

****Classe, Métodos, Atributos****

**domain driven design**

**Thiago Yamamoto**

Versão 1

HISTÓRICO DE REVISÕES

|  |  |  |  |
| --- | --- | --- | --- |
| Versão | Data | Responsável | Descrição |
| 1.0 |  | Thiago Yamamoto | Versão Inicial do documento |
| 1.1 | 26/03/2015 | Marta Hiromi | Ajustes para Revisão 1 |
| 1.2 | 23/4/2015 | Márcia Dias | Revisão 1 |
| 1.3 | 29/11/2016 | Fernanda Meireles | Inserção de imagens |
| 1.4 | 02/01/2017 | Márcia Dias | Revisão 2 |
|  |  |  |  |
|  |  |  |  |

FICHA CATALOGRÁFICA   
**[NÃO PREENCHER - PARA USO DO DEPTO DE EAD E BIBLIOTECA]**

A000a Sobrenome, Nome

Título [livro eletrônico] / Nome Sobrenome. -- São Paulo : Fiap, 2016.

x MB ; ePUB

Bibliografia.

ISBN 000-00-00000-00-0

Categoria. 2. Subcategoria. S., Nome. II. Título.

CDU 000.000.00

RESUMO

Nos capítulos anteriores, vimos os conceitos de orientação a objetos e os conceitos básicos na programação Java. Agora é hora de entender como implementar os conceitos aprendidos de orientação a objetos na programação Java.

Palavras-chave: Java. Atributos. Métodos. Classe.

LISTA DE Figuras

[Figura 1 – Exemplo de atributos 12](#_Toc512508417)

[Figura 2 – Método **recuperarSaldo** 14](#_Toc512508418)

[Figura 3 – Exemplo de parâmetros de métodos 15](#_Toc512508419)

[Figura 4 – Sobrecarga de métodos 15](#_Toc512508420)

[Figura 5 – Palavra **this** para referenciar ao atributo da classe 16](#_Toc512508421)

[Figura 6 – Exemplo construtor 17](#_Toc512508422)

[Figura 9 – Classe Conta com três construtores 19](#_Toc512508423)

[Figura 10 – Utilizando a palavra this para chamar um construtor da própria classe 19](#_Toc512508424)

[Figura 11 – Classe Conta 20](#_Toc512508425)

[Figura 12 – Classe de teste 21](#_Toc512508426)

[Figura 13 – Instanciando uma classe Conta 21](#_Toc512508427)

[Figura 14 – Exemplo de utilização de objetos 22](#_Toc512508428)

[Figura 15 – Resultado da execução da classe de teste 22](#_Toc512508429)

[Figura 16 – Visão de variáveis que armazenam referências aos objetos Java 23](#_Toc512508430)

[Figura 17 – Projeto exemplo 25](#_Toc512508431)

[Figura 18 – Classe conta e modificadores de acesso 25](#_Toc512508432)

[Figura 19 – Criando um novo pacote 26](#_Toc512508433)

[Figura 20 – Novo pacote para a classe Teste 26](#_Toc512508434)

[Figura 21 – Classe de Teste 27](#_Toc512508435)

[Figura 22 – Visão geral da API Java 28](#_Toc512508436)

[Figura 23 – Exemplos de comentários no Java 30](#_Toc512508437)

[Figura 24 – Exemplo de comentário de documentação 31](#_Toc512508438)

[Figura 25 – Classe Conta com comentários de documentação 32](#_Toc512508439)

[Figura 26 – Gerando documentação – Parte 1 33](#_Toc512508440)

[Figura 27 – Gerando documentação – Parte 2 34](#_Toc512508441)

[Figura 28 – Documentação da Classe Conta 34](#_Toc512508442)

[Figura 29 – Atributo idade encapsulado 36](#_Toc512508443)

[Figura 30 – Atributo do tipo boolean encapsulado 37](#_Toc512508444)

[Figura 31 – Método encapsulado 37](#_Toc512508445)

[Figura 32 – Classe JavaBean Conta 39](#_Toc512508446)

[Figura 33 – Classe de Teste 39](#_Toc512508447)

LISTA DE QUADROS

[Quadro 1 –Visão geral sobre os modificadores 24](#_Toc512508448)

[Quadro 2 – Tags de documentação 31](#_Toc512508449)

LISTA DE TABELAS

[Tabela 1 – Valores padrões para variáveis de instância. 13](#_Toc512508450)

Sumário

[3 Classe, métodos, atributos 10](#_Toc512508495)

[3.1 Classe 10](#_Toc512508496)

[3.2 Atributos 11](#_Toc512508497)

[3.3 Métodos 13](#_Toc512508498)

[3.3.1 Sobrecarga de métodos 15](#_Toc512508499)

[3.3.2 Construtor 16](#_Toc512508500)

[3.4 Trabalhando com objetos 20](#_Toc512508501)

[3.5 Modificadores de acesso 24](#_Toc512508502)

[3.6 Java API specification 28](#_Toc512508503)

[3.7 Comentários 29](#_Toc512508504)

[3.8 Javadoc 30](#_Toc512508505)

[3.9 Java bean e encapsulamento 35](#_Toc512508506)

[REFERÊNCIAS 40](#_Toc512508507)

# 3 Classe, métodos, atributos

## 3.1 Classe

Uma classe possui o modelo ou estrutura a partir do qual os objetos serão criados. Pense em uma classe que precisa implementar uma conta bancária. Quais são as informações que a conta precisa armazenar? E quais são as ações que ela deve realizar?

As informações relevantes para uma conta bancária podem ser o saldo, número da conta, agência, tipo de conta etc. E as ações ou comportamentos importantes de uma classe Conta são: sacar, depositar, verificar o saldo etc.

Dessa forma, podemos desenvolver uma classe *Conta* que contenha essas informações e comportamentos. Porém, essa classe é somente o modelo para o conceito de Conta Bancária dentro do nosso sistema. Assim como em um Banco Financeiro do “mundo real”, antes de guardar dinheiro na conta e depositar ou retirar, é preciso ir ao Banco para abrir uma Conta. No mundo orientado a objetos, primeiro precisamos criar um objeto utilizando a classe Conta para depois utilizá-la. Esse processo de criação de um objeto a partir de uma classe é chamado de instanciação.

Em resumo, um objeto é uma instância de uma Classe. Para instanciar uma classe utilizamos o operador **new:**

**new** Conta();

No exemplo acima foi criado um objeto do tipo Conta. Porém, precisamos armazenar esse objeto em alguma variável para utilizá-lo posteriormente. Para isso, podemos declarar uma variável do tipo da Classe (Conta) e atribuir o objeto a variável com o operador **=.**

Conta cc = **new** Conta();

Conta poupanca = **new** Conta();

Dessa forma, a variável **cc** e **poupança** armazenam a referência de seus respectivos objetos. E assim como um banco pode possuir várias contas, no programa, podemos instanciar várias classes do mesmo tipo, neste caso o tipo Conta.

As classes Java são definidas em arquivos separados com a extensão **.java** e o nome do arquivo deve ser igual ao nome da Classe. Por convenção, o nome segue o padrão UpperCamelCase, no qual as palavras sempre se iniciam com a letra em maiúscula. Por exemplo: Conta, ContaCorrente, ContaPoupanca.

Para definir uma classe em Java, utilizamos a palavra reservada **class:**

[modificador] class [NomeDaClasse] {

}

Exemplo:

public class Conta{

}

Neste capítulo, vamos falar sobre os modificadores. Não se preocupe!

