

 02

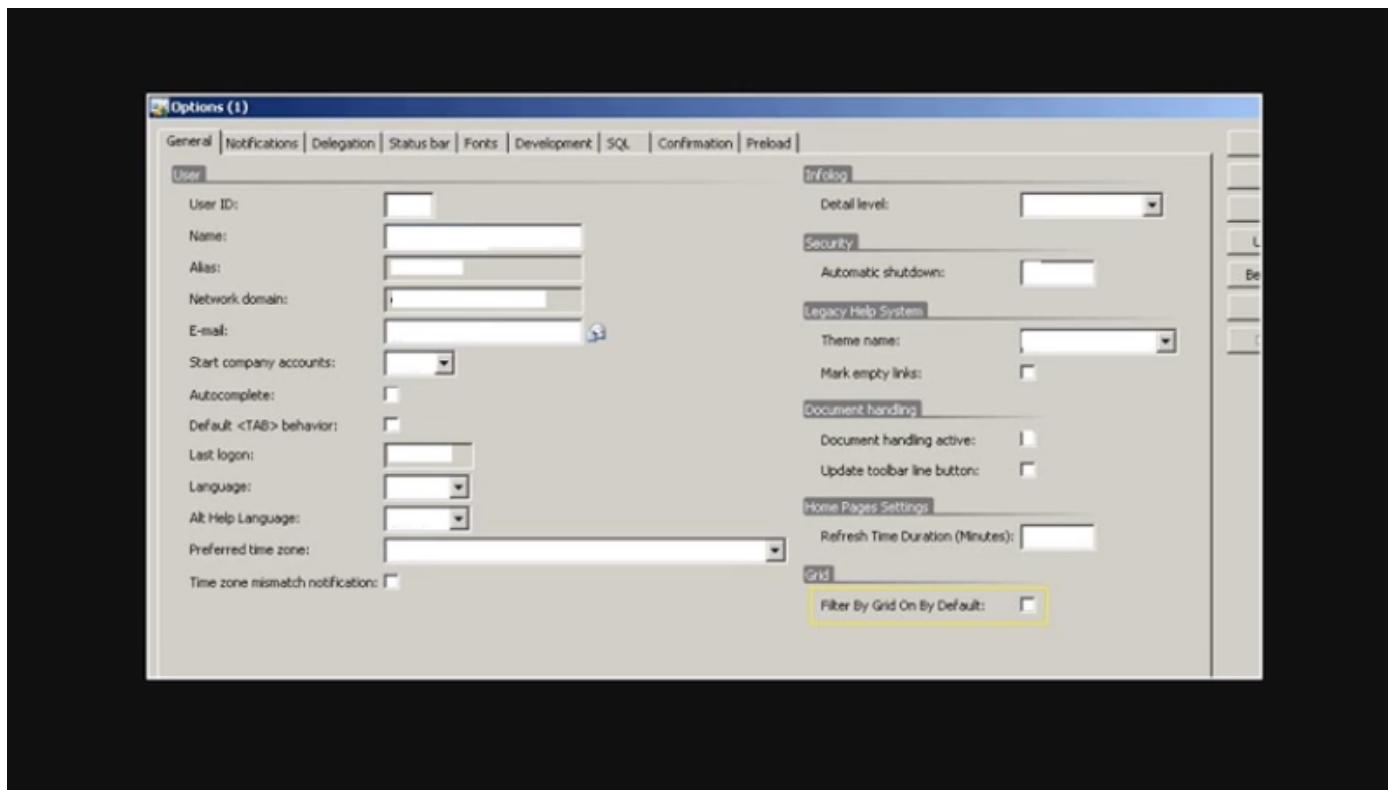
Paradigma procedural vs Objetos

Transcrição

Para entendermos a grande vantagem da orientação aos objetos, veremos quais são as dificuldades que e impulsionaram a criação desse paradigma.

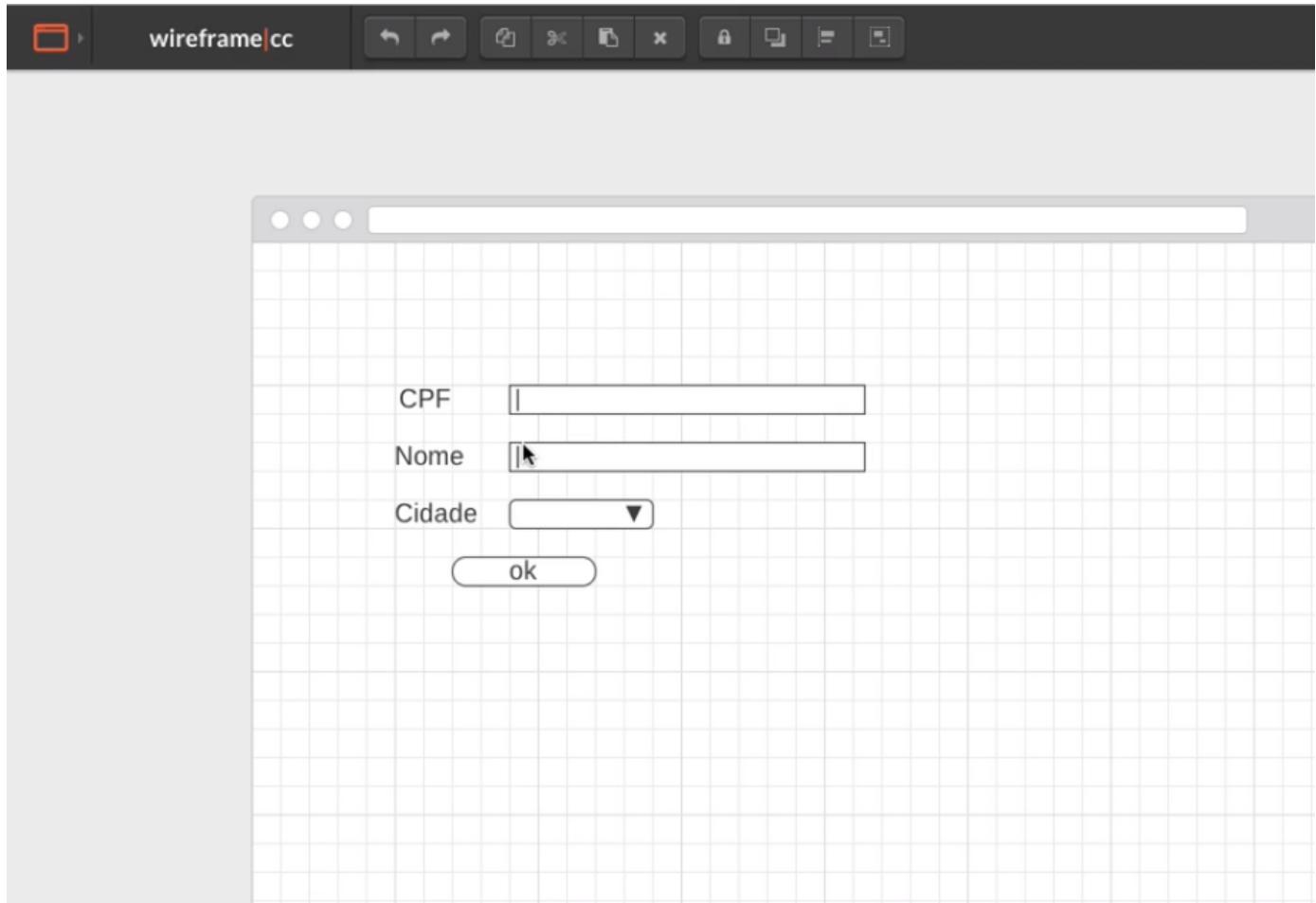
Antigamente tudo era procedural, não havia o conceito de programar voltado para um objeto. Mesmo hoje, com o desenvolvimento de tantas linguagens, alguns programadores ainda utilizam técnicas arcaicas da programação procedural. Esse método ainda faz sentido em alguns ambientes, mas na maioria dos casos, isso não se aplica ao Java.

No começo da década de noventa, os programadores trabalhavam com formulários longos e times enxutos, não havia uma equipe grande de desenvolvedores trabalhando em um projeto. Quem escrevia todo o formulário, cuidava de sua validação e de seu banco de dados, muitas vezes era um único programador ou programadora. Portanto, não havia como se atentar para todos os detalhes. Hoje em dia temos programas muito complexos, e o sistema de formulário se tornou insustentável.



Iremos verificar como trabalhávamos com os formulários para entendermos a diferença de paradigma gerada pela orientação aos objetos.

Usaremos o site [Wire Frame \(<https://wireframe.cc/>\)](https://wireframe.cc/), muito utilizado para quem quer esboçar UX Design, para analisarmos um exemplo de como eram criados os formulários antigamente. Usaremos uma linguagem e sintaxe hipotéticas. Temos um formulário simples de cadastro que contém **CPF, Nome e Cidade**.



Veremos como seria o código fonte desse formulário nessa linguagem imaginária. No campo CPF, poderíamos escrever que a variável `cpf` recebe do `formulario1` o campo denominado `CPF`.

```
var cpf := formulario1->CPF
```

[COPIAR CÓDIGO](#)

Repetir o mesmo procedimento para todos os campos do formulário.

```
var cpf := formulario1->CPF  
var nome := formulario1->Nome  
var cidade := formulario1->Cidade
```

[COPIAR CÓDIGO](#)

Suponhamos que este seja um formulário utilizado no sistema de uma padaria. Neste ponto, poderíamos acionar uma função denominada `gravar`, que salvaria no banco dados as informações de cadastro do cliente.

```
var cpf := formulario1->CPF  
var nome := formulario1->Nome  
var cidade := formulario1->Cidade  
  
gravarCliente(cpf, nome, cidade)
```

[COPIAR CÓDIGO](#)

Segundo a linguagem hipotética, o código está funcional.

Imagine a seguinte situação: seu chefe pede uma validação para o campo "CPF". Da forma como o código está organizado no momento, o campo "CPF" aceita qualquer tipo de dado, como letras e números aleatórios. Em outras palavras, o que o código faz é somente guardar o *input text* no banco de dados, independente do conteúdo.

Poderíamos solucionar esse problema adicionando outras funções ao código. Antes da gravação no banco de dados, adicionaremos uma função que valida o CPF - existe um cálculo específico para gerar CPFs - e passa uma variável. Teríamos como retorno um dado *booleano*; caso seja um CPF válido (`if`) a informação será gravada no banco de dados. Caso contrário (`else`) será emitida uma mensagem de erro.

```
var cpf := formulario1->CPF  
var nome := formulario1->Nome  
var cidade := formulario1->Cidade  
  
var sucesso = validaCpf(cpf)  
if(sucesso)
```

```
    gravarCliente(cpf, nome, cidade)
else
    mostraErro()
```

[COPIAR CÓDIGO](#)

Não existe nenhum problema estrutural no código, e muitas vezes é dessa forma que solucionaremos questões na programação.

Mas imaginem a seguinte situação: no sistema padaria não existe apenas o formulário de cadastro, mas também um formulário de busca de clientes através do CPF, e esse CPF precisa ser validado antes da busca ser realizada.

Poderíamos começar o nosso código fonte da seguinte maneira:

```
var cpf := formularioBusca->CPF
buscaNoBanco(cpf)
```

[COPIAR CÓDIGO](#)

Poderíamos utilizar a função de validação que conhecemos, basta copiá-la do código de cadastro e colá-la, fazendo pequenas alterações. Ao invés de salvar no banco de dados, iremos procurar no banco um CPF específico.

```
var cpf := formularioBusca->CPF
var sucesso = validaCpf(cpf)
if(sucesso)
    buscaNoBanco(cpf)
else
    mostrarErro()
```

[COPIAR CÓDIGO](#)

Conseguimos atender as novas demandas da empresa, e o nosso código está funcional. Mas existem problemas nesse tipo de abordagem. Caso tenhamos trinta e seis formulários diferentes que articulam a informação "CPF", a nova demanda da empresa é que cada CPF seja validado, e caso não, o texto ficará em vermelho e surgirá uma mensagem de erro.

Com uma quantidade grande de formulários para configurar, teremos dificuldade em descobrir o trecho adequado do código.

Acionar o atalho "Ctrl + F" e procurar cada trecho de código que contenha a palavra "CPF" seria muito trabalhoso.

Um problema mais grave: caso entre um novo integrante na equipe e sua primeira tarefa é lidar com um novo sistema que cadastre receitas sugeridas pelos clientes. E esse novo sistema da empresa faz uso da informação do CPF dos usuários. O novo integrante terá dificuldade em validar o CPF.

Poderíamos, por exemplo, criar um manual do sistema da empresa para os novos funcionários, mas essa não é a alternativa mais simples.

O ideal é que possamos fazer uma alteração em um único local do sistema, e assim, os CPFs em todas as interfaces de usuário precisariam ser validados.

Com a orientação a objetos, a ideia de dados e funcionalidades - ou "comportamentos" - estarão interligados, gerando uma enorme facilidade na organização e manutenção de um determinado sistema.

01

Primeira classe - Contas

Transcrição

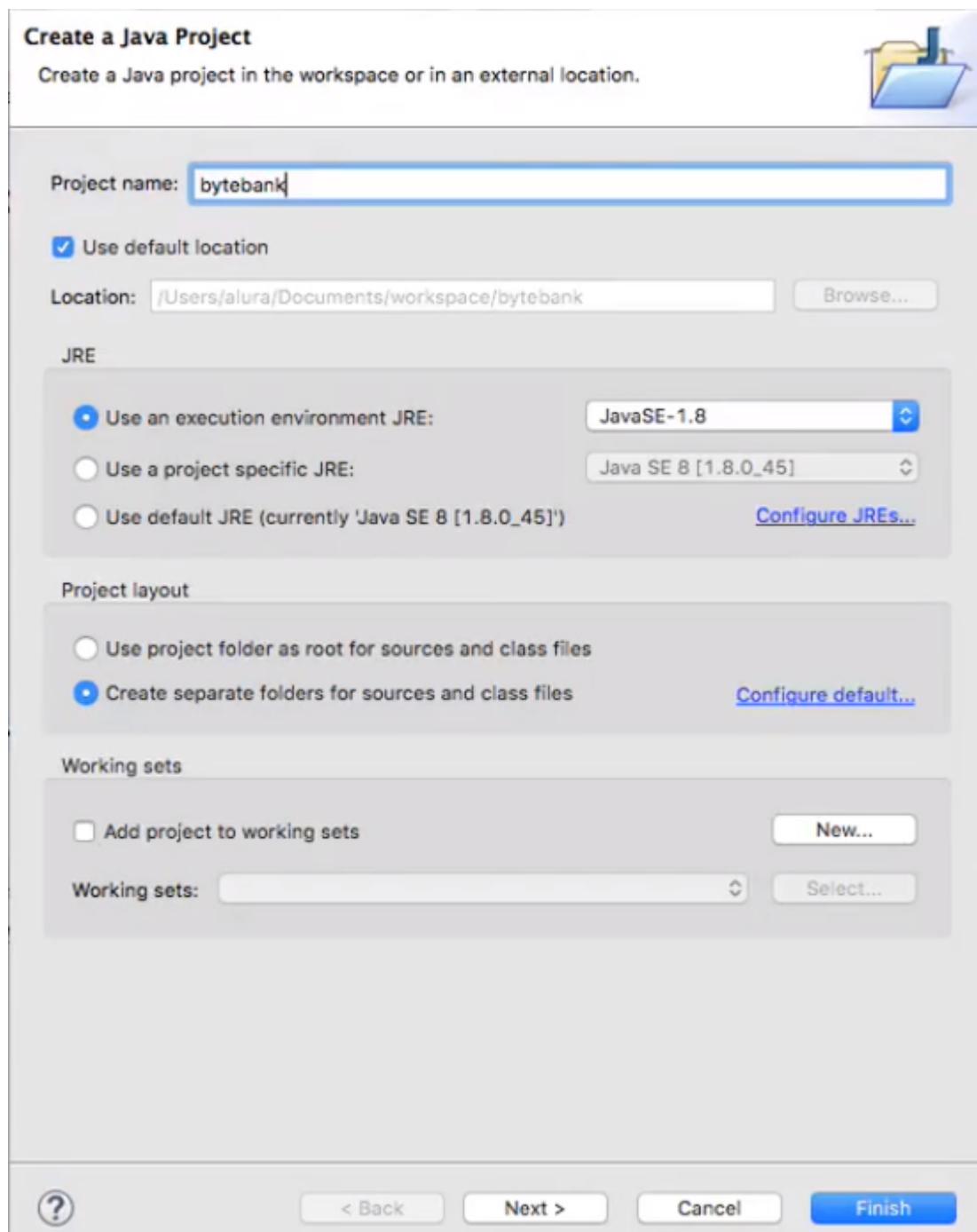
Faremos o **ByteBank**, um projeto de banco da **Alura**. Para isso, precisaremos criar o sistema que o compõe. Iremos descobrir quais dados serão necessários para esse sistema.

A menor unidade que iremos trabalhar é a *conta bancária*, e para isso, precisaremos fazer uma breve pesquisa para sabermos do que ela é composta.

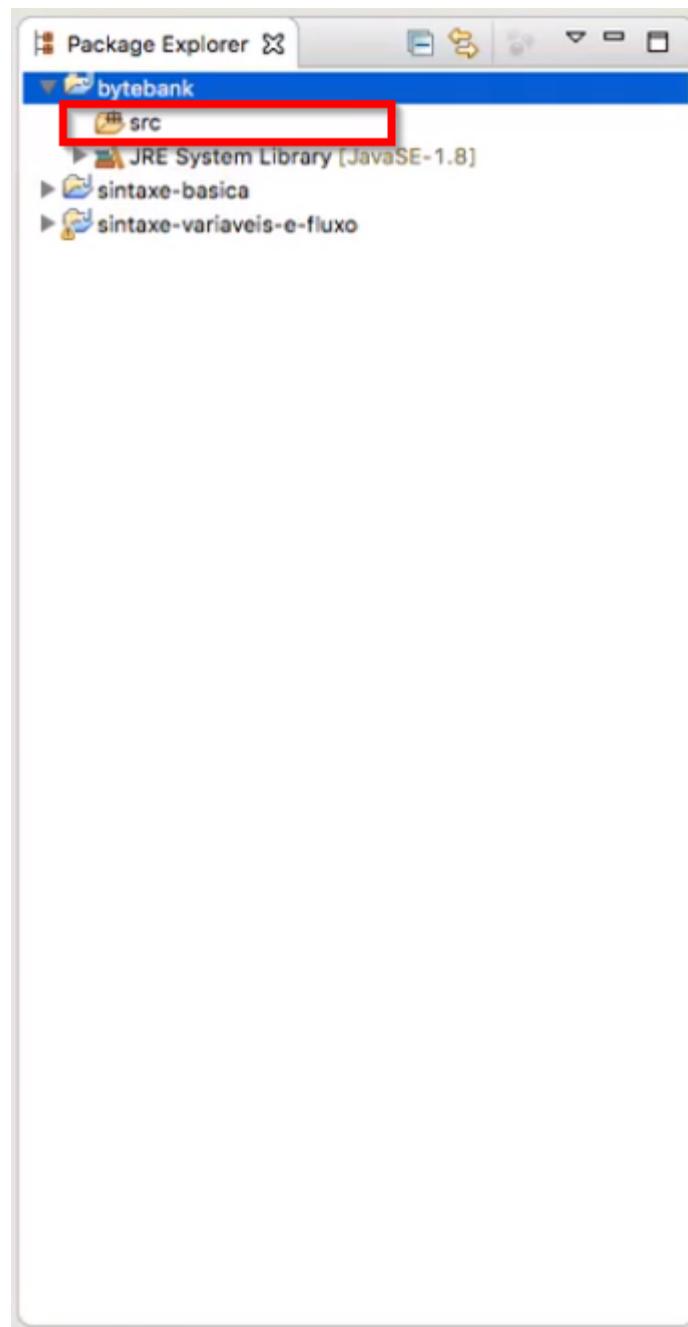
Se olharmos a definição de conta corrente no [WikiPedia](#) (<https://pt.wikipedia.org/wiki/Conta-corrente>), descobriremos uma série de referências acerca de juros e taxas, mas a questão que iremos nos atentar é que as contas bancárias acionam informações diferentes para fins variados.

Façamos uma analogia com uma seguradora de veículos: é importante que ela tenha informações como **marca**, **modelo**, **chassi** e a **cor** do carro. Mas é irrelevante para a seguradora saber quantos cilindros o carro tem ou a versão do motor. Já para uma empresa que fabrica veículos, esses dados fazem parte do **domínio** do problema.

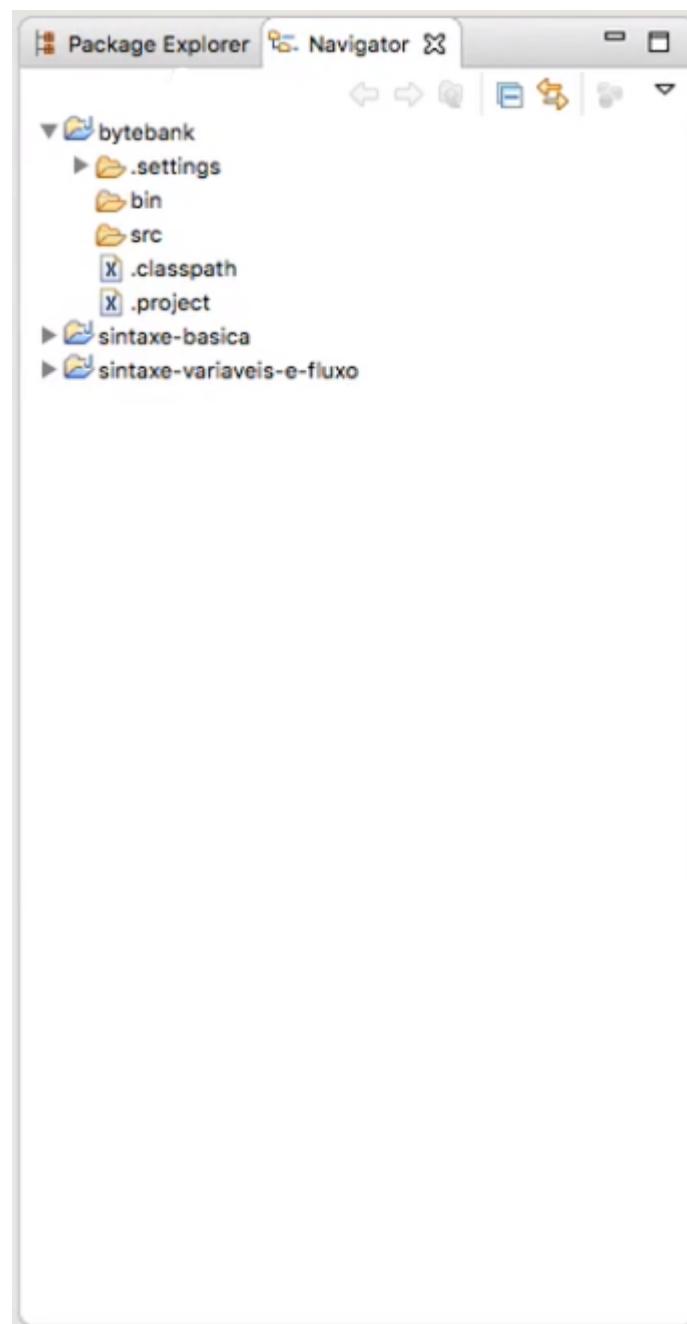
Criaremos um domínio para o nosso sistema baseado em um projeto de banco. No Eclipse, faremos um novo projeto Java intitulado `bytebank`. Depois, clique em "Finish".



Feito isso, será criado um novo projeto na área do *Package Explorer*. Ao clicarmos no projeto `bytebank`, veremos a pasta `src`.



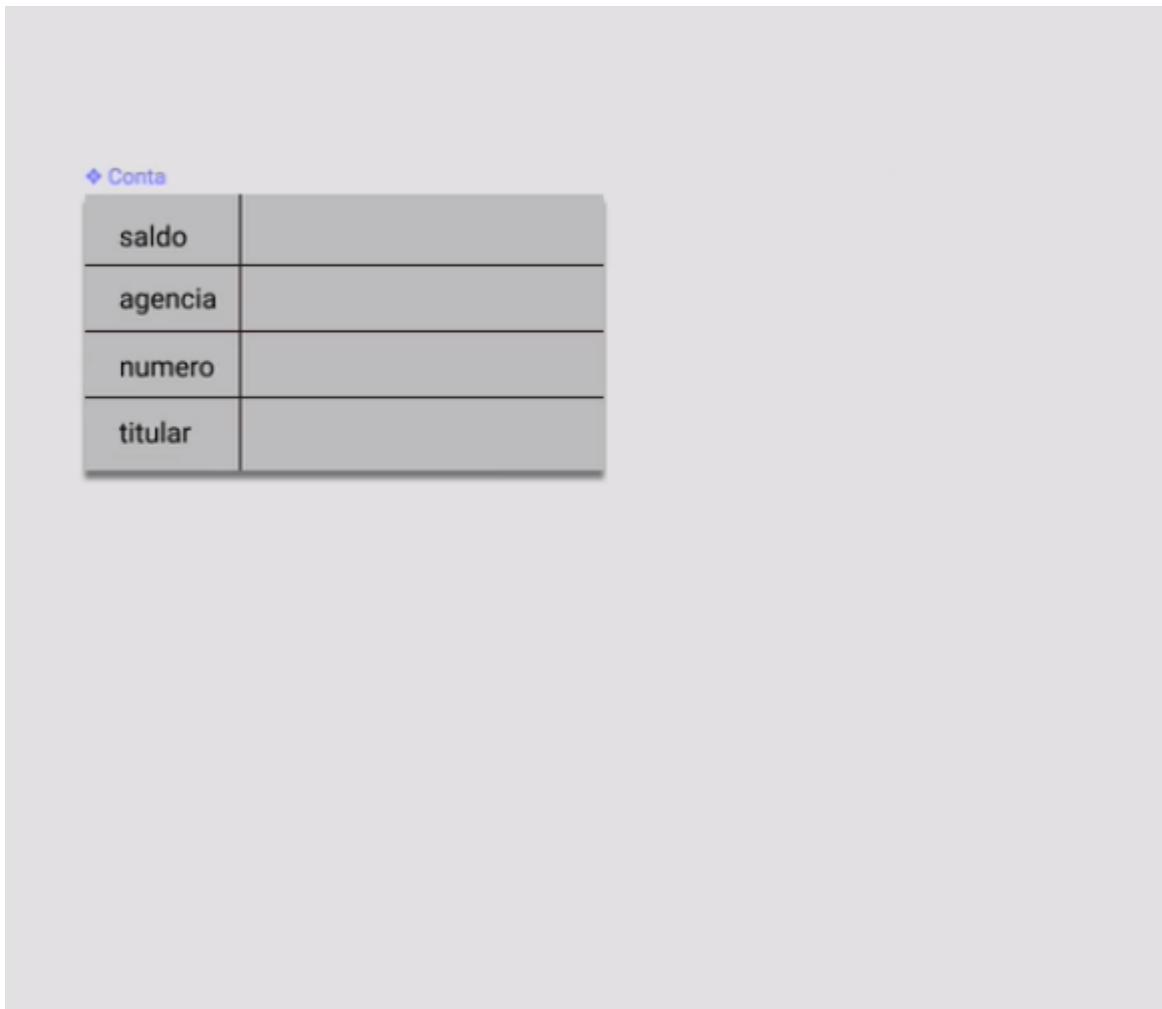
Lembrando: você pode selecionar "Window > Show View > Navigator". O *Navigator* possui uma ideia parecida com o Windows Explorer ou Finder, nele você visualizará mais diretórios internos. O *Package Explorer* oculta a visualização desses diretórios.



Em `src`, criaremos alguns códigos para guardarmos os dados sobre nossas contas bancárias. Uma característica interessante do Java é que ele nos possibilita a criação de **tipos**. Em um "tipo" do Java, armazenaremos todas as informações que constituem uma conta bancária.

Usaremos o [Figma \(<https://www.figma.com/>\)](https://www.figma.com/) - um *software* de criação de interface do usuário - para definirmos as informações básicas de uma conta bancária.

A nossa conta possui quatro características importantes para o nosso banco: **saldo**, **agência**, **número**, **titular**.

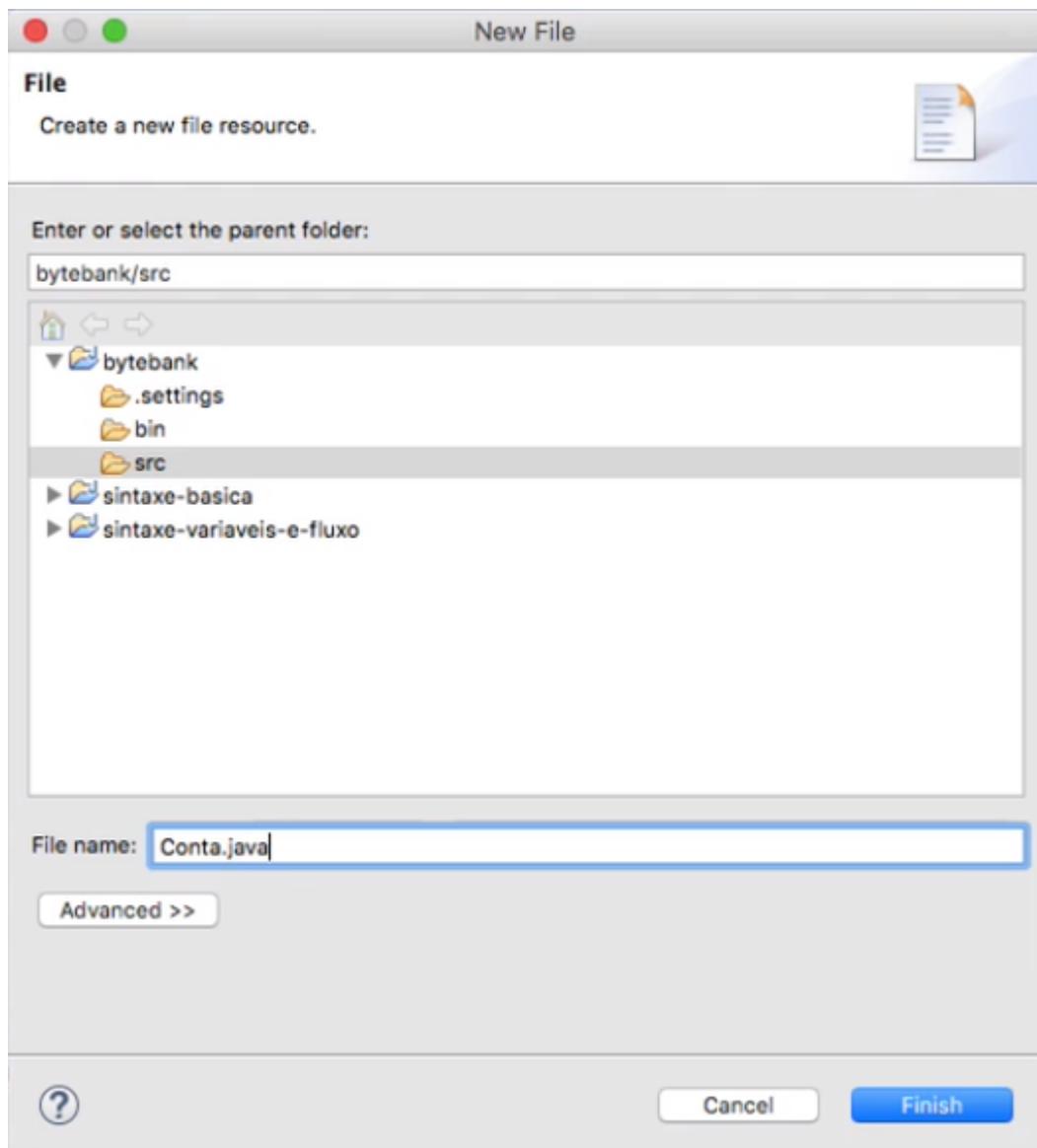


Veremos que no Java não serão guardados apenas os dados de conta bancária, mas também serão atribuídos comportamentos para a conta. Iremos definir algumas transações bancárias típicas, como realizar saques, depósitos e transferências.

O esquema que criamos no Figma pode ser reproduzido no Java, mas antes precisamos refletir: esse quadro com campos a serem preenchidos, pode ser caracterizado como uma conta? Não podemos realizar funções básicas de uma conta bancária apenas com esse quadro cinza, portanto, não se trata de uma conta, e sim, de uma especificação de um tipo `conta`.

