String linha = scanner.nextLine();
                        System.out.println(linha);
```

```
                    Scanner linhaScanner = new Scanner(linha);
                    linhaScanner.useDelimiter(",");

                    String valor1 = linhaScanner.next();
                    int valor2 = linhaScanner.nextInt();
                    int valor3 = linhaScanner.nextInt();
                    String valor4 = linhaScanner.next();
                    double valor5 = linhaScanner.nextDouble();

                    System.out.println(valor1 + valor2 + valor3 + valor4 + valor5);

                    linhaScanner.close();

//                      String[] valores = linha.split(",");
//                      System.out.println(valores[1]);
                }
                scanner.close();
        }
}
```

COPIAR CÓDIGO

Ainda temos uma peculiaridade. No saldo, temos um ponto (.) separando os números inteiros dos decimais, contudo, alguns lugares convencionam o ponto, enquanto outros utilizam a vírgula (,) para este fim. O que determina se a máquina virtual seguirá um padrão ou outro é o sistema operacional da máquina, ela seguirá o padrão do idioma da máquina em que o código está sendo escrito.

Por exemplo, como neste curso estamos utilizando uma máquina cujo sistema está em Inglês, o padrão é o ponto (.), se tentarmos utilizar outro caractere, ocorrerá um erro.

Para evitar esta regra automática, podemos especificar no código a regra que queremos seguir utilizando o método `useLocale()` :

```
//Código omitido

public class TesteLeitura2 {

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

                Scanner scanner = new Scanner(new File("contas.csv"));
                while(scanner.hasNextLine()) {
                        String linha = scanner.nextLine();
                        System.out.println(linha);

                        Scanner linhaScanner = new Scanner(linha);
                        linhaScanner.useLocale(Locale);
                        linhaScanner.useDelimiter(",");

                        String valor1 = linhaScanner.next();
                        int valor2 = linhaScanner.nextInt();
                        int valor3 = linhaScanner.nextInt();
                        String valor4 = linhaScanner.next();
                        double valor5 = linhaScanner.nextDouble();
```

```
                        System.out.println(valor1 + valor2 + valor3 + valor4 + valor5);

                        linhaScanner.close();

//                        String[] valores = linha.split(",");
//                        System.out.println(valores[1]);
                }
                scanner.close();
        }
}
```

COPIAR CÓDIGO

Nela, acessaremos o nome da classe, portanto `Locale`, seguida de um ponto (`.`), e o Eclipse nos mostrará uma série de opções de regras. No caso, utilizaremos `US`, que é o padrão americano:

```
//Código omitido

public class TesteLeitura2 {

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

                Scanner scanner = new Scanner(new File("contas.csv"));
                while(scanner.hasNextLine()) {
                        String linha = scanner.nextLine();
                        System.out.println(linha);

                        Scanner linhaScanner = new Scanner(linha);
```

```
                        linhaScanner.useLocale(Locale.US);
                        linhaScanner.useDelimiter(",");

                        String valor1 = linhaScanner.next();
                        int valor2 = linhaScanner.nextInt();
                        int valor3 = linhaScanner.nextInt();
                        String valor4 = linhaScanner.next();
                        double valor5 = linhaScanner.nextDouble();

                        System.out.println(valor1 + valor2 + valor3 + valor4 + valor5);

                        linhaScanner.close();

//                          String[] valores = linha.split(",");
//                          System.out.println(valores[1]);
                }
                scanner.close();
        }
}
```

COPIAR CÓDIGO

Desta forma, não importa o sistema operacional, o código e a máquina virtual sempre respeitarão o padrão americano.

Salvaremos e executaremos, e o código continua funcionando perfeitamente.

Adiante, falaremos sobre a formatação.

Transcrição

Até o momento, utilizamos o `Scanner` para ler o arquivo, e para analisar a linha com o `Delimiter` - já utilizando os métodos específicos que fazem o `parseInt()`.

A seguir, falaremos sobre a formatação. Nosso foco não será no `Scanner`, tampouco no `java.io`.

Comentaremos o `System.out.println(linha)` para focarmos somente naquele que imprime os valores individualizados:

```
//Código omitido

public class TesteLeitura2 {

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

                Scanner scanner = new Scanner(new File("contas.csv"));
                while(scanner.hasNextLine()) {
                        String linha = scanner.nextLine();
//                        System.out.println(linha);
```

```
                    Scanner linhaScanner = new Scanner(linha);
                    linhaScanner.useLocale(Locale.US);
                    linhaScanner.useDelimiter(",");

                    String valor1 = linhaScanner.next();
                    int valor2 = linhaScanner.nextInt();
                    int valor3 = linhaScanner.nextInt();
                    String valor4 = linhaScanner.next();
                    double valor5 = linhaScanner.nextDouble();

                    System.out.println(valor1 + valor2 + valor3 + valor4 + valor5);

                    linhaScanner.close();

//                      String[] valores = linha.split(",");
//                      System.out.println(valores[3]);
                }
                scanner.close();
        }
}
```

COPIAR CÓDIGO

Se tentarmos separar cada um dos valores com vírgulas, nosso código ficará cada vez mais verboso, o que não é uma solução ideal. Para melhorarmos isso, podemos usar a formatação na classe `String`, com o método `format()`.

Este por sua vez recebe dois parâmetros, `format` e `args`. A ideia do primeiro é definir o formato, de forma genérica, e em seguida - no segundo - passamos os parâmetros, que no nosso caso são os valores que imprimimos acima.

Para estas regras de formatação, utilizamos apenas um `s` para representar uma `String`:

```
//Código omitido

public class TesteLeitura2 {

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

                Scanner scanner = new Scanner(new File("contas.csv"));
                while(scanner.hasNextLine()) {
                        String linha = scanner.nextLine();
//                        System.out.println(linha);

                        Scanner linhaScanner = new Scanner(linha);
                        linhaScanner.useLocale(Locale.US);
                        linhaScanner.useDelimiter(",");

                        String valor1 = linhaScanner.next();
                        int valor2 = linhaScanner.nextInt();
                        int valor3 = linhaScanner.nextInt();
                        String valor4 = linhaScanner.next();
                        double valor5 = linhaScanner.nextDouble();
```

```
                        String.format("", valor1, valor2, valor3, valor4, valor5);
                                                System.out.println(valor1 + valor2 + valo

                        linhaScanner.close();

//                          String[] valores = linha.split(",");
//                          System.out.println(valores[3]);
                }
                scanner.close();
        }
}
```

COPIAR CÓDIGO

Ele nos permite, inclusive, a criação de um novo valor no próprio método. Se quiséssemos, poderíamos ter inserido um novo elemento, desde que respeitada a regra de separação.

Para definir o formato, começamos com o símbolo de porcentagem (`%`), e indicamos o tipo do valor, `String` , que neste caso é representado apenas por um `s` . Precisamos repetir tantas vezes quanto houverem valores, assim, para cinco valores, serão cinco `%s` s:

```
//Código omitido

public class TesteLeitura2 {

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

```
                Scanner scanner = new Scanner(new File("contas.csv"));
                while(scanner.hasNextLine()) {
                        String linha = scanner.nextLine();
//                        System.out.println(linha);

                        Scanner linhaScanner = new Scanner(linha);
                        linhaScanner.useLocale(Locale.US);
                        linhaScanner.useDelimiter(",");

                        String valor1 = linhaScanner.next();
                        int valor2 = linhaScanner.nextInt();
                        int valor3 = linhaScanner.nextInt();
                        String valor4 = linhaScanner.next();
                        double valor5 = linhaScanner.nextDouble();

                        String.format("%s %s %s %s %s", valor1, valor2, valor3, valor4, v
                                                System.out.println(valor1 + valor2 + valo

                        linhaScanner.close();

//                        String[] valores = linha.split(",");
//                        System.out.println(valores[3]);
                }
                scanner.close();
        }
}
```

COPIAR CÓDIGO

Ou seja, concentramos a formatação em apenas uma `String`, sem a necessidade de fazer uma série de concatenações. A legibilidade do nosso código melhora consideravelmente desta forma - concatenar muitas `String`s é sempre má prática.

O resultado de `format` é uma nova `String`, que representa a o texto completo, já com a formatação. Ele será representado pela variável `valorFormatado`, que utilizaremos para imprimirmos:

```
//Código omitido

public class TesteLeitura2 {

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

                Scanner scanner = new Scanner(new File("contas.csv"));
                while(scanner.hasNextLine()) {
                        String linha = scanner.nextLine();
//                        System.out.println(linha);

                        Scanner linhaScanner = new Scanner(linha);
                        linhaScanner.useLocale(Locale.US);
                        linhaScanner.useDelimiter(",");

                        String valor1 = linhaScanner.next();
                        int valor2 = linhaScanner.nextInt();
                        int valor3 = linhaScanner.nextInt();
                        String valor4 = linhaScanner.next();
                        double valor5 = linhaScanner.nextDouble();
```

```
                        String valorFormatado = String.format("%s %s %s %s %s", valor1, v
                                                System.out.println(valorFormatado);

                        linhaScanner.close();

//                          String[] valores = linha.split(",");
//                          System.out.println(valores[3]);
                }
                scanner.close();
        }
}
```

COPIAR CÓDIGO

Executaremos, e temos o seguinte resultado no console:

```
CC 22 33 Nico Steppat 210.1
CP 11 55 Luan Silva 1300.98
CC 22 44 Ana Garcias 350.4
```

COPIAR CÓDIGO

Funcionou! Temos tudo separado por espaços.

Podemos separa-los por hifens (-), se quisermos, para melhorar a legibilidade, ou então uma vírgula (,), dois pontos (:), enfim, é possível formatarmos de maneira mais fácil:

```
//Código omitido

public class TesteLeitura2 {

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

                Scanner scanner = new Scanner(new File("contas.csv"));
                while(scanner.hasNextLine()) {
                        String linha = scanner.nextLine();
//                        System.out.println(linha);

                        Scanner linhaScanner = new Scanner(linha);
                        linhaScanner.useLocale(Locale.US);
                        linhaScanner.useDelimiter(",");

                        String valor1 = linhaScanner.next();
                        int valor2 = linhaScanner.nextInt();
                        int valor3 = linhaScanner.nextInt();
                        String valor4 = linhaScanner.next();
                        double valor5 = linhaScanner.nextDouble();

                        String valorFormatado = String.format("%s - %s-%s, %s: %s", valor
                                                System.out.println(valorFormatado);

                        linhaScanner.close();

//                        String[] valores = linha.split(",");
//                        System.out.println(valores[3]);
```

```
                }
                scanner.close();
        }
}
```

COPIAR CÓDIGO

Executando, temos o seguinte resultado no console:

```
CC - 22-33, Nico Steppat: 210.1
CP - 11-55, Luan Silva: 1300.98
CC - 22-44, Ana Garcias: 350.4
```

COPIAR CÓDIGO

E como saber que a `%s` significa `String`? Se estudarmos o Java, sabemos que esta informação pode ser encontrada na documentação. Portanto, não há necessidade de decorar as informações que serão passadas aqui, se surgir a necessidade de utilizar alguma ferramenta de formatação, basta olharmos a documentação para sabermos as abreviações aplicáveis.

Se pesquisarmos pelo termo "java printf" no Google (http://www.google.com), encontraremos links com a documentação disponível (https://docs.oracle.com/javase/tutorial/java/data/numberformat.html).