## 3.2 Atributos

Uma classe pode conter nenhum ou vários atributos. Depois de instanciar a classe, os atributos serão utilizados para armazenar informações do objeto. Essas informações diferenciam um objeto do outro.

Pense novamente na classe *Conta*, já identificamos alguns atributos necessários, como: Saldo, Número da Conta, Agência, Tipo da Conta. Em um programa para gerenciar as contas bancárias, serão necessários vários objetos do tipo Conta, cada um deles com as informações de sua própria Conta.

Em uma classe, os atributos são definidos por variáveis, que podem ser do tipo primitivo (como visto no capítulo anterior) ou do tipo de referência, no qual a variável armazena uma referência ao objeto. No nosso exemplo, a classe conta pode conter uma variável para armazenar a referência de um objeto Cliente. Assim, a classe Conta possui um Cliente.

Por convenção, os nomes dos atributos seguem o lowerCamelCase, cuja primeira letra é minúscula e as demais palavras começam com a letra maiúscula. Pela boa prática, devemos utilizar substantivos e nomes bem definidos para os atributos, como por exemplo: saldo, dataNascimento, email etc. Nomes poucos sugestivos devem ser evitados: x, y, abc entre outros.

As variáveis que definem um atributo em uma classe são chamadas de variáveis de instância, pois só é possível armazenar informação nessa variável após a instanciação da Classe, ou seja, no objeto.

Declarar uma variável de instância segue a mesma sintaxe das variáveis locais, visto no capítulo anterior.

A Figura 1 traz alguns exemplos de atributos:

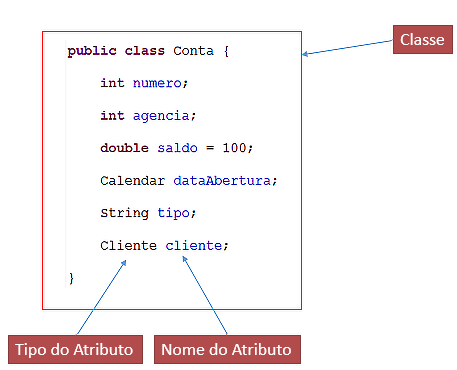


Figura 1 – Exemplo de atributos

Fonte: FIAP (2015)

Diferentemente das variáveis locais, as variáveis de instância recebem valores- padrão quando não atribuímos valores à sua declaração.

Por exemplo:

class Conta {

double saldo;

}

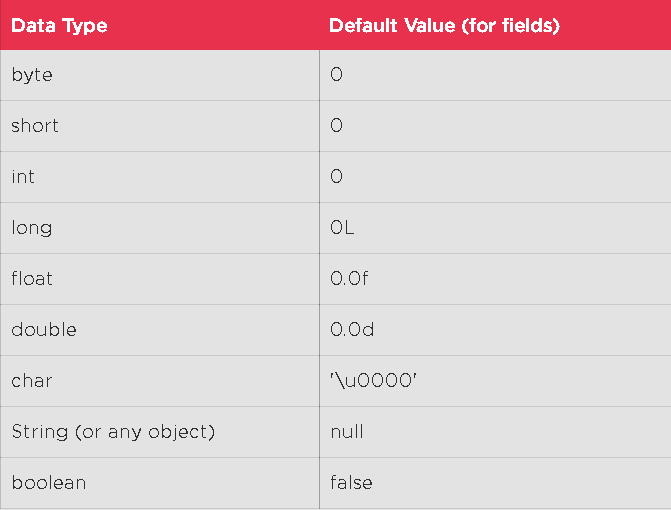
Neste caso, a variável saldo recebe o valor padrão 0. Podemos também, atribuir um valor na hora de declarar um atributo, assim o valor padrão não será utilizado:

class Conta {

double saldo = 100;

}

Tabela 1 – Valores padrões para variáveis de instância.



Fonte: FIAP (2015)

## 3.3 Métodos

Os métodos definem os comportamentos que o objeto possui. O comportamento pode ser entendido como uma ação ou serviço, por exemplo, um objeto do tipo *Conta* possui comportamentos como recuperar o Saldo, depositar e retirar dinheiro da conta.

Dessa forma, podemos definir um método como um comportamento específico, residente no objeto, que define como ele deve agir quando exigido, definindo assim as habilidades do objeto.

Por convenção, o nome dos métodos, assim como os seus atributos, devem sempre ser escritos em lowerCamelCase, e geralmente utilizamos verbos para os nomes. Exemplos: exibirSaldo, depositar, calcularTaxa, pois os métodos executam ações.

A sintaxe básica para declarar um método é:

<modificador> <tipo de retorno> <nomeDoMetodo>(<[lista de argumentos]>){

[instrucoes];

}

A Figura 2 exemplifica o método **recuperarSaldo**

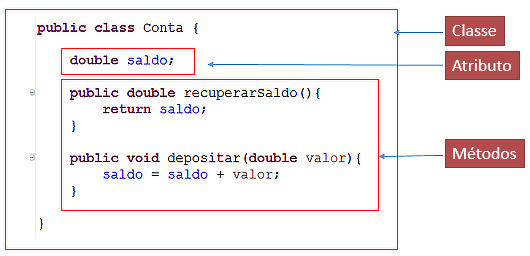


Figura 2 – Método **recuperarSaldo**

Fonte: FIAP (2015)

Precisamos definir o tipo de retorno que o método deve devolver. No exemplo acima (Figura 2), o método **recuperarSaldo** retorna um valor do tipo **double**. A instrução **return** é utilizada para retornar o valor; o método retorna o valor do atributo saldo.

Caso o método não precise retornar nenhum valor**,** podemos definir o retorno como **void.** No exemplo acima, o método **depositar** não retorna nenhum valor.

Os métodos podem receber valores, como é o caso do método **depositar.** Os parâmetros dos métodos são declarados pela **[lista de argumentos]** que são um conjunto de declarações de variáveis que são separados por vírgulas e definidas dentro de parênteses. Esses parâmetros se tornam variáveis locais no método, recebendo seus valores quando o método for chamado. Como exemplificado na Figura 3.

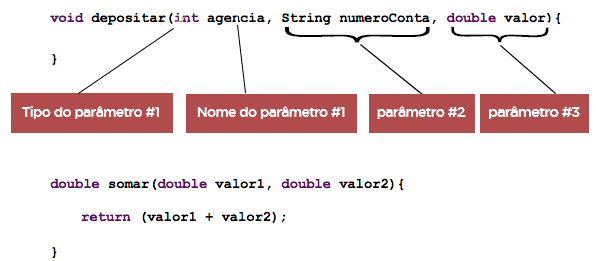


Figura 3 – Exemplo de parâmetros de métodos

Fonte: FIAP (2015)

## 3.3.1 Sobrecarga de métodos

Um recurso usual em Programação Orientada a Objetos é o uso de sobrecarga de métodos. Sobrecarregar um método significa prover mais de uma versão de um mesmo método. As versões devem, necessariamente, conter parâmetros diferentes, seja no tipo ou número desses parâmetros. O tipo de retorno não é relevante.

Dessa forma, duas características diferenciam os métodos com o mesmo nome: o número de parâmetros e o tipo deles. Essas características fazem parte da assinatura de um método. O uso de vários métodos com o mesmo nome e assinaturas diferentes é chamado de sobrecarga de métodos.