Podemos fazer a seguinte analogia: a planta de uma casa não é uma casa, mas o esquema ou design de uma.

Recriaremos esse esquema de dados de conta bancária no Java. Em `src`, criaremos um novo arquivo chamado `Conta.java`.



Faremos a especificação dos componentes do tipo `Conta`.

tipo `Conta`:
 saldo
 agencia

numero

titular

COPiar CÓDIGO

Obviamente, essa não é a sintaxe de Java. Precisaremos utilizar as regras da linguagem para realizarmos com sucesso a especificação do tipo `Conta`.

Escreveremos ao lado de `Conta` a palavra `class`. Trata-se de uma palavra-chave que define um tipo. Para indicarmos que abrimos e fechamos um bloco de informações usaremos as chaves, `{}`.

```
class Conta {  
    saldo  
    agencia  
    numero  
    titular  
}
```

COPiar CÓDIGO

É também necessário declarar o tipo das outras categorias da conta. Para `saldo`, usaremos o tipo `double`, pois ele armazena dados de ponto flutuante.

Os componentes `agencia` e `numero` serão do tipo `int`. Para finalizar, `titular` será uma `String`, que guarda um conteúdo de texto.

Acionaremos o atalho "Ctrl + S" para salvarmos nosso trabalho. Com isso, já temos um código fonte válido em Java que já foi compilado. Ao observarmos a área `Navigator` e selecionarmos o diretório `bin`, veremos o arquivo `Conta.class`.

Porém, não podemos iniciar a execução do programa. Para que um programa possa ser iniciado em Java, ele precisa ter o ponto de entrada `public static void`

`main(String[] args)` , a classe que criamos não possui esse ponto de entrada, pois trata-se de um arquivo de suporte.

Em uma aplicação Java é comum existirem vários arquivos que se comunicam, o primeiro a ser executado contém o método `main` com a "assinatura" (tipo de retorno e parâmetros) acima.

02

Instanciação, atributos e referências

Transcrição

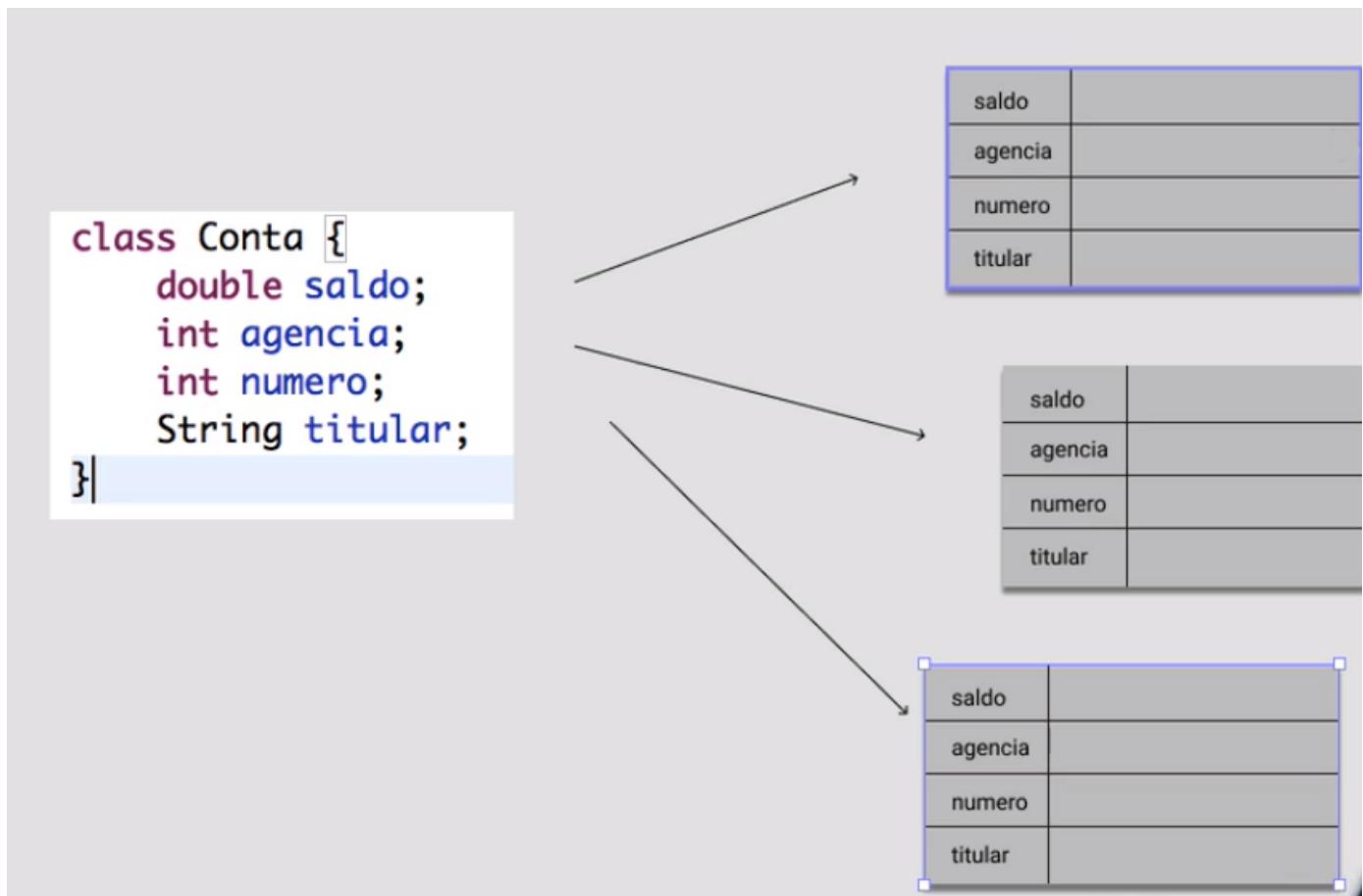
Temos nossa primeira classe `Conta`, que possui quatro características: `saldo`, `agencia`, `numero` e `titular`. O nome que damos para tais características é **atributos**. Diremos que as contas do *ByteBank* possuem quatro atributos.

```
class Conta {  
    double saldo;  
    int agencia;  
    int numero;  
    String titular;  
}
```

[COPIAR CÓDIGO](#)

Ainda não temos uma conta bancária, pois não podemos fazer operações básicas que envolvem uma conta. O que possuímos é - como o quadro cinza feito no Figma - uma especificação de conta. A partir dessa especificação, podemos construir várias contas bancárias individuais.

A tabela cinza, que é produzida através da classe `Conta`, chamaremos de **objeto** ou **instância**. A partir da especificação, construímos objetos ou instâncias do tipo `Conta`.



Da mesma forma que o processo de retirar do papel a planta de uma casa chama-se "construção", no caso da linguagem orientada a objeto, fala-se em "instanciação".

Dada uma classe `Conta`, instanciamos um objeto do tipo `Conta`. Discutiremos esses conceitos com mais profundidade no decorrer do curso.

Na atual condição do nosso código, não podemos depositar dinheiro na nossa conta bancária, mas podemos inserir um texto no campo "saldo", incluindo um valor de 200 reais.

saldo	200
agencia	
numero	
titular	

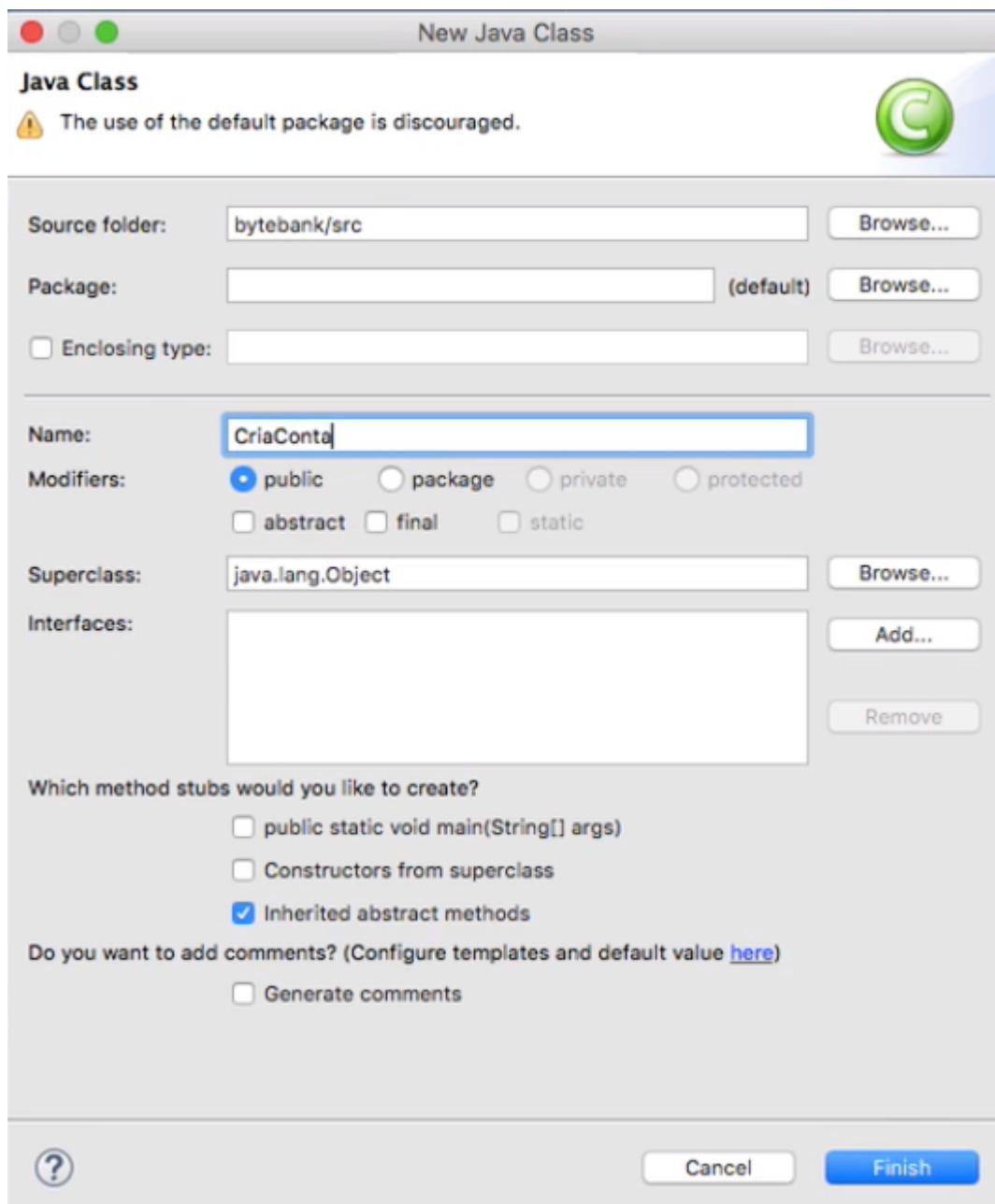
Tratando-se de um objeto do tipo `Conta`, podemos alterar um de seus atributos. Não podemos alterar a especificação do que é uma conta, mas as contas de forma individual.

Voltando ao código, adicionaremos o `public`, pois frequentemente o Eclipse irá inseri-lo automaticamente. Veremos com mais atenção o que isso significa no decorrer do curso.

```
public class Conta {  
    saldo;  
    agencia;  
    numero;  
    titular;  
}
```

[COPIAR CÓDIGO](#)

Na área *Package Explorer* selecionaremos a pasta `default value` e criaremos uma nova classe chamada `CriaConta`.



Essa nova classe faz exatamente o que seu nome anuncia: *cria contas*. Ela testará se o nosso código inicial está de fato funcionando como o esperado.

Na `CriaConta` incluiremos o `main` e, pressionando "Ctrl + Space", será incluído o método `public static void main` que é condição essencial para inicializar uma aplicação, ou seja, rodar o programa.

```
public class CriaConta {
```

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

[COPIAR CÓDIGO](#)

Nosso objetivo é "tirar do papel" o código fonte e construir contas bancárias funcionais a partir dele. Para isso, adicionaremos a palavra-chave `new` e ao lado dela, adicionaremos o nome da classe que servirá para a criação de objetos. Neste caso, será a classe `Conta`. Vamos inserir também dois parênteses `()`. Não se preocupe! Entenderemos sua função posteriormente.

```
public class CriaConta {  
  
    public static void main(String[] args) {  
        new Conta();  
    }  
}
```

[COPIAR CÓDIGO](#)

Temos um programa que, dentro da nossa `main`, instancia - ou constrói - um objeto do tipo `Conta`. Neste ponto, o programa já pode ser executado, embora ainda esteja simples e não gere nenhum resultado, é um programa válido e é isso que queríamos construir.

Nosso próximo passo é escrever no código que o valor do saldo de uma conta específica vale `200`. Podemos incluir essa informação no código da seguinte maneira:

```
public class CriaConta {
```

```
public static void main(String[] args) {  
    new Conta();  
    saldo = 200;  
}  
}
```

[COPIAR CÓDIGO](#)

Veremos que a palavra `saldo` não é identificada, pois no nosso contexto não existe nenhuma variável denominada `saldo`. Precisamos também criar um mecanismo de referenciação, ou seja, queremos assinalar que o saldo de `200` reais é referente à uma conta específica. Podemos fazer isso guardando o retorno de `new Conta()` em uma variável. Chamaremos essa variável de `primeiraConta`.

```
public class CriaConta {  
  
    public static void main(String[] args) {  
        primeiraConta = new Conta();  
        saldo = 200;  
    }  
}
```

[COPIAR CÓDIGO](#)

A variável `primeiraConta` também não será compilada, pois ela não existe no contexto. Mas quando formos declará-la ela será do tipo `Conta`.

```
public class CriaConta {  
  
    public static void main(String[] args) {  
        Conta primeiraConta = new Conta();  
        saldo = 200;  
    }  
}
```

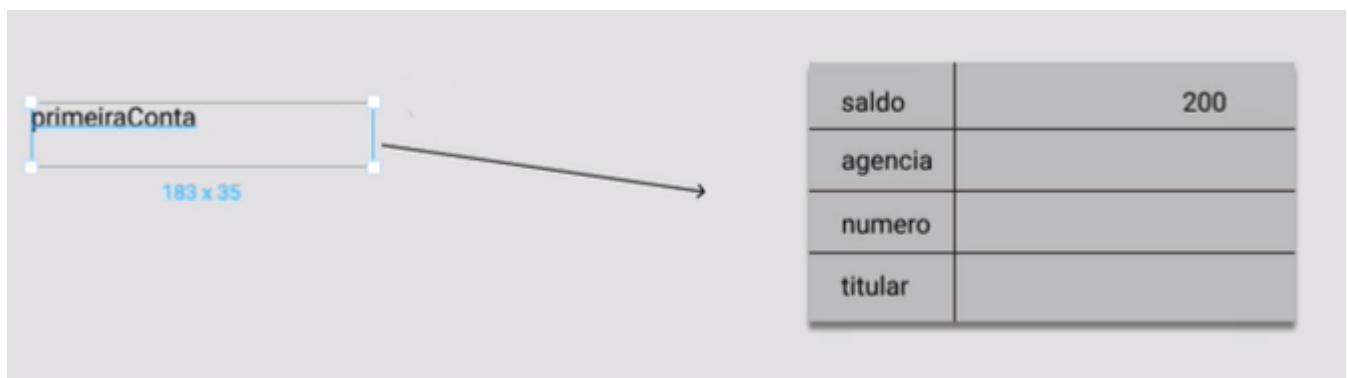
[COPIAR CÓDIGO](#)

Pode soar estranho ter de escrever o tipo `Conta` à direita e à esquerda da variável, mas existe o conceito de polimorfismo no Java que será desenvolvido futuramente durante o curso.

Algumas ideias ficam nebulosas nos primeiros passos no Java, mas precisamos seguir essa curva de aprendizado para começarmos a escrever códigos básicos.

Conseguimos referenciar uma conta específica. É comum ter a impressão de que a palavra-chave `new` nos devolve o objeto em si, e de que a variável `primeiraConta` contém o objeto, mas isso nunca ocorre.

No Java, assim como em outras linguagens, um objeto **nunca** está dentro de uma variável. O que tem dentro de uma variável é somente uma indicação a um objeto específico, uma *referência* a um objeto específico.



No nosso código, especificaremos que o valor de `saldo` a ser exibido é referente à `primeiraConta`. A navegação entre `primeiraConta` e `saldo` ocorre através do caractere `.`

```
public class CriaConta {
```

```
public static void main(String[] args) {  
    Conta primeiraConta = new Conta();  
    primeiraConta.saldo = 200;  
}  
}
```

[COPIAR CÓDIGO](#)

Ao rodar o programa, nada acontece. O que falta é *mostrar* o valor do saldo na tela. Vamos imprimir o valor 200 acessando o atributo saldo de primeiraConta . Através do atalho sysout e depois "Ctrl + Space", será mostrado o System.out.println() , e então imprimiremos a variável primeiraConta acessando o seu saldo através do caractere ..

```
public class CriaConta {  
  
    public static void main(String[] args) {  
        Conta primeiraConta = new Conta();  
        primeiraConta.saldo = 200;  
        System.out.println(primeiraConta.saldo);  
    }  
}
```

[COPIAR CÓDIGO](#)

Uma dica sobre o atalho "Ctrl+ Space": toda a vez que possui variáveis com nomes grandes, podemos acionar esse atalho e sugestões serão apresentadas.

The screenshot shows an IDE interface with two tabs: 'Conta.java' and '*CriaConta.java'. The '*CriaConta.java' tab is active, displaying the following Java code:

```
1
2 public class CriaConta {
3
4     public static void main(String[] args) {
5         Conta primeiraConta = new Conta();
6         primeiraConta.saldo = 200;
7         System.out.println(primeiraConta);
8     }
9 }
10
```

A code completion tooltip is open at the line 'System.out.println(primeiraConta);'. The tooltip lists several suggestions, all starting with 'primeiraConta':

- ① primeiraConta : Conta
- ② PrimitiveIterator - java.util
- ③ PrimitiveType - javax.lang.model.type
- ④ Principal - java.security
- ⑤ Principal - org.omg.CORBA
- ⑥ PrincipalHolder - org.omg.CORBA
- ⑦ Printable - java.awt.print
- ⑧ PrintConversionEvent - javax.xml.bind
- ⑨ PrintConversionEventImpl - javax.xml.bind.helpers
- ⑩ PrinterAbortException - java.awt.print
- ⑪ PrinterException - java.awt.print

At the bottom of the tooltip, it says 'Press '^Space' to show Template Proposals'.

Ao executarmos aplicação, o resultado será o valor do saldo da conta específica referenciada, ou seja, 200 .

```
1  
2 public class CriaConta {  
3  
4     public static void main(String[] args) {  
5         Conta primeiraConta = new Conta();  
6         primeiraConta.saldo = 200;  
7         System.out.println(primeiraConta.saldo);  
8     }  
9 }  
10
```

The screenshot shows a Java application window. The title bar says "CriaConta [Java Application]". The status bar at the bottom indicates the path: "/Library/Java/JavaVirtualMachines/jdk1.8.0_45.jdk/Contents/Home/bin/java" and the date/time: "(Apr 7, 2017, 3:4)". The main area displays the output of the program: "200.0". The number "200.0" is highlighted with a red rectangular box.

05

Segunda Instância

Transcrição

Iremos trabalhar mais com a `primeiraConta`. Lembrando que nosso código se encontra deste modo:

```
public class CriaConta {  
  
    public static void main(String[] args) {  
        Conta primeiraConta = new Conta();  
        primeiraConta.saldo = 200;  
        System.out.println(primeiraConta.saldo);  
    }  
}
```

[COPIAR CÓDIGO](#)

Além de atribuirmos valores utilizando sinal `=`, podemos fazer uso de outros elementos da sintaxe básica do Java, como `+=`. Este elemento significa que o valor final de `primeiraConta` é aquele que já foi atribuído anteriormente (`200`) mais `100`.

```
public class CriaConta {  
  
    public static void main(String[] args) {  
        Conta primeiraConta = new Conta();  
        primeiraConta.saldo = 200;  
        System.out.println(primeiraConta.saldo);  
    }  
}
```

```
primeiraConta.saldo += 100;  
System.out.println(primeiraConta.saldo);  
}  
}
```

[COPIAR CÓDIGO](#)

Ao executarmos a aplicação, veremos que será impresso o valor de 300 .

The screenshot shows an IDE interface with two tabs: 'Conta.java' and 'CriaConta.java'. The 'CriaConta.java' tab is active, displaying the following code:

```
1  
2 public class CriaConta {  
3  
4     public static void main(String[] args) {  
5         Conta primeiraConta = new Conta();  
6         primeiraConta.saldo = 200;  
7         System.out.println(primeiraConta.saldo);  
8  
9         primeiraConta.saldo += 100;  
10        System.out.println(primeiraConta.saldo);  
11    }  
12 }  
13
```

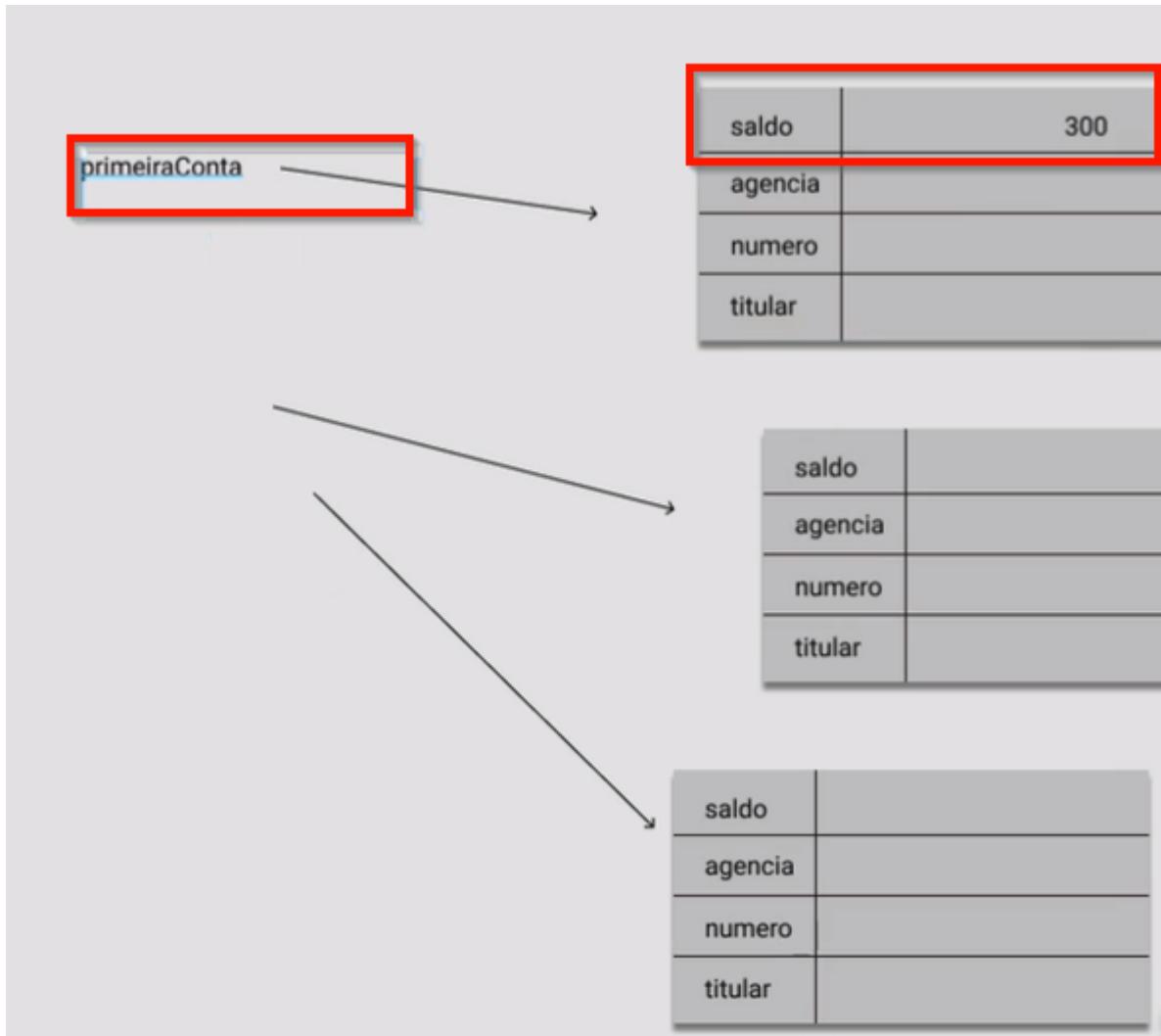
Below the code editor is a 'Console' tab showing the application's output:

```
Problems @ Javadoc Declaration Console  
<terminated> CriaConta [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_45.jdk/Contents/Home/bin/java (Apr 7, 2017, 4:00:00 PM)  
200.0  
300.0
```

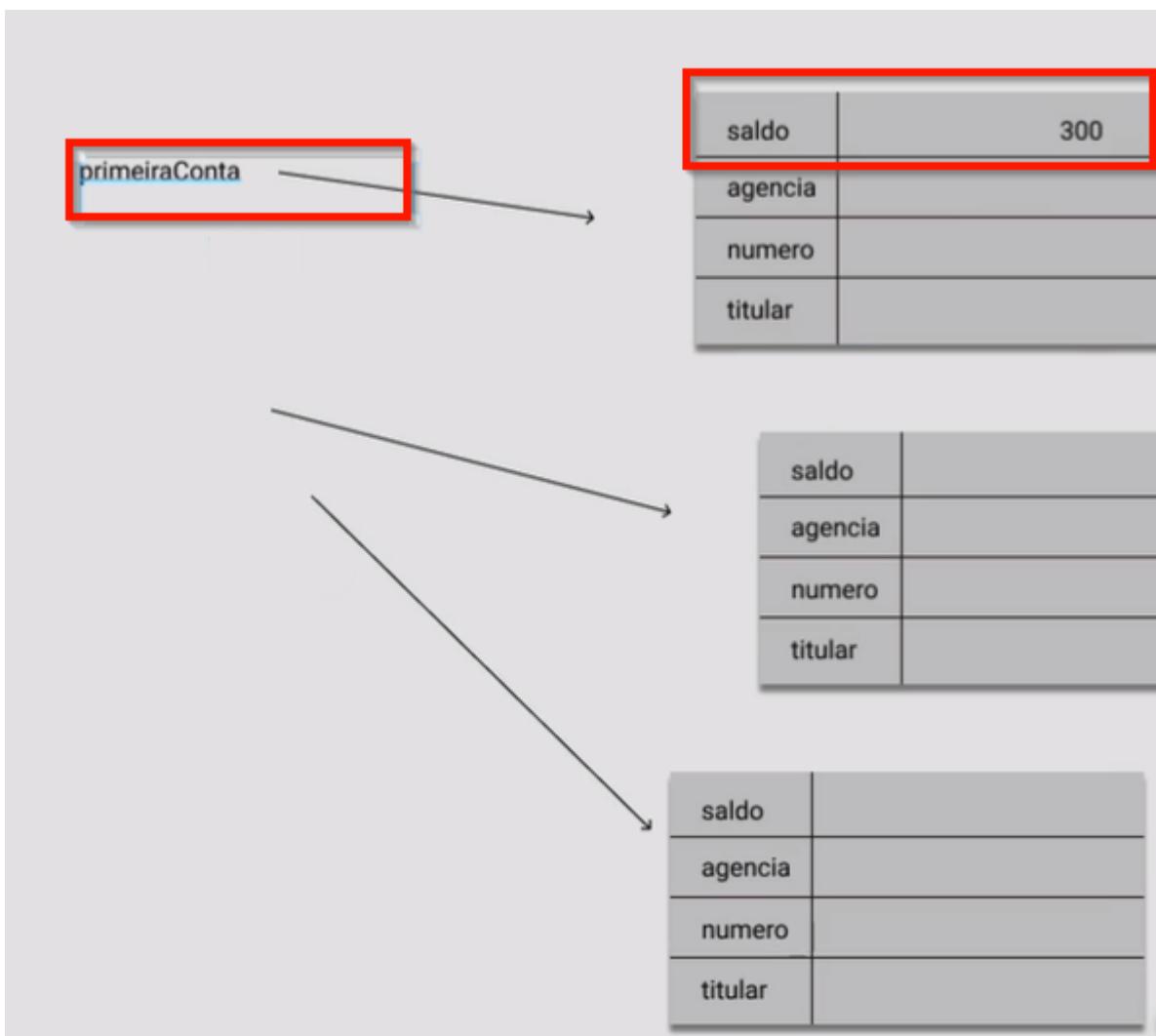
The line '300.0' is highlighted with a red box.

O novo valor de `primeiraConta.saldo` é o resultado da soma do saldo anterior 200 mais 100 . Chegando ao valor de 300 . A variável `primeiraConta` é uma referência

uma conta específica, que chamamos de **objeto**.



Iremos modificar outras contas do nosso banco. Faremos uma referência à outra conta bancária através da referência `segundaConta`.



Como já vimos, daremos uma instanciação através do nosso código base embutido em `Conta`. Com isso, já temos os atributos criados (`saldo`, agência, número e titular) na conta nova que iremos trabalhar. Adicionaremos, também, a palavra chave `new`. A informação devolvida por `new` será guardada dentro da referência `segundaConta`, que será uma variável do tipo `Conta`.

```
public class CriaConta {  
  
    public static void main(String[] args) {  
        Conta primeiraConta = new Conta();  
        primeiraConta.saldo = 200;  
        System.out.println(primeiraConta.saldo);  
    }  
}
```

```
primeiraConta.saldo += 100;  
System.out.println(primeiraConta.saldo);  
  
Conta segundaConta = new Conta();  
}  
}
```

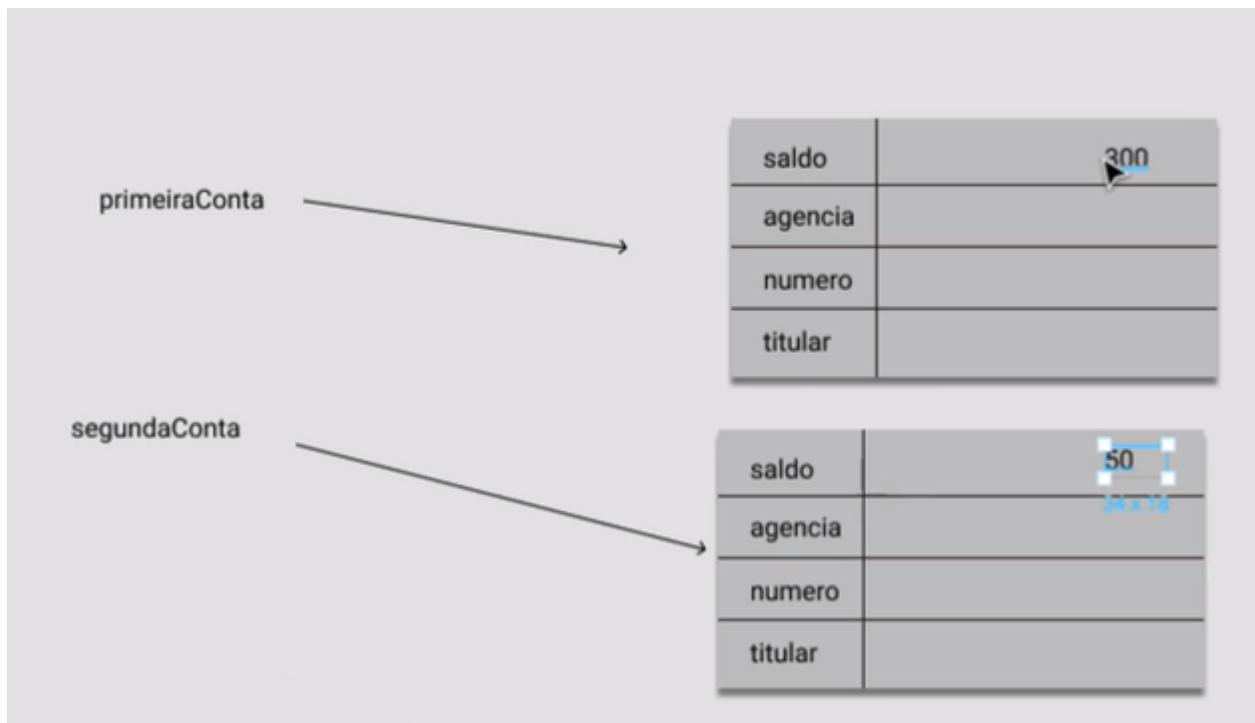
[COPIAR CÓDIGO](#)

Declararemos que em `segundaConta` há um saldo de 50 .

```
public class CriaConta {  
  
    public static void main(String[] args) {  
        Conta primeiraConta = new Conta();  
        primeiraConta.saldo = 200;  
        System.out.println(primeiraConta.saldo);  
  
        primeiraConta.saldo += 100;  
        System.out.println(primeiraConta.saldo);  
  
        Conta segundaConta = new Conta();  
        segundaConta.saldo = 50;  
    }  
}
```

[COPIAR CÓDIGO](#)

Portanto, `segundaConta` possui um saldo de 50 reais. Esse valor não possui qualquer ligação com o saldo de `primeiraConta` .



