

Apresentação do componente.

DESENVOLVIMENTO PARA SERVIDORES I (4)²⁴

Objetivos gerais. Esta disciplina fornecerá uma visão geral da linguagem *script* PHP associado a um gerenciador de banco de dados que utilize a linguagem SQL e como usar essas tecnologias para gerar sites dinâmicos. Introduzir práticas de codificação seguras.

Objetivos específicos. Os estudantes deverão ser capazes de desenvolver um CMS simples ou um aplicativo *Web* completo (lado cliente e lado servidor).

Ementa. PHP histórico e emprego. Instalação e configuração básica do PHP e um IDE. Sintaxe básica do PHP. Usando o PHP como um mecanismo de modelo simples. Panorama das melhores práticas com PHP. Conceitos de programação HTTP. Codificação de caracteres. Localidades, fusos horários e funções de tempo. *Strings*. Uso de Array e funções de matriz. Orientação a objetos em PHP (Classes, objetos, herança, encapsulamento, polimorfismo, agregação, composição e métodos). Tratamento de exceções de erro. Arquitetura do lado do servidor. Manipulação de dados postados. Enviando e-mail. Sessões e autenticação. Cookies. Arquivo manuseio e armazenamento de dados em arquivos de texto. Gerenciador de banco de dados e suas funções. *Frameworks*. Web Services, API, RSS, JSON e Ajax. Hospedagem compartilhada.

Bibliografia básica

BEIGHLEY, L; MORRISON, M. *Use a cabeça! PHP & MySQL*. São Paulo: Alta Books, 2011.

DALL'OGGIO, P. *PHP - programando com orientação a objetos*. São Paulo: Novatec, 2009.

YANK, K. *Build your own database driven web site using PHP*. New York: Oreilly & Assoc, 2012.

Bibliografia complementar

DOYLE, M. *Beginning PHP 5.3*. Indianapolis: Wiley Pub, 2009.

GILMORE, W. J. *Dominando PHP e MYSQL do iniciante ao profissional*. Rio de Janeiro: Starlin Alta Consult, 2008.

1-) Comunicação de alunos com alunos e professores:

- Um e-mail para a sala é de grande valia para divulgação de material, notícias e etc.
- Criação de grupo nas redes sociais também é interessante.

2-) Uso de celulares:

Para o bom andamento das aulas, recomendo que utilizem os celulares em *vibracall*, para não atrapalhar o andamento da aula.

3-) Material das aulas:

A disciplina trabalha com NOTAS DE AULA que são disponibilizadas ao final de cada aula, para isso vamos utilizar o GitHub, nosso repositório de aula será: <https://github.com/marcossousa33dev/FATECSRDSI202502>

4-) Prazos de trabalhos e atividades:

Toda atividade solicitada terá uma data limite de entrega, de forma alguma tal data será postergada, ou seja, se não for entregue até a data limite a mesma receberá nota 0, isso tanto para atividades entregues de forma impressa ou enviadas ao e-mail da disciplina. Qualquer problema que tenham, me procurem com antecedência para verificarmos o que pode ser realizado.

5-) Qualidade do material de atividades:

- Impressas ou manuscritas:
Muita atenção na qualidade do que será entregue, atividades sem grampear, faltando nome e número de componentes, rasgadas, amassadas, com rebarba de folha de caderno e etc. serão desconsiderados por mim.
- Digitais:
Ao enviarem atividades para o e-mail da disciplina, SEMPRE no assunto deverá ter o nome da atividade que está sendo enviada, e no corpo do e-mail deverá ter o(s) nome(s) do(s) integrante(s) da atividade, sem estar desta forma a atividade será DESCONSIDERADA.

6-) Critérios de avaliação:

Cada trimestres teremos as seguintes formas de avaliação:

- Práticas em Laboratório;
- Assiduidade;

- Outras que se fizerem necessário.

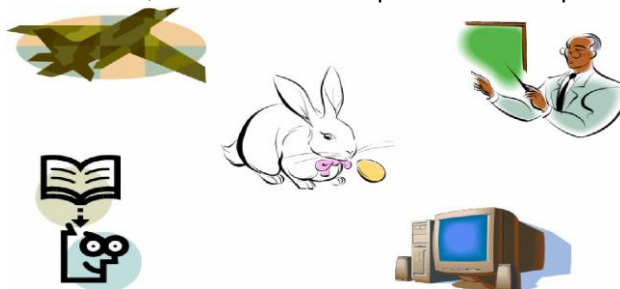
1. Introdução

Iremos relatar nessa fase de nosso curso a orientação à objetos com definições que serão importantes e que podemos implementar em qualquer linguagem de programação que seja OO.

2. Objetos

O Que São Objetos?

De acordo com o dicionário: - Objeto: "1. Tudo que se oferece aos nossos sentidos ou à nossa alma. 2. Coisa material: Havia na estante vários objetos. 3. Tudo que constitui a matéria de ciências ou artes. 4. Assunto, matéria. 5. Fim a que se mira ou que se tem em vista".



A figura destaca uma série de objetos. Objetos podem ser não só coisas concretas como também coisas inanimadas, como por exemplo uma matrícula, as disciplinas de um curso, os horários de aula.

Na programação orientada a objetos, implementa-se um conjunto de classes que definem os objetos presentes no *software*. Cada classe possui um comportamento (definidos pelos métodos) e estados possíveis (valores dos atributos) de seus objetos, assim como o relacionamento com outros objetos.

Há alguns anos, Alan Kay, um dos pais do paradigma da orientação a objetos, formulou a chamada "analogia biológica". Nessa analogia ele imaginou como seria um sistema de software que funcionasse como um ser vivo, no qual cada "célula" interagiria com outras células através do envio de mensagens para realização do objetivo comum.

De uma forma mais geral, Kay pensou em como construir um sistema de software a partir de agentes autônomos que interagem entre si. Ele então estabeleceu os seguintes princípios da orientação a objetos:

- (a) Qualquer coisa é um objeto;
- (b) Objetos realizam tarefas através da requisição de serviços a outros objetos;
- (c) Cada objeto pertence a uma determinada classe. Uma classe agrupa objetos similares;
- (d) A classe é um repositório para comportamento associado ao objeto;
- (e) Classes são organizadas em hierarquias.

Quando temos um problema e queremos resolvê-lo usando um computador, necessariamente temos que fazer um programa. Este nada mais é do que uma série de instruções que indicam ao computador como proceder em determinadas situações. Assim, o grande desafio do programador é estabelecer a associação entre o modelo que o computador usa e o modelo que o problema lhe apresenta. Isso geralmente leva o programador a modelar o problema apresentado para o modelo utilizado em alguma linguagem. Se a linguagem escolhida for LISP, o problema será traduzido como listas encadeadas. Se for Prolog, o problema será uma cadeia de decisões. Assim, a representação da solução para o problema é característica da linguagem usada, tornando a escrita difícil.

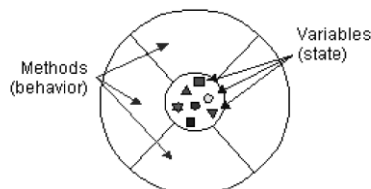
A orientação a objetos tenta solucionar esse problema. A orientação a objetos é geral o suficiente para não impor um modelo de linguagem ao programador, permitindo a ele escolher uma estratégia, representando os aspectos do problema em objetos. Assim, quando se "lê" um programa orientado a objeto, podemos ver não apenas a solução, mas também a descrição do problema em termos do próprio problema. O programador consegue "quebrar" o grande problema em pequenas partes que juntas fornecem a solução.

Olhando à sua volta é possível perceber muitos exemplos de objetos do mundo real: seu cachorro, sua mesa, sua televisão, sua bicicleta. Esses objetos têm duas características básicas, que são o **estado** e o **comportamento**. Por exemplo, os cachorros tem nome, cor, raça (estados)

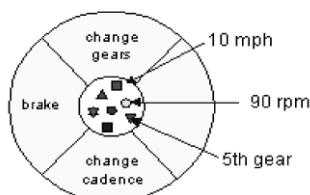
e eles balançam o rabo, comem, e latem (comportamento). Uma bicicleta tem 18 marchas, duas rodas, banco (estado) e elas brecam, aceleram e mudam de marcha (comportamento).

De maneira geral, definimos objetos como um conjunto de variáveis e métodos, que representam seus estados e comportamentos.

Veja a ilustração:



Temos aqui representada (de maneira lógica) a ideia que as linguagens orientadas a objeto utilizam. Tudo que um objeto sabe (estados ou variáveis) e tudo que ele pode fazer (comportamento ou métodos) está contido no próprio objeto. No exemplo da bicicleta, poderemos representar o objeto como no exemplo a seguir:



Temos métodos para mudar a marcha, a cadência das pedaladas, e para brecar. A utilização desses métodos altera os valores dos estados. Ao brecar, a velocidade diminui. Ao mudar a marcha, a cadência é alterada. Note que os diagramas mostram que as variáveis do objeto estão no centro do mesmo. Os métodos cercam esses valores e os “escondem” de outros objetos. Deste modo, só é possível ter acesso a essas variáveis através dos métodos. Esse tipo de construção é chamada de encapsulamento, e é a construção ideal para objetos.

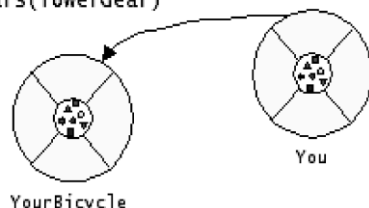
Mas um objeto sozinho geralmente não é muito útil. Ao contrário, em um programa orientado a objetos temos muitos objetos se relacionando entre si. Uma bicicleta encostada na garagem nada mais é que um pedaço de ferro e borracha. Por si mesma, ela não tem capacidade de realizar nenhuma tarefa. Apenas com outro objeto (você) utilizando a bicicleta, ela é capaz de realizar algum trabalho.

A interação entre objetos é feita através de mensagens. Um objeto “chama” os métodos de outro, passando parâmetros quando necessário. Quando você passa a mensagem “mude de marcha” para o objeto bicicleta você precisa dizer qual marcha você deseja.

A figura a seguir mostra os três componentes que fazem parte de uma mensagem:

- Objeto para o qual a mensagem é dirigida (bicicleta)
- Nome do método a ser executado (muda marcha)
- Os parâmetros necessários para a execução do método.

`changeGears(lowerGear)`



A troca de mensagens nos fornece ainda um benefício importante. Como todas as interações são feitas através delas, não importa se o objeto faz parte do mesmo programa; ele pode estar até em outro computador. Existem maneiras de enviar essas mensagens através da rede, permitindo a comunicação remota entre objetos.

Assim, um programa orientado a objetos nada mais é do que um punhado de objetos dizendo um ao outro o que fazer. Quando você quer que um objeto faça alguma coisa, você envia a ele uma "mensagem" informando o que quer fazer, e o objeto faz. Se ele precisar de outro objeto que o auxiliar a realizar o "trabalho", ele mesmo vai cuidar de enviar mensagem para esse outro objeto. Deste modo, um programa pode realizar atividades muito complexas baseadas apenas em uma mensagem inicial.

Você pode definir vários tipos de objetos diferentes para seu programa. Cada objeto terá suas próprias características, e saberá como interpretar certos pedidos. A parte interessante é que objetos de mesmo tipo se comportam de maneira semelhante. Assim, você pode ter funções genéricas que funcionam para vários tipos diferentes, economizando código.

Vale lembrar que métodos são diferentes de procedimentos e funções das linguagens procedurais, e é muito fácil confundir os conceitos. Para evitar a confusão, temos que entender o que são classes.

3. O que é uma classe?

Assim como os objetos do mundo real, o mundo da Programação Orientada a Objetos (POO) agrupa os objetos pelos comportamentos e atributos comuns.

A biologia classifica todos os cães, gatos, elefantes e seres humanos como mamíferos. Características compartilhadas dão a essas criaturas separadas um senso de comunidade. No mundo do software, as classes agrupam objetos relacionados da mesma maneira.

Uma classe define todas as **características comuns a um tipo de objeto**. Especificamente, a classe **define todos os atributos e comportamentos** expostos pelo objeto. A classe define às quais mensagens o seu objeto responde. Quando um objeto quer exercer o comportamento de outro objeto, ele não faz isso diretamente, mas pede ao outro objeto para que se mude, normalmente baseado em alguma informação adicional. Frequentemente, isso é referido como “envio de mensagem”.

Uma *classe* define os atributos e comportamentos comuns compartilhados por um tipo de objeto. Imagine a classe como a forma do objeto. Os objetos de certo tipo ou classificação compartilham os mesmos comportamentos e atributos. As classes atuam de forma muito parecida com um cortador de molde ou biscoito, no sentido de que você usa uma classe para criar ou *instanciar* objetos.

Os *atributos* são as características de uma classe visíveis externamente. A cor dos olhos e a cor dos cabelos são exemplos de atributos. Um objeto pode expor um atributo fornecendo um link direto a alguma variável interna ou retornando o valor através de um método.

Comportamento é uma ação executada por um objeto quando passada uma mensagem ou em resposta a uma mudança de estado: é algo que um objeto faz. Um objeto pode executar o *comportamento* de outro, executando uma operação sobre esse objeto. Você pode ver os termos: *chamada de método*, *chamada de função* ou *passar uma mensagem*; usados em vez de executar uma *operação*. O que é importante é que cada uma dessas ações omite o comportamento de um objeto.

A definição de classes e seus inter-relacionamentos é o principal resultado da etapa de projeto de software. Em geral, esse resultado é expresso em termos de alguma linguagem de modelagem, tal como UML.

Uma classe é um gabarito para a definição de objetos. Através da definição de uma classe, descreve-se que propriedades — ou atributos — o objeto terá.

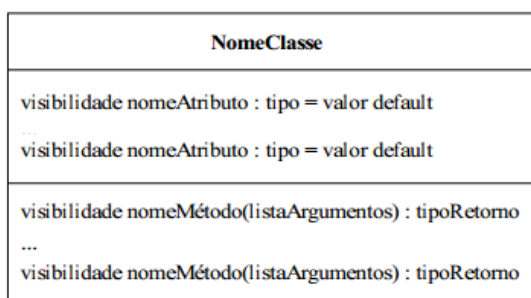
Além da especificação de atributos, a definição de uma classe descreve também qual o comportamento de objetos da classe, ou seja, que funcionalidades podem ser aplicadas a objetos da classe.

Essas funcionalidades são descritas através de métodos. Um método nada mais é que o equivalente a um procedimento ou função, com a restrição que ele manipula apenas suas variáveis locais e os atributos que foram definidos para a classe.

Uma vez que estejam definidas quais serão as classes que irão compor uma aplicação, assim como qual deve ser sua estrutura interna e comportamento, é possível criar essas classes em Java.

Na *Unified Modeling Language* (UML), a representação para uma classe no diagrama de classes é tipicamente expressa na forma gráfica, como mostrado na Figura a seguir.

Como se observa nessa figura, a especificação de uma classe é composta por três regiões: o nome da classe, o conjunto de atributos da classe e o conjunto de métodos da classe.



O nome da classe é um identificador para a classe, que permite referenciá-la posteriormente por exemplo, no momento da criação de um objeto.

O conjunto de atributos descreve as propriedades da classe. Cada atributo é identificado por um nome e tem um tipo associado. Em uma linguagem de programação orientada a objetos pura, o tipo é o nome de uma classe. Na prática, a maior parte das linguagens de programação orientada a objetos oferecem um grupo de tipos primitivos, como inteiro, real e caráter, que podem ser usados na descrição de atributos. O atributo pode ainda ter um valor_default opcional, que especifica um valor inicial para o atributo.

Os métodos definem as funcionalidades da classe, ou seja, o que será possível fazer com objetos dessa classe. Cada método é especificado por uma assinatura, composta por um identificador para o método (o nome do método), o tipo para o valor de retorno e sua lista de argumentos, sendo cada argumento identificado por seu tipo e nome.

Através do mecanismo de sobrecarga (*overloading*), dois métodos de uma classe podem ter o mesmo nome, desde que suas assinaturas sejam diferentes. Tal situação não gera conflito pois o compilador é capaz de detectar qual método deve ser escolhido a partir da análise dos tipos dos argumentos do método. Nesse caso, diz-se que ocorre a ligação prematura (*early binding*) para o método correto.

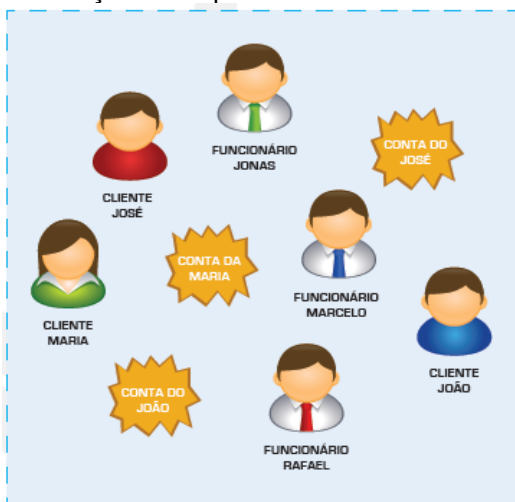
O modificador de visibilidade pode estar presente tanto para atributos como para métodos. Em princípio, três categorias de visibilidade podem ser definidas:

- público, denotado em UML pelo símbolo +: nesse caso, o atributo ou método de um objeto dessa classe pode ser acessado por qualquer outro objeto (visibilidade externa total);
- privativo, denotado em UML pelo símbolo -: nesse caso, o atributo ou método de um objeto dessa classe não pode ser acessado por nenhum outro objeto (nenhuma visibilidade externa);
- protegido, denotado em UML pelo símbolo #: nesse caso, o atributo ou método de um objeto dessa classe poderá ser acessado apenas por objetos de classes que sejam derivadas dessa através do mecanismo de herança.

4. Domínio e Aplicação

Um domínio é composto pelas entidades, informações e processos relacionados a um determinado contexto. Uma aplicação pode ser desenvolvida para automatizar ou tornar factível as tarefas de um domínio. Portanto, uma aplicação é basicamente o “reflexo” de um domínio.

Para exemplificar, suponha que estamos interessados em desenvolver uma aplicação para facilitar as tarefas do cotidiano de um banco. Podemos identificar clientes, funcionários, agências e contas como entidades desse domínio. Assim como podemos identificar as informações e os processos relacionados a essas entidades.



DOMÍNIO BANCÁRIO

Observação:

A identificação dos elementos de um domínio é uma tarefa **difícil**, pois depende fortemente do conhecimento das entidades, informações e processos que o compõem. Em geral, as pessoas que possuem esse conhecimento ou parte dele estão em contato constante com o domínio e não possuem conhecimentos técnicos para desenvolver uma aplicação.

Desenvolvedores de software buscam constantemente mecanismos para tornar mais eficiente o entendimento dos domínios para os quais eles devem desenvolver aplicações.

5. Objetos, Atributos e Métodos

As entidades identificadas no domínio devem ser representadas de alguma forma dentro da aplicação correspondente. Nas aplicações orientadas a objetos, as entidades são representadas por objetos.