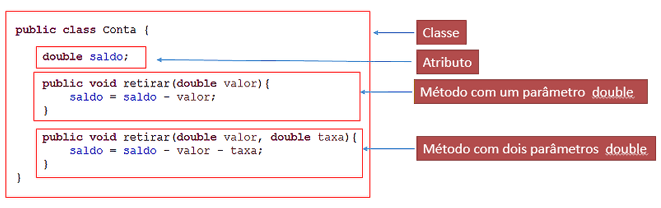


Figura 4 – Sobrecarga de métodos

Fonte: FIAP (2015)

No exemplo acima (Figura 4), a classe Conta possui dois métodos com o nome retirar, com assinaturas diferentes. Um que recebe um valor para retirada e outro que recebe um valor de retirada e o valor da taxa de retirada.

A sobrecarga de métodos torna possível que os métodos se comportem de modo diferente, dependendo dos argumentos que recebem. Quando nós chamamos um método em um objeto, o Java verifica o nome do método e os parâmetros enviados para escolher o melhor método a ser invocado.

A palavra reservada **this** faz referência ao próprio objeto. É por meio dela que é possível acessar atributos, métodos e construtores do objeto em questão.

Quando houver duas variáveis com o mesmo nome, uma sendo uma variável de instância (atributo da classe) e outra pertencente ao método, utilizaremos a palavra **this** para referenciar o atributo da classe, como mostra a Figura 5.

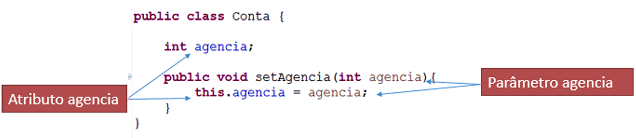


Figura 5 – Palavra **this** para referenciar ao atributo da classe

Fonte: FIAP (2015)

No decorrer do curso, iremos utilizar a palavra **this** para as outras situações citadas acima.

## 3.3.2 Construtor

Podemos construir métodos especiais, chamados de construtores, que são executados automaticamente quando os objetos dessa classe são criados. Esses métodos auxiliam na construção do objeto, podendo ser utilizado para inicializar os atributos com valores-padrão ou valores informados e chamar métodos em outros objetos.

Um método construtor é chamado quando o objeto é construído, ou seja, é invocado quando utilizamos a instrução **new** para criar uma instância da classe.

No momento em que criamos uma instância da classe, três passos são executados pelo Java:

* Alocar memória para o Objeto.
* Inicializar os atributos com os valores iniciais ou padrões.
* Chamar o método Construtor da classe.

Os construtores se parecem muito com métodos comuns, mas têm três diferenças básicas:

* Têm o mesmo nome da Classe.
* **Não** têm tipo de retorno (Nem mesmo void).
* **Não** podem retornar valor no método usando a instrução return.

Toda classe tem pelo menos um construtor. Quando o construtor não é especificado, a linguagem Java fornece um construtor *default*(padrão) – vazio – que não recebe parâmetros. Mas se for declarado algum construtor na classe, a linguagem Java não fornecerá mais o construtor padrão.

A Figura 6 mostra o exemplo de um construtor.

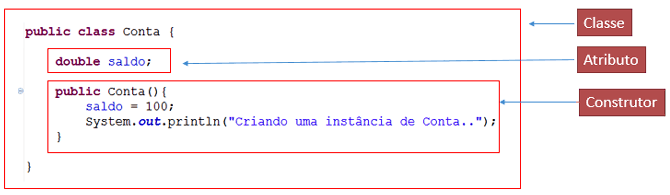
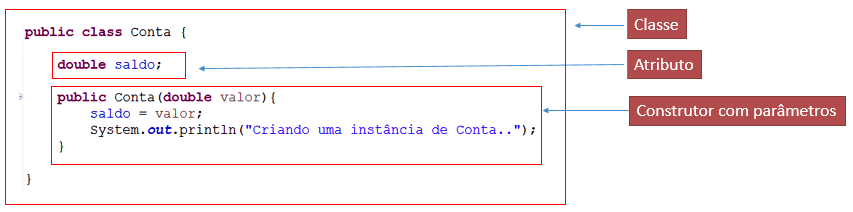


Figura 6 – Exemplo construtor

Fonte: FIAP (2015)

Dessa forma, quando uma instância de Conta for criada (**new Conta();**), o atributo saldo será inicializado com o valor 100 e será impresso no Console a frase “Criando uma instância de Conta.”.

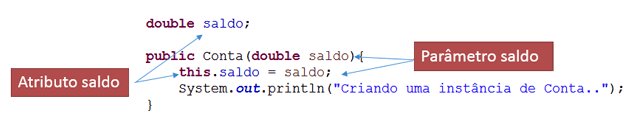
Podemos adicionar um construtor que recebe parâmetros (Figura 7):



Neste exemplo, o construtor padrão (sem parâmetros) não será fornecido pelo Java. O único construtor que a classe Conta tem é o que recebe um valor double como parâmetro. Esse valor é utilizado para inicializar o valor do atributo saldo.

Exemplo de utilização: **new Conta(100);**

Da mesma forma que os métodos, a instrução **this** foi utilizada para acessar as variáveis de instância de um objeto atual:



No exemplo acima, o construtor recebe um parâmetro chamado saldo e atribui o seu valor ao atributo saldo. Para diferenciar o atributo do parâmetro, utiliza-se a instrução **this.**

Assim como os métodos, uma classe pode ter vários construtores com diferentes tipos e quantidades de argumentos. Isso é chamado de sobrecarga de métodos construtores. Dessa forma, uma classe pode ser instanciada com qualquer construtor.

O exemplo a seguir (Figura 9) apresenta uma classe Conta com três construtores:

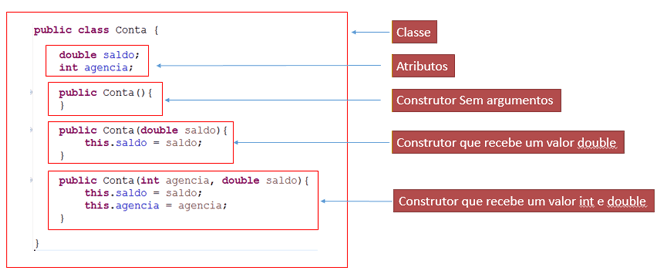


Figura 9 – Classe Conta com três construtores

Fonte: FIAP (2015)

Dessa forma, tempos três opções para instanciar a classe Conta:

* new Conta().
* new Conta(100).
* new Conta(10,100).

Outro uso para a palavra reservada **this** é na chamada de um construtor por outro construtor da própria classe, como exemplifica a Figura 10.

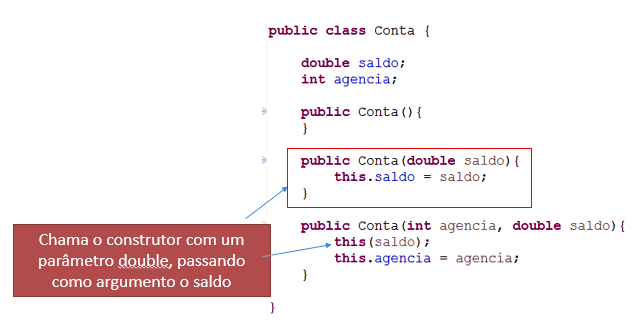


Figura 10 – Utilizando a palavra this para chamar um construtor da própria classe

Fonte: FIAP (2015)

## 3.4 Trabalhando com objetos

Para consolidar os conceitos, vamos criar uma classe *Conta* com os atributos saldo, agência e número. Vamos desenvolver também os métodos de retirar, depositar e verificarSaldo. Para facilitar a construção dos objetos dessa classe, vamos implementar dois construtores: um construtor padrão e outro que recebe três argumentos: o saldo, a agência e o número.