Para testarmos essa individualidade das contas, podemos pedir o saldo de `primeiraConta` e adicionar uma String "primeira conta tem" . Faremos o mesmo procedimento com `segundaConta` .

Lembre-se de utilizar o + para a concatenação dos elementos.

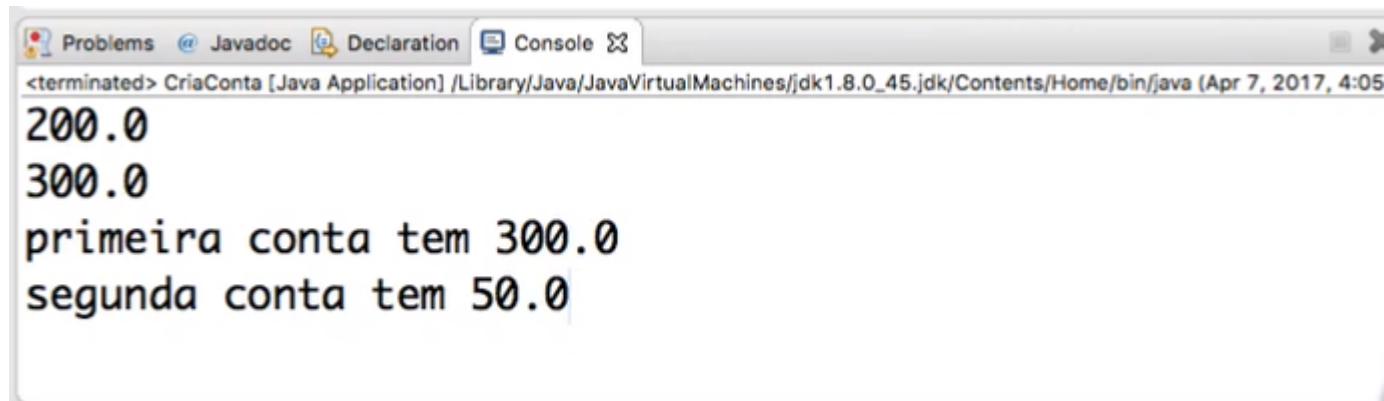
```
public class CriaConta {  
  
    public static void main(String[] args) {  
        Conta primeiraConta = new Conta();  
        primeiraConta.saldo = 200;  
        System.out.println(primeiraConta.saldo);  
  
        primeiraConta.saldo += 100;  
        System.out.println(primeiraConta.saldo);  
  
        Conta segundaConta = new Conta();  
        segundaConta.saldo = 50;
```

```
System.out.println("primeira conta tem " + primeiraConta.saldo);
System.out.println("segunda conta tem " + segundaConta.saldo);
}
}
```

[COPIAR CÓDIGO](#)



Ao executarmos a aplicação, veremos o saldos das contas apresentando individualmente, pois são instâncias diferentes.



Não podemos simplesmente escrever no nosso código `saldo = 50`, pois o programa não achará a variável `saldo` no escopo, e ainda que achasse, existem múltiplos saldos devido a variedade de contas bancárias. Por isso, sempre devemos escrever a referência `. atributo`, ou seja, trabalhar de uma forma *orientada ao objeto*.

 06

Valores default de atributos

Transcrição

Vimos que os tipos de variáveis apresentadas no primeiro curso não possuíam valor padrão, ou seja, valor *default*. Não era possível se quer executar a aplicação com esse tipo de variável, pois não havia valores definidos.

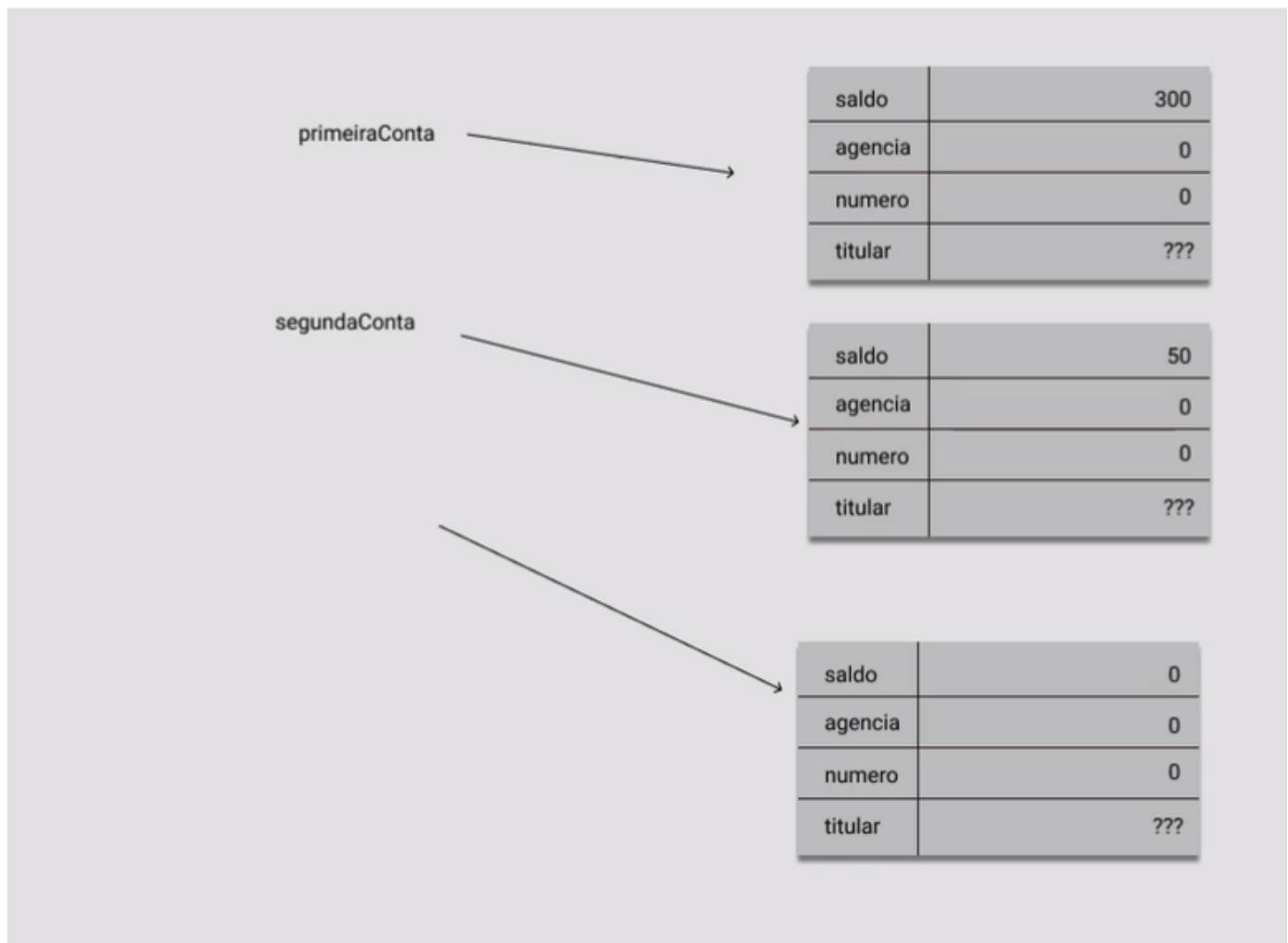
O tipos de variáveis que estamos trabalhando na classe `Conta`, não são as mesmas que ficam dentro do método `main`. As variáveis que estamos nos referindo ficam diretamente na classe.

Essas variáveis nós chamamos de *atributos* ou características de um objeto, comparativamente, elas são similares às especificações da planta de uma casa: se terá quatro quartos, uma sala, um banheiro. No caso de uma conta bancária, seus atributos seriam agência, conta, titular. Esses atributos podem ser chamados de **campo** ou **propriedade**.

Esse tipo de variável especial que possui significado para objetos tem um comportamento diferente. Quando acionamos a palavra-chave `new` e o Java instancia o objeto, todo os campos são zerados.

Para entendermos melhor essa ideia, observaremos os cartões cinzas que representam as especificações das nossas contas bancárias.

Percebam que os campos que não possuem valores estipulados estão zerados. Como `titular` é um tipo `String`, ou seja, não numérico, utilizamos as interrogações.



Para vermos como isso está representado no Eclipse, iremos na nossa classe `CriaConta` e solicitaremos a impressão do valor de `agencia` e `numero` de `primeiraConta`.

```
public class CriaConta {
    public static void main(String[] args) {
        Conta primeiraConta = new Conta();
        primeiraConta.saldo = 200;
        System.out.println(primeiraConta.saldo);
        primeiraConta.saldo += 100;
        System.out.println(primeiraConta.saldo);

        Conta segundaConta = new Conta();
        segundaConta.saldo = 50;
```

```
System.out.println("primeira conta tem " + primeiraConta.saldo);
System.out.println("segunda conta tem " + segundaConta.saldo);

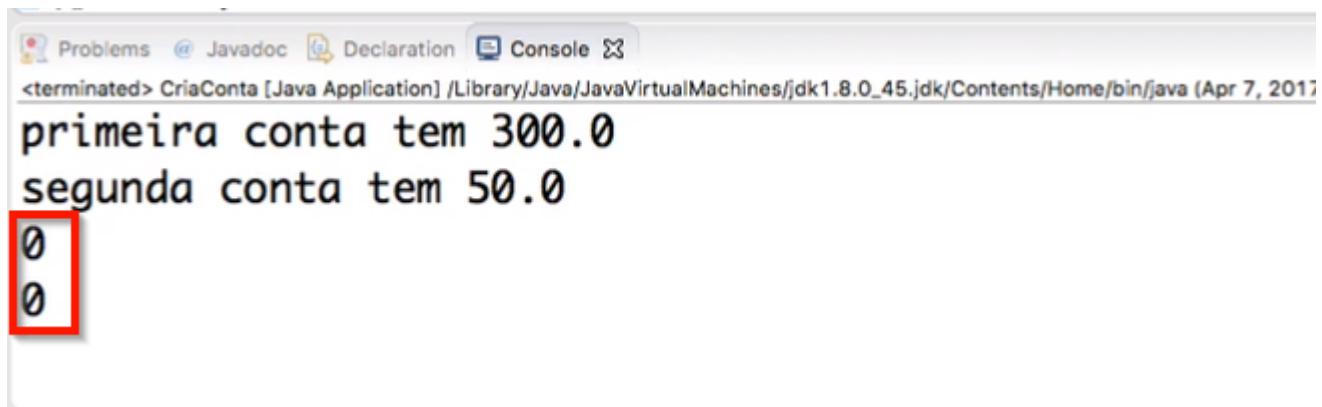
        System.out.println(primeiraConta.agencia);
        System.out.println(primeiraConta.numero);
    }

}
```

COPIAR CÓDIGO



Ao solicitarmos a execução da aplicação, teremos o resultado dos valores de `agencia` e `numero`, que como sabemos, será `0` para ambos os casos. Percebem que não houve erros na execução, mesmo não existindo valores definidos para estes atributos.



Quando o Java constrói objetos, todos os seus atributos são zerados. `0` é o valor *default* de vários tipos numéricos, como `int`, `double` e `long`. No caso do tipo `boolean` o valor é *false*.

Poderíamos configurar valores padrão diferentes de zero, mas isso não seria interessante no nosso caso. Por exemplo, poderíamos dizer o valor de `agencia` sempre será `42` para todas as contas bancárias.

```
public class Conta {
    double saldo;
```

```
int agencia = 42;  
int numero;  
String titular;  
}
```

[COPIAR CÓDIGO](#)

A agência de todas as contas bancárias será 42 . Podemos verificar isso solicitando a impressão de agencia da segundaConta e da primeiraConta .

```
public class CriaConta {  
    public static void main(String[] args) {  
        Conta primeiraConta = new Conta();  
        primeiraConta.saldo = 200;  
        System.out.println(primeiraConta.saldo);  
  
        primeiraConta.saldo += 100;  
        System.out.println(primeiraConta.saldo);  
  
        Conta segundaConta = new Conta();  
        segundaConta.saldo = 50;  
  
        System.out.println("primeira conta tem " + primeiraConta.saldo);  
        System.out.println("segunda conta tem " + segundaConta.saldo);  
  
        System.out.println(primeiraConta.agencia);  
        System.out.println(primeiraConta.numero);  
  
        System.out.println(segundaConta.agencia);  
    }  
}
```

[COPIAR CÓDIGO](#)

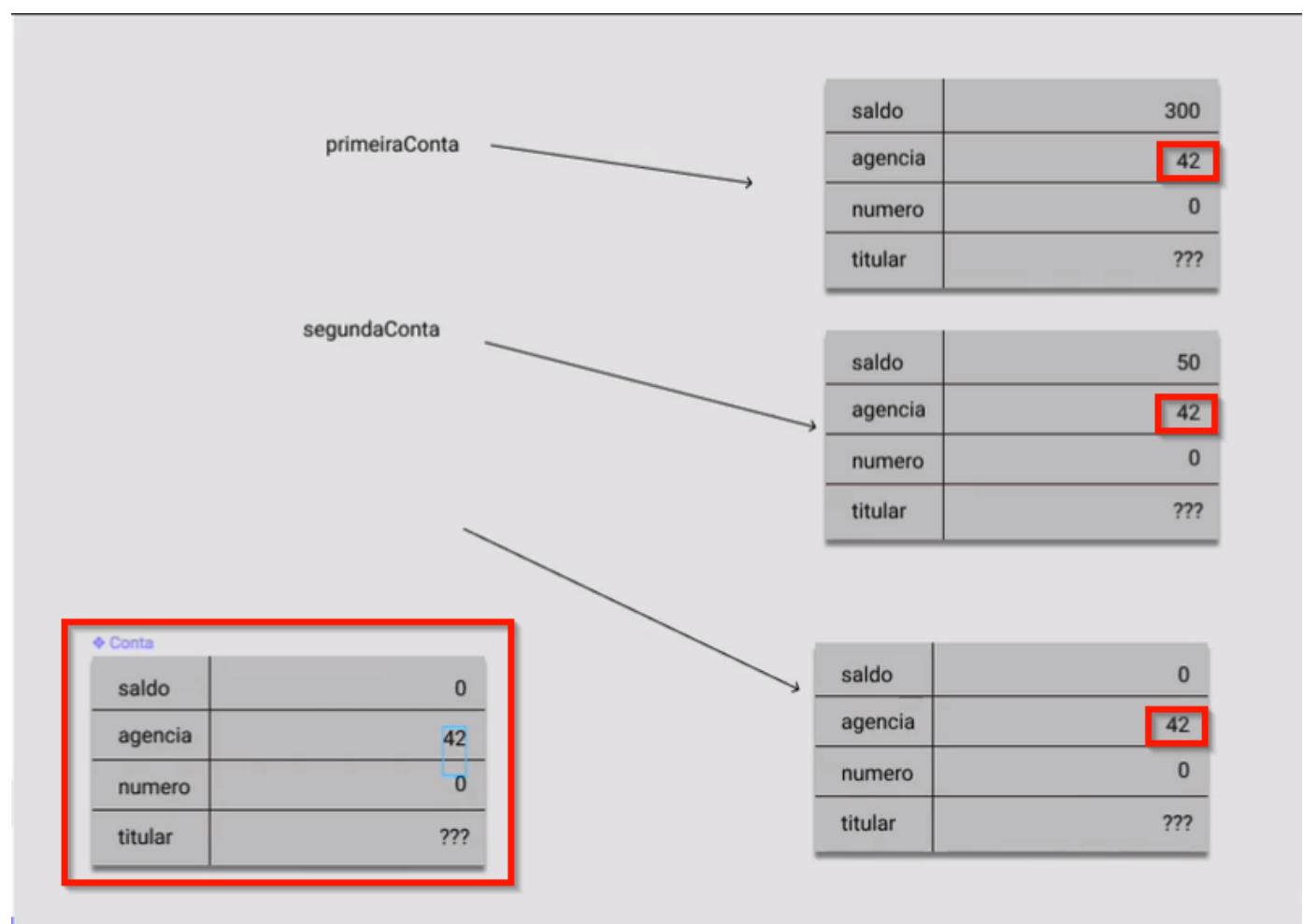
O resultado será 42 para ambos os casos.

```

<terminated> CriaConta [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_45.jdk/Contents/Home
segunda conta tem 50.0
42
0
42

```

Configuramos o valor 42 como default para todas as agências. Podemos representar essa ideia tomando como configuração padrão o cartão cinza no canto esquerdo da tela.



Podemos alterar o valor no objeto em si. Como por exemplo, redefiniremos um novo valor de `agencia` para `segundaConta`, que será 146 .

```
public class CriaConta {  
    public static void main(String[] args) {  
        Conta primeiraConta = new Conta();  
        primeiraConta.saldo = 200;  
        System.out.println(primeiraConta.saldo);  
  
        primeiraConta.saldo += 100;  
        System.out.println(primeiraConta.saldo);  
  
        Conta segundaConta = new Conta();  
        segundaConta.saldo = 50;  
  
        System.out.println("primeira conta tem " + primeiraConta.saldo);  
        System.out.println("segunda conta tem " + segundaConta.saldo);  
  
        System.out.println(primeiraConta.agencia);  
        System.out.println(primeiraConta.numero);  
  
        System.out.println(segundaConta.agencia);  
  
        segundaConta.agencia = 146;  
        System.out.println("agora a segunda conta está na agencia " +  
    }  
}
```

[COPIAR CÓDIGO](#)



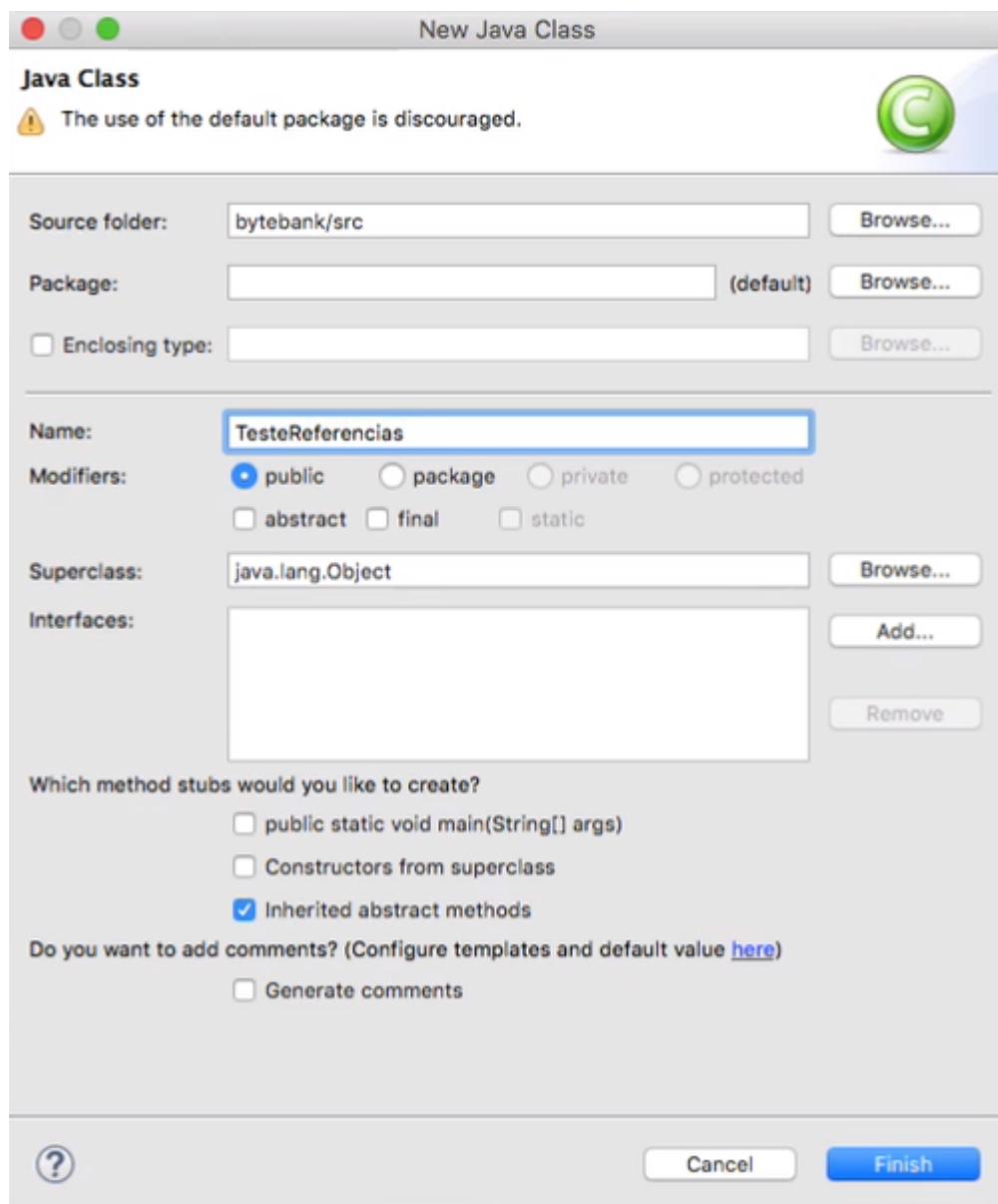
Com isso, modificamos o valor de `agencia` de `segundaConta` para 146 .

08

Referências vs Objetos

Transcrição

Nos atentaremos para um detalhe que causa confusão mesmo em programadores mais experientes. Criaremos uma nova classe chamada `TesteReferencias`.

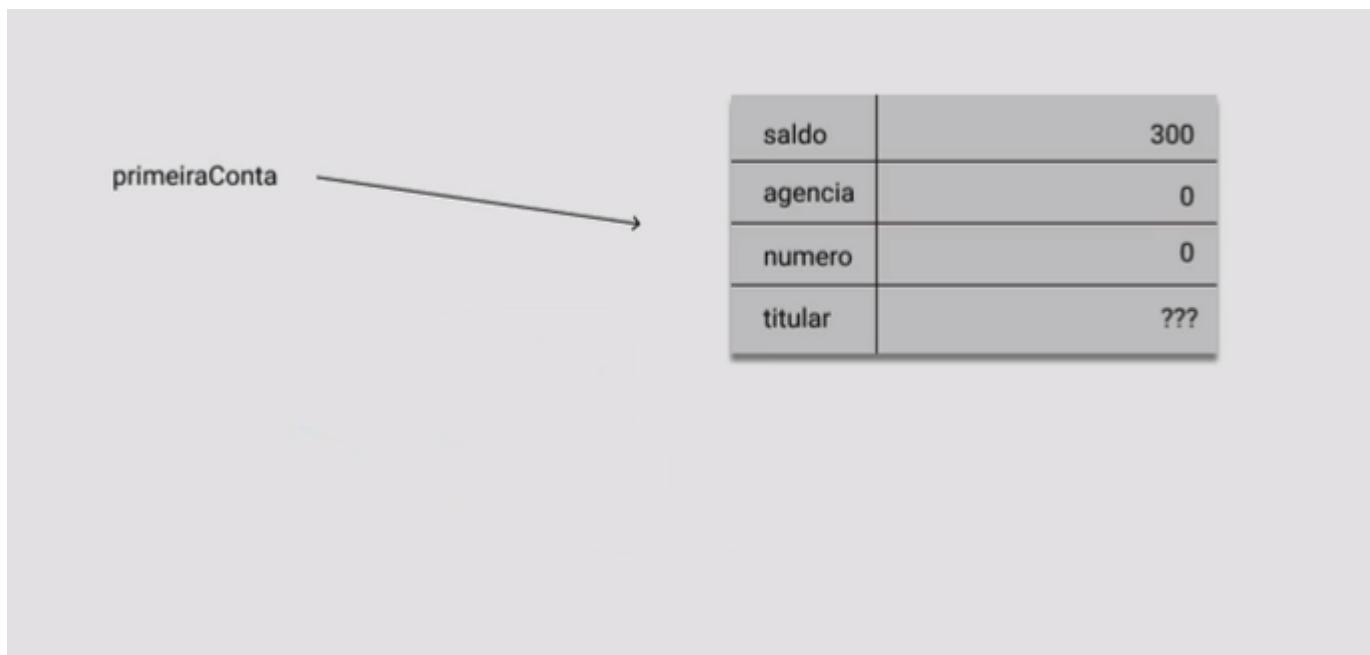


Nesta nova classe, adicionaremos a `main`, e criaremos uma conta chamada `primeiraConta`. Estipularemos o valor de `300` para `saldo`.

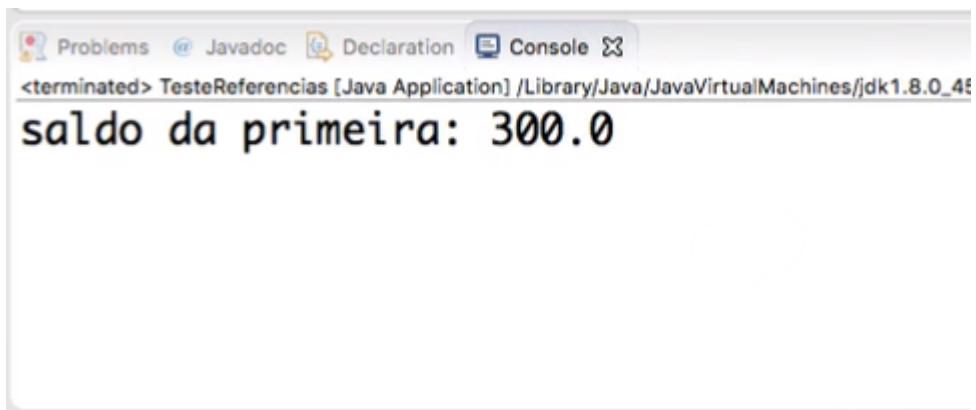
```
public class TesteReferencias {  
    public static void main(String [] args) {  
        Conta primeiraConta = new Conta();  
        primeiraConta.saldo = 300;  
  
        System.out.println("saldo da primeira: " + primeiraConta.sa
```

COPIAR CÓDIGO

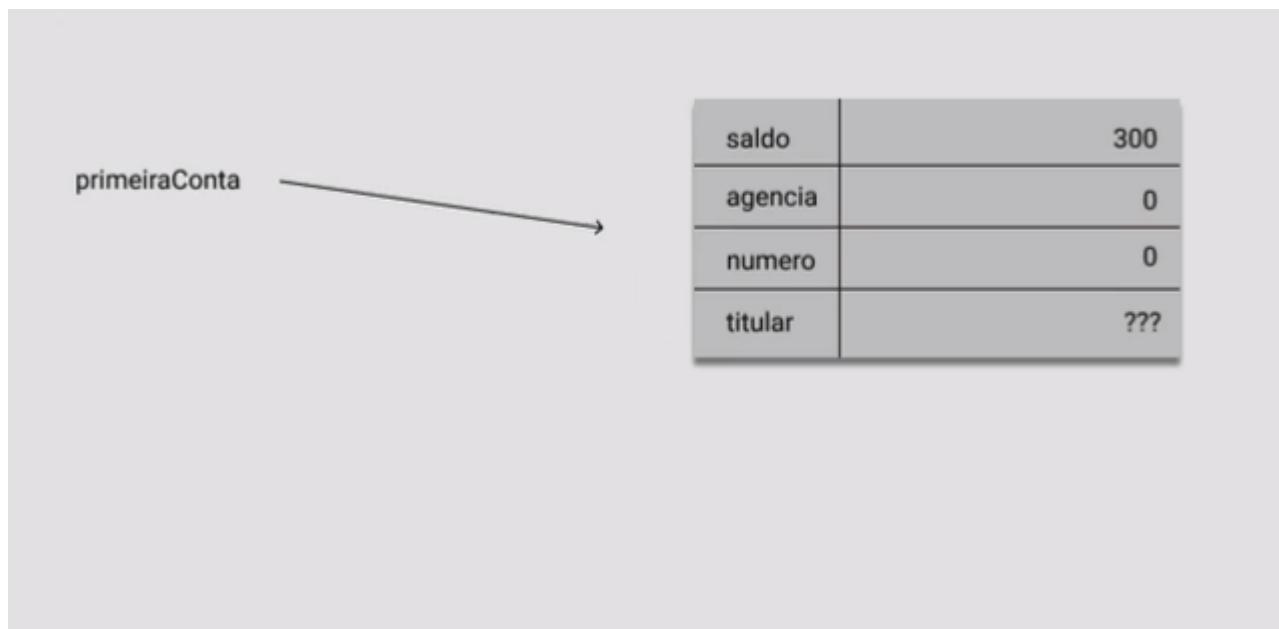
Temos a seguinte representação no cartão cinza:



Executaremos o programa e verificaremos que o código está funcional.



Nos atentaremos para uma questão importante: a diferença entre o tipo (`Conta`) e a variável desse tipo (`primeiraConta`). A variável não é um objeto `Conta`, e sim, uma indicação a um objeto específico, uma *referência* de um objeto. Sua representação gráfica seria a flecha que referencia o objeto.



Observem a seguinte declaração:

```
public class TesteReferencias {  
    public static void main(String[] args) {  
        Conta primeiraConta = new Conta();  
        primeiraConta.saldo = 300;  
  
        System.out.println("saldo da primeira: " + primeiraConta);  
    }  
}
```

```

    Conta segundaConta = primeiraConta;
}

}

```

[COPIAR CÓDIGO](#)

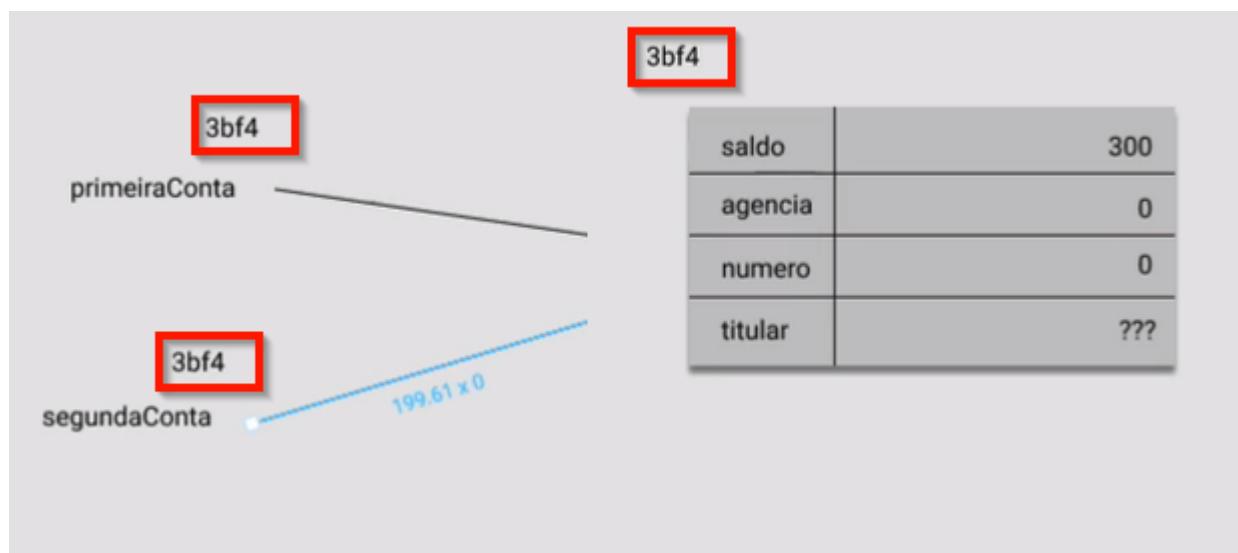


A princípio, podemos pensar que esta declaração gera uma cópia da `primeiraConta` para a `segundaConta` e teríamos uma espécie de "clone de objeto".