- Uma aplicação orientada a objetos é composta por objetos.
- Em geral, um objeto representa uma entidade do domínio.

Para exemplificar, suponha que no domínio de um determinado banco exista um cliente chamado João. Dentro de uma aplicação orientada a objetos correspondente a esse domínio, deve existir um objeto para representar esse cliente.

Suponha que algumas informações do cliente João como nome, data de nascimento e sexo são importantes para o banco. Já que esses dados são relevantes para o domínio, o objeto que representa esse cliente deve possuir essas informações. Esses dados são armazenados nos atributos do objeto que representa o João.

- Um atributo é uma variável que pertence a um objeto.
- Os dados de um objeto são armazenados nos seus atributos.

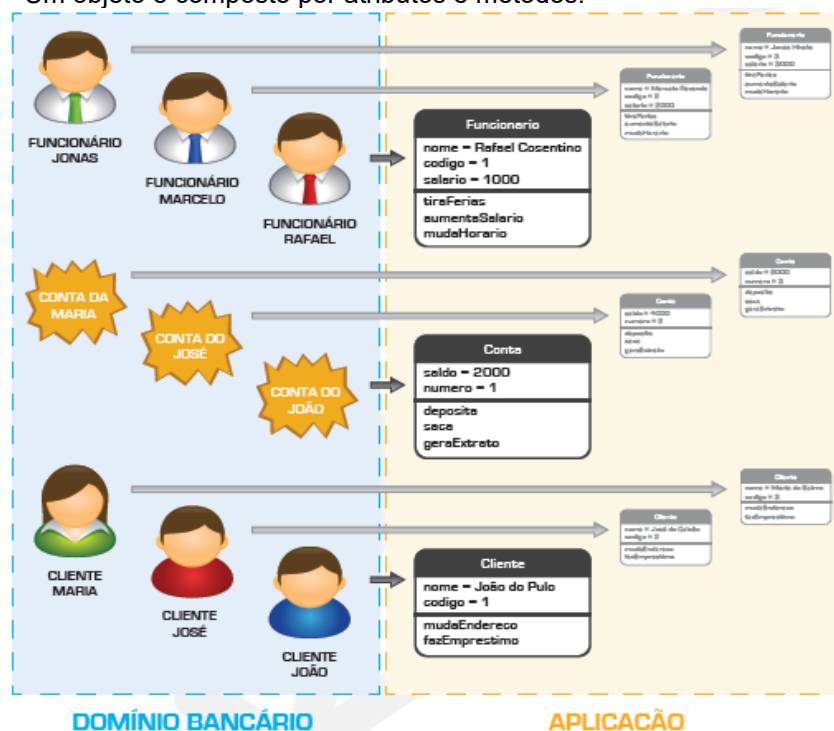
O próprio objeto deve realizar operações de consulta ou alteração dos valores de seus atributos.

Essas operações são definidas nos métodos do objeto.

Os métodos também são utilizados para possibilitar interações entre os objetos de uma aplicação.

Por exemplo, quando um cliente requisita um saque através de um caixa eletrônico do banco, o objeto que representa o caixa eletrônico deve interagir com o objeto que representa a conta do cliente.

- As tarefas que um objeto pode realizar são definidas pelos seus métodos.
- Um objeto é composto por atributos e métodos.



Observações:

- Em geral, não é adequado utilizar o objeto que representa um determinado cliente para representar outro cliente do banco, pois os dados dos clientes podem ser diferentes.

Dessa forma, para cada cliente do banco, deve existir um objeto dentro do sistema para representá-lo.

- Os objetos não representam apenas coisas concretas como os clientes do banco. Eles também devem ser utilizados para representar coisas abstratas como uma conta de um cliente ou um serviço que o banco ofereça.

6. Convenções de nomenclatura: Camel, Pascal, Kebab e Snake case

Se você estuda programação, provavelmente se vê a todo momento nomeando coisas, como variáveis, classes, métodos, funções e muito mais. Mas, afinal, será que existe alguma convenção para ajudar nesse processo de nomeação?

Há diversas maneiras de aplicar boas práticas no seu código, uma delas é a convenção de nomenclatura, que são recomendações para nomear identificadores ajudando a tornar seu código mais limpo e fácil de compreender, mantendo um padrão para o leitor e até mesmo podendo fornecer informações sobre a função do identificador.

Além disso, podemos encontrar outras linguagens, como Go, que dependem muito de você conhecer a diferença entre os casos de uso, pois a convenção utilizada no nome possui um efeito semântico também.

Agora, vamos conhecer os quatro tipos mais comuns de convenções de nomenclatura para combinar palavras em uma única *String*.

6.1. Camel Case

Camel case deve começar com a primeira letra minúscula e a primeira letra de cada nova palavra subsequente maiúscula:



CoisasParaFazer

IdadeDoAmigo

ValorFinal

6.2. Snake case

Em snake case, conhecido também como “*underscore case*”, utilizamos *underline* no lugar do espaço para separar as palavras. Quando o *snake case* está em caixa alta, ele é chamado de “*screaming snake case*”:



coisas-para-fazer

idade-do-amigo

valor-final

PRIMEIRO-NOME

LISTA-INICIAL

6.3. Kebab case

Kebab case utiliza o traço para combinar as palavras. Quando o kebab case está em caixa alta, ele é chamado de “*screaming kebab case*”:

● ● ●

coisas-para-fazer

idade-do-amigo

valor-final

PRIMEIRO-NOME

LISTA-INICIAL

6.4. Convenções Java

O Java possui o *Java Secure Coding Guidelines*, uma documentação disponibilizada pela Oracle, baseada nos padrões de codificação apresentados no *Java Language Specification*. É importante ressaltar que essas convenções de código Java foram escritas em 1999 e não foram atualizadas desde então, portanto algumas informações contidas no documento podem não estar atualizadas. As convenções mais comuns são e que usamos:

- *camelCase* para variáveis, atributos e métodos;
- *PascalCase* para classes;
- *SCREAMING_SNAKE_CASE* para constantes.

● ● ●

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

        String primeiroNome = "Maria";
        int idade = 22;
        double alturaAtual = 1.65;
        final String MENSAGEM_PADRAO = "Olá";
    }
}
```

7. Semântica e Sintaxe, vamos entender?

Se você respondeu sim! Parabéns!

Na vasta paisagem da programação, duas palavras - semântica e sintaxe - surgem frequentemente, mas nem sempre são entendidas em sua totalidade. A semântica e a sintaxe são como duas faces de uma mesma moeda, ambas essenciais para a compreensão e execução bem-sucedida de um código. No entanto, é fundamental entender a diferença entre elas e como elas se complementam para criar uma experiência de programação poderosa e eficaz.

7.1. O que é Semântica?

A semântica está relacionada ao significado ou interpretação do código. Em outras palavras, trata-se de entender o propósito e a função por trás das instruções escritas em um programa. Uma semântica forte garante que o código não apenas funcione corretamente, mas também faça sentido do ponto de vista lógico.

Por exemplo, ao escrever um código para calcular a média de uma lista de números, a semântica envolveria a compreensão de que estamos tentando encontrar um valor médio e que isso requer a soma de todos os números na lista, dividida pelo total de elementos na lista.

7.2. O que é Sintaxe?

A sintaxe, por outro lado, diz respeito à estrutura e à gramática do código. Ela se concentra em seguir as regras específicas da linguagem de programação para garantir que o código seja escrito de forma correta e compreensível pelo computador. Uma sintaxe correta é fundamental para evitar erros de compilação ou interpretação.

Continuando com o exemplo anterior, a sintaxe seria a forma como escrevemos o código em uma linguagem de programação específica, seguindo suas regras gramaticais. Por exemplo, em Python, a sintaxe para calcular a média de uma lista de números pode envolver o uso de loops e operadores aritméticos de acordo com a estrutura definida pela linguagem.

7.3. O Equilíbrio entre Semântica e Sintaxe

Para escrever um código eficiente e de fácil manutenção, é essencial encontrar o equilíbrio certo entre semântica e sintaxe. Um código com uma semântica forte, mas com uma sintaxe confusa, pode ser difícil de entender e dar manutenção. Da mesma forma, um código com uma sintaxe impecável, mas com uma semântica fraca, pode não atender às necessidades do usuário final.

O verdadeiro poder da programação reside na capacidade de comunicar ideias complexas de forma clara e concisa. Isso requer não apenas um domínio da sintaxe da linguagem de programação escolhida, mas também uma compreensão profunda dos princípios subjacentes e da lógica por trás do problema a ser resolvido.

7.4. Como Melhorar a Semântica e a Sintaxe do seu Código

Compreender o Problema: Antes de começar a escrever código, certifique-se de entender completamente o problema que está tentando resolver. Isso ajudará a orientar tanto a semântica quanto a sintaxe do seu código.

Escolher a Linguagem Adequada: Cada linguagem de programação tem suas próprias nuances de semântica e sintaxe. Escolha a linguagem que melhor se alinha com os requisitos do seu projeto e com a sua própria compreensão.

Praticar Regularmente: A prática leva à perfeição. Dedique tempo para praticar a escrita de código em diferentes cenários para aprimorar tanto a semântica quanto a sintaxe.

Revisar e Refatorar: Sempre revise seu código em busca de possíveis melhorias na semântica e na sintaxe. A refatoração é uma parte essencial do processo de desenvolvimento de software e pode ajudar a tornar seu código mais claro e eficiente.

Em última análise, a semântica e a sintaxe são duas partes igualmente importantes do processo de programação. Ao compreender e aplicar esses conceitos de forma equilibrada, os programadores podem criar soluções poderosas e elegantes para uma ampla gama de problemas.

8. Linguagem Java

Iremos agora abordar nossas aulas uma das maiores linguagens e que demanda um conhecimento específico, por isso que iremos aplicar os conceitos vistos em **Orientação à Objetos** nesta linguagem.

Java é uma linguagem de programação orientada a objeto, multiplataforma que é executada em bilhões de dispositivos em todo o mundo. Ele capacita aplicativos, sistemas operacionais de smartphones, software empresarial e muitos programas conhecidos. Apesar de estar sendo usado há mais de 20 anos, Java é atualmente a linguagem de programação mais popular para desenvolvedores de aplicativos. Java é:

Multiplataforma: Java foi marcado com o slogan "escreva uma vez, execute em qualquer lugar", e isso ainda é válido hoje. O código de programação Java escrito para uma plataforma, como o sistema operacional Windows, pode ser facilmente transferido para outra plataforma, como um sistema operacional de um celular, e vice-versa, sem ser completamente reescrito. O Java funciona em várias plataformas porque quando um programa Java é compilado, o compilador cria um arquivo de código de bytes .class que pode ser executado em qualquer sistema operacional que tenha a Máquina Virtual Java (JVM) instalada nele. Normalmente, é fácil instalar o JVM na maioria dos principais sistemas operacionais, incluindo iOS, o que nem sempre foi o caso.

Orientado a objeto: Java estava entre as primeiras linguagens de programação orientadas a objeto. Uma linguagem de programação orientada a objeto organiza seu código em torno de classes e objetos, em vez de funções e comandos. A maioria das linguagens de programação modernas, incluindo C++, C#, Python e Ruby, é orientada a objeto.

8.1. Quando o Java foi criado?

Java foi inventado por James Gosling em 1995 enquanto ele estava trabalhando na Sun Microsystems. Embora tenha obtido popularidade rapidamente após seu lançamento, o Java não foi iniciado como a linguagem de programação *powerhouse* que é hoje.



James Gosling

O desenvolvimento do que viria a ser o Java começou na Sun Microsystems em 1991. O projeto, inicialmente chamado de Oak, foi originalmente projetado para televisão interativa. Quando o Oak foi considerado avançado demais para a tecnologia de cabo digital disponível na época, Gosling e sua equipe mudaram seu foco para a criação de uma linguagem de programação e renomearam o projeto como Java, em homenagem a um tipo de café da Indonésia. O Gosling viu o Java como uma chance de resolver problemas que ele previu que estavam a caminho de linguagens de programação menos portáteis à medida que mais dispositivos se tornavam conectados em rede.

O Java foi projetado com um estilo de sintaxe semelhante à linguagem de programação C++ para que já fosse familiar para os programadores quando eles comessem a usá-lo. Com o slogan "escreva uma vez, execute em qualquer lugar" em seu núcleo, um programador poderia escrever código Java para uma plataforma que seria executada em qualquer outra plataforma que tivesse um interpretador Java (ou seja, Máquina Virtual Java) instalado. Com o surgimento da Internet e a proliferação de novos dispositivos digitais em meados da década de 1990, o Java foi rapidamente adotado pelos desenvolvedores como uma linguagem de programação verdadeiramente multiplataforma.

A primeira versão pública do Java, Java 1.0, foi lançada em 1996. Em cinco anos, ele tinha 2,5 milhões de desenvolvedores em todo o mundo. Hoje, o Java capacita tudo, desde o sistema operacional móvel Android até software empresarial.

8.2. Para que a linguagem de programação Java é usada?

O Java é uma linguagem de programação extremamente transferível usada entre plataformas e diferentes tipos de dispositivos, de smartphones a TVs inteligentes. É usado para criar aplicativos móveis e Web, software empresarial, dispositivos de Internet das Coisas (IoT), jogos, Big Data, distribuídos e aplicativos baseados em nuvem entre outros tipos. Aqui estão alguns exemplos específicos do mundo real de aplicativos programados com o Java.

8.2.1. Aplicativos móveis

Muitos, se não a maioria dos aplicativos móveis são criados com Java. Java é a linguagem preferida dos desenvolvedores de aplicativos móveis devido à sua plataforma estável e versatilidade. Os aplicativos móveis populares codificados em Java incluem Spotify, Signal e Cash App.

8.2.2. Aplicativos Web

Uma ampla variedade de aplicativos Web é desenvolvida usando Java. O Twitter e o LinkedIn estão entre os mais conhecidos.

8.2.3. Software empresarial

O software empresarial destina-se a atender a um grande grupo ou organização. Inclui softwares como sistemas de cobrança e programas de gerenciamento da cadeia de fornecedores. A alta escalabilidade do Java o torna uma linguagem atraente para desenvolvedores que escrevem software empresarial.

8.2.4. Jogos

Os jogos populares escritos em linguagem de programação Java incluem o Minecraft original e o *RuneScape*.

8.2.5. Aplicativos de IoT

Os aplicativos de IoT estão em todos os lugares — TVs inteligentes, carros, máquinas pesadas, instalações de trabalho e muito mais — e o Java é usado para programar muitos deles. O Java é uma opção popular para desenvolvedores de IoT devido à facilidade com que seu código pode ser transferido entre plataformas.

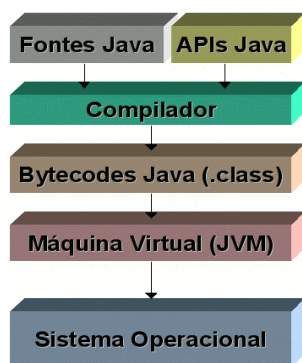
8.3. Como funciona o Java?

Conforme explicado anteriormente, o Java é uma linguagem de programação multiplataforma. Isso significa que ele pode ser escrito para um sistema operacional e executado em outro. Como isso é possível?

O código Java é escrito primeiro em um Kit de Desenvolvimento Java, que está disponível para Windows, Linux e macOS. Os programadores escrevem na linguagem de programação Java, que o kit traduz em código de computador que pode ser lido por qualquer dispositivo com o software certo. Isso é feito com um software chamado compilador. Um compilador usa um código de computador de alto nível como Java e o converte em uma linguagem que os sistemas operacionais entendem chamado código de bytes.

O código de bytes é então processado por um interpretador chamado Máquina Virtual Java (JVM). As JVMs estão disponíveis para a maioria das plataformas de software e hardware, e isso é o que permite que o código Java seja transferido de um dispositivo para outro. Para executar o Java, os JVMs carregam o código, verificam-no e fornecem um ambiente de *runtime*.

Dada a alta portabilidade do Java, não é de admirar que muitas pessoas queiram aprender como escrevê-lo. Felizmente, há muitos recursos disponíveis para começar a aprender Java.



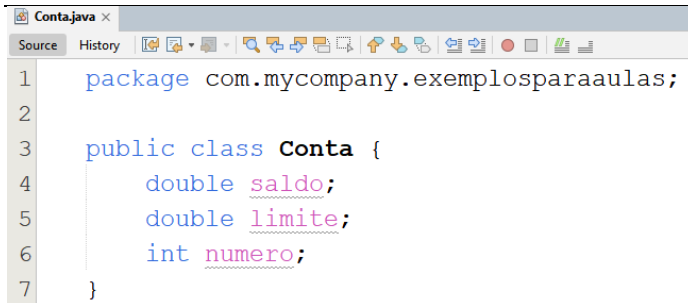
Funcionamento do Java

9. Classes em Java

O conceito de classe apresentado anteriormente é genérico e pode ser aplicado em diversas linguagens de programação. Mostraremos como a classe *Conta* poderia ser escrita utilizando a linguagem Java. Inicialmente, discutiremos apenas sobre os atributos. Os métodos serão abordados posteriormente.

A classe Java *Conta* é declarada utilizando a palavra reservada **class**. No corpo dessa classe, são declaradas três variáveis que são os atributos que os objetos possuirão. Como a linguagem Java é estaticamente tipada, os tipos dos atributos são definidos no código. Os atributos *saldo* e *limite* são do tipo *double*, que permite armazenar números com casas decimais, e o atributo *numero* é do tipo *int*, que permite armazenar números inteiros.

Por convenção, os nomes das classes na linguagem Java devem seguir o padrão “Pascal Case” e “Camel Case” como vimos:



```
1 package com.mycompany.exemplosparaaulas;
2
3 public class Conta {
4     double saldo;
5     double limite;
6     int numero;
7 }
```

10. Criando objetos em Java

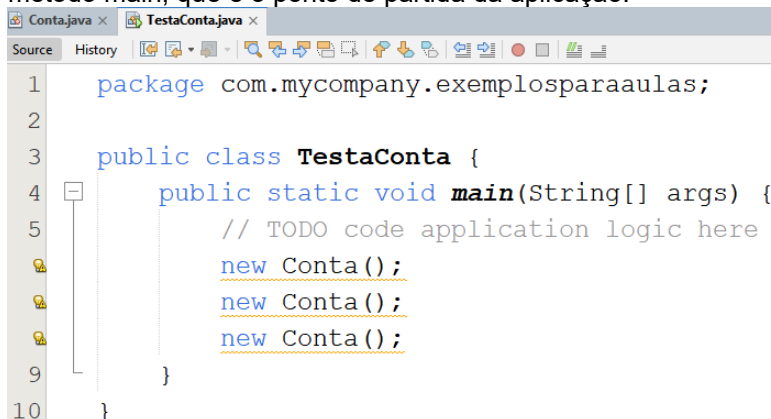
Após definir a classe `Conta`, podemos criar objetos a partir dela. Esses objetos devem ser alocados na memória RAM do computador. Felizmente, todo o processo de alocação do objeto na memória é gerenciado pela máquina virtual. O gerenciamento da memória é um dos recursos mais importantes oferecidos pela máquina virtual.

Do ponto de vista da aplicação, basta utilizar um comando especial para criar objetos e a máquina virtual se encarrega do resto. O comando para criar objetos é o **new**.



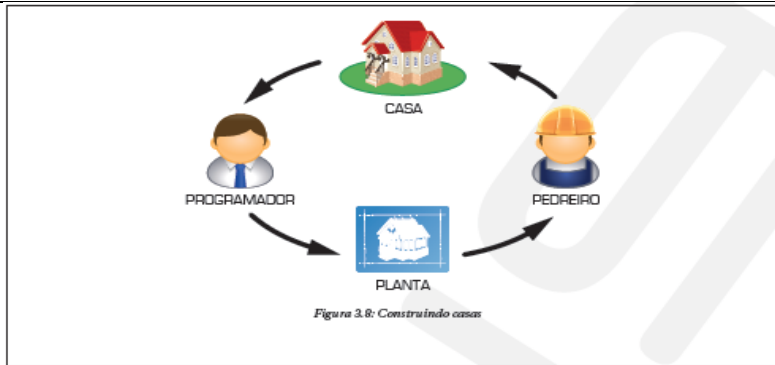
```
1 package com.mycompany.exemplosparaaulas;
2
3 public class TestaConta {
4     public static void main(String[] args) {
5         // TODO code application logic here
6         new Conta();
7     }
8 }
```

A linha com o comando **new** poderia ser repetida cada vez que desejássemos criar (instanciar) um objeto da classe `Conta`. A classe `TestaConta` serve apenas para colocarmos o método `main`, que é o ponto de partida da aplicação.