Na página, encontramos uma tabela contendo as abreviações disponíveis e suas respectivas correspondências. Veremos que há um tipo específico para um inteiro decimal, que é representado pela letra `d`, sendo assim, substituiremos o `s` pelo tipo mais específico, já que estamos trabalhando com numéricos nos valores 2 e 3:

```
//Código omitido

public class TesteLeitura2 {

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

                Scanner scanner = new Scanner(new File("contas.csv"));
                while(scanner.hasNextLine()) {
                        String linha = scanner.nextLine();
//                        System.out.println(linha);

                        Scanner linhaScanner = new Scanner(linha);
                        linhaScanner.useLocale(Locale.US);
                        linhaScanner.useDelimiter(",");

                        String valor1 = linhaScanner.next();
                        int valor2 = linhaScanner.nextInt();
                        int valor3 = linhaScanner.nextInt();
                        String valor4 = linhaScanner.next();
                        double valor5 = linhaScanner.nextDouble();

                        String valorFormatado = String.format("%s - %d-%d, %s: %s", valor
                                                System.out.println(valorFormatado);

                        linhaScanner.close();

//                        String[] valores = linha.split(",");
//                        System.out.println(valores[3]);
```

```
                }
                scanner.close();
        }
}
```

COPIAR CÓDIGO

Já o último valor é um `double`, sendo assim, utilizamos a abreviação `f`:

```
//Código omitido

public class TesteLeitura2 {

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

                Scanner scanner = new Scanner(new File("contas.csv"));
                while(scanner.hasNextLine()) {
                        String linha = scanner.nextLine();
//                        System.out.println(linha);

                        Scanner linhaScanner = new Scanner(linha);
                        linhaScanner.useLocale(Locale.US);
                        linhaScanner.useDelimiter(",");

                        String valor1 = linhaScanner.next();
                        int valor2 = linhaScanner.nextInt();
                        int valor3 = linhaScanner.nextInt();
                        String valor4 = linhaScanner.next();
```

```
                        double valor5 = linhaScanner.nextDouble();

                        String valorFormatado = String.format("%s - %d-%d, %s: %f", valor
                                                System.out.println(valorFormatado);

                        linhaScanner.close();

//                        String[] valores = linha.split(",");
//                        System.out.println(valores[3]);
                }
                scanner.close();
        }
}
```

COPIAR CÓDIGO

Em teoria, nada deveria mudar no resultado final visualmente. Executaremos, e percebemos que funcionou.

O `%d` funciona somente se o valor for um inteiro `int`, por exemplo, se utilizarmos esta formatação no `valor1`, que é uma `String`:

```
//Código omitido

public class TesteLeitura2 {

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

                Scanner scanner = new Scanner(new File("contas.csv"));
```

```
                while(scanner.hasNextLine()) {
                        String linha = scanner.nextLine();
//                        System.out.println(linha);

                        Scanner linhaScanner = new Scanner(linha);
                        linhaScanner.useLocale(Locale.US);
                        linhaScanner.useDelimiter(",");

                        String valor1 = linhaScanner.next();
                        int valor2 = linhaScanner.nextInt();
                        int valor3 = linhaScanner.nextInt();
                        String valor4 = linhaScanner.next();
                        double valor5 = linhaScanner.nextDouble();

                        String valorFormatado = String.format("%d - %d-%d, %s: %f", valor
                                                System.out.println(valorFormatado);

                        linhaScanner.close();

//                        String[] valores = linha.split(",");
//                        System.out.println(valores[3]);
                }
                scanner.close();
        }
}
```

COPIAR CÓDIGO

Temos um erro ao executarmos o programa, pois ele tentará interpretar o `String` como se fosse um `int`.

Assim, retornaremos à formatação correta, `%s`:

```
//Código omitido

public class TesteLeitura2 {

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

                Scanner scanner = new Scanner(new File("contas.csv"));
                while(scanner.hasNextLine()) {
                        String linha = scanner.nextLine();
//                        System.out.println(linha);

                        Scanner linhaScanner = new Scanner(linha);
                        linhaScanner.useLocale(Locale.US);
                        linhaScanner.useDelimiter(",");

                        String valor1 = linhaScanner.next();
                        int valor2 = linhaScanner.nextInt();
                        int valor3 = linhaScanner.nextInt();
                        String valor4 = linhaScanner.next();
                        double valor5 = linhaScanner.nextDouble();

                        String valorFormatado = String.format("%s - %d-%d, %s: %f", valor
                                                System.out.println(valorFormatado);

                        linhaScanner.close();
```

```
//                          String[] valores = linha.split(",");
//                          System.out.println(valores[3]);
                }
                scanner.close();
        }
}
```

COPIAR CÓDIGO

Na documentação, há outros exemplos de formatação. Vide o seguinte, em que temos um `long`:

```
//Código omitido

public class TestFormat {

        public static void main(String[] args) {
            long n = 461012;
            System.out.format("%d%n", n)  // --> "461012"
            System.out.format("%08d%n", n)  // --> "00461012"
//Código omitido
```

COPIAR CÓDIGO

Temos um tipo de abreviação que é o `%08`, no nosso caso, vamos testar como `%04`:

```
//Código omitido
```

```
public class TesteLeitura2 {

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

		Scanner scanner = new Scanner(new File("contas.csv"));
		while(scanner.hasNextLine()) {
			String linha = scanner.nextLine();
//			  System.out.println(linha);

			Scanner linhaScanner = new Scanner(linha);
			linhaScanner.useLocale(Locale.US);
			linhaScanner.useDelimiter(",");

			String valor1 = linhaScanner.next();
			int valor2 = linhaScanner.nextInt();
			int valor3 = linhaScanner.nextInt();
			String valor4 = linhaScanner.next();
			double valor5 = linhaScanner.nextDouble();

			String valorFormatado = String.format("%s - %04d-%d, %s: %f", val
					System.out.println(valorFormatado);

			linhaScanner.close();

//			  String[] valores = linha.split(",");
//			  System.out.println(valores[3]);
		}
		scanner.close();
```

```
        }
}
```

COPIAR CÓDIGO

Executaremos, e temos o seguinte resultado:

```
CC - 0022-33, Nico Steppat: 210.100000
CP - 0011-55, Luan Silva: 1300.980000
CC - 0022-44, Ana Garcias: 350.400000
```

COPIAR CÓDIGO

Notamos que o `valor2` foi formatado, e ganhou duas casas à esquerda, de modo que possui quatro caracteres no total. Portanto, se o número da agência fosse composto por quatro dígitos, não teríamos problema. A seguir, faremos o mesmo com o número da conta, só que conferiremos `%08` caracteres.

Para melhorar a didática de nosso código, daremos ao `valor1` o nome de `tipoConta`, ao `valor2` o nome de `agencia`, ao `valor3` o nome de `numero`, `valor4` será o `titular` e `valor5` o `saldo`:

```
//Código omitido

public class TesteLeitura2 {

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

                Scanner scanner = new Scanner(new File("contas.csv"));
```

```
                while(scanner.hasNextLine()) {
                        String linha = scanner.nextLine();
//                        System.out.println(linha);

                        Scanner linhaScanner = new Scanner(linha);
                        linhaScanner.useLocale(Locale.US);
                        linhaScanner.useDelimiter(",");

                        String tipoConta = linhaScanner.next();
                        int agencia = linhaScanner.nextInt();
                        int numero = linhaScanner.nextInt();
                        String titular = linhaScanner.next();
                        double saldo = linhaScanner.nextDouble();

                        String valorFormatado = String.format("%s - %04d-%08d, %s: %f", t
                                                System.out.println(valorFormatado);

                        linhaScanner.close();

//                        String[] valores = linha.split(",");
//                        System.out.println(valores[3]);
                }
                scanner.close();
        }
}
```

COPIAR CÓDIGO

Assim, executando, temos o seguinte resultado no console:

```
CC - 0022-00000033, Nico Steppat: 210.100000
CP - 0011-00000055, Luan Silva: 1300.980000
CC - 0022-00000044, Ana Garcias: 350.400000
```

COPIAR CÓDIGO

Funcionou, temos oito caracteres, em que os novos foram preenchidos com o número zero (`0`).

Em seguida, alteraremos a formatação do `saldo` , fazendo com que ganhe mais caracteres antes do ponto (`.`), bem como delimitando duas casas decimais após, fazemos isso utilizando o `010.2` , da seguinte forma:

```
//Código omitido

public class TesteLeitura2 {

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

                Scanner scanner = new Scanner(new File("contas.csv"));
                while(scanner.hasNextLine()) {
                        String linha = scanner.nextLine();
//                        System.out.println(linha);

                        Scanner linhaScanner = new Scanner(linha);
                        linhaScanner.useLocale(Locale.US);
                        linhaScanner.useDelimiter(",");

                        String tipoConta = linhaScanner.next();
```

```
                        int agencia = linhaScanner.nextInt();
                        int numero = linhaScanner.nextInt();
                        String titular = linhaScanner.next();
                        double saldo = linhaScanner.nextDouble();

                        String valorFormatado = String.format("%s - %04d-%08d, %s: %010.2
                                                System.out.println(valorFormatado);

                        linhaScanner.close();

//                          String[] valores = linha.split(",");
//                          System.out.println(valores[3]);
                }
                scanner.close();
        }
}
```

COPIAR CÓDIGO

Neste caso, `010` representa o número total de caracteres, sendo assim, ao executarmos teremos o seguinte resultado no console:

```
CC - 0022-00000033, Nico Steppat: 0000210.10
CP - 0011-00000055, Luan Silva: 0001300.98
CC - 0022-00000044, Ana Garcias: 0000350.40
```

COPIAR CÓDIGO

Podemos diminuir, e utilizar `08.2` para que não hajam tantos zeros. Da mesma forma, podemos limitar os caracteres em uma `String`, por exemplo, limitaremos o `titular` em `20` caracteres:

```
//Código omitido

public class TesteLeitura2 {

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

                Scanner scanner = new Scanner(new File("contas.csv"));
                while(scanner.hasNextLine()) {
                        String linha = scanner.nextLine();
//                        System.out.println(linha);

                        Scanner linhaScanner = new Scanner(linha);
                        linhaScanner.useLocale(Locale.US);
                        linhaScanner.useDelimiter(",");

                        String tipoConta = linhaScanner.next();
                        int agencia = linhaScanner.nextInt();
                        int numero = linhaScanner.nextInt();
                        String titular = linhaScanner.next();
                        double saldo = linhaScanner.nextDouble();

                        String valorFormatado = String.format("%s - %04d-%08d, %20s: %05.
                                                System.out.println(valorFormatado);

                        linhaScanner.close();
```

```
//                      String[] valores = linha.split(",");
//                      System.out.println(valores[3]);
                }
                scanner.close();
        }
}
```

COPIAR CÓDIGO

Executando, temos o seguinte resultado:

```
CC - 0022-00000033,         Nico Steppat: 00210.10
CP - 0011-00000055,         Luan Silva: 01300.98
CC - 0022-00000044,         Ana Garcias: 00350.40
```

COPIAR CÓDIGO

Por fim, alteraremos a formatação para que as casas não sejam divididas por um ponto (`.`). A formatação neste caso depende da região do mundo em que estamos, contudo, trabalharemos com a hipótese de transformarmos este ponto (`.`) em uma vírgula (`,`).

O método `format()` pode receber uma `String` , ou um `Locale` . Isso significa que em uma hipótese podemos informar como parâmetro em qual região do planeta estamos programando.

