

02

## Criação de pacotes

### Transcrição

Antes de trabalharmos no código propriamente, iremos preparar nosso ambiente.

Temos o Java instalado e na linha de comando escreveremos `version` e veremos que o Java executado está na versão `9.0.4`.

```
Last login: Thu Mar 8 13:50:45 on console
Aluras-iMac:~ alura$ java -version
java version "9.0.4"
Java(TM) SE Runtime Environment (build 9.0.4+11)
Java HotSpot(TM) 64-Bit Server VM (build 9.0.4+11, mixed mode)
Aluras-iMac:~ alura$
```

[COPIAR CÓDIGO](#)

Da mesma forma, as outras ferramentas que estão juntas ao *development kit* também estão na versão `9.0.4`.

```
Last login: Thu Mar 8 13:50:45 on console
Aluras-iMac:~ alura$ java -version
java version "9.0.4"
Java(TM) SE Runtime Environment (build 9.0.4+11)
Java HotSpot(TM) 64-Bit Server VM (build 9.0.4+11, mixed mode)
Aluras-iMac:~ alura$ javac - version
```

```
javac 9.0.4
```

```
Alura-iMac: ~ alura$
```

[COPIAR CÓDIGO](#)

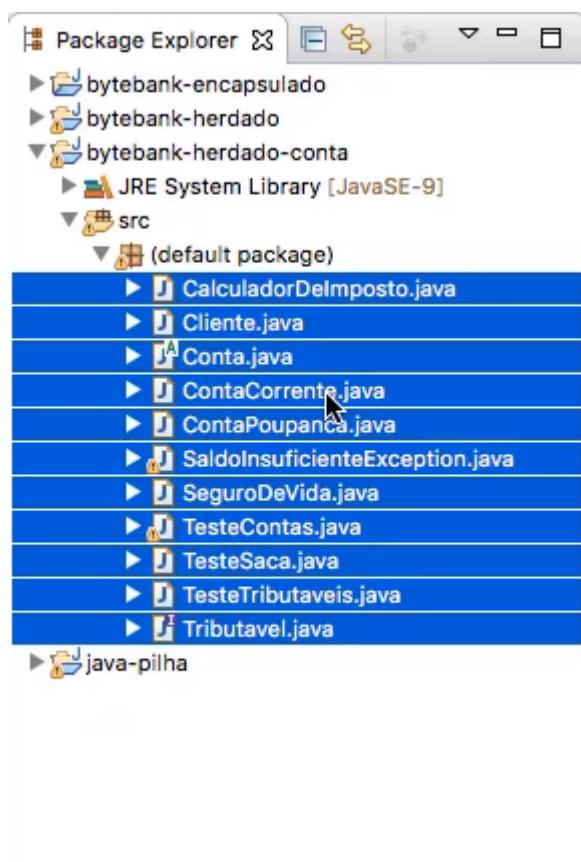
A linha de comando está configurada, o Eclipse também está instalado e iremos inicializa-lo. Estamos utilizando a versão Oxygen, mas uma versão anterior poderia ser utilizada sem problemas. No caso do Java, para nosso projeto, bastaria a versão 1.7 ou 1.8.

No Package Explorer temos os projetos que foram desenvolvidos nas outras partes do curso. Você pode fazer [download dos arquivos do projeto.](#)

(<https://s3.amazonaws.com/caelum-online-public/843-java-packages/java5-projetos-inicias.zip>)

Estamos com o ambiente preparado e podemos iniciar.

Você já possui uma base sólida em Java, mas ainda existem alguns pontos que podemos conhecer melhor. Na área do Package Explorer, selecionaremos o projeto `bytebank-herdado-conta` e veremos que existem onze arquivos `.java` de código fonte.



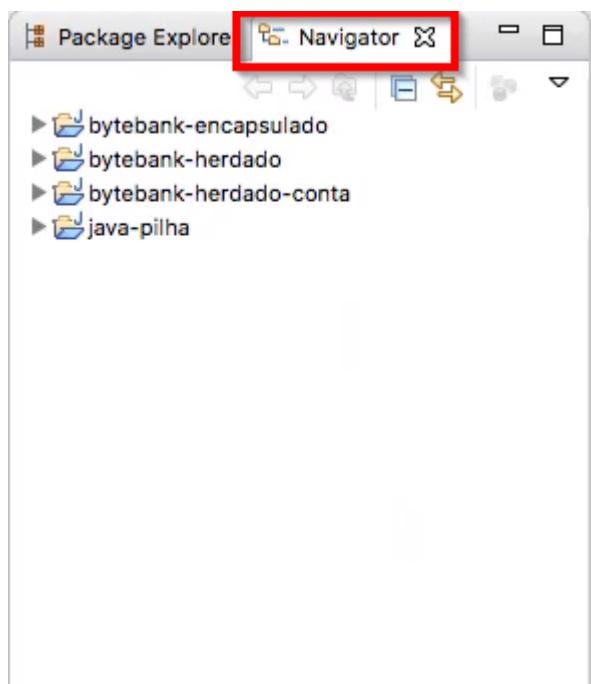
Em um projeto real teríamos facilmente por volta de quinhentos arquivos, mas podemos chegar no ponto da discussão apenas com estes onze arquivos.

Queremos organizar melhor esses arquivos, e agrupa-los todos em uma única pasta não é uma boa solução. Em nosso exemplo, podemos identificar pelo menos dois tipos de classes: reparem que temos os testes ( `TesteContas` , `TesteSaca` e `TesteTributaveis` ) e todas as outras fazem parte do nosso domínio.

Iremos separar esses dois tipos de classes; faremos isso organizando as classes em diretórios específicos. Em um primeiro momento, esse procedimento será feito da forma mais custosa, para depois aprendermos a fazer uso de todas as facilidades disponibilizadas pelo Eclipse.

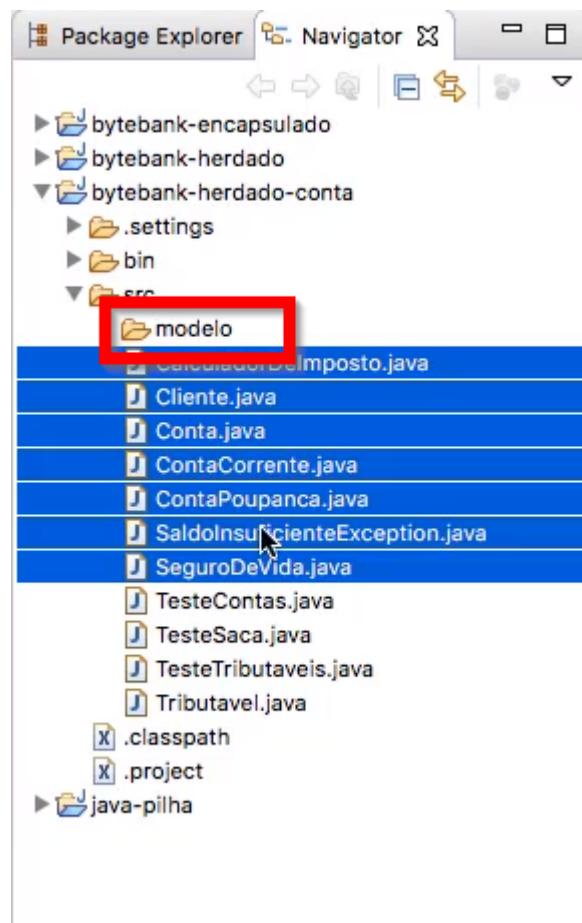
Estamos utilizando a visualização do Package Explorer, deixaremos de fazer isso neste momento.

Na parte superior esquerda, teremos um buscador e procuraremos por outra view conhecida por "Navigator". Selecionaremos a opção correspondente e ela se tornará visível ao lado esquerdo da tela.

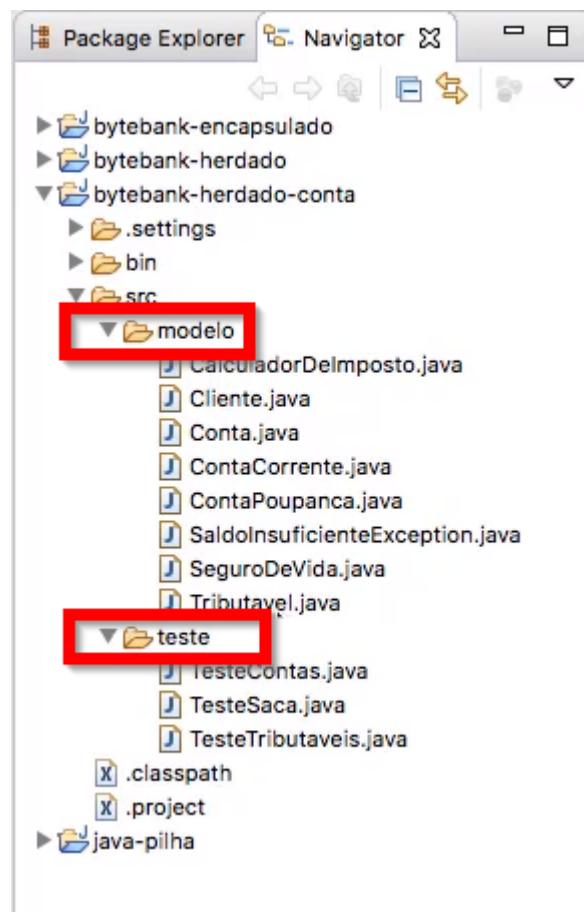


O Navigator opera como um explorer do Windows, exibe as pastas do nosso interesse de forma mais completa. Já o Package Explorer oculta alguns conteúdos a fim de simplificar a visualização, focando nos itens mais importantes. Percebam que no Navigator a pasta `bin` é exibida, diferentemente do Package Explorer.

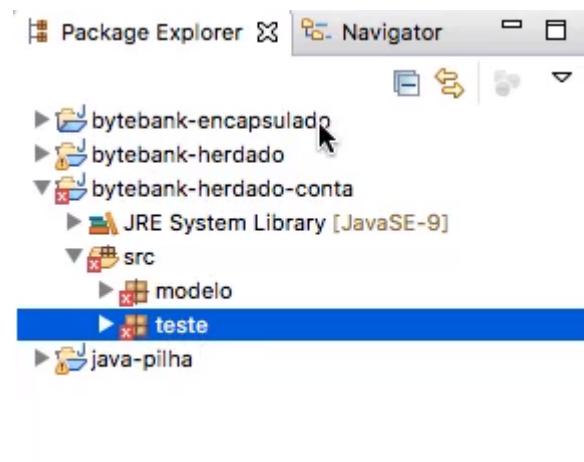
Nosso objetivo é separar nossas classes em diretórios diferentes, como já foi dito. Em `src`, clicaremos com o botão direito e selecionaremos a opção "New > Folder". O primeiro diretório se chamará `modelo`, e dentro dessa pasta colocaremos todos os arquivos que não são de teste, como `CalculadorDeImposto`, `Cliente`, `Conta` e assim por diante.



Criaremos uma segunda pasta que chamaremos de `teste`, em que, claramente, guardaremos os arquivos de teste. Com isso temos as duas classes divididas entre diretórios específicos, visualizados no Navigator.



Ao clicarmos no visualizador Package Explorer reparem que há ocorrência de erros que podem ser vistos através da mudança na visualização dos diretórios que criamos, há um pequeno alerta vermelho.



Pastas e diretórios possuem um nome especial dentro do mundo Java: são os **pacotes**, ou em inglês *packages*. Esses pacotes existem para organizarmos melhor o

nosso código. No entanto, quando colocamos uma classe dentro de um pacote, essa ação precisa ser evidenciada no código fonte.

Vamos entender melhor como se dá esse processo.

Abriremos o arquivo `CalculadorDeImposto`, localizado no diretório `modelo`. Percebemos que o Eclipse já aponta erros de compilação quando abrimos esta classe, afinal, ela faz parte do pacote `modelo` e isso precisa ser definido no código fonte, e deve ser a primeira declaração do código.

Por isso, inseriremos a palavra chave `package` e inseriremos o nome da pasta, no caso, `modelo`.

```
package modelo;

public class CalculadorDeImposto {

    private double totalImposto;

    public void registra(Tributavel t) {
        double valor = t.getValorImposto();
        this.totalImposto += valor;
    }
}
```

[COPIAR CÓDIGO](#)

Com essa declaração, o código já está sendo compilado normalmente. O nome da pasta que deve ser escrito após o uso da palavra chave `package` deve sempre partir do diretório `src`, o local em que está armazenado o código fonte e para onde o compilador do Eclipse olhará.

Esse mesmo processo deve ser feito para todos os arquivos, por isso copiaremos a declaração `package modelo` por meio do atalho "Ctrl + C" e colaremos em todos as

classes. Para as classes na pasta teste a declaração será package teste . Esse é o jeito mais trabalhoso de executar essa ação, mas depois aprenderemos outra forma.

Os erros de declaração do pacote não existem mais, no entanto, são apresentadas outras falhas pelo Eclipse.

Separamos nossos arquivos e eles se encontram em pacotes separados. Observamos o código da classe testeTributaveis :

```
package teste;

public class TesteTributaveis {

    public static void main(String[] args) {
        ContaCorrente cc = new ContaCorrente(222, 333);
        cc.deposita(100.0);

        SeguroDeVida seguro = new SeguroDeVida();

        CalculadorDeImposto calc = new CalculadorDeImposto();
        calc.registra(cc);
        calc.registra(seguro);

        System.out.println(calc.getTotalImposto());
    }
}
```

**COPIAR CÓDIGO**

A classe faz referência à ContaCorrente , e é justamente nesse ponto em que surgem problemas, pois essa classe está no diretório modelo , e não em teste . O package não é apenas um diretório simples, ele passa a fazer parte do nome da classe, portanto a classe ContaCorrente não possui mais essa nomeação, passa a ser modelo.ContaCorrente . Portanto, para que o código seja compilado, devemos

atualizar os nomes simples das classes que se encontram no pacote `modelo` em todos os arquivos de teste .

```
package teste;

public class TesteTributaveis {

    public static void main(String[] args) {
        modelo.ContaCorrente cc = new modelo.ContaCorrente(222, 333
        cc.deposita(100.0);

        modelo.SeguroDeVida seguro = new modelo.SeguroDeVida();

        modelo.CalculadorDeImposto calc = new modelo.CalculadorDeImposto();
        calc.registra(cc);
        calc.registra(seguro);

        System.out.println(calc.getTotalImposto());
    }
}
```

[COPIAR CÓDIGO](#)

Esse nome completo da classe que contém o nome de seu diretório é conhecido como **Full Qualified Name** ou **FQN**.

05

## Importando pacotes

### Transcrição

Começamos anteriormente, a discutir a questão dos pacotes. Vimos como o Package Explorer atua na visualização simplificada dos arquivos, mas por hora continuaremos fazendo uso do Navigator que exibe mais conteúdos.

Criamos os pacotes `modelo` e `teste` para separarmos as classes. Observamos que essa informação de armazenamento deve constar no código fonte usando a palavra chave `package`, e que o nome da classe se modifica ao realizarmos a reorganização dos arquivos. Esse nome completo da classe é conhecido por *full qualified name*; internamente a máquina sempre utilizará esse nome completo para fazer a identificação das classes.

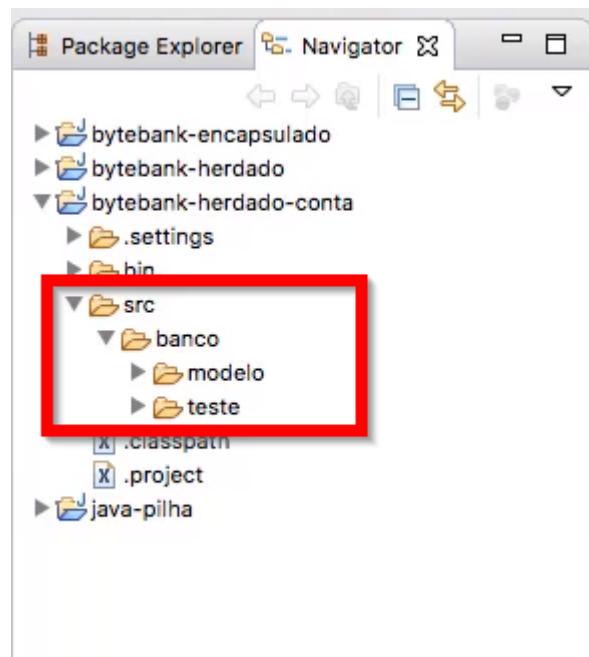
No entanto, podemos perceber que a nossa nomenclatura ainda é frágil, olhemos para a classe `CalculadorDeImposto`, por exemplo: o Java é uma linguagem muito popular no Brasil, imaginem quantas classes chamadas `CalculadorDeImposto` não existem no país? Além disso, o nome do pacote `modelo` também é muito comum, algumas pessoas utilizam o nome `dominio`.

Como esses nomes são bem comuns, é muito provável que já tenhamos utilizado em outro projeto, isso é um problema no momento em que precisamos utilizar um código escrito por outra pessoa e a mesma nomenclatura foi utilizada em seu próprio projeto.

A ideia geral é: como podemos aproveitar classes com o mesmo nome em um outro projeto, sem precisar renomear nenhuma delas?

No sistema operacional, ao abrimos uma pasta, sabemos que dentro dela não podem ser criados dois arquivos com o mesmo nome, o mesmo vale para o mundo Java.

Em `src` criaremos uma pasta com o nome do projeto, ou seja, `banco`. Dentro dessa pasta colocaremos todas as classes de `modelo`; para isso, selecionamos `modelo` no visualizador Navigator e o arrastamos para dentro da pasta `banco`. Faremos o mesmo procedimento com `teste`.



Veremos que assim que fizemos essa modificação, os erros novamente surgem, afinal o pacote não é reconhecido apenas por `modelo`, pois incluímos esta pasta dentro de outra denominada `banco`. Precisaremos fazer a seguinte modificação para que os códigos compilem corretamente:

```
package banco.modelo;  
  
public class CalculadorDeImposto {
```

```
private double totalImposto;

public void registra(Tributabel t) {
    double valor = t.getValorImposto();
    this.totalImposto += valor;
}
```

**COPiar CÓDIGO**

Estamos utilizando o nome do projeto na nomenclatura das classes `banco.modelo`, adotando o nome mais específico. No entanto, a comunidade Java ainda considera essa nomeação frágil, e conflitos ainda podem ocorrer no desenvolvimento de um projeto.

Iremos incluir outra pasta que terá o nome da empresa, que por sua vez guardará todas as pastas e classes específicas.

O nome da nossa pasta será `bytebank`, e ela conterá a pasta `banco`, que por sua vez contém `modelo` e `teste`. Mesmo assim, a comunidade Java não está satisfeita, a nomenclatura pode ainda gerar confusões já que podem existir duas ou mais empresas no mundo com o mesmo nome.

O que de fato identifica uma empresa de forma única no mundo inteiro? No caso da **Alura**, o que a identifica na web é seu endereço, ou seja [www.alura.com.br](http://www.alura.com.br) (<http://www.alura.com.br>). Só existe esse endereço para esta página.

Aproveitaremos a ideia dos endereços da internet nos pacotes Java. Criaremos um conjunto de pastas que recriam o endereço do portal, iniciando pelo país, ou seja, em `src` criaremos uma pasta denominada `br`; depois criaremos uma pasta `com` que representa o sub-domínio. Portanto, temos a seguinte organização de arquivos: "src > br > com > bytebank > banco > modelo,teste", essa é a sistematização do Java que encontraremos em nosso dia a dia.

Qual será o *full qualified name* da classe `CalculadorDeImposto` nesta altura do projeto?