```
1 package com.mycompany.exemplosparaaulas;
2
3 public class TestaConta {
4     public static void main(String[] args) {
5         // TODO code application logic here
6         new Conta();
7         new Conta();
8         new Conta();
9     }
10 }
```

Chamar o comando **new** passando uma classe Java é como se estivéssemos contratando uma construtora passando a planta da casa que queremos construir. A construtora se encarrega de construir a casa para nós de acordo com a planta. Assim como a máquina virtual se encarrega de construir o objeto na memória do computador.



11. Referências

Todo objeto possui uma referência. A referência de um objeto é a única maneira de acessar os seus atributos e métodos. Dessa forma, devemos guardar as referências dos objetos que desejamos utilizar.

A princípio, podemos comparar a referência de um objeto com o endereço de memória desse objeto. De fato, essa comparação simplifica o aprendizado. Contudo, o conceito de referência é mais amplo. Uma referência é o elemento que permite que um determinado objeto seja acessado. Uma referência está para um objeto assim como um controle remoto está para um aparelho de TV. Através do controle remoto de uma TV você pode aumentar o volume ou trocar de canal. Analogamente, podemos controlar um objeto através da referência do mesmo.



12. Referências em Java

Ao utilizar o comando `new`, um objeto é alocado em algum lugar da memória. Para que possamos acessar esse objeto, precisamos de sua referência. O comando `new` devolve a referência do objeto que foi criado.

Para guardar as referências devolvidas pelo comando `new`, devemos utilizar variáveis não primitivas.

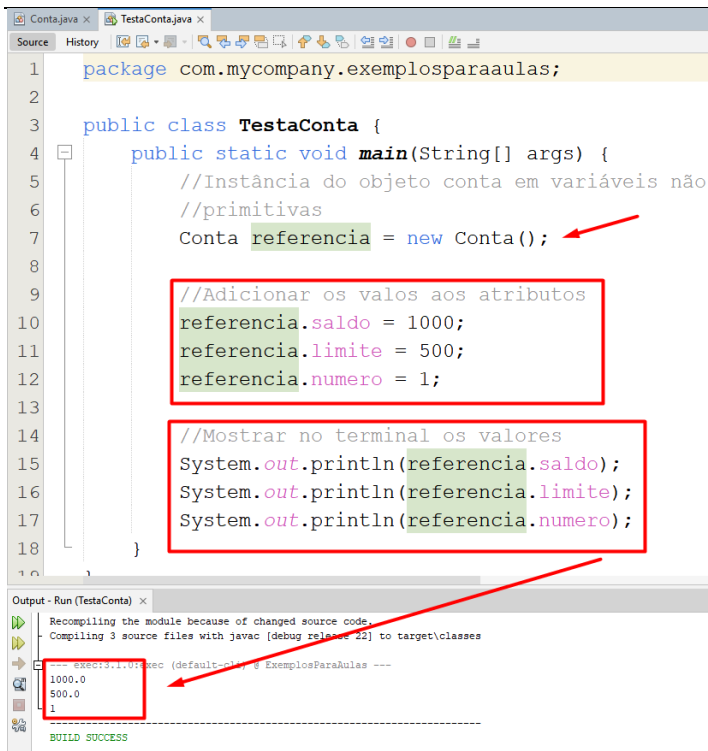
```

1 package com.mycompany.exemplosparaaulas;
2
3 public class TestaConta {
4     public static void main(String[] args) {
5         // TODO code application logic here
6         Conta referencia = new Conta();
7     }
8 }
    
```

No código Java acima, a variável **referencia** receberá a referência do objeto criado pelo comando `new`. Essa variável é do tipo `Conta`. Isso significa que ela só pode armazenar referências de objetos do tipo `Conta`.

13. Manipulando Atributos

Podemos alterar ou acessar os valores guardados nos atributos de um objeto se tivermos a referência a esse objeto. Os atributos são acessados pelo nome. No caso específico da linguagem Java, a sintaxe para acessar um atributo utiliza o operador `"."`.



```

1 package com.mycompany.exemplosparaaulas;
2
3 public class TestaConta {
4     public static void main(String[] args) {
5         //Instância do objeto conta em variáveis não
6         //primitivas
7         Conta referencia = new Conta();
8
9         //Adicionar os valores aos atributos
10        referencia.saldo = 1000;
11        referencia.limite = 500;
12        referencia.numero = 1;
13
14        //Mostrar no terminal os valores
15        System.out.println(referencia.saldo);
16        System.out.println(referencia.limite);
17        System.out.println(referencia.numero);
18    }
19 }

```

Output - Run (TestaConta) x

```

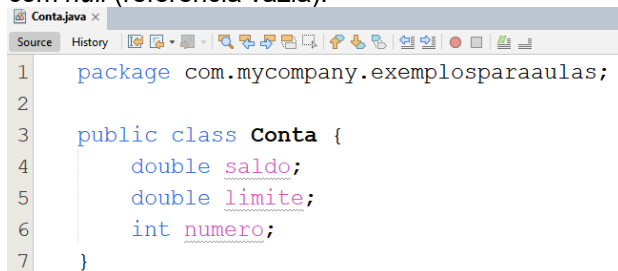
Recompiling the module because of changed source code.
Compiling 3 source files with javac [debug release 22] to target\classes
--- EXECUTING JAR (default-class) of ExemplosParaAulas ---
1000.0
500.0
1
BUILD SUCCESS

```

No código acima, o atributo saldo recebe o valor 1000. O atributo limite recebe o valor 500 e o numero recebe o valor 1. Depois, os valores são impressos na tela através da função `System.out.println()`.

14. Valores Padrão

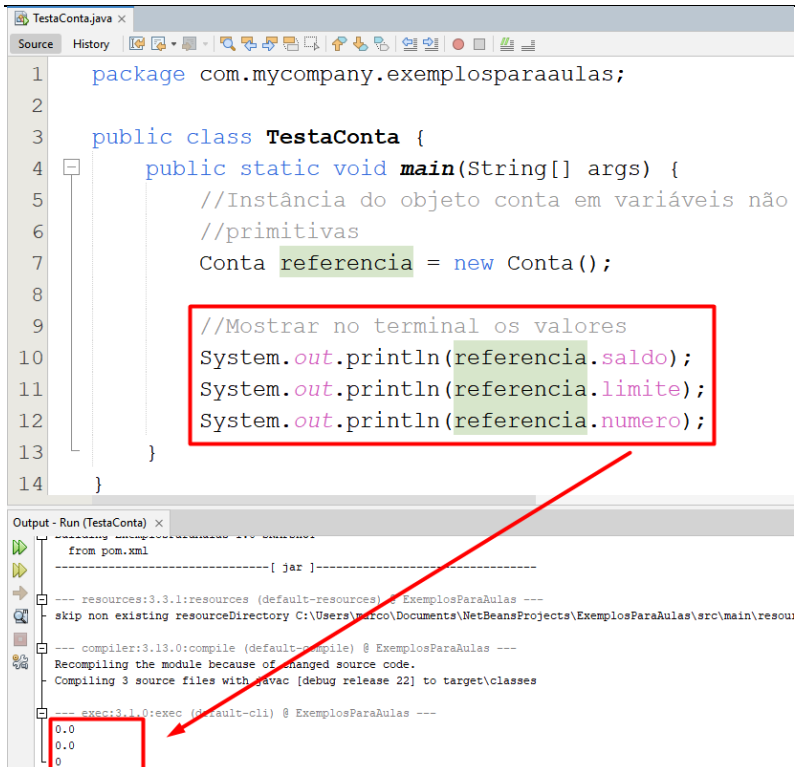
Poderíamos instanciar um objeto e utilizar seus atributos sem inicializá-los explicitamente, pois os atributos são inicializados com valores padrão. Os atributos de tipos numéricos são inicializados com 0, os atributos do tipo *boolean* são inicializados com false e os demais atributos com *null* (referência vazia).



```

1 package com.mycompany.exemplosparaaulas;
2
3 public class Conta {
4     double saldo;
5     double limite;
6     int numero;
7 }

```

```

1 package com.mycompany.exemplosparaaulas;
2
3 public class TestaConta {
4     public static void main(String[] args) {
5         //Instância do objeto conta em variáveis não
6         //primitivas
7         Conta referencia = new Conta();
8
9         //Mostrar no terminal os valores
10        System.out.println(referencia.saldo);
11        System.out.println(referencia.limite);
12        System.out.println(referencia.numero);
13    }
14 }

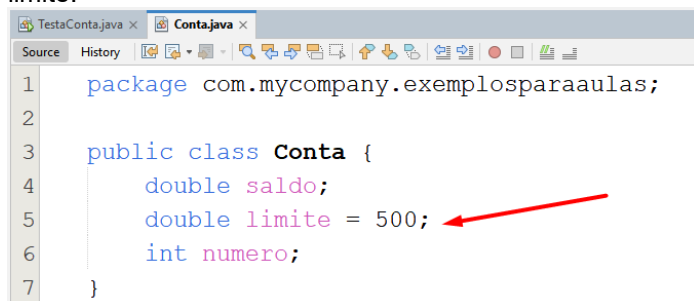
```

```

Output - Run (TestaConta) x
-----[ jar ]-----
--- resources:3.3.1:resources (default-resources) @ ExemplosParaAulas ---
skip non existing resourceDirectory C:\Users\marco\Documents\NetBeansProjects\ExemplosParaAulas\src\main\resou
--- compiler:3.13.0:compile (default-compile) @ ExemplosParaAulas ---
Recompiling the module because of changed source code.
Compiling 3 source files with javac [debug release 22] to target\classes
--- exec:3.1.0:exec (default-cli) @ ExemplosParaAulas ---
0.0
0.0
0

```

A inicialização dos atributos com os valores padrão ocorre na instanciação, ou seja, quando o comando new é utilizado. Dessa forma, todo objeto “nasce” com os valores padrão. Em alguns casos, é necessário trocar esses valores. Para trocar o valor padrão de um atributo, devemos inicializá-lo na declaração. Por exemplo, suponha que o limite padrão das contas de um banco seja R\$ 500. Nesse caso, seria interessante definir esse valor como padrão para o atributo limite.



```

1 package com.mycompany.exemplosparaaulas;
2
3 public class Conta {
4     double saldo;
5     double limite = 500;
6     int numero;
7 }

```