Portanto, utilizaremos a classe `Locale` . Já existe uma série de constantes para ela, mas como não há uma específica para o Brasil, utilizaremos `GERMANY` :

```
//Código omitido

public class TesteLeitura2 {

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

                Scanner scanner = new Scanner(new File("contas.csv"));
                while(scanner.hasNextLine()) {
                        String linha = scanner.nextLine();
//                        System.out.println(linha);

                        Scanner linhaScanner = new Scanner(linha);
                        linhaScanner.useLocale(Locale.US);
                        linhaScanner.useDelimiter(",");

                        String tipoConta = linhaScanner.next();
                        int agencia = linhaScanner.nextInt();
                        int numero = linhaScanner.nextInt();
                        String titular = linhaScanner.next();
                        double saldo = linhaScanner.nextDouble();

                        String valorFormatado = String.format(Locale.GERMANY , "%s - %04d

                                                System.out.println(valorFormatado);

                        linhaScanner.close();

//                        String[] valores = linha.split(",");
```

```
//                          System.out.println(valores[3]);
                }
                scanner.close();
        }
}
```

COPIAR CÓDIGO

Executando mais uma vez, nos é apresentado o seguinte:

```
CC - 0022-00000033,         Nico Steppat: 00210,10
CP - 0011-00000055,         Luan Silva: 01300,98
CC - 0022-00000044,         Ana Garcias: 00350,40
```

COPIAR CÓDIGO

Funcionou, temos as vírgulas.

Contudo, temos que considerar que estamos no Brasil. Apesar de não haver uma constante pré-existente, é possível criarmos nossa própria, como o construtor `new`. Ele receberá um parâmetro, que é o idioma, no caso `pt`:

```
//Código omitido

public class TesteLeitura2 {

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

```
                Scanner scanner = new Scanner(new File("contas.csv"));
                while(scanner.hasNextLine()) {
                        String linha = scanner.nextLine();
//                        System.out.println(linha);

                        Scanner linhaScanner = new Scanner(linha);
                        linhaScanner.useLocale(Locale.US);
                        linhaScanner.useDelimiter(",");

                        String tipoConta = linhaScanner.next();
                        int agencia = linhaScanner.nextInt();
                        int numero = linhaScanner.nextInt();
                        String titular = linhaScanner.next();
                        double saldo = linhaScanner.nextDouble();

                        String valorFormatado = String.format(new Locale("pt"), "%s - %04

                                                System.out.println(valorFormatado);

                        linhaScanner.close();

//                        String[] valores = linha.split(",");
//                        System.out.println(valores[3]);
                }
                scanner.close();
        }
}
```

COPIAR CÓDIGO

Executando, temos o mesmo resultado, portanto funcionou.

Para especificarmos que se trata de português do Brasil, e não de Portugal, colocamos o país de origem logo após o idioma. No caso, o Brasil é representado pelo código `BR`:

```
//Código omitido

public class TesteLeitura2 {

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

                Scanner scanner = new Scanner(new File("contas.csv"));
                while(scanner.hasNextLine()) {
                        String linha = scanner.nextLine();
//                        System.out.println(linha);

                        Scanner linhaScanner = new Scanner(linha);
                        linhaScanner.useLocale(Locale.US);
                        linhaScanner.useDelimiter(",");

                        String tipoConta = linhaScanner.next();
                        int agencia = linhaScanner.nextInt();
                        int numero = linhaScanner.nextInt();
                        String titular = linhaScanner.next();
                        double saldo = linhaScanner.nextDouble();
```

```
                        String valorFormatado = String.format(new Locale("pt", "BR"), "%s

                                                System.out.println(valorFormatado);

                        linhaScanner.close();

//                        String[] valores = linha.split(",");
//                        System.out.println(valores[3]);
                }
                scanner.close();
        }
}
```

COPIAR CÓDIGO

Para economizarmos linhas de código, podemos inserir a formatação direto no método de impressão:

```
//Código omitido

public class TesteLeitura2 {

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

                Scanner scanner = new Scanner(new File("contas.csv"));
                while(scanner.hasNextLine()) {
                        String linha = scanner.nextLine();
//                        System.out.println(linha);
```

```
                        Scanner linhaScanner = new Scanner(linha);
                        linhaScanner.useLocale(Locale.US);
                        linhaScanner.useDelimiter(",");

                        String tipoConta = linhaScanner.next();
                        int agencia = linhaScanner.nextInt();
                        int numero = linhaScanner.nextInt();
                        String titular = linhaScanner.next();
                        double saldo = linhaScanner.nextDouble();

                                                System.out.format(new Locale("pt", "BR"),


                        linhaScanner.close();

//                          String[] valores = linha.split(",");
//                          System.out.println(valores[3]);
                }
                scanner.close();
        }
}
```

COPIAR CÓDIGO

Ao executarmos, temos a seguinte impressão:

```
CC - 0022-00000033,        Nico Steppat: 00210,10 CP - 0011-00000055,        Luan Silva
```

COPIAR CÓDIGO

Isso acontece porque o `format()` não faz a quebra de linha automaticamente. Teremos de inseri-la manualmente, por meio do `%n` :

```
//Código omitido

public class TesteLeitura2 {

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

                Scanner scanner = new Scanner(new File("contas.csv"));
                while(scanner.hasNextLine()) {
                        String linha = scanner.nextLine();
//                        System.out.println(linha);

                        Scanner linhaScanner = new Scanner(linha);
                        linhaScanner.useLocale(Locale.US);
                        linhaScanner.useDelimiter(",");

                        String tipoConta = linhaScanner.next();
                        int agencia = linhaScanner.nextInt();
                        int numero = linhaScanner.nextInt();
                        String titular = linhaScanner.next();
                        double saldo = linhaScanner.nextDouble();
```

```
                                        System.out.format(new Locale("pt", "BR"),

                        linhaScanner.close();

//                          String[] valores = linha.split(",");
//                          System.out.println(valores[3]);
                }
                scanner.close();
        }
}
```

COPIAR CÓDIGO

Com isso nossa formatação já retorna ao modelo antigo, em três linhas separadas.

Adiante, daremos mais foco no `java.io` .

Para saber mais: java.util.Properties

Em projetos mais complexos (que você verá ainda em outros cursos) temos muitas configurações a fazer para nossa aplicação funcionar. Por exemplo, é preciso configurar usuários, senhas, endereços ou portas para acessar outros aplicativos e serviços. Um exemplo clássico é o acesso ao banco de dados, que precisa do login/senha, etc.

Essas configurações podem ficar dentro código fonte, mas isso exige a recompilação do código fonte assim que uma configuração muda. O melhor seria externalizá-las e colocá-las em um arquivo separado, por exemplo um arquivo de texto, que não exige de compilação. Dessa forma só precisamos alterar esse texto, sem mexer no código fonte Java.

Ótimo, e exatamente isso é feito em milhares de projetos Java e para facilitar e padronizar mais ainda, foi criado um mini-padrão para esse tipo de arquivo. Eles são chamados de arquivos de propriedade ou simplesmente ***properties***.

Um arquivo *properties* associa o nome da configuração com o seu valor. Veja o exemplo:

```
login = alura
senha = alurapass
endereco = www.alura.com.br
```

COPIAR CÓDIGO

61%

A configuração `login` tem o seu valor `alura`, `senha` tem o valor `alurapass` e assim em diante. Sempre tem uma **chave** (*key*) e um **valor** (*value*) associados. E como isso é tão comum, já foi criado a classe específica `java.util.Properties`, para trabalhar com esses pares de chave/valor, mas claro, poderíamos usar um *Scanner* também!

O uso dessa classe é muito simples. Veja a representação dos valores acima, através de um objeto da classe `Properties`:

```
//import deve ser java.util.Properties
Properties props = new Properties();
props.setProperty("login", "alura"); //chave, valor
props.setProperty("senha", "alurapass");
props.setProperty("endereco", "www.alura.com.br");
```

COPIAR CÓDIGO

Com o objeto criado podemos ver como escrever esses dados no HD.

Escrita de properties

Agora só falta gravar um arquivo para realmente externalizar as configurações. Para tal, você usa o método `store`, da classe `Properties`, que recebe um *stream* ou *writer*, além dos comentários desejados:

```
props.store(new FileWriter("conf.properties"), "algum come
```

COPIAR CÓDIGO

Isso cria um arquivo **conf.properties**, com os dados do objeto acima:

```
#algum comentário
#Thu May 10 14:29:38 BRT 2018
senha=alurapass
login=alura
endereco=www.alura.com.br
```

COPIAR CÓDIGO

Leitura de properties

Para ler esse arquivo de *properties*, basta usar o método `load`:

```
Properties props = new Properties();
props.load(new FileReader("conf.properties"));

String login = props.getProperty("login");
String senha = props.getProperty("senha");
String endereco = props.getProperty("endereco");

System.out.println(login + ", " + senha  + ", " +  enderec
```

COPIAR CÓDIGO

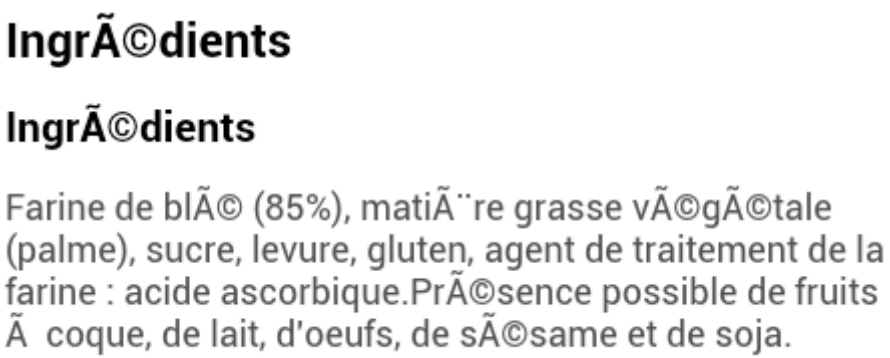
Repare que, uma vez lido o arquivo, podemos usar o método `getProperty(key)`, da classe `Properties`, para recuperar o seu valor.

Transcrição

*O novo arquivo **contas.csv** pode ser baixado aqui (https://s3.amazonaws.com/caelum-online-public/857-java-io/05/contas.csv). E caso queira, você pode fazer o download (https://s3.amazonaws.com/caelum-online-public/857-java-io/05/java7-aula5.zip) do projeto completo feito até a aula anterior.*

Olá! Bem-vindos novamente ao curso `java.io`.

É comum encontrarmos textos na internet onde há caracteres estranhos, que nos impedem de compreende-los por inteiro. É o caso da imagem a seguir:



Estes problemas estão relacionados com o pacote `java.io`, mas não são exclusivos à esta linguagem, acontecem em muitos outros casos.

Nesta aula, entenderemos o motivo pelo qual isto acontece, e o que podemos fazer para solucionarmos este tipo de erro, em Java. Isto é muito importante na vida prática de um programador.

Nas aulas anteriores, havíamos trabalhado com um arquivo no formato `.csv`, com as seguintes informações:

```
CC,22,33,Nico Steppat,232.9
CP,11,44,Paulo Silveira,2167.0
CC,22,11,Sérgio Lopes,2200.3
```

COPIAR CÓDIGO

A seguir, temos a apresentação binária deste arquivo:

```
01000011 01000011 00101100 00110010 00110010 00101100
00110011 00110011 00101100 01001110 01101001 01100011
01101111 00100000 01010011 01110100 01100101 01110000
01110000 01100001 01110100 11100011 11100011 00101100
00110010 00110000 00110000 00101110 00110000 00001010
```

COPIAR CÓDIGO

Onde a letra `C` é representada pela sequência binária `01000011`, e assim sucessivamente, cada caractere é convertido em uma sequência de `1` e `0`.

Há padrões de conversão de caracteres em binários, e um dos mais comuns está registrado na tabela ASCII (https://www.asciitable.com/) (*American Standard Code for Information Interchange*). Temos a letra, e a ela há um número associado, a partir deste, é criada uma sequência binária. Por exemplo, a letra C é representada pelo número 67, e resulta na sequência `01000011` .

Contudo, temos um problema pois esta tabela engloba todos os caracteres da língua inglesa, mas como sabemos há outros além destes. Por exemplo, aqueles que são acentuados. Há ainda outros alfabetos, com símbolos diferentes.

Para solucionar este problema, foram criadas as *codepages* (http://www.ascii.ca/ascii_standard.htm), único formato capaz de englobar a quantidade de informação correspondente ao número de línguas e caracteres existentes.

Temos por exemplo a página (http://www.ascii.ca/iso8859.1.htm) com a apresentação dos caracteres latinos, que contém as vogais com todos os tipos de acentuação.

Isto é útil quando estamos pensando em um sistema contido, entretanto, ao trabalharmos em rede, pode haver complicações, como a da página que vimos no começo desta aula.

Para tentar unificar os padrões, e minimizar este tipo de problemas, foi criado o ***unicode***. Trata-se de uma tabela (https://en.wikipedia.org/wiki/List_of_Unicode_characters) cujo objetivo é de apresentar todos os caracteres existentes no mundo.

Ela também conta com um número associado a cada caractere.

Contudo, o *unicode* não define a forma como as informações devem ser armazenadas no HD, isto é tarefa dos ***encodings***. É o caso dos "UTFs", como o `UTF-8` e `UTF-16`, esta sigla significa "*Unicode Transformation Format*. Ela está vinculada desde o nascimento com a tabela de Unicode, para traduzir os *codepoints* para um formato binário.

Além do `UTF` há outros exemplos de *Encodings*, como o `ASCII` e o `Windows 1252`. Adiante, veremos como eles podem variar de acordo com o sistema operacional.

Para saber mais: FAQ do UNICODE

Conheça o site do Unicode Consortium (https://www.unicode.org/), instituição que mantém o Unicode e leia as perguntas básicas (https://www.unicode.org/faq/basic_q.html) relacionadas (FAQ em inglês).

Outro link interessante é contem a tabela completa (https://unicode-table.com/pt/) dos caracteres unicode.

Transcrição

Como havíamos falado anteriormente, veremos o funcionamento do Java em diferentes sistemas operacionais. Começaremos pelo Windows. Já temos todas as instalações necessárias e importamos o projeto com o qual trabalhávamos.

Na janela "Package Explorer", localizada na lateral esquerda da tela, clicaremos com o botão direito do mouse, e selecionaremos a opção "Import...> General > Existing Projects into Workspace".

Surgirá uma caixa de diálogo, com a opção "Select archive file", escolheremos o [arquivo (https://s3.amazonaws.com/caelum-online-public/857-java-io/05/java7-aula5.zip)](https://s3.amazonaws.com/caelum-online-public/857-java-io/05/java7-aula5.zip) `java7-aula5.zip` .

Pode ser que apareça uma exclamação vermelha no ícone do arquivo, isso acontece pois é comum haver diferenças entre as configurações de um computador para o outro.

Clicaremos com o botão direito do mouse sobre o nome do arquivo `java-io` , selecionaremos a opção "Buil Path > Configure Build Path...". Surgirá uma caixa de diálogo, onde temos uma aba chamada "Libraries" com uma pasta `Classpath` .

Clicaremos sobre a pasta e temos uma subpasta, com o nome `JRE System Library [Java SE 10.0.0] (unbound)`. Removeremos este último item, e adicionaremos a JRE instalada em nosso sistema.

Na lateral direita, selecionaremos a opção "Add Library", em seguida, "JRE System Library". Clicaremos em "Next", e selecionaremos "Workspace default JRE (jre-10.0.1)". Para concluir, clicaremos em "Finish" e "Apply".

Com isso, temos todos os arquivos com os quais estávamos trabalhando no outro computador.

Na pasta `br.com.alura.java.io.teste` criaremos uma nova classe, chamada `TesteUnicodeEEncoding`, com um método `main` já estabelecido. Nela teremos uma `String`, `s`, que receberá a letra `"C"`:

```
package br.com.alura.java.io.teste;

public class TesteUnicodeEEncoding {

	public static void main(String[] args) {

		String s = "C";

	}
}
```

COPIAR CÓDIGO

Em seguida, faremos um `System.out.println()` para descobrirmos qual é o *codepoint*, que é o número associado na tabela de *unicodes*. Ao digitarmos, ele aparece como `codePointAt()` pois o `String` pode ser uma série de caracteres:

```
//Código omitido

public class TesteUnicodeEEncoding {

        public static void main(String[] args) {

                String s = "CCCC";

                System.out.println(s.codePointAt(arg0));

        }
}
```

COPIAR CÓDIGO

Precisamos especificar a posição exata, que no caso será `0`. E manteremos o caractere `"C"`:

```
//Código omitido

public class TesteUnicodeEEncoding {

        public static void main(String[] args) {

                String s = "C";

                System.out.println(s.codePointAt(0));
```

```
        }
}
```

COPIAR CÓDIGO

Executando o código, temos o seguinte resultado no console:

```
67
```

COPIAR CÓDIGO

Correspondente ao número que vimos na tabela ASCII, e que também vale para a tabela *unicode*. A diferença é que o unicode é apenas o mapeamento do caractere, e não nos fornece a informação de como armazenar este, isso é o trabalho do *encoding*.

Sendo assim, temos que descobrir qual *encoding* é utilizado por padrão pelo Java. A primeira coisa que precisamos considerar é que isso varia de acordo com o sistema operacional, o Java se adapta de acordo com o SO.

Como estamos trabalhando com o Windows, para descobrirmos, utilizamos uma classe que representa o *encoding*, chamada de `Charset`, que em uma tradução simplificada pode ser compreendida como um conjunto de caracteres. Esta por sua vez contém uma série de métodos estáticos, dentre eles o `defaultCharset()`, que nos interessa, quando estamos falando em descobrir qual é o *encoding* padrão.

Nosso retorno será, também, um `Charset`:

```
//Código omitido

public class TesteUnicodeEEncoding {

        public static void main(String[] args) {

                String s = "C";
                System.out.println(s.codePointAt(0));

                Charset charset = Charset.defaultCharset();
        }
}
```

COPIAR CÓDIGO

Por meio do `Charset`, podemos utilizar um novo método que imprima o seu nome, que é o `charset.displayName()`:

```
//Código omitido

public class TesteUnicodeEEncoding {

        public static void main(String[] args) {

                String s = "C";
                System.out.println(s.codePointAt(0));

                Charset charset = Charset.defaultCharset();
```

```
                System.out.println(charset.displayName());
        }
}
```

COPIAR CÓDIGO

Executamos. No console, é exibido o seguinte:

```
67
windows-1252
```

COPIAR CÓDIGO

Funcionou. Desta forma, descobrimos que o *encoding* utilizado é o `windows-1252`.

É importante termos esta informação pois, é este `charset` que define como traduzir o *codepoint* em uma sequência de bits e bytes.

A partir da `String s` é possível utilizarmos um `get` - padrão - para obtermos os bytes:

```
//Código omitido

public class TesteUnicodeEEncoding {

        public static void main(String[] args) {

                String s = "C";
                System.out.println(s.codePointAt(0));
```

```
                Charset charset = Charset.defaultCharset();
                System.out.println(charset.displayName());

                s.getBytes();
        }
}
```

COPIAR CÓDIGO

Com o cursor sobre o método `get`, utilizaremos o atalho "Ctrl + 1" para criarmos uma variável local, da seguinte forma:

```
//Código omitido

public class TesteUnicodeEEncoding {

        public static void main(String[] args) {

                String s = "C";
                System.out.println(s.codePointAt(0));

                Charset charset = Charset.defaultCharset();
                System.out.println(charset.displayName());

                byte[] bytes = s.getBytes();
        }
}
```

COPIAR CÓDIGO

Será utilizado o *encoding* padrão, que como vimos, é o `windows-1252`, para a criação dos bytes. Para verificarmos se tudo está funcionando, faremos a impressão do tamanho do array, utilizando o `bytes.length`, e concatenaremos com `charset` utilizado, no caso o `windows-1252`:

```
//Código omitido

public class TesteUnicodeEEncoding {

        public static void main(String[] args) {

                String s = "C";
                System.out.println(s.codePointAt(0));

                Charset charset = Charset.defaultCharset();
                System.out.println(charset.displayName());

                byte[] bytes = s.getBytes();
                System.out.println(bytes.length + ", windows-1252");
        }
}
```

COPIAR CÓDIGO

Executando, temos o seguinte resultado no console:

```
67
windows-1252
1, windows-1252
```

COPIAR CÓDIGO

Além disso, é possível também definirmos o `charset` que gostaríamos de utilizar, independentemente do padrão. Para isso, utilizamos o método `getBytes()` cujo parâmetro recebe uma `String` referente ao `charsetName`, ou seja, o nome do `charset` - definiremos como a `String` `windows-1252`.

Precisamos estabelecer o `throws` com a exceção `UnsupportedEncodingException` para nos assegurarmos de que o `charset` existe:

```
//Código omitido

public class TesteUnicodeEEncoding {

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

                String s = "C";
                System.out.println(s.codePointAt(0));

                Charset charset = Charset.defaultCharset();
                System.out.println(charset.displayName());

                byte[] bytes = s.getBytes("windows-1252");
                System.out.println(bytes.length + ", windows-1252");
```

```
        }
}
```

COPIAR CÓDIGO

Testando novamente, temos o mesmo resultado no console:

```
67
windows-1252
1, windows-1252
```

COPIAR CÓDIGO

Em seguida, utilizando o mesmo código como base, testaremos um novo `charset`, no caso, o `UTF-16`:

```
//Código omitido

public class TesteUnicodeEEncoding {

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

                String s = "C";
                System.out.println(s.codePointAt(0));

                Charset charset = Charset.defaultCharset();
                System.out.println(charset.displayName());

                byte[] bytes = s.getBytes("windows-1252");
```

```
                System.out.println(bytes.length + ", windows-1252");

                bytes = s.getBytes("UTF-16");
                System.out.println(bytes.length + ", UTF-16");
        }
}
```

COPIAR CÓDIGO

Executaremos, e temos o seguinte resultado no console:

```
67
windows-1252
1, windows-1252
4, UTF-16
```

COPIAR CÓDIGO

Podemos testar com o `UTF-8`, e veremos que o tamanho é de 1 byte. Manteremos em `UTF-16`.

Veremos uma nova forma de acessar um `Charset`. Isto pode ser feito por intermédio de uma classe chamada `StandardCharsets`, do pacote `java.nio.charset`, onde `nio` significa novo `io`, novas formas de input e output. Nesta classe há algumas constantes, utilizaremos a `US_ASCII`, e imprimiremos a mesma:

```
//Código omitido

public class TesteUnicodeEEncoding {
```

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

                String s = "C";
                System.out.println(s.codePointAt(0));

                Charset charset = Charset.defaultCharset();
                System.out.println(charset.displayName());

                byte[] bytes = s.getBytes("windows-1252");
                System.out.println(bytes.length + ", windows-1252");

                bytes = s.getBytes("UTF-16");
                System.out.println(bytes.length + ", UTF-16");

                bytes = s.getBytes(StandardCharsets.US_ASCII);
                System.out.println(bytes.length + ", US-ASCII");

        }
}
```

COPIAR CÓDIGO

Executando, temos o seguinte resultado:

```
67
windows-1252
1, windows-1252
4, UTF-16
1, US-ASCII
```

COPIAR CÓDIGO

A seguir, veremos como transformar a representação binária em uma `String`, para uma destas `Charsets`. Para isso, utilizaremos o construtor da classe `String`.