Além da palavra chave `package` inseriremos no começo do código fonte  
`br.com.bytebank.banco.modelo`. E o nome da classe será  
`br.com.bytebank.banco.modelo.CalculadorDeImposto`.

```
package br.com.bytebank.banco.modelo;

public class CalculadorDeImposto {

    private double totalImposto;

    public void registra(Tributavel t) {
        double valor = t.getValorImposto();
        this.totalImposto += valor;
    }
//...
```

**COPIAR CÓDIGO**

Esse tipo de estruturação pode parecer exagerada em um primeiro momento, mas essa prática se tornou muito útil durante o desenvolvimento de projetos. Não precisamos criar todas as pastas manualmente como fizemos até este ponto, a ferramenta Eclipse criará as pastas de forma automática e rápida.

Aprendemos que o nome completo da classe `CalculadorDeImposto` é  
`br.com.bytebank.banco.modelo.CalculadorDeImposto`. Veremos o nome da classe `TesteTributaveis`. Percebem que o Eclipse irá autocompletar a linha, porque já conhece a estrutura padrão.

Mas precisamos alterar as classes que estão sendo citadas ao longo do código, incluindo seus respectivos nomes completos para que o código seja compilado

corretamente.

```
package br.com.bytebank.banco.teste.TesteTributaveis;

public class TesteTributaveis {

    public static void main(String[] args) {
        br.com.bytebank.banco.modelo.ContaCorrente cc = new br.com.

        br.com.bytebank.banco.modelo.SeguroDeVida seguro = new mode.

        br.com.bytebank.banco.modelo.CalculadorDeImposto calc = new
```

**COPIAR CÓDIGO**



Além de ser um processo trabalhoso, esse tipo de ação dificulta a legibilidade do código. Não precisamos fazer deste modo, existe a opção de importarmos o pacote necessário e com isso podemos continuar utilizando o nome simples das classes.

As importações são expressas logo abaixo da palavra-chave `package`, sendo essa uma estrutura regular de organização do código. Acionaremos a palavra chave `import` e vamos declarar o que queremos importar.

```
package br.com.bytebank.banco.teste;

import br.com.bytebank.banco.modelo.*;
```

**COPIAR CÓDIGO**

Com isso, podemos manter nosso código mais limpo e otimizar as adaptações necessárias.

Como vocês puderam ver, há muitas adequações que devemos realizar em nosso código atualmente, precisamos realizar as importações necessárias e corrigir as classes que aparecem com nomes incorretos. Façam isso, observem o que precisa ser retificado e, apenas então, prosseguiremos nas próximas lições.

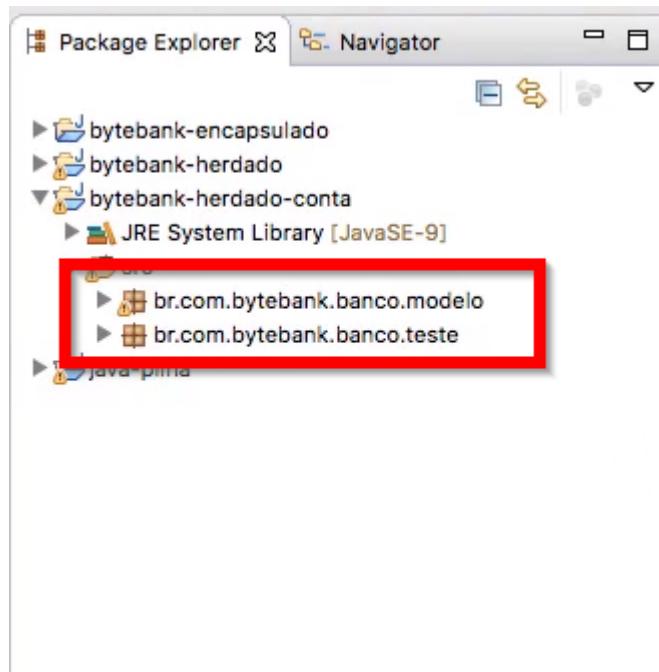
08

## Gerenciando pacotes com Eclipse

### Transcrição

Nesta altura do projeto já criamos nossos pacotes e uma hierarquia de pastas, também fizemos as declarações adequadas no `package` de cada classe. No caso das classes da pasta `teste`, tivemos de acionar a palavra-chave `import` para realizarmos as importações da pasta `modelo`.

Iremos nos voltar para o visualizador Package Explorer. Reparem que o Eclipse exibe o pacote `br.com.bytebank.banco.modelo`, sabemos que existem pastas únicas para `br`, `com` e assim sucessivamente, no entanto essas pastas são agrupadas para facilitar a visualização pelo usuário.



Acessaremos o arquivo `TesteContas` para explorarmos as facilidades disponibilizadas pelo Eclipse. Não há necessidade de criarmos nenhuma pasta manualmente, a própria ferramenta realiza essa ação.

Observem o código da classe:

```
package br.com.bytebank.banco.teste;

import br.com.bytebank.banco.modelo.*;

public class TesteContas {

    public static void main(String[] args) throws SaldoInsuficienteException {
        ContaCorrente cc = new ContaCorrente(111, 111);
        cc.deposita(100.0);

        ContaPoupanca cp = new ContaPoupanca(222, 222);
        cp.deposita(200.0);

        cc.transfere(10.0, cp);

        System.out.println("CC: " + cc.getSaldo());
        System.out.println("CP: " + cp.getSaldo());
    }
}
```

**COPIAR CÓDIGO**

Retiraremos as importações da classe `TesteContas` e veremos que um erro é apontado para a `SaldoInsuficienteException`, afinal, não estamos conseguindo identificar a procedência dessa classe se não há as devidas especificações.

```
package br.com.bytebank.banco.teste;

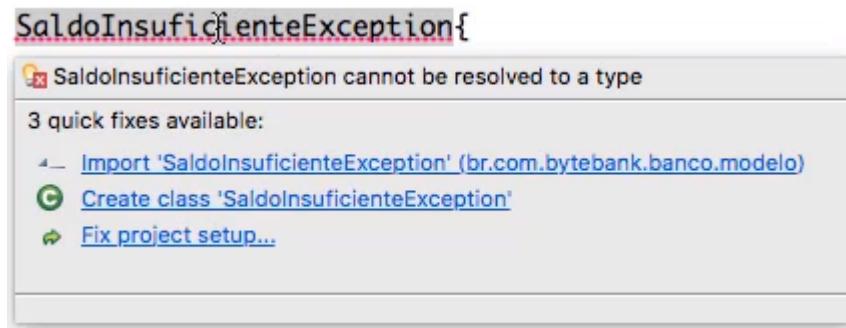
public class TesteContas {

    public static void main(String[] args) throws SaldoInsuficiente
//...
```

**COPIAR CÓDIGO**



No entanto, ao selecionarmos com o mouse a classe `SaldoInsuficienteException` no código fonte de `TesteContas`, veremos que Eclipse sugere a importação.



Ao aceitarmos a importação sugerida pelo Eclipse, veremos que existem algumas diferenças entre o modelo de importação que estávamos realizando, pois fazia uso do asterisco (\*). O asterisco simboliza que todas as classes de `modelo` são importadas. Mas a importação sugerida pelo Eclipse é específica, e aponta somente para `SaldoInsuficienteException`.

```
package br.com.bytebank.banco.teste;

import br.com.bytebank.banco.modelo.SaldoInsuficienteException;

public class TesteContas {

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

[COPIAR CÓDIGO](#)

Normalmente as importações específicas são privilegiadas, pois elas garantem que não importemos nenhuma outra classe que não seja de nossa escolha. As importações não possuem nenhum impacto no desempenho do projeto. Neste caso, precisamos importar outras classes que não são mencionadas no código, como `ContaCorrente` e `ContaPoupanca`.

```
package br.com.bytebank.banco.teste;

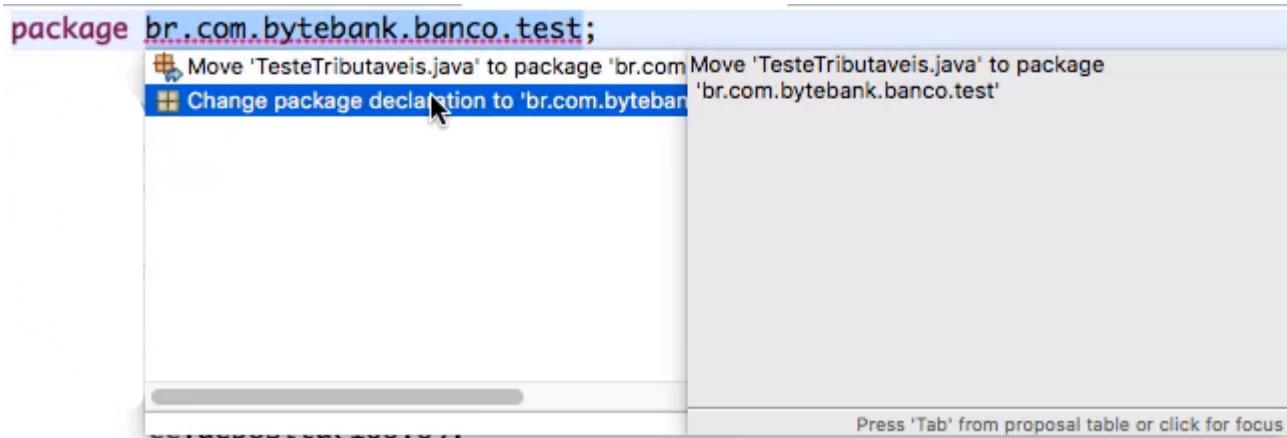
import br.com.bytebank.banco.modelo.ContaCorrente;
import com.bytebank.banco.modelo.ContaPoupanca;
import br.com.bytebank.banco.modelo.SaldoInsuficienteException;
```

[COPIAR CÓDIGO](#)

No entanto, não precisamos selecionar com o mouse as classes que precisam ser importadas, basta acionarmos com o atalho "Ctrl + Shift + O" e a ferramenta Eclipse irá realizar automaticamente as importações necessárias no código fonte.

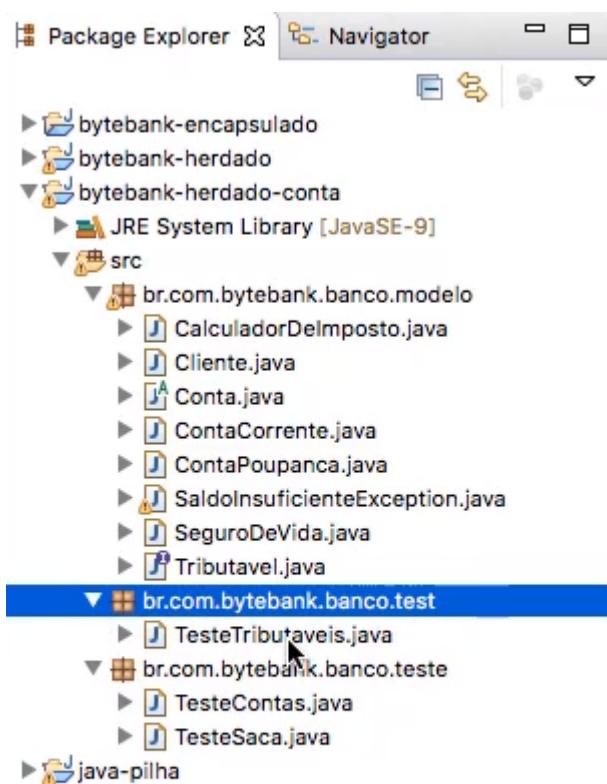
Não existe a necessidade de digitar nenhum `import` no Eclipse, assim como não precisamos realizar manualmente a declaração do `package` ou criar pastas.

Retiraremos uma letra da declaração de `package` na classe `TesteTributaveis`, escrevendo `test` em vez de `teste` ao final da declaração, de forma que o Eclipse aponte um erro. Ao selecionarmos a declaração, veremos que a ferramenta nos aponta duas soluções: a renomeação ou a criação de um novo pacote.



Press 'Tab' from proposal table or click for focus

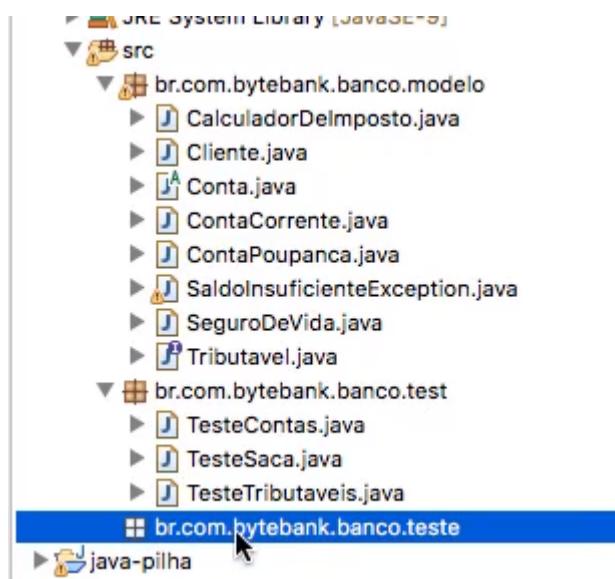
Escolheremos a opção de criação de um novo pacote. Percebam que o Eclipse criou esse novo pacote `br.com.bytebank.banco.test` que contém a classe `br.com.bytebank.banco.test.`



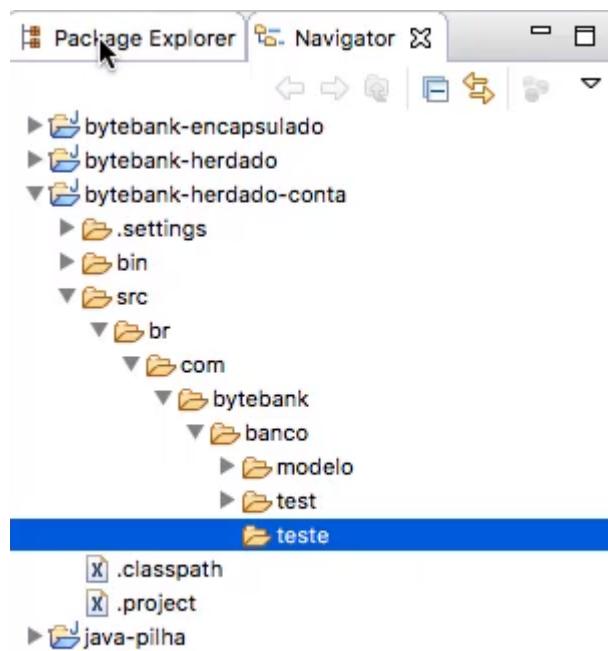
Podemos mover outras classes para dentro do pacote que criamos, basta selecionar os arquivos e arrasta-los para o pacote fim. Selecionearemos `TesteContas` e `TesteSaca`, que antes estavam no pacote `br.com.bytebank.banco.teste`.

Caso ocorra algum erro durante o processo de movimentar os arquivos para um novo pacote, entre no arquivo de origem, pressione com o botão direito na linha de declaração de `package` e escolha a opção "Move [classe] to [pacote]"

O pacote `br.com.bytebank.banco.teste` ficou vazio, isso é simbolizado por um ícone branco ao lado do nome do pacote.



Ao observarmos o Navigator veremos que a pasta `teste` existe, mas não está mais armazenando nenhum arquivo, afinal todas as classes foram movidas para `test`.



Podemos excluir apenas a pasta `teste` sem que isso altere a ordem da hierarquia dos arquivos, e como ela está vazia, assim o faremos.

Temos toda essa estrutura de pastas dentro de `src` do Eclipse, onde se localiza nosso código fonte; mas temos, ainda, a pasta `bin` que armazena as classes compiladas. Reparem que em `bin` temos a mesma estrutura hierárquica de pastas, ou seja "bin > br > com > bytebank > banco > modelo > test".

Com isso, já sabemos como organizar nossas classes no mundo Java por meio dos pacotes, sempre seguindo a nomenclatura padrão.

Nos falta comentar sobre os modificadores de visibilidade, que podem ser dos tipos `public`, `protected` ou `private`, por exemplo. Trabalharemos nessa questão nas próximas aulas.



20 12

## O que aprendemos?

### O que aprendemos?

- packages servem para organizar o nosso código
- packages fazem parte do FQN (*Full Qualified Name*) da classe
- o nome completo da classe (FQN) é composto de:  
`PACKAGE.NOME_SIMPLES_CLASSE`
- a definição do package deve ser a primeira declaração no código fonte
- para facilitar o uso de classes de outros packages podemos importá-los
- os `import`s ficam logo após da declaração do `package`
- a nomenclatura padrão é usar o nome do domínio na web ao contrário junto com o nome do projeto, por exemplo:

`br.com.caelum.geradornotas`

`br.com.alura.gnarus`

`br.gov.rj.notas`

`de.adidas.lager`

[COPIAR CÓDIGO](#)

Uma vez organizado as nossas classes podemos revisar o modificadores de visibilidades que dependem dos pacotes. Vamos continuar?

01

## Revisitando modificadores de acesso

### Transcrição

Continuaremos estudando os modificadores de acesso/visibilidade. Por meio dos pacotes conseguimos explicar todos os modificadores e quais são suas diferenças. Se observarmos o seguinte esquema, veremos que está organizado na ordem do "mais visível" ao "menos visível", sendo o `public` visível em todas as áreas e `private` visível somente dentro da classe.

Modificadores de Acesso/Visibilidade	
-----	-----
<code>public</code>	
<code>protected</code>	
<code>&lt;&lt;package private&gt;&gt;</code>	
<code>private</code>	

COPiar Código

Temos em vista um termo novo, o `<<package private>>`. Veremos como esse modificador de visibilidade atua. Abriremos a classe `Conta`, localizada no pacote `br.com.bytebank.banco.modelo`.

```
package br.com.bytebank.banco.modelo;  
  
public abstract class Conta {
```

```
protected double saldo;  
private int agencia;  
private int numero;  
private Cliente titular;  
private static int total = 0;  
//...
```

[COPIAR CÓDIGO](#)

Percebem que essa classe possui o modificador `public`, isso significa, como já sabemos, que ela é visível em todos os espaços dentro e fora do pacote. No código de `TesteSaca`, localizado no pacote `br.com.bytebank.banco.test`, há uma referência a classe `Conta`.

```
package br.com.bytebank.banco.test;  
  
import br.com.bytebank.banco.modelo.Conta;  
  
public class TesteSaca {  
  
    public static void main(String[] args) {  
  
        Conta conta = new ContaCorrente(123, 321);
```

[COPIAR CÓDIGO](#)

Essa referência só pode ser feita porque a classe `Conta` é pública e pode ser acessada por elementos que estão fora de seu pacote, como é `TesteSaca`. Caso o `public` do código da classe `Conta` seja apagado, imediatamente o código de `TesteSaca` apresenta erros em sua compilação. As outras classes que compartilham o mesmo pacote de `Conta` operam normalmente.

Abriremos a classe `ContaCorrente`, observemos que há o modificador `public` em frente ao construtor `ContaCorrente`.