```
1 package com.mycompany.exemplosparaaulas;
2
3 public class TestaConta {
4     public static void main(String[] args) {
5         //Instância do objeto conta em variáveis não
6         //primitivas
7         Conta referencia = new Conta();
8
9         //Mostrar no terminal os valores
10        System.out.println(referencia.saldo);
11        System.out.println(referencia.limite);
12        System.out.println(referencia.numero);
13    }
14 }
```

Output - Run (TestaConta) x

from pom.xml

[jar]-----

--- resources:3.3.1:resources (default-resources) @ ExemplosParaAulas ---

skip non existing resourceDirectory C:\Users\marco\Documents\NetBeansProjects\ExemplosParaAulas\src\main\resouro

--- compiler:3.13.0:compile (default-compile) @ ExemplosParaAulas ---

Recompiling the module because of changed source code.

Compiling 3 source files with javac [debug release 22] to target\classes

--- exec:3.13.0:exec (default-cli) @ ExemplosParaAulas ---

0.0

500.0

0

15. Algumas categorias de projetos que podemos trabalhar em Java

Podemos categorizar nossos projetos em Java e podemos falar sobre Java com Maven, Java com Gradle e Java com Ant, como algumas categorias que aparecem quando criamos um projeto novo no Apache Netbeans por exemplo, nestes casos, estamos tratando de ferramentas de *build automation* (automação de compilação) que são usadas para compilar, empacotar, testar, e implantar aplicativos Java de forma eficiente. Cada uma dessas ferramentas tem uma abordagem distinta, mas todas têm o objetivo de facilitar o processo de desenvolvimento e tornar a construção de projetos mais ágil e menos propensa a erros, para nossas aulas e práticas, a ideia é utilizarmos a Maven ou Ant. Vamos ver cada uma delas em detalhes:

15.1. Java com Maven

Maven é uma das ferramentas de build mais populares no ecossistema Java. Ele usa um modelo baseado em XML para descrever o ciclo de vida de construção do projeto.

15.1.1. Principais Características:

- **Dependências:** Maven facilita o gerenciamento de dependências. Ele se conecta a repositórios (como o Maven Central) e baixa automaticamente as bibliotecas necessárias para o seu projeto. Isso significa que você não precisa gerenciar manualmente os arquivos JAR e suas versões.
- **Arquitetura baseada em Plugins:** O Maven executa tarefas por meio de plugins (como compilação, testes e empacotamento) que são configurados no arquivo pom.xml (*Project Object Model*).
- **Ciclo de Vida:** O Maven segue um ciclo de vida definido, que inclui fases como *compile*, *test*, *package*, *install*, *deploy*. Cada uma dessas fases pode ser configurada para realizar tarefas específicas.

15.1.2. Vantagens do Maven:

- Gerenciamento de dependências automático.
- Repositórios centralizados (Maven Central).
- Integração fácil com servidores de build e CI/CD.
- Documentação e convenção forte (se você seguir as convenções do Maven, o projeto é fácil de entender).

15.1.3. Desvantagens:

- Complexidade em grandes projetos.
- XML pode ser verboso.

15.2. Java com Gradle

Gradle é uma ferramenta de automação de build moderna que tem ganhado popularidade nos últimos anos, principalmente por ser mais flexível e rápida que o Maven.

15.2.1. Principais Características:

- **Baseado em Groovy ou Kotlin:** Gradle usa uma linguagem de script (Groovy ou Kotlin DSL) para descrever o build, o que torna a configuração mais flexível e programática.
- **Desempenho:** Gradle é conhecido por ser mais rápido que o Maven e o Ant devido ao seu build incremental (onde ele apenas recompila partes do código que foram alteradas).
- **Gerenciamento de Dependências:** Assim como o Maven, Gradle também oferece fácil integração com repositórios (como Maven Central), mas com maior flexibilidade.
- **Configuração:** Gradle permite configurações mais complexas e personalizadas, sendo ideal para projetos mais complexos.

15.2.2. Vantagens do Gradle:

- Desempenho superior (builds incrementais e paralelismo).
- Flexibilidade: Configurações personalizadas e lógica programática.
- Suporte a múltiplas linguagens: Além de Java, Gradle pode ser usado com outros tipos de projeto (Android, Groovy, Scala, etc.).
- DSL mais conciso: Arquivos de configuração mais simples e expressivos que o Maven (usando Groovy ou Kotlin).

15.2.3. Desvantagens:

- Curva de aprendizado: A flexibilidade de Gradle pode tornar o processo de configuração um pouco mais complexo para iniciantes.
- Menos documentação tradicional (se comparado ao Maven, por exemplo).

15.3. Java com Ant

Ant é uma das ferramentas de build mais antigas para Java. Ao contrário do Maven e do Gradle, o Ant não segue uma convenção padrão para construir o projeto e não tem uma definição de ciclo de vida, o que significa que você tem que definir manualmente como o projeto deve ser construído.

15.3.1. Principais Características:

- **Arquivo build.xml:** O Ant usa um arquivo XML chamado build.xml para definir as tarefas de construção, como compilar código, empacotar JARs, executar testes e mais.
- **Sem Convenção:** Diferente do Maven, o Ant não segue convenções. O desenvolvedor tem total liberdade para definir como o build deve ocorrer.
- **Extensível:** Ant permite adicionar novos tipos de tarefas personalizadas com facilidade.

15.3.2. Vantagens do Ant:

- Extremamente flexível: Você tem controle total sobre o processo de build.
- Pode ser usado em projetos não-Java: Ant pode ser usado para outras linguagens também.
- Facilidade em adicionar tarefas personalizadas.

15.3.3. Desvantagens:

- Não tem ciclo de vida predefinido: O desenvolvedor deve definir manualmente como o processo de construção será.

- Mais difícil de gerenciar dependências: Ao contrário do Maven e Gradle, você precisa gerenciar dependências manualmente ou com plugins adicionais.
- Configuração verbosa: Arquivos build.xml podem ser grandes e difíceis de manter à medida que o projeto cresce.

15.4. Comparação resumida

Característica	Maven	Gradle	Ant
Configuração	Baseado em XML (pom.xml)	Baseado em Groovy ou Kotlin (build.gradle)	Baseado em XML (build.xml)
Gerenciamento de dependências	Sim, integrado com repositórios Maven	Sim, mas mais flexível que o Maven	Necessário configuração manual
Desempenho	Médio (builds não incrementais)	Rápido (builds incrementais e paralelismo)	Médio, depende de como é configurado
Facilidade de uso	Fácil para iniciantes, com convenções fortes	Maior curva de aprendizado, mas flexível	Flexível, mas sem ciclo de vida predefinido
Popularidade	Muito popular, especialmente em grandes empresas	Crescendo rapidamente, especialmente para Android	Menos usado atualmente, mas ainda popular em projetos antigos

Tabela de comparação entre categorias

16. Definição e Instalação do JDK e NetBeans

16.1. O que é o JDK (Java Development Kit)?

O JDK (Java Development Kit) é o kit de desenvolvimento oficial para a linguagem de programação Java, fornecido pela Oracle Corporation e outras distribuidoras como *OpenJDK* e Amazon Corretto. Ele contém todas as ferramentas necessárias para escrever, compilar e executar programas em Java.

16.1.1. Componentes do JDK

O JDK é composto por vários elementos essenciais para o desenvolvimento Java. Abaixo estão os principais:

1. JRE (Java Runtime Environment)

O JRE é um subconjunto do JDK responsável apenas pela execução de programas Java. Ele inclui:

- Java Virtual Machine (JVM) → Executa *bytecode* Java.
- Bibliotecas Padrão (API Java) → Conjunto de classes e métodos prontos para uso.
- Ferramentas de suporte → Como carregamento de classes e gerenciamento de memória.

Importante: Se você apenas deseja executar programas Java (**e não desenvolver**), basta instalar o JRE em vez do JDK.

2. JVM (Java Virtual Machine)

A JVM é o ambiente que permite que programas Java sejam executados em diferentes sistemas operacionais. Ela funciona como um interpretador do *bytecode* (código intermediário gerado pelo compilador Java), tornando o Java uma linguagem portátil e independente de plataforma.

Princípio do Java: “Write Once, Run Anywhere” (Escreva uma vez, execute em qualquer lugar).

3. Ferramentas do JDK

O JDK também inclui um conjunto de ferramentas para facilitar o desenvolvimento:

Ferramenta	Função
javac	Compilador Java (transforma código-fonte <code>.java</code> em <i>bytecode</i> <code>.class</code>)
java	Interpretador que executa programas Java usando a JVM
jar	Empacotador de arquivos <code>.class</code> em um único <code>.jar</code>

16.2. Baixar e Instalar o JDK

Passo 1: Baixar o JDK

- Acesse o site oficial da Oracle: <https://www.oracle.com/java/technologies/javase-downloads.html>
- Escolha a versão mais recente do JDK.
- Na seção Windows, clique em Installer (.exe) para baixar o arquivo de instalação.

Passo 2: Instalar o JDK

- Execute o arquivo .exe baixado.
- Clique em Next para iniciar a instalação.
- Escolha um diretório de instalação (ou deixe o padrão).
- Clique em Next e aguarde a instalação.
- Ao finalizar, clique em Close.

Passo 3: Testar a instalação do Java

- Abra o Prompt de Comando (Win + R → digite cmd e pressione Enter).
- Digite o comando: java -version
- Se aparecer a versão do Java, significa que a instalação foi bem-sucedida.

16.3. Baixar e Instalar o NetBeans

Passo 1: Baixar o NetBeans

- Acesse o site oficial: <https://netbeans.apache.org/download/index.html>
- Escolha a versão mais recente e clique em **Download**.
- Baixe o instalador para Windows (**EXE Installer**).

Passo 2: Instalar o NetBeans

- Execute o arquivo .exe baixado.
- Clique em **Next**.
- Aceite os termos da licença e clique em **Next**.
- Escolha o diretório de instalação (ou deixe o padrão).
- Certifique-se de que o caminho do JDK foi detectado corretamente.
- Clique em **Install** e aguarde a instalação.
- Após a conclusão, clique em **Finish**.

Passo 3: Testar o NetBeans

- Abra o NetBeans pelo menu Iniciar.
- Clique em **File** → **New Project**.
- Escolha **Java with Maven** → **Java Application** e clique em **Next**.
- Dê um nome ao projeto e clique em **Finish**.
- No editor, escreva um código de teste:

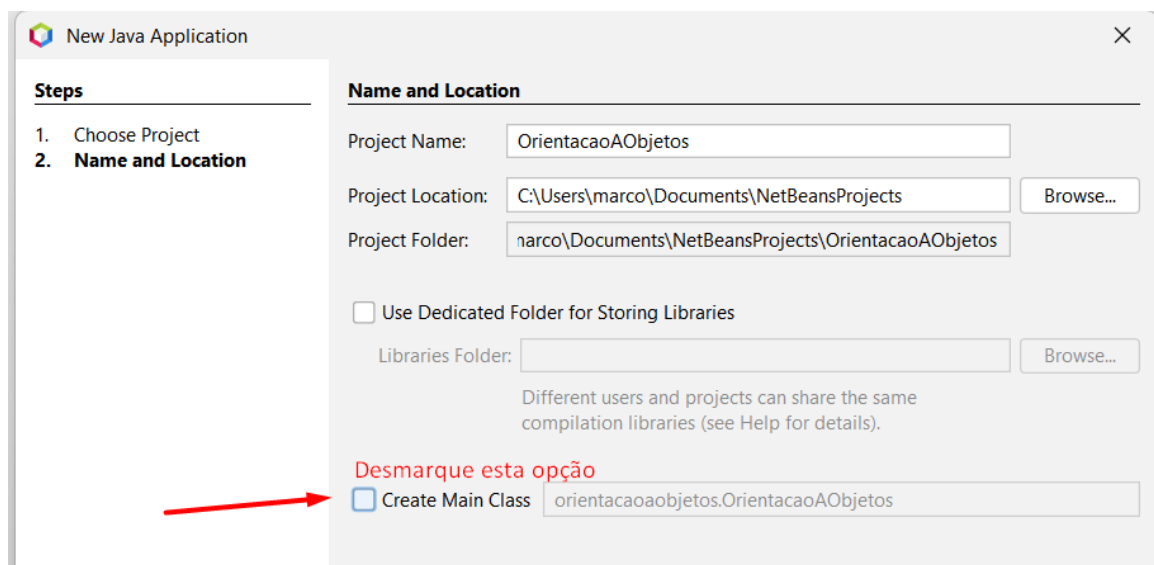
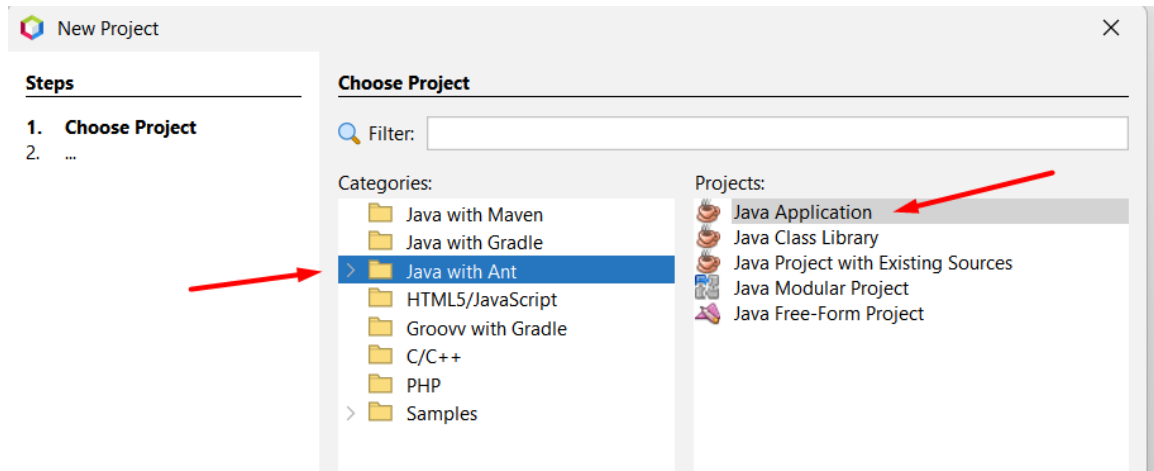
```
public class Main {  
    public static void main(String[] args) {  
        System.out.println("Olá, NetBeans!");  
    }  
}
```

- Pressione **F6** para executar. Se aparecer "Olá, NetBeans!" no console, está tudo certo!

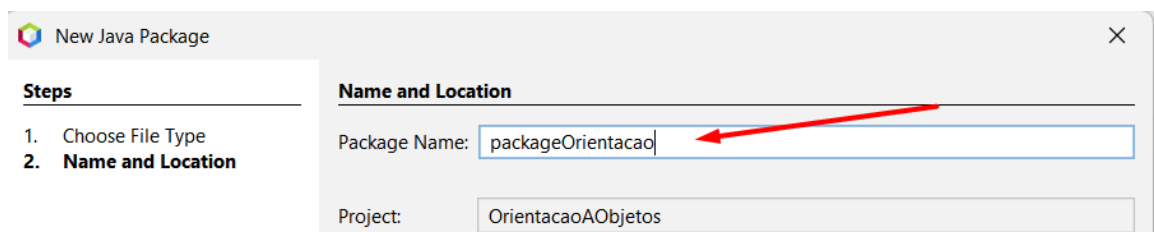
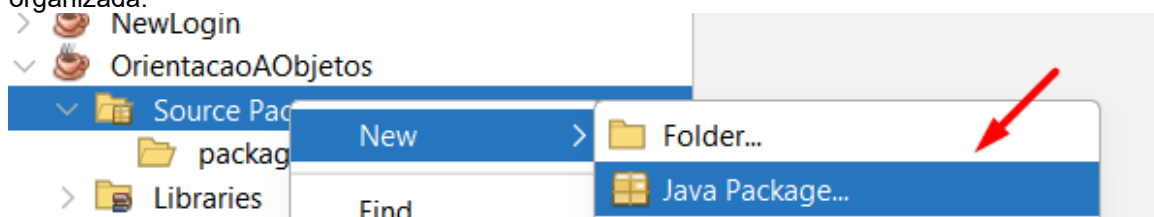
17. Exercícios de Fixação

A seguir, iremos realizar este exercício para fixação de conteúdo, portanto, vocês devem codificar o passo a passo a seguir e verificar os resultados obtidos, como sempre falo, vocês podem alterar ou melhorar o código, isso é importante para que consigam ratificar o conteúdo visto até o momento, como informei anteriormente vocês podem utilizar o NetBeans ou a ferramenta que acharem melhor, porém, nas aulas aqui, abordarei o NetBeans.

1 – Para isso crie um novo projeto chamado **OrientacaoAObjetos** para os arquivos desenvolvidos neste exercício.

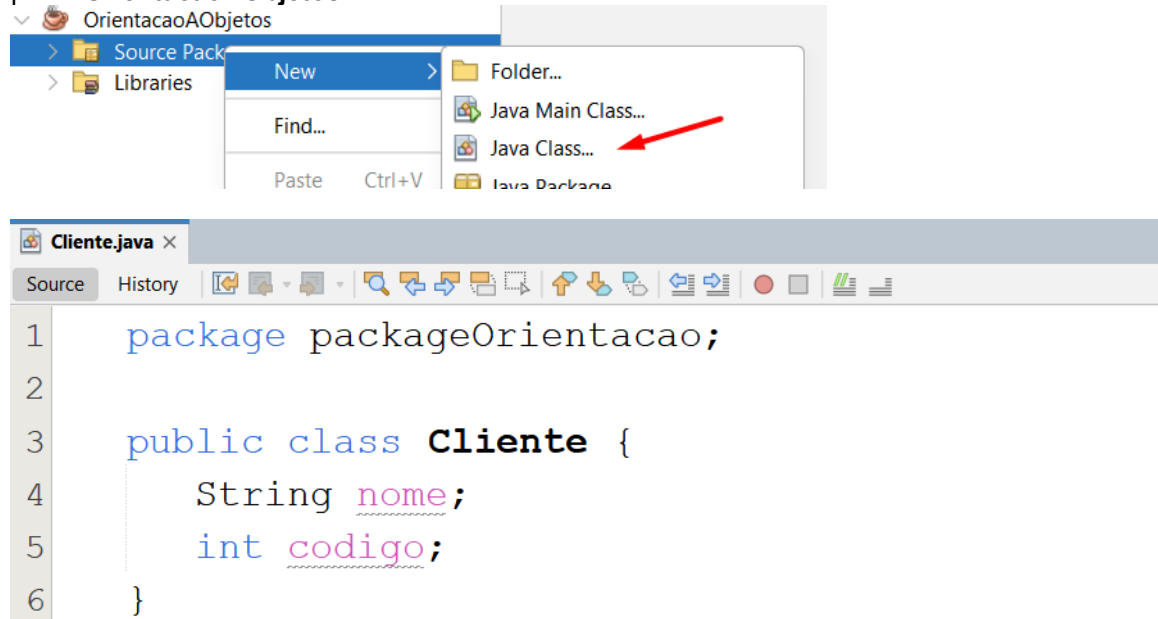


2 – Vamos criar um pacote para armazenar as classes de nosso projeto de forma organizada.

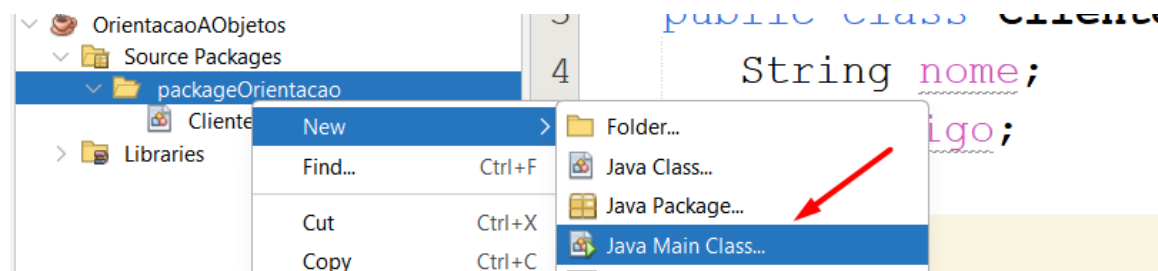


3 - Implemente uma classe para definir os objetos que representarão os clientes de um banco. Mas antes disso, vamos implementar o pacote da classe Essa classe deve declarar dois

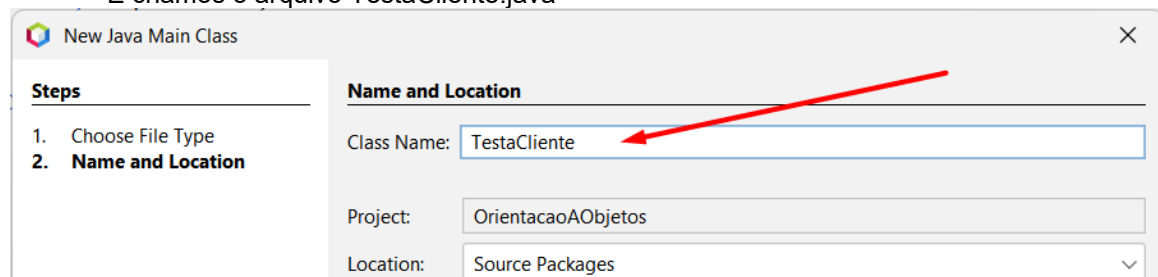
atributos: um para os nomes e outro para os códigos dos clientes. Adicione o seguinte arquivo na pasta **OrientacaoAObjetos**

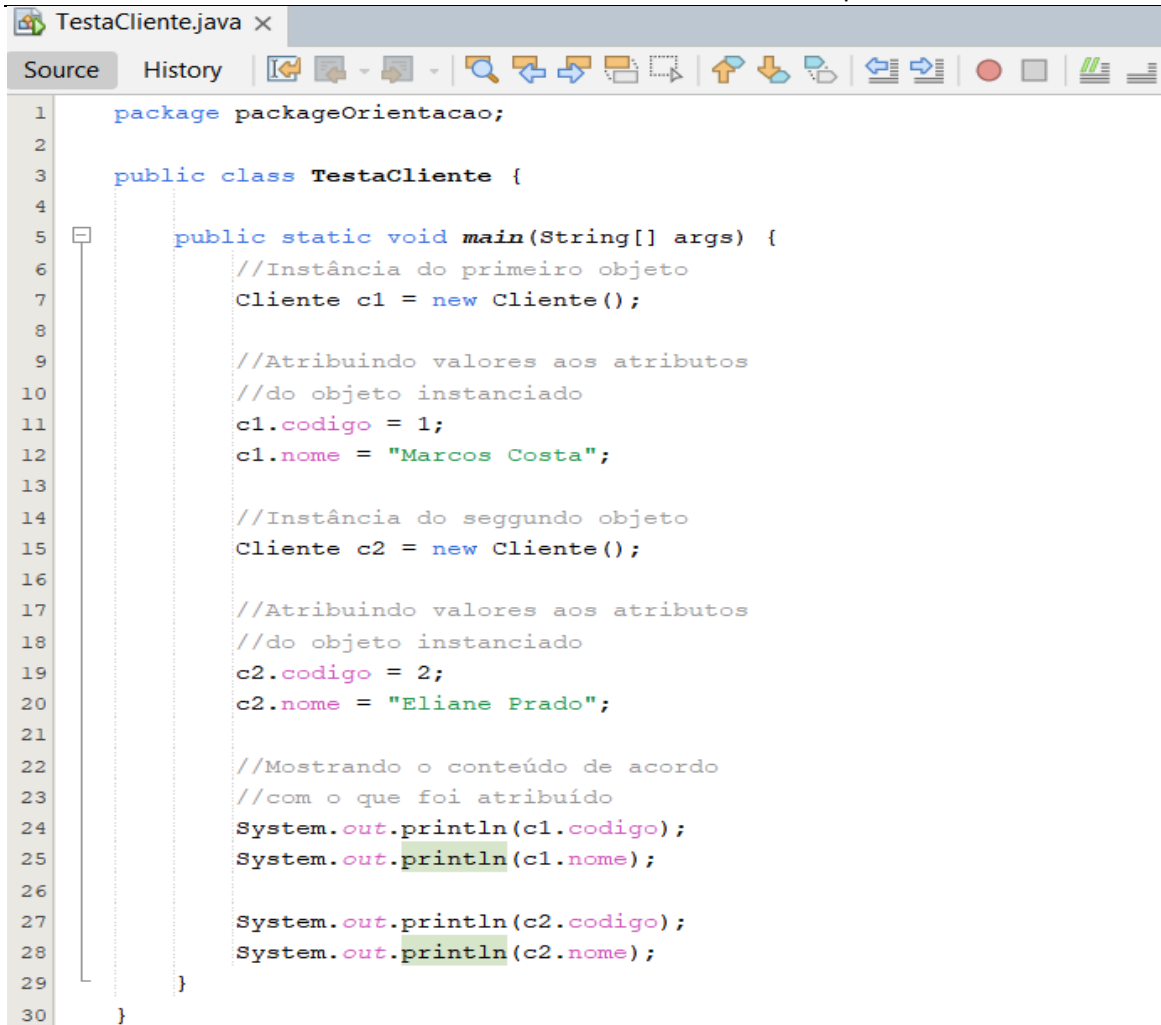


4 - Faça um teste criando dois objetos da classe Cliente. Vamos adicionar agora uma Java Main Class, adicione o seguinte arquivo na pasta **OrientacaoAObjetos**.



E criamos o arquivo TestaCliente.java



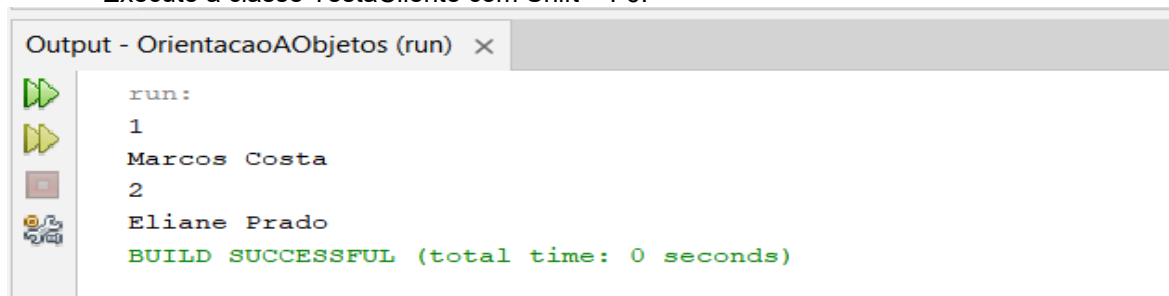


```

1  package packageOrientacao;
2
3  public class TestaCliente {
4
5      public static void main(String[] args) {
6          //Instância do primeiro objeto
7          Cliente c1 = new Cliente();
8
9          //Atribuindo valores aos atributos
10         //do objeto instanciado
11         c1.codigo = 1;
12         c1.nome = "Marcos Costa";
13
14         //Instância do segundo objeto
15         Cliente c2 = new Cliente();
16
17         //Atribuindo valores aos atributos
18         //do objeto instanciado
19         c2.codigo = 2;
20         c2.nome = "Eliane Prado";
21
22         //Mostrando o conteúdo de acordo
23         //com o que foi atribuído
24         System.out.println(c1.codigo);
25         System.out.println(c1.nome);
26
27         System.out.println(c2.codigo);
28         System.out.println(c2.nome);
29     }
30 }

```

Execute a classe TestaCliente com Shift + F6.