Para isso, crie uma nova Classe chamada Conta dentro do pacote **br.com.fiap.banco** e depois implemente os atributos, métodos e construtores.

A figura 11 mostra o resultado final:

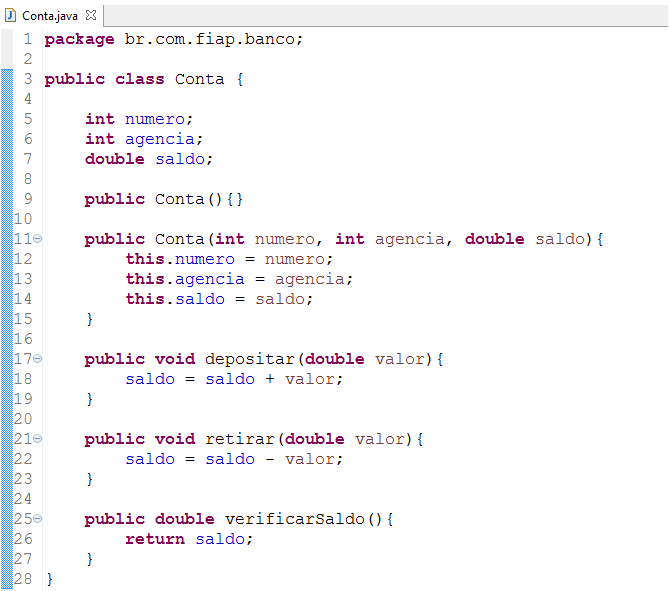


Figura 11 – Classe Conta

Fonte: FIAP (2015)

Agora, vamos criar uma classe de Teste (Figura 12) com o método **main** para criar instâncias da classe Conta. Para isso, crie uma nova classe chamada Teste, no pacote **br.com.fiap.banco,** e implemente o método main:

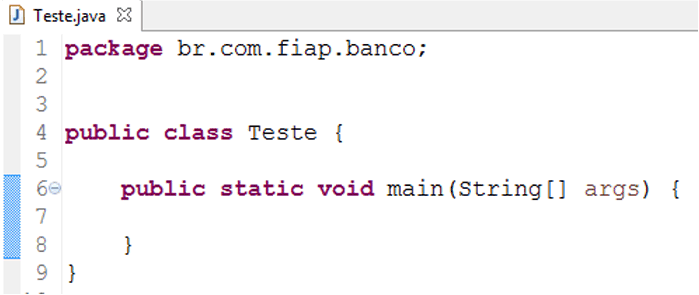


Figura 12 – Classe de teste

Fonte: FIAP (2015)

Dentro do método main, vamos instanciar uma Classe conta (Figura 13) e armazenar a referência desse objeto em uma variável.

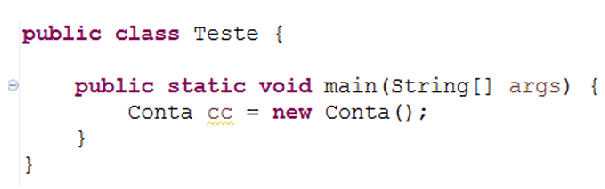


Figura 13 – Instanciando uma classe Conta

Fonte: FIAP (2015)

A variável **cc** tem uma referência ao objeto Conta. É por meio dela que podemos acessar os atributos e métodos do objeto.

Utilizando o operador ponto (.), podemos acessar as variáveis de instância e métodos do objeto. Por exemplo, para acessar o atributo **saldo** do objeto conta que está referenciada pela variável **cc**, podemos utilizar o seguinte código:

double valor = cc.saldo;

System.out.println(cc.saldo);

Na primeira linha, recuperamos o valor do atributo saldo do objeto conta e atribuindo a variável valor. Na segunda linha, imprimimos o valor do saldo no console.

Para atribuir um valor ao atributo do objeto, utilizamos o operador de atribuição:

cc.saldo = 1000;

Chamar um método de um objeto é semelhante ao acesso de um atributo: é utilizada a notação de ponto:

cc.depositar(100);

cc.verificarSaldo();

A variável que referencia o objeto (**cc**) fica do lado esquerdo e o nome do método e seus argumentos ficam do lado direito do ponto. Dentro dos parênteses são informados os argumentos do método. Caso o método tenha mais de um argumento, esses são separados por vírgulas. Métodos que não recebem parâmetros não precisam receber nenhum valor, como é o exemplo do método **verificarSaldo().**

Vamos criar mais uma instância da classe conta, atribuir alguns valores aos seus atributos e chamar os métodos.

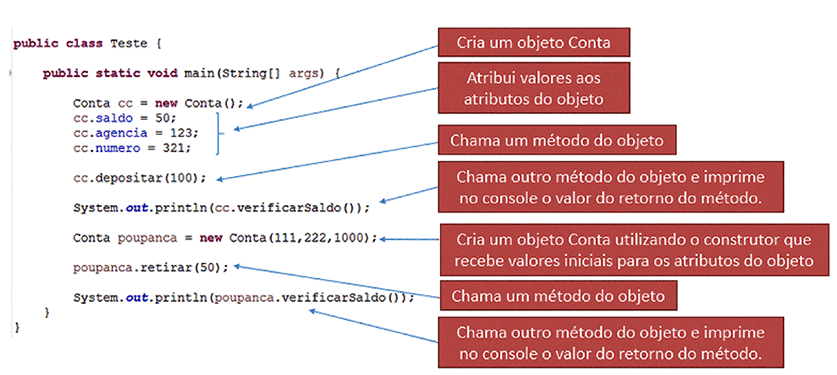


Figura 14 – Exemplo de utilização de objetos

Fonte: FIAP (2015)

No exemplo acima, foram criados dois objetos do tipo Conta. No primeiro objeto (**cc**), foram atribuídos valores aos seus atributos e invocados os métodos depositar e verificarSaldo, imprimindo o resultado no console. O segundo objeto criado (**poupança**) recebeu os valores iniciais de seus atributos pelo construtor. Após isso, foram chamados os comportamentos de retirar e verificarSaldo.

Reflita: qual o resultado da execução do programa de teste?

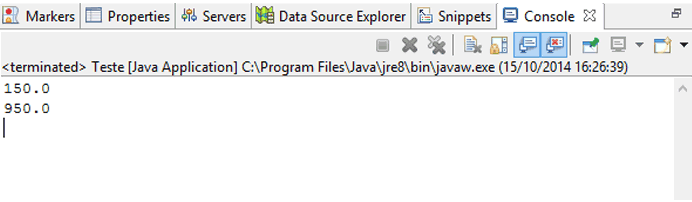


Figura 15 – Resultado da execução da classe de teste

Fonte: FIAP (2015)

O objeto referenciado na variável cc recebeu um valor inicial de 50. Depois, foi adicionado ao saldo o valor 100 através do método *depositar*. Assim, quando o método *verificarSaldo*é chamado, o valor retornado é 150.

O segundo objeto recebe um valor inicial de 1000. Após isso o método *retirar* é acionado com o valor 50. Assim, o saldo final é 950.

Observe que cada objeto possui os seus próprios valores para as variáveis de instância e seus métodos atuam dentro do próprio objeto, independentemente.

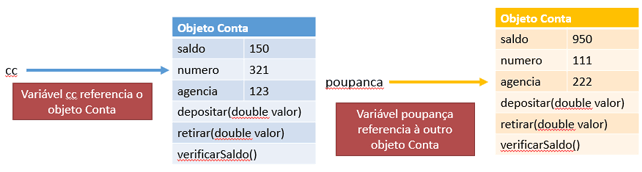


Figura 16 – Visão de variáveis que armazenam referências aos objetos Java

Fonte: FIAP (2015)

Uma variável que armazena a referência de um objeto pode ter o valor *null.* Esse valor quer dizer que a variável está vazia e não faz referência a nenhum objeto. Devemos tomar cuidado, pois no caso de tentar acessar um atributo ou método em uma variável vazia, irá ocorrer um erro na execução do programa.