```
package br.com.bytebank.banco.modelo;

public class ContaCorrente extends Conta implements Tributavel {

    public ContaCorrente(int agencia, int numero) {
        super(agencia, numero);
    }
```

**COPiar CÓDIGO**

Se `public` for removido, a classe será visível, mas não o construtor.

Consequentemente haverá a ocorrência de erros na classe `TesteSaca`, que faz referência ao construtor `ContaCorrente`. Podemos definir a visibilidade para a classe, construtor, atributo, método e qualquer outro membro do código.

Quando não expressamos diretamente um modificador de visibilidade como `public`, a classe se torna visível apenas dentro de seu próprio pacote, isso significa `<<package private>>`, não se trata de uma palavra utilizada no mundo Java, mas a condição de uma classe nos termos de visibilidade, caso não especifiquemos essa condição.

Temos, ainda, o `protected`. Vamos entender esse conceito. Observemos o código da classe `Conta`:

```
package br.com.bytebank.banco.modelo;

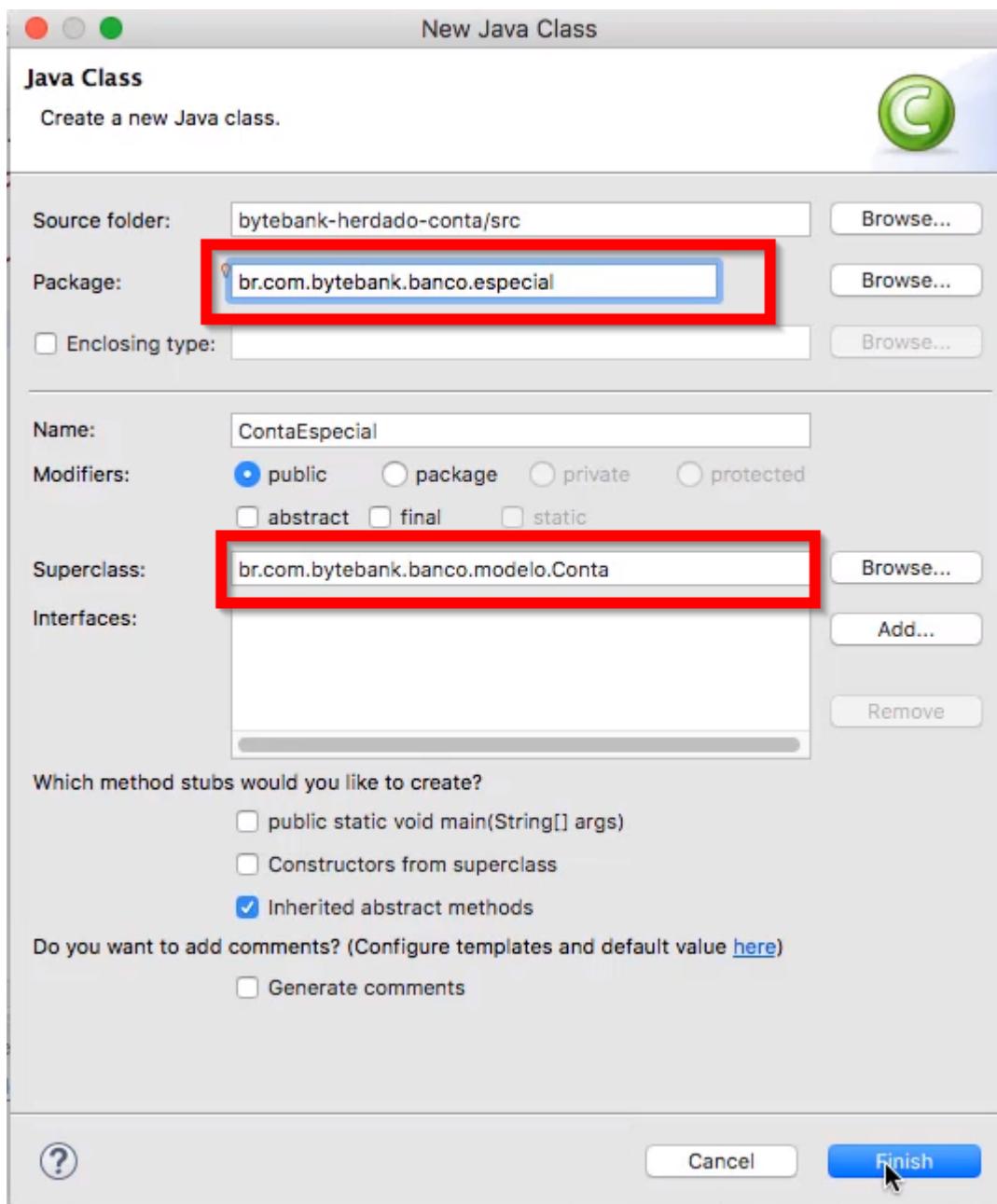
public abstract class Conta {

    protected double saldo;
    private int agencia;
    private int numero;
    private Cliente titular;
    private static int total = 0;
```

[COPIAR CÓDIGO](#)

Ao removermos o `protected double saldo`, não percebemos nenhum erro de compilação no código, ou seja, não há nenhuma modificação aparente. Para compreendermos essa suposta "inalteração", criaremos uma nova classe que chamaremos de `ContaEspecial`, que irá estender a classe `Conta`, portanto, na opção "Superclass" no painel de criação de classes, escolheremos a classe `Conta`.

Na opção "Package", criaremos um novo pacote para essa classe, que chamaremos de `br.com.bytebank.banco.especial`. Percebam que basta assinalarmos um novo pacote no painel de criação de classes que o Eclipse criará, também, um novo pacote para armazenar essa classe.



Vejamos o código da classe ContaEspecial :

```
package br.com.bytebank.banco.especial;

import br.com.bytebank.banco.modelo.Conta;

public class ContaEspecial extends Conta {

    @Override
    public void deposita(double valor) {
```

```
// TODO auto-generated method stub  
  
}  
}
```

[COPIAR CÓDIGO](#)

Há uma extensão de `Conta` mas existe um erro de compilação. Precisamos implementar o construtor específico da classe `Conta`. Tentaremos fazer de uma forma mais rápida e copiaremos o construtor de `ContaCorrente`.

```
package br.com.bytebank.banco.modelo;  
  
//new ContaCorrente()  
public class ContaCorrente extends Conta implements Tributavel {  
  
    public ContaCorrente(int agencia, int numero) {  
        super(agencia, numero);  
    }  
}
```

[COPIAR CÓDIGO](#)

Colaremos o construtor na classe `ContaEspecial` fazendo as devidas renomeações.

```
package br.com.bytebank.banco.especial;  
  
import br.com.bytebank.banco.modelo.Conta;  
  
public class ContaEspecial extends Conta {  
  
    public ContaEspecial(int agencia, int numero) {  
        super(agencia, numero);  
    }  
}
```

```
@Override  
public void deposita(double valor) {  
    // TODO auto-generated method stub  
  
}  
}
```

[COPIAR CÓDIGO](#)

A implementação da classe `ContaEspecial` não é foco da explicação, e sim o modificador `protected`, então não vamos investir muito tempo nesse processo. De volta a classe `Conta`, lembre-se que retiramos o `protected` de `double saldo`, isso significa que na questão de visibilidade este atributo passa a ser `<<package private>>`, ou seja, apenas visível dentro de seu próprio pacote.

```
package br.com.bytebank.banco.modelo;  
  
public abstract class Conta {  
  
    double saldo;  
    private int agencia;  
    private int numero;  
    private Cliente titular;  
    private static int total = 0;
```

[COPIAR CÓDIGO](#)

A classe `ContaEspecial`, mesmo que sendo uma filha da classe `Conta`, não possui acesso ao atributo `saldo`, pois este não se encontra visível fora de seu pacote. As duas classes possuem relacionamentos íntimos, sendo classe `Conta` a mãe e `ContaEspecial` a filha. O modificador de visibilidade `protected` torna elementos públicos para as **classes filhas**.

Mesmo que a classe filha esteja em outro pacote, o atributo `saldo` em `Conta` será visível caso seja acompanhado por `protected`. O acesso ao atributo `saldo` da classe `Conta` não é possível de nenhum outro local, apenas da classe filha.

Caso criemos em `TesteContas` uma `ContaEspecial` e chamássemos seu `saldo`, o código não seria compilado e o Eclipse sugere utilizar o `getSaldo()`.

```
package br.com.bytebank.banco.test;

import br.com.bytebank.banco.especial.ContaEspecial;

public class TesteContas {

    public static void main(String[] args) throws SaldoInsuficiente
        ContaEspecial ce = new ContaEspecial(123, 555);
        ce.saldo
    //...
}
```

**COPIAR CÓDIGO**

Com isso, aprendemos sobre os modificadores de acesso/visibilidade, e temos que:

- **public:** visível em todos os espaços
- **protected:** visível dentro do pacote e público para os filhos
- **package private:** visível apenas dentro do pacote
- **private:** visível apenas dentro da classe



01

## Conhecendo o Javadoc

### Transcrição

A partir deste ponto o curso se tornará mais simples, pois a base da linguagem e seus conceitos nós já aprendemos; falamos sobre orientação básica, polimorfismo, exceções e pacotes. O que veremos será mais específico, como funcionalidades de algumas classes, ou seja, quais classes e pacotes o Java irá nos oferecer.

Para entendermos melhor como instrumentalizar algumas classes, começaremos a analisar a documentação Java. Digamos que seja necessário passar nosso projeto atual para outra equipe, para que essa transferência de projeto se dê da melhor forma precisamos nos preocupar com a documentação do código, o mínimo para ajudar os outros desenvolvedores. Precisamos, ainda, pensar em um modo de repassar todas as classes do projeto para a outra equipe da maneira mais simples o possível.

No caso da documentação, o Java oferece um recurso específico. Reparem na classe `SaldoInsuficienteException`:

```
package br.com.bytebank.banco.modelo;

public class SaldoInsuficienteException extends Exception{

    public SaldoInsuficienteException(String msg) {
        super(msg);
```

```
}
```

[COPIAR CÓDIGO](#)

A exceção que criamos estende uma classe do mundo Java, a `Exception`. Ao selecionarmos essa classe veremos uma série de comentários especiais:

```
/**  
 *The class {@code Exception} and its subclasses are a  
 *form of {@code Throwable} that indicates conditions that a reasonable  
 *application might want to catch.  
 *  
 *  
 *<p>The class {@code Exception} and any subclasses that are not also  
 *subclasses of {@link RuntimeException} are <em>checked  
 *exceptions</em>. Checked exceptions need to be declared in a  
 *method or constructor's {@code throws} clause if they can be thrown  
 *by the execution of the method or constructor and propagate outside  
 *the method or constructor boundary.  
  
 *@author Frank Yellin  
 *@see java.lang.Error  
 *@jls 11.2 Compile-Time Checking of Exceptions  
 *@since 1.0  
 */  
public class Exception extends Throwable {  
    static final long serialVersionUID = -3387516993124229948L
```

[COPIAR CÓDIGO](#)

Estes comentários exibidos pelo Eclipse são textos da documentação oficial Java. Reparem que esses comentários são azuis, quando realizamos nossos comentários.

no código utilizando as barras ( // ), o texto surge em cor verde, o que significa que ele não faz parte da documentação oficial.

Para escrevermos um comentário em azul utilizamos uma `/**` e fechamos com `*/`. Nesse espaço existe uma sintaxe específica, como `@author`, que indica quem escreveu o comentário. Existem ainda outros, como `@version` que indicam a versão.

```
/**  
 *  
 *@author Nico Steppat  
 *@version 0.1  
 */
```

[COPIAR CÓDIGO](#)

Normalmente os comentários especiais possuem algum valor importante dentro do código. Por exemplo, especificar a função de alguma classe ou fornecer informações de valor acerca de algum elemento, como faz a classe `Exception`.

```
/**  
 * Classe que representa um cliente no Bytebank.  
 *  
 *@author Nico Steppat  
 *@version 0.1  
 */
```

[COPIAR CÓDIGO](#)

Faremos um comentário especial em nossa classe `Conta`.

```
package br.com.bytebank.banco.modelo;  
  
/**
```

```
* Classe representa a moldura de uma conta  
*  
*@author Nico Steppat  
*  
*/  
public abstract class Conta {
```

&lt;.....!.....&gt;

[COPIAR CÓDIGO](#)

Os comentários podem ser escritos logo acima da classe referida, mas podem estar presentes em qualquer outro membro público. Lembrando: queremos passar as classes para um outro desenvolvedor e ele só enxergará nessa classe os membros públicos. Podemos deixar comentários que se referem a atributos privados, mas isso não terá nenhum significado para o desenvolvedor, já que ele não verá esse atributo.

Seguiremos fazendo comentários em nosso código para facilitar o trabalho dos outros desenvolvedores. Deixaremos um comentário no construtor da classe `Conta`.

```
/**  
 * Construtor para inicializar o objeto a partir da agencia e numero  
*  
*@param agencia  
*@param numero  
*/  
public Conta(int agencia, int numero){  
    Conta.total++;
```

&lt;.....!.....&gt;

[COPIAR CÓDIGO](#)

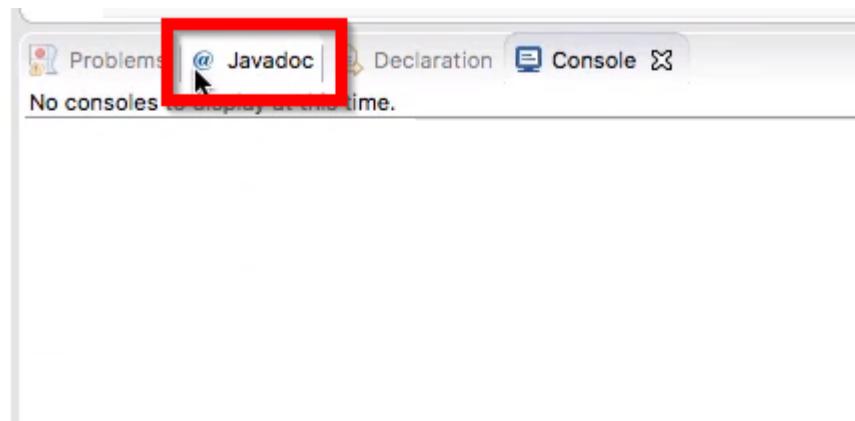
Faremos mais um comentário para apontarmos a exceção que o método joga.

```
/**  
 *  
 *  
 *@param valor  
 *@throws SaldoInsuficienteException  
 */  
public void saca(double valor) throws SaldoInsuficienteException{  
  
<....!....>
```

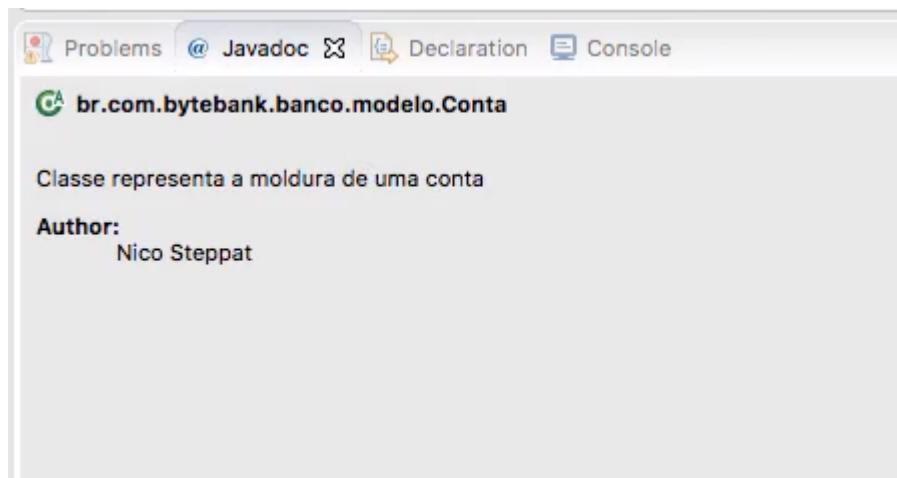
[COPIAR CÓDIGO](#)



Veremos a seguir como gerar a documentação, o chamado **Javadoc**. Há um botão disponível na área de console.

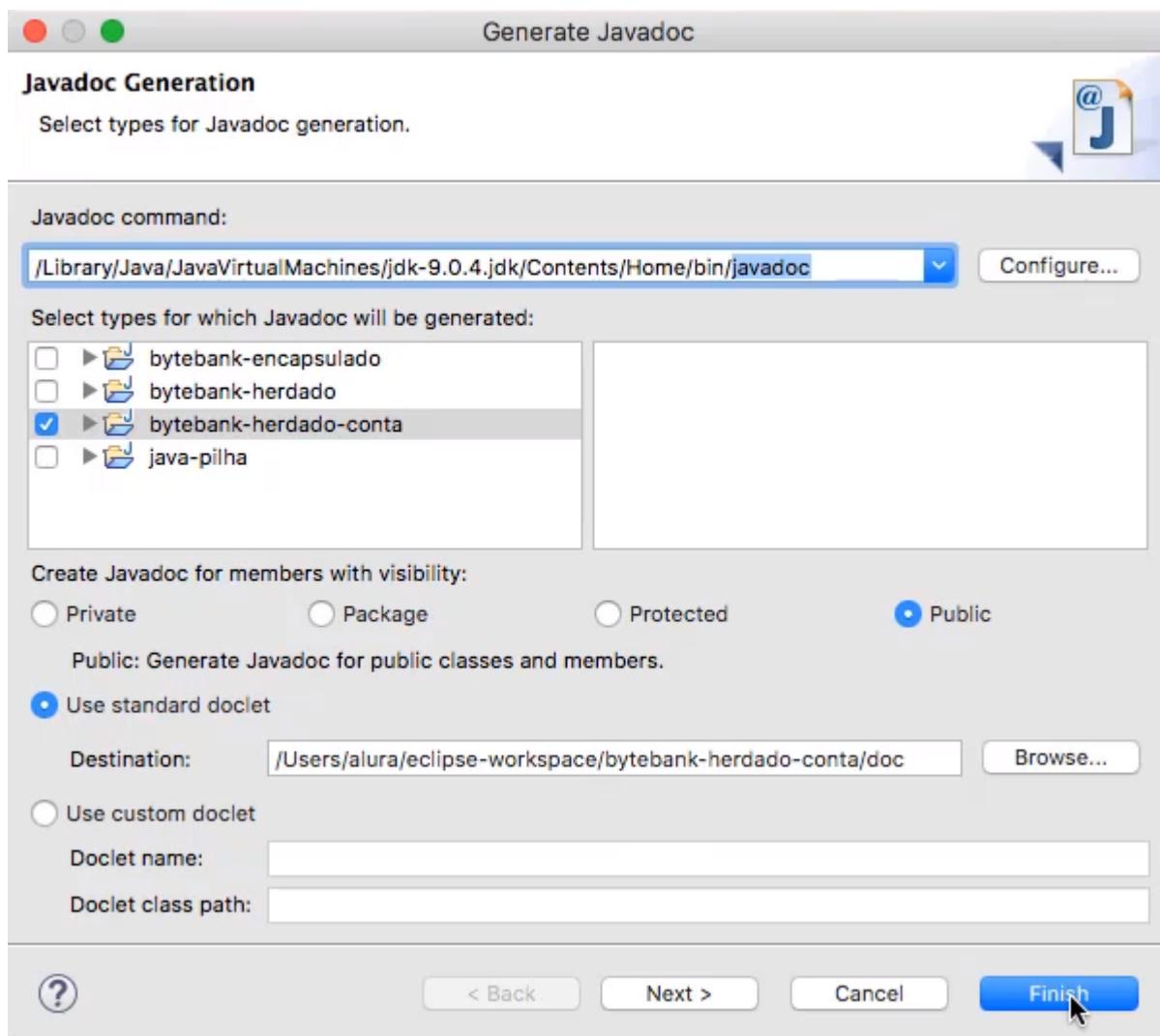


Caso essa opção não esteja disponível na área de console, basta selecionar o buscador na parte superior direta da tela do Eclipse e procurar por "Javadoc". Depois de acionarmos o Javadoc, ao clicarmos na classe os comentários são exibidos no console de forma mais organizada.

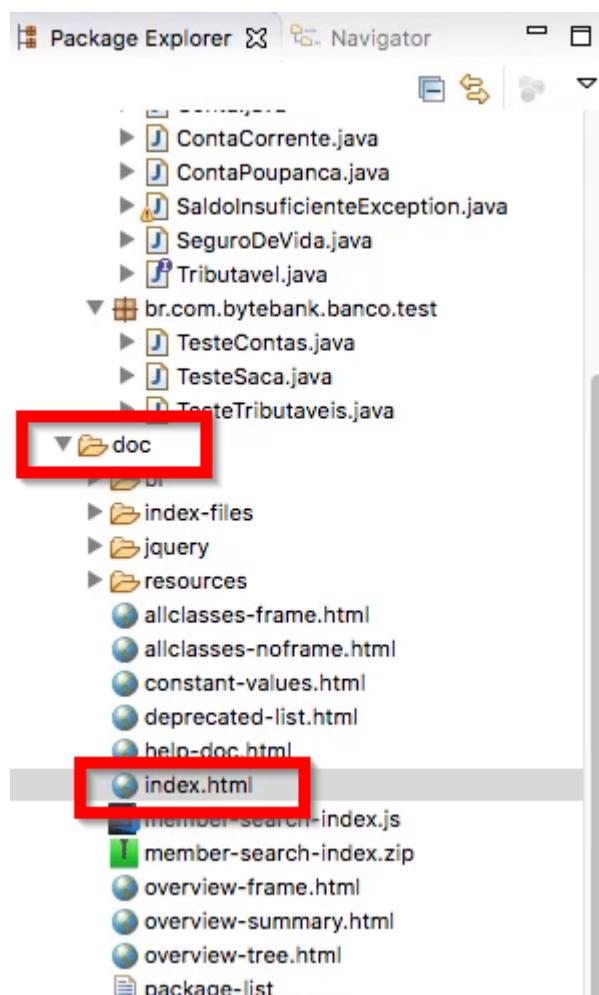


Essa seria um *preview* da documentação oficial do Java. A ideia é que a partir dos comentários elaborados, possamos criar um documento separado. Para isso, no cabeçalho do Eclipse, clicaremos em "Project -> Generate Javadoc".

Na caixa de diálogo, selecionaremos o projeto desejado, neste caso, bytebank-herdado-conta . É importante dizer que esta ferramenta de Javadoc vem junto com o JDK, portanto é crucial te-lo instalado. Salvaremos o projeto na pasta padrão.



Ao darmos o "Ok", serão percorridas todas as classes e será criada uma página HTML que contém todas as informações do projeto. Foi criada uma pasta `doc`, e vários arquivos a partir dessa varredura de classes, mas o que nos interessa neste momento é o `index`



Veremos, justamente, o índice da documentação com todos os pacotes, são eles

`br.com.bybank.banco.especial`, `br.com.bybank.banco.modelo` e

`br.com.bybank.banco.test`.

The screenshot shows a web browser displaying the JavaDoc for the `br.com.bytebank.banco.modelo` package. The URL in the address bar is `file:///Users/alura/eclipse-workspace/bytbank-herdado-conta/doc/overview-summary.html`. The browser interface includes a navigation bar with buttons for back, forward, search, and refresh, and a menu bar with links like OVERVIEW, PACKAGE, CLASS, USE, TREE, DEPRECATED, INDEX, and HELP. Below the menu is a search bar labeled "SEARCH: Search". The main content area is titled "Packages" and lists the following packages:

Package	Description
<code>br.com.bytebank.banco.especial</code>	
<code>br.com.bytebank.banco.modelo</code>	
<code>br.com.bytebank.banco.test</code>	

At the bottom of the page is a navigation bar with links for OVERVIEW, PACKAGE, CLASS, USE, TREE, DEPRECATED, INDEX, and HELP.

Ao selecionarmos um deles, no caso `br.com.bytebank.banco.modelo`, teremos acesso sumário do pacote com todas as interfaces, classes e exceções e os comentários

realizados.

The screenshot shows the Java Javadoc interface with the following navigation bar:

OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP  
PREV PACKAGE NEXT PACKAGE FRAMES NO FRAMES ALL CLASSES

**Interface Summary**

Interface	Description
Tributavel	

**Class Summary**

Class	Description
CalculadorDeImposto	
Cliente	Classe que representa um cliente no Bytebank.
Conta	Classe representa a moldura de uma conta
ContaCorrente	
ContaPoupanca	

No sumário de classes, podemos selecionar alguma classe específica para obtermos mais informações. Selecionaremos a classe `Cliente`.

Veremos o comentário que elaboramos ( `Classe que representa um cliente no Bytebank` ), a versão e o autor. Também veremos seus construtores, e seus métodos públicos.

The screenshot shows the Java Javadoc interface. The navigation bar at the top includes links for OVERVIEW, PACKAGE, CLASS (which is highlighted in orange), USE, TREE, DEPRECATED, INDEX, and HELP. Below the navigation bar are links for PREV CLASS, NEXT CLASS, FRAMES, NO FRAMES, and ALL CLASSES. At the bottom of this section are links for SUMMARY: NESTED | FIELD | CONSTR | METHOD and DETAIL: FIELD | CONSTR | METHOD. The main content area displays the class hierarchy: java.lang.Object and br.com.bytebank.banco.modelo.Cliente. Below this, the source code for the Cliente class is shown:

```
public class Cliente
extends java.lang.Object
```

A descriptive text follows: "Classe que representa um cliente no Bytebank."

Below the class description, there are sections for Version (0.1) and Author (Nico Steppat). A button labeled "Constructor Summary" is visible.

Voltaremos nossa atenção para a classe Conta .

Percebiam que são exibidas diversas informações, inclusive suas classes filhas, que são ContaCorrente , ContaEspecial e ContaPoupanca .

OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP  
PREV CLASS NEXT CLASS FRAMES NO FRAMES ALL CLASSES  
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

**Package** br.com.bytebank.banco.modelo

## Class Conta

java.lang.Object  
br.com.bytebank.banco.modelo.Conta

**Direct Known Subclasses:**

ContaCorrente, ContaEspecial, ContaPoupanca

---

```
public abstract class Conta
extends java.lang.Object
```

Classe representa a moldura de uma conta

**Author:**

Nico Steppat

São exibidas ainda as mesmas informações que vimos na classe Cliente , como construtores e métodos.

Notem que estamos lidando com uma documentação bem profissional, de desenvolvedor para desenvolvedor, e não para um usuário final que está fazendo uso da aplicação.

O primeiro passo para repassar um projeto é justamente gerar essa documentação. O Java iniciou a proposta de incluir a documentação dentro do código, como vimos nos comentários especiais. Se você realiza alguma modificação, as atualizações na documentação podem ser realizadas facilmente.

Nos falta pensar em como podemos passar todas as classes para a outra equipe de desenvolvedores de forma inteligente.



 04

## Para saber mais: Todas as tags

Já vimos nessa aula algumas tags (ou anotações) do *javadoc* como `@version` ou `@author`. Segue a lista completa:

- `@author` (usado na classe ou interface)
- `@version` (usado na classe ou interface)
- `@param` (usado no método e construtor)
- `@return` (usado apenas no método)
- `@exception` ou `@throws` (no método ou construtor)
- `@see`
- `@since`
- `@serial`
- `@deprecated`

Importante é que as tags do *javadoc* existem apenas para padronizar alguns dados fundamentais do seu código fonte como o autor e a versão.

## Outras anotações

Nos cursos você também já viu uma anotação fora do *javadoc*, a anotação `@Override`. Essa anotação é considerada uma configuração, nesse caso interpretado pelo compilador.

As anotações vão muito além das tags *javadoc* e são muito mais sofisticadas e poderosas. Elas só entraram na plataforma Java a partir da versão 1.5 enquanto

*javadoc* está presente desde o nascimento da plataforma Java. O interessante é que as anotações foram inspirados pelas tags do *javadoc*.

Se você ainda não está seguro sobre o uso das anotações, fique tranquilo pois você verá ainda muitas usadas pelas bibliotecas por ai para definir dados e configurações. Aguarde!

05

## Criando uma biblioteca com JAR

### Transcrição

Seguindo de onde paramos anteriormente, nosso próximo objetivo é repassar todas as classes para uma nova equipe.

Em um primeiro momento nos parece que basta acionar o atalho "Ctrl + C" e "Ctrl + V" em todas as classes compiladas e trabalho concluído. No entanto, essa não é a forma mais inteligente de distribuir o meu código.

A ideia é que passemos para o desenvolvedor o código compilado dentro de um arquivo ".zip", e a documentação, afinal, ele não necessariamente precisa do código fonte, que é de nossa responsabilidade.

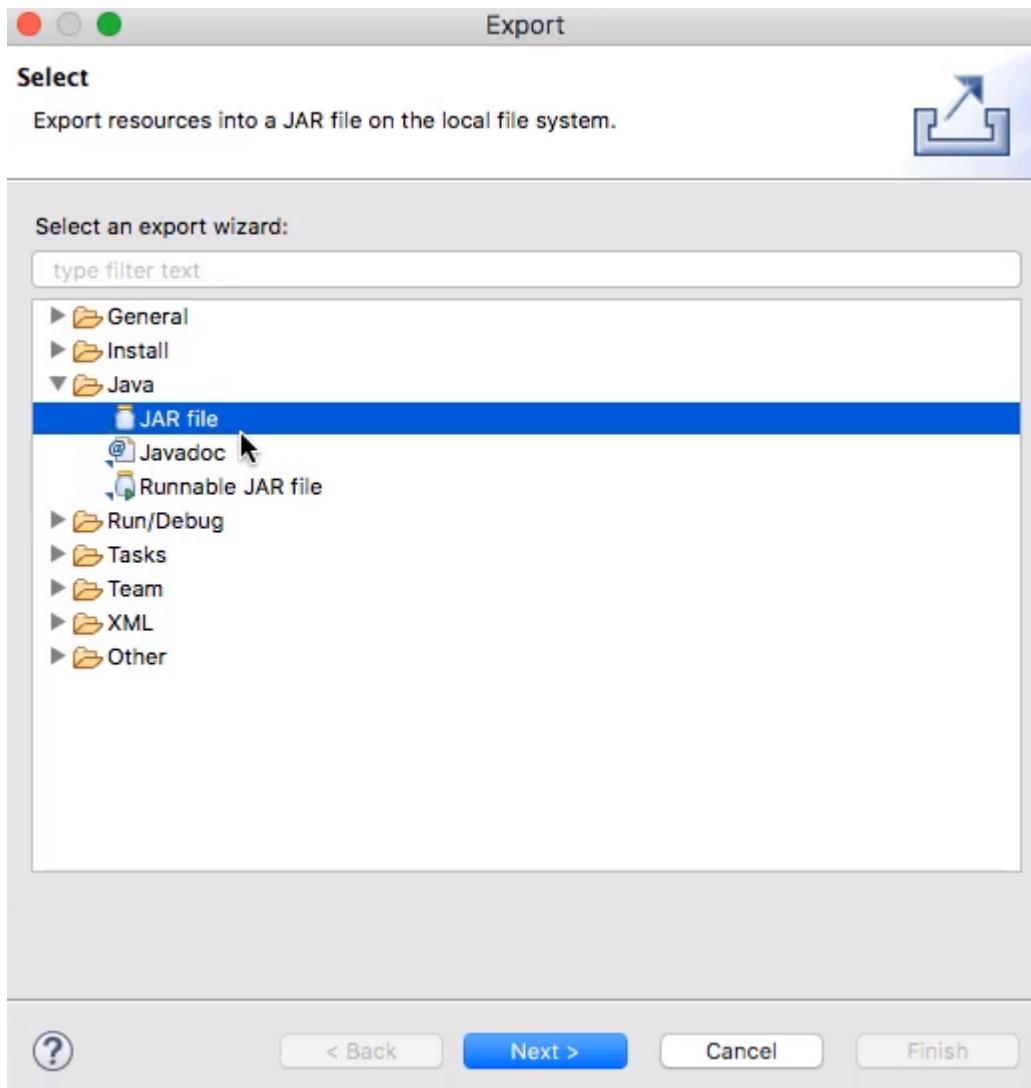
No entanto, o Java não chama esse tipo de arquivo de código compilado de ".zip", o tipo do arquivo na verdade é "JAR".

Poderíamos criar esse arquivo ".jar" manualmente, manipulando todo o conteúdo através do visualizador "Navigator". Porém, o Eclipse nos fornece facilidades para esse tipo de ação.

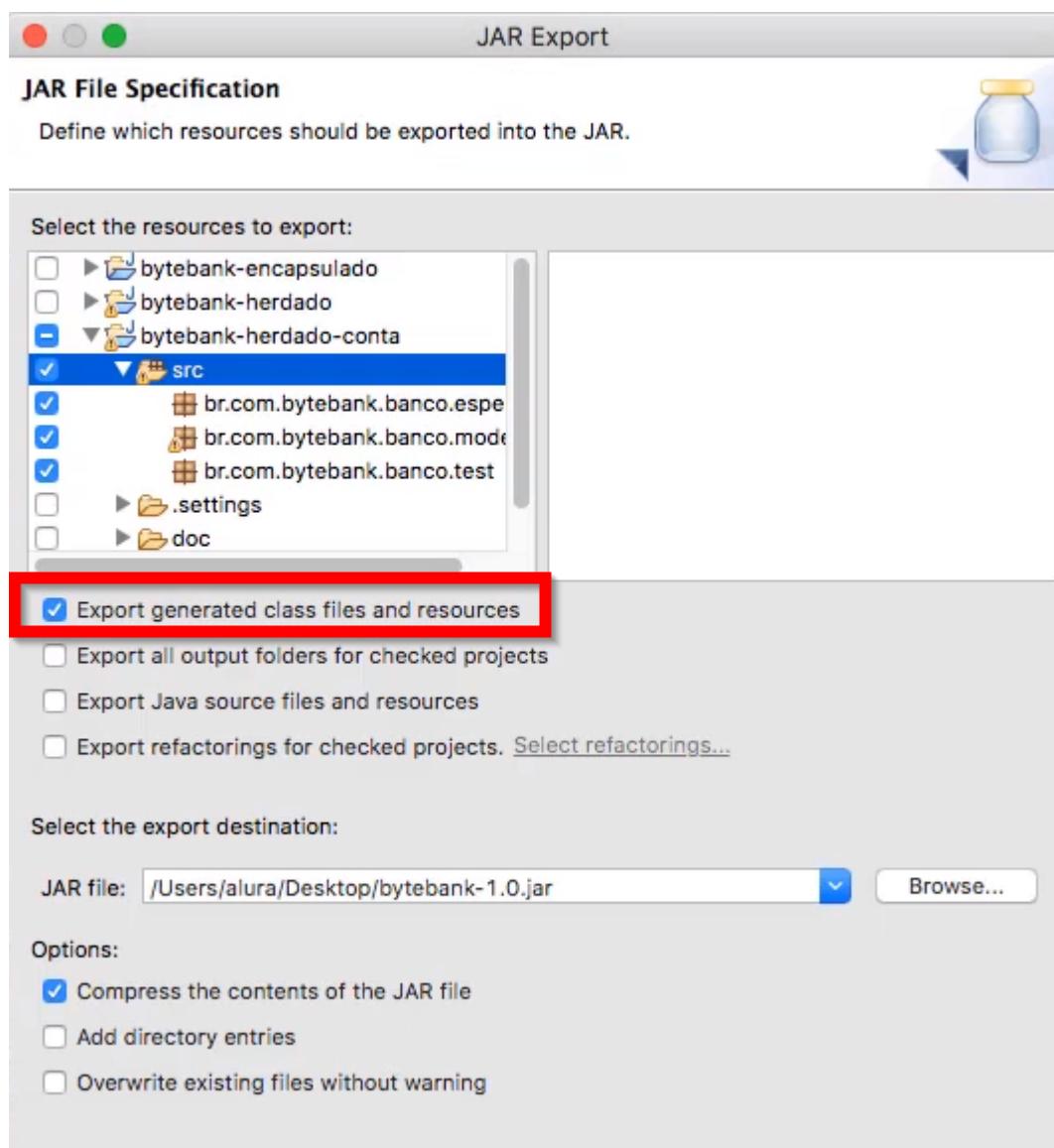
Na área "Package Explorer", selecionaremos nosso projeto `bytebank-herdado-conta`, clicaremos com o botão direito do mouse e selecionaremos a opção "Export".

Na caixa de diálogo que será aberta, veremos que há vários modos de exportar nosso projeto. Como queremos apenas o código compilado, selecionaremos a opção "JA

file" e em seguida "next" para avançarmos na exportação.



Na nova caixa de diálogo, selecionaremos as pastas que serão exportadas. Não exportaremos os arquivos internos do Eclipse ( .classpath e .project ), e sim, todo o conteúdo da pasta `src`. Lembrando que não estamos exportando o código fonte, apenas o **código compilado**, que o Eclipse denomina "class files". Exportaremos o código para o desktop com o nome de `bytebank-1.0.jar`.



Temos o arquivo JAR, que atua basicamente como um ".zip", mas em uma extensão diferente. Ao extrairmos os arquivos, veremos que todas as classes estarão presentes dentro da estrutura de pacotes.

Neste ponto do curso, já sabemos como exportar nosso projeto de forma inteligente e organizada. Como próximo passo, analisaremos o processo do ponto de vista da equipe que recebe o projeto.

No Eclipse, fecharemos todas classes que estão abertas através do atalho "Ctrl + Shift + W". Criaremos um novo projeto que chamaremos de `bytebank-biblioteca`. Para que não haja nenhuma confusão, fecharemos todos os outros projetos.

Clicaremos com o botão direito sobre `bytebank-biblioteca` e selecionaremos a opção "Close Unrelated Projects".

Normalmente, criamos uma pasta que armazenará todas as bibliotecas a serem utilizadas. Não existe um nome de pasta padrão, mas é muito comum utilizarmos o nome `libs`, que remete ao termo em inglês *library*. Nesta pasta arrastaremos nosso arquivo `bytebank-1.0.jar`.

O arquivo já faz parte do nosso projeto. O próximo passo é criar uma nova classe que chamaremos de `TesteBiblioteca`, que estará inserida em um novo pacote que nomearemos como `br.com.alura.bytebank`.

Observemos o código da nossa classe:

```
package br.com.alura.bytebank

public class TesteBiblioteca {

    public static void main(String[] args) {

    }

}
```

[COPIAR CÓDIGO](#)

Queremos utilizar a classe `Conta`.

```
package br.com.alura.bytebank

public class TesteBiblioteca {

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

```
    Conta c = new ContaCorrente(123, 321);  
}  
}
```

**COPIAR CÓDIGO**

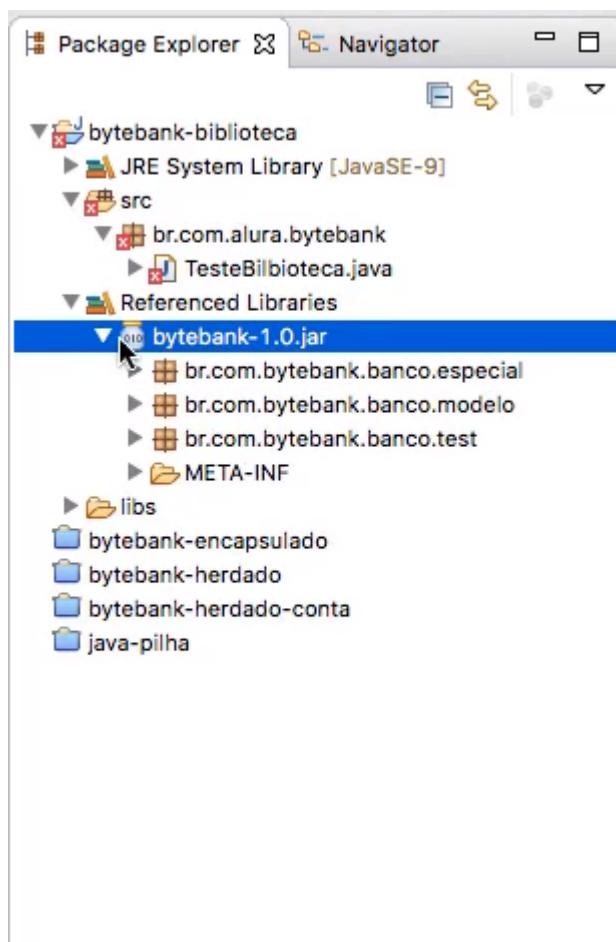
Percebiam que utilizamos os nomes simples das classes, o que gera erros na compilação do código. Precisamos utilizar o *full qualified name* neste caso, portanto, precisamos importar estas classes.

O Eclipse, nestes casos, sugere uma importação das classes. No entanto, essa opção não é mostrada, a sugestão é que se cria uma classe `Conta`, muito embora essa classe já exista dentro do arquivo `bytebank-1.0.jar`.