```

Output - OrientacaoAObjetos (run) x
run:
1
Marcos Costa
2
Eliane Prado
BUILD SUCCESSFUL (total time: 0 seconds)

```

5 - Os bancos oferecem aos clientes a possibilidade de obter um cartão de crédito que pode ser utilizados para fazer compras. Um cartão de crédito possui um número e uma data de validade. Crie uma classe (Java Class) para modelar os objetos que representarão os cartões de crédito. Adicione o seguinte arquivo na pasta **OrientacaoAObjetos**.

```

1 package packageOrientacao;
2
3 public class CartaoDeCredito {
4     int numero;
5     String dataDeValidade;
6 }

```

6 - Faça um teste criando dois objetos da classe CartaoDeCredito. Altere e imprima os atributos desses objetos. Adicione o seguinte arquivo TestaCartaoDeCredito (Java Main Class) na pasta **OrientacaoAObjetos**.

```

1 package packageOrientacao;
2
3 public class TestaCartaoDeCredito {
4
5     public static void main(String[] args) {
6         //Instância do primeiro objeto
7         CartaoDeCredito cdc1 = new CartaoDeCredito();
8
9         //Atribuindo valores aos atributos
10        //do objeto instanciado
11        cdc1.numero = 1111;
12        cdc1.dataDeValidade = "01/01/2025";
13
14        //Atribuindo valores aos atributos
15        //do objeto instanciado
16        CartaoDeCredito cdc2 = new CartaoDeCredito();
17        cdc2.numero = 2222;
18        cdc2.dataDeValidade = "01/01/2027";
19
20        //Mostrando o conteúdo de acordo
21        //com o que foi atribuído
22        System.out.println(cdc1.numero);
23        System.out.println(cdc1.dataDeValidade);
24
25        System.out.println(cdc2.numero);
26        System.out.println(cdc2.dataDeValidade);
27    }
28
29 }

```

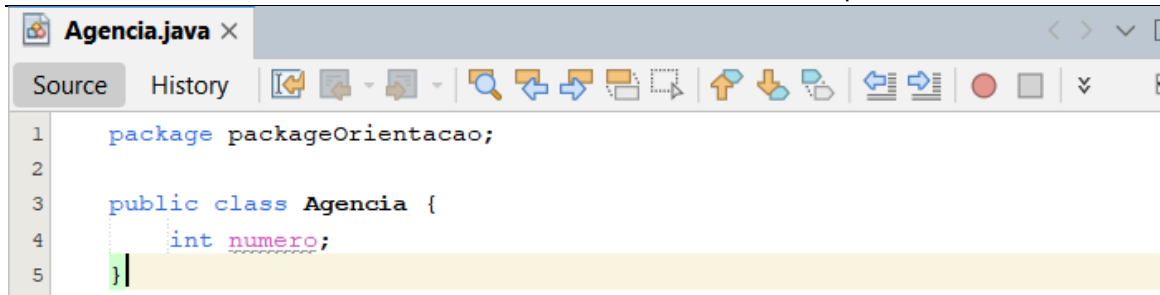
Execute a classe TestaCartaoDeCredito.

```

Output - OrientacaoAObjetos (run) x
run:
1111
01/01/2025
2222
01/01/2027
BUILD SUCCESSFUL (total time: 0 seconds)

```

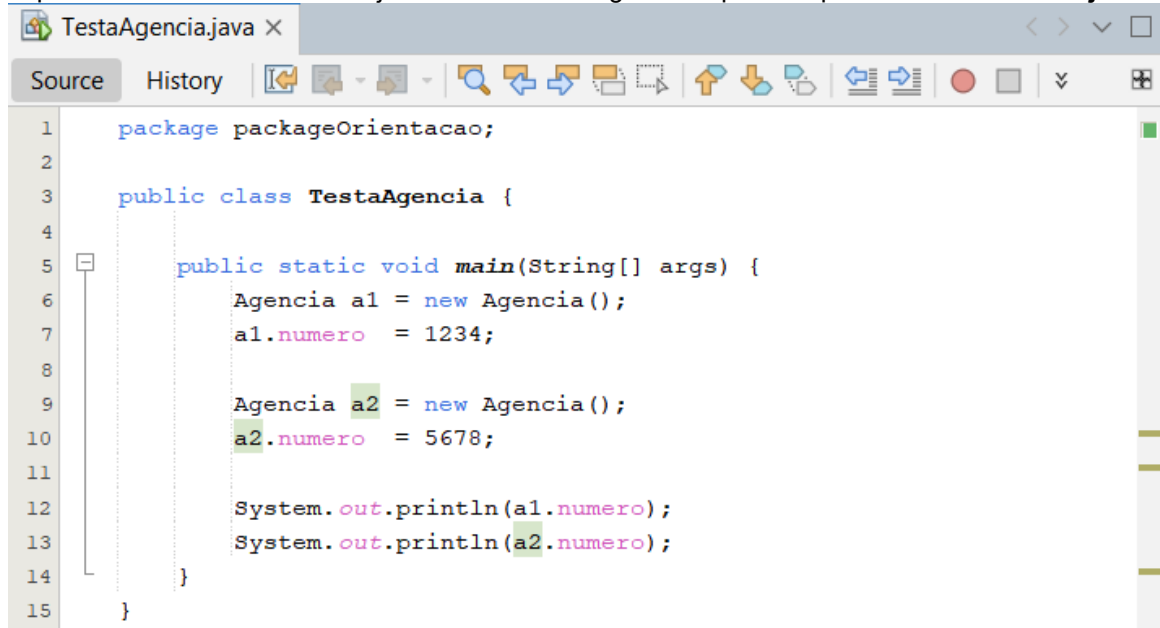
7 - As agências do banco possuem número. Crie uma classe(Java Class) para definir os objetos que representarão as agências.



```

1 package packageOrientacao;
2
3 public class Agencia {
4     int numero;
5 }
  
```

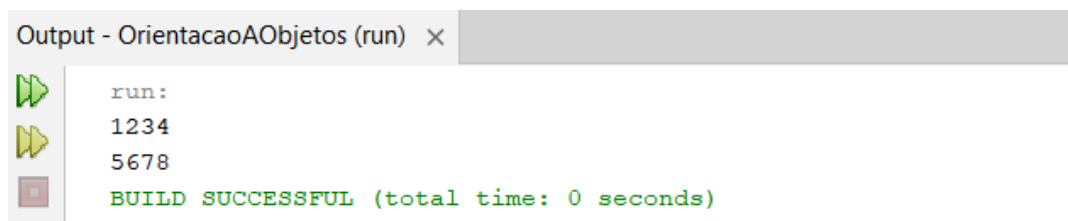
8 - Faça um teste(Java Main Class) criando dois objetos da classe Agencia. Altere e imprima os atributos desses objetos. Adicione o seguinte arquivo na pasta **OrientacaoAObjetos**.



```

1 package packageOrientacao;
2
3 public class TestaAgencia {
4
5     public static void main(String[] args) {
6         Agencia a1 = new Agencia();
7         a1.numero = 1234;
8
9         Agencia a2 = new Agencia();
10        a2.numero = 5678;
11
12        System.out.println(a1.numero);
13        System.out.println(a2.numero);
14    }
15 }
  
```

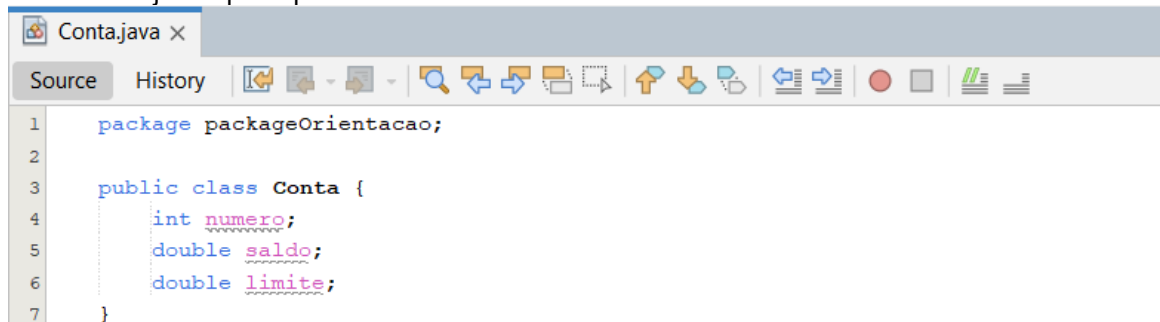
Execute a classe TestaAgencia.



```

Output - OrientacaoAObjetos (run) x
run:
1234
5678
BUILD SUCCESSFUL (total time: 0 seconds)
  
```

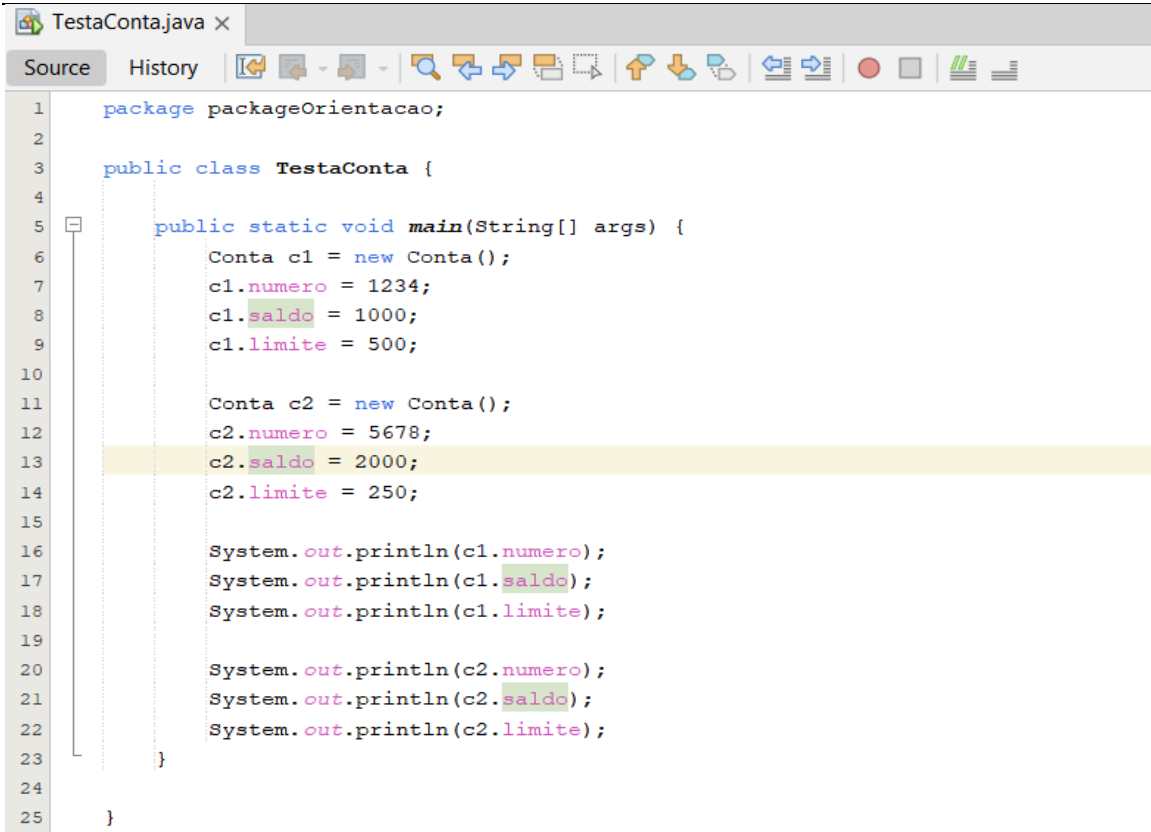
9 - As contas do banco possuem número, saldo e limite. Crie uma classe(Java Class) para definir os objetos que representarão as contas.



```

1 package packageOrientacao;
2
3 public class Conta {
4     int numero;
5     double saldo;
6     double limite;
7 }
  
```

10 - Faça um teste criando dois objetos da classe Conta(Java Main Class). Altere e imprima os atributos desses objetos. Adicione o seguinte arquivo na pasta **OrientacaoAObjetos**.

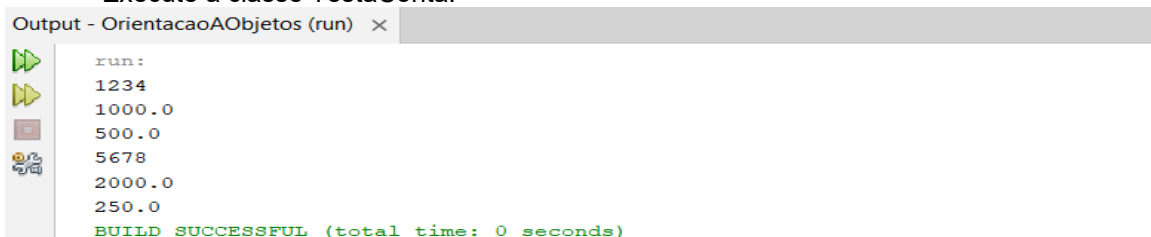


```

1 package packageOrientacao;
2
3 public class TestaConta {
4
5     public static void main(String[] args) {
6         Conta c1 = new Conta();
7         c1.numero = 1234;
8         c1.saldo = 1000;
9         c1.limite = 500;
10
11         Conta c2 = new Conta();
12         c2.numero = 5678;
13         c2.saldo = 2000;
14         c2.limite = 250;
15
16         System.out.println(c1.numero);
17         System.out.println(c1.saldo);
18         System.out.println(c1.limite);
19
20         System.out.println(c2.numero);
21         System.out.println(c2.saldo);
22         System.out.println(c2.limite);
23     }
24 }
25

```

Execute a classe TestaConta.

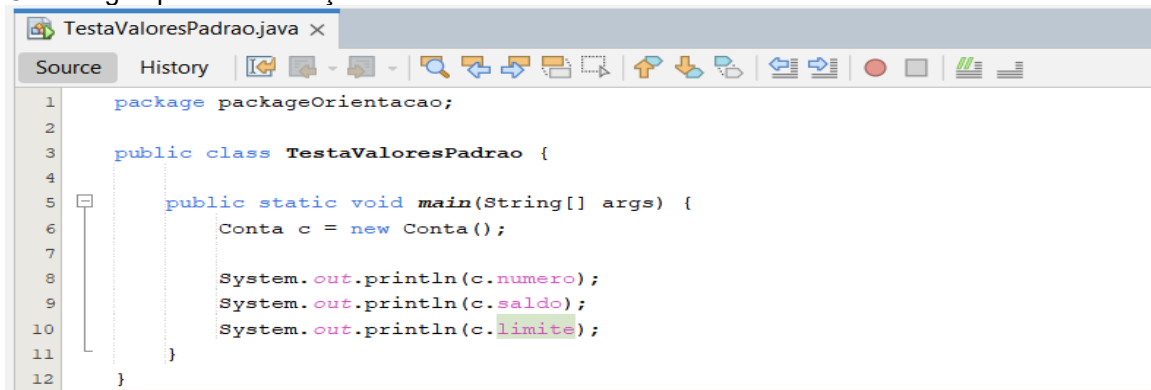


```

run:
1234
1000.0
500.0
5678
2000.0
250.0
BUILD SUCCESSFUL (total time: 0 seconds)

```

11 - Faça um teste(Java Main Class) que imprima os atributos de um objeto da classe Conta logo após a sua criação.

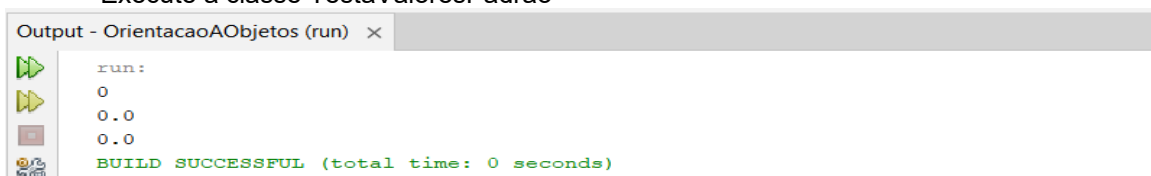


```

1 package packageOrientacao;
2
3 public class TestaValoresPadrao {
4
5     public static void main(String[] args) {
6         Conta c = new Conta();
7
8         System.out.println(c.numero);
9         System.out.println(c.saldo);
10        System.out.println(c.limite);
11    }
12

```

Execute a classe TestaValoresPadrao

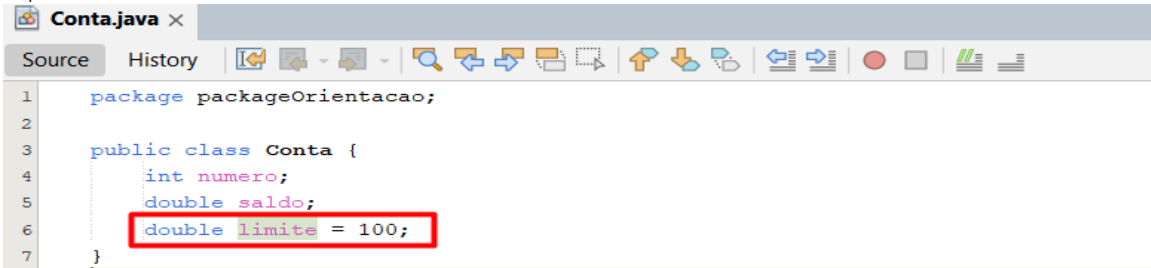


```

run:
0
0.0
0.0
BUILD SUCCESSFUL (total time: 0 seconds)