É possível utilizar o valor *null* na lógica do seu programa sempre que for necessário verificar se a variável faz referência a um objeto ou não.

Exemplos:

Conta cc = null;

if (cc != null){

System.out.println(“Existe uma conta”);

}

## 3.5 Modificadores de acesso

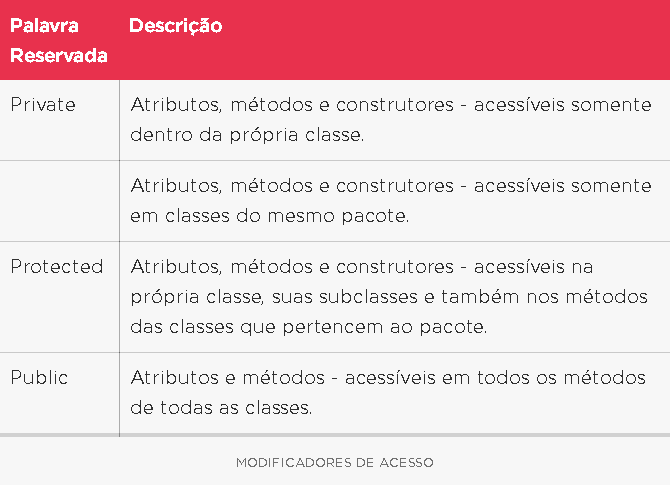
Os modificadores são palavras-chave que alteram as definições de uma classe, método, atributo ou construtor. Existem vários modificadores na linguagem Java, na qual fazem parte: *static, abstract e final.*

No decorrer do curso abordaremos cada uma delas, mas neste momento, vamos estudar os modificadores de acesso, que são as palavras-chave utilizadas para controlar o acesso a uma classe, variável de instância, método ou construtor.

O Java disponibiliza três modificadores de acesso: *public, protected e private.* Quando nenhum modificador é utilizado, o nível de acesso padrão (*default*) é utilizado. Esse nível de acesso também é conhecido como *package.*

Os modificadores usados com mais frequência nos sistemas são aqueles que controlam o acesso a métodos, atributos e construtores. Os modificadores irão determinar quais variáveis, métodos e construtores serão visíveis a outras classes.

O Quadro 1, a seguir, apresenta uma visão geral sobre os modificadores:



Quadro 1 –Visão geral sobre os modificadores

Fonte: FIAP (2015)

Os membros da classe declarados sem nenhum modificador de acesso terão o nível de acesso padrão. Dessa forma, os métodos, atributos e construtores serão acessíveis a outras classes que estiverem dentro do mesmo pacote.

Pacote, como visto no capítulo anterior, é uma forma de organizar as classes em uma estrutura de diretórios.

Observe no package explorer do eclipse a estrutura de pacotes e classes do projeto (Figura 17):

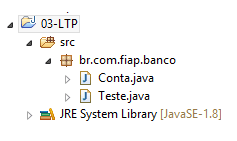


Figura 17 – Projeto exemplo

Fonte: FIAP (2015)

As classes Teste e Conta estão no mesmo pacote: **br.com.fiap.banco**

Observe agora, na Figura 18, a estrutura da classe Conta:

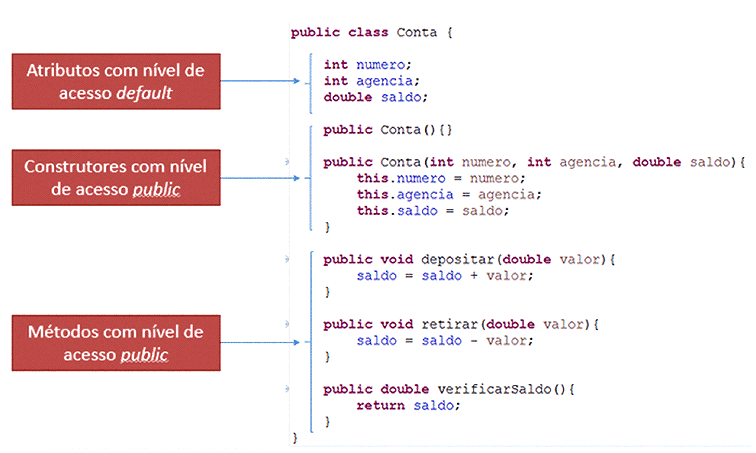


Figura 18 – Classe conta e modificadores de acesso

Fonte: FIAP (2015)

Todos os atributos não foram marcados com nenhum modificado de acesso, assim o nível de acesso é o *default*. Os construtores e métodos têm o nível de acesso *public*.

Dessa forma, os atributos são visíveis somente a classes que estiverem dentro do mesmo pacote da classe Conta. Os construtores e métodos são visíveis em qualquer outra classe, independentemente do pacote em que esteja.

Vamos criar um novo pacote para a classe *Teste*, chamado br.com.fiap.banco.teste (Figura 19):

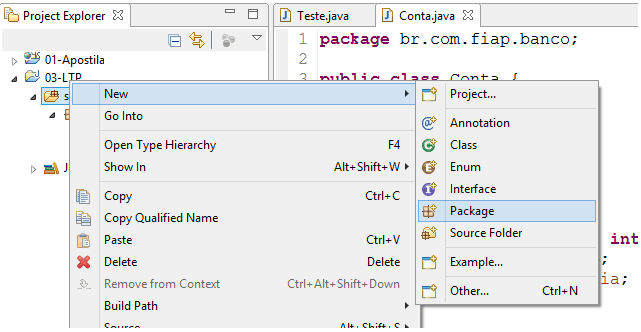


Figura 19 – Criando um novo pacote

Fonte: FIAP (2015)

Arraste a classe para o novo pacote, como mostra a Figura 20:

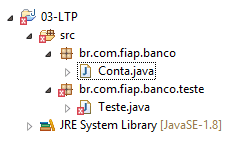


Figura 20 – Novo pacote para a classe Teste

Fonte: FIAP (2015)

Observe acima, que a classe de Teste está marcada com um X vermelho, indicando que existe algum erro na classe:

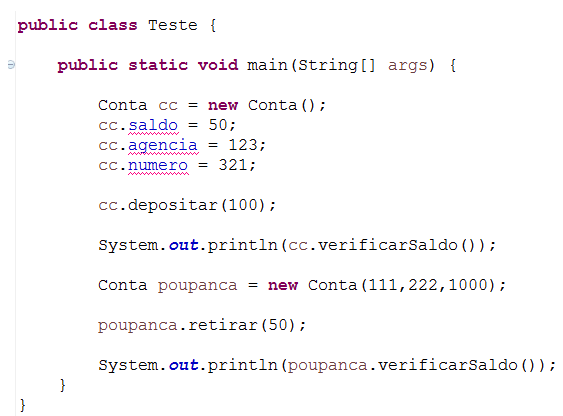


Figura 21 – Classe de Teste

Fonte: FIAP (2015)

O problema é que a classe Teste não tem mais acesso aos atributos, pois estes têm o nível de acesso *default* e as classes estão em pacotes diferentes. Por enquanto, marque os atributos da classe Conta com *public*, para corrigir o problema.