O que precisamos realizar é uma comunicação do Eclipse com os arquivos do projeto. Para isso existe uma configuração que ainda não realizamos, simplesmente copiamos o arquivo ".jar" para a ferramenta e isso não é o suficiente para que as classes se tornem visíveis e usuais.

Para realizarmos essa configuração, selecionamos o arquivo `bytebank-1.0.jar` na área do Package Explorer e pressionamos o botão direito do mouse, e selecionamos as opções "Build Path > Add to Build Path".

Assim feito, veremos que uma representação gráfica de jarra surge ao lado do nome do arquivo, apontado como `Referenced Libraries`, e dentro dele são exibidas todas os pacotes, que por sua vez, armazenam as classes.



Com isso, voltando ao código de `TesteBiblioteca` o Eclipse sugere a importação das classes `Conta` e `ContaCorrente`, afinal, elas estão acessíveis.

Faremos um pequeno teste para avaliar se o nosso código está de fato funcional, acionando o método `deposita()`.

```
package br.com.alura.bytebank

public class TesteBiblioteca {

    public static void main(String[] args) {

        Conta c = new ContaCorrente(123, 321);

        c.deposita(200.3);
    }
}
```

```
System.out.println(c.getSaldo());  
}  
}
```

[COPIAR CÓDIGO](#)

Tudo opera como o esperado.

Nesta aula aprendemos duas ações: primeiramente como criar a documentação dentro do código fonte e gerar os HTMLs organizados, depois, aprendemos como criar uma biblioteca que seja funcional para os desenvolvedores que recebem o projeto através de um arquivo ".jar".

O mundo Java possui uma série de ".jar"s, caso você queira criar um gráfico ou abrir uma conexão com um banco de dados, sempre haverá um ".jar" ou mais que auxiliarão em seu trabalho.

## JAR executável (Opcional)

### Transcrição

Nesta aula, iremos nos aprofundar na questão das bibliotecas jar.

A forma como criamos um arquivo jar em nosso projeto será a mais comum quando utilizamos esse recurso.

Há na internet alguns repositórios de ".jar" que podem ser baixados; existem, ainda, ferramentas que podem nos auxiliar na plena utilização de ".jar", como quantas bibliotecas deste tipo precisaremos em um determinado projeto.

Há, também, ferramentas que ajudam a gerenciar as dependências que uma biblioteca ".jar" pode vir a ter, como Maven; na Alura temos [um curso](https://cursos.alura.com.br/course/maven-build-do-zero-a-web) (<https://cursos.alura.com.br/course/maven-build-do-zero-a-web>) dedicado a esta ferramenta. Quem utiliza o .NET sabe o que o Visual Studio possui um gerenciador de dependências integrado, o que não ocorre com o Eclipse.

No entanto, existem outras possíveis aplicações para o ".jar". Na Alura, temos uma série de pastas com o nome dos professores e seus respectivos cursos. Juntamente com estas pastas temos o um arquivo ".jar" chamado `revisor-beta.jar`. Este ".jar" não foi pensado para ser uma biblioteca e ser usado através do desenvolvedor, mas sim uma ferramenta disponível para que os instrutores possam executar códigos e revisar a nomenclatura dos vídeos. Ou seja, este ".jar" foi pensado para o **usuário final** e não para o desenvolvedor.

Veremos rapidamente como esse ".jar" voltado para o usuário funciona; primeiramente, acessaremos o terminal e buscaremos por revisor-beta.jar

```
Last login: Fri Mar 9 11:01:12 on console
Aluras - iMac:~ alura$ cd /Volumes/Dados_MAC/
Aluras - iMac: DADOS_MAC alura$ ls
danilo-maximo      flavio-almeida      nico steppat
fabio-chaves       leonardo-codeiro    revisor-beta.jar
Aluras-iMac:DADOS_MAC alura$
```

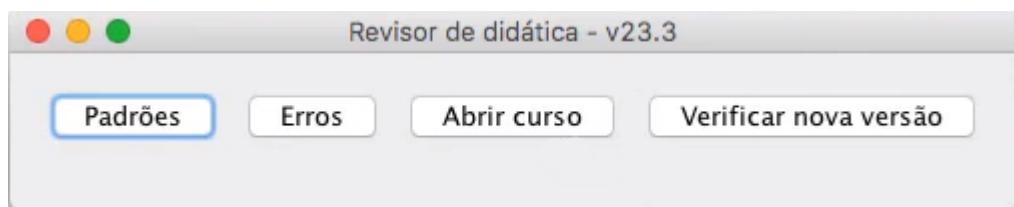
**COPIAR CÓDIGO**

Para que possamos executar o ".jar", precisamos utilizar a máquina virtual. Não iremos copiar esse ".jar" e colá-lo no Eclipse, pois trata-se de uma aplicativo para o usuário final.

```
Last login: Fri Mar 9 11:01:12 on console
Aluras - iMac:~ alura$ cd /Volumes/Dados_MAC/
Aluras - iMac: DADOS_MAC alura$ ls
danilo-maximo      flavio-almeida      nico steppat
fabio-chaves       leonardo-codeiro    revisor-beta.jar
Aluras-iMac:DADOS_MAC alura$ java -jar revisor-beta.jar
```

**COPIAR CÓDIGO**

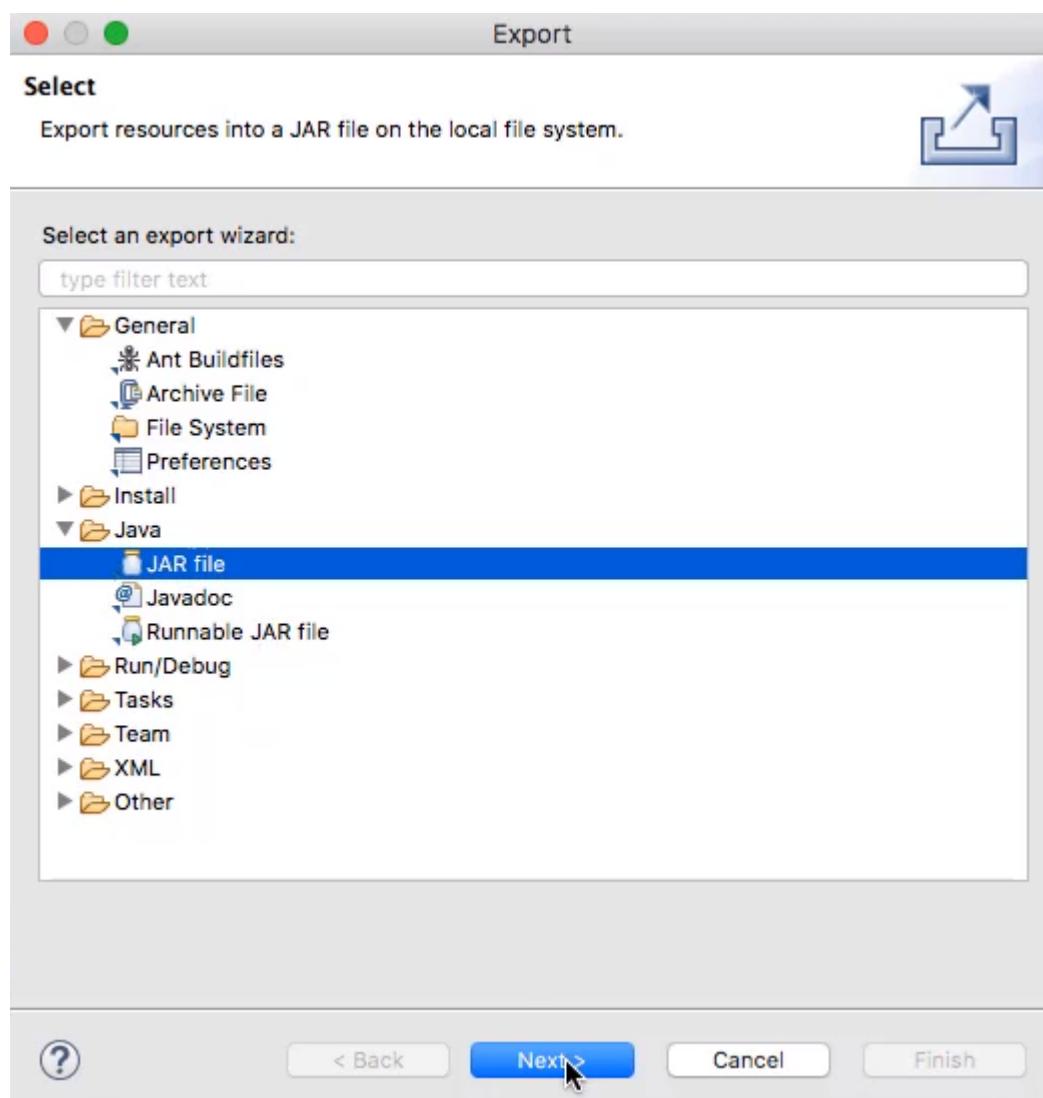
Ao executarmos o ".jar", percebam que há uma interface gráfica, portanto existem classes no mundo Java que possibilitam a construção de uma caixa de diálogo com botões.



Faremos algo parecido para a nossa aplicação. Não criaremos uma janela com botões, pois isso demandaria outro curso que se articule, mas criaremos um ".jar" que possa ser executado.

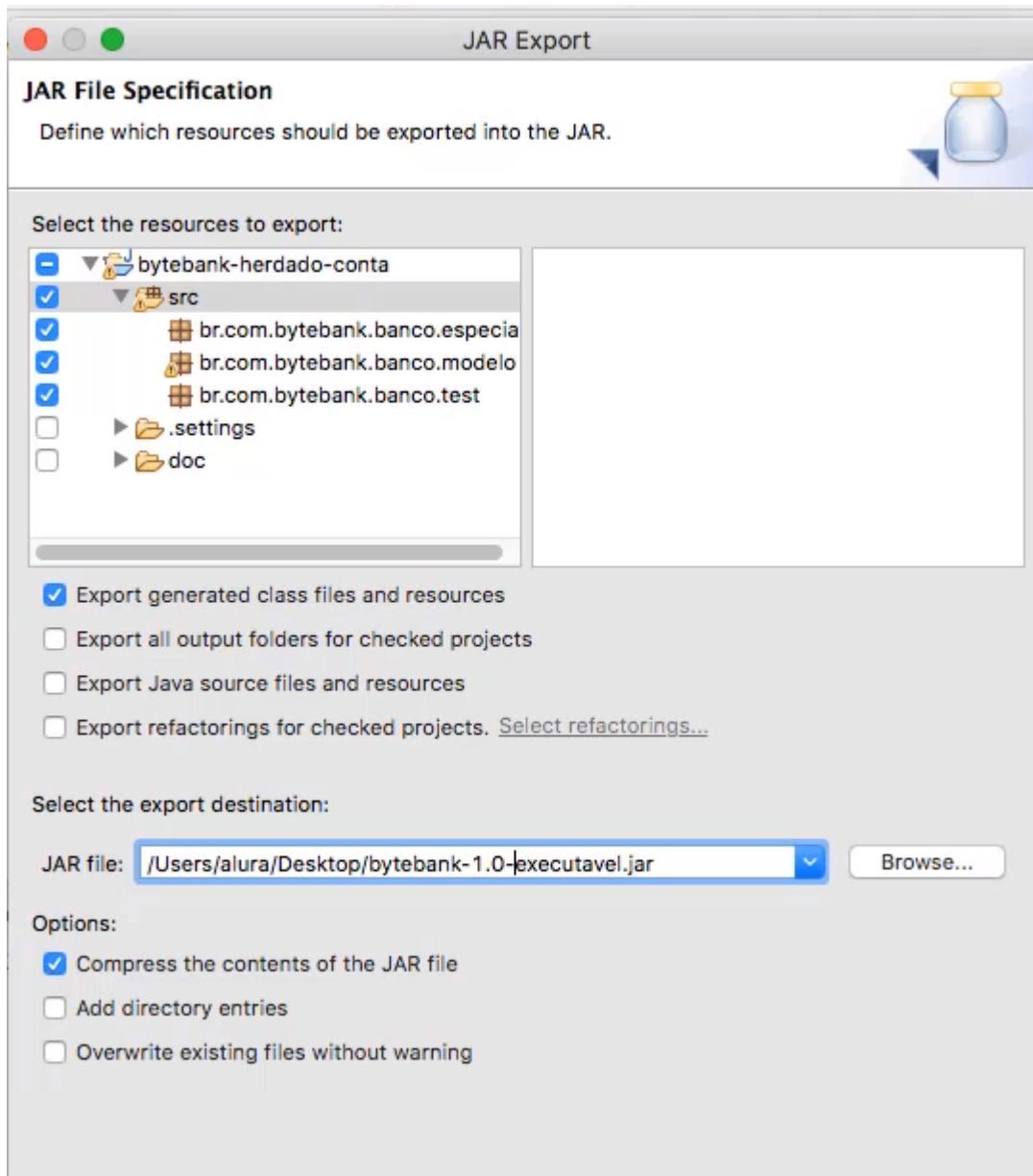
De volta ao Eclipse, abriremos novamente o projeto `bytebank-herdado-conta` e novamente criaremos um ".jar", mas desse vez com uma finalidade diferente: a aplicação utilizada pelo cliente.

Com o projeto `bytebank-herdado-conta` selecionado, pressionaremos o botão direito e escolheremos a opção "Export". Na caixa de diálogo aberta, selecionaremos o JAR file dentro da pasta Java .



Como já sabemos, não iremos exportar os arquivos de configuração do Eclipse.

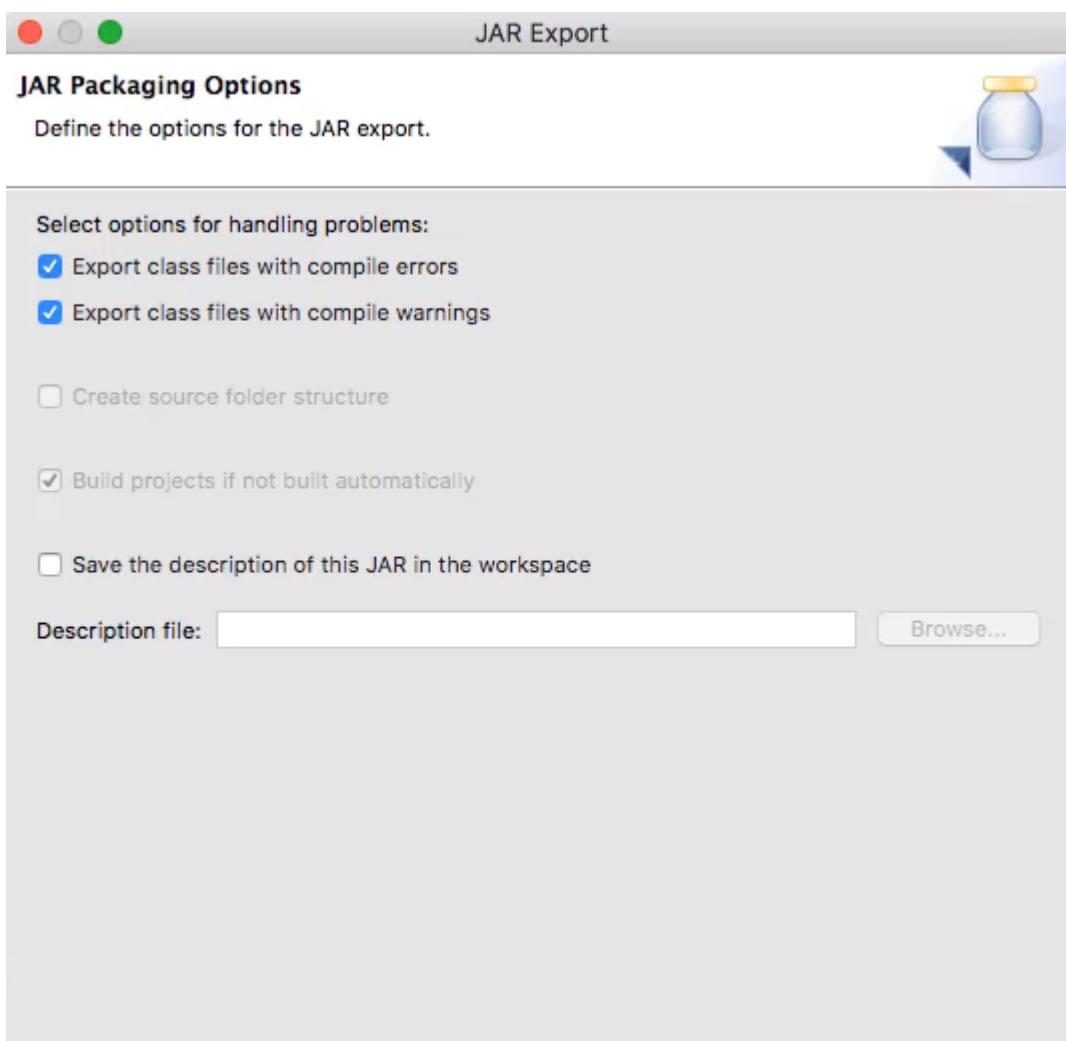
Selecionaremos a pasta `src` e os três pacotes. A documentação (`doc`) também não será exportada. Modificaremos o nome do arquivo `".jar"` para `bytebank-1.0-executavel`.



Assim feito, pressionaremos o botão "Next" para prosseguirmos com as configurações de exportação.

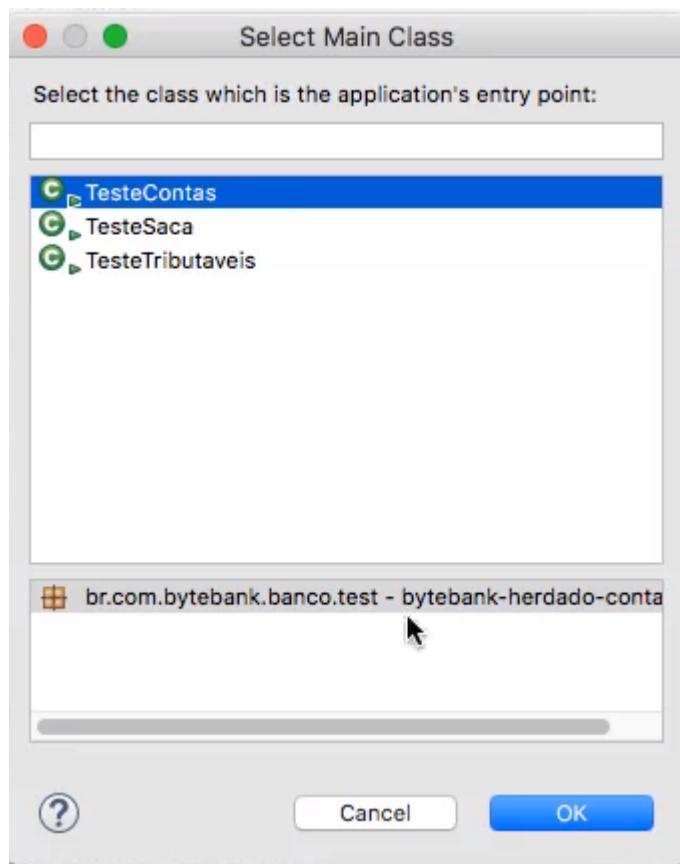
Veremos uma nova caixa de diálogo que apresenta opções a serem selecionadas, são elas "Export class files with compile errors" e "Export class files with compile

warnings". As duas opções devem ser selecionadas, pressionaremos "Next".



Na próxima caixa de diálogo, veremos que na parte inferior existe o campo "Select the class of the application entry point". Nós deveríamos direcionar a entrada da aplicação; qualquer aplicação Java sendo executada mediante um ".jar" se inicia por um método `main()`, e esse método está dentro de uma classe, portanto precisamos definir de alguma forma qual será nossa classe inicial a ser enxergada pela máquina virtual como *entry point*.