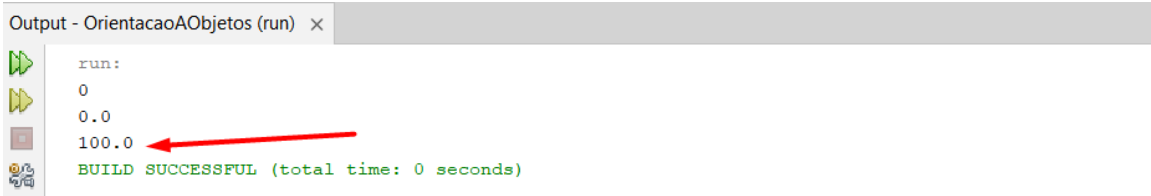
```

12. Altere a classe Conta para que todos os objetos criados a partir dessa classe possuam R\$ 100 de limite inicial.



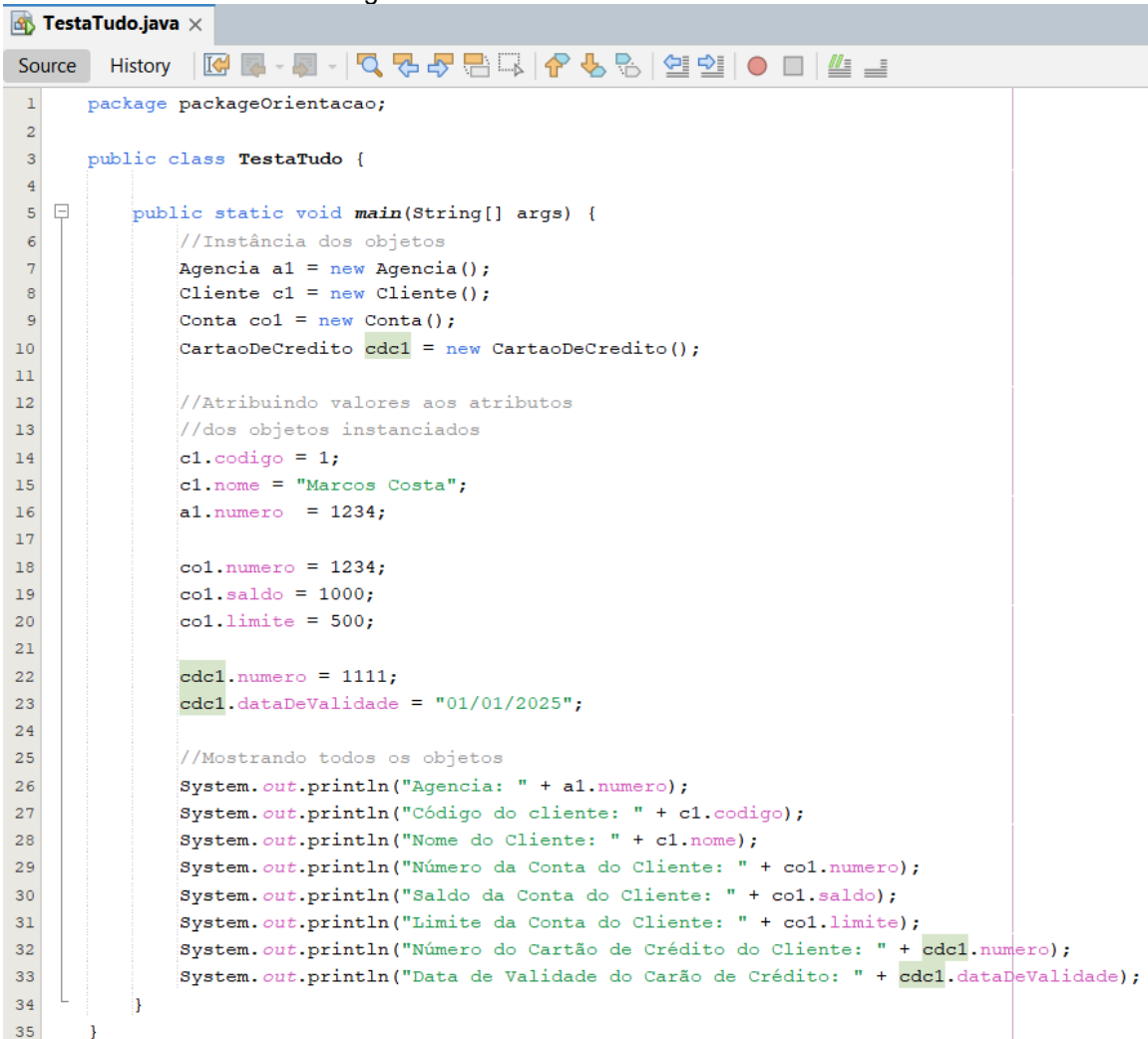
```
1 package packageOrientacao;
2
3 public class Conta {
4     int numero;
5     double saldo;
6     double limite = 100;
7 }
```

Execute a classe TestavaloresPadrao.



```
run:
0
0.0
100.0
BUILD SUCCESSFUL (total time: 0 seconds)
```

13. Criaremos uma classe Testatudo(Java Main Class) para criarmos todos os objetos, e alocarmos as chamadas e integrar todos.



```
1 package packageOrientacao;
2
3 public class Testatudo {
4
5     public static void main(String[] args) {
6         //Instância dos objetos
7         Agencia a1 = new Agencia();
8         Cliente c1 = new Cliente();
9         Conta co1 = new Conta();
10        CartaoDeCredito cdc1 = new CartaoDeCredito();
11
12        //Atribuindo valores aos atributos
13        //dos objetos instanciados
14        c1.codigo = 1;
15        c1.nome = "Marcos Costa";
16        a1.numero = 1234;
17
18        co1.numero = 1234;
19        co1.saldo = 1000;
20        co1.limite = 500;
21
22        cdc1.numero = 1111;
23        cdc1.dataDeValidade = "01/01/2025";
24
25        //Mostrando todos os objetos
26        System.out.println("Agencia: " + a1.numero);
27        System.out.println("Código do cliente: " + c1.codigo);
28        System.out.println("Nome do Cliente: " + c1.nome);
29        System.out.println("Número da Conta do Cliente: " + co1.numero);
30        System.out.println("Saldo da Conta do Cliente: " + co1.saldo);
31        System.out.println("Limite da Conta do Cliente: " + co1.limite);
32        System.out.println("Número do Cartão de Crédito do Cliente: " + cdc1.numero);
33        System.out.println("Data de Validade do Carão de Crédito: " + cdc1.dataDeValidade);
34    }
35 }
```


Executando o resultado fica assim:

```
Output - OrientacaoAObjetos (run) X
run:
Agencia: 1234
Codigo do cliente: 1
Nome do Cliente: Marcos Costa
Numero da Conta do Cliente: 1234
Saldo da Conta do Cliente: 1000.0
Limite da Conta do Cliente: 500.0
Numero do Cartão de Crédito do Cliente: 1111
Data de Validade do Cartão de Crédito: 01/01/2025
BUILD SUCCESSFUL (total time: 0 seconds)
```

Agora é com vocês, para praticar, instalem e desenvolvam este exercício.

18. Métodos

No banco, é possível realizar diversas operações em uma conta: depósito, saque, transferência, consultas e etc. Essas operações podem modificar ou apenas acessar os valores dos atributos dos objetos que representam as contas.

Essas operações são realizadas em métodos definidos na própria classe Conta. Por exemplo, para realizar a operação de depósito, podemos acrescentar o seguinte método na classe Conta.

```
1 void deposita(double valor) {
2     // implementação
3 }
```

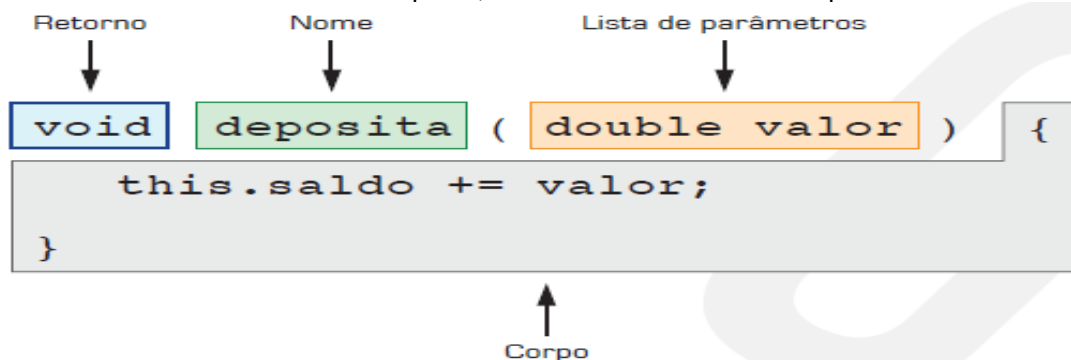
Podemos dividir um método em quatro partes:

Nome: É utilizado para chamar o método. Na linguagem Java, é uma boa prática definir os nomes dos métodos utilizando a convenção “Camel Case” com a primeira letra minúscula.

Lista de Parâmetros: Define os valores que o método deve receber. Métodos que não devem receber nenhum valor possuem a lista de parâmetros vazia.

Corpo: Define o que acontecerá quando o método for chamado.

Retorno: A resposta que será devolvida ao final do processamento do método. Quando um método não devolve nenhuma resposta, ele deve ser marcado como palavra reservada void.



Para realizar um depósito, devemos chamar o método `deposita()` através da referência do objeto que representa a conta que terá o dinheiro creditado.

```
//Referencia de um objeto
```

```
Conta c = new Conta();
```

```
//Chamando o método deposita()
```

```
c.deposita(1000);
```

Normalmente, os métodos acessam ou alteram os valores armazenados nos atributos dos objetos.

Por exemplo, na execução do método `deposita()`, é necessário alterar o valor do atributo `saldo` do objeto que foi escolhido para realizar a operação.

Dentro de um método, para acessar os atributos do objeto que está processando o método, devemos utilizar a palavra reservada `this`.

```
void deposita (double valor){
    this.saldo += valor;
}
```

O método `deposita()` não possui nenhum retorno lógico. Por isso, foi marcado com `void`. Mas, para outros métodos, pode ser necessário definir um tipo de retorno específico.

Considere, por exemplo, um método para realizar a operação que consulta o saldo disponível das contas. Suponha também que o saldo disponível é igual a soma do saldo e do limite. Então, esse método deve somar os atributos `saldo` e `limite` e devolver o resultado. Por outro lado, esse método não deve receber nenhum valor, pois todas as informações necessárias para realizar a operação estão nos atributos dos objetos que representam as contas.

```
double consultaSaldoDisponivel () {
    return this.saldo + this.limite;
}
```

Ao chamar o método `consultaSaldoDisponivel()` a resposta pode ser armazenada em uma variável do tipo `double`.

```
//Referencia de um objeto
```

```
Conta c = new Conta();
```

```
//Armazenando a resposta de um método em variável
```

```
double saldoDisponivel = c.consultaSaldoDisponivel();
```

```
System.out.println("Saldo disponível: " + saldoDisponivel);
```

19. Exercícios de Fixação

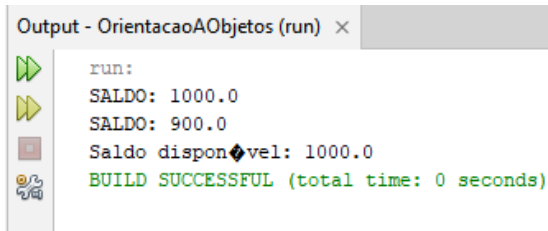
1-) Acrescente alguns métodos na classe `Conta` (Java Class) para realizar as operações de depósito, saque, impressão de extrato e consulta do saldo disponível.

```
1 package packageOrientacao;
2
3 public class Conta {
4     int numero ;
5     double saldo ;
6     double limite = 100;
7
8     void deposita (double valor){
9         this.saldo += valor;
10    }
11
12    void saca (double valor){
13        this.saldo -= valor;
14    }
15
16    void imprimeExtrato () {
17        System.out.println("SALDO: " + this.saldo);
18    }
19
20    double consultaSaldoDisponivel () {
21        return this.saldo + this.limite;
22    }
23 }
```

2-) Teste os métodos da classe Conta (Java Main Class), para isso crie:

```
1 package packageOrientacao;  
2  
3 public class TestaMetodosConta {  
4  
5     public static void main(String[] args) {  
6         //Referencia de um objeto  
7         Conta c = new Conta();  
8  
9         //Chamando o método deposita()  
10        c.deposita(1000);  
11        c.imprimeExtrato();  
12  
13        c.saca(100);  
14        c.imprimeExtrato();  
15  
16        //Armazenando a resposta de um método em variável  
17        double saldoDisponivel = c.consultaSaldoDisponivel();  
18        System.out.println("Saldo disponível: " + saldoDisponivel);  
19    }  
20 }
```

Teremos este resultado:



```
Output - OrientacaoAObjetos (run) x  
run:  
SALDO: 1000.0  
SALDO: 900.0  
Saldo disponível: 1000.0  
BUILD SUCCESSFUL (total time: 0 seconds)
```

20. Exercícios Complementares

Para esta atividade, crie um novo projeto que pode ser chamado de ExercicioMetodos, execute os passos seguintes, após isso, submeta o projeto ao GitHub, e nas tarefas no Microsoft Teams, informem o link do repositório. Prazo, até o dia 28/08/2025, esta atividade servirá para caso no final do semestre, tenhamos alguém precisando de nota e contará como participação e ajudará na mesma. Na aula seguinte, faremos a implementação do mesmo e continuidade com a matéria.

1-) Em uma empresa, temos os funcionários, que precisam ser representados em nossa aplicação. Então implemente uma classe chamada Funcionario que contenha dois atributos: o primeiro para o nome e o segundo para o salário dos funcionários.

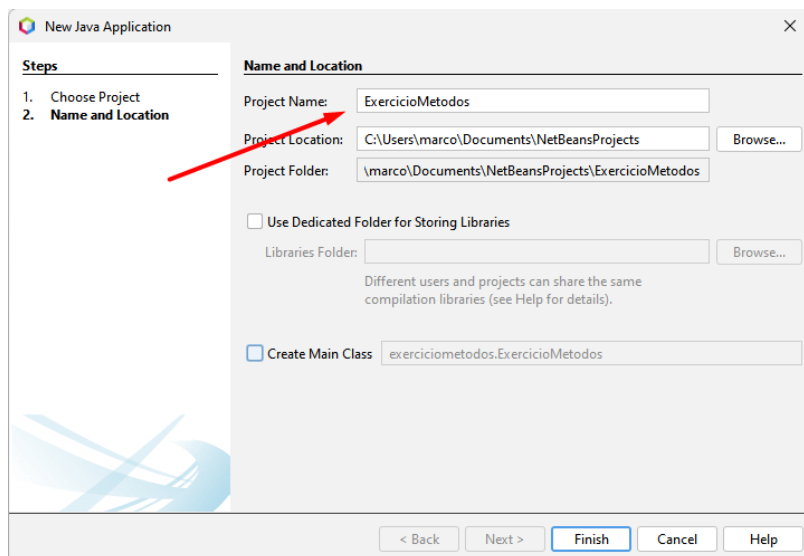
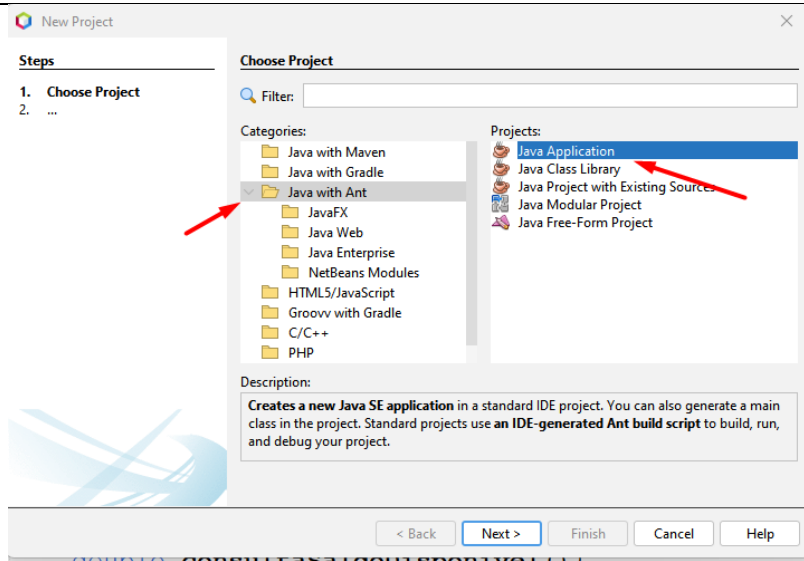
2-) Faça uma classe chamada TestaFuncionario e crie dois objetos da classe Funcionario atribuindo valores a eles. Mostre na tela as informações desses objetos.

3-) Adicione na classe Funcionario dois métodos: um para aumentar o salário e outro para consultar os dados dos funcionários.

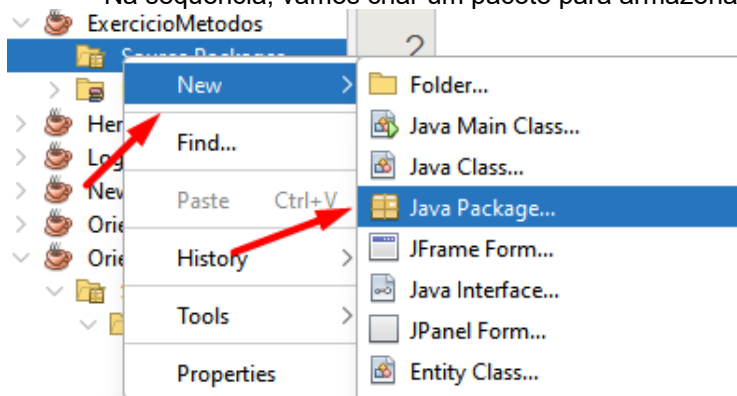
4-) Na classe TestaFuncionario teste novamente os métodos de um objeto da classe Funcionario.

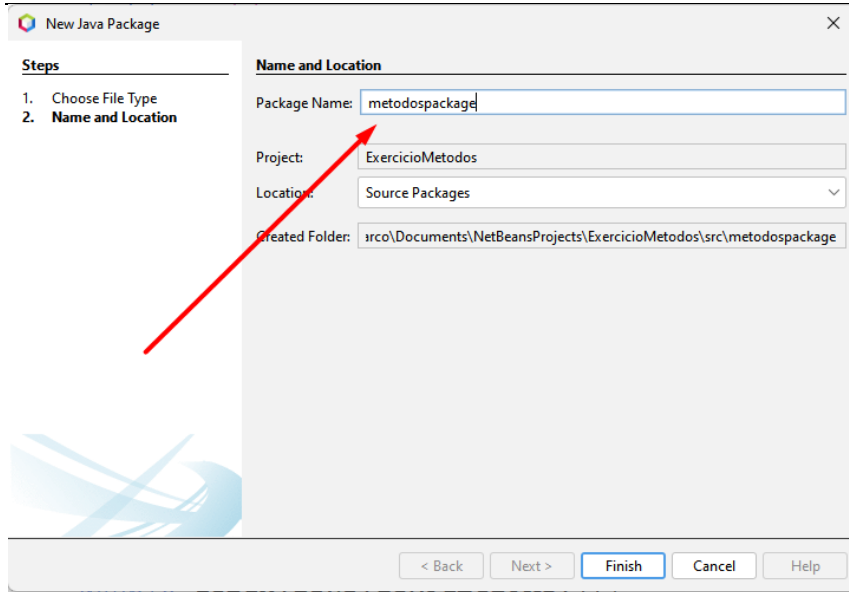
20.1. Correção dos Exercícios Complementares

A primeira atividade de nosso exercício, é a criação do projeto, portanto, devemos:

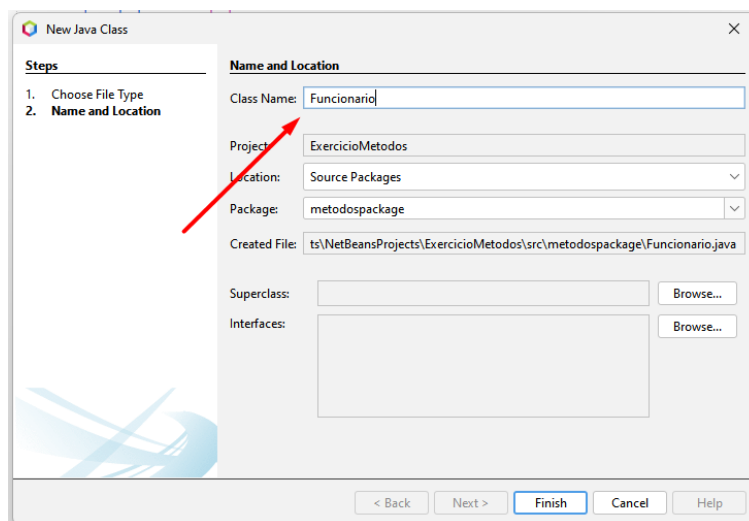
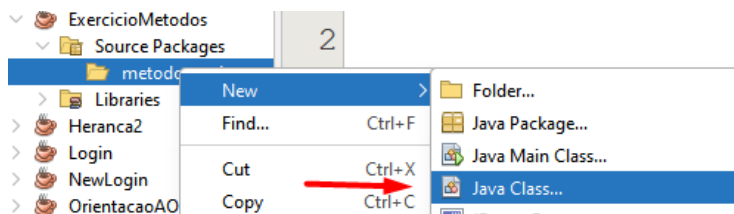


Na sequência, vamos criar um pacote para armazenar nossos arquivos.

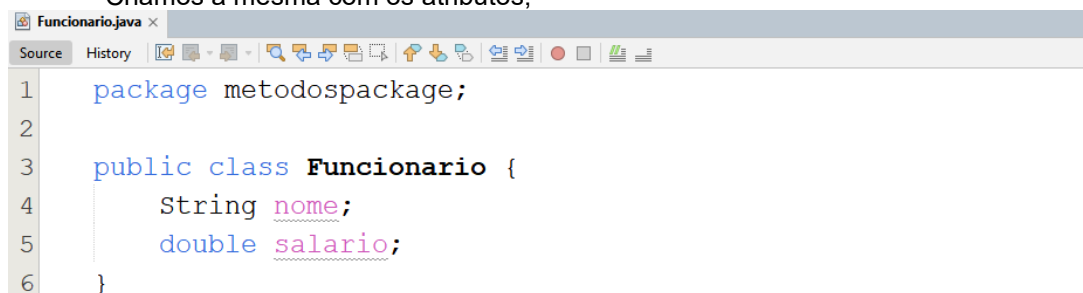




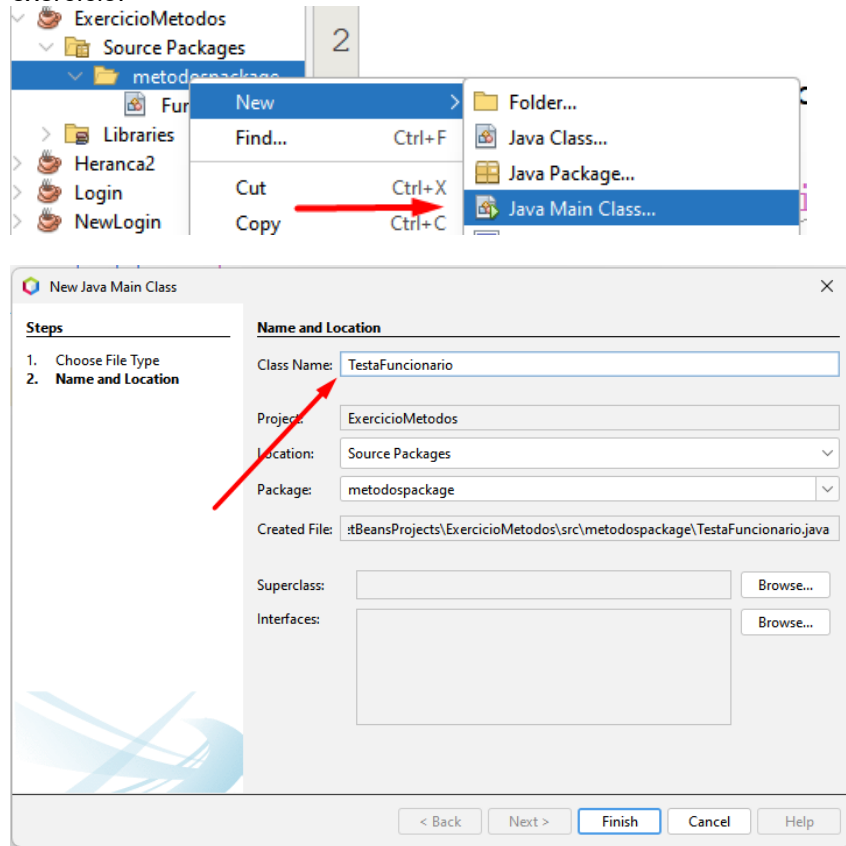
No primeiro exercício, vamos implementar a classe Funcionario (*Java Class*) e os atributos solicitados.



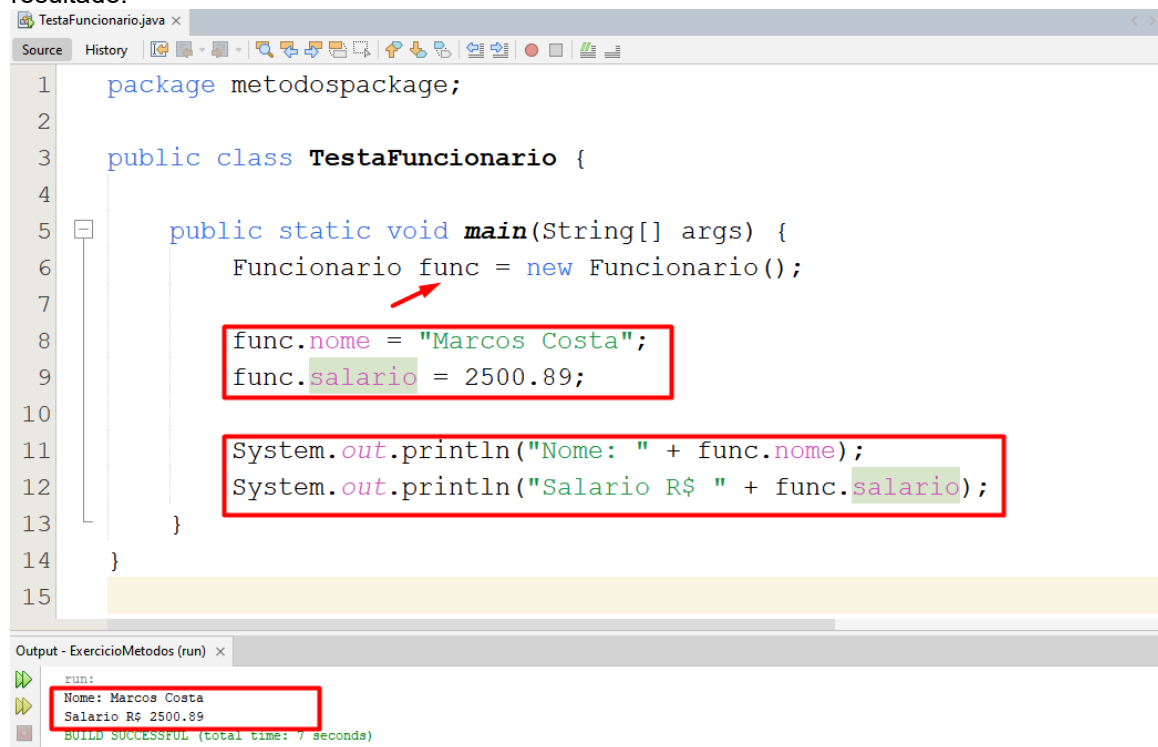
Criamos a mesma com os atributos;



Agora, dando continuidade, vamos criar uma *TestaFuncionario (Java Main Class)* para instanciar o objeto e fazer a inclusão dos dados nos atributos e os mostrar, conforme pedido no exercício:



Agora, codificamos conforme pedido no item 2 de nosso exercício e mostramos o resultado:



Continuando nosso exercício, agora é pedido a criação de dois métodos na classe Funcionario, um para aumento de salário, neste caso, vou fazer um cálculo com um percentual que será passado, e outro método que mostrará os dados do funcionário, ou seja, a própria classe Funcionario, irá ter um método para mostrar, ao invés de fazermos isso na *Java Main Class*.

```
Funcionario.java x
Source History
1 package metodospackage;
2
3 public class Funcionario {
4     String nome;
5     double salario;
6
7     void aumentaSalario(double percAumento) {
8         this.salario = this.salario + (this.salario * percAumento);
9     }
10
11     void mostraDadosFuncionario() {
12         System.out.println("Nome: " + this.nome);
13         System.out.println("Salario R$ " + this.salario);
14     }
15 }
```

Agora, a ideia é alterarmos a *Java Main Class* TestaFuncionario, atribuindo as execuções, conforme solicitado no item 4 de nossa atividade.

```
TestaFuncionario.java x
Source History
1 package metodospackage;
2
3 public class TestaFuncionario {
4
5     public static void main(String[] args) {
6         Funcionario func = new Funcionario();
7
8         func.nome = "Marcos Costa";
9         func.salario = 2500.89;
10
11         func.aumentaSalario(10);
12         func.mostraDadosFuncionario();
13     }
14 }
```

Output - ExercicioMetodos (run) x

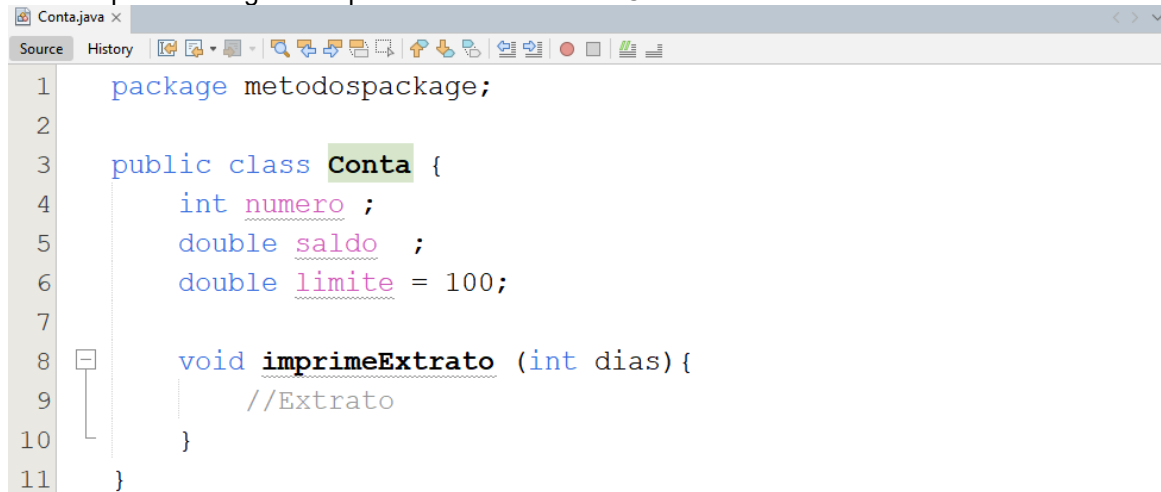
```
run:
Nome: Marcos Costa
Salario R$ 27509.789999999997
BUILD SUCCESSFUL (total time: 0 seconds)
```

Com isso, finalizamos esta correção, sempre lembro que temos outras formas de resolver este mesmo exercício, o que importa é a implementação do que é solicitado no enunciado do mesmo, portanto, se fez de uma forma diferente, mas atingiu o mesmo objetivo, parabéns, este é o caminho.

21. Sobrecarga (Overloading)

Voltando ao nosso conceito, os clientes dos bancos costumam consultar periodicamente informações relativas às suas contas.

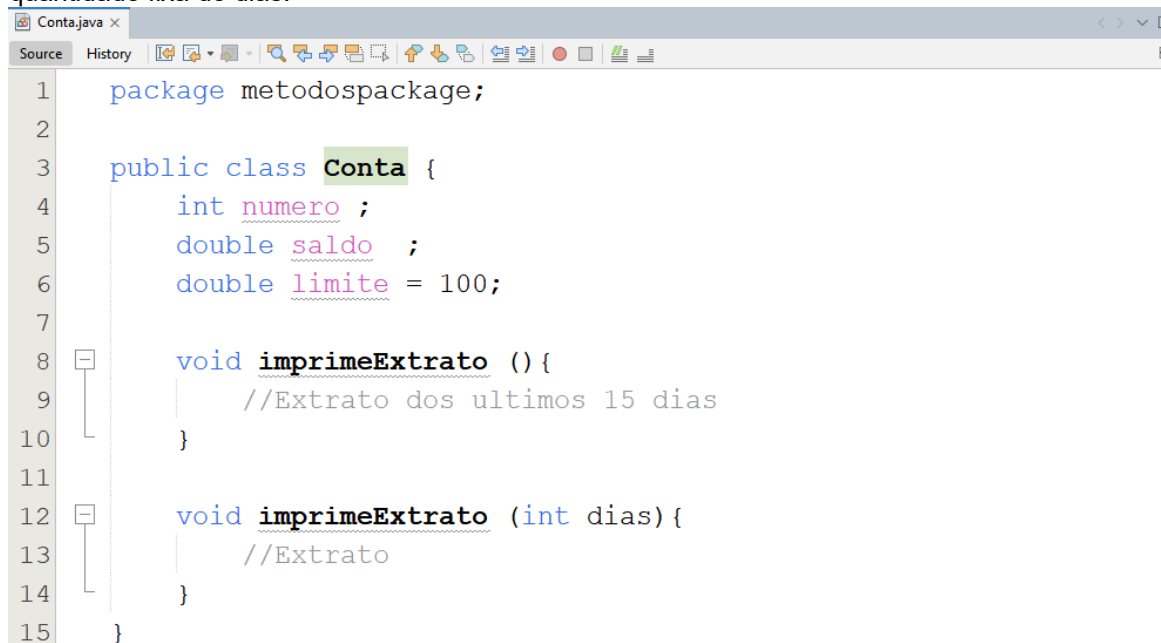
Geralmente, essas informações são obtidas através de extratos. No sistema do banco, os extratos podem ser gerados por métodos da classe Conta.



```
1 package metodospackage;
2
3 public class Conta {
4     int numero ;
5     double saldo ;
6     double limite = 100;
7
8     void imprimeExtrato (int dias){
9         //Extrato
10    }
11 }
```

O método `imprimeExtrato()` recebe a quantidade de dias que deve ser considerada para gerar o extrato da conta. Por exemplo, se esse método receber o valor 30 então ele deve gerar um extrato com as movimentações dos últimos 30 dias.

Em geral, extratos dos últimos 15 dias atendem as necessidades dos clientes. Dessa forma, poderíamos acrescentar um método na classe Conta para gerar extratos com essa quantidade fixa de dias.



```
1 package metodospackage;
2
3 public class Conta {
4     int numero ;
5     double saldo ;
6     double limite = 100;
7
8     void imprimeExtrato () {
9         //Extrato dos ultimos 15 dias
10    }
11
12     void imprimeExtrato (int dias){
13         //Extrato
14    }
15 }
```

O primeiro método não recebe parâmetros pois ele utilizará uma quantidade de dias padrão definida pelo banco para gerar os extratos (15 dias).

O segundo recebe um valor inteiro como parâmetro e deve considerar essa quantidade de dias para gerar os extratos.

Os dois métodos possuem o mesmo nome e lista de parâmetros diferentes. Quando dois ou mais métodos são definidos na mesma classe com o mesmo nome, dizemos que houve uma **sobrecarga** de métodos. Uma sobrecarga de métodos só é válida se as listas de parâmetros dos métodos são diferentes entre si.

No caso dos dois métodos que geram extratos, poderíamos evitar repetição de código fazendo um método chamar o outro.