Lembrem-se = no Java copia o que está na direita e cola na esquerda.

A questão é que não há um objeto `Conta` à direita, e sim uma **referência**. O que copiamos é a referência para um mesmo objeto.

Pense da seguinte forma: existe uma espécie de Id dos objetos, que chamaremos de `3bf4`. A variável `primeiraConta` possui o valor `3bf4`, fazendo referência ao `Id` do objeto. Quando declaramos que `primeiraConta = segundaConta`, na verdade estamos copiando esse Id `3bf4` que é a referência, e não o objeto em si.



O que temos são duas referências para o mesmo objeto. É como se duas cartas fossem endereçadas ao mesmo local. Embora sejam cartas diferentes, possuem o

mesmo destino.

Veremos qual é o resultado dessa dupla referenciação no nosso código. Faremos o `sysout` no saldo de `segundaConta`.

```
public class TesteReferencias {  
    public static void main(String[] args) {  
        Conta primeiraConta = new Conta();  
        primeiraConta.saldo = 300;  
  
        System.out.println("saldo da primeira: " + primeiraConta.sa...  
  
        Conta segundaConta = primeiraConta;  
    }  
}
```

COPIAR CÓDIGO

Ao executarmos o código, veremos que o programa irá imprimir o valor 300, pois temos duas variáveis, e não dois objetos.

Prosseguiremos com um desafio mais complicado. Primeiramente, vamos incluir mais 100 reais em `segundaConta`.

```
public class TesteReferencias {  
    public static void main(String[] args) {  
        Conta primeiraConta = new Conta();  
        primeiraConta.saldo = 300;  
  
        System.out.println("saldo da primeira: " + primeiraConta.sa...  
  
        Conta segundaConta = primeiraConta;
```

```
System.out.println("saldo da segunda conta: " + segundaConta.saldo);
}
}
```

[COPIAR CÓDIGO](#)

Ao executarmos a aplicação teremos um saldo de 400 reais para `segundaConta`. Feito isso, iremos acionar o `sysout` no saldo de `primeiraConta`.

```
public class TesteReferencias {
    public static void main(String[] args) {
        Conta primeiraConta = new Conta();
        primeiraConta.saldo = 300;

        System.out.println("saldo da primeira: " + primeiraConta.saldo);

        Conta segundaConta = primeiraConta;

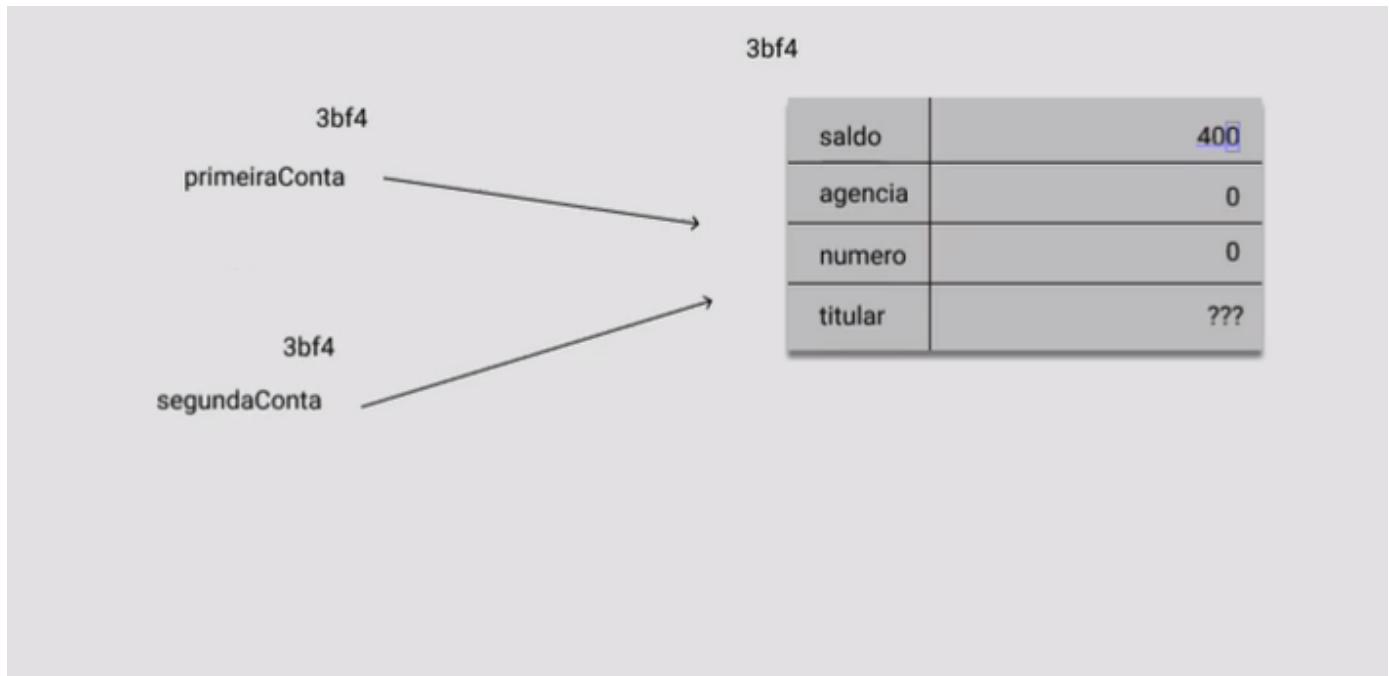
        System.out.println("saldo da segunda conta: " + segundaConta.saldo);

        segundaConta.saldo += 100;
        System.out.println("saldo da segunda conta " + segundaConta.saldo);

        System.out.println(primeiraConta.saldo);
    }
}
```

[COPIAR CÓDIGO](#)

Quando executarmos o programa teremos um saldo de 300 ou 400? Continuaremos tendo duas referências para apenas um objeto, portanto, o saldo será de 400.



Podemos verificar se `primeiraConta` possui as mesmas informações de `segundaConta`, fazendo um `if`.

```
public class TesteReferencias {
    public static void main(String[] args) {
        Conta primeiraConta = new Conta();
        primeiraConta.saldo = 300;

        System.out.println("saldo da primeira: " + primeiraConta.sa

        Conta segundaConta = primeiraConta;

        System.out.println("saldo da segunda conta: " + segundaConta

        segundaConta.saldo += 100;
        System.out.println("saldo da segunda conta " + segundaConta

        System.out.println(primeiraConta.saldo);

        if(primeiraConta == segundaConta) {
            System.out.println("é a mesma conta");
        }
    }
}
```

```

    }
}

}

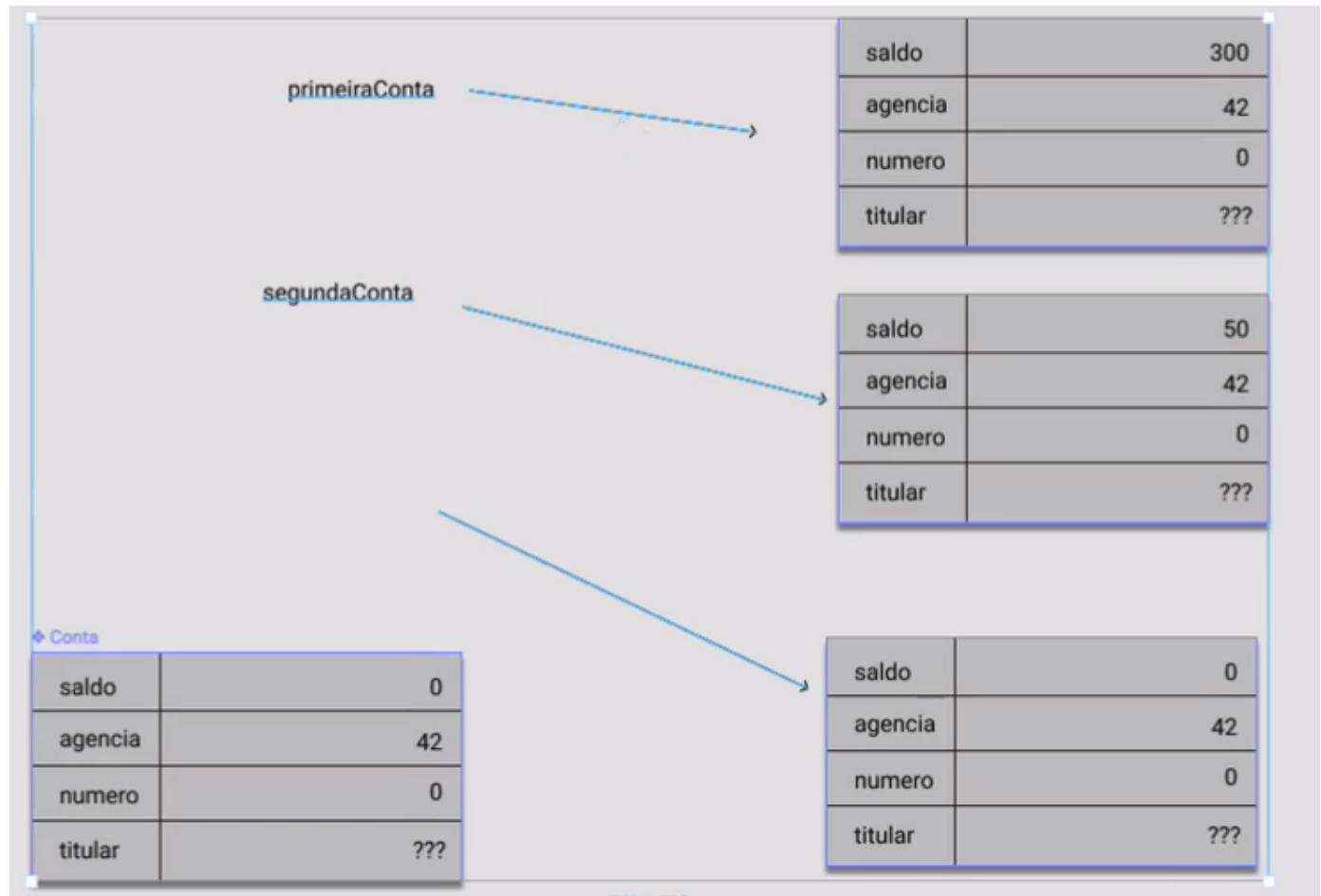
```

COPIAR CÓDIGO



Os números de referência são iguais, portanto, são a mesma conta, fazem referência ao mesmo objeto neste código.

Se formos na classe `CriaConta` que fizemos nas aulas passadas, teremos as variáveis `primeiraConta` e `segundaConta` referenciando objetos diferentes. Há dois `new`s no nosso código, um para cada referência. Podemos perceber a individualidade das referências na representação gráfica do código anterior.



Na classe `CriaConta`, iremos fazer um `if` e um `else` para vermos como o nosso código se comporta.

```
public class CriaConta {  
  
    public static void main(String[] args) {  
        Conta primeiraConta = new Conta();  
        primeiraConta.saldo = 200;  
        System.out.println(primeiraConta.saldo);  
  
        primeiraConta.saldo += 100;  
        System.out.println(primeiraConta.saldo);  
  
        Conta segundaConta = new Conta();  
        segundaConta.saldo = 50;  
  
        System.out.println("primeira conta tem " + primeiraConta.saldo);  
        System.out.println("segunda conta tem " + segundaConta.saldo);  
  
        System.out.println(primeiraConta.agencia);  
        System.out.println(primeiraConta.numero);  
  
        System.out.println(segundaConta.agencia);  
  
        segundaConta.agencia = 146;  
        System.out.println("agora a segunda conta está na agencia ");  
  
        if(primeiraConta == segundaConta) {  
            System.out.println("mesma conta");  
        } else {  
            System.out.println("contas diferentes");  
        }  
    }  
}
```

COPIAR CÓDIGO

O resultado da execução dessa aplicação será contas diferentes . O sinal == irá comparar referências, e não objetos.

De volta à classe TesteReferencias , acionaremos o sysout para primeiraConta .

```
public class TesteReferencias {  
    public static void main(String[] args) {  
        Conta primeiraConta = new Conta();  
        primeiraConta.saldo = 300;  
  
        System.out.println("saldo da primeira: " + primeiraConta.sa...  
  
        Conta segundaConta = primeiraConta;  
  
        System.out.println("saldo da segunda conta: " + segundaConta...  
  
        segundaConta.saldo += 100;  
        System.out.println("saldo da segunda conta " + segundaConta...  
  
        System.out.println(primeiraConta.saldo);  
  
        if(primeiraConta == segundaConta) {  
            System.out.println("é a mesma conta");  
        }  
        System.out.println(primeiraConta);  
    }  
}
```

[COPIAR CÓDIGO](#)

Ao executarmos o programa, veremos que o Java irá imprimir Conta@15db9742 .

Estamos fazendo uma referência a um objeto do tipo Conta , e que este objeto está dentro de uma "gaveta de memória" identificada por essa sequência numérica,

estamos falando da mesma ideia do "Id" que vimos anteriormente com o `3bf4`.

Faremos o mesmo procedimento com `segundaConta`.

```
public class TesteReferencias {  
    public static void main(String[] args) {  
        Conta primeiraConta = new Conta();  
        primeiraConta.saldo = 300;  
  
        System.out.println("saldo da primeira: " + primeiraConta.sa...  
  
        Conta segundaConta = primeiraConta;  
  
        System.out.println("saldo da segunda conta: " + segundaConta...  
  
        segundaConta.saldo += 100;  
        System.out.println("saldo da segunda conta " + segundaConta...  
  
        System.out.println(primeiraConta.saldo);  
  
        if(primeiraConta == segundaConta) {  
            System.out.println("é a mesma conta");  
        }  
  
        System.out.println(primeiraConta);  
        System.out.println(segundaConta);  
    }  
}
```

COPIAR CÓDIGO

O resultado impresso será o mesmo, pois são duas referências apontando para o mesmo objeto.



```
Problems Javadoc Declaration Console
<terminated> TesteReferencias [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_45.jdk/Contents/Hom
400.0
sao a mesmissima conta
Conta@15db9742
Conta@15db9742
```

Faremos o mesmo procedimento na classe `CriaConta` para que vejamos a diferença de comportamento de um código para o outro.

```
public class CriaConta {
    public static void main(String[] args) {
        Conta primeiraConta = new Conta();
        primeiraConta.Saldo = 200;
        System.out.println(primeiraConta.saldo);

        primeiraConta.saldo += 100;
        System.out.println(primeiraConta.saldo);

        Conta segundaConta = new Conta();
        segundaConta.saldo = 50;

        System.out.println("primeira conta tem " + primeiraConta.saldo);
        System.out.println("segunda conta tem " + segundaConta.saldo);

        System.out.println(primeiraConta.agencia);
        System.out.println(primeiraConta.numero);

        System.out.println(segundaConta.agencia);

        segundaConta.agencia = 146;
        System.out.println("agora a segunda conta está na agencia " +
```

```
if(primeiraConta == SegundaConta) {  
    System.out.println("mesma conta");  
} else {  
    System.out.println("contas diferentes");  
}  
  
System.out.println(primeiraConta);  
System.out.println(segundaConta);  
}
```

COPIAR CÓDIGO



Quando iniciamos a aplicação, veremos que o Java imprimiu os valores

Conta@15db972 e Conta@6d06d69c diferentes , afinal, estamos trabalhando com referências orientadas para dois objetos diferentes.

Façamos um teste: colocaremos a mesma quantidade de saldo e a mesmo número em agencia nas duas contas, e verificaremos se a numeração Id continua diferente.

```
public class CriaConta {  
    public static void main(String[] args) {  
        Conta primeiraConta = new Conta();  
        primeiraConta.saldo = 200;  
        System.out.println(primeiraConta.saldo);  
  
        primeiraConta.saldo += 100;  
        System.out.println(primeiraConta.saldo);  
  
        Conta segundaConta = new Conta();  
        segundaConta.saldo = 300;  
  
        System.out.println("primeira conta tem " + primeiraConta.sa
```

```
System.out.println("segunda conta tem " + segundaConta.saldo);

segundaConta.agencia = 146;
System.out.println(primeiraConta.agencia);
System.out.println(primeiraConta.numero);

System.out.println(segundaConta.agencia);

segundaConta.agencia = 146;
System.out.println("agora a segunda conta está na agencia ")

if(primeiraConta == segundaConta) {
    System.out.println("mesma conta");
} else {
    System.out.println("contas diferentes");
}

System.out.println(primeiraConta);
System.out.println(segundaConta);
}
```

COPIAR CÓDIGO

Temos o saldo de 300 para ambas as contas, bem como a agencia de número 146 , e a numeração das contas continua diferente, sendo Conta@15db9742 e Conta@6d06d69c . O Java não irá comparar objetos, e sim referências.

01

Nosso primeiro método

Transcrição

Estamos prontos para evoluir o nosso modelo de classes, bem como os objetos que estamos construindo a partir delas.

Em nossa classe `Conta` nós temos quatro atributos.

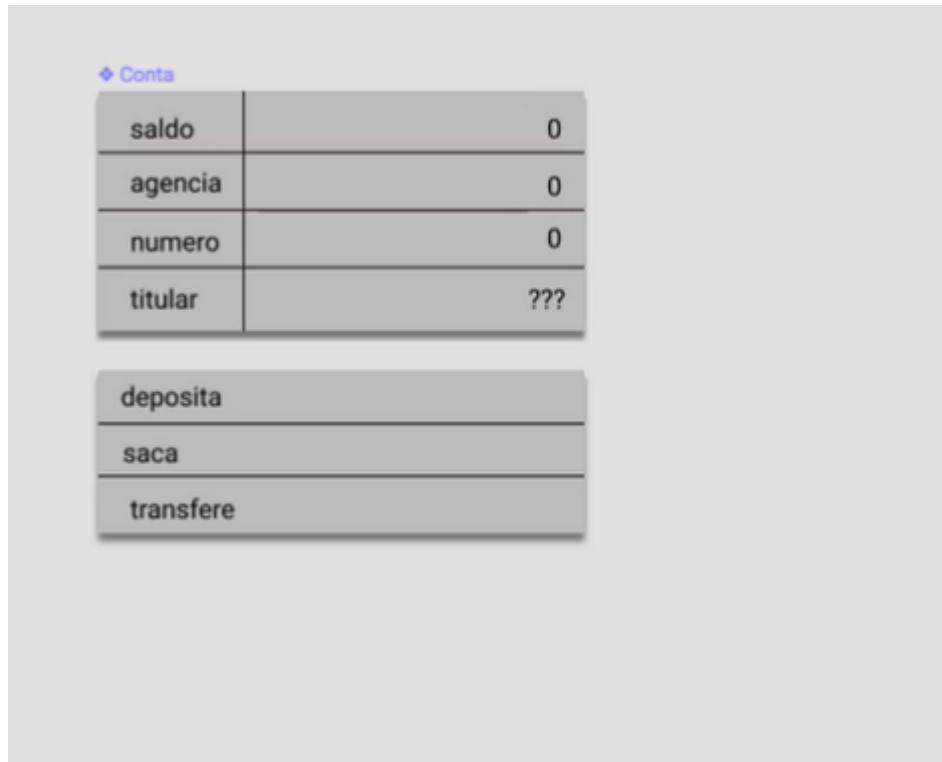
```
public class Conta {  
    double saldo;  
    int agencia;  
    int numero;  
    String titular;  
}
```

[COPIAR CÓDIGO](#)

Sabemos que no momento em que acionamos a palavra-chave `new`, objetos são criados e os atributos referentes a esses objetos são zerados, inclusive os de tipo `String`.

Além de modificarmos os atributos de um determinado objeto, criaremos algumas funcionalidades. No caso de uma conta bancária é estranho que simplesmente surja 300 reais de saldo do nada. O interessante é que haja uma funcionalidade de *depósito* que inclua valores ao atributo `saldo`. Portanto, criaremos certos comportamentos para o objeto "conta bancária".

Podemos ordenar para a nossa conta funções como *sacar*, *depositar* e *transferir*.



Nossa conta, então, possuirá quatro atributos e três funções. Quando formos transferir essas informações para o Java, chamaremos essas funções de **métodos**, ou seja, uma maneira de fazer algo. Veremos posteriormente quais são as diferenças entre o termo "função" utilizado em outras linguagens e "método" no Java.

Escreveremos o método `deposita()` em nossa classe `Conta`. Atenção para a sintaxe utilizada: no parênteses `()` adicionaremos o que está sendo recebido pelo método, ou seja, um **parâmetro**. No caso de uma conta bancária, precisamos adicionar um valor a ser depositado. Em Java não se pode declarar uma variável sem especificar seu tipo, portanto, especificaremos a variável `valor` como sendo do tipo `double`.

```
public class Conta {
    double saldo;
    int agencia;
    int numero;
    String titular;
```

```
    deposita(double valor)  
}
```

[COPIAR CÓDIGO](#)

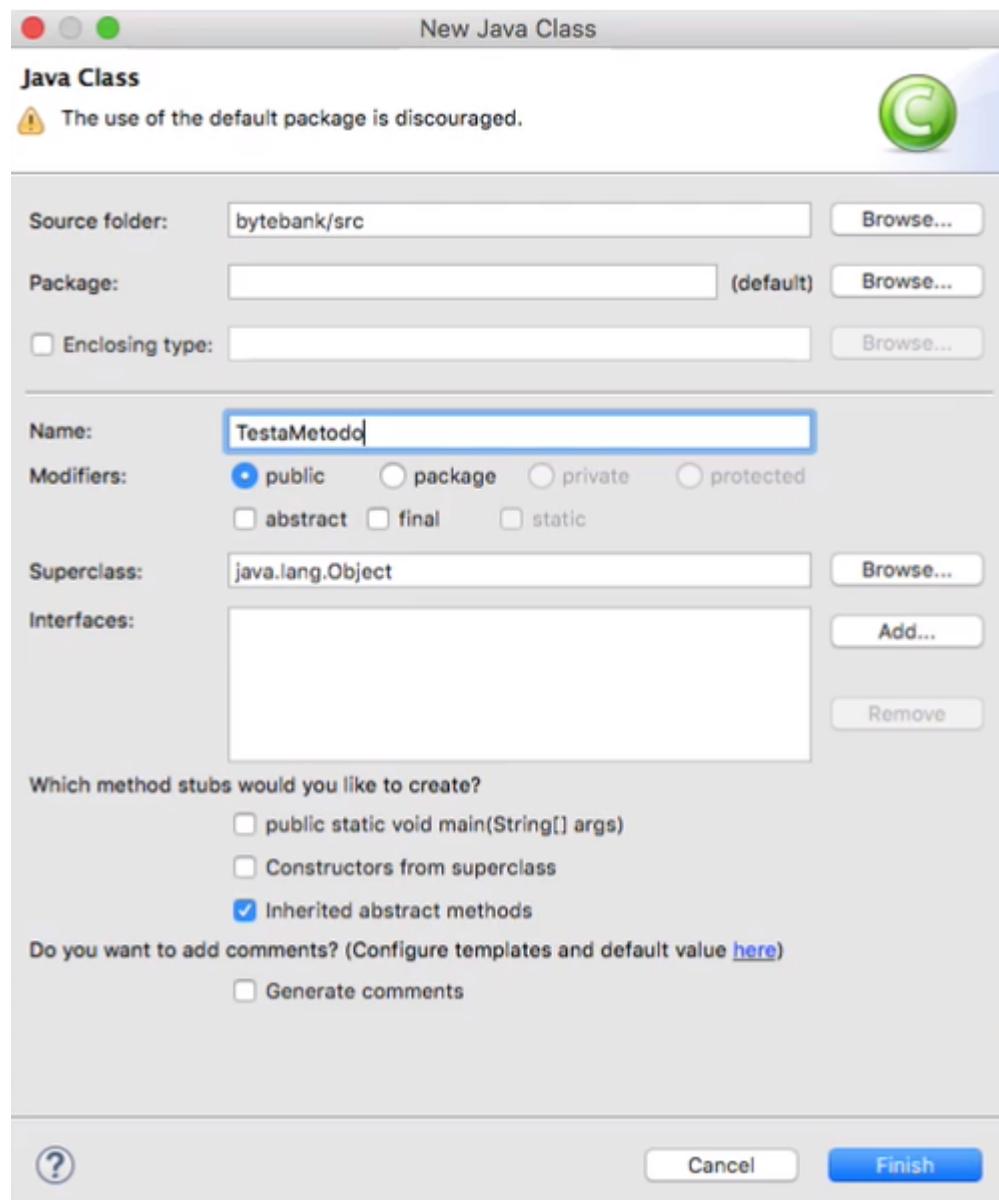
Depois que depositamos um valor em uma determinada conta bancária, poderemos receber uma mensagem, um número, uma espécie de comprovante ou algo do gênero. No caso do nosso projeto **ByteBank**, não há qualquer tipo de retorno à ação de depósito. Quando não existe qualquer tipo de retorno ao acionarmos um método, utilizamos a palavra-chave `void`. Feito isso, fecharemos o bloco utilizando as chaves `{}`

```
public class Conta {  
    double saldo;  
    int agencia;  
    int numero;  
    String titular;  
  
    void deposita(double valor) {  
  
    }  
}
```

[COPIAR CÓDIGO](#)

O método `deposita()` não produz nenhum resultado por enquanto, mas sua sintaxe é válida.

Agora que já conhecemos a sintaxe, aprenderemos como acionar e utilizar os métodos em Java. Criaremos uma nova classe intitulada `TestaMetodo`



Para invocarmos o método `deposita()`, é necessário nos referenciar à uma conta específica, neste caso, usaremos uma variável chamada de `contaDoPaulo`.

Lembrem-se: é comum o nome de uma variável ser igual ao da classe, sendo que a variável por convenção é escrita com letra minúscula.

```
public class TestaMetodo {  
    public static void main(String[] args) {  
        Conta contaDoPaulo = new Conta();
```

```
}
```

[COPIAR CÓDIGO](#)

O saldo de contaDoPaulo terá um valor de 100 . Para invocarmos o método deposita() utilizaremos o caractere ponto . seguindo dos parênteses que contém o valor que queremos depositar, que no caso será 50 .

```
public class TestaMetodo {  
    public static void main(String[] args) {  
        Conta contaDoPaulo = new Conta();  
        contaDoPaulo.saldo = 100;  
        contaDoPaulo.deposita(50);  
  
    }  
}
```

[COPIAR CÓDIGO](#)

Nosso método é válido, e mesmo não gerando nenhum resultado, poderá ser executado sem nenhum erro. Podemos realizar um teste adicionando o Sysout a nossa classe TestaMetodo :

```
public class TestaMetodo {  
    public static void main(String[] args) {  
        Conta contaDoPaulo = new Conta();  
        contaDoPaulo.saldo = 100;  
        contaDoPaulo.deposita(50);  
        System.out.println(contaDoPaulo.saldo);  
    }  
}
```

[COPIAR CÓDIGO](#)

Ao executarmos a aplicação veremos que o resultado impresso será 100 , ou seja, não foi depositado nenhum novo valor à `contaDoPaulo` . Nada ocorreu, pois ao evocarmos o método `deposita` estamos nos referindo ao código escrito na classe `Conta` , e este código não executa nada, como já sabemos.

```
public class Conta {  
    double saldo;  
    int agencia;  
    int numero;  
    String titular;  
  
    void deposita(double valor) {  
  
    }  
}
```

[COPIAR CÓDIGO](#)

Queremos que o método `deposita()` adicione valores à uma determinada conta. Existem muitas maneiras de escrevermos esse código, mas faremos da seguinte forma:

Adicionamos o `public` ao método `deposita()` , não se atente para isso neste momento do curso, discutiremos essa questão posteriormente

```
public class Conta {  
    double saldo;  
    int agencia;  
    int numero;  
    String titular;  
  
    public void deposita(double valor) {
```

```
    saldo = saldo + valor;  
}  
}
```

[COPIAR CÓDIGO](#)

Reparem que há uma diferença de cores entre `saldo` e `valor`, pois o primeiro é um atributo do objeto `Conta`, enquanto o segundo é uma variável.

Feitas estas alterações, retornaremos à classe `TestaMetodo` e executaremos a aplicação.

```
public class TestaMetodo {  
    public static void main(String[] args) {  
        Conta contaDoPaulo = new Conta();  
        contaDoPaulo.saldo = 100;  
        contaDoPaulo.deposita(50);  
        System.out.println(contaDoPaulo.saldo);  
    }  
}
```

[COPIAR CÓDIGO](#)

Teremos, dessa vez, um valor impresso de `150`. O método `deposita()` alterou o valor do atributo `saldo`, que é uma característica de conta.

Para deixarmos mais clara essa noção, usaremos outra palavra-chave do Java que é opcional neste caso específico, o `this`. Ao observarmos classe `Conta`, veremos que no bloco do método `deposita()` há uma referência de `saldo`, mas qual saldo? De que conta estamos falando?

```
public class Conta {  
    double saldo;
```

```
int agencia;
int numero;
String titular;

public void deposita(double valor) {
    saldo = saldo + valor;
}

}
```

[COPIAR CÓDIGO](#)

Podemos escrever nosso código da seguinte forma:

```
public class Conta {
    double saldo;
    int agencia;
    int numero;
    String titular;

    public void deposita(double valor) {
        contaDoPaulo = saldo + valor;
    }
}
```

[COPIAR CÓDIGO](#)

Não conseguiremos compilar nosso código desta forma, pois `contaDoPaulo` não é uma variável presente neste escopo. Queremos que `saldo` seja relacionado à conta que está evocando o método `deposita()`, para isso, faremos uso da palavra-chave `this`.

```
public class Conta {
    double saldo;
```

```
int agencia;  
int numero;  
String titular;  
  
public void deposita(double valor) {  
    this.saldo = this.saldo + valor;  
}  
}
```

[COPIAR CÓDIGO](#)

Como o método está sendo invocado pela `contaDoPaulo`, o `saldo` é referente a esta conta. Não incluímos a palavra-chave `this` junto à variável `valor`, pois ela **não** é um atributo de um objeto.

Criamos, assim, nosso primeiro método. Escrevemos `deposita()`, parâmetros e qual será a devolução gerada pelo método.

05

Métodos com retorno

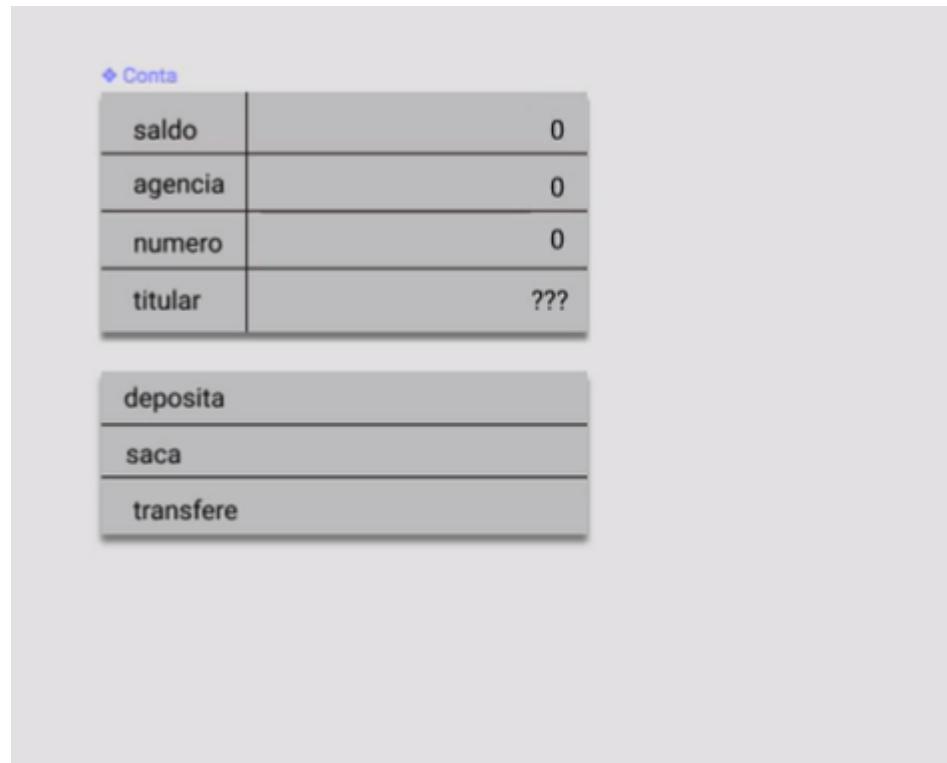
Transcrição

Trabalharemos em um método mais elaborado. O `deposita()` que criamos, não devolvia nenhum tipo de informação para quem o invocou.

```
public class TestaMetodo {  
    public static void main (String[] args) {  
        Conta contaDoPaulo = new Conta();  
        contaDoPaulo.saldo = 100;  
        contaDoPaulo.deposita(50);  
        System.out.println(contaDoPaulo.saldo);  
    }  
}
```

[COPIAR CÓDIGO](#)

Ao observarmos o diagrama, notamos que existe um método chamando `saca()`, que tem por função *retirar dinheiro de uma conta específica*.