Selecionaremos o botão "Browse" e serão exibidas as três classes que possuem o método `main()` no projeto, são elas `TesteContas`, `TesteSaca` e `TesteTributaveis`. Selecionaremos `TesteContas`, pois essa classe contém o `System.out.println()`, o que significa que teremos uma saída para verificar se a execução da aplicação funcionou.



A entrada da aplicação ou *entry point*, portanto, passou a ser

`br.com.bytebank.baco.test.TesteContas`.

Para executarmos o arquivo ".jar" `bytebank-1.0-executavel`, iremos até o terminal e acionaremos o comando `java -jar`.

```
Last login: Fri Mar 9 11:01:12 on console
Aluras - iMac:~ alura$ cd /Volumes/Dados_MAC/
Aluras - iMac: DADOS_MAC alura$ ls
danilo-maximo      flavio-almeida      nico steppat
fabio-chaves       leonardo-codeiro    revisor-beta.jar
Aluras-iMac:DADOS_MAC alura$ java -jar revisor-beta.jar
Alura -iMac:DADDOS_MAC alura$ cd
Alura- iMac: alura$ pwd
/Users/alura
Aluras-iMac:~ alura$ cd Desktop/
Aluras-iMac: Desktop alura$ ls
```

Screen Shot 20-03-09 at 13-40-20-png desktop

bytebank-1.0-executavel.jar

Aluras-iMac:Desktop alura\$ java -jar bytebank-1.0-executavel.jar

[COPIAR CÓDIGO](#)

Dentro do ".jar" existe uma configuração que se comunica com a máquina virtual indicado que a classe `TesteContas` contém o método `main()`. Ao executarmos o ".jar", perceberemos que não existe uma janela de diálogo com botões, pois não desenvolvemos nenhum tipo de interface gráfica. São exibidos apenas os valores de saída da console.

CC:89.0

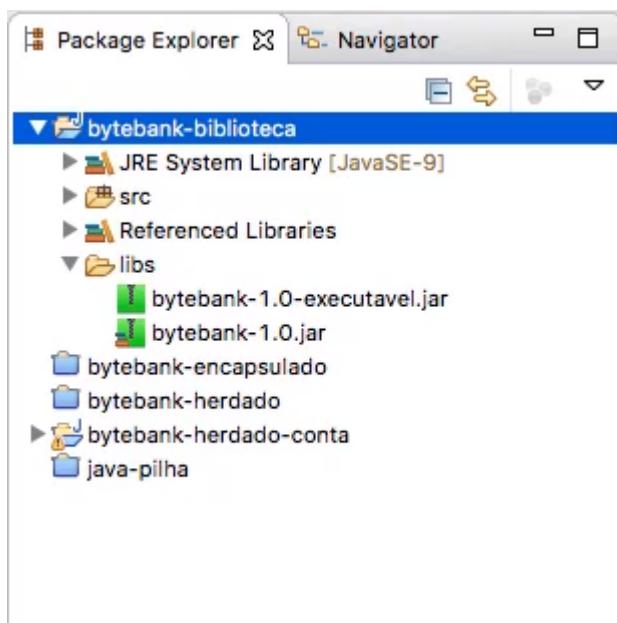
CP:210.0

[COPIAR CÓDIGO](#)

São os mesmos valores exibidos pelo Eclipse na linha de comando quando executamos a classe `TesteConta`.

Com isso, provamos que é possível criar um ".jar" voltado para o usuário final.

Faremos um pequeno teste: Transferiremos o arquivo ".jar" que acabamos de gerar (`bytebank-1.0-executavel`) para o projeto `bytebank-biblioteca`, na pasta `libs`.



Feito isso, selecionaremos o ".jar" e pressionaremos o botão direito e escolheremos as opções "Build Path > Add to Build Path", ou seja, adicionaremos esse ".jar" como se fosse uma biblioteca.

Ao abrirmos o arquivo, veremos os pacotes do projeto como o esperado, no entanto há uma pasta `META-INF`, uma pasta de configuração que contém um arquivo denominado `MANIFEST.MF`.

Dentro desse arquivo há um conteúdo muito simples, contém a versão do arquivo e seu `main class`.

`Manifest-Version: 1.0`

`Main-Class: br.com.bytebank.banco.test.TesteContas`

**COPIAR CÓDIGO**

Vemos como seria um ".jar" executável, que apresenta algumas diferenças. Nas próximas aulas falaremos de outras bibliotecas padrão do mundo Java.



 10

## Para saber mais: Maven

Java é uma plataforma de desenvolvimento completa que se destaca com sua grande quantidade de projetos *open source*. Para a maioria dos problemas no dia a dia do desenvolvedor já existem bibliotecas para resolver. Ou seja, se você gostaria de se conectar com um banco dados, ou trabalhar no desenvolvimento web, na área de data science, criação de serviços ou Android, já existem bibliotecas para tal, muitas vezes mais do que uma.

Aí existe a necessidade de organizar, centralizar e versionar os JARs dessa biblioteca e gerenciar as dependências entre elas. Para resolver isso, foram criadas ferramentas específicas e no mundo Java se destacou o Maven. O Maven organiza os JARs (código compilado, código fonte e documentação) em um repositório central que é público e pode ser pesquisado:

<https://mvnrepository.com/> (<https://mvnrepository.com/>)

Lá você pode ver e até baixar os JARs, mas o melhor é que a ferramenta Maven pode fazer isso para você. Se ficou interessado em aprender o Maven que ainda tem outros recursos bem legais, dá uma olhada no nosso curso específico:

[Maven: Build do zero a web](https://cursos.alura.com.br/course/maven-build-do-zero-a-web) (<https://cursos.alura.com.br/course/maven-build-do-zero-a-web>)

Obs: Se você é usuário Linux, o Maven é bem parecido com os gerenciadores `apt` ou `rpm`. No MacOS existe o `brew` com o mesmo propósito. No mundo .Net temos o `nuget` e a plataforma `node.js` usa `npm`. Gerenciar dependências é um problema do cotidiano do desenvolvedor, e cada sistema ou plataforma possui a sua solução.



01

## Conhecendo java.lang

### Transcrição

Nesta aula, falaremos sobre as classes fundamentais do mundo Java.

Faremos um breve resumo sobre os pacotes: aprenderemos que eles são úteis para organizarmos nossas classes e evitar conflito de nomes, uma vez que Java é uma plataforma muito popular e muitos desenvolvedores utilizam Java. Para isso, organizamos os diretórios em hierarquias, de forma que eles passam a fazer parte do nome da classe.

Para utilizarmos uma classe utilizamos o `import`, ou importação, caso isso não seja feito ela não será encontrada, ou então deveremos sempre qualifica-la com seu nome completo.

A classe não utiliza mais seu nome simples no sistema de pacotes, e sim o *full qualified name*, como o exemplo de `ContaCorrente` logo abaixo:

```
import br.com.bytebank.banco.modelo.ContaCorrente;
```

[COPIAR CÓDIGO](#)

A partir desse ponto vocês não encontrarão mais classes sem pacote, e nem irão escrever classes sem pacote, afinal esta não é uma boa prática e devemos trabalhar para que isso não ocorra.

Iremos nos atentar para uma questão importante, observem uma parte do código da classe TesteConta :

```
public class TesteContas {  
  
    public static void main(String[] args) throws SaldoInsuficiente  
  
        ContaCorrente cc = new ContaCorrente(111, 111);  
        cc.deposita(100.00);  
  
        ContaPoupança cp = new ContaPoupança(222, 222);  
        cp.deposita(200.0);  
  
        cc.transfere(10.0, cp);  
  
        System.out.println("CC: " + cc.getSaldo());  
        System.out.println("CP: " + cp.getSaldo());
```

[COPIAR CÓDIGO](#)

Percebam que estamos utilizando `String` e `System`, que também são classes. Falamos anteriormente que todas as classes estão dentro de um pacote, mas como lidamos com essas duas? Devemos utilizá-las a partir de um `import`, como com todas as outras classes, certo?

As classes `String` e `System` estão, sim, dentro de um pacote, mas não é necessário importá-las. O único pacote que não necessita ser importado é `java.lang`. Por ser de suma importância para o desenvolvimento de qualquer aplicação Java, ele é incluído automaticamente.

Ao observarmos o full qualified name da classe `String` nos deparamos com:

java.lang.String;

**COPiar CÓDIGO**

 03

## Exceções do java.lang

Quando falamos sobre exceções já vimos várias classes como `Exception`, `RuntimeException`, `NullPointerException` ou `ArithmetricException`.

Todas essas classes vem do pacote `java.lang` e por isso não era preciso importá-las.

04

## String e a imutabilidade

### Transcrição

O foco dessa aula será o pacote `java.lang`.

Começaremos com, talvez, a classe mais importante de todas: a `String`. Quanto mais conhecermos esta classe, mais fácil será o seu dia a dia enquanto desenvolvedor, pois trata-se de uma classe fundamental no desenvolvimento de projetos.

Criaremos uma classe para testarmos a `String`; o nome dessa classe será `TesteString`.

Nesta nova classe, criaremos uma `String` e definiremos seu nome como `Alura`

```
package br.com.bytebank.banco.test;

public class TesteString {

    public static void main(String[] args) {

        String nome = "Alura";
    }
}
```

[COPIAR CÓDIGO](#)

Até este ponto não há nada de novo; temos a classe `String` que pertence ao pacote `java.lang`, e `nome` é uma referência.

Ao observarmos a classe `TesteContas` veremos que `cc` é uma referência, `ContaCorrente` é uma classe que representa um tipo, e ao lado direito temos o objeto.

```
public static void main(String[] args) throws SaldoInsuficienteExce|  
    ContaCorrente cc = new ContaCorrente(111, 111);  
    cc.deposita(100.0);
```

[COPIAR CÓDIGO](#)

Ou seja, em `TesteString` também estamos criando um objeto, `String`s também são objetos. No entanto, não é preciso utilizar o `new` para criar este objeto. Isso ocorre para facilitar a vida do desenvolvedor, não precisamos utilizar o `new` todas as vezes que formos trabalhar com Strings, mas nada impede que você o faça. Portanto, as duas formas são funcionais, embora o segundo caso seja considerado uma má prática. A partir da primeira forma a máquina virtual consegue executar algumas otimizações, o que é impossível no segundo caso.

```
String nome = "Alura";  
String outro = new String("Alura");
```

[COPIAR CÓDIGO](#)

Chamamos essa sintaxe de **objeto literal**. Na sintaxe não há diferença entre inicializar um inteiro(`int`) e uma `String`. `nome` é uma referência, então nada nos impede de executar métodos em cima dela. Ao escrevermos no Eclipse `nome` veremos diversas sugestões de métodos, vale a pena estudar isso com calma, pois isso facilitará muito o seu dia a dia. A melhor forma de estudar é experimentando.

métodos, testando e conversando com outras pessoas, mas claro, existe uma forma mais oficial e tradicional de se realizar este estudo.

Nas aulas anteriores criamos um `Javadoc` para nosso projeto, da mesma forma os desenvolvedores oficiais do Java fizeram uma documentação que nos auxiliará nos estudos.

Procuraremos a documentação oficial do Java 9, muito embora você possa utilizar a da versão 8 ou 7, já que a classe `String` se modificou muito pouco. Veremos que a estética da documentação é idêntica ao `Javadoc` gerado por nós. Estamos analisando especificamente a [documentação](#)

(<https://docs.oracle.com/javase/9/docs/api/java/lang/String.html/>) sobre a `String`.

The screenshot shows the Java 9 API documentation for the `String` class. The top navigation bar includes links for OVERVIEW, MODULE, PACKAGE, CLASS (which is highlighted in orange), USE, TREE, DEPRECATED, INDEX, and HELP. Below the navigation bar are links for PREV CLASS, NEXT CLASS, FRAMES, NO FRAMES, and ALL CLASSES. The SUMMARY section lists NESTED, FIELD, CONSTR, and METHOD. The DETAIL section also lists FIELD, CONSTR, and METHOD. The **Module** is `java.base` and the **Package** is `java.lang`. The **Class String** is described as extending `java.lang.Object` and implementing `java.lang.String`. It also implements `Serializable`, `CharSequence`, and `Comparable<String>`. The source code for the `String` class is partially visible at the bottom.

```
public final class String
extends Object
```

Encontraremos muitas explicações a respeito da classe e uma detalhada lista sobre seus construtores e métodos, como por exemplo, o `replace()`, que faremos muito uso. Caso você tenha alguma dúvida acerca do funcionamento dos métodos, vale a pena consultar a documentação oficial Java.

Voltaremos a classe `TesteString`.

Testaremos o recém citado `replace()`, existem duas versões para este método e veremos suas diferenças. A ideia é substituir "A" por "a", ao final, faremos um `System.out` para imprimirmos `nome`.

```
package br.com.bytebank.banco.test;

public class TesteString {

    public static void main(String[] args) {

        String nome = "Alura";

        nome.replace("A", "a");

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

[COPIAR CÓDIGO](#)

Ao executarmos o código veremos que o `replace()` não foi bem sucedido, afinal o resultado impresso continua sendo `Alura`, com "A" maiúsculo. Faremos um outro teste lançando uso sobre o método `toLowerCase()`, que basicamente substitui todas as letras por minúsculas.

```
package br.com.bytebank.banco.test;

public class TesteString {

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

```
String nome = "Alura";  
  
    nome.replace("A", "a");  
  
    nome.toLowerCase();  
  
    System.out.println(nome);  
}  
}
```

[COPIAR CÓDIGO](#)

Mais uma vez o resultado impresso continua sendo `Alura`.

Conheceremos um conceito fundamental da `String`: uma vez que foi criada, ela não poderá ser modificada posteriormente. Chamamos o conceito de um objeto não poder ser alterado de **imutabilidade**. Caso você queira alterar algo em uma `String`, você terá de criar uma `String` que refletirá uma nova ação, ou seja, teremos dois objetos, como duas "Aluras", sendo uma com "A" e outra com "a".

Ao consultarmos a documentação oficial, veremos que o `replace()` retorna uma `String`.

```
replace String replace(char oldChar, char newChar)
```

[COPIAR CÓDIGO](#)

Se quisermos imprimir `alura`, deveremos criar uma `String` que chamaremos de `outra`, e que apontará para o objeto de nosso interesse.

```
package br.com.bytebank.banco.test;  
  
public class TesteString {
```

```
public static void main(String[] args) {  
  
    String nome = "Alura";  
  
    String outra = nome.replace("A", "a");  
  
    System.out.println(outra);  
}  
}
```

[COPIAR CÓDIGO](#)

Com isso, temos o resultado impresso de `alura`.

Na verdade, temos duas `String` na memória, sendo uma `nome` que aponta para o objeto `Alura` e a `String` `outra` que aponta para um novo objeto, no caso, `alura`.

Todos os métodos funcionam nessa linha: devolvem uma nova `String`, respeitando o conceito de imutabilidade. Esse é um conceito importante e algumas classes do nosso projeto seguem essa ideia.

Nem todas classes são imutáveis, a nossa classe `Conta` não é imutável, afinal a ideia de uma conta é necessariamente dinâmica, o saldo de uma conta está sempre em movimento e transformação.

06

## Metodos da classe String

### Transcrição

Nesta aula, faremos mais experimentos com a classe `String`, e conheceremos mais alguns métodos importantes. Lembrando que vale a pena explorar a documentação oficial do Java para estudar melhor o funcionamento da classe em detalhes.

Na última aula fizemos uso do método `toLowerCase()`, em oposição a ele temos o método `toUpperCase()` que converte as letras para maiúsculas.

Falamos, também, que existem duas versões do método `replace()`, e os dois variam nos parâmetros, ambos recebem dois, mas os tipos são diferentes. O conceito de métodos que variam nos tipos de parâmetros é chamado de **sobrecarga**, muito comum no mundo Java.

Veremos que um dos `replace()` recebe `CharSequence` e outro um `char`; o primeiro está relacionado a conversão de um conjunto de caracteres, por exemplo:

```
String outra = nome.replace("Al", "aL");
```

COPIAR CÓDIGO

A outra opção altera apenas um caractere. É importante lembrar que no mundo Java um único caractere é acompanhado de aspas simples.

```
String outra = nome.replace('A', 'a');
```

[COPIAR CÓDIGO](#)

Esse tipo é chamado `char`, ou seja, uma `String` que contém apenas um caractere. Observem um exemplo ainda mais claro:

```
char c = 'A'  
char d = 'a'  
  
String outra = nome.replace(c,d);
```

[COPIAR CÓDIGO](#)

Em seguida, conhceremos o método `charAt()`.

Suponhamos que você tenha uma `String` e queira saber qual é o caractere que ocupa uma posição específica. Faremos um exemplo com a posição `2`, e imprimiremos o resultado, observe:

```
String nome = "Alura";  
  
char c = nome.charAt(2);  
System.out.println(c);
```

[COPIAR CÓDIGO](#)

Percebam que o resultado impresso foi a letra `u`, o que pode parecer um pouco estranho, uma vez que a segunda letra da `String` `Alura` seria `1`. Por que isso ocorreu? O Java, como quase todas as linguagens, trabalha com o numeral inicial `"0"`, portanto:

A = 0  
L = 1

U = 2

R = 3

A = 4

[COPIAR CÓDIGO](#)

O próximo método que conheceremos é o `indexOf()`.

Esse método possui muitas variantes, recebendo diferentes parâmetros. Usaremos o método que recebe uma `String`. Para o método podemos passar uma sequencia de caracteres, passaremos `ur`. Assim feito, podemos perguntar qual é a posição dessa `String` dentro de `Alura`.

```
String nome = "Alura";  
  
int pos = nome.indexOf("ur");  
System.out.println(pos);
```

[COPIAR CÓDIGO](#)

Será impressa a posição 2. O método `indexOf()` opera de forma inversa ao `charAt()`, uma vez que aquele com base no caractere exibe sua posição, e este utiliza a posição para exibir o caractere.

Existe um método que possibilita a criação de uma sub-`String`, chamado apropriadamente de `substring()`. Este método possui duas sobrecargas. A primeira informação que vamos fornecer é a partir de que posição iniciamos a contagem; colocaremos 1.

```
String nome = "Alura";  
  
String sub = nome.substring(1);  
System.out.println(sub);
```

[COPIAR CÓDIGO](#)

Ao executarmos o código, teremos o resultado impresso de `lura`.

Muitas vezes precisamos saber qual é o tamanho de uma determinada `String`, para isso utilizamos o método `length()`.

```
String nome = "Alura";
System.out.println(nome.length());
```

[COPIAR CÓDIGO](#)

O valor impresso será `5`, como o esperado. Com isso, podemos criar um laço, iniciando por `0`, e imprimirmos caractere por caractere utilizando o método `charAt()`.

```
String nome = "Alura";
System.out.println(nome.length());

for(int i = 0; i < nome.length(); i++) {
    System.out.println(nome.charAt(i));
}
```

[COPIAR CÓDIGO](#)

Teremos o seguinte resultado impresso:

5  
A  
l  
u

r

a

[COPIAR CÓDIGO](#)

O próximo método que conheceremos é o `isEmpty()`, em que podemos perguntar se a `String` está vazia. Para testarmos esse método criaremos uma `String` vazia; lembrando que esse objeto existe, apenas não possui nenhum caractere, em uma situação real para exemplificarmos: uma `String` vazia pode representar um campo de formulário que não foi preenchido pelo usuário.

```
String vazio = "";  
System.out.println(vazio.isEmpty());
```

[COPIAR CÓDIGO](#)

Recebemos como retorno um booleano, neste caso, recebemos um `true`. Se inserirmos um espaço na `String`, do ponto de vista do Java, não teremos mais uma `String` vazia, e o resultado será `false`.

Como do ponto de vista do usuário um espaço não é considerado um caractere, iremos utilizar outro método para anular todos os espaços em uma `String`. Esse método é chamado de `trim()`, e ele devolve uma nova apresentação, que chamaremos de `outroVazio`.

```
String vazio = " ";  
String outroVazio = vazio.trim();  
  
System.out.println(outroVazio.isEmpty());
```

[COPIAR CÓDIGO](#)

O resultado é true , afinal, removemos todos os espaços e temos efetivamente uma String vazia.