O modificador de acesso *private*permite o acesso aos métodos, atributos e construtores somente dentro da própria classe que as tem. Dessa forma, nenhuma outra classe terá acesso aos membros *privados*.

Esse modificador é utilizado para o encapsulamento, que será abordado em breve neste capítulo. Mas, previamente falando, o encapsulamento é um dos pilares da orientação a objetos e tem o objetivo de proteger o acesso indevido de seus atributos e métodos por outras classes.

E, finalmente, o modificador *protected*, que limita a visibilidade dos métodos, construtores e atributos de duas formas:

* Subclasses de uma classe.
* Outras classes no mesmo pacote.

Ou seja, esse modificador é parecido com o nível *default,* pois permite a visibilidade dentro do mesmo pacote. A diferença é a visibilidade dos membros da classe em classes filhas. Em uma herança, os atributos, métodos e construtores protegidos serão acessíveis nas subclasses dessa classe.

## 3.6 Java API specification

Vimos como criar as nossas classes, porém a biblioteca de classes do Java contém milhares de classes prontas e interfaces para o desenvolvimento de aplicações. Essas classes e interfaces estão agrupadas em pacotes, de acordo com suas funcionalidades. Por exemplo, as classes que são utilizadas para manipular arquivos estão dentro do pacote java.io.

Existe uma documentação para descrever as classes e seus membros public e protected. Essa documentação, conhecida como Java API Specification, apresenta uma visão geral de todas as classes com detalhes de seus construtores, métodos e campos públicos ou protegidos.

Essa documentação pode ser visualizada on-line em: <http://docs.oracle.com/javase/8/docs/api/>

A documentação está dividida em três janelas: a janela superior esquerda, que apresenta os pacotes da API Java em ordem alfabética; a janela logo abaixo, do lado esquerdo, lista as classes e interfaces, pode ser filtrado pela escolha do pacote da janela anterior; e a janela à direita, que exibe as informações da classe, pacote ou interface, dependendo da escolha das janelas anteriores, como mostra a Figura 22:

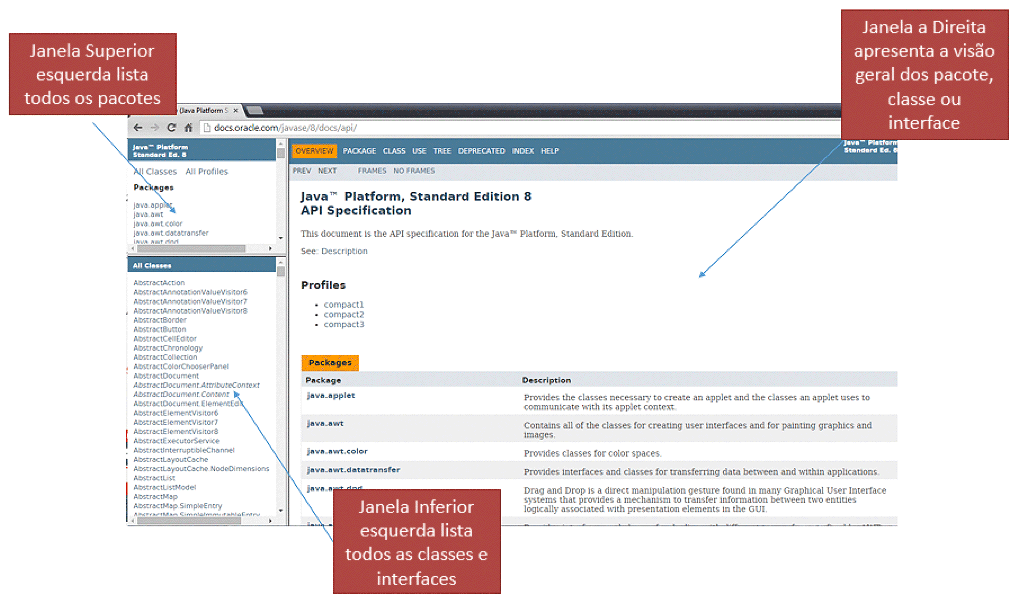


Figura 22 – Visão geral da API Java

Fonte: FIAP (2015)

## 3.7 Comentários

São informações incluídas no código fonte que não interferem no programa. É uma forma de melhorar a legibilidade e ajudar os desenvolvedores na organização e no entendimento do código.

Podemos utilizar os comentários para descrever o que for necessário, pois o compilador Java ignora totalmente os comentários ao preparar uma versão executável de um programa Java.

Existem três tipos de comentários:

* Para comentar uma linha, utilizam-se duas barras (//), tudo desde as barras até o final da linha será considerado comentário, sendo desconsiderado pelo compilador Java:

//comentário

* Caso seja necessário comentar mais de uma linha, pode-se iniciar o comentário com /\* e terminá-lo com \*/, tudo que estiver entre essas marcações será considerado comentário:

/\* Comentário

de várias

linhas \*/

* Comentário de documentação, ou seja, o comentário será legível tanto para computadores quanto para os desenvolvedores. Esse tipo de comentário é interpretado como sendo documentação oficial que descreve o funcionamento de uma classe e seus métodos. Dessa forma, podemos gerar uma documentação para outros desenvolvedores conhecerem o funcionamento de nossa classe. A sintaxe desse tipo de comentário é semelhante ao comentário de várias linhas, se inicia com /\*\* e termina \*/, ou seja:

/\*\* Comentário de Documentação \*/

Exemplos:

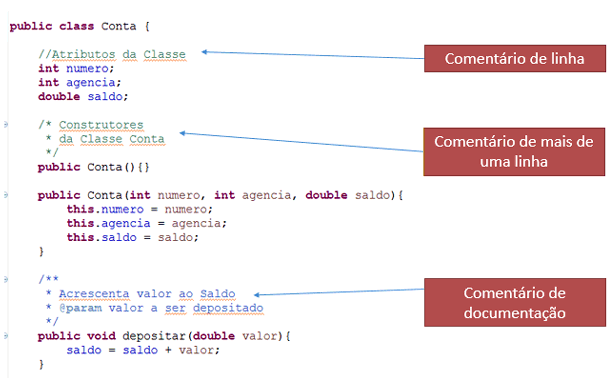


Figura 23 – Exemplos de comentários no Java

Fonte: FIAP (2015)

## 3.8 Javadoc

Agora que vimos a documentação oficial da Oracle, vamos criar a documentação das classes que desenvolvemos. Para isso, utilizaremos a ferramenta de documentação **javadoc.**, que permite que as classes criadas sejam documentadas no formato HTML.

Podemos utilizar um comentário de documentação para detalhar melhor as nossas classes, métodos, atributos etc.

Esse comentário tem a seguinte estrutura:

/\*\* Descrição

\* @tag descrição dessa tag

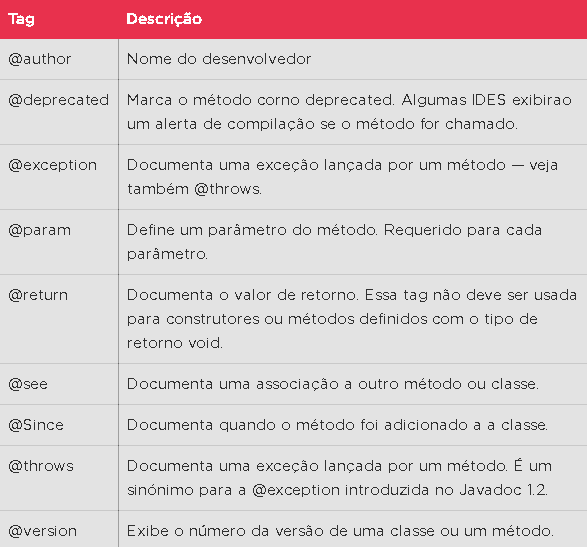
\*/

Os comentários de documentação devem ser inseridos imediatamente acima do elemento que está sendo documentado. Por exemplo, para documentar um método da classe, precisamos colocar o comentário logo acima do dele.