Criaremos o novo método `saca()` na classe `Conta`. Para realizarmos um saque, é preciso atribuir um `valor`, - uma variável do tipo `double` - ao saque.

```
public class Conta {  
    double saldo;  
    int agencia;  
    int numero;  
    String titular;  
  
    public void deposita(double valor) {  
        this.saldo = this.saldo + valor;  
    }  
  
    saca(double valor);  
}
```

COPIAR CÓDIGO

Não precisamos ter a informação de qual conta iremos sacar neste ponto, e essa é uma diferença da forma de programar orientada a métodos e a orientada a funções.

Em métodos sempre há um "sujeito" do código à esquerda (no exemplo, `contaDoPaulo`), de forma que sabemos o direcionamento de determinado comando.

```
contaDoPaulo.saldo = 100
```

[COPIAR CÓDIGO](#)

Iremos fazer com o que o método `saca()` nos retorne um `boolean : true` caso o saque seja efetivado, `false` caso não.

Para isso, escreveremos o `public` e `boolean` na linha do método e fechamos as chaves `{}`. Reparem que o método `deposita()` está em um bloco e método `saca()` em outro, não existe método dentro de método.

```
public class Conta {  
    double saldo;  
    int agencia;  
    int numero;  
    String titular;  
  
    public void deposita(double valor) {  
        this.saldo = this.saldo + valor;  
    }  
  
    public boolean saca(double valor) {  
    }  
}
```

[COPIAR CÓDIGO](#)

Da forma como escrevemos o código ele não será compilado, pois acionamos um `boolean` sem escrever qual seria o retorno dado pelo método. Para prosseguirmos com o nosso código, acionaremos a palavra-chave `if`. Usaremos, também, o `this`,

que aciona a referência para a conta que está acionando o método. Se o saldo de this for \geq ao valor de saque, o novo saldo será o saldo de this - valor de saque. O método, então, retornará o valor true . Caso o contrário, (else) será retornado o valor false .

```
public class Conta {  
    double saldo;  
    int agencia;  
    int numero;  
    String titular;  
  
    public void deposita(double valor) {  
        this.saldo = this.saldo + valor;  
    }  
  
    public boolean saca(double valor) {  
        if(this.saldo >= valor) {  
            this.saldo = this.saldo - valor;  
            return true;  
        } else {  
            return false;  
        }  
    }  
}
```

[COPIAR CÓDIGO](#)

Iremos na classe TestaMetodo para analisarmos se o nosso código está funcional.

Iremos sacar 20 reais de contaDoPaulo :

```
public class TestaMetodo {  
    public static void main (String[] args) {  
        Conta contaDoPaulo = new Conta();  
        contaDoPaulo.saldo = 100;
```

```

contaDoPaulo.deposita(50);
System.out.println(contaDoPaulo.saldo);
contaDoPaulo.saca(20);
System.out.println(contaDoPaulo.saldo);
}
}

```

[COPIAR CÓDIGO](#)

Ao rodarmos a aplicação, veremos que o resultado será 130 . Ou seja, o nosso comando de saque foi efetivo. A respeito do true e false do boolean , quando passamos o mouse sobre o código vemos o retorno do método, podemos ou não utilizar esse valor.

```

1
2 public class TestaMetodo {
3
4     public static void main(String[] args) {
5         Conta contaDoPaulo = new Conta();
6         contaDoPaulo.saldo = 100;
7         contaDoPaulo.deposita(50);
8         System.out.println(contaDoPaulo.saldo);
9         contaDoPaulo.saca(20);
10        System.out.pr

```

Podemos guardar esse valor dentro de uma variável que chamaremos de conseguiRetirar , e acionaremos o sysout .

```

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

```

```
Conta contaDoPaulo = new Conta();
contaDoPaulo.saldo = 100;
contaDoPaulo.deposita(50);
System.out.println(contaDoPaulo.saldo);

boolean conseguiuRetirar = contaDoPaulo.saca(20);
System.out.println(contaDoPaulo.saldo);
System.out.println(conseguiuRetirar);

}

}
```

COPIAR CÓDIGO

Ao executarmos o programa, veremos que o resultado será `true`.

Abordaremos algumas melhorias de navegação que podemos executar no Eclipse. Normalmente, deixamos muitas abas abertas no editor, uma forma mais rápida e fácil de transitar no projeto que está sendo desenvolvido é manter pressionado atalho "Ctrl", com isso, o Eclipse irá transformar muitos elementos em links.

Ao clicarmos em um dos links, seremos transportados para o código referente ao método ou classe que estamos acionando.

```
1
2 public class TestaMetodo {
3
4     public static void main(String[] args) {
5         Conta contaDoPaulo = new Conta();
6         contaDoPaulo.saldo = 100;
7         contaDoPaulo.deposita(50);
8         System.out.println(contaDoPaulo.saldo);
9
10        boolean conseguiuRetirar = contaDoPaulo.saca(20);
11        System.out.println(contaDoPaulo.saldo);
12        System.out.println(conseguiuRetirar);
13    }
}
```

Outra melhoria que podemos fazer quanto a sua formatação, é alterar a linha

`this.saldo = this.saldo + valor` utilizando `+=`, e na linha `this.saldo = this.saldo - valor` fazer uso de `-=`.

Essas alterações deixarão o código mais enxuto, e além de ser mais comum na comunidade de programadores Java este tipo de formato, o resultado final será o mesmo.

```
public void deposita(double valor) {  
    this.saldo += valor;  
}  
  
public boolean saca(double valor) {  
    if(this.saldo >= valor) {  
        this.saldo -= valor;  
        return true;  
    } else {  
        return false;  
    }  
}
```

COPIAR CÓDIGO

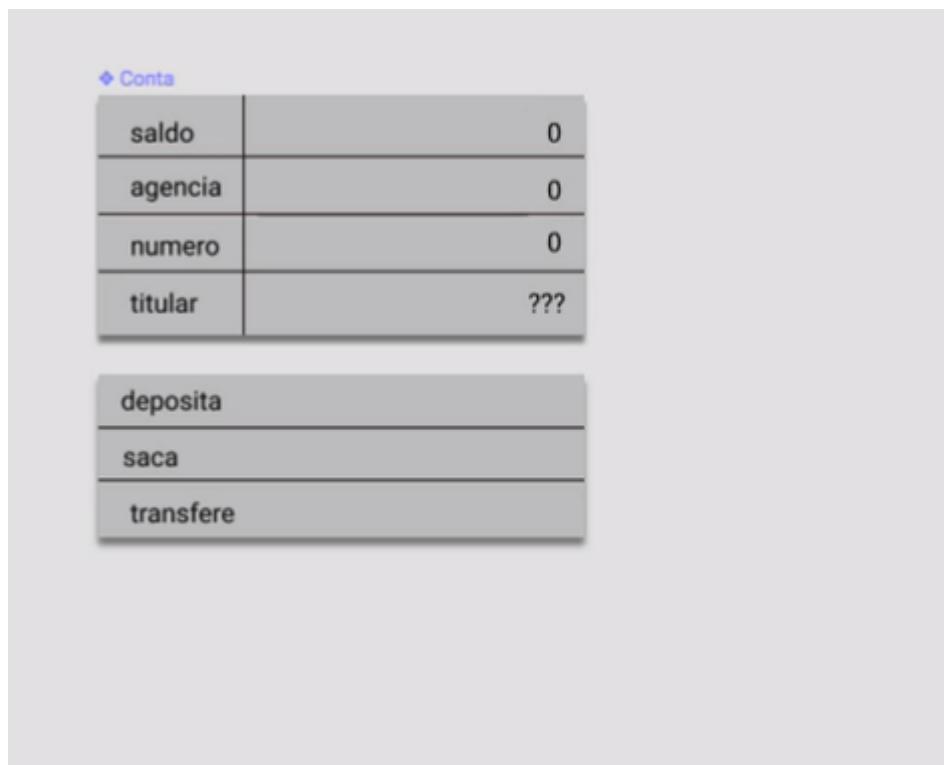
08

Métodos com referência e mais retorno

Transcrição

Tenho um desafio para que você saiba se entendeu o assunto **métodos**.

De acordo com o diagrama de contas, chegamos ao nosso terceiro método: o `transfere()`, que tem como finalidade **transferir dinheiro** de uma conta para outra.



Vamos incluir o novo método na classe `Conta`.

Abaixo do método `saca()`, adicionaremos `transfere()`.

Pensaremos sobre um ponto: quando transferimos dinheiro de uma conta para outra, receberemos uma informação de volta? Neste caso, sim. Para que uma transferência seja realizada com sucesso, é necessário haver dinheiro o suficiente em uma conta. Caso contrário, o retorno será `false`.

Usaremos o tipo `boolean`, com uma finalidade parecida com a do método `saca()`.

```
public boolean saca(double valor){  
    if(this.saldo >= valor) {  
        this.saldo -= valor;  
        return true;  
    } else {  
        return false  
    }  
}  
  
public boolean transfere() {  
}
```

[COPIAR CÓDIGO](#)

E quais argumentos o método recém-criado receberá? Nós, certamente, transferiremos um valor, expressaremos isso adicionando a variável `valor` que será do tipo `double`.

```
public boolean transfere(double valor, Conta destino) {  
}
```

[COPIAR CÓDIGO](#)

Observe que adicionamos um segundo argumento, que foi separado por uma vírgula (,) do primeiro. Foi necessário incluir um segundo argumento no método referente

a **conta de destino** do depósito, ou seja, a conta para onde o dinheiro será transferido.

No entanto, será que é preciso especificar a conta de onde o dinheiro será retirado? Quantos argumentos serão necessários?

Nós adicionamos a `Conta destino`. Pela primeira vez, recebemos uma variável que usa também letras maiúsculas. Falta nela a tipagem de `double` ou `int` como fizemos anteriormente. Agora, estamos fazendo uma referência para outra conta.

Será que devemos receber a `Conta origem`? Não. E por quê?

Antes, você vai perceber que a linha com `transfere()` está vermelha. Isto ocorre, porque nosso código não está sendo compilado, afinal, falta adicionar o retorno (`return`).

Em seguida, acessaremos `TestaMetodo.java`. Abaixo do último `println` de `conseguiRetirar`, criaremos uma referência a uma outra conta, que chamaremos de `contaDaMarcela`.

Será nesta conta que depositaremos 1000 reais. Nossa objetivo será transferir 300 reais de `contaDaMarcela` para `contaDoPaulo`.

O interessante da orientação a objetos é que a nossa primeira frase será uma *referência* para a variável.

```
public class TestaMetodo {  
    public static void main(String[] args) {  
        Conta contaDoPaulo = new Conta();  
        contaDoPaulo.saldo = 100;  
        contaDoPaulo.deposita(50);  
        System.out.println(contaDoPaulo.saldo);  
    }  
}
```

```
boolean conseguiRetirar = contaDoPaulo.saca(20);
System.out.println(contaDoPaulo.saldo);
System.out.println(conseguiRetirar);

Conta contaDaMarcela = new Conta();
contaDaMarcela.deposita(1000);

}

}
```

COPIAR CÓDIGO

Primeiramente, escreveremos `contaDaMarcela` e depois, passaremos o método `transfere()`.

Enquanto digitamos o código, o *autocomplete* do Eclipse vai disponibilizar diversos métodos, incluindo `transfere()`, que ainda tem um erro no outro arquivo.

No entanto, se tentássemos já executar o código, receberíamos um aviso de possível problema na compilação.

Como parâmetro, passaremos o valor de `300`, que será transferido, e como parâmetro de destino, incluiremos uma referência de `contaDoPaulo`. Foi desnecessário colocar, por exemplo, o `id` da conta do Paulo. O método vai receber um variável do tipo `conta`, que não é um objeto `Conta`.

Você pode ter ficado com a impressão de que estamos passando uma conta dentro de um método. Este não é o caso. Nós mandamos um "número interno" que o Java enxerga e não precisa tanto da nossa atenção.

```
public class TestaMetodo {
    public static void main(String[] args) {
        Conta contaDoPaulo = new Conta();
        contaDoPaulo.saldo = 100;
```

```
contaDoPaulo.deposita(50);
System.out.println(contaDoPaulo.saldo);

boolean conseguiuRetirar = contaDoPaulo.saca(20);
System.out.println(contaDoPaulo.saldo);
System.out.println(conseguiuRetirar);

Conta contaDaMarcela = new Conta();
contaDaMarcela.deposita(1000);

contaDaMarcela.transfere(300, contaDoPaulo);
}

}
```

COPIAR CÓDIGO

Evitamos adicionar `ContaDaMarcela`, porque a conta de origem é o valor no primeiro parâmetro. Ou seja, vamos passar apenas a conta de destino, considerando que a origem já é o objeto ao qual invocamos o método.

Seria equivalente a referência `this` do método `transfere()` no outro lado. Agora, só falta implementá-lo na classe `Conta`.

Para isto, usaremos `if`: caso o `saldo` de `this` - fazendo referência a `contaDaMarcela` - seja igual ou superior ao `valor` de transferência, será subtraído de `saldo` o `valor` referente à transferência.

```
public class Conta {

    // atributos
    // métodos

    public boolean transfere(double valor, Conta destino) {
        if(this.saldo >= valor) {
```

```
this.saldo -= valor;  
}  
  
}  
}
```

[COPIAR CÓDIGO](#)

Por enquanto, apenas retiramos um `valor` de `saldo`, precisaremos que este valor seja transferido para uma conta destino, no caso, `contaDoPaulo`. Existem duas formas de executar essa função, uma delas é esta:

```
destino.saldo += valor;
```

[COPIAR CÓDIGO](#)

Ou, podemos reutilizar um método da classe `Conta`, o `deposita()`.

Retiramos um `valor` e ele foi depositado. Neste caso, a transferência foi realizada com sucesso, e o retorno será `true`. Caso não haja dinheiro o suficiente para realizar a transferência, será retornado `false`.

A utilização do `else` é opcional no código, o retorno `false` ocorrerá mesmo que esta palavra não tenha sido utilizada.

```
public class Conta {  
  
    // atributos  
    // métodos  
  
    public boolean transfere(double valor, Conta destino) {  
        if(this.saldo >= valor) {  
            this.saldo -= valor;  
        }  
        return this.saldo >= valor;  
    }  
}
```

```
        destino.deposita(valor);
        return true;
    } else {
        return false;
}
}
```

[COPIAR CÓDIGO](#)

Em `TestaMetodo`, acionaremos o `sysout` para `contaDaMarcela`.

```
public class TestaMetodo {
    public static void main(String[] args) {
        Conta contaDoPaulo = new Conta();
        contaDoPaulo.saldo = 100;
        contaDoPaulo.deposita(50);
        System.out.println(contaDoPaulo.saldo);

        boolean conseguiuRetirar = contaDoPaulo.saca(20);
        System.out.println(contaDoPaulo.saldo);
        System.out.println(conseguiuRetirar);

        Conta contaDaMarcela = new Conta();
        contaDaMarcela.deposita(1000);

        contaDaMarcela.transfere(300, contaDoPaulo);
        System.out.println(contaDaMarcela.saldo);
    }
}
```

[COPIAR CÓDIGO](#)

Ao executarmos o programa veremos que o valor impresso será 700, ou seja, nosso código funcionou perfeitamente.

Podemos, inclusive, verificar se no saldo de contaDoPaulo há 300 reais a mais, como o esperado. Para isso. acionaremos o sysout novamente para contaDoPaulo .

```
contaDaMarcela.transfere(300, contaDoPaulo);
System.out.println(contaDaMarcela.saldo);
System.out.println(contaDoPaulo.saldo);
```

[COPIAR CÓDIGO](#)

Veremos que o valor impresso ao final da execução será de 430 reais, comprovando o sucesso do método. Não utilizamos o valor devolvido pelo boolean, poderíamos ter guardado esse valor através da utilização de if. Estamos invocando o método transfere() - lembre-se que dentro de if só são possíveis expressões booleanas - que devolve boolean, e portanto, o if pode ser compilado.

Caso o resultado dessa transferência tenha dado true , iremos imprimir "transferência com sucesso". Caso contrário, imprimiremos "faltou dinheiro".

```
public class TestaMetodo {
    public static void main(String[] args) {
        Conta contaDoPaulo = new Conta();
        contaDoPaulo.saldo = 100;
        contaDoPaulo.deposita(50);
        System.out.println(contaDoPaulo.saldo);

        boolean conseguiuRetirar = contaDoPaulo.saca(20);
        System.out.println(contaDoPaulo.saldo);
        System.out.println(conseguiuRetirar);

        Conta contaDaMarcela = new Conta();
        contaDaMarcela.deposita(1000);

        if(contaDaMarcela.transfere(300, contaDoPaulo)) {
```

```
        System.out.println("transferencia com sucesso");
    } else {
        System.out.println("faltou dinheiro");
    }

    System.out.println(contaDaMarcela.saldo);
    System.out.println(contaDoPaulo);
}
```

COPIAR CÓDIGO

Caso a linha que inserimos o `if` fique muito grande ou complicada, é normal quebrarmos em duas linhas. Poderíamos alocar a parte do código referente ao método `transfere()` e inserir uma variável `sucessoTransferencia` em seu lugar.

```
public class TestaMetodo {
    public static void main(String[] args) {
        Conta contaDoPaulo = new Conta();
        contaDoPaulo.saldo = 100;
        contaDoPaulo.deposita(50);
        System.out.println(contaDoPaulo.saldo);

        boolean conseguiRetirar = contaDoPaulo.saca(20);
        System.out.println(contaDoPaulo.saldo);
        System.out.println(conseguiRetirar);

        Conta contaDaMarcela = new Conta();
        contaDaMarcela.deposita(1000);

        boolean sucessoTransferencia = contaDaMarcela.transfere(300
            if(sucessoTransferencia) {
                System.out.println("transferencia com sucesso");
            } else {
                System.out.println("faltou dinheiro");
```

```
    }
    System.out.println(contaDaMarcela.saldo);
    System.out.println(contaDoPaulo.saldo);
}
}
```

COPIAR CÓDIGO



01

Composição de Objetos

Transcrição

Demos pouca atenção para o atributo `titular` do tipo `String`. Podemos utilizar esse atributo dentro da classe `TestaMetodo` sem dificuldades.

```
public class TestaMetodo {  
  
    System.out.println(contaDaMarcela.saldo);  
    System.out.println(contaDoPaulo.saldo);  
  
    contaDoPaulo.titular = "paulo silveira";  
    System.out.println(contaDoPaulo.titular);  
}
```

[COPIAR CÓDIGO](#)

Foi dito que Java zera o valor dos atributos quando acionamos a palavra-chave `new`. Agora compreenderemos melhor o que acontece no caso dos tipos não numéricos como `String`.

Suponhamos que a conta bancária do `ByteBank` além de guardar as informações de saldo, agência, número e o nome do titular, também guardará o *CPF* do titular e sua *profissão*. Uma possível solução seria incluir esses novos atributos à classe `Conta`.

```
public class Conta {  
    double saldo;
```

```
int agencia;  
int numero;  
String titular;  
String cpf;  
String profissao;
```

<!-- ... -->

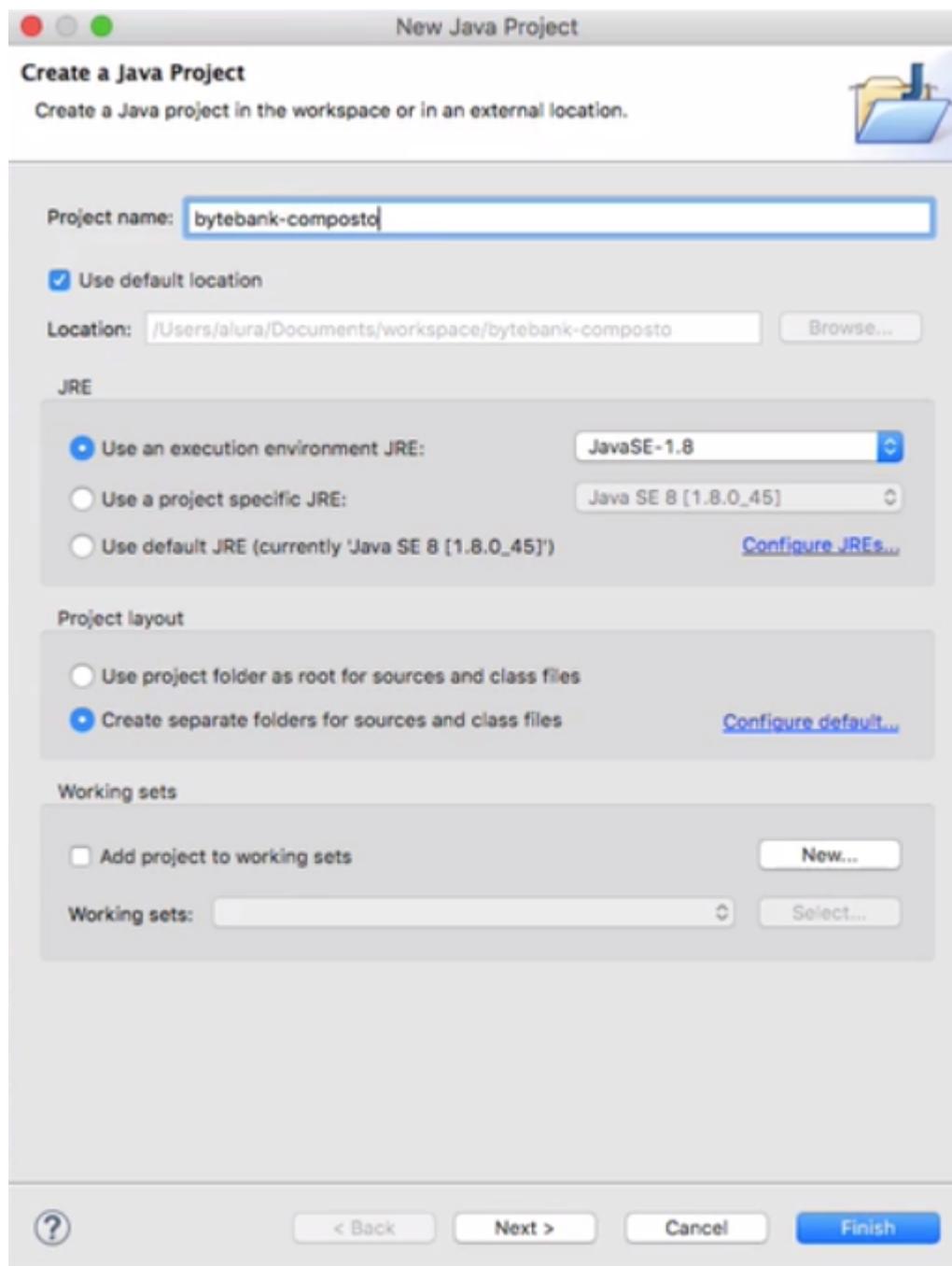
COPiar CÓDIGO

Percebam que a classe começa a ficar "inchada" e com muitas informações que não dizem respeito exatamente a uma conta bancária, como a profissão do titular e seu CPF.

Para resolver essa questão, podemos criar um tipo novo chamado `Cliente`, que terá os atributos de `nome`, `cpf` e `profissão`.

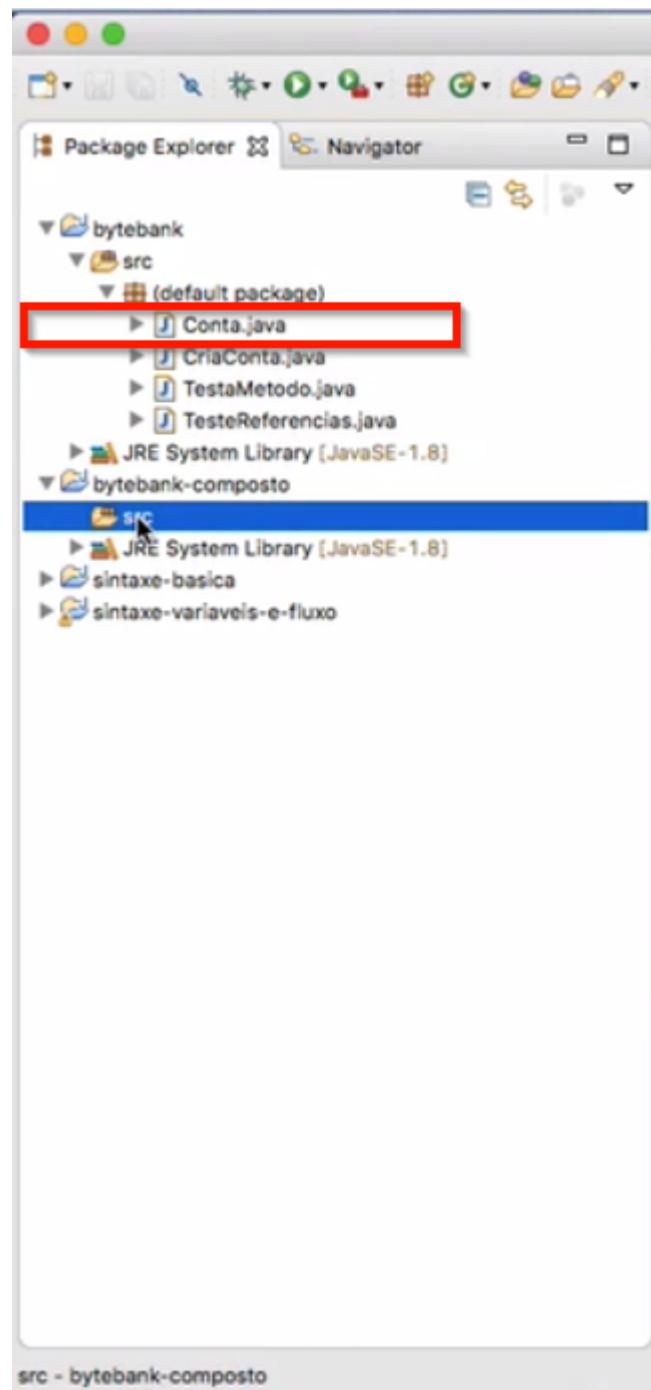


Para separar bem a organização das contas bancárias e dos titulares, criaremos um novo projeto Java intitulado `bytebank-composto`.



Na área do *Package Explorer*, selecionaremos a classe `Conta.java` e a copiaremos utilizando o atalho "Ctrl + C", e a colaremos na pasta `src` via atalho "Ctrl + V".

Atenção! Quando temos duas classes com o mesmo nome, ainda que em projetos diferentes, podemos nos confundir e fazer edições na classe errada. O Eclipse disponibiliza um recurso para que se feche o projeto que não está sendo trabalhado. Na área *Package Navigator*, basta clicar com o botão direito sobre o nome do projeto e selecionar a opção "Close Project".



No novo projeto, criaremos uma nova classe intitulada `Cliente`. Tal classe conterá os atributos de `nome`, `cpf` e `profissao` como já foi dito.

```
public class Cliente {  
    String nome;  
    String cpf;  
    String profissao;  
}
```

[COPIAR CÓDIGO](#)

Iremos estabelecer uma relação entre `Conta` e `Cliente`, ou seja, toda `Conta` faz uma referência a um `Cliente`.

Não é mais interessante para o nosso projeto que o atributo `titular` seja uma `String`, e sim que faça referência a um cliente específico.

Criaremos uma nova classe chamada `TestaBanco`. Faremos um `main` e criaremos uma referência para um cliente que chamaremos de `paulo`.

```
public class TestaBanco {  
    public static void main(String[] args) {  
        Cliente paulo = new Cliente();  
    }  
}
```

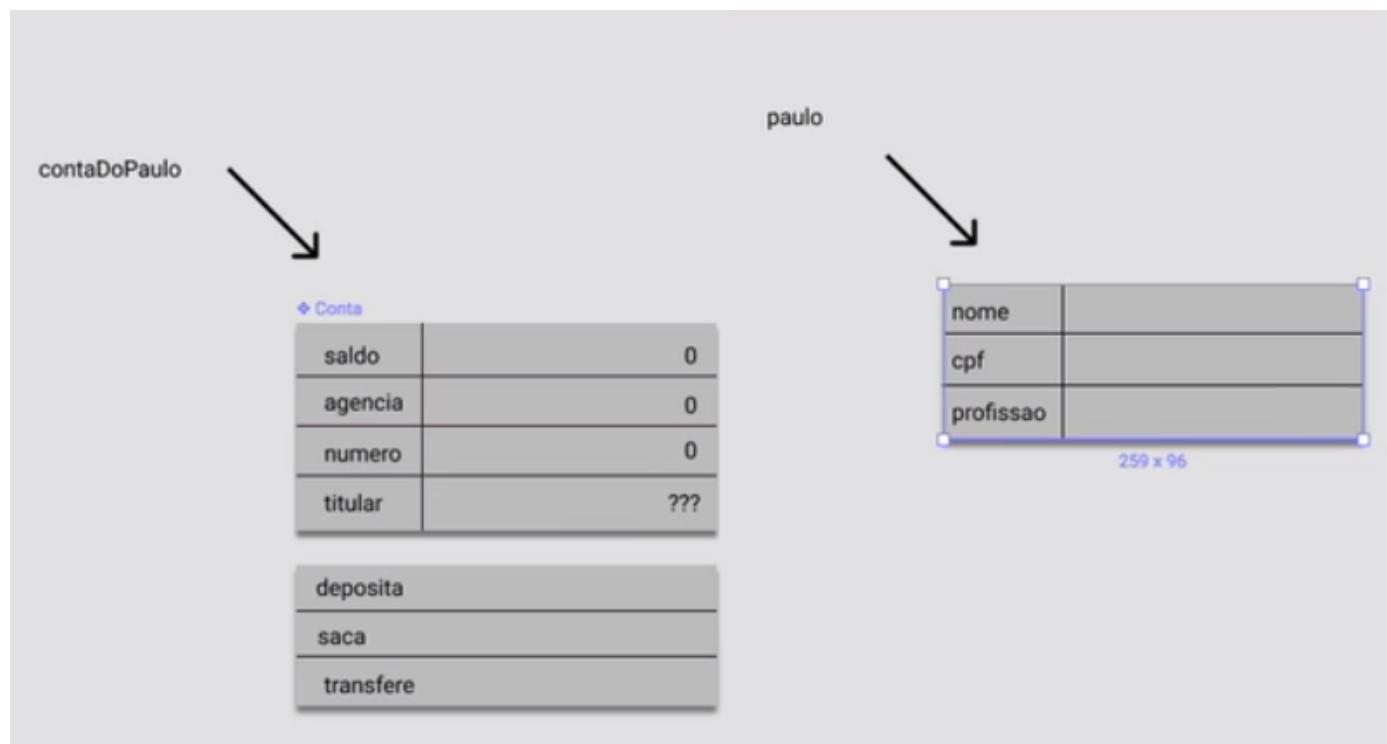
[COPIAR CÓDIGO](#)

Vamos popular este objeto, criando seus atributos.

```
public class TestaBanco {  
    public static void main(String[] args) {  
        Cliente paulo = new Cliente();  
        paulo.nome = "Paulo Silveira";  
        paulo.cpf = "222.222.222-22";  
        paulo.profissao = "programador";  
    }  
}
```

[COPIAR CÓDIGO](#)

A referência para este cliente está populada com os dados estipulados.