Sendo assim, teremos `new String()`, e com o atalho "Ctrl + Barra de Espaço" nos será apresentada uma lista de diferentes tipos de parâmetros possíveis:

```
//Código omitido

public class TesteUnicodeEEncoding {

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

                String s = "C";
                System.out.println(s.codePointAt(0));

                Charset charset = Charset.defaultCharset();
                System.out.println(charset.displayName());

                byte[] bytes = s.getBytes("windows-1252");
                System.out.println(bytes.length + ", windows-1252");
                new String();

                bytes = s.getBytes("UTF-16");
                System.out.println(bytes.length + ", UTF-16");
```

```
                bytes = s.getBytes(StandardCharsets.US_ASCII);
                System.out.println(bytes.length + ", US-ASCII");

        }
}
```

COPIAR CÓDIGO

Utilizaremos a que recebe bytes, e nos retorna uma nova `String`. Para visualiza-la, a imprimiremos:

```
//Código omitido

public class TesteUnicodeEEncoding {

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

                String s = "C";
                System.out.println(s.codePointAt(0));

                Charset charset = Charset.defaultCharset();
                System.out.println(charset.displayName());

                byte[] bytes = s.getBytes("windows-1252");
                System.out.println(bytes.length + ", windows-1252");
                String sNovo = new String(bytes);
                System.out.println(sNovo);

                bytes = s.getBytes("UTF-16");
```

```
                System.out.println(bytes.length + ", UTF-16");

                bytes = s.getBytes(StandardCharsets.US_ASCII);
                System.out.println(bytes.length + ", US-ASCII");

        }
}
```

COPIAR CÓDIGO

Executaremos, e temos o seguinte resultado no console:

```
67
windows-1252
1, windows-1252
C
4, UTF-16
1, US-ASCII
```

COPIAR CÓDIGO

Vemos que a letra "C" foi imprimida, portanto, funcionou. Mas qual foi o `Charset` utilizado ao transformar para a classe `String` ? Justamente o padrão, ou seja, `windows-1252` .

Contudo, assim como fizemos anteriormente, também é possível definir o `Charset` explicitamente. Para fazer isto, basta separarmos por vírgulas e colocarmos o nome do `Charset` entre aspas (`""`), da seguinte forma:

```
//Código omitido

public class TesteUnicodeEEncoding {

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

                String s = "C";
                System.out.println(s.codePointAt(0));

                Charset charset = Charset.defaultCharset();
                System.out.println(charset.displayName());

                byte[] bytes = s.getBytes("windows-1252");
                System.out.println(bytes.length + ", windows-1252");
                String sNovo = new String(bytes, "windows-1252");
                System.out.println(sNovo);

                bytes = s.getBytes("UTF-16");
                System.out.println(bytes.length + ", UTF-16");

                bytes = s.getBytes(StandardCharsets.US_ASCII);
                System.out.println(bytes.length + ", US-ASCII");

        }
}
```

COPIAR CÓDIGO

Como já estávamos usando este `Charset`, o resultado no console será o mesmo. Faremos o mesmo processo com os demais `Charset`s:

```
//Código omitido

public class TesteUnicodeEEncoding {

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

                String s = "C";
                System.out.println(s.codePointAt(0));

                Charset charset = Charset.defaultCharset();
                System.out.println(charset.displayName());

                byte[] bytes = s.getBytes("windows-1252");
                System.out.println(bytes.length + ", windows-1252");
                String sNovo = new String(bytes, "windows-1252");
                System.out.println(sNovo);

                bytes = s.getBytes("UTF-16");
                System.out.println(bytes.length + ", UTF-16");
                sNovo = new String(bytes, "windows-1252");
                System.out.println(sNovo);

                bytes = s.getBytes(StandardCharsets.US_ASCII);
                System.out.println(bytes.length + ", US-ASCII");
                sNovo = new String(bytes, "windows-1252");
```

```
                System.out.println(sNovo);

        }
}
```

COPIAR CÓDIGO

Desta forma estamos criando um problema, pois primeiro temos um `UTF-16`, ou `US_ASCII`, e abaixo estamos forçando o `windows-1252`. Testaremos imprimir desta forma, tirando a quebra de linha nestas duas ocasiões:

```
//Código omitido

public class TesteUnicodeEEncoding {

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

                String s = "C";
                System.out.println(s.codePointAt(0));

                Charset charset = Charset.defaultCharset();
                System.out.println(charset.displayName());

                byte[] bytes = s.getBytes("windows-1252");
                System.out.print(bytes.length + ", windows-1252, ");
                String sNovo = new String(bytes, "windows-1252");
                System.out.println(sNovo);

                bytes = s.getBytes("UTF-16");
```

```
                System.out.print(bytes.length + ", UTF-16, ");
                sNovo = new String(bytes, "windows-1252");
                System.out.println(sNovo);

                bytes = s.getBytes(StandardCharsets.US_ASCII);
                System.out.print(bytes.length + ", US-ASCII, ");
                sNovo = new String(bytes, "windows-1252");
                System.out.println(sNovo);
                .
        }
}
```

COPIAR CÓDIGO

Executando, temos o seguinte resultado:

```
67
windows-1252
1, windows-1252, C
4, UTF-16,þÿ C
1, US-ASCII, C
```

COPIAR CÓDIGO

Notamos que o *codepoint* não muda, continua `67`. No primeiro exemplo, como não alteramos o `Charset`, temos o `windows-1252`, já no exemplo do `UTF-16` tivemos um problema, apareceram caracteres especiais `þÿ`. Isso aconteceu porque ele recebeu 4 bytes, mas o `windows-1252` tem apenas 1 byte. No `US_ASCII` não tivemos este problema, porque eles são iguais nos caracteres comuns do alfabeto.

Isso pode ser observado de forma prática facilmente. Trocaremo o"C" por "ç":

```
//Código omitido

public class TesteUnicodeEEncoding {

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

                String s = "ç";
                System.out.println(s.codePointAt(0));

                Charset charset = Charset.defaultCharset();
                System.out.println(charset.displayName());

                byte[] bytes = s.getBytes("windows-1252");
                System.out.print(bytes.length + ", windows-1252, ");
                String sNovo = new String(bytes, "windows-1252");
                System.out.println(sNovo);

                bytes = s.getBytes("UTF-16");
                System.out.print(bytes.length + ", UTF-16, ");
                sNovo = new String(bytes, "windows-1252");
                System.out.println(sNovo);

                bytes = s.getBytes(StandardCharsets.US_ASCII);
                System.out.print(bytes.length + ", US-ASCII, ");
                sNovo = new String(bytes, "windows-1252");
                System.out.println(sNovo);
```

```
                .
        }
}
```

COPIAR CÓDIGO

Executando, temos o seguinte retorno:

```
231
windows-1252
1, windows-1252, ç
4, UTF-16,þÿ ç
1, US-ASCII, ?
```

COPIAR CÓDIGO

Tivemos um problema de impressão com o `US_ASCII` , pois ele não define esse tipo de caractere.

Isto é um problema comum, o recebimento de informação em bits e bytes, que acreditamos estar codificado em algo, mas que na realidade não está.

Para que nosso código funcione, precisamos indicar o *encoding* correto, para cada um dos casos respectivamente:

```
//Código omitido

public class TesteUnicodeEEncoding {
```

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

                String s = "ç";
                System.out.println(s.codePointAt(0));

                Charset charset = Charset.defaultCharset();
                System.out.println(charset.displayName());

                byte[] bytes = s.getBytes("windows-1252");
                System.out.print(bytes.length + ", windows-1252, ");
                String sNovo = new String(bytes, "windows-1252");
                System.out.println(sNovo);

                bytes = s.getBytes("UTF-16");
                System.out.print(bytes.length + ", UTF-16, ");
                sNovo = new String(bytes, "UTF-16");
                System.out.println(sNovo);

                bytes = s.getBytes(StandardCharsets.US_ASCII);
                System.out.print(bytes.length + ", US-ASCII, ");
                sNovo = new String(bytes, "US-ASCII");
                System.out.println(sNovo);
                .
        }
}
```

COPIAR CÓDIGO

Executaremos novamente e temos o seguinte resultado:

```
231
windows-1252
1, windows-1252, ç
4, UTF-16, ç
1, US-ASCII, ?
```

COPIAR CÓDIGO

O `ASCII` continua apresentando erro, pois a `ç` não está presenta na tabela padrão.

Vimos o *codepoint*, e como ele é imutável, e vimos também os diferentes tipos de *encoding*. Adiante, veremos como aplicar este conhecimento na leitura e escrita de arquivos, o que acontece na prática no celular ou mesmo no navegador. Até lá!

Transcrição

Já vimos como funciona o *encoding* no Java. Aprendemos que cada caractere dentro de uma `String` possui um *codepoint* associado, registrado na tabela *unicode*, e vimos também que há um `Charset` padrão, que é aplicado conforme o sistema operacional que está sendo utilizado. No Windows OS, por exemplo, este padrão é `windows-1252`.

O codepoint de um caractere nunca muda, é sempre o mesmo. O codepoint de `"ç"` é `231`, e sempre será.

Fizemos testes com os bytes, vimos qual a sua quantidade, e aprendemos a tranformá-los em `Strings`.

A seguir, trabalharemos com o foco nos arquivos. Abriremos o `contas.csv`, clicando com o botão direito do mouse, e selecionando a opção "Text Editor". Vemos o seguinte:

```
CC,22,33,Nico Steppat,210.1
CP,11,55,Luan Silva,1300.98
CC,22,44,SÃ©rgio  Lopes,350.40
```

COPIAR CÓDIGO

Temos um problema de encoding, pois o nome "Sérgio" tem um acento agudo na letra "e". Isso acontece pois o encoding utilizado por padrão no Windows não tem o registro deste caractere `é`.

Clicaremos com o botão direito do mouse, sobre o nome do arquivo `contas.csv` e selecionaremos a opção "Properties". Podemos observar nas propriedades que foi aplicado o encoding `windows-1252`. Na própria caixa de diálogo é possível alterar isso, clicando na opção "Other", selecionaremos `UTF-8`, ao fazermos isso temos o seguinte resultado:

```
CC,22,33,Nico Steppat,210.1
CP,11,55,Luan Silva,1300.98
CC,22,44,Sérgio  Lopes,350.40
```

COPIAR CÓDIGO

Problemas de encoding em decorrência do uso de um `Charset` errado não são incomuns. Estes problemas não são exclusivos do mundo Java.

Em seguida, abriremos a classe `TesteLeitura2`. Nela, temos um `Scanner()` que lê o arquivo `contas.csv`. Ao fazer isso, ele precisa determinar qual encoding será utilizado. Por padrão, é aplicado o do sistema operacional, que neste caso é o `windows-1252`. Já dentro do laço, temos um outro `Scanner`, que separa as informações contidas em cada uma das linhas:

```
//Código omitido

public class TesteLeitura2 {
```

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

    Scanner scanner = new Scanner(new File("contas.csv"));
    while(scanner.hasNextLine()) {
            String linha = scanner.nextLine();
            //System.out.println(linha);

            Scanner linhaScanner = new Scanner(linha);
            linhaScanner.useLocale(Locale.US);
            linhaScanner.useDelimiter(",");

            String tipoConta = linhaScanner.next();
            int agencia = linhaScanner.nextInt();
            int numero = linhaScanner.nextInt();
            String titular = linhaScanner.next();
            double saldo = linhaScanner.nextDouble();

            System.out.format(new Locale("pt","BR"), %s - %04d0%08d, %20s: %08.2f %n"
                    tipoConta, agencia, numero, titular, saldo );

            linhaScanner.close();

            //Código omitido
```

COPIAR CÓDIGO

Aplicando o `Locale` , se necessário, para fazer o `parseInt()` dos dados.