Para deixarmos mais claro como o método trim() opera, iremos escrever Alura em String vazio e adiconaremos muitos espaços extras, tanto no começo como ao final da palavra.

```
String vazio = "    Alura    ";
String outroVazio = vazio.trim();

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

[COPIAR CÓDIGO](#)

Com o uso do método que remove os espaços da String, temos o seguinte resultado

Alura

[COPIAR CÓDIGO](#)

Reparem em como seria a impressão sem o uso do método trim()

Alura

[COPIAR CÓDIGO](#)

O último método que veremos nesta aula é o contains() . Este método verifica se uma String possui uma sub-String. Perguntaremos se a String vazio possui uma sub-String Alu .

```
String vazio = "    Alura    ";
String outroVazio = vazio.trim();
```

```
System.out.println(vazio.contains("Alu"));
```

[COPIAR CÓDIGO](#)

O valor impresso será `true`, como o esperado.

Aprendemos vários métodos relacionados a String que podem nos ajudar no dia a dia.

09

## A interface CharSequence

Nos vídeos talvez você tenha percebido que alguns métodos da classe `String` recebem uma variável do tipo `CharSequence`. O tipo `CharSequence` é uma interface que a própria classe `String` implementa (pois uma `String` é uma sequência de caracteres!):

```
public class String implements CharSequence {
```

COPIAR CÓDIGO

Quando usamos a classe `String` até poderíamos declarar a variável com o tipo da interface, mas isso é raro de se ver:

```
CharSequence seq = "é uma sequencia de caracteres";
```

COPIAR CÓDIGO

O interessante é que existem outras classes que também implementam a interface `CharSequence`. Em outras palavras, existem outras classes que são sequências de caracteres além da classe `String`. Por quê?

## A classe `StringBuilder`

Vimos que a classe `String` é especial pois gera objetos imutáveis. Isso é considerado benéfico pensando no design mas é ruim pensando no desempenho (e por isso devemos usar aspas duplas na criação, pois a JVM quer contornar os problemas no desempenho com otimizações).

Agora vem um problema: imagina que você precisa criar um texto enorme e precisa concatenar muitas `String`, por exemplo:

```
String texto = "Socorram";
texto = texto.concat("-");
texto = texto.concat("me");
texto = texto.concat(", ");
texto = texto.concat("subi ");
texto = texto.concat("no ");
texto = texto.concat("ônibus ");
texto = texto.concat("em ");
texto = texto.concat("Marrocos");
System.out.println(texto);
```

[COPIAR CÓDIGO](#)

Nesse pequeno exemplo já criamos vários objetos, só porque estamos concatenando algumas `Strings`. Isso é nada bom pensando no desempenho e para resolver isso existe a classe `StringBuilder` que ajuda na concatenação de `Strings` de forma mais eficiente.

Veja o mesmo código usando o `StringBuilder`:

```
StringBuilder builder = new StringBuilder("Socorram");
builder.append("-");
builder.append("me");
builder.append(", ");
builder.append("subi ");
builder.append("no ");
builder.append("ônibus ");
builder.append("em ");
builder.append("Marrocos");
String texto = builder.toString();
System.out.println(texto);
```

[COPIAR CÓDIGO](#)

O `StringBuilder` é uma classe comum. Repare que usamos o `new` para a criação do objeto. Além disso, como o objeto é mutável, utilizamos a mesma referência, sem novas atribuições.

## A interface CharSequence

Agora o legal é que a classe `StringBuilder` também implementa a interface `CharSequence`:

```
public class StringBuilder implements CharSequence {
```

[COPIAR CÓDIGO](#)

CharSequence cs = new StringBuilder("também é uma sequencia de ca

[COPIAR CÓDIGO](#)

Isso faz que alguns métodos da classe `String` saibam trabalhar com o `StringBuilder`, por exemplo:

```
String nome = "ALURA";  
CharSequence cs = new StringBuilder("al");
```

```
nome = nome.replace("AL", cs);
```

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

[COPIAR CÓDIGO](#)

Vice-versa a classe `StringBuilder` tem métodos que recebem o tipo `CharSequence`. Dessa forma podemos trabalhar de maneira compatível com as duas classes, baseado numa interface comum.

01

## Analisando o sysout

### Transcrição

Continuaremos nosso curso de `java.lang`, aprendendo mais sobre o pacote e suas classes.

Analisaremos rapidamente uma linha que estamos utilizando desde o início do projeto:

```
//...
System.out.println("Alura");
//...
```

[COPIAR CÓDIGO](#)

Temos três partes que compõem essa linha, separadas por meio de um ponto ( . ), são elas `System` , `out` e `println` . Vamos dissecar essa linha e entender o que são cada um desses elementos.

Primeiro, o `System` .

Sabemos que se trata de uma classe, afinal a palavra se inicia com letra maiúscula, e que provém do pacote `java.lang` , portanto não é necessário importa-la.

No que diz respeito a visibilidade, a classe `System` deve ser pública, caso contrário não conseguíramos acessa-la.

Em segundo lugar, analisaremos o `out`.

Sabemos que não se trata de uma classe, afinal não se inicia com letra maiúscula, tampouco trata-se de um método.

`out` é um atributo público para que seja acessível fora de sua classe. A próxima questão que devemos nos perguntar é: este atributo é um primitivo? Suponhamos a classe `System`:

```
public class System {  
  
    public ??? out  
  
}
```

**COPIAR CÓDIGO**

Está faltando definir esse atributo. Qual seria a melhor forma de fazer isso?

Inserindo `int` ou `double`? Na verdade, isso não seria nem possível pois o `out` está entre pontos. Nas aulas anteriores aprendemos que uma variável que representa um primitivo não pode conter pontos, pois um primitivo é apenas um valor que não possui um método associado a ele, pois não é baseado em um classe.

O espaço só pode ter sido preenchido por uma outra classe, `out` é uma referência. Não sabemos o tipo da referência, pois essa informação não está visível quando analisamos a linha de código.

Sabemos outro dado interessante sobre o atributo `out`; quando falamos sobre a conta no bytebank, no princípio o saldo era público, portanto criávamos uma `Conta` e, mediante uma referência, acessávamos um objeto `saldo`.

```
Conta c = new Conta();
c.saldo = 55.5;
```

[COPIAR CÓDIGO](#)

Para acessarmos o `saldo` utilizamos a referência, lembrando que `saldo` é um atributo público da classe `Conta`.

No caso, o `out` é acessado por meio de uma classe, a `System`. O que significa que há um `static` dentro da classe `System`, ou seja, temos um `public static`, mas ainda não sabemos o tipo da referência.

```
public class System {
    public static ??? out
```

[COPIAR CÓDIGO](#)

Concluímos, então, que o nosso `out` possui acesso estático.

Já vimos esse conteúdo nas aulas anteriores, muito embora ele ainda não esteja tão sedimentado e necessita de estudo e prática.

O próximo elemento é o `println()`.

Não é uma classe, tampouco um atributo e, sim, um método. Podemos verifica-lo observando os parênteses `()`. É público e pode ser utilizado fora do pacote de origem. Será que este método possui um acesso estático? Para respondermos a essa pergunta, devemos observar o elemento anterior na linha de código, ou seja, o `out`. Da mesma forma que olhamos para `System` ao definirmos o acesso de `out`.

Como o elemento anterior ao `println()` é uma referência, o método não possui acesso estático.

Há algumas outras informações acerca de `println()`: ele pode receber uma `String`, mas também um `int`. Aparentemente, existem várias versões de um mesmo método, ou seja, muitas sobrecargas. Atenção, não confundam **sobrecarga** com **sobrescrita**, esta última é relacionada à herança, possui a mesma assinatura e não pode variar nos parâmetros.

Sabemos que `println()` não joga exceções do tipo `checked`, nunca fomos obrigados a fazer algum tratamento de exceção.

Observem como a linha de código `System.out.println()` é compacta, e quantas informações podemos retirar dela.

Nas próximas aulas aprenderemos novos recursos do pacote `java.lang`.

04

## Usando a classe Object

### Transcrição

Nesta aula daremos continuidade aos nossos estudos em `java.lang`.

Estudamos juntos a classe `String`, fizemos uma revisão e análise da linha de código `System.out.println()`, e agora veremos uma classe fundamental.

No Eclipse, criaremos uma nova classe que chamaremos simplesmente de `Teste`.

Nesta nova classe faremos o `System`.

```
package br.com.bytebank.banco.test;

public class Teste {

    public static void main(String[] args) {

        System.out.println();

    }
}
```

**COPIAR CÓDIGO**

Reparam que o código é compilado ainda que não tenhamos passado nenhum parâmetro para `println()`, mas podemos escrever uma `String`, inteiro ou um `booleano`.

```
package br.com.bytebank.banco.test;

public class Teste {

    public static void main(String[] args) {

        System.out.println("x");
        System.out.println(3);
        System.out.println(false);

    }

}
```

[COPIAR CÓDIGO](#)

Estamos nos deparando com várias versões do mesmo método. Para simular essa ideia, usaremos métodos `println()`. Na classe que implementou o `println()` existe um método com essa assinatura, mas veremos como se dá sua implementação.

```
package br.com.bytebank.banco.test;

public class Teste {

    public static void main(String[] args) {

        System.out.println("x");
        System.out.println(3);
        System.out.println(false);

        println();
    }
}
```

```
static void println() {  
}  
  
static void println(int a) {  
}  
  
static void println(boolean valor) {  
}  
}
```

**COPIAR CÓDIGO**

Dentro da classe que representa o `out`, existem no mínimo estas três versões do método `println()`, ou seja, são vários métodos com o mesmo nome e assinatura, apenas variando os parâmetros. Isso se chama **sobrecarga**, lembrem-se de não confundir com **sobrescrita**, que é relacionada à herança e sempre mantém os mesmos parâmetros.

Criaremos uma `ContaCorrente` e uma `ContaPoupanca` com os respectivos números de conta e agência.

```
package br.com.bytebank.banco.test;  
  
public class Teste {  
  
    public static void main(String[] args) {  
  
        System.out.println("x");  
        System.out.println(3);  
        System.out.println(false);  
  
        ContaCorrente cc = new ContaCorrente(22, 33);  
    }  
}
```

```
ContaPoupanca cp = new ContaPoupanca(33, 22);

        println();
    }
```

[COPIAR CÓDIGO](#)

Queremos que o `System.out.println()` funcione com `cc` e `cp`. De forma mais clara, a ideia é:

```
System.out.println(cc);
System.out.println(cp);
```

[COPIAR CÓDIGO](#)

Como podemos implementar isso? Teremos de adicionar mais uma versão do método: `println(ContaCorrente conta)`. Feito isso, acionamos o método `println()` passando para ele `cc`. Tudo deve funcionar perfeitamente, pois o método que recebe `ContaCorrente` foi implementado mais abaixo do código.

```
package br.com.bytebank.banco.test;

public class Teste {

    public static void main(String[] args) {

        System.out.println("x");
        System.out.println(3);
        System.out.println(false);

        ContaCorrente cc = new ContaCorrente(22, 33);
        ContaPoupanca cp = new ContaPoupanca(33, 22);

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

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

println(cc);

}

static void println() {

}

static void println(int a) {

}

static void println(boolean valor) {

}

static void println(ContaCorrente conta) {

}

}
```

[COPIAR CÓDIGO](#)

A próxima questão é: dentro de `System.out`, ou seja, a classe que foi associada ao atributo `out`, existe o método `println()` que recebe `ContaCorrente`? Não há! Mas mesmo assim funciona, podemos passar para o `println()` qualquer tipo de referência. Como isso funciona?

Primeiramente, vamos remover os espaços do nosso código e estudar essa questão mais de perto.

```
System.out.println(cc);
System.out.println(cp);
```

```
        println(cc);  
  
    }  
  
    static void println() {}  
    static void println(int a) {}  
    static void println(boolean valor) {}  
  
    static void println(ContaCorrente conta) {  
    }  
  
}
```

**COPIAR CÓDIGO**

Caso alteremos `ConcaCorrente` para `ContaPoupanca` na última linha do código, não dará certo, pois o `println()` mais acima continua recebendo `cc` e não `cp`.

```
System.out.println(cc);  
System.out.println(cp);  
  
println(cc);  
  
}  
  
static void println() {}  
static void println(int a) {}  
static void println(boolean valor) {}  
  
static void println(ContaPoupanca conta) {  
}  
  
}
```

[COPIAR CÓDIGO](#)

No entanto, reparem que `ConcaPoupanca` e `ContaCorrente` têm algo em comum: ambas são contas. Portanto podemos escrever o código adicionando simplesmente `Conta` e realizando sua importação.

```
import br.com.bytebank.banco.modelo.Conta;
import br.com.bytebank.banco.modelo.ContaCorrente;
import br.com.bytebank.banco.modelo.ConcaPoupanca;

public class Teste {

    public static void main(String[] args) {

        System.out.println("x");
        System.out.println(3);
        System.out.println(false);

        ContaCorrente cc = new ContaCorrente(22, 33);
        ContaPoupanca cp = new ContaPoupanca(33, 22);

        System.out.println(cc);
        System.out.println(cp);

        println(cc);

    }
}

static void println() {}
static void println(int a) {}
static void println(boolean valor) {}
```

```
static void println(Conta conta) {  
}  
•  
}  
•
```

[COPIAR CÓDIGO](#)

Escrevendo o código desta maneira, o `println()` se torna mais genérico, aceitando tanto `cc` quanto `cp`. Caso queiramos um tipo mais genérico e uma referência qualquer, como isso seria feito?

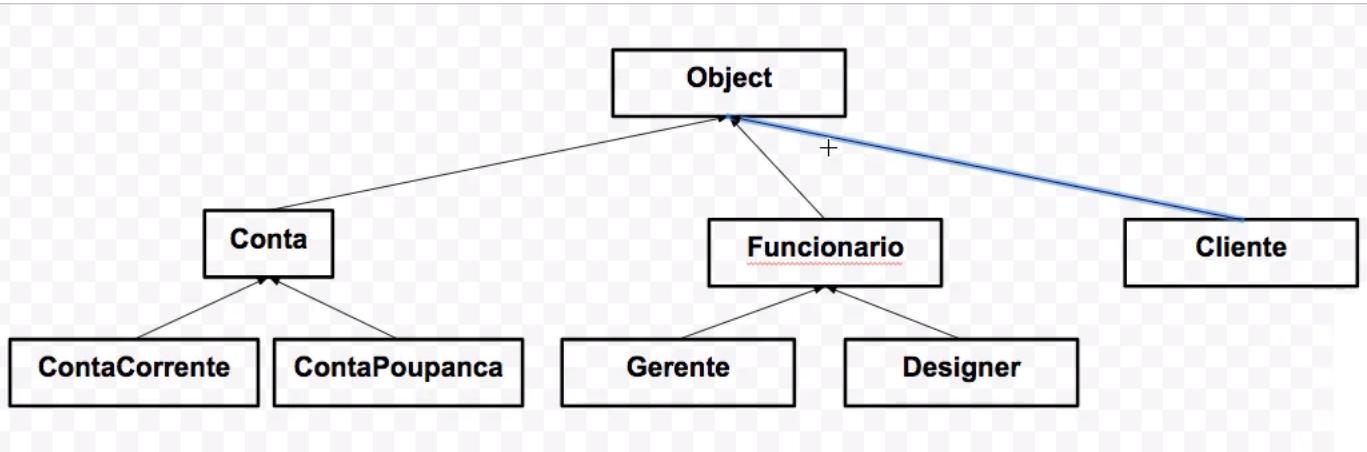
```
//Código omitido  
  
    println(cc);  
  
}  
•  
•  
    static void println() {}  
    static void println(int a) {}  
    static void println(boolean valor) {}  
  
    static void println(???? referencia) {  
    }  
  
}
```

[COPIAR CÓDIGO](#)

Falamos bastante sobre herança em outro curso, e criamos a hierarquia com base em `Conta`, `Funcionario` e `Cliente`.

Queremos fazer uma classe mãe que é estendida por todas essas classes que estão no primeiro nível; lembre-se que em Java só podemos estender uma classe.

O nome da nossa classe mãe será `Object`, e ela estará no topo da hierarquia.



Porém, ao abrirmos a classe `Conta` ou `Cliente`, veremos que não existe `extends Object`, mas se você realiza essa extensão, o computador automaticamente preenche o código.

```

public class Cliente extends Object {

    private String nome;
    private String cpf;
    private String profissao;

    public String getNome() {
        return nome;
    }
}
  
```

[COPIAR CÓDIGO](#)

Temos a classe `Object` no topo da hierarquia, isso significa que também são herdados métodos dessa classe, ou seja, reutilização. Lembrem-se que herança possui duas características: reutilização de código e polimorfismo.

Na classe `Teste` estávamos justamente analisando o polimorfismo. Qual o tipo de referência genérica que podemos incluir em nosso código?

```
//Código omitido

    println(cc);

}

•
static void println() {}
static void println(int a) {}
static void println(boolean valor) {}

static void println(???? referencia) {
}

}
```

**COPIAR CÓDIGO**

Adicionaremos, justamente, `Object`, que funcionará com `ContaCorrente` e `ContaPoupanca`. Adicionaremos, `Cliente`, realizando sua devida importação, dessa forma, `println()` funcionará inclusive com esta classe.

```
//Código omitido

    ContaCorrente cc = new ContaCorrente(22, 33);
    ContaPoupanca cp = new ContaPoupanca(33, 22);
    Cliente cliente = new Cliente();

    println(cliente);

}

•
static void println() {}
static void println(int a) {}
```

```
static void println(boolean valor) {}

static void println(Object referencia) {
}

}
```

[COPIAR CÓDIGO](#)

Com "Ctrl" pressionado, passaremos o cursor do mouse sobre o método `println()` para acessá-lo. Dentro do método encontraremos alguns elementos que não são familiares, mas se atentem para o tipo de referência que o método recebe: `Object`, por isso ele funciona com qualquer referência, afinal, tudo é um objeto.

```
//Código omitido

public void println(Object x) {
    String s = String.valueOf(x);
    synchronized (this) {
        print(s);
        newLine();
    }
}
```

[COPIAR CÓDIGO](#)

Para ilustrarmos melhor essa ideia, podemos expressar nosso código de `Teste` da seguinte maneira: substituindo `ContaCorrente`, `ContaPoupanca` e `Cliente` por `Object`.

```
//Código omitido

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

```
System.out.println("x");
System.out.println(3);
System.out.println(false);

Object cc = new ContaCorrente(22, 33);
Object cp =new ContaPoupanca(33, 22);
Object = new Cliente();

System.out.println(cc);
System.out.println();

println(cliente);

}
```

[COPIAR CÓDIGO](#)

Esse é um bom exemplo de polimorfismo, temos uma referência genérica que se liga a um objeto mais específico. Podemos utilizar tanto a referência mais específica quanto a mais genérica para designar `ContaCorrente`, `ContaPoupanca` e `Cliente`.

Comentaremos a linha de código referente ao `println(cliente)` e executaremos o código.

```
//Código omitido

public static void main(String[] args) {

    System.out.println("x");
    System.out.println(3);
    System.out.println(false);

    Object cc = new ContaCorrente(22, 33);
    Object cp = new ContaPoupanca(33, 22);
```

```
Object = new Cliente();  
  
System.out.println(cc);  
System.out.println();  
  
//println(cliente);  
  
}
```

[COPIAR CÓDIGO](#)

Observem o que foi impresso:

```
x  
3  
br.com.bytebank.banco.modelo.ContaCorrente@3abfe836  
br.com.bytebank.banco.modelo.ContaPoupanca@2ff5659e
```

[COPIAR CÓDIGO](#)

O que nos interessa são as últimas duas linhas, que estão relacionadas aos dois `System.out.println()` encontrados em nosso código de `Testes`.