Criaremos a conta do cliente referido, e depositaremos um valor de 100 reais.

```
public class TestaBanco {
    public static void main(String[] args) {
        Cliente paulo = new Cliente();
        paulo.nome = "Paulo Silveira";
        paulo.cpf = "222.222.222-22";
        paulo.profissao = "programador";

        Conta contaDoPaulo = new Conta();
        contaDoPaulo.deposita(100);
    }
}
```

COPIAR CÓDIGO

Agora temos uma classe **Conta** e outra **Cliente**. Queremos que o atributo **titular** não seja uma **String**, mas sim, uma referência para um objeto do tipo **Cliente**.

nossa classe `Conta`, alteraremos o tipo do atributo `titular` para ser do tipo

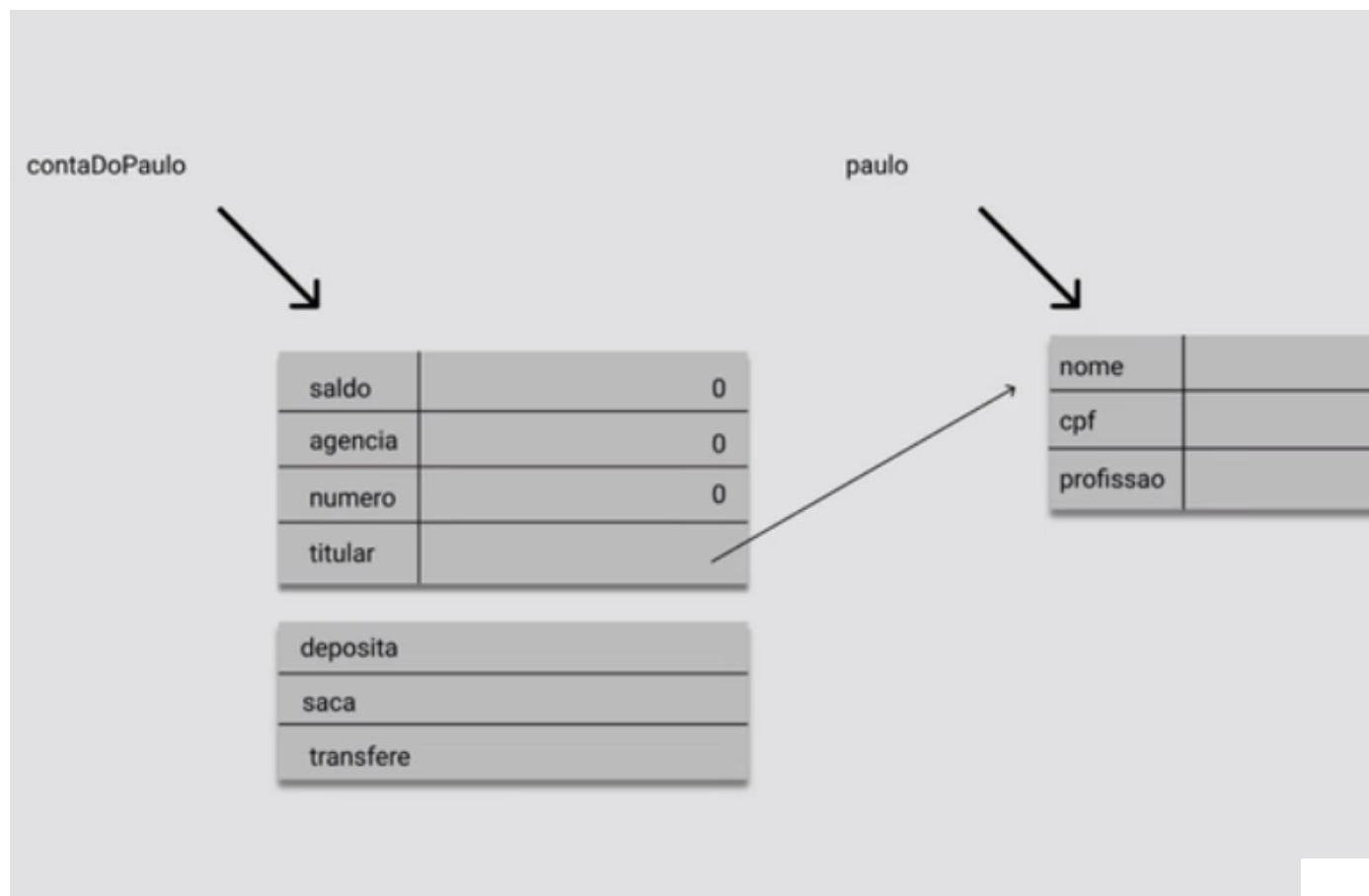
`Cliente`.

```
public class Conta {  
    double saldo;  
    int agencia;  
    int numero;  
    Cliente titular;
```

<!-- ... -->

COPIAR CÓDIGO

A nossa ideia pode ser ilustrada pelo diagrama. Queremos que o atributo `titular` faça uma referência a um cliente específico, ou seja, iremos fazer uma associação entre objetos.



Faremos essa associação na classe `TestaBanco`, montando, assim, a nossa composição de objetos.

```
public class TestaBanco {  
    public static void main(String[] args) {  
        Cliente paulo = new Cliente();  
        paulo.nome = "Paulo Silveira";  
        paulo.cpf = "222.222.222-22";  
        paulo.profissao = "programador";  
  
        Conta contaDoPaulo = new Conta();  
        contaDoPaulo.deposita(100);  
  
        contaDoPaulo.titular = paulo;  
        System.out.println(contaDoPaulo.titular.nome);  
    }  
}
```

[COPIAR CÓDIGO](#)

Ao executarmos a aplicação, veremos que será impresso o resultado `Paulo Silveira`.

Para testarmos o comportamento no programa, tentaremos imprimir apenas o `titular`.

```
public class TestaBanco {  
    //...  
    contaDoPaulo.titular = paulo;  
    System.out.println(contaDoPaulo.titular.nome);  
    System.out.println(contaDoPaulo.titular);  
}  
}
```

[COPIAR CÓDIGO](#)

Ao executarmos a aplicação veremos que o resultado será uma espécie de Id (Cliente@15db9742), que possui o mesmo valor que a variável paulo , afinal, trata-se da referência para um mesmo objeto.

03

Referência Null

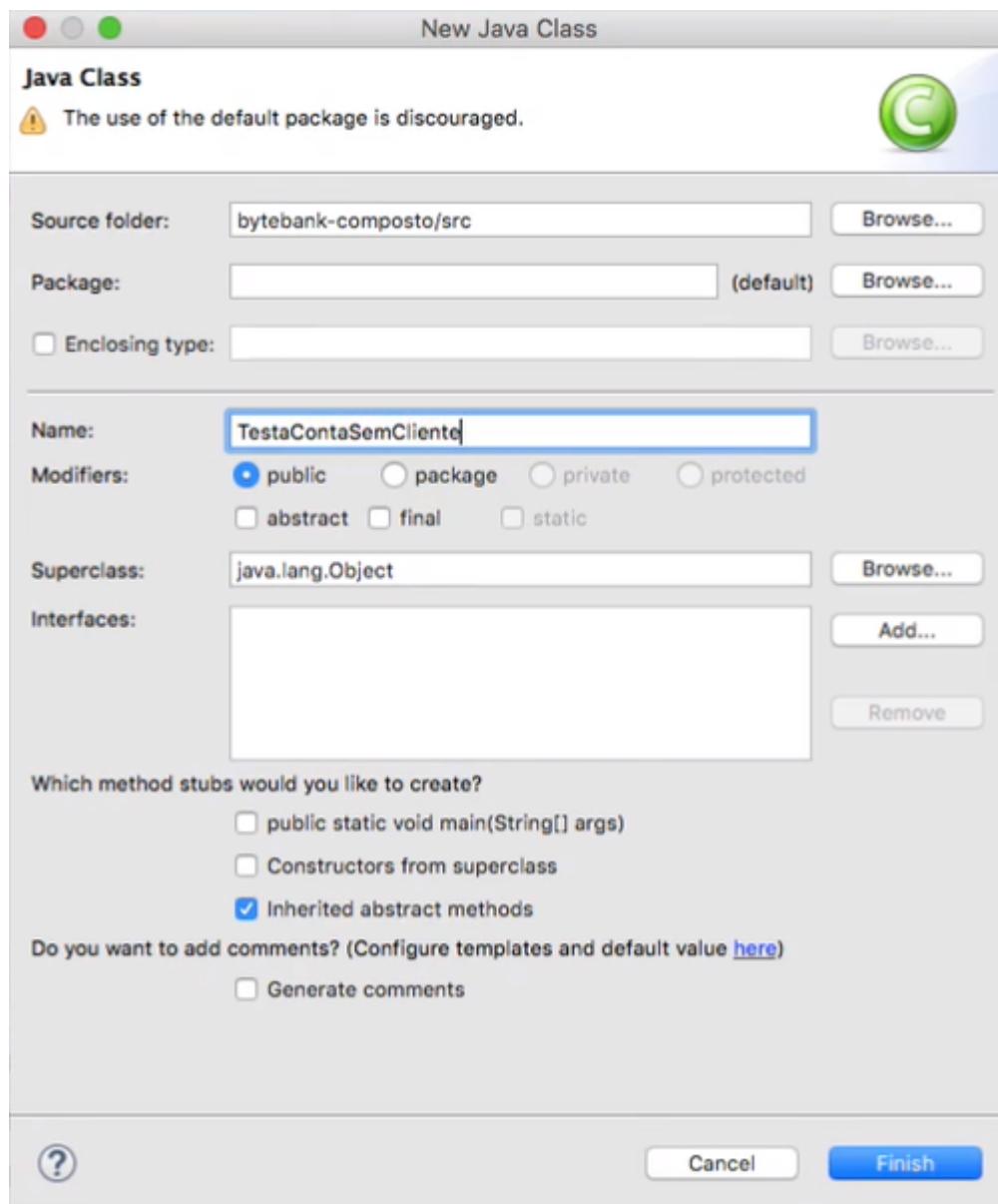
Transcrição

Voltaremos para a questão do valor `0` dos atributos e como isso se dá nos casos do tipo `String`. Qual seria o valor zerado do atributo `titular` da classe `Conta`, uma vez que ela faz referência à outra classe `Cliente`?

```
public class Conta {  
    double saldo;  
    int agencia;  
    int numero;  
    Cliente titular;  
  
    // ...
```

[COPIAR CÓDIGO](#)

Realizaremos um teste para tentar descobrir o que acontece caso não criemos um `Cliente`. Para isso, criaremos uma nova classe chamada `TestaContaSemCliente`.



Criaremos na nova classe o `main`, bem como uma referência para `Conta` chamada `contaDaMarcela` utilizando a palavra-chave `new`.

```
public class TestaContaSemCliente {
    public static void main(String[] args) {
        Conta contaDaMarcela = new Conta();
        System.out.println(contaDaMarcela.saldo);
    }
}
```

COPIAR CÓDIGO

Acionando o `sysout` para `saldo`, o código é compilado e o atributo zerado sem nenhum problema. Mas se tentarmos fazer um procedimento parecido com `titular`, sem definirmos um `Cliente` para este atributo?

```
public class TestaContaSemCliente {  
    public static void main(String[] args) {  
        Conta contaDaMarcela = new Conta();  
        System.out.println(contaDaMarcela.saldo);  
  
        contaDaMarcela.titular.nome = "Marcela";  
        System.out.println(contaDaMarcela.titular.nome);  
    }  
}
```

[COPIAR CÓDIGO](#)

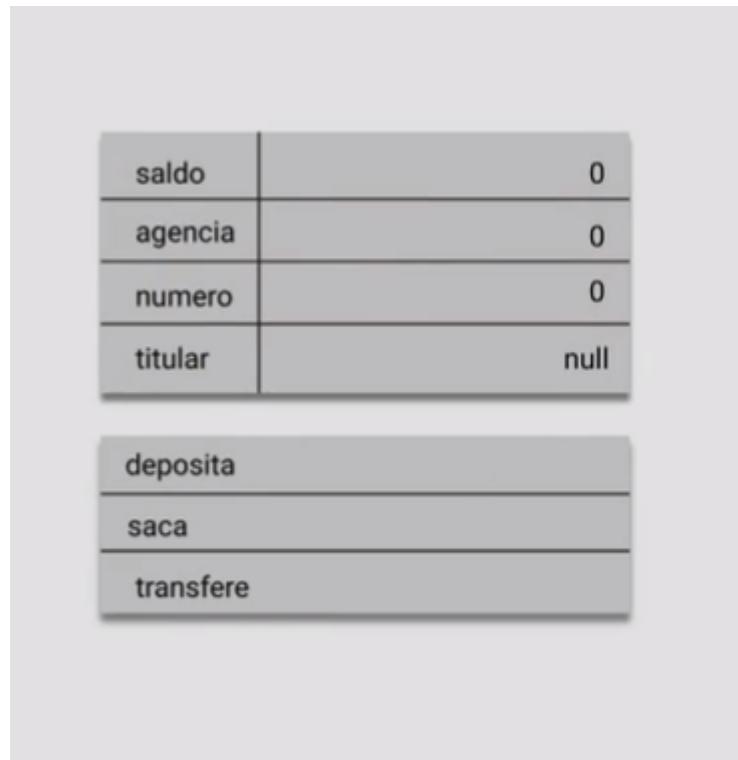
Ao executarmos a aplicação, veremos que houve uma mensagem de erro. Por enquanto, o conteúdo dessa mensagem ficará nebuloso, mas analisaremos os termos da mensagem posteriormente.

Percebam que está em destaque a linha que ocasionou o erro de aplicação, que é `TestaContaSemClinete.java:8`.



The screenshot shows an IDE's console window. The title bar includes tabs for 'Problems', 'Javadoc', 'Declaration', and 'Console'. Below the title bar, the text reads: '<terminated> TestaContaSemCliente [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_45.jdk/Contents/Home/bin/java (Apr 11, 2017, 6:35:31 PM)'. The console output starts with '0.0' followed by a red error message: 'Exception in thread "main" java.lang.NullPointerException at TestaContaSemCliente.main(TestaContaSemCliente.java:8)'.

O "zero" de um atributo ou variável do tipo referência, chamamos de `null`, que significa algo como "referência para lugar nenhum".



Podemos ter uma referência para nada no nosso código. Na linha 7 do nosso código, podemos realizar um `sysout` em `contaDaMarcela` e `titular`.

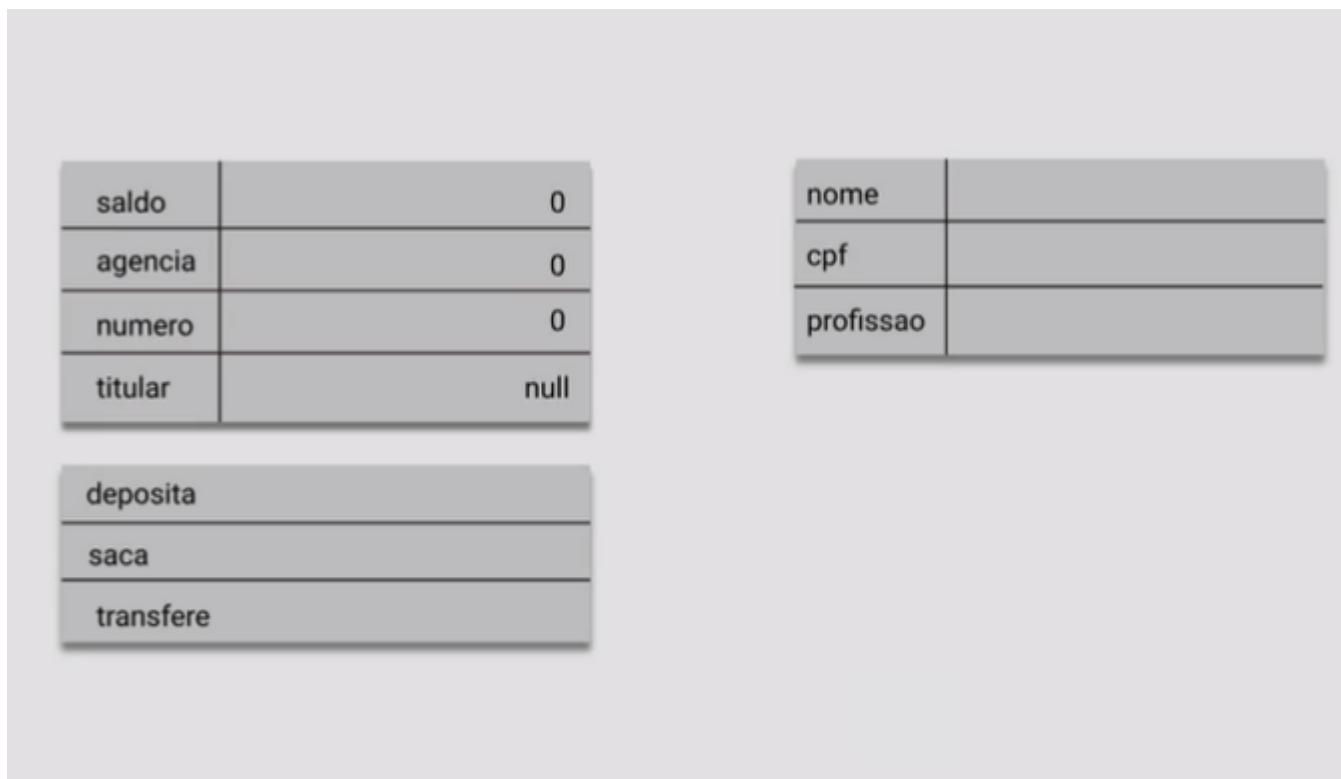
```
public class TestaContaSemCliente {  
    public static void main(String[] args) {  
        Conta contaDaMarcela = new Conta();  
        System.out.println(contaDaMarcela.saldo);  
  
        System.out.println(contaDaMarcela.titular);  
  
        contaDaMarcela.titular.nome = "Marcela";  
        System.out.println(contaDaMarcela.titular.nome);  
    }  
}
```

COPIAR CÓDIGO

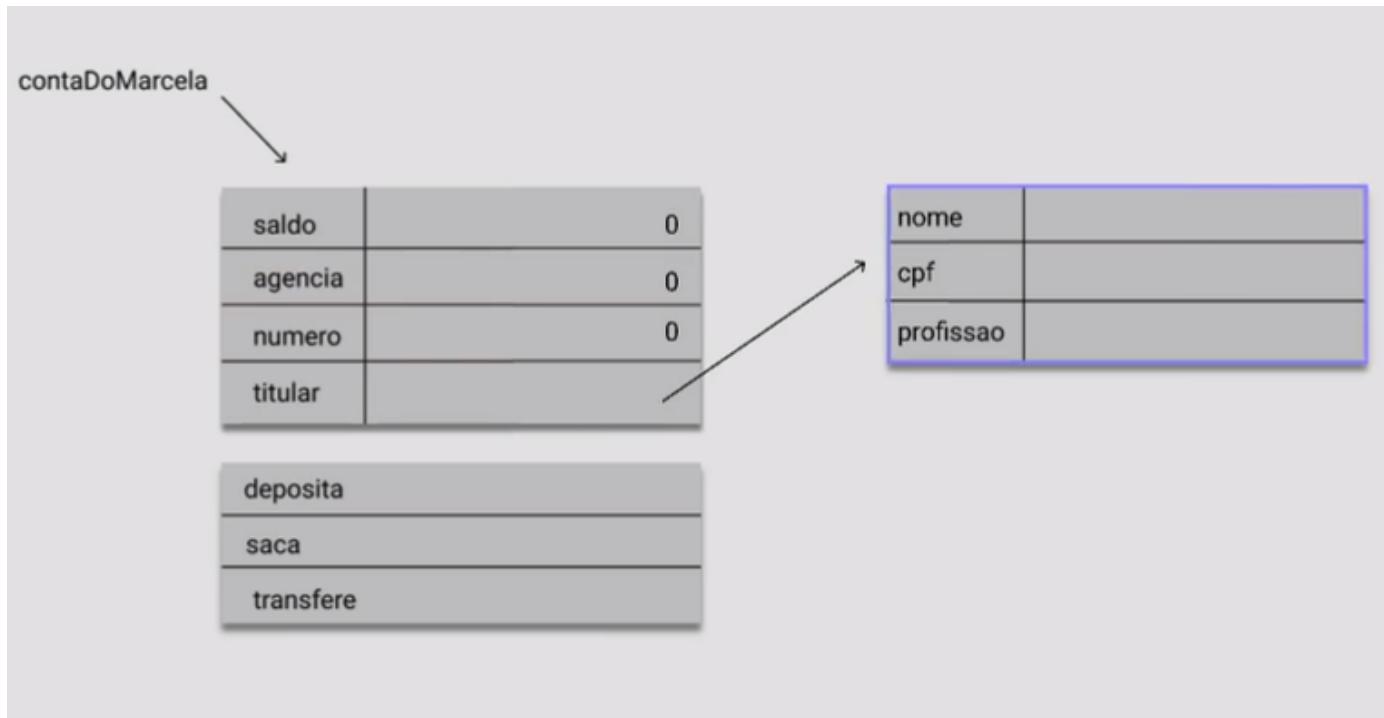
Ao executarmos novamente o código, veremos que o resultado da impressão será a mensagem `null` antes do surgimento do erro.

The screenshot shows an IDE's console window. At the top, there are tabs for 'Problems', '@ Javadoc', 'Declaration', and 'Console'. Below the tabs, the text '<terminated> TestaContaSemCliente [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_45.jdk/Contents/Home/bin/java (Apr 11, 2017, 6:36:32 PM)' is displayed. The console output is:
0.0
null
Exception in thread "main" java.lang.NullPointerException
at TestaContaSemCliente.main(TestaContaSemCliente.java:10)

O nome do `titular` ("Marcela") não faz referência a nenhum tipo `Cliente`. Para que a aplicação seja executada corretamente, precisamos criar um novo cliente e fazer a associação entre `Conta` e `Cliente`.



Já temos no código a associação ao objeto `Conta` através da variável `contaDaMarcela`. Nosso próximo passo é fazer a associação entre `titular` e `nome`. Nas atividades anteriores, declararamos que a variável `paulo` era responsável por essa associação entre objetos. Neste caso, faremos de outro modo.



Faremos com que `titular` deixe de ser `null`, fazendo-o receber um novo cliente:

```
new Cliente .
```

```

public class TestaContaSemCliente {
    public static void main(String[] args) {
        Conta contaDaMarcela = new Conta();
        System.out.println(contaDaMarcela.saldo);

        contaDaMarcela.titular = new Cliente();
        System.out.println(contaDaMarcela.titular);

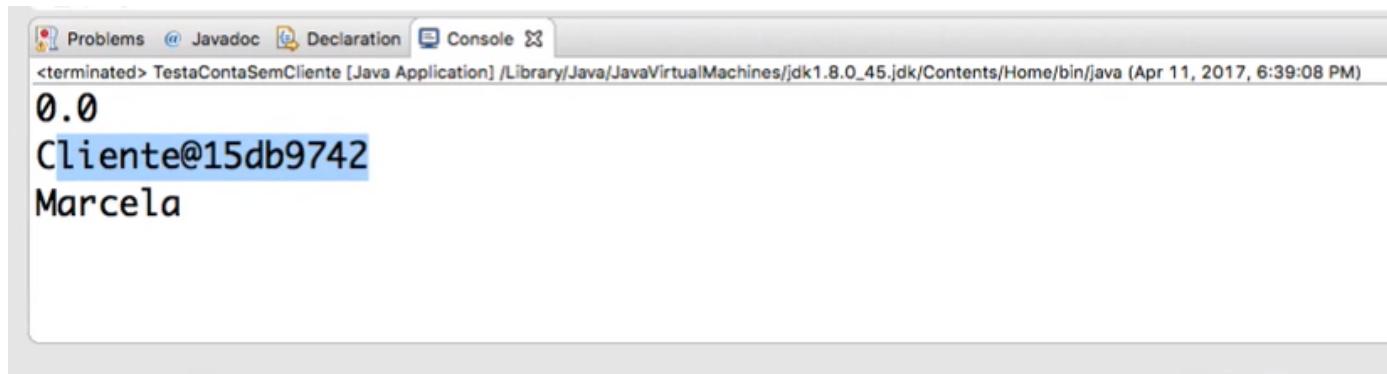
        contaDaMarcela.titular.nome = "Marcela";
        System.out.println(contaDaMarcela.titular.nome);
    }
}

```

COPiar CÓDIGO

Há casos em que não há necessidade de criar uma variável temporária, podemos criar a associação em uma linha, como é o caso. Ao executarmos a aplicação, sera:

impresso um `Id` referente ao `Cliente`, revelando a associação feita entre os objetos.



```
Problems @ Javadoc Declaration Console
<terminated> TestaContaSemCliente [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_45.jdk/Contents/Home/bin/java (Apr 11, 2017, 6:39:08 PM)
0.0
Cliente@15db9742
Marcela
```

Normalmente no Java, são criadas grandes redes de objetos interconectados que se referenciam, e através da invocação de métodos conseguimos fazer com que eles trabalhem entre si. O resultado é que temos códigos curtos, mas que atuam em grandes conjuntos. Com isso, temos uma maior organização no projeto, pois são códigos mais fáceis de ler e de realizar manutenções.

O `null` é uma referência que você encontrará com muita frequência, e não há necessidade de se preocupar com ela.

Uma referência é tida como `null` porque ainda não foi populada.

Para popular uma referência basta inserirmos um valor dentro dela, normalmente através de um `new` ou apontando para uma referência já existente, como fizemos no código anterior com `paulo`.

Um último desafio: lembre-se que podemos setar valores *default*. Na classe `Conta`, podemos dizer que toda vez que uma conta é aberta no **ByteBank** o saldo se inicia com `100`.

```
public class Conta {
    double saldo = 100;
```

```
int agencia;  
int numero;  
Cliente titular;
```

```
// ...
```

[COPIAR CÓDIGO](#)

Do mesmo modo, podemos fazer com o que toda a vez que o `new` é acionado para uma `Conta`, temos um novo `Cliente`. Ou seja, toda `Conta` já se associa a um `Cliente`, com isso, não nos preocuparíamos com o `null` neste caso em particular.

```
public class Conta {  
    double saldo = 100;  
    int agencia;  
    int numero;  
    Cliente titular = new Cliente();
```

```
// ...
```

[COPIAR CÓDIGO](#)

No nosso projeto não é uma opção muito interessante, pois toda a conta tem de ser associada à um cliente novo, banindo a possibilidade de um cliente ter duas contas, por exemplo. Porém, em muitos casos, essa é uma alternativa interessante.

01

Atributos privados e encapsulamento

Transcrição

Neste ponto do curso, já sabemos como trabalhar com métodos e atributos, inclusive, atributos que servem como referência para outros objetos. Veremos qual é o nosso novo fator limitante para o progresso do projeto **ByteBank**.

Criaremos uma nova classe de teste chamada `TesteSacaNegativo`.

Nosso objetivo é ficar com uma quantidade negativa de dinheiro. Usaremos o nome da variável `conta`, mas poderia ser qualquer outro de sua escolha, diferente do tipo `Conta`, que, obrigatoriamente, precisa ser o nome de uma classe que exista no sistema.

Lembrando que é muito comum criar uma variável que tenham o mesmo nome da classe. Depositaremos 100 reais na `conta`.

```
public class TesteSacaNegativo {  
    public static void main(String[] args) {  
        Conta conta = new Conta();  
        conta.deposita(100);  
    }  
}
```

COPIAR CÓDIGO

Feito isso, tentaremos sacar 200 reais, e verificaremos o que acontece quando executamos a aplicação.

```
public class TesteSacaNegativo {  
    public static void main(String[] args) {  
        Conta conta = new Conta();  
        conta.deposita(100);  
        conta.saca(200);  
        System.out.println(conta.saldo);  
    }  
}
```

[COPIAR CÓDIGO](#)

O resultado da aplicação foi a impressão do saldo 100 . Isso ocorreu, pois o saque não foi efetivado, já que o valor disponível na conta não era o suficiente.

Quando mantemos o "Ctrl" pressionado e passamos o mouse sobre a código, vemos que o método retornou o valor false . Podemos passar o valor booleano diretamente para o sysout , ou seja, não precisamos sempre guardar o retorno de um método dentro de uma variável.

```
public class TesteSacaNegativo {  
    public static void main(String[] args) {  
        Conta conta = new Conta();  
        conta.deposita(100);  
        System.out.println(conta.saca(200));  
        System.out.println(conta.saldo);  
    }  
}
```

[COPIAR CÓDIGO](#)

Ao executarmos novamente aplicação, veremos o valor `false` impresso.

Conseguimos estipular o critério de que *nenhuma conta pode ter valores negativos*.

Porém, existe um truque que tentaremos fazer. Ao tentarmos sacar 101 reais da nossa conta, cujo saldo é 100, o procedimento não será efetivado. A não ser que, diretamente no atributo `saldo`, estipulamos que o seu valor é o `saldo - 101`.

```
public class TesteSacaNegativo {  
    public static void main(String[] args) {  
        Conta conta = new Conta();  
        conta.deposita(100);  
        System.out.println(conta.saca(101));  
        System.out.println(conta.saldo);  
  
        conta.saldo = conta.saldo - 101;  
        System.out.println(conta.saldo);  
    }  
}
```

[COPIAR CÓDIGO](#)

No término da execução do programa, veremos que o valor impresso será -1.0.

Supondo que seja um requisito do **ByteBank** que não haja valores negativos nas contas, temos um problema.

Uma solução é avisar para os funcionários do nosso banco para que nunca acessem o atributo `saldo` diretamente, e sim, invocando o método adequado.

O ideal é que sempre trabalhemos utilizando os **métodos**, nunca diretamente os atributos. Podemos utilizar a analogia do funcionamento de um carro: utilizamos o pedal de aceleração para que o carro se locomova, não precisamos abrir o capo e revirarmos mecanismos de injeção de gasolina de forma manual para gerar a aceleração necessária.

A ideia é utilizar a interface adequada para a melhor funcionalidade de um sistema.

A forma mecânica de funcionamento do carro está escondida, ou seja, *encapsulada*.

O que precisamos saber operar é a metodologia de funcionamento do carro, o mesmo vale para a conta bancária.

Queremos, portanto, que a manipulação direta de atributos seja proibida.

```
public class TesteSacaNegativo {  
    public static void main(String[] args) {  
        Conta conta = new Conta();  
        conta.deposita(100);  
        System.out.println(conta.saca(200));  
        System.out.println(conta.saldo);  
  
        //proibido  
        conta.saldo = conta.saldo - 101;  
        System.out.println(conta.saldo);  
    }  
}
```

[COPIAR CÓDIGO](#)

No Java, existe a possibilidade de ocultarmos um atributo, deixá-lo privado. Na classe `Conta`, escreveremos ao lado do atributo `saldo` que queremos o encapsular. Para isso, utilizamos a palavra-chave `private`

```
public class Conta {  
    private double saldo;  
    int agencia;  
    int numero;  
    Cliente titular;
```

```
// ...
```

COPiar Código

A partir do momento em que um atributo se torna **privado**, isso quer dizer que ele não pode ser lido ou modificado, a não ser na própria classe. Esse é o conceito principal de encapsulamento.

Ainda existe um problema: na classe `TesteSacaNegativo`, não podemos mais imprimir o valor de `saldo` através do `sysout`. É preciso utilizar um novo método para acessar o atributo `saldo`.

Na classe `Conta`, criaremos um método que devolve/informe o `saldo`. Chamaremos esse novo método de `pegaSaldo`, que não precisará receber parâmetro, mesmo assim, os parênteses são obrigatórios.

Ao lado esquerdo do método, colocaremos o seu retorno: um `double`. Escreveremos, também, o `public`. Dentro do método, diremos que ele simplesmente retorna o valor de `saldo`, utilizando a palavra-chave `return`.

```
public class Conta {  
    private double saldo;  
    int agencia;  
    int numero;  
    Cliente titular;  
  
    public void deposita(double valor) {...}  
  
    public boolean saca(double valor) {...}  
  
    public boolean transfere(double valor, Conta destino, Conta ^n|
```

```
public double pegaSaldo() {  
    return this.saldo;  
}
```

[COPIAR CÓDIGO](#)

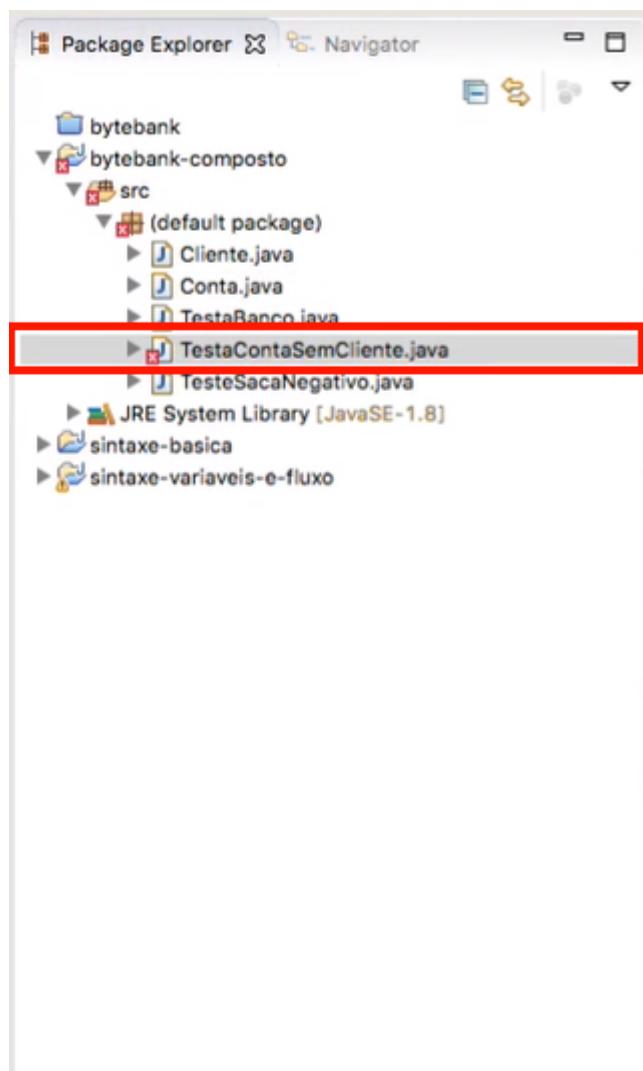
É interessante que as funções sejam bem localizadas no código, por isso, o objetivo desse método é simplificar os processos de manutenção do sistema sem que precisemos fazer alterações em múltiplos trechos do código.