Executaremos, e temos o seguinte resultado no console:

```
CC - 0022-00000033,      Nico Steppat: 00210,10
CP - 0011-00000055,        Luan Silva: 01300,98
CC - 0022-00000044,     SÃ©rgio Lopes: 00350,40
```

COPIAR CÓDIGO

A execução funcionou, contudo, temos um problema no encoding, com o caractere especial `é`, no nome "Sérgio". Isso acontece porque o `contas.csv` é `UTF-8`, enquanto o Java utiliza por padrão, do Windows OS, o `windows-1252`. Sendo assim, precisamos especificar o `UTF-8` para o Java.

O responsável por ler o arquivo é o `Scanner`, então é nele que deve ser inserida esta especificação. Selecionaremos o construtor que recebe a `File source`, bem como o `charsetName` formato `String`. Podemos então escrever isso no código:

```
//Código omitido

public class TesteLeitura2 {

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

        Scanner scanner = new Scanner(new File("contas.csv"), "UTF-8");
        while(scanner.hasNextLine()) {
                String linha = scanner.nextLine();
                //System.out.println(linha);
```

```
                Scanner linhaScanner = new Scanner(linha);
                linhaScanner.useLocale(Locale.US);
                linhaScanner.useDelimiter(",");

                String tipoConta = linhaScanner.next();
                int agencia = linhaScanner.nextInt();
                int numero = linhaScanner.nextInt();
                String titular = linhaScanner.next();
                double saldo = linhaScanner.nextDouble();

                System.out.format(new Locale("pt","BR"), %s - %04d0%08d, %20s: %08.2f %n"
                        tipoConta, agencia, numero, titular, saldo );

                linhaScanner.close();

                //Código omitido
```

COPIAR CÓDIGO

Executando novamente, temos o seguinte resultado:

```
CC - 0022-00000033,      Nico Steppat: 00210,10
CP - 0011-00000055,        Luan Silva: 01300,98
CC - 0022-00000044,      Sérgio Lopes: 00350,40
```

COPIAR CÓDIGO

Temos o programa funcionando, com todos os nomes impressos corretamente.

Ao abrirmos um arquivo temos um fluxo de entrada de bits e bytes, que é transformado em um texto, neste momento é aplicado um `Charset`. Nos é dada a possibilidade de defini-lo, de modo a melhor servir ao nosso programa.

Veremos como isso pode funcionar com outra classe, no caso `TesteLeitura`

Testaremos isso em outra classe, desta vez, `TesteLeitura`. Trabalhávamos com o arquivo `lorem.txt`, onde não há nenhum caractere especial, ou seja, com acentuação, mas que de qualquer forma permite a definição do `Charset`.

A definição do `Charset` ocorre somente a partir do `InputStreamReader`, pois é ele quem transforma o fluxo de bytes em caracteres, utilizando justamente o `Charset`.

No construtor `InputStreamReader()` temos a possibilidade de passar tanto um `Charset`, quando um `charsetNamet` do tipo `String`. No caso, utilizaremos a segunda opção. Definiremos como `UTF-8`, já que é o padrão do arquivo:

> Caso o arquivo não esteja neste formato, é possível altera-lo, clicando com o botão direito do mouse sobre seu nome, e selecionando a opção "Properties". Em "Text file encoding", selecionaremos "Other > UTF-8".

```
//Código omitido

public class TesteLeitura {

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

```
                //Fluxo de Entrada com Arquivo
                InputStream fis = new FileInputStream("lorem.txt");
                Reader isr = new InputStreamReader(fis, "UTF-8");
                BufferedReader br = new BufferedReader(isr);

                String linha = br.readLine();

                while(linha != null) {

                        System.out.println(linha);
                        linha = br.readLine();
                }

                br.close();

        }
}
```

COPIAR CÓDIGO

Executando, temos o texto completo do arquivo `lorem.txt` no console, indicando que tudo está funcionando corretamente.

Ainda que nesse caso estivéssemos trabalhando apenas com caracteres padrão, este exemplo serve para aprendemos que é possível impormos um `Charset` via o `InputStreamReader()`.

O mesmo vale para a escrita. Na classe `TesteEscritaPrintStreamWriter`, utilizaremos o construtor `PrintWriter()` para este fim, que recebe um `File` e um `Charset csn`. O `csn` neste caso significa o nome de qualquer `Charset` suportado pelo Java:

```
//Código omitido
public class TesteEscritaPrintStreamWriter {
        public static void main(String[] args) throws IOException {
        //Código omitido
        PrintWriter ps = new PrintWriter("lorem2.txt", "UTF-8");
        //Código omitido
        }
}
```

COPIAR CÓDIGO

Executando, abriremos o arquivo `lorem2.txt` e veremos que o texto foi gravado, mas ainda com um caractere não reconhecido:

```
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod
```

```
asfasdfsafdas dfs sdf asf assdfÃfÄ.
```

COPIAR CÓDIGO

Isso acontece porque o Eclipse ainda acha que o padrão do arquivo é o `windows-1252`. Para solucionar, clicaremos sobre o nome do arquivo com o botão direito do mouse, e selecionaremos a opção "Properties", na seção "Text file encoding" definiremos "Other > UTF-8".

Continua errado. Acontece que o código fonte está errado, pois isso se aplica também à ele. Precisaremos repetir o processo indicado acima também ao código fonte.

Com todos estes ajustes feitos, testamos novamente, e o texto está correto tanto no código quanto no arquivo `lorem2.txt`.

Se, por exemplo, estivéssemos trabalhando em equipe, não haveria problema, desde que fosse definido um único encoding para o projeto.

Adiante, falaremos sobre serialização. Até lá!

Transcrição

Caso queira, você pode fazer o download (https://s3.amazonaws.com/caelum-online-public/857-java-io/06/java7-aula6.zip) do projeto completo feito até a aula anterior.

Olá! Finalizamos o projeto no Windows OS e daremos seguimento a partir de agora com o Mac OS. Todo o projeto foi exportado para este novo sistema operacional. O `Charset` padrão, neste caso, é o `UTF-8`.

Abriremos o projeto `bytebank-herdado-conta`, e em seguida a classe `SaldoInsuficienteException`, que herda a `Exception`. O Eclipse nos exibe o nome desta classe sublinhado em amarelo, o que significa que é apenas um **alerta**, e não um erro de compilação. Ao clicarmos sobre este alerta, ele nos sugere a adição de uma "*default serial version ID* (ou "ID serial na versão padrão).

Na documentação, surge a palavra chave "*serialization*", em português, "serialização", que será o nosso tópico adiante.

Dentro da máquina virtual, ou JVM, temos a memória de objetos (HEAP), e o `main`, que controla estes objetos. A **serialização** é a transformação do objeto Java, localizado na memória, em um fluxo de bits e bytes, e vice-versa.

Isto é possível graças a duas classes:

- `java.io.ObjectOutputStream` = Objeto -> Bits e Bytes
- `java.io.ObjectInputStream` = Bits e Bytes -> Objeto

A primeira para transformar o objeto em um fluxo de bits e bytes, e a segunda para fazer o caminho inverso.

Qual o propósito de aprendermos isso? Uma situação que pode ocorrer é, por exemplo, gravarmos um objeto em um HD, e podermos recupera-lo posteriormente. Isto está relacionado com a persistência.

O Java foi concebido com a intenção de funcionar em rede, ou seja, fazer com que duas máquinas virtuais se comuniquem em rede. Assim, é possível termos uma funcionalidade em uma JVM e transferi-la para outras, via rede, recebendo e enviando dados. No mundo Java, dados são objetos, portanto, este fluxo é realizado mediante a serialização.

Retornaremos ao Eclipse, e criaremos uma nova classe no pacote `br.com.alura.java.io.teste`, chamada `TesteSerializacao`, com o método `main`.

Utilizaremos um objeto padrão Java, uma classe `String`, que receberá um `nome`:

```
package br.com.alura.java.io.teste;

public class TesteSerializacao {

        public static void main(String[] args) {

            String nome = "Nico Steppat";

        }
}
```

COPIAR CÓDIGO

Em seguida, transformaremos este objeto em um fluxo de bits e bytes. Para isso, faremos uso da classe `ObjectOutputStream`, sem esquecer de importa-la, e criaremos um novo objeto:

```
package br.com.alura.java.io.teste;

public class TesteSerializacao {

	public static void main(String[] args) {

		String nome = "Nico Steppat";

		ObjectOutputStream oos = new ObjectOutputStream();

	}
}
```

COPIAR CÓDIGO

Entretanto, o código não compila. Falta informarmos qual é o fluxo concreto. O construtor precisa receber um `OutputStream` , que é o que está no HD, no caso, o `FileOutputStream("objeto.bin")` . Trata-se de um arquivo binário:

```
package br.com.alura.java.io.teste;

public class TesteSerializacao {

	public static void main(String[] args) {

		String nome = "Nico Steppat";

		ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("objeto.bin"));

	}
}
```

COPIAR CÓDIGO

Precisamos complementar inserindo uma `IOException`:

```
package br.com.alura.java.io.teste;

public class TesteSerializacao throws IOException {

        public static void main(String[] args) {

            String nome = "Nico Steppat";

            ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("objeto.bin"));

        }
}
```

COPIAR CÓDIGO

Utilizaremos o `oos`, para chamarmos o método `writeObject()`, responsável por receber o objeto e transforma-lo em um fluxo de bits e bytes, passar para o `FileOutputStream`, que então grava no HD. Enfim, fecharemos o `oos`:

```
package br.com.alura.java.io.teste;

public class TesteSerializacao throws IOException {

        public static void main(String[] args) {

            String nome = "Nico Steppat";

            ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("objeto.bin"));
            oos.writeObject(nome);
            oos.close();
```

```
        }
}
```

COPIAR CÓDIGO

Executaremos, e para sabermos se tudo funcionou precisamos verificar se um arquivo `objeto.bin` foi criado, visualizando o diretório de pastas na lateral esquerda da tela. No nosso caso foi, indicando que tudo funciona normalmente.

Faremos a seguir o caminho inverso, transformaremos um fluxo de bits e bytes em um objeto. Comentaremos o que acabamos de fazer:

```
package br.com.alura.java.io.teste;

public class TesteSerializacao throws IOException {

        public static void main(String[] args) {

//            String nome = "Nico Steppat";

//            ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("objeto.bin"));
//            oos.writeObject(nome);
//            oos.close();

        }
}
```

COPIAR CÓDIGO

Utilizaremos a classe `ObjectInputStream` , e criaremos um objeto deste tipo, que receberá o nome do arquivo, que no caso será `objecto.bin` :

```
package br.com.alura.java.io.teste;

public class TesteSerializacao throws IOException {

        public static void main(String[] args) {

//            String nome = "Nico Steppat";

//            ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("objeto.bin"));
//            oos.writeObject(nome);
//            oos.close();

              ObjectInputStream ois = new ObjectInputStream(new FileInputStream("objeto.bin"));

        }
}
```

COPIAR CÓDIGO

O `bin` indica que se trata de um arquivo binário.

Com o `ois` chamaremos o método `readObject()`, e ele lerá o fluxo e nos retornará uma `ClassNotFoundException`, já que o objeto pode estar baseado em uma classe ainda não definida, ou que foi apagada.

Teremos portanto um objeto `String`, que nos retornará também uma `String`, e faremos um cast, para especificarmos - uma vez que o `readObject()`, por padrão, nos dá o retorno mais genérico possível. Por fim, fecharemos o `ois`, e imprimiremos o resultado:

```
package br.com.alura.java.io.teste;
```

```
public class TesteSerializacao throws IOException {

        public static void main(String[] args) {

//            String nome = "Nico Steppat";

//            ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("objeto.bin"));
//            oos.writeObject(nome);
//            oos.close();

              ObjectInputStream ois = new ObjectInputStream(new FileInputStream("objeto.bin"));
              String nome = (String) ois.readObject();
              ois.close();
              System.out.println(nome);