```

Conta.java x
Source History
1 package metodospackage;
2
3 public class Conta {
4     int numero ;
5     double saldo ;
6     double limite = 100;
7
8     void imprimeExtrato () {
9         this.imprimeExtrato(15);
10    }
11
12    void imprimeExtrato (int dias){
13        //Extrato
14    }
15 }

```

21.1 Exercícios de Fixação

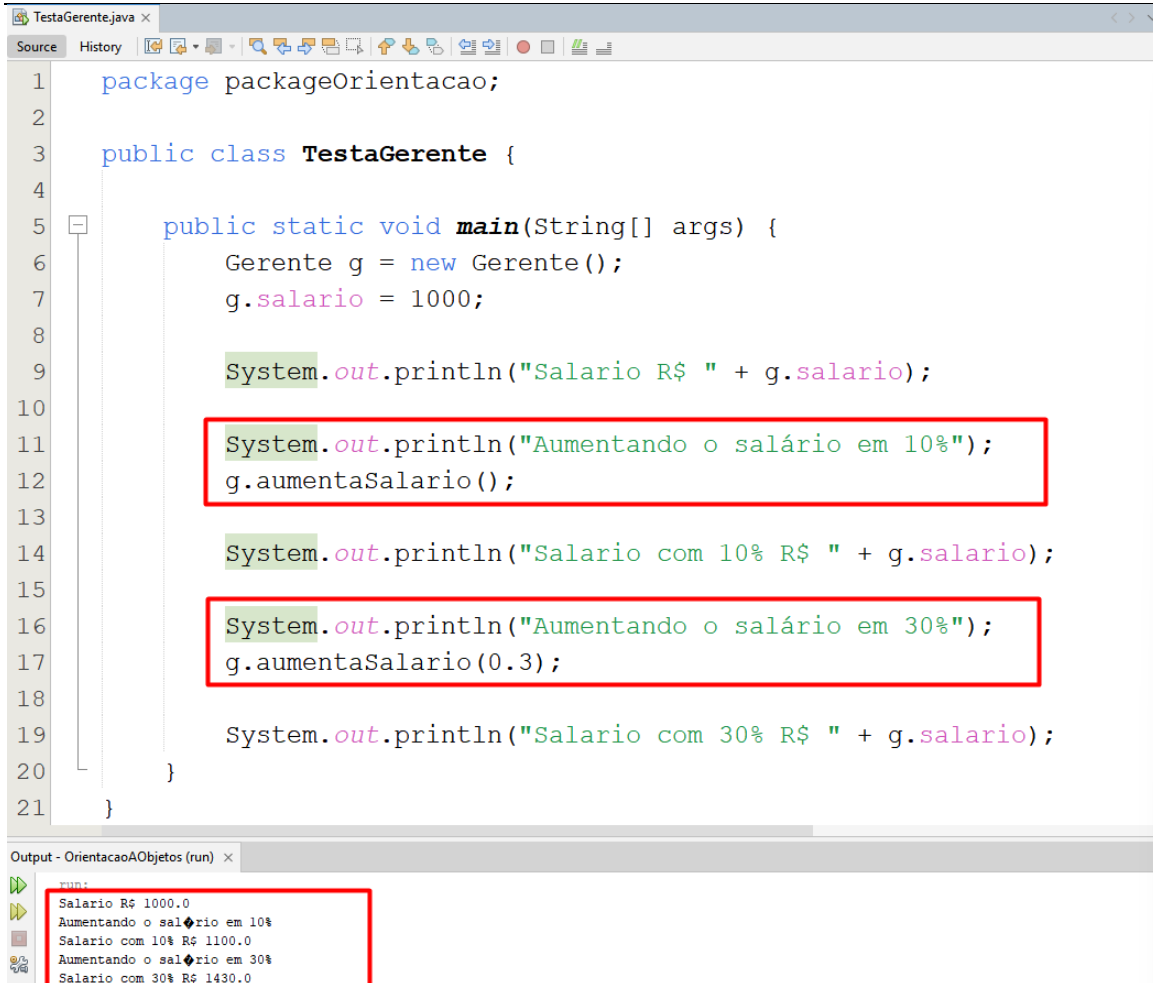
1-) Crie uma classe (Java Class) chamada **Gerente** para definir os objetos que representarão os gerentes do banco. Defina dois métodos de aumento salarial nessa classe. O primeiro deve aumentar o salário com uma taxa fixa de 10%. O segundo deve aumentar o salário com uma taxa variável.

```

Gerente.java x
Source History
1 package packageOrientacao;
2
3 public class Gerente {
4     String nome;
5     double salario;
6
7     void aumentaSalario() {
8         this.aumentaSalario(0.1);
9     }
10
11    void aumentaSalario(double taxa) {
12        this.salario += this.salario * taxa;
13    }
14 }

```

Teste os métodos de aumento salarial definidos na classe Gerente.



```
1 package packageOrientacao;
2
3 public class TestaGerente {
4
5     public static void main(String[] args) {
6         Gerente g = new Gerente();
7         g.salario = 1000;
8
9         System.out.println("Salario R$ " + g.salario);
10
11         System.out.println("Aumentando o salário em 10%");
12         g.aumentaSalario();
13
14         System.out.println("Salario com 10% R$ " + g.salario);
15
16         System.out.println("Aumentando o salário em 30%");
17         g.aumentaSalario(0.3);
18
19         System.out.println("Salario com 30% R$ " + g.salario);
20     }
21 }
```

Output - OrientacaoAOBJetos (run) ×

```
run:
Salario R$ 1000.0
Aumentando o salário em 10%
Salario com 10% R$ 1100.0
Aumentando o salário em 30%
Salario com 30% R$ 1430.0
```

Compile e execute a classe TestaGerente.

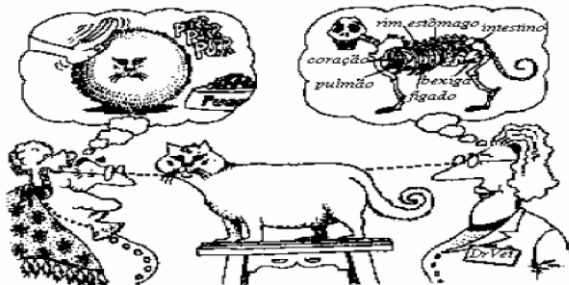
22. Abstração

Uma das principais formas do ser humano lidar com a **complexidade é através do uso de abstrações**. As pessoas tipicamente tentam compreender o mundo, construindo modelos mentais de partes dele. Tais modelos são uma visão simplificada de algo, onde apenas elementos relevantes são considerados. Modelos mentais, portanto, são mais simples do que os complexos sistemas que eles modelam.

Consideremos, por exemplo, um mapa como um modelo do território que ele representa. Um mapa é útil porque abstrai apenas aquelas características do território que se deseja modelar. Se um mapa incluísse todos os detalhes do território, provavelmente teria o mesmo tamanho do território e, portanto, não serviria a seu propósito.

Da mesma forma que um mapa precisa ser significativamente menor que o território que mapeia, incluindo apenas informações cuidadosamente selecionadas, um modelo mental abstrai apenas as características relevantes de um sistema para seu entendimento. Assim, podemos definir abstração como sendo o princípio de ignorar aspectos não relevantes de um assunto, segundo a perspectiva de um observador, tornando possível uma concentração maior nos aspectos principais do mesmo. De fato, a abstração consiste na seleção que um observador faz de alguns aspectos de um assunto, em detrimento de outros que não demonstram ser relevantes para o propósito em questão, isto é, a abstração é aplicada de acordo com o interesse do observador, e por isso de um mesmo objeto pode-se ter diferentes visões, como demonstra a figura 1 abaixo.

Só devem ser mapeados os objetos que são relevantes ao problema, bem como as características (propriedades e comportamento) desses objetos que forem necessários.



No que tange ao desenvolvimento de software, duas formas adicionais de abstração têm grande importância: a abstração de dados e a abstração de procedimentos.

23. Atributos Privados

No sistema do banco, cada objeto da classe `Funcionario` possui um atributo para guardar o salário do funcionário que ele representa.

```
Funcionario.java x
Source History
1 package packageOrientacao;
2
3 public class Funcionario {
4     double salario;
5 }
```

O atributo `salario` pode ser acessado ou modificado por código escrito em qualquer classe que esteja no mesmo diretório que a classe `Funcionario`. Portanto, o controle desse atributo é descentralizado.

Para identificar algum erro relacionado a manipulação dos salários dos funcionários, é necessário verificar o código de todos os arquivos da pasta onde a classe `Funcionario` está definida. Quanto maior o número de arquivos, menos eficiente será a manutenção da aplicação.

Podemos obter um controle centralizado tornando o atributo `salario` **privado** e definindo métodos para implementar todas as lógicas que utilizam ou modificam o valor desse atributo.

```
Funcionario.java x
Source History
1 package packageOrientacao;
2
3 public class Funcionario {
4     private double salario;
5
6     void aumentaSalario(double aumento) {
7         //Lógica para aumentar o salário
8     }
9 }
```

Um atributo privado só pode ser acessado ou alterado por código escrito dentro da classe na qual ele foi definido. Se algum código fora da classe `Funcionario` tentar acessar ou alterar o valor do atributo privado `salario`, um erro de compilação será gerado.

Definir todos os atributos como privado e métodos para implementar as lógicas de acesso e alteração é quase uma regra da orientação a objetos. O intuito é ter sempre um controle centralizado dos dados dos objetos para facilitar a manutenção do sistema e a detecção de erros.

24. Métodos Privados

O papel de alguns métodos pode ser o de auxiliar outros métodos da mesma classe. E muitas vezes, não é correto chamar esses métodos auxiliares de fora da sua classe diretamente.

No exemplo abaixo, o método `desconta Tarifa()` é um método auxiliar dos métodos `deposita()` e `saca()`. Além disso, ele não deve ser chamado diretamente, pois a tarifa só deve ser descontada quando ocorre um depósito ou um saque.

```
Conta.java x
Source History
1 package packageOrientacao;
2
3 public class Conta {
4     private double saldo;
5
6     void deposita(double valor){
7         this.saldo += valor;
8         this.descontaTarifa();
9     }
10
11    void saca(double valor){
12        this.saldo -= valor;
13        this.descontaTarifa();
14    }
15
16    void descontaTarifa(){
17        this.saldo -= 0.1;
18    }
19 }
```

Para garantir que métodos auxiliares não sejam chamados por código escrito fora da classe na qual eles foram definidos, podemos torná-los privados, acrescentando o modificador `private`.

```
16 private void descontaTarifa(){
17     this.saldo -= 0.1;
18 }
```

Qualquer chamada ao método `descontaTarifa()` realizada fora da classe `Conta` gera um erro de compilação.

25. Métodos Públicos

Os métodos que devem ser chamados a partir de qualquer parte do sistema devem possuir o modificador de visibilidade `public`.

```
Conta.java x
Source History
1 package packageOrientacao;
2
3 public class Conta {
4     private double saldo;
5
6     public void deposita(double valor){
7         this.saldo += valor;
8         this.descontaTarifa();
9     }
10
11    public void saca(double valor){
12        this.saldo -= valor;
13        this.descontaTarifa();
14    }
15
16    private void descontaTarifa(){
17        this.saldo -= 0.1;
18    }
19 }
```

26. Por que encapsular?

Uma das ideias mais importantes da orientação a objetos é o encapsulamento. Encapsular significa esconder a implementação dos objetos. O encapsulamento favorece principalmente dois aspectos de um sistema: a manutenção e o desenvolvimento.

A manutenção é favorecida pois, uma vez aplicado o encapsulamento, quando o funcionamento de um objeto deve ser alterado, em geral, basta modificar a classe do mesmo.

O desenvolvimento é favorecido pois, uma vez aplicado o encapsulamento, conseguimos determinar precisamente as responsabilidades de cada classe da aplicação.

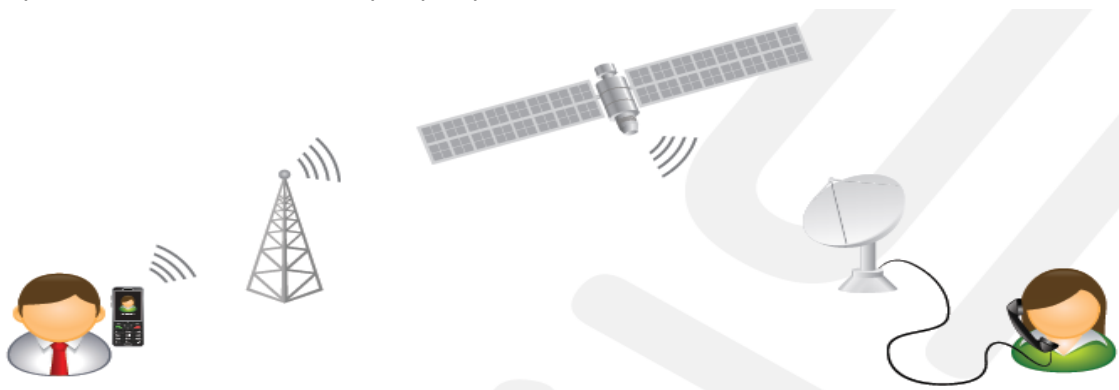
O conceito de encapsulamento pode ser identificado em diversos exemplos do cotidiano. Mostraremos alguns desses exemplos para esclarecer melhor a ideia.

26.1.1.1. Celular - Escondendo a complexidade

Hoje em dia, as pessoas estão acostumadas com os celulares. Os botões, a tela e os menus de um celular formam a **interface de uso** do mesmo. Em outras palavras, o usuário interage com esses aparelhos através dos botões, da tela e dos menus. Os dispositivos internos de um celular e os processos que transformam o som capturado pelo microfone em ondas que podem ser transmitidas para uma antena da operadora de telefonia móvel constituem a **implementação** do celular.

Do ponto de vista do usuário de um celular, para fazer uma ligação, basta digitar o número do telefone desejado e clicar no botão que efetua a ligação. Porém, diversos processos complexos são realizados pelo aparelho para que as pessoas possam conversar através dele. Se os usuários tivessem que possuir conhecimento de todo o funcionamento interno dos celulares, certamente a maioria das pessoas não os utilizariam.

No contexto da orientação a objetos, aplicamos o encapsulamento para criar objetos mais simples de serem utilizados em qualquer parte do sistema.



26.1.2. Carro - Evitando efeitos colaterais

A interface de uso de um carro é composta pelos dispositivos que permitem que o motorista conduza o veículo (volante, pedais, alavanca do câmbio, etc).

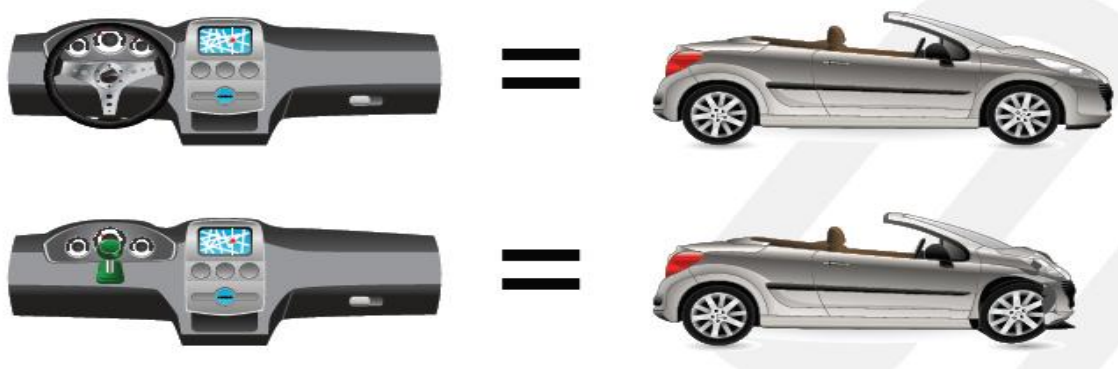
A implementação do carro é composta pelos dispositivos internos (motor, caixa de câmbio, radiador, sistema de injeção eletrônica ou carburador, etc) e pelos processos realizados internamente por esses dispositivos.

Nos carros mais antigos, o dispositivo interno que leva o combustível para o motor é o carburador. Nos carros mais novos, o carburador foi substituído pelo sistema de injeção eletrônica. Inclusive, algumas oficinas especializadas substituem o carburador pelo sistema de injeção eletrônica.

Essa alteração na implementação do carro não afeta a maneira que o motorista dirige. Todo mundo que sabe dirigir um carro com carburador também sabe dirigir um carro com injeção eletrônica. Hoje em dia, as montadoras fabricam veículos com câmbio mecânico ou automático. O motorista acostumado a dirigir carros com câmbio mecânico pode ter dificuldade para dirigir carros com câmbio automático e vice-versa. Quando a interface de uso do carro é alterada, a maneira de dirigir é afetada, fazendo com que as pessoas que sabem dirigir tenham que se adaptar.

No contexto da orientação a objetos, aplicando o conceito do encapsulamento, as implementações dos objetos ficam “escondidas”. Dessa forma, podemos modificá-las sem afetar a maneira de utilizar esses objetos. Por outro lado, se alterarmos a interface de uso que está exposta, afetaremos a maneira de usar os objetos.

Considere, por exemplo, a mudança do nome de um método público. Todas as chamadas a esse método devem ser alteradas, o que pode causar diversos efeitos colaterais nas classes da aplicação.



26.1.3. Máquinas de Porcarias - Aumentando o controle

Estamos acostumados a utilizar máquinas de refrigerantes, de salgadinhos, de doces, de café, etc. Em geral, essas máquinas oferecem uma interface de uso composta por:

- Entradas para moedas ou cédulas.
- Botões para escolher o produto desejado.
- Saída do produto.
- Saída para o troco.

Normalmente, essas máquinas são extremamente protegidas. Elas garantem que nenhum usuário mal intencionado (ou não) tente alterar a implementação da máquina, ou seja, tente alterar como a máquina funciona por dentro.

Levando essa ideia para um sistema orientado a objetos, um objeto deve ser bem protegido para que outros objetos não prejudiquem o seu funcionamento interno.



27. Acessando ou modificando atributos

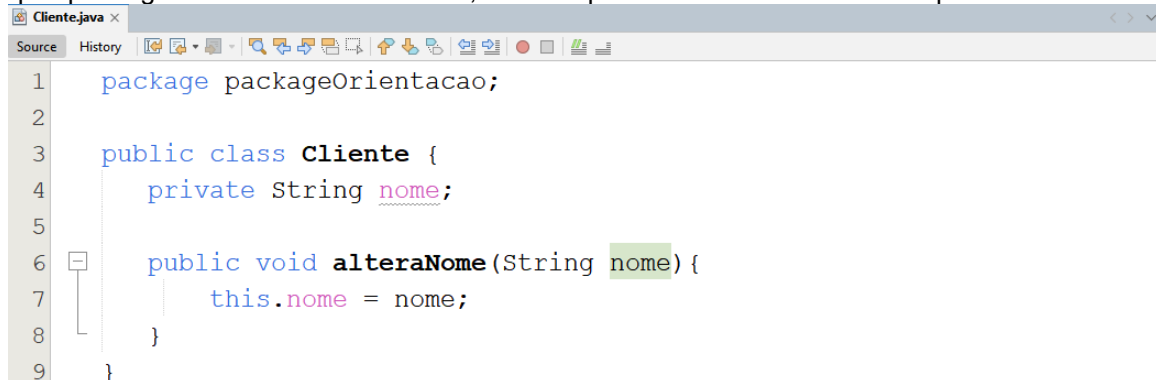
Aplicando a ideia do encapsulamento, os atributos deveriam ser todos privados. Consequentemente, os atributos não podem ser acessados ou modificados por código escrito fora da classe na qual eles foram definidos.

Porém, muitas vezes, as informações armazenadas nos atributos precisam ser consultadas de qualquer lugar do sistema. Nesse caso, **podemos disponibilizar métodos para consultar os valores dos atributos.**

```

Cliente.java x
Source History
1 package packageOrientacao;
2
3 public class Cliente {
4     private String nome;
5
6     public String consultaNome() {
7         return this.nome;
8     }
9 }
    
```

Da mesma forma, eventualmente, é necessário modificar o valor de um atributo a partir de qualquer lugar do sistema. Nesse caso, também poderíamos criar um método para essa tarefa.



```

1 package packageOrientacao;
2
3 public class Cliente {
4     private String nome;
5
6     public void alteraNome(String nome) {
7         this.nome = nome;
8     }
9 }

```

Muitas vezes, é necessário consultar e alterar o valor de um atributo a partir de qualquer lugar do sistema. Nessa situação, podemos definir os dois métodos discutidos anteriormente. Mas, o que é melhor? Criar os dois métodos (um de leitura e outro de escrita) ou deixar o atributo público?

Quando queremos consultar a quantidade de combustível de um automóvel, olhamos o painel ou abrimos o tanque de combustível?

Quando queremos alterar o toque da campainha de um celular, utilizamos os menus do celular ou desmontamos o aparelho?