De volta a classe `TesteNegativo`, iremos invocar o método `pegaSaldo`:

```
public class TesteSacaNegativo {  
    public static void main(String[] args) {  
        Conta conta = new Conta();  
        conta.deposita(100);  
        System.out.println(conta.saca(101));  
  
        conta.saca(101);  
  
        System.out.println(conta.pegaSaldo());  
    }  
}
```

[COPIAR CÓDIGO](#)

Há uma classe no nosso projeto que ainda está acessando o atributo `saldo` da maneira antiga.



O `TestaContaSemCliente` está dessa forma:

```
public static void main(String[] args) {  
    Conta contaDaMarcela = new Conta();  
    System.out.println(contaDaMarcela.saldo);  
  
    // ...
```

[COPIAR CÓDIGO](#)

Vamos corrigir o código :

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

```
Conta contaDaMarcela = new Conta();  
System.out.println(contaDaMarcela.pegaSaldo());
```

```
// ...
```

[COPIAR CÓDIGO](#)

 03

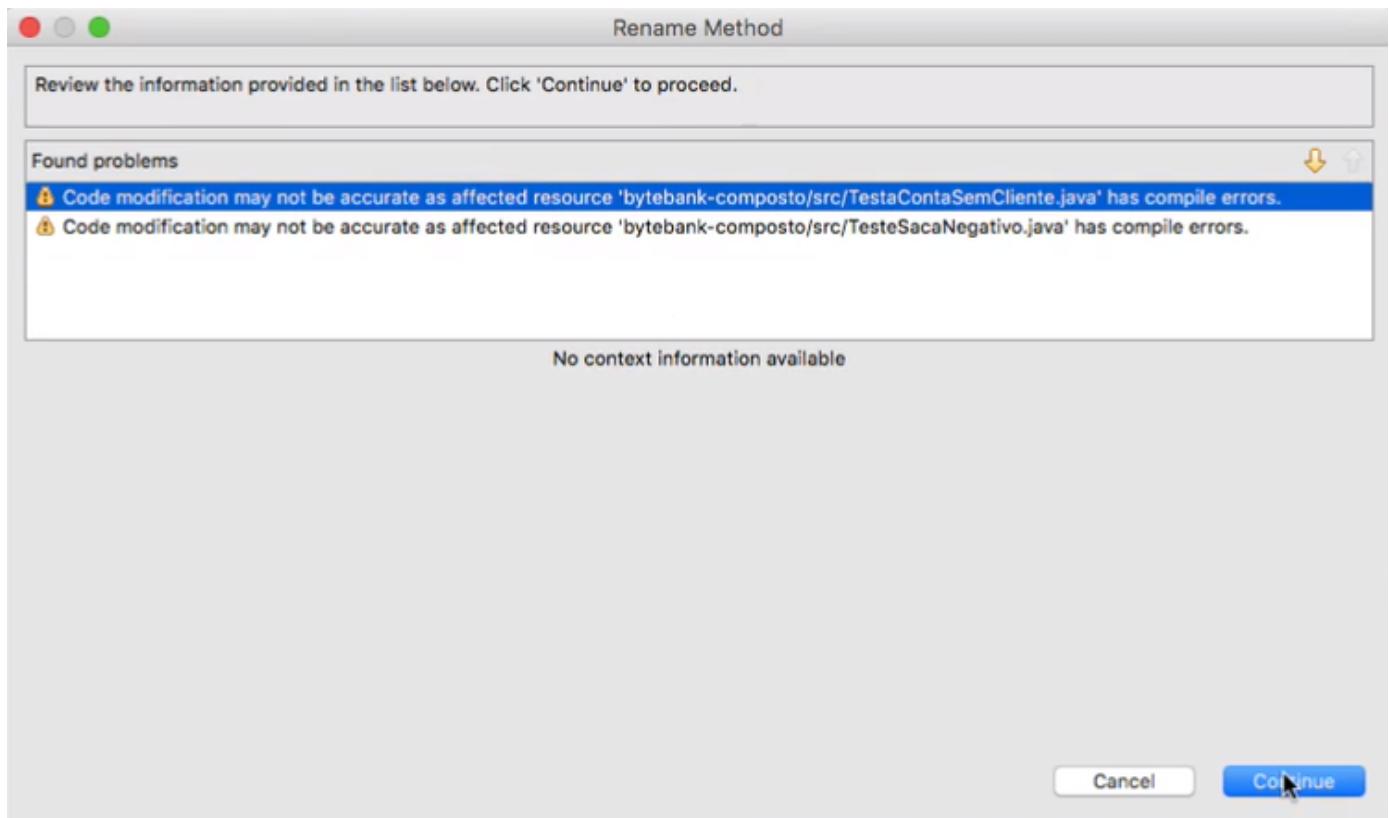
Getters e Setters

Transcrição

O nome do método `pegaSaldo()` que criamos, poderia ter qualquer outro nome. Por uma questão de convenção, alteraremos o nome para `getSaldo()`.

Devido à essa alteração, os outros arquivos não serão compilados, porque estarão com o nome antigo `pegaSaldo()`. Para resolvemos esse problema, na classe `Conta`, selecionaremos o método e daremos um clique duplo sobre ele. Feito isso, escolheremos ao opção "Refactor > Rename".

Surgirá uma caixa de diálogo e pressionamos "Continue". Com isso, o nome do método foi alterado em todos os arquivos.



O nome desse tipo de método que simplesmente *exibe uma informação* é **getter**. Não se trata de um elemento que compõe a sintaxe do Java, **get** não é uma palavra-chave do Java.

Dizemos que temos um método *getter* para `saldo`, pois este atributo é privado e sentimos a necessidade ao longo do projeto de acessar a informação contida em `saldo`.

Percebam que não há a necessidade de criarmos novo método equivalente - algo como "`setSaldo`" - para *modificar o atributo* `saldo`, essas modificações serão feitas pelos já conhecidos `saca()`, `deposita()` e `transfere()`. Queremos que as alterações de saldo em nossas contas do **ByteBank** sejam sempre feitas através de **saques, depósitos ou transferências**.

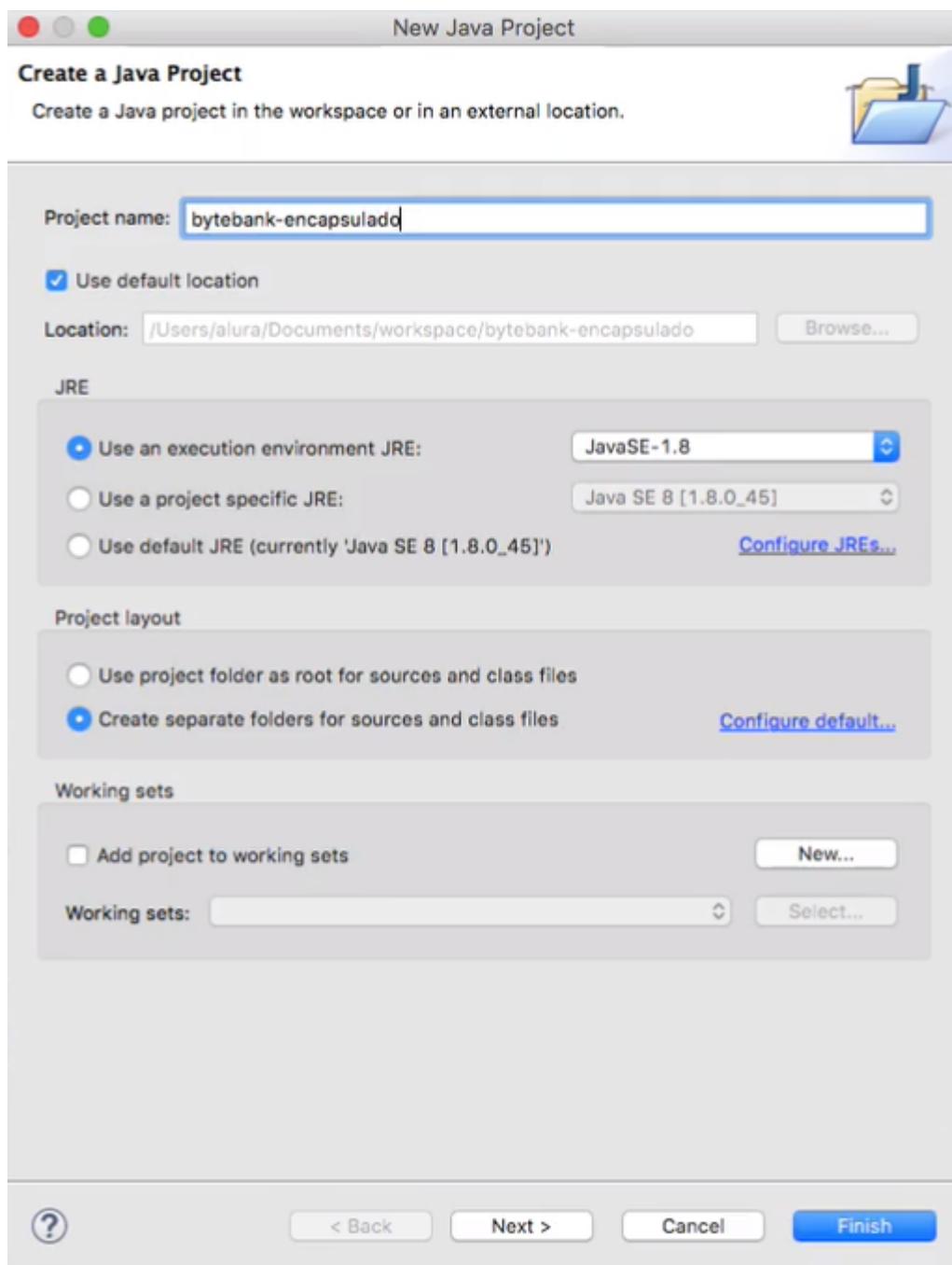
A ideia é diferente para `agencia`, `numero` e `titular`. Não temos métodos para alterar esses atributos, e por enquanto, só conseguimos fazer modificações diretas, o que quer dizer que podemos inserir números negativos como valores, por exemplo:

Com o tempo, entenderemos que o ideal é que todos os atributos sejam privados.

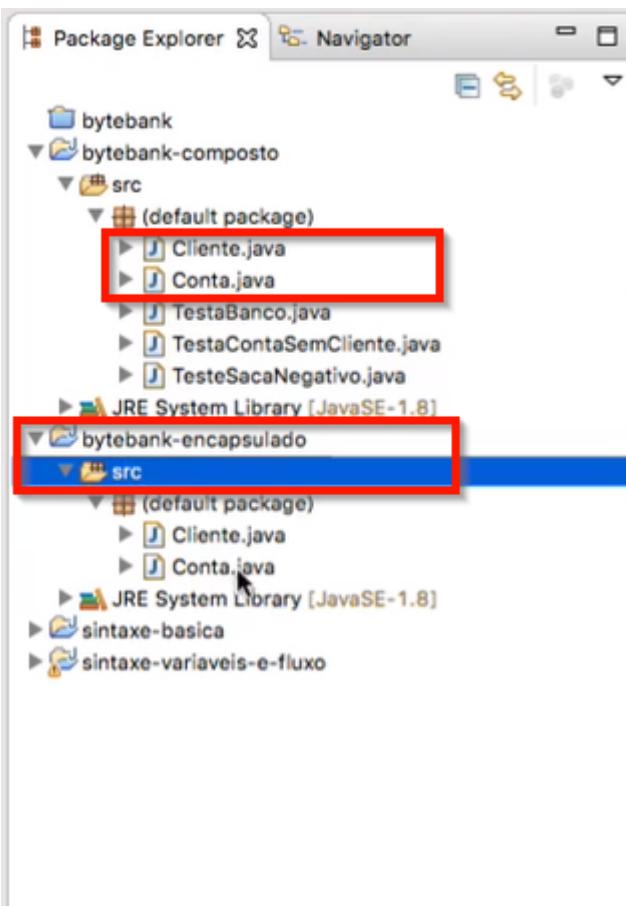
Existem casos raros de atributos públicos, mas são realmente exceções.

Criaremos um novo projeto Java que chamaremos de `bytebank-encapsulado`.

"Encapsulado" é o termo utilizado para elementos ocultos, escondidos.



Na área *Package Explorer*, copiaremos as classes `Cliente` e `Conta` e as colaremos no default package do novo projeto `bytebank-encapsulado`.



Utilizando o atalho "Ctrl + W", fecharemos todas as abas do Eclipse. Também fecharemos o projeto `bytebank-composto` para não confundirmos as classes dos dois projetos, tornado o ambiente de trabalho mais claro e limpo.

Na classe `Conta`, transformaremos todos os atributos em `private`.

```
public class Conta {  
    private double saldo;  
    private int agencia;  
    private int numero;  
    private Cliente titular;  
  
    public void deposita(double valor) {  
        this.saldo += valor;  
    }  
}
```

COPIAR CÓDIGO

O fato de tornarmos um atributo privado facilita a modificação e atualização do código. Com uma classe sendo responsável por seus próprios atributos, a manutenção do sistema se torna localizada, por conseguinte, mais simples.

Para que os atributos sejam acessados fora da classe, utilizaremos os *getters*. Na classe `Conta` criaremos um método público denominado `getNumero()` que devolve um `int` e retorna `numero`.

```
public class Conta {  
    private double saldo;  
    private int agencia;  
    private int numero;  
    private Cliente titular;  
  
    public void deposita(double valor) {...}  
  
    public boolean saca (double valor) {...}  
  
    public boolean transfere(double valor, Conta destino, Conta origem){  
        double novoSaldo = pegaSaldo() - valor;  
        if (novoSaldo < 0){  
            System.out.println("Saldo insuficiente");  
            return false;  
        }  
        saldo = novoSaldo;  
        destino.deposita(valor);  
        return true;  
    }  
    public double pegaSaldo(){  
        return saldo;  
    }  
    public int getNumero(){  
        return this.numero;  
    }  
}
```

COPIAR CÓDIGO

Além de termos um método que devolve qual é o `numero` de uma conta, queremos também um método que *altere* esse mesmo `numero`. Esse tipo de método é chamado de **set**.

Diferente do `get`, há um parâmetro a ser passado para o método, afinal queremos modificar o número e precisamos informar qual será essa modificação. Usaremos uma variável `novoNumero` e o método não retornará nada.

```
public class Conta {  
    private double saldo;  
    private int agencia;  
    private int numero;  
    private Cliente titular;  
  
    public void deposita(double valor) {...}  
  
    public boolean saca(double valor) {...}  
  
    public boolean transfere(double valor, Conta destino, Conta origem) {...}  
  
    public double pegaSaldo() {...}  
  
    public int getNumero() {  
        return this.numero;  
    }  
  
    public void setNumero(int novoNumero) {  
        this.numero = novoNumero;  
    }  
}
```

[COPIAR CÓDIGO](#)

Criaremos numa nova classe para utilizar tanto o `getter` como o `setter`. Essa nova classe será chamada de `TestaGetESet`. Na nova classe, criaremos uma nova conta chamada `conta` e a instanciaremos através da palavra-chave `new`. Feito isso, escreveremos o `numero` desta nova `conta` como sendo `1337`.

```
public class TestaGetESet {  
    public static void main(String[] args) {  
        Conta conta = new Conta();  
        conta.numero = 1337  
    }  
}
```

[COPIAR CÓDIGO](#)

Da forma como o nosso código está escrito ele não será compilado, afinal, `numero` é um atributo privado e *precisa* ser acessado através de um método. Utilizaremos o método `setNumero()`

```
public class TestaGetESet {  
    public static void main(String[] args) {  
        Conta conta = new Conta();  
        conta.setNumero(1337);  
    }  
}
```

[COPIAR CÓDIGO](#)

Uma das vantagens de utilizar os métodos, é que dentro do próprio método `setNumero()` podemos adicionar `if`s, gerando especificações, mensagens de erro, e assim por diante.

Para imprimirmos o valor de `numero`, utilizamos o `sysout` e o método `getNumero()`. Ao executarmos a aplicação, teremos o resultado de `1337`.

```
public class TestaGetESet {  
    public static void main(String[] args) {  
        Conta conta = new Conta();  
        conta.setNumero(1337);  
    }  
}
```

```
        System.out.println(conta.getNumero());  
    }  
}
```

[COPIAR CÓDIGO](#)

Como já foi dito, no caso do atributo `saldo`, não criaremos um método setter, pois trabalharemos com os métodos `deposita()`, `saca()` e `transfere()` para modificar seu valor.

Criaremos métodos `get` e `set` para o atributo `agencia`, mas dessa vez, de uma maneira mais simples.

Antes, voltemos ao método `setNumero()` na classe `Conta`.

```
public void setNumero(int novoNumero) {  
    this.numero = novoNumero;  
}
```

[COPIAR CÓDIGO](#)

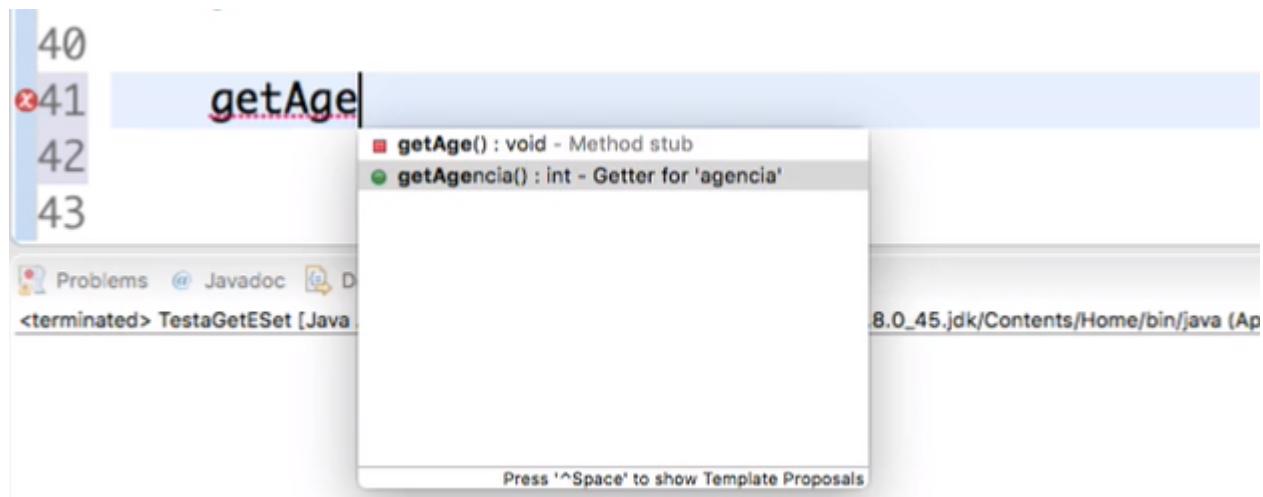
Em muitos casos, em vez de escrever o nome da variável `novoNumero`, utiliza-se o mesmo nome do atributo, no caso, `numero`.

A princípio, podemos nos confundir, mas basta nos atentarmos para o uso da palavra-chave `this`, ao lado esquerdo do código, que marca a referência ao atributo.

```
public void setNumero(int numero) {  
    this.numero = numero;  
}
```

[COPIAR CÓDIGO](#)

Escreveremos o método get para o atributo `agencia`. Percebiam que basta iniciarmos a escrever o código e pressionarmos o atalho "Ctrl + Space", será sugerido a criação automática de `getAgencia()`. Escolheremos esta opção e pressionaremos "Enter".



Feito isso, o código será automaticamente escrito na classe `Conta`.

```
public class Conta {  
    // atributos  
    // método deposita  
    // método saca  
    // método transfere  
    // método pegaSaldo  
  
    public int getNumero() {  
        return this.numero;  
    }  
  
    public void setNumero(int numero) {  
        this.numero = numero;  
    }  
  
    public int getAgencia() {  
        return agencia;
```

```
}
```

[COPIAR CÓDIGO](#)

Adicionaremos o `this` para marcarmos bem o que é atributo e variável temporária.

```
public int getAgencia() {  
    return this.agencia;  
}
```

[COPIAR CÓDIGO](#)

Podemos realizar o mesmo procedimento com o método `set` do atributo `agencia`. O Eclipse possui teclas de atalho que facilitam muito a construção do código.

No cabeçalho de ferramentas existe opção de gerar *getters* e *setters*, basta selecionarmos em "Source > Generate Getters and Setters", mas não há necessidade de acessá-los desta forma, já que o "Ctrl + Space" disponibiliza a mesma função, neste caso.

05

Getters e Setters de referência

Transcrição

Elevaremos o nível de dificuldade na prática de métodos **getters** e **setters** considerando o tipo `Cliente`.

Neste ponto, já trabalhamos com os atributos `saldo`, `agencia` e `numero`. Lembrando que para o caso de `saldo`, não precisamos utilizar um setter, porque outros métodos como `deposita()` e `saca()` já servem para modificar o atributo.

É possível que haja atributos em que não exista a necessidade tanto de um método `get` quanto de um `set`, como por exemplo, uma conexão para o banco de dados que só possui um uso interno com relação à própria classe `Conta`.

No caso do atributo `titular`, é de nosso interesse que possamos acessar o titular de uma determinada conta e modificá-lo.

Na classe `TestaGetESet`, criaremos um novo `Cliente` e faremos a atribuição do `titular` cujo valor é `null`, para um novo titular identificado como `paulo`.

```
public class TestaGetESet {  
    public static void main(String[] args) {  
        Conta conta = new Conta();  
  
        conta.setNumero(1337);  
        System.out.println(conta.getNumero());
```

```
    Cliente paulo = new Cliente();
    conta.titular = paulo;
}
}
```

[COPIAR CÓDIGO](#)

O código desta forma não compila, pois `titular` é um atributo privado.

Precisaremos invocar algum método para finalizar nossa operação.

Poderíamos acionar um método hipotético, como `setTitular()`. Esse código também não compila, pois ainda não criamos seu **setter**.

```
public class TestaGetESet {
    public static void main(String[] args) {
        Conta conta = new Conta();

        conta.setNumero(1337);
        System.out.println(conta.getNumero());

        Cliente paulo = new Cliente();
        conta.setTitular(paulo);
    }
}
```

[COPIAR CÓDIGO](#)

Na classe `Conta`, criamos no novo método `setTitular()`. Feito isso, criaremos, também, o método `getTitular()`.

```
public class Conta {
    // atributos
```

```
// método deposita  
// método saca  
// método transfere  
// método pegaSaldo  
  
public int getNumero() {  
    return this.numero;  
}  
  
public void setNumero(int numero) {  
    this.numero = numero;  
}  
  
public int getAgencia() {  
    return this.agencia;  
}  
  
public void setAgencia(int agencia){  
    this.agencia = agencia;  
}  
  
public void setTitular(Cliente titular) {  
    this.titular = titular;  
}  
  
public Cliente getTitular() {  
    return titular;  
}  
}
```

COPIAR CÓDIGO

Feito isso, veremos que o nosso código na classe `TestaGetESet` já está sendo compilado e podemos fazer alterações no atributo `titular`. Mudaremos o nome

titular para paulo silveira e queremos incluir cpf e profissao .

Poderíamos modificar o atributo nome de forma direta:

```
public class TestaGetESet {  
    public static void main(String[] args) {  
        Conta conta = new Conta();  
  
        conta.setNumero(1337);  
        System.out.println(conta.getNumero());  
  
        Cliente paulo = new Cliente();  
        paulo.nome = "paulo silveira";  
  
        conta.setTitular(paulo);  
    }  
}
```

[COPIAR CÓDIGO](#)

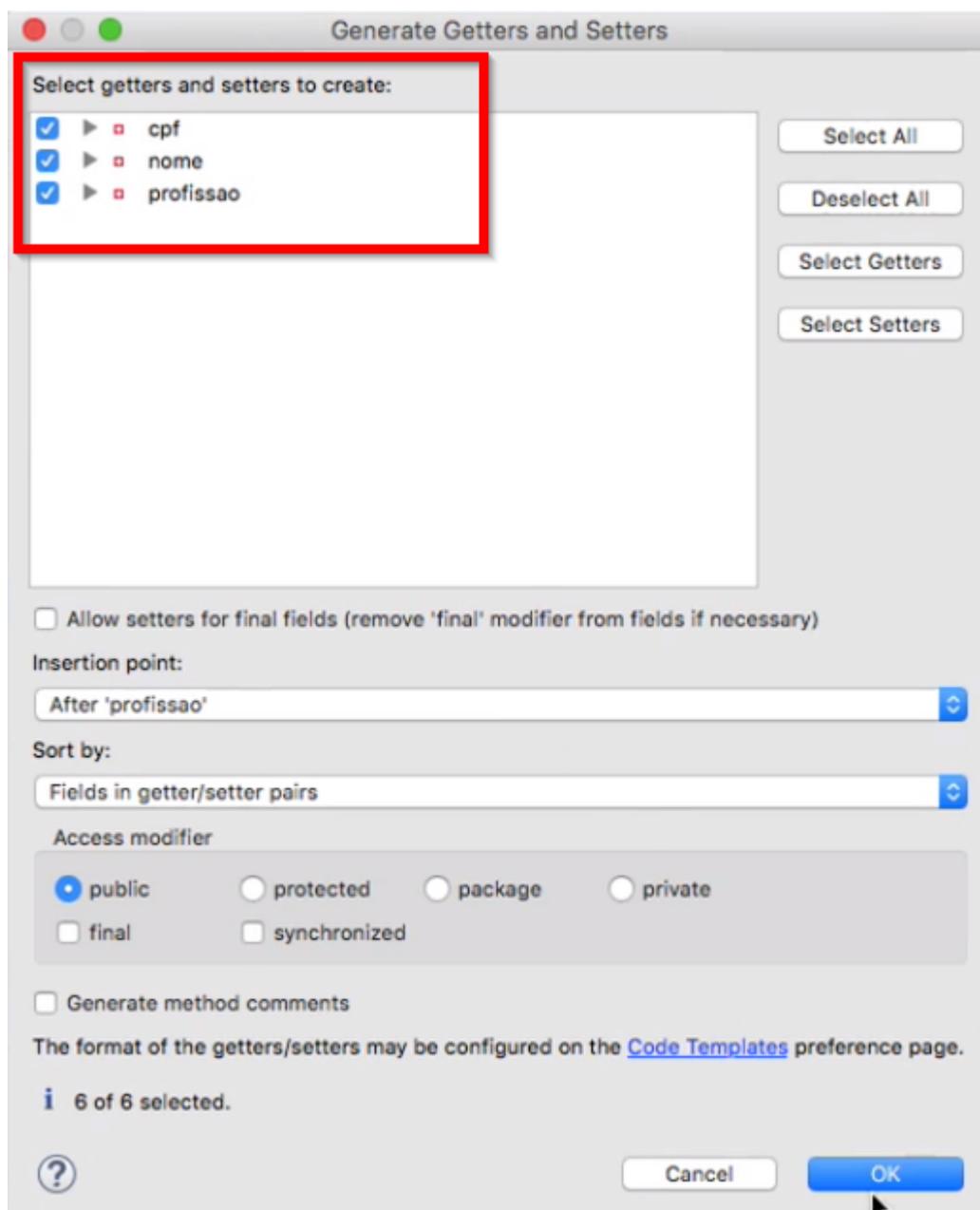
Essa modificação direta foi possível, pois na classe Cliente todos os atributos são não-privados, diferentemente da classe Conta .

Vamos transformar os atributos da classe Cliente em *privados*, pois queremos colocar padrões específicos em cada atributo, como não permitir números em nome , por exemplo.

```
public class Cliente {  
    private String nome;  
    private String cpf;  
    private String profissao;  
}
```

[COPIAR CÓDIGO](#)

Para que você não precise ficar escrevendo todos os getters e setters necessários, no cabeçalho do Eclipse ação "Source > Generate Getters and Setters..." Em "Select getters and setters to create" selecionaremos todas as opções de atributos.



Com isso, todos os métodos foram automaticamente gerados.

```
public class Cliente {  
    private String nome;  
    private String cpf;  
    private String profissao;
```

```
public String getNome() {  
    return nome;  
}  
  
public String getCpf() {  
    return cpf;  
}  
  
public String getProfissao() {  
    return profissao;  
}  
  
public void setProfissao(String profissao) {  
    this.profissao = profissao  
}  
}
```

[COPIAR CÓDIGO](#)

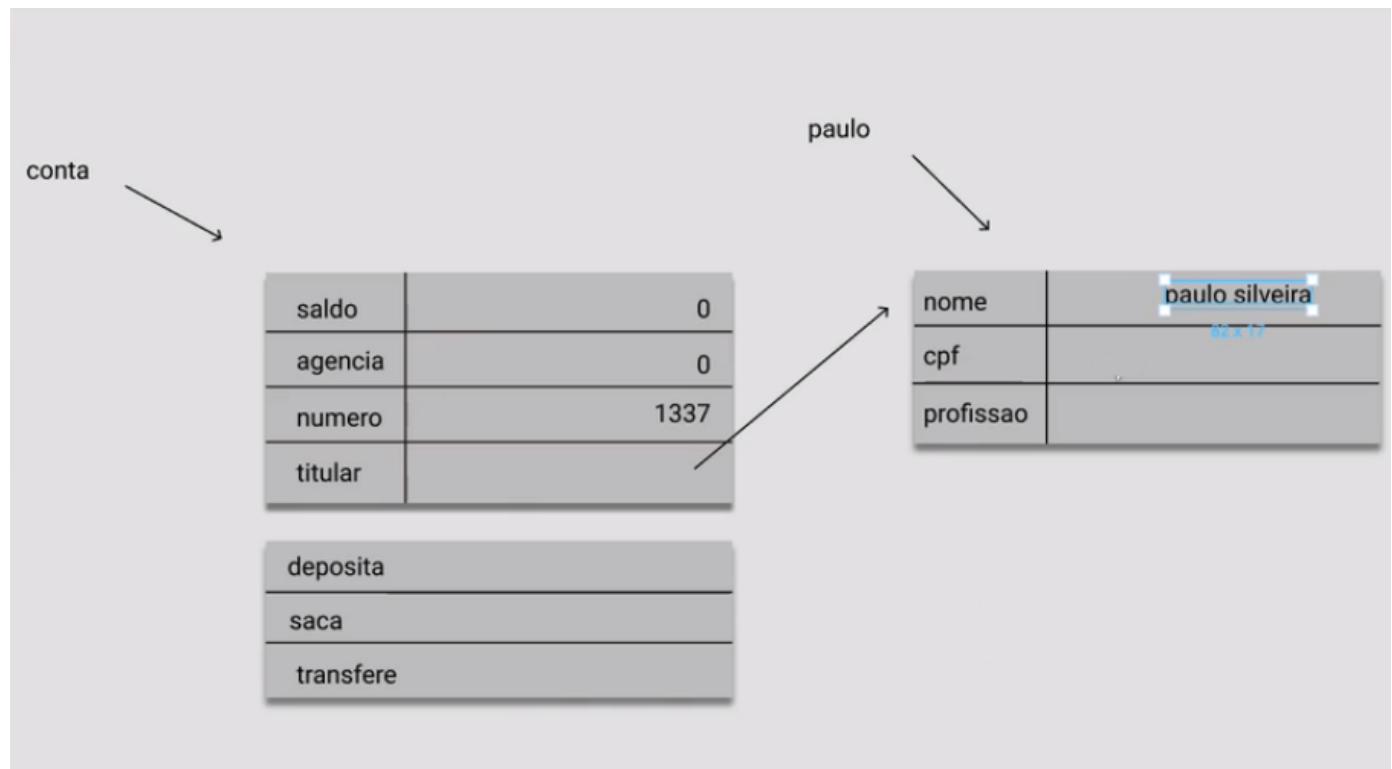
Com isso, na classe `TestaGetESet`, não podemos mais alterar diretamente os atributos de `Cliente`, precisamos invocar métodos.

```
public class TestaGetESet {  
    public static void main(String[] args) {  
        Conta conta = new Conta();  
  
        conta.setNumero(1337);  
        System.out.println(conta.getNumero());  
  
        Cliente paulo = new Cliente();  
        paulo.setNome("paulo silveira");  
  
        conta.setTitular(paulo);  
    }  
}
```

```
}
```

[COPIAR CÓDIGO](#)

Nosso código no diagrama está representado da seguinte maneira: temos uma variável `conta` que faz referência a um objeto `Conta`. O atributo `titular` da conta bancária para o atributo `nome` do objeto `Cliente`, referenciado pela variável `paulo`.