Observem que foi impresso o nome da classe em seu *full qualified name*, mais `@3abfe836`. De onde vem esse código e quem o implementou? A classe `Object`; essa classe possui alguma funcionalidade que resulta na produção dessa String.

Essa funcionalidade é um método, afinal, dentro da funcionalidade existe um método encapsulado. Esse método é chamado `toString()`, e ele foi herdado de `Object`.

Observem: Caso digitemos o ponto (.) em `System.out.println(cc)` serão sugeridos pelo Eclipse todos os métodos implementados na classe `Object`, afinal estamos

trabalhando com essa referência genérica em nosso código.

```
//Código omitido

public static void main(String[] args) {

    System.out.println("x");
    System.out.println(3);
    System.out.println(false);

    Object cc = new ContaCorrente(22, 33);
    Object cp = new ContaPoupanca(33, 22);
    Object = new Cliente();

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

**COPIAR CÓDIGO**

No entanto, se modificarmos o código trabalhando novamente com referências mais específicas, substituindo `Object` por `ContaCorrente`, veremos as sugestões de métodos que foram implementados na classe `ContaCorrente`, `Conta` e o método `ToString()` da classe `Object`, por isso conseguimos evoca-lo.

```
//Código omitido

public static void main(String[] args) {

    System.out.println("x");
    System.out.println(3);
    System.out.println(false);

    ContaCorrente cc = new ContaCorrente(22, 33);
    Object cp = new ContaPoupanca(33, 22);
```

```
Object = new Cliente();  
  
System.out.println(cc.toString());  
System.out.println();
```

[COPIAR CÓDIGO](#)

Ao executarmos nosso código, obteremos a mesma saída, apenas a numeração se modifica, já que está relacionada ao endereço do objeto e não temos muito controle sobre isso.

br.com.bytebank.banco.modelo.ContaCorrente@3abfe836

[COPIAR CÓDIGO](#)

Caso observemos o método `toString()` utilizando o botão "Ctrl", veremos o seguinte código:

```
//Código omitido  
  
public String toString() {  
    return getClass().getName() + "@" + Integer.toHexString(hashCode());  
}  
//Código omitido
```

[COPIAR CÓDIGO](#)

Nesta aula, vimos exemplos de **polimorfismo** e **reutilização de código**.



06

## O método `toString()`

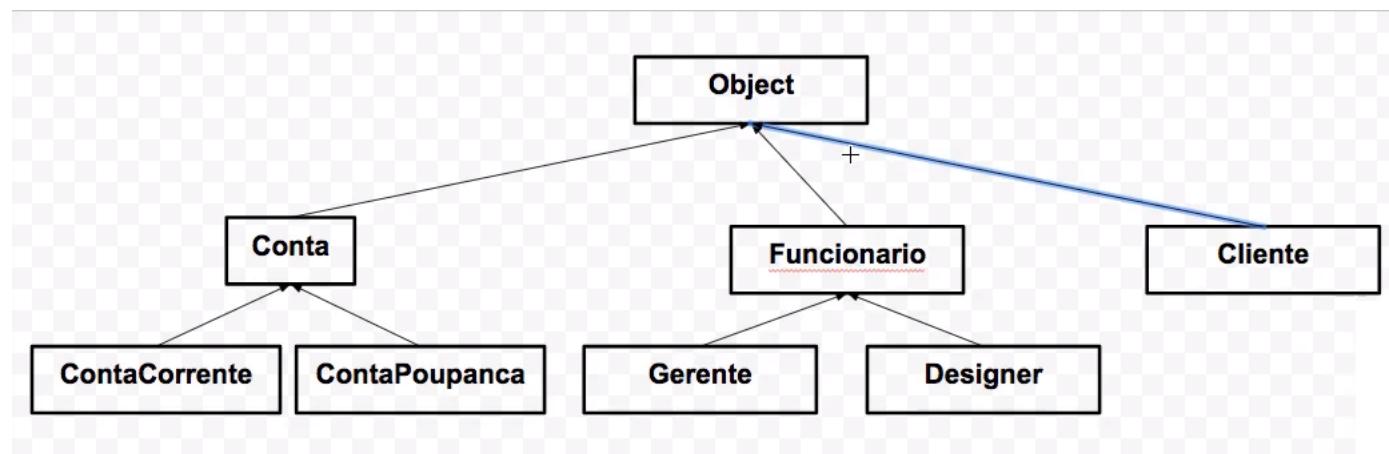
### Transcrição

Na última aula, fizemos uma revisão sobre polimorfismos e aproveitamento de código, e nosso foco agora é melhorar a saída impressa por `Teste.java`.

A classe `Object` nos fornece uma implementação do método `toString()`, no entanto ela não nos serve de nada neste momento. A ideia é que as classes reimplementem o método para atribuí-lo a um significado maior. Vamos lá?

Abriremos a classe `ContaCorrente` e sobrescreveremos o método `toString()`.

O Eclipse sabe que todas as classes trabalham tendo em base a classe `Object` no topo da hierarquia.



Ao começarmos a escrever o método `toString()`, quando chegamos a escrever `toStr` pressionaremos o atalho "Ctrl + Space" para acionarmos o *autocomplete*.

Eclipse apresenta duas sugestões: implementar o método `toString()` ou implementar o `toString()`; escolheremos, claro, a segunda opção.

Com isso, é posta uma implementação padrão com o `@Override`, para realmente garantir que estamos sobrescrevendo o método.

```
//...
@Override
public void deposita(double valor){
    super.saldo +=valor;
}

@Override
public double getValorImposto() {
    return super.saldo * 0.01;
}

@Override
public String toString() {
    // TODO Auto-generated method stub
    return super.toString();
}

//...
```

**COPiar CÓDIGO**

Temos na última linha da implementação o chamado `super.toString()`. Essa linha não nos interessa, pois não queremos chamar um método da classe `Object`, e sim, dar um significado maior para o método.

Apagaremos o comentário, bem como a linha, e em frente ao `return` escreveremos `ContaCorrente`.

```
//...
@Override
public void deposita(double valor){
    super.saldo +=valor;
}

@Override
public double getValorImposto() {
    return super.saldo * 0.01;
}

@Override
public String toString() {
    return "ContaCorrente";
}
}

//...
```

**COPIAR CÓDIGO**

Voltaremos à classe `Teste` para testarmos nossa saída. Deixaremos como comentário os `System.out.println()`s.

```
//...

public static void main(String[] args) {

//    System.out.println("x");
//    System.out.println(3);
//    System.out.println(false);

ContaCorrente cc = new ContaCorrente(22, 33);
Object cp = new ContaPoupanca(33, 22);
Object cliente = new Cliente();
```

```
System.out.println(cc.toString());  
System.out.println(cp);
```

&lt;....!....&gt;

[COPIAR CÓDIGO](#)

Ao executarmos o código, temos a seguinte saída:

```
ContaCorrente  
br.com.bytebank.banco.modelo.ContaPoupanca@3abfe836
```

[COPIAR CÓDIGO](#)

Está diferente, nosso método `toString()` foi chamado, lembrando que temos um objeto `ContaCorrente`, e para ele será utilizado um método específico. Quem define qual método será escolhido é o objeto.

Mesmo se alterarmos a referência para `Object`, a saída no nosso código continua a mesma.

```
//...  
Object cc = new ContaCorrente(22, 33);  
Object cp = ContaPoupanca(33, 22);  
Object cliente = new Cliente();  
  
System.out.println(cc.toString());  
System.out.println(cp);  
//...
```

[COPIAR CÓDIGO](#)

Podemos melhorar ainda mais a nossa saída.

Em `ContaCorrente`, iremos concatenar uma informação do objeto. Nessa fase, o desenvolvedor irá definir uma informação interessante, que faz sentido e facilita a leitura e o entendimento.

Colocaremos o número da conta, acionando o método `getNumero()`.

```
//...
@Override
public void deposita(double valor) {
    super.saldo += valor;
}

@Override
public double getValorImposto() {
    return super.saldo * 0.01;
}

@Override
public String toString() {
    return "ContaCorrente: " + super.getNumero();
}
//...
```

[COPIAR CÓDIGO](#)

Novamente executaremos o código em `Teste`. Feitas as modificações, temos impresso o número da conta.

```
ContaCorrente: 33
br.com.bytebank.banco.modelo.ContaPoupanca@3abfe836
```

[COPIAR CÓDIGO](#)

Podemos melhorar ainda mais a nossa saída escrevendo o texto `Numero`.

```
//...
@Override
public String toString() {
    return "ContaCorrente, Numero: " + super.getNumero();
}
//...
```

[COPIAR CÓDIGO](#)

Observem o resultado da impressão ao executarmos a classe `Teste`.

```
ContaCorrente, Numero: 33
br.com.bytebank.banco.modelo.ContaPoupanca@3abfe836
```

[COPIAR CÓDIGO](#)

Faremos o mesmo procedimento na classe `ContaPoupanca`, inserindo o seguinte trecho de código na classe:

```
//...
@Override
public String toString() {
    return "ContaPoupanca, Numero: " + super.getNumero();
}
//...
```

[COPIAR CÓDIGO](#)

Na classe `Teste`, não precisamos evocar o `toString()` na linha do `System.out` explicitamente, sendo uma prática um pouco estranha para um desenvolvedor Java.

pois o `println()` possui internamente um mecanismo que evoca o método `toString()`.

```
//...
Object cc = new ContaCorrente(22, 33);
Object cp = new ContaPoupanca(33, 22);
Object cliente = new Cliente();

System.out.println(cc);
System.out.println(cp);
//...
```

[COPIAR CÓDIGO](#)

Ao executarmos o código, teremos o seguinte resultado de saída:

```
ContaCorrente, Numero: 33
ContaPoupanca, Numero: 22
```

[COPIAR CÓDIGO](#)

Com isso, realizamos uma pequena melhoria em nosso projeto, refinando o comportamento de alguns elementos.

Realizaremos mais modificações no âmbito da herança.

Entre as duas classes, `ContaCorrente` e `ContaPoupanca`, temos um trecho de código muito parecido, com exceção dos próprios nomes das classes.

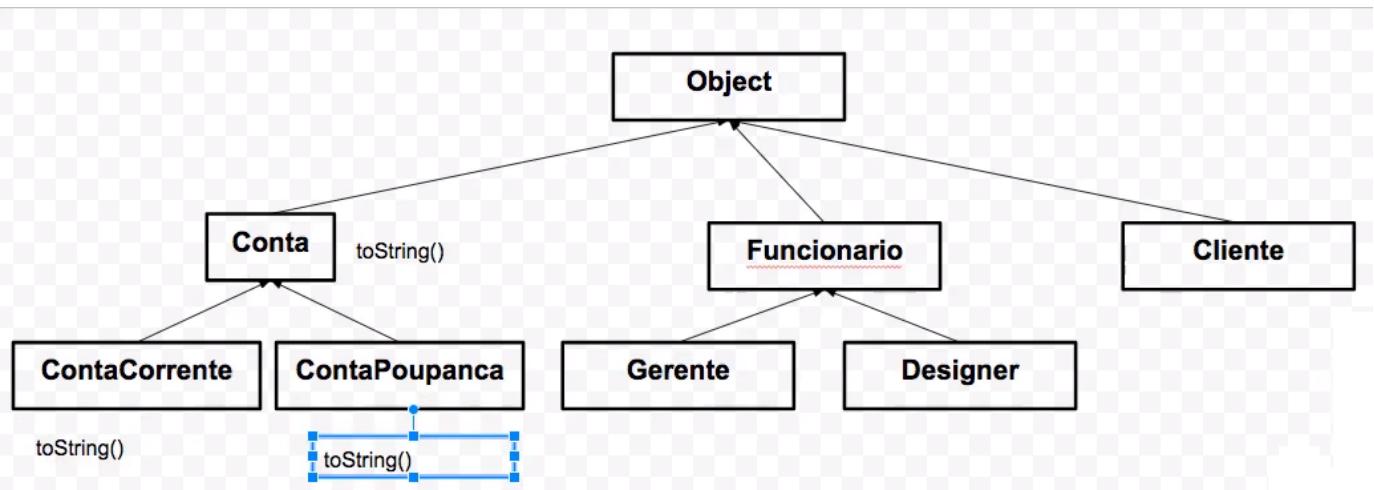
```
//...
@Override
public String toString() {
    return "ContaPoupanca, Numero: " + super.getNumero();
```

```
}
```

```
//...
```

[COPIAR CÓDIGO](#)

Tendo isso em vista, criaremos um método `toString()` mais genérico dentro da classe `Conta`, neste método imprimiremos o número da conta e da agência. Feito isso, iremos refinar esse método em cada filho da classe `Conta`. Portanto quando chamarmos o método na classe filha, na verdade estamos aproveitando o método mais genérico da classe `Conta`.



Na classe `Conta`, iremos reaproveitar o trecho de código utilizado nas classes `ContaCorrente` e `ContaPoupanca`, fazendo uma pequena modificação: iremos imprimir somente o `Numero`, sem especificar um ou outro tipo de conta; retiraremos o `super`, pois ele seria a classe `Object`, e utilizaremos o `this.numero`.

Portanto o `toString()` da classe `Conta` nos devolve `Numero` concatenado com o número da conta em questão.

```
//...
@Override
public String toString() {
    return "Número: " + this.numero;
```

```
}
```

```
//...
```

[COPIAR CÓDIGO](#)

Nos filhos da classe `Conta` também iremos realizar modificações.

Em `ContaCorrente` iremos retirar `Numero`, que apareceria duplicado. Também substituiremos o `super.getNumero()` por `super.toString()`

```
//...
```

```
@Override
```

```
public String toString() {
```

```
    return "ContaCorrente, " + super.toString();
```

```
}
```

```
//...
```

[COPIAR CÓDIGO](#)

A mesma revisão da herança ocorre em `ContaPoupanca`.

```
//...
```

```
@Override
```

```
public String toString() {
```

```
    return "ContaPoupanca, " + super.toString();
```

```
}
```

```
//...
```

[COPIAR CÓDIGO](#)

Ao executarmos o código na classe `Teste`, teremos o seguinte resultado:

```
ContaCorrente, Numero: 33
```

```
ContaPoupanca, Numero: 22
```

[COPIAR CÓDIGO](#)

É interessante destacar que caso modifiquemos o `toString()` da classe mãe, automaticamente o mesmo se dá para as classes filhas. Por exemplo, iremos concatenar a informação de `Agencia`.

```
//...
@Override
public String toString() {
    return "Numero: " + this.numero + "Agencia:" + this.agencia;
}
//...
```

[COPIAR CÓDIGO](#)

Feito o acréscimo na classe mãe, ao executarmos o código em `Teste` obteremos o seguinte resultado:

```
ContaCorrente, Numero: 33, Agencia: 22
ContaPoupanca, Numero: 22 Agencia: 33
```

[COPIAR CÓDIGO](#)

Com isso está mais claro de que tipo de objeto se trata e seus valores.

09

## Revisão e Conclusão

### Transcrição

Faremos uma pequena revisão de todo o conteúdo visto até o presente momento em nosso curso; caso você se sinta seguro, você pode pular este vídeo, pois ele não terá nenhum conteúdo inédito.

Todas as classes trabalham com a regra de que a classe `Object` está no topo da hierarquia, sendo esta a classe mãe. Não é necessário escrever `extends Object` nas classes filhas como `Cliente`, pois o compilador insere essa informação automaticamente.

É raro encontrarmos `extends Object`, porque todo desenvolvedor sério de Java sabe que as classes estendem a classe `Object`.

A classe `Object` foi inventada a partir do conceito de que de que tudo é um objeto. Com isso, conseguimos algumas vantagens relacionadas à herança, como **polimorfismo** e **reutilização de código**.

Nós vimos apenas um método, mas existem muitos outros que dependem da classe `Object`, vários laços foram escritos de forma dependente dessa classe, pois são utilizados alguns métodos básicos que já foram implementados dentro dessa classe, então todos as classes associadas à `Object` terão a mesma funcionalidade.

O exemplo que utilizamos no curso foi o método `println()`, que como podemos observar na classe `Teste`, funciona com qualquer tipo de referência, afinal todas ...

compatíveis com a classe `Object`.

```
//...
Object cc = new ContaCorrente(22, 33);
Object cp = new ContaPoupanca(33, 22);
Object cliente = new Cliente();

System.out.println(cc);
System.out.println(cp);
//...
```

[COPIAR CÓDIGO](#)

Ao acessarmos `println()` utilizando a tecla "Ctrl" pressionada e clicando sobre o método, veremos em seu código fonte que ele funciona com qualquer tipo de referência, afinal, recebe a referência mais genérica o possível, que é `Object`.

```
//...
public void println(Object x) {
    String s = String.valueOf(x);
    synchronized (this) {
        print(s)
        newLine();
    }
}
//...
```

[COPIAR CÓDIGO](#)

Não é possível fazer muita coisa com o `Object`, por exemplo, na classe `Teste` escreveremos `cc`, que é uma referência do tipo `Object`, e chama alguns outros métodos genéricos, como `toString()`.

`cc.toString()`[COPIAR CÓDIGO](#)

É exatamente o que acontece no `System.out.println()`, observemos o método estático da classe `String`, o `valueOf()`.

```
public void println(Object x) {  
    String s = String.valueOf(x);  
    synchronized (this) {  
        print(s)  
        newLine();  
    }  
}
```

[COPIAR CÓDIGO](#)

Acessaremos esse método através do "Ctrl" pressionado, como já sabemos.

Dentro de `valueOf()` verificaremos que ele também é dependente de `Object`, o tipo mais genérico o possível. Ao longo do código existe um operador ternário - que é basicamente um `if` mais enxuto - que verifica se o objeto é nulo, caso sim, é devolvido `null` como `String`, se não (essa condicional é representada pelos dois pontos `:`) é devolvido `obj.toString()`.

```
public static String valueOf(Object obj) {  
    return (obj == null) ? "null" : obj.toString();  
}
```

[COPIAR CÓDIGO](#)

A ideia do `toString()` da classe `Object` é ser sobreescrito, assim como a maioria dos outros métodos "querem" ser sobreescritos pelos filhos, com isso eles ganham mai

significado. É o que fizemos na classe `Conta`, para procurarmos rapidamente o método, acionamos o atalho "Ctrl + O" e o procuramos dentro da classe e observamos a atribuição de significado.

```
//...
public static int gettotal(){
    return
}

@Override
public String toString() {
    return "Número: " + this.numero + ", Agencia : " + this.agencia
}

}
//...
```

**COPIAR CÓDIGO**

Nós também refinamos o `toString` na classe filha, adicionando uma informação e depois a implementação da classe. Observem o código de `ContaCorrente`:

```
//...
@Override
public double getValorImposto() {
    return super.saldo *0.01;
}

@Override
public String toString() {
    return "ContaCorrente, " + super.toString();
}
//...
```

**COPIAR CÓDIGO**

Resumindo, existe um universo com muitas classes que se baseiam em métodos definidos na classe `Object`, e o desenvolvedor deve sobreescrivê-los para atribuir maior significado a eles. Analisamos o método `println()` que internamente chama `toString()`, mais genérico do que, por exemplo, o `java.lang`.

Existem no mínimo dois pacotes super importantes que veremos nos próximos cursos, como `java.util`, em que encontraremos as coleções, conjuntos e mapas. Teremos, ainda, o `java.io`, que contém as classes que se preocupam com a leitura e escrita de dados, streams, *readers* e *writers*.

Precisamos dominar a classe `Object` para conseguirmos trabalhar, principalmente, com o `java.util` e conhecermos outros métodos como `hashCode()` e `equals()` que deverão ser sobreescritos.

Nos próximos cursos, focaremos na API e aprenderemos mais classes para analisar como o Java trabalha com estrutura de dados e leitura. Muito obrigado a todos que assistiram o curso comigo, e já lhes deixo o convite para participarem do próximo. A grande parte conceitual da linguagem nós já vimos, e agora podemos nos focar nas bibliotecas oferecidas pelo Java.

Até mais!