Além do texto que descreve o elemento, podemos utilizar algumas tags para dar um significado próprio a trechos do comentário. Como por exemplo, determinar o autor ou a versão do programa.

Essas tags são precedidas por um sinal de @, como @author ou @version.

As principais tags são apresentadas o Quadro 2, a seguir:



Quadro 2 – Tags de documentação

Fonte: FIAP (2015)

A Figura 24 apresenta um exemplo de utilização do comentário de documentação com as tags:

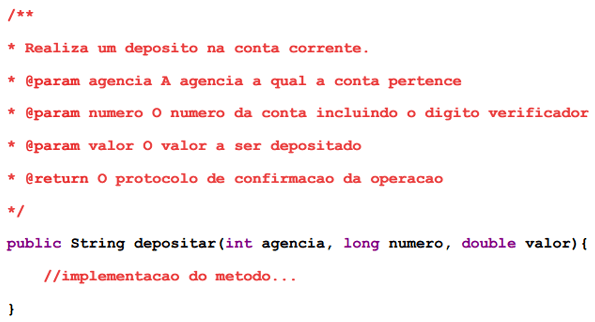


Figura 24 – Exemplo de comentário de documentação

Fonte: FIAP (2015)

Agora vamos alterar a classe Conta adicionando os comentários para a construção da documentação (Figura 25):

/\*\*

\* Classe que abstrai uma Conta Bancária

\* **@author** thiagoyama

\* **@version** 1.0

\*/

**public** **class** Conta {

/\*\*

\* Número da Conta

\*/

**public** **int** numero;

/\*\*

\* Número da Agencia

\*/

**public** **int** agencia;

/\*\*

\* Saldo da Conta

\*/

**public** **double** saldo;

**public** Conta(){}

**public** Conta(**int** numero, **int** agencia, **double** saldo){

**this**.numero = numero;

**this**.agencia = agencia;

**this**.saldo = saldo;

}

/\*\*

\* Deposita um valor ao saldo da conta

\* **@param** valor Valor a ser depositado

\*/

**public** **void** depositar(**double** valor){

saldo = saldo + valor;

}

/\*\*

\* Retira um valor do saldo da conta

\* **@param** valor Valor a ser retirado

\*/

**public** **void** retirar(**double** valor){

saldo = saldo - valor;

}

/\*\*

\* Verifica o saldo da conta

\* **@return** Valor do saldo da conta

\*/

**public** **double** verificarSaldo(){

**return** saldo;

}

}

Figura 25 – Classe Conta com comentários de documentação

Fonte: FIAP (2015)

Agora vamos gerar a documentação da classe Conta. A ferramenta javadoc fica dentro da pasta de instalação do Java (jdk) na pasta bin. Nesta pasta, existe o programa executável javadoc.exe.

Vamos utilizar o eclipse para gerar a documentação. Para isso, acesse o menu Project -> Generate javadoc, conforme a imagem abaixo (Figura 26):

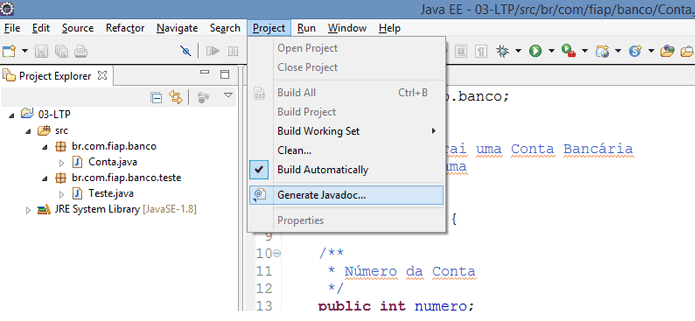


Figura 26 – Gerando documentação – Parte 1

Fonte: FIAP (2015)

Após essa operação, uma janela será aberta. Nela, primeiro configure a ferramenta javadoc, navegue até a pasta em que está a ferramenta e escolha o programa javadoc.exe.

Agora, escolha os pacotes que terão as suas classes e interfaces documentadas. Configure também uma pasta de destino para a criação dos arquivos de documentação, como mostra a Figura 27:

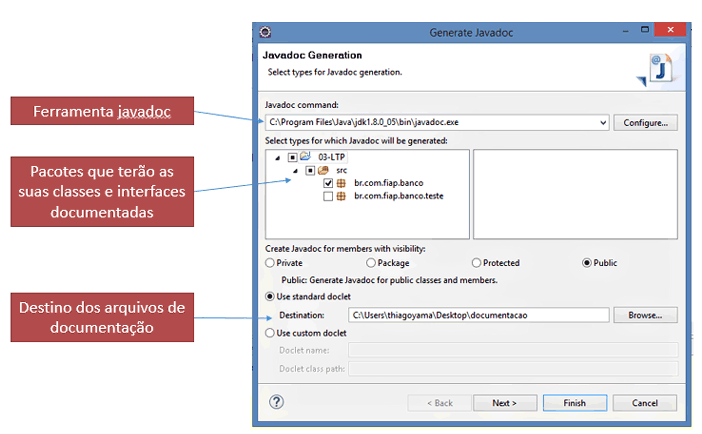


Figura 27 – Gerando documentação – Parte 2

Fonte: FIAP (2015)

O resultado pode ser visualizado na pasta que foi escolhida como destino dos arquivos de documentação. Abra o arquivo index.html e navegue à vontade!

A figura 28 apresenta a documentação da classe Conta:

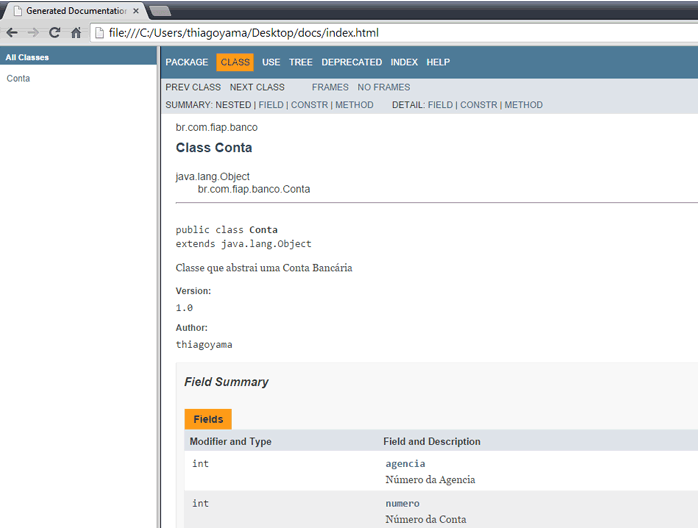


Figura 28 – Documentação da Classe Conta

Fonte: FIAP (2015)

## 3.9 Java bean e encapsulamento

Uma das principais vantagens da programação orientada a objetos é a capacidade de reutilizar o mesmo objeto em diferentes classes e programas. A especificação JavaBeans define diretrizes que ditam os nomes de suas variáveis, os nomes e tipos de retorno de seus métodos e alguns outros aspectos para que o objeto, chamado de beans, sejam reutilizáveis.

A ideia é criar pequenos componentes que possam ser reaproveitados ao máximo, simplificando o processo de desenvolvimento.