Tentamos imprimir apenas o `titular` da `conta` para averiguarmos como o código se comporta.

```
public class TestaGetESet {  
    public static void main(String[] args) {  
        Conta conta = new Conta();  
  
        conta.setNumero(1337);  
        System.out.println(conta.getNumero());  
    }  
}
```

```
Cliente paulo = new Cliente();
paulo.setNome("paulo silveira");

conta.setTitular(paulo);

System.out.println(conta.getTitular());
}

}
```

[COPIAR CÓDIGO](#)

Ao executarmos a aplicação, veremos que foi impresso o valor de `Cliente@15db9742`. Não era esse o resultado que estávamos querendo, pois esse é o valor da referência ao objeto, e nós queremos imprimir o atributo `nome` do objeto.

```
public class TestaGetESet {
    public static void main(String[] args) {
        Conta conta = new Conta();

        conta.setNumero(1337);
        System.out.println(conta.getNumero());

        Cliente paulo = new Cliente();
        paulo.setNome("paulo silveira");

        conta.setTitular(paulo);

        System.out.println(conta.getTitular());
    }
}
```

[COPIAR CÓDIGO](#)

Reconfiguraremos a linha, utilizando os métodos getters para chegarmos à informação do atributo `nome` do objeto.

```
public class TestaGetESet {  
    public static void main(String[] args) {  
        Conta conta = new Conta();  
        conta.setNumero(1337);  
        System.out.println(conta.getNumero());  
  
        Cliente paulo = new Cliente();  
        paulo.setNome("paulo silveira");  
  
        conta.setTitular(paulo);  
  
        System.out.println(conta.getTitular().getNome());  
    }  
}
```

[COPIAR CÓDIGO](#)

Ao rodarmos o programa, veremos que o resultado impresso é paulo silveira , como o esperado.

Caso seja do nosso interesse alterar alguma informação do titular, o percurso é parecido. Iremos incluir a profissão programador: acionaremos `getTitular()` e depois o `setProfissao()` .

```
System.out.println(conta.getTitular().getNome());  
conta.getTitular().setProfissao("programador");
```

[COPIAR CÓDIGO](#)

Podemos quebrar o código em duas linhas.

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

```
Conta conta = new Conta();

conta.setNumero(1337);
System.out.println(conta.getNumero());

Cliente paulo = new Cliente();
paulo.setNome("paulo silveira");

conta.setTitular(paulo);

System.out.println(conta.getTitular().getNome());

conta.getTitular().setProfissao("programador");
//agora em duas linhas:
Cliente titularDaConta = conta.getTitular();
titularDaConta.setProfissao("programador");
}

}
```

[COPIAR CÓDIGO](#)

O resultado é mesmo, a única diferença é que criamos uma variável temporária. Podemos quebrar a linha do código para facilitar a leitura nos momentos em que ela estiver muito grande.

A variável `titularDaConta` possui o mesmo endereço da variável `paulo`. Como podemos verificar, realizando o `sysout`.

```
public class TestaGetESet {
    public static void main(String[] args) {
        Conta conta = new Conta();

        conta.setNumero(1337);
        System.out.println(conta.getNumero());
```

```
Cliente paulo = new Cliente();
paulo.setNome("paulo silveira");

conta.setTitular(paulo);

System.out.println(conta.getTitular().getNome());

conta.getTitular().setProfissao("programador");
//agora em duas linhas:
Cliente titularDaConta = conta.getTitular();
titularDaConta.setProfissao("programador");

System.out.println(titularDaConta);
System.out.println(paulo);
System.out.println(conta.getTitular());

}

}
```

[COPIAR CÓDIGO](#)

Ao executarmos a aplicação, veremos que todas as referências são para o mesmo endereço de memória `Cliente@15db9742`.

 08

Para saber mais: Cuidado com o Modelo Anêmico

Durante o aprendizado dos Getters e Setters é normal pensar sempre na necessidade deles para alterar algum estado dos nossos objetos.

Mas o uso dessa prática nem sempre é a mais indicada e expressa a realidade.

Observe a classe `Conta` representada abaixo que usa apenas getter e setters como métodos:

```
class Conta{  
    private String titular;  
    private double saldo;  
  
    public void setTitular(String titular){  
        this.titular = titular;  
    }  
  
    public String getTitular(){  
        return titular;  
    }  
  
    public void setSaldo(double saldo){  
        this.saldo = saldo;  
    }  
  
    public double getSaldo(){  
        return saldo;  
    }  
}
```

[COPIAR CÓDIGO](#)

Continuamos usando atributos privados e nosso modelo parece seguir perfeitamente a proposta do encapsulamento onde a própria classe gerencia o seus estados(atributos). Uma utilização clássica dessa Conta nos levaria ao seguinte cenário:

```
Conta conta = new Conta();
conta.setTitular("Fábio")
conta.setSaldo(100.0);
```

[COPIAR CÓDIGO](#)

Tudo parece perfeito, agora imagine que seja necessário sacar 50.0 dessa conta. Essa operação vai exigir que o saldo seja suficiente. Uma simples verificação como a seguir asseguraria que o saldo não tenha ficado negativo. Nesse nosso exemplo não há limite além do saldo :)

```
Conta conta = new Conta();
conta.setTitular("Fábio")
conta.setSaldo(100.0);

double valorSaque = 50.0

if(conta.getSaldo() >= valorSaque){
    double novoSaldo = conta.getSaldo() - valorSaque;
    conta.setSaldo(novoSaldo)
}
```

[COPIAR CÓDIGO](#)

Funcionou! Mas um problema é que essa lógica de restringir o saque à quantidade de saldo vai ter que ser refeita toda vez que for necessária uma operação de saque na nossa conta. Além do problema de duplicações desse trecho, um problemão para encapsulamento é que quem está de fato controlando as regras do saldo da conta é quem está usando a Conta. Em outras palavras nada impede que alguém implemente um limite extra para isso e tenha uma regra completamente diferente dos demais objetos do tipo Conta:

```
Conta conta = new Conta();
conta.setTitular("Fábio");
conta.setSaldo(100.0);

double valorSaque = 50.0;

if(conta.getSaldo() + 1000.0 >= valorSaque){
    double novoSaldo = conta.getSaldo() - valorSaque;
    conta.setSaldo(novoSaldo)
}
```

[COPIAR CÓDIGO](#)

Quando construímos classes que se limitam a ter atributos privados com os setters e getters normalmente dizemos que são classes que só carregam valor, por isso são comumente chamados de classes fantoches ou `Value Objects`.

Uma classe fantoche é a que não possui responsabilidade alguma, a não ser carregar um punhado de atributos! Definitivamente isso não é a Orientação a Objetos! Esse modelo embora usado em alguns momentos não deve ser prática comum ao desenvolver o domínio do nosso projeto com risco de se cair no Modelo Anêmico que é exatamente o que a `Conta` hoje é. Uma classe onde os dados e comportamentos/lógicas não estão juntos.

Voltando ao nosso exemplo da Conta, percebe-se que no mundo real as operações poderiam ser representadas com métodos como `saca()` e `deposita()` em vez de só termos `setSaldo()`:

```
class Conta{  
    private String titular;  
    private double saldo;  
  
    public void setTitular(String titular){  
        this.titular = titular;  
    }  
  
    public String getTitular(){  
        return titular;  
    }  
  
    public void saca(double valor){  
        if(valor > 0 && saldo >= valor){  
            saldo -= valor;  
        }  
    }  
  
    public void deposita(double valor){  
        if(valor>0){  
            saldo += valor;  
        }  
    }  
  
    public double getSaldo(){  
        return saldo;  
    }  
}
```

COPIAR CÓDIGO

Perceba que as lógicas de saque e depósito estão representados dentro da classe e além disso nosso `setSaldo()` deixa de fazer sentido para o usuário da Conta. A maneira de interagir com o saldo é sempre via uma das operações anteriores:

```
Conta conta = new Conta();
conta.setTitular("Fábio");
conta.deposita(100.0);

double valorSaque = 50.0;
conta.saca(valorSaque);

double valorDeposito = 70.0;
conta.deposita(valorDeposito)
```

[COPIAR CÓDIGO](#)

Muito melhor não é mesmo? Nada de duplicações de código por aí e muito menos outras classes controlando o estado da nossa Conta como tínhamos anteriormente.

Conclusão

Setters e Getters devem ser usados com cautela e nem todos os atributos privados precisam ter expostos esses dois métodos com riscos de cairmos em um `modelo anêmico` que tem os seus comportamentos controlados por outras classes.

01

Construtores

Transcrição

Chegamos ao ponto em que temos uma conta bem encapsulada, de forma que não é mais possível inserir um saldo negativo, pois ele está velado pelos métodos de manipulação da conta, como `saca()`, `transfere()` e `deposita()`.

Caso encontremos algum *bug*, como a inserção de um depósito negativo, basta adicionar um `if` no método `deposita()`. As soluções para possíveis problemas do nosso código estão bem localizadas, ou seja, alterando o código em um único ponto podemos realizar a manutenção do nosso sistema.

O próximo ponto que iremos nos voltar é se as nossas contas conseguem ser criadas de uma maneira que sempre possuam dados consistentes.

Quando pensamos em objetos consistentes, queremos dizer que seus atributos funcionam de acordo com as regras de negócios estipuladas por uma empresa, chefe ou algo do gênero.

Para testarmos a consistência dos objetos do nosso banco, criaremos mais uma classe chamada `TestaValores`. Sabemos que não é possível criar uma conta e deixá-la com um valor negativo, pois um saque não seria permitido neste caso e nem outros métodos de alteração de saldo seriam eficientes neste sentido. Entretanto podemos criar uma conta e atribuir ao número de `agencia` valores negativos.

```
public class TestaValores {  
    public static void main(String[] args) {  
        Conta conta = new Conta();  
        conta.setAgencia(-50);  
        conta.setNumero(-330);  
    }  
}
```

[COPIAR CÓDIGO](#)

Na classe `Conta`, precisaremos adicionar `if`s nos métodos `setAgencia` e `setNumero`, afirmando que caso seja postulado um valor menor ou igual a zero, ocorrerá uma mensagem de erro e a execução será interrompida.

```
public class Conta {  
    // atributos  
  
    // método deposita  
    // método saca  
    // método tranfere  
    // método pegaSaldo  
  
    public int getNumero() {...}  
  
    public void setNumero(int numero) {  
        if (numero <= 0) {  
            System.out.println("não pode valor <= 0");  
            return;  
        }  
        this.numero = numero;  
    }  
  
    public int getAgencia() {...}
```

```
public void setAgencia(int agencia) {  
    if (agencia <= 0) {  
        System.out.println("nao pode valor menor igual a 0");  
        return;  
    }  
    this.agencia = agencia;  
}  
  
public void setTitular(Cliente titular) {...}  
  
public Cliente getTitular() {...}  
}
```

COPIAR CÓDIGO

Com isso, ao tentarmos executar o programa da classe `TestaValores`, que está com números negativos para o atributo `agencia`, será impressa a mensagem `nao pode valor menor igual a 0`.

Porém, não estamos totalmente protegidos desses valores negativos. Percebam o problema caso escrevamos a seguinte linha de código:

```
public class TestaValores {  
    public static void main(String[] args) {  
        conta.setAgencia(-50);  
        conta.setNumero(-330);  
        System.out.println(conta.getAgencia());  
    }  
}
```

COPIAR CÓDIGO

Veremos como resultado da aplicação o valor impresso `0`. Portanto, não solucionamos o problema da numeração, porque estamos trabalhando com o val-

default, lembrem-se que no momento em que acionamos o `new` em um objeto, os atributos são zerados, e os atributos de tipo referência recebem valor `null`.

Muitas vezes encontramos objetos que nascem em um estado inconsistente com relação à regra de negócio. Existe uma forma de restringirmos dados: toda a vez que criamos um objeto somos obrigados a passar informações específicas, fazemos isso através de um **construtor**.

Os parênteses `()` fechados ao lado do objeto `Conta` estão invocando um construtor.

```
public class TestaValores {  
    public static void main(String[] args) {  
        Conta conta = new Conta();  
  
        // ...
```

[COPIAR CÓDIGO](#)

O construtor é um trecho de código caracterizado pela seguinte conformação:

```
public Conta() {  
}
```

[COPIAR CÓDIGO](#)

Não escrevemos esse código ao longo do curso. O Java automaticamente insere esse código, é o que chamamos de **construtor padrão**.

Iremos escrever esse código em nossa classe `Conta` e acionaremos o `sysout`.

```
public class Conta {  
    private double saldo;
```

```
private int agencia  
private int numero  
private Cliente titular;  
  
public Conta() {  
    System.out.println("estou criando uma conta");  
  
    // ...  
}
```

[COPIAR CÓDIGO](#)

Feito isso, executaremos a aplicação da classe `TestaValores`. Lembrando que o construtor está executando a terceira linha do código da classe.

```
public class TestaValores {  
    public static void main(String[] args) {  
        Conta conta = new Conta();  
  
        // ...
```

[COPIAR CÓDIGO](#)

Veremos que o resultado impresso será `estou criando uma conta`.

Os parênteses estão passando um construtor. Não se trata de um método, o construtor não possui um retorno `void`, `double`, ele é uma rotina de inicialização.

O construtor é executado apenas uma vez no momento em que construímos um objeto. Não há como executar duas vezes o construtor para um mesmo objeto.

O construtor nos oferece a possibilidade de inicializar alguns dados, como por exemplo, podemos estipular que o saldo inicial de uma conta vale `100` reais. Mas o

mais interessante é que o construtor pode receber parâmetros.

Para que o construtor seja invocado na abertura de uma nova conta, é necessário obrigatoriamente que seja passada a agência dessa conta e seu número. Feito isso, os atributos podem ser populados. Na execução iremos exibir o número da conta, assim conseguimos ver a atuação do construtor.

```
public class Conta {  
    private double saldo;  
    private int agencia;  
    private int numero;  
    private Cliente titular;  
  
    public Conta( int agencia, int numero ) {  
        this.agencia = agencia;  
        this.numero = numero;  
        System.out.println("estou criando uma conta" + this.numero)  
    }  
  
    // ...  
}
```

[COPIAR CÓDIGO](#)

O construtor padrão que o Java estipula caso não tenhamos escrito nenhum outro construtor, deixa de existir.

Através do construtor podemos definir restrições e exigir informações específicas do objeto. O que será exigido pelo construtor varia de acordo com as regras de negócio, no caso do nosso banco é interessante que o saldo inicie com zero, por exemplo.

Definimos alguns parâmetros para o construtor, então, para cada objeto criado, precisaremos primeiramente comunicar a `agencia` (1337) e o `número` da conta (24226).

```
public class TestaValores {  
    public static void main(String[] args) {  
        Conta conta = new Conta(1337, 24226);  
  
        // ...
```

[COPIAR CÓDIGO](#)

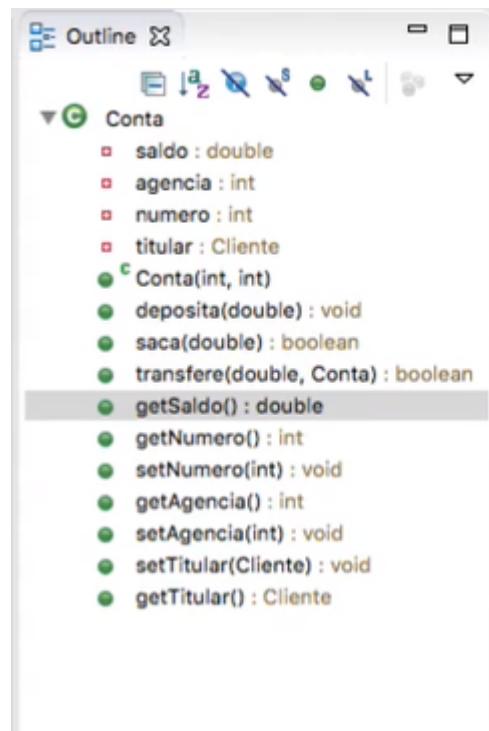
Ao executarmos a aplicação, o resultado impresso será `estou criando uma conta 24226`.

Podemos adicionar `if`s no nosso código para gerar as condições ideais de criação do objeto de acordo com as regras de negócio estipuladas. Por exemplo, que o número de `agencia` deve ser maior ou igual a zero.

Com a presença do construtor e os parâmetros que a ele foram passados, não é mais necessário que haja no nosso código os métodos `setAgencia()` e `setNumero()`. Se for estipulado como regra de negócio que uma conta sempre terá o mesmo número e a mesma agência, não há necessidade de evocar métodos para alterá-la.

Trata-se de um **atributo imutável**, e há vários casos em que essa é uma opção interessante.

Uma dica de navegação no Eclipse: a nossa classe `Conta` está grande. Para facilitar o acesso ao conteúdo da classe, no cabeçalho do Eclipse, selecionaremos a opção "Window > Show View > Outline" (atälho "Ctrl + O"). Ao lado esquerdo da tela, surgirá um resumo de todos os métodos e atributos presentes na classe, podemos clicar em cada um deles para termos um acesso rápido.



04

Static

Transcrição

Mais um desafio: precisaremos saber quantas contas foram abertas no sistema. Na linguagem Java, saberemos quantas contas foram instanciadas.

Na classe `TestaValores`, criaremos uma variável chamada `total`, que começa com o valor `0`.

Lembre-se que variável local precisa ser zerada.

Feito isso, escreveremos que a cada vez que surgir uma nova conta através do `new`, isso seja contabilizado na variável `total++`.

```
public class TestaValores {  
    public static void main(String[] args) {  
        int total = 0;  
        Conta conta = new Conta(1337, 24226);  
        total++;  
  
        System.out.println(conta.getAgencia());  
  
        conta.setAgencia(1232123);  
    }  
}
```

[COPIAR CÓDIGO](#)

Essa saída é funcional, mas apresenta problemas de ordem prática, por exemplo, toda a vez que uma nova conta for aberta, o desenvolvedor deve escrever a variável `total++`.

Uma forma melhor de contabilizar as contas que foram criadas no nosso banco é acionar o **construtor**. Na classe `Conta`, adicionaremos no construtor a requisição da variável `total++`.

```
public class Conta {  
    private double saldo;  
    private int agencia;  
    private int numero;  
    private Cliente titular;  
  
    public Conta(int agencia, int numero) {  
        total++;  
        this.agencia = agencia;  
        this.numero = numero;  
        System.out.println("estou criando uma conta " + this.numero  
    }  
}
```

COPIAR CÓDIGO

Para que o nosso código compile, iremos declarar `total` como um atributo do tipo `int` privado. Feito isso, iremos imprimir o total de contas adicionando o texto o total de contas é mais a variável.

```
public class Conta {  
    private double saldo;  
    private int agencia;  
    private int numero;  
    private Cliente titular;
```

```
private int total;

public Conta(int agencia, int numero) {
    total++;
    System.out.println("o total de contas é " + total);
    this.agencia = agencia;
    this.numero = numero;
    System.out.println("estou criando uma conta " + this.numero
}
```

[COPIAR CÓDIGO](#)

A variável `total++` que escrevemos na classe `TestaValores` não tem qualquer ligação com o atributo `total` que criamos. Portanto, podemos excluí-la, já que descobrimos uma forma mais prática de contabilizar o total de contas.

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

        Conta conta = new Conta(1337, 24226);
        System.out.println(conta.getAgencia());
        conta.setAgencia(1232123);
    }
}
```

[COPIAR CÓDIGO](#)

Ao executarmos nosso programa, veremos que o valor impresso será `o total de contas é 1`. O resultado, portanto, está correto. Adicionaremos mais duas novas contas para testarmos se o programa está operando corretamente.

```
public class TestaValores {
    public static void main(String[] args) {
        Conta conta = new Conta(1337, 24226);
```

```
System.out.println(conta.getAgencia());  
conta.setAgencia(1232123);  
  
Conta conta2 = new Conta(1337, 16549);  
Conta conta3 = new Conta(2112, 14660);  
}  
}
```

[COPIAR CÓDIGO](#)

Veremos que o resultado do programa é o total de contas é 1. Ou seja, não houve a atualização quanto ao número de contas. Percebemos que houve um erro na criação do atributo total na classe Conta. É como se cada objeto conta tivesse um saldo, agencia, numero, titular e total.

O que queremos é que total fosse uma variável que não ficasse em cada instância, mas em algum lugar da classe Conta, algo como um atributo compartilhado e não de um objeto especificamente.

Para isso, existe a palavra-chave static. O static faz com que o atributo seja da classe, e não mais do objeto. Com isso, todo o objeto conta possui acesso a um único total.

Ao lado esquerdo da variável total++ podemos adicionar a classe Conta. Afinal, não estamos mais fazendo referência à uma conta - como o this fazia - e sim à classe Conta.

```
public class Conta {  
    private double saldo;  
    private int agencia;  
    private int numero;  
    private Cliente titular;  
    private static int total;
```

```
public Conta(int agencia, int numero) {  
    Conta.total++;  
    System.out.println("o total de contas é " + Conta.total);  
    this.agencia = agencia;  
    this.numero = numero;  
    System.out.println("estou criando uma conta " + this.numero  
}
```

[COPIAR CÓDIGO](#)

Ao executarmos a aplicação, veremos que o valor impresso será o total de contas é 3.

Caso retirássemos os `sysout` da classe `Conta`, o programa não imprimiria mais o total de contas.

```
public Conta(int agencia,int numero) {  
    Conta.total++;  
    //System.out.println("o total de contas é " + Conta.total);  
    this.agencia = agencia;  
    this.numero = numero;  
    //System.out.println("estou criando uma conta " + this.numero  
}
```

[COPIAR CÓDIGO](#)

Tendo isso em vista, poderíamos imprimir o total de contas na classe `TestaValor` fazendo um `sysout` no atributo `total`. Isso funcionaria se o atributo não fosse privado. A questão é que não há mais necessidade de marcar `conta2` ou `conta3`, pois o `total` é compartilhado entre as instâncias.

```
public class TestaValores {  
    public static void main(String[] args) {  
        Conta conta = new Conta(1337, 24226);  
        System.out.println(conta.getAgencia());  
        conta.setAgencia(1232123);  
  
        Conta conta2 = new Conta(1337, 16549);  
        Conta conta3 = new Conta(2112, 14660);  
  
        System.out.println(Conta.total);  
    }  
}
```

[COPIAR CÓDIGO](#)

Como já vimos, é interessante que este atributo seja privado para preservarmos características essenciais segundo a regra de negócios que rege o banco. Portanto, deveremos criar um getter para o atributo `total` na classe `Conta`.

```
public int getTotal() {  
    return Conta.total;  
}
```

[COPIAR CÓDIGO](#)

Feito isso, acionaremos o getter na classe `TestaValores` usando o `System.out`. Não precisamos especificar `conta1` ou `conta2`, porque estamos nos referindo à classe `Conta`.

```
public class TestaValores {  
    public static void main(String[] args) {  
        Conta conta = new Conta(1337, 24226);  
        System.out.println(conta.getAgencia());  
        conta.setAgencia(1232123);
```

```
    Conta conta2 = new Conta(1337, 16549);
    Conta conta3 = new Conta(2112, 14660);

    System.out.println(Conta.getTotal());
}

}
```

[COPIAR CÓDIGO](#)

Veremos que o código não está sendo compilado, vamos entender o porquê: todos os métodos declarados eram da instância `conta`, ou seja, de uma conta específica. Neste caso, estamos nos comunicando com um atributo `total`. Para isso, precisaremos declarar que o método é `static`.

```
public static int getTotal() {
    return Conta.total;
}
```

[COPIAR CÓDIGO](#)

O método `static` possui usos muito interessantes, mas possui suas limitações. Embora não seja do interesse do nosso projeto, não poderíamos acessar um atributo de instância via `this`, como `saldo`.

```
public static int getTotal() {
    System.out.println(this.saldo);
    return Conta.total;
```

[COPIAR CÓDIGO](#)

Os métodos estáticos acessam apenas atributos estáticos.

 07

Para saber mais: reaproveitamento entre construtores

Nesse capítulo o nosso aprendizado foi focado nos construtores. Eles são elaborados visando que os objetos tenham seus atributos inicializados na própria construção. Essa estratégia evita estados inconsistentes no nosso objeto. Veja essa classe:

```
public class Carro{  
    private int ano;  
    private String modelo;  
    private double preco;  
  
    //getters e setters omitidos  
  
}
```

[COPIAR CÓDIGO](#)

Como já se sabe, quando o construtor não está declarado na classe usa-se o padrão, que não recebe parâmetro algum. Logo, uma utilização da classe poderia ser como a seguir:

```
Carro carro = new Carro();  
carro.setAno(2013);  
carro.setPreco(35000.0);
```

[COPIAR CÓDIGO](#)

Ficou faltando uma informação preciosa! Qual o modelo dele? Para evitar esse tipo de problema devemos exigir os dados que fazem sentido o Carro ter logo ná:

criação. Algo como:

```
public class Carro{  
    private int ano;  
    private String modelo;  
    private double preco;  
  
    public Carro(int ano, String modelo, double preco){  
        this.ano = ano;  
        this.modelo = modelo;  
        this.preco = preco;  
    }  
  
    //getters e setters omitidos  
  
}
```

[COPIAR CÓDIGO](#)

Agora a utilização exige a presença dos 3 parâmetros definidos.

```
Carro carro = new Carro(2013, "Gol", 35000.0);
```

[COPIAR CÓDIGO](#)

Tudo funciona bem! Até que um dia é pedido que o nosso sistema aceite a criação com a passagem somente do modelo e valor. Nessa situação deve-se encarar o ano como sendo 2017. Uma solução seria:

```
public class Carro{  
    private int ano;  
    private String modelo;  
    private double preco;
```

```
public Carro(int ano, String modelo, double preco){  
    this.ano = ano;  
    this.modelo = modelo;  
    this.preco = preco;  
}  
  
//Novo construtor AQUI!  
public Carro(String modelo, double preco){  
    this.ano = 2017;  
    this.modelo = modelo;  
    this.preco = preco;  
}  
  
//getters e setters omitidos  
}
```

[COPIAR CÓDIGO](#)

E dessa forma pode-se construir carros com qualquer um dos dois construtores:

```
Carro carro = new Carro(2013, "Gol", 35000.0);  
Carro outroCarro = new Carro("Civic", 95000.0);
```

[COPIAR CÓDIGO](#)

Só que na empresa onde esse sistema está sendo codificado existe uma equipe de testes que verificou que o nosso sistema permite a criação de um Carro com datas anteriores ao primeiro automóvel que chegou ao Brasil, um Peugeot trazido por Santos Dumont em 1891. (Alura também é história!) Além de também permitir que o modelo não seja passado(null) e o preço inválido.

O desenvolvedor logo tratou de implementar essa regra em um dos construtores

```
public class Carro{  
    private int ano;  
    private String modelo;  
    private double preco;  
  
    public Carro(int ano, String modelo, double preco){  
        if(ano >= 1891){  
            this.ano = ano;  
        }else{  
            System.out.println("O ano informado está inválido. Por isso usaremos o ano de 2017.");  
            this.ano = 2017;  
        }  
  
        if( modelo != null){  
            this.modelo = modelo;  
        }else{  
            System.out.println("O modelo não foi informado. Por isso usaremos o modelo Gol.");  
            this.modelo = "Gol";  
        }  
  
        if(preco > 0){  
            this.preco = preco;  
        }else{  
            System.out.println("O preço não é válido. Por isso usaremos o preço de 40000.0.");  
            this.preco = 40000.0;  
        }  
    }  
    //....  
}
```

COPiar CÓDIGO

Perceba que como temos dois construtores a regra também deveria valer para o outro:

```
public class Carro{  
    private int ano;  
    private String modelo;  
    private double preco;  
  
    public Carro(int ano, String modelo, double preco){  
        if(ano >= 1891){  
            this.ano = ano;  
        }else{  
            System.out.println("O ano informado está inválido. Por isso usaremos o ano 2017.");  
            this.ano = 2017;  
        }  
  
        if( modelo != null){  
            this.modelo = modelo;  
        }else{  
            System.out.println("O modelo não foi informado. Por isso usaremos o modelo Gol.");  
            this.modelo = "Gol";  
        }  
  
        if(preco > 0){  
            this.preco = preco;  
        }else{  
            System.out.println("O preço não é válido. Por isso usaremos o preço 40000.0.");  
            this.preco = 40000.0;  
        }  
    }  
  
    //Novo construtor AQUI!  
    public Carro(String modelo, double preco){  
    }
```

```
this.ano = 2017;
if( modelo != null){
    this.modelo = modelo;
}else{
    System.out.println("O modelo não foi informado. Por isso usaremos o Gol");
    this.modelo = "Gol";
}

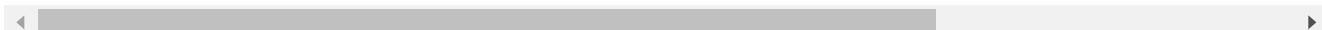
if(preco > 0){
    this.preco = preco;
}else{
    System.out.println("O preço não é válido. Por isso usaremos o valor padrão de 40000.0");
}

}

//getters e setters omitidos
```

}

COPIAR CÓDIGO



Funcionou mas o código está duplicado e nossa classe começa a cheirar mal! Códigos duplicados exigem manutenção em dobro no futuro e em grande parte das vezes um futuro nem tão distante. Seria ótimo se fosse possível reaproveitar a lógica de validação do primeiro construtor declarado não é mesmo?

Reaproveitariammos todo ele e qualquer mudança também traria o impacto direto. No Java podemos chamar a implementação de um construtor através de outro usando simplesmente `this()` com os parâmetros exigidos pelo construtor.

Observe como ficaria o segundo construtor da nossa classe:

```
public Carro(String modelo, double preco){  
    //chamando o construtor que recebe int, String e double. Noss:  
    this(2017, modelo, preco);  
}
```

[COPIAR CÓDIGO](#)

Muito mais simples de manter não é mesmo? Nossa classe, Carro, ficaria portanto assim:

```
public class Carro{  
    private int ano;  
    private String modelo;  
    private double preco;  
  
    public Carro(int ano, String modelo, double preco){  
        if(ano >= 1891){  
            this.ano = ano;  
        }else{  
            System.out.println("O ano informado está inválido. Po:  
            this.ano = 2017;  
        }  
  
        if( modelo != null){  
            this.modelo = modelo;  
        }else{  
            System.out.println("O modelo não foi informado. Por i:  
            this.modelo = "Gol";  
        }  
  
        if(preco > 0){  
            this.preco = preco;  
        }else{  
            System.out.println("O preço não é válido. Por isso us:  
        }  
    }  
}
```

```
this.preco = 40000.0;  
}  
}  
  
//Novo construtor AQUI!  
public Carro(String modelo, double preco){  
    this(2017, modelo, preco);  
}  
  
//getters e setters omitidos  
  
}
```

COPIAR CÓDIGO

Conclusão

No Java é possível fazer a chamada de um construtor dentro de outro e faz-se isso para evitar duplicações de códigos e regras. Afinal uma regra aplicada em um construtor normalmente será a mesma para o outro caso. Para isso usa-se o `this()` passando os parâmetros correspondentes ao construtor que se queira chamar.