        }
}
```

COPIAR CÓDIGO

Executando, temos o seguinte resultado:

```
Nico Steppat
```

COPIAR CÓDIGO

Funcionou.

Conseguimos serializar um objeto Java padrão, no caso, uma `String`. Nosso próximo passo será fazer o mesmo com classes e `Strings` que são baseadas em classes criadas por nós mesmos, por exemplo `Cliente` e `Conta`.

Transcrição

*A classe **Cliente** pode ser baixada [aqui (https://s3.amazonaws.com/caelum-online-public/857-java-io/06/Cliente.java)](https://s3.amazonaws.com/caelum-online-public/857-java-io/06/Cliente.java).*

Anteriormente, fizemos as operações de serialização, e o seu caminho inverso. Entretanto, tratava-se de um objeto do tipo `String`, ou seja, utilizamos uma classe Java padrão.

Nosso objetivo a partir de agora será fazer o mesmo processo, contudo, utilizando uma classe criada pelo próprio programador, ou seja, nós mesmos.

Abriremos o pacote `br.com.bytebank.banco.modelo` e em seguida a classe `Cliente`, e a copiaremos para o projeto `java.io`.

Criaremos também uma cópia da classe `TesteSerializacao`, que chamaremos de `TesteSerializacaoCliente`.

Inseriremos um novo objeto do tipo `Cliente()`, que terá um nome, profissão e um CPF:

```
//Código omitido

public class TesteSerializacao {

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

                Cliente cliente = new Cliente();
```

```
                cliente.setNome("Nico");
                cliente.setProfissao("Dev");
                cliente.setCpf("23413131");

//              String nome = "Nico Steppat";
//              ObjectOutputStream oos = new ObjectOutputStream("objeto.bin"));
//              oos.writeObject(nome);
//              oos.close();

                ObjectInputStream ois = new ObjectInputStream(new FileInputStream("objeto.bin"));
                String nome = (String) ois.readObject();
                ois.close();
                System.out.println(nome);

        }
}
```

COPIAR CÓDIGO

A seguir, comentaremos as linha de leitura, apagaremos primeira linha com o `String nome` , e "descomentaremos" as três seguintes:

```
//Código omitido

public class TesteSerializacao {

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

                Cliente cliente = new Cliente();
                cliente.setNome("Nico");
                cliente.setProfissao("Dev");
                cliente.setCpf("23413131");
```

```
                ObjectOutputStream oos = new ObjectOutputStream("objeto.bin"));
                oos.writeObject(nome);
                oos.close();

//                ObjectInputStream ois = new ObjectInputStream(new FileInputStream("objeto.bin"));
//                String nome = (String) ois.readObject();
//                ois.close();
//                System.out.println(nome);

        }
}
```

COPIAR CÓDIGO

Com a ressalva de que o objeto que gravaremos não é o `nome`, e sim o `cliente`, por isso precisamos fazer a respectiva alteração em `writeObject()`, e para `cliente.bin`:

```
//Código omitido

public class TesteSerializacao {

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

                Cliente cliente = new Cliente();
                cliente.setNome("Nico");
                cliente.setProfissao("Dev");
                cliente.setCpf("23413131");

                ObjectOutputStream oos = new ObjectOutputStream("cliente.bin"));
                oos.writeObject(cliente);
```

```
                oos.close();

//                ObjectInputStream ois = new ObjectInputStream(new FileInputStream("objeto.bin"));
//                String nome = (String) ois.readObject();
//                ois.close();
//                System.out.println(nome);

        }
}
```

COPIAR CÓDIGO

Executando, temos um erro `NotSerializableException`, que é como no dia-a-dia, ao trabalharmos com uma biblioteca de mais alto nível. Isso acontece porque qualquer objeto que queremos serializar precisa "assinar um contrato".

Nossa classe `String` "assina o contrato" automaticamente, portanto, não temos este erro. No caso da classe `Cliente`, precisamos fazer isso manualmente.

Qual seria então este "contrato"? Abriremos a classe `String`:

```
//Código omitido

public final class String
        implements fava.io.Serializable, Comparable<String>, CharSequence {

        //Código omitido
```

COPIAR CÓDIGO

É justamente esta interface `Serializable` que precisamos implementar.

Abrindo ela, vemos o seguinte:

```
//Código omitido

public interface Serializable {
}
```

COPIAR CÓDIGO

Como já vimos, uma interface funciona como um "contrato", assinado para que sejam atendidas as obrigações nela estabelecidas. Estas "obrigações" são, na verdade, os métodos.

Contudo, como podemos observar, esta interface `Serializable` não possui nenhum método. Por isso, é chamada de **interface de marcação**, ela só marca os objetos, sem definir um "contrato formal". Quando o Java introduziu isto, não havia outra forma de marcação.

Dessa forma, nossa classe `Cliente` precisa implementar a interface, sem esquecermos de importa-la:

```
//Código omitido
public class Cliente implements Serializable {

        private String nome;
        private String cpf;
        private String profissao;

        //Código omitido
```

COPIAR CÓDIGO

O Eclipse nos exibe um alerta, sobre a "*serial version ID*". Veremos isso posteriormente, por enquanto, nossa preocupação é a implementação.

Não temos a obrigação de implementar nenhum método, já que se trata de uma interface de marcação.

Executando novamente nossa classe `TesteSerialização`, vemos que ela funcionou, não é exibido nenhum erro. Atualizando o `JRE System Library`, vemos que surge o `cliente.bin`.

A seguir, faremos o caminho inverso. Comentaremos todas as linhas de código que utilizamos para este primeiro exemplo, inclusive a criação do objeto, e descomentaremos as linhas referentes ao `ObjectInputStream`:

```
//Código omitido

public class TesteSerializacao {

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

//                Cliente cliente = new Cliente();
//                cliente.setNome("Nico");
//                cliente.setProfissao("Dev");
//                cliente.setCpf("23413131");

//                ObjectOutputStream oos = new ObjectOutputStream("cliente.bin"));
//                oos.writeObject(cliente);
//                oos.close();

                ObjectInputStream ois = new ObjectInputStream(new FileInputStream("objeto.bin"));
                String nome = (String) ois.readObject();
                ois.close();
                System.out.println(nome);
```

```
        }
}
```

COPIAR CÓDIGO

Precisamos importar as classes `ObjectInputStream` e `FileInputStream`, e alterar o nome do arquivo para o `cliente.bin`. Além disso, precisamos especificar que estamos lendo um objeto do tipo `Cliente`:

```
//Código omitido

public class TesteSerializacao {

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

//                Cliente cliente = new Cliente();
//                cliente.setNome("Nico");
//                cliente.setProfissao("Dev");
//                cliente.setCpf("23413131");

//                ObjectOutputStream oos = new ObjectOutputStream("cliente.bin"));
//                oos.writeObject(cliente);
//                oos.close();

                ObjectInputStream ois = new ObjectInputStream(new FileInputStream("cliente.bin"));
                Cliente cliente = (Cliente) ois.readObject();
                ois.close();
                System.out.println(cliente.getNome());

        }
}
```

COPIAR CÓDIGO

Executando, temos o seguinte resultado no console:

```
Nico
```

COPIAR CÓDIGO

Funcionou, o nome foi recuperado. Podemos tentar o mesmo com o CPF, alterando o método para `getCpf()`:

```
//Código omitido

public class TesteSerializacao {

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

//                Cliente cliente = new Cliente();
//                cliente.setNome("Nico");
//                cliente.setProfissao("Dev");
//                cliente.setCpf("23413131");

//                ObjectOutputStream oos = new ObjectOutputStream("cliente.bin"));
//                oos.writeObject(cliente);
//                oos.close();

                ObjectInputStream ois = new ObjectInputStream(new FileInputStream("cliente.bin"));
                Cliente cliente = (Cliente) ois.readObject();
                ois.close();
                System.out.println(cliente.getCpf());
```

```
        }
}
```

COPIAR CÓDIGO

Executando, temos o seguinte resultado:

```
234113131
```

COPIAR CÓDIGO

Funcionou.

Retornaremos à classe `Cliente`, onde temos a observação do Eclipse, no símbolo de lâmpada, com relação ao ID serial, onde lê-se: "*Add default serial version ID to the selected type*". Ao clicarmos nesta opção, nosso código fica da seguinte forma:

```
//Código omitido
public class Cliente implements Serializable {

        private static final long serialVersionUID = 1L;

        private String nome;
        private String cpf;
        private String profissao;

        //Código omitido
```

COPIAR CÓDIGO

Foi adicionado um atributo estático, do tipo `long`, que é um valor inteiro grande. É uma identificação da própria `Classe`, já que não faz parte do objeto.

Retornando à classe `TesteSerializacao`, a executaremos, e temos um erro no console:

```
Exception in thread "main" java.io.InvalidClassException:

//mensagem de erro continua
```

COPIAR CÓDIGO

O erro `InvalidClassException` está relacionado com a serialização padrão Java. Ao gravarmos um objeto, estamos armazenando as informações que digitamos e, ao mesmo tempo, sem que isso dependa de nossa vontade, uma identificação da classe, um número que identifica a versão da classe.

Se não especificamos um número de versão para a classe, o Java preenche automaticamente.

Na classe `Cliente`, definimos a ID como `1`, contudo, no arquivo - e podemos ver isso no console - foi definido um número de ID `9205117266306915548`. O próprio Java criou esta segunda ID automaticamente, e a armazenou no arquivo `cliente.bin`. Ao recuperar o arquivo, o Java acessa este número serial **do arquivo**, e o compara com o ID da classe `Cliente`, se forem iguais, ele continua, caso contrário, ele para. Como neste caso eles são diferentes, tivemos um erro.

No ponto de vista do Java, a classe que foi utilizada para criar o `cliente.bin` é diferente do objeto da classe que estamos usando para recuperar. As duas classes que estamos utilizando, em momentos diferentes, não são compatível - do ponto de vista do Java.

Para que funcione, copiaremos o número serial criado automaticamente pelo Java, ou seja, `9205117266306915548`, e colaremos no `serialVersionUID` da classe `Cliente`:

```
//Código omitido
public class Cliente implements Serializable {

        private static final long serialVersionUID = 9205117266306915548L;

        private String nome;
        private String cpf;
        private String profissao;

        //Código omitido
```

COPIAR CÓDIGO

Assim, podemos tentar executar novamente. Agora a classe que temos no projeto, e que será armazenada na memória da JVM, tem a mesma versão da ID que a classe utilizada no `cliente.bin`.

Executando, temos o seguinte resultado no console:

```
234113131
```

COPIAR CÓDIGO

Ao trabalharmos com serialização, é uma boa prática inserirmos o `serialVersionUID`.

A justificativa por trás disso pode ser entendida da seguinte forma:

- Gravamos o `cliente.bin`, e em algum momento posterior o recuperaremos;
- Neste intervalo de tempo, a classe `Cliente` pode mudar, por exemplo, ela pode ganhar mais um método;
- Caso não haja o número serial, a cada alteração na classe utilizada para criar o arquivo `cliente.bin`, causa uma nova versão, ou seja, um novo número gerado;

- Por este motivo, é boa prática forçar um número de versão, desta forma, as alterações - **desde que compatíveis** - ficarão armazenadas.

A versão deve ser alterada somente quando forem feitas mudanças incompatíveis com o que foi gravado no arquivo. Se a versão for a mesma, mas for feita uma alteração dos atributos, incompatível com o arquivo, também surgirá um erro.

Este atributo, `serialVersionUID`, serve para administrar a versão da classe.

Ao implementarmos `Serializable`, admitimos que estes objetos podem ser transformados em um fluxo de bits e bytes. Portanto, devemos refletir isso no atributo estático do `serialVersionUID`, e administra-lo.

Isso significa que começamos com uma versão, `1L` por exemplo, e precisamos prestar atenção às mudanças na classe, caso haja uma incompatível com o fluxo de bits e bytes que já foi gravado, com base nos objetos desta classe, precisamos aumentar o número da versão, desta forma deixando claro que a mudança foi incompatível.

Cada alteração nos atributos, por exemplo a sua inclusão ou exclusão, merece uma alteração no número na versão.

Vamos para os exercícios, e até a próxima!

Transcrição

*O projeto **bytebank-herdado-conta** pode ser baixado [aqui (https://s3.amazonaws.com/caelum-online-public/857-java-io/06/bytebank-herdado-conta.zip)](https://s3.amazonaws.com/caelum-online-public/857-java-io/06/bytebank-herdado-conta.zip).*

Anteriormente, vimos conceitos básicos de "serialização" com Java, especialmente, como fazê-lo com base em uma classe própria, e também como "desserializar". Por fim, vimos a questão da `serialVersionUID`.

Veremos a seguir a serialização com base no exemplo da `Conta`, com herança e composição.

Abriremos o projeto `bytebank-herdado-conta`, e no pacote `br.com.bytebank.banco.test` criaremos um sub pacote chamado `br.com.bytebank.test.io`.

No pacote `br.com.bytebank.test.io` criaremos uma classe `TesteSerializacao`, com o método `main`, e o seguinte código:

```
package br.com.bytebank.banco.test.io;

import java.io.FileOutputStream;
import java.io.ObjectOutputStream;

import br.com.alura.java.io.teste.Cliente;

public class TesteSerializacao {
```

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

                Cliente cliente = new Cliente();
                cliente.setNome("Nico");
                cliente.setProfissao("Dev");
                cliente.setCpf("234113131");

                ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("cliente.bin")
                oos.writeObject(cliente);
                oos.close();

        }
}
```

COPIAR CÓDIGO

Neste caso, não gravaremos apenas um `cliente`, mas também uma conta corrente `cc`:

```
//Código omitido

public class TesteSerializacao {

        public static void main(String[] args) {

                Cliente cliente = new Cliente();
                cliente.setNome("Nico");
                cliente.setProfissao("Dev");
                cliente.setCpf("234113131");

                ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("cc.bin"));
```

```
                oos.writeObject(cliente);
                oos.close();

        }
}
```

COPIAR CÓDIGO

Precisamos incluir a exceção `FileNotFoundException` e `IOException`:

```
//Código omitido

public class TesteSerializacao {

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

                Cliente cliente = new Cliente();
                cliente.setNome("Nico");
                cliente.setProfissao("Dev");
                cliente.setCpf("234113131");

                ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("cc.bin"));
                oos.writeObject(cliente);
                oos.close();

        }
}
```

COPIAR CÓDIGO

O próximo passo será de fato criarmos uma conta corrente, que receberá as informações de agência e número da conta:

```
//Código omitido

public class TesteSerializacao {

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

                Cliente cliente = new Cliente();
                cliente.setNome("Nico");
                cliente.setProfissao("Dev");
                cliente.setCpf("234113131");

                ContaCorrente cc = new ContaCorrente(222, 333);

                ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("cc.bin"));
                oos.writeObject(cc);
                oos.close();

        }
}
```

COPIAR CÓDIGO

Além disso, depositaremos uma quantia na conta, para termos um saldo, e determinaremos um `titular`:

```
//Código omitido

public class TesteSerializacao  {

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

                Cliente cliente = new Cliente();
```

```
                cliente.setNome("Nico");
                cliente.setProfissao("Dev");
                cliente.setCpf("234113131");

                ContaCorrente cc = new ContaCorrente(222, 333);
                cc.setTitular(cliente);
                cc.deposita(222.3);

                ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("cc.bin"));
                oos.writeObject(cc);
                oos.close();

        }
}
```

COPIAR CÓDIGO

Executaremos a classe. Temos um erro de `NotSerializableException`. Isso acontece pois a classe `ContaCorrente` não implementa a interface `Serializable`.

Abriremos a classe `ContaCorrente` e faremos este ajuste:

```
//Código omitido

public class ContaCorrente extends Conta implements Tributavel, Serializable {

//Código omitido

}
```

COPIAR CÓDIGO

Lembrando que é possível implementarmos quantas interfaces forem necessárias.

Retornaremos à classe `TesteSerializacao` e a executaremos novamente. Funcionou. No projeto, temos um arquivo `cc.bin`.

Em seguida, faremos o teste de leitura. No pacote `br.com.bytebank.banco.test.io`, criaremos a classe `TesteDeserializacao`, com o método `main`, e criaremos um `ObjectInputStream`, que recebe um `FileInputStream` que recebe o arquivo `cc.bin`:

```
package br.com.bytebank.banco.test.io;

import java.io.FileInputStream;
import java.io.ObjectInputStream;

public class TesteDeserializacao throws FileNotFoundException, IOException {

        public static void main(String[] args) {
                ObjectInputStream ois = new ObjectInputStream(new FileInputStream("cc.bin"));
        }
}
```

COPIAR CÓDIGO

Queremos ler uma `ContaCorrente`, e para isso utilizaremos o método `readObject()`, com o cast de `ContaCorrente`, já que ele devolve uma referência do tipo objeto:

```
//Código omitido

public class TesteDeserializacao throws FileNotFoundException, IOException {

        public static void main(String[] args) {
                ObjectInputStream ois = new ObjectInputStream(new FileInputStream("cc.bin"));
```

```
                ContaCorrente cc = (ContaCorrente) ois.readObject();
        }
}
```

COPIAR CÓDIGO

Com isso, precisaremos adicionar mais uma exceção, que é a `ClassNotFoundException`, e por fim, fecharemos com o método `close()` e imprimiremos o `saldo` e o `cliente`:

```
//Código omitido

public class TesteDeserializacao throws FileNotFoundException, IOException, ClassNotFoundException

        public static void main(String[] args) {
                ObjectInputStream ois = new ObjectInputStream(new FileInputStream("cc.bin"));
                ContaCorrente cc = (ContaCorrente) ois.readObject();
                ois.close();
                System.out.println(cc.getSaldo());
                System.out.println(cc.getTitular());
        }
}
```

COPIAR CÓDIGO

Executaremos, e temos um erro de `InvalidClassException`. O Eclipse está apontando que a classe `ContaCorrente` não possui um construtor válido.

Ela se baseia na classe mãe, inclusive utilizando os construtores desta.

Na classe mãe, `Conta`, vemos que ela não é serializável, para corrigirmos o erro, precisamos ajustar isso, implementando a interface `Serializable` nela:

```
//Código omitido
public abstract class Conta extends Object implements Comparable<Conta>, Serializable {

//Código omitido

}
```

COPIAR CÓDIGO

Com isso, podemos inclusive remover a implementação que havíamos feito na classe `ContaCorrente`:

```
//Código omitido

public class ContaCorrente extends Conta implements Tributavel {

//Código omitido

}
```

COPIAR CÓDIGO

Retornaremos à classe `TesteSerializacao` e a executaremos. Temos um outro erro, desta vez `NotSerializableException` para a classe `Cliente`. Isso significa que a serialização também é aplicada aos objetos que se associam por agregação. A classe `Conta` possui um atributo `Cliente titular`.

Portanto, a classe `Cliente` precisa implementar a interface `Serializable`:

```
//Código omitido
public class Cliente implements Serializable {
```

```
//Código omitido
```

COPIAR CÓDIGO

Executaremos a classe `TesteSerialização` novamente, e funcionou.

Em seguida, executaremos a classe `TesteDeserializacao` , e vemos o seguinte resultado no console:

```
222.3
br.com.bytebank.banco.modelo.Cliente@5a39699c
```

COPIAR CÓDIGO

Funcionou, mas ainda temos uma saída estranha. Para melhorarmos, utilizaremos o método `getNome()` :

```
//Código omitido

public class TesteDeserializacao throws FileNotFoundException, IOException, ClassNotFoundException

        public static void main(String[] args) {
                ObjectInputStream ois = new ObjectInputStream(new FileInputStream("cc.bin"));
                ContaCorrente cc = (ContaCorrente) ois.readObject();
                ois.close();
                System.out.println(cc.getSaldo());
                System.out.println(cc.getTitular().getNome());
        }
}
```

COPIAR CÓDIGO

Executando, temos o seguinte resultado no console:

```
222.3
Nico
```

COPIAR CÓDIGO

Funcionou.

Para que não seja necessária a serialização da classe `Cliente`, podemos declarar que o `Cliente` é `transient`. É uma palavra chave do mundo Java, e significa que ele não faz parte da serialização, ou seja, não será gravado no objeto:

```
//Código omitido

public abstract class Conta extends Object implements Comparable<Conta>,Serializable {

        protected double saldo;
        private int agencia;
        private int numero;
        private transient Cliente titular;
        private static int total = 0;

//Código omitido
```

COPIAR CÓDIGO

Hoje, isso seria representado como uma anotação `@transient`, mas ao nascimento do Java, não existia esta modalidade de sintaxe.

Executaremos novamente a classe `TesteSerializacao` e vemos que ela funcionou.

Quanto à classe `Cliente`, podemos inclusive remover a implementação da interface `Serializable`:

```
//Código omitido
public class Cliente {

//Código omitido
```

COPIAR CÓDIGO

A classe `TesteSerializacao` continua executando normalmente. O arquivo existe, no entanto, sem os dados do `Cliente`.

Partiremos então para a classe `TesteDeserializacao`, e comentaremos a linha referente ao titular:

```
//Código omitido

public class TesteDeserializacao throws FileNotFoundException, IOException, ClassNotFoundException

        public static void main(String[] args) {
                ObjectInputStream ois = new ObjectInputStream(new FileInputStream("cc.bin"));
                ContaCorrente cc = (ContaCorrente) ois.readObject();
                ois.close();
                System.out.println(cc.getSaldo());
//                System.out.println(cc.getTitular().getNome());
        }
}
```

COPIAR CÓDIGO

Executando, temos o saldo impresso normalmente.

A seguir, executaremos novamente, desta vez imprimindo as informações do titular, além do saldo:

```
//Código omitido

public class TesteDeserializacao throws FileNotFoundException, IOException, ClassNotFoundException

        public static void main(String[] args) {
                ObjectInputStream ois = new ObjectInputStream(new FileInputStream("cc.bin"));
                ContaCorrente cc = (ContaCorrente) ois.readObject();
                ois.close();
                System.out.println(cc.getSaldo());
            System.out.println(cc.getTitular());
        }
}
```

COPIAR CÓDIGO

Executando, temos o seguinte resultado no console:

```
222.3
null
```

COPIAR CÓDIGO

Ele imprime `null`, pois não recuperou o `Cliente`, já que não estava no arquivo pois não foi gravado.

Vamos para os exercícios, e até a próxima!

O que aprendemos?

Nesta aula, falamos bastante sobre a serialização de objetos com Java. Vimos que:

- A criação do fluxo binário a partir de um objeto é chamado de **serialização**;
- A criação de um objeto a partir de um um fluxo binário é chamado de **desserialização**;
- A classe deve implementar a interface `java.io.Serializable` ;
- A serialização/desserialização funciona em cascata e também com herança;
- Existe a palavra-chave `transient` para indicar que o atributo não deve ser serializado;
- É boa prática colocar o atributo estático `serialVersionUID` para versionar a classe;
- A versão sempre fica guardada no fluxo binário;
- Se não colocarmos explicitamente o `serialVersionUID` , a versão será gerada dinamicamente;
- É raro usar a serialização na "unha", mas é um conhecimento importante, pois será utilizado por outras bibliotecas.