As regras para que uma classe seja um Java Bean são:

* Precisa ter um construtor padrão, sem argumentos.
* Encapsular os seus atributos, provendo métodos para o acesso a eles em outras classes.
* É uma boa prática a implementação da interface *java.io.Serializable*

Uma classe bean pode conter construtores com argumentos, porém ela deve ter também um construtor sem argumentos.

Vamos falar sobre interfaces ainda, agora só vamos deixar registrado que é uma boa prática a implementação da interface *Serializable*, que permite a serialização de objetos.

O encapsulamento é aplicado a métodos e atributos de uma classe e consiste em proteger os dados, ou até mesmo escondê-los.

Para encapsular um método, devemos utilizar o acesso mais restritivo quando desejamos que as outras classes não tenham acesso ao método. Permitindo assim o uso somente de dentro da própria classe.

Para limitar ou controlar o conteúdo de um atributo, métodos devem ser utilizados para atribuir ou alterar valores dos atributos de um objeto.

Dessa forma, sempre marcamos os atributos de uma classe com o nível de visibilidade mais restritiva (*private*), a não ser que exista um bom motivo para deixá-la com outro nível de acesso. Permitir o acesso total a um atributo não é uma boa prática, pois qualquer mudança na estrutura interna da classe acarretaria em mudanças nas outras classes que a utilizam, limitando a flexibilidade de mudança do código.

Outros benefícios do encapsulamento são:

* Esconder a implementação de uma classe: para utilizar uma classe que envia e-mail, precisamos saber somente os valores que a classe precisa receber para realizar o envio, não precisamos saber como ela o faz.
* Definir o modo de acesso aos dados: escrita, leitura, escrita e leitura.
* Proteger os dados que estão dentro dos objetos, evitando-se que eles sejam alterados erroneamente.

O uso de métodos de leitura (get) e escrita (set) visam desacoplar os atributos de uma classe dos clientes que a utilizam. No exemplo a seguir (Figura 29), o atributo idade está encapsulado:

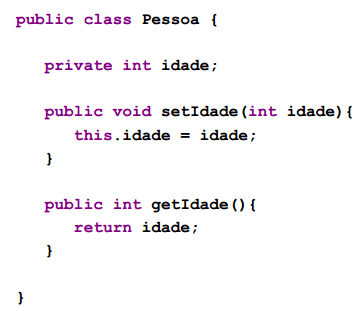


Figura 29 – Atributo idade encapsulado

Fonte: FIAP (2015)

A única forma de recuperar ou alterar o valor do atributo idade é utilizando os métodos assessores, get e set. Como podemos perceber, a convenção de nome para métodos que alteram o valor do atributo é adicionar a palavra **set** antes do nome do atributo. Para os métodos que recuperam a informação, utiliza-se a palavra **get** seguida do nome do atributo. Para atributos do tipo **boolean,** também é possível utilizar a palavra **is** antes do atributo. Como mostra a Figura 30.

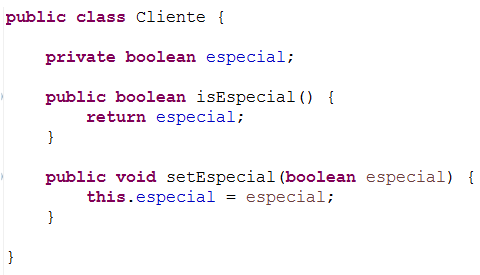


Figura 30 – Atributo do tipo boolean encapsulado

Fonte: FIAP (2015)

Para os métodos, podemos encapsular da seguinte forma:

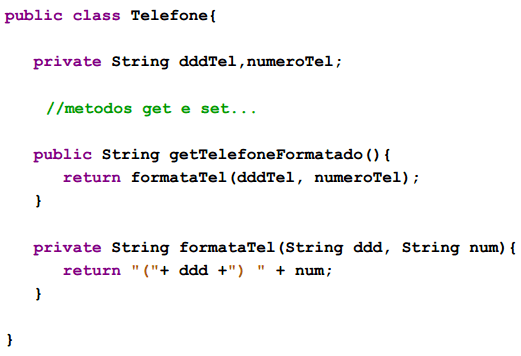


Figura 31 – Método encapsulado

Fonte: FIAP (2015)

A Figura 31 exibe o método **formataTel** encapsulado, pois está marcado como private*,* que permite o acesso somente de dentro da própria classe Telefone.

Agora, vamos alterar a Classe conta para seguir a especificação JavaBean (Figura 32). Altere a visibilidade dos atributos de public para private. Crie os métodos assessores (gets e sets) dos atributos.

Observe que o método **setSaldo** não foi implementado, pois não podemos deixar a alteração do valor do saldo seja feito de qualquer maneira. Para isso, existem os métodos **depositar** e **retirar**.

Outro detalhe foi a remoção do método **verificarSaldo**, que foi substituído pelo método **getSaldo,** já que os dois tinham o mesmo comportamento.

/\*\*

\* Classe que abstrai uma Conta Bancária

\* **@author** thiagoyama

\* **@version** 1.0

\*/

**public** **class** Conta {

/\*\*

\* Número da Conta

\*/

**private** **int** numero;

/\*\*

\* Número da Agencia

\*/

**private** **int** agencia;

/\*\*

\* Saldo da Conta

\*/

**private** **double** saldo;

**public** Conta(){}

**public** Conta(**int** numero, **int** agencia, **double** saldo) {

**super**();

**this**.numero = numero;

**this**.agencia = agencia;

**this**.saldo = saldo;

}

/\*\*

\* Deposita um valor ao saldo da conta

\* **@param** valor Valor a ser depositado

\*/

**public** **void** depositar(**double** valor){

saldo = saldo + valor;

}

/\*\*

\* Retira um valor do saldo da conta

\* **@param** valor Valor a ser retirado

\*/

**public** **void** retirar(**double** valor){

saldo = saldo - valor;

}

**public** **int** getNumero() {

**return** numero;

}

**public** **void** setNumero(**int** numero) {

**this**.numero = numero;

}

**public** **int** getAgencia() {

**return** agencia;

}

**public** **void** setAgencia(**int** agencia) {

**this**.agencia = agencia;

}

**public** **double** getSaldo() {

**return** saldo;

}

}

Figura 32 – Classe JavaBean Conta

Fonte: FIAP (2015)

Modifique também a classe de teste (Figura 33), para corrigir os erros que surgiram.

**public** **class** Teste {

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

Conta cc = **new** Conta();

cc.depositar(50);

cc.setAgencia(123);

cc.setNumero(321);

cc.depositar(100);

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

Conta poupanca = **new** Conta(111,222,1000);

poupanca.retirar(50);

System.***out***.println(poupanca.getSaldo());

}

}

Figura 33 – Classe de Teste

Fonte: FIAP (2015)

## **REFERÊNCIAS**

BARNES, David J. **Programação Orientada a Objetos com Java**: Uma introdução Prática Utilizando Blue J. São Paulo: Pearson, 2004.

CADENHEAD, Rogers; LEMAY, Laura. **Aprenda em 21 dias Java 2 Professional Reference.** 5. ed. Rio de Janeiro: Elsevier, 2003.

DEITEL, Paul; DEITEL, Harvey. **Java Como Programar**. 8. ed. São Paulo. Pearson, 2010.

HORSTMANN, Cay; CORNELL, Gary. **Core Java**: Volume I. Fundamentos. 8. ed. São Paulo: Pearson 2009.

SIERRA, Kathy; BATES, Bert. **Use a cabeça! Java.** Rio de Janeiro: Alta Books, 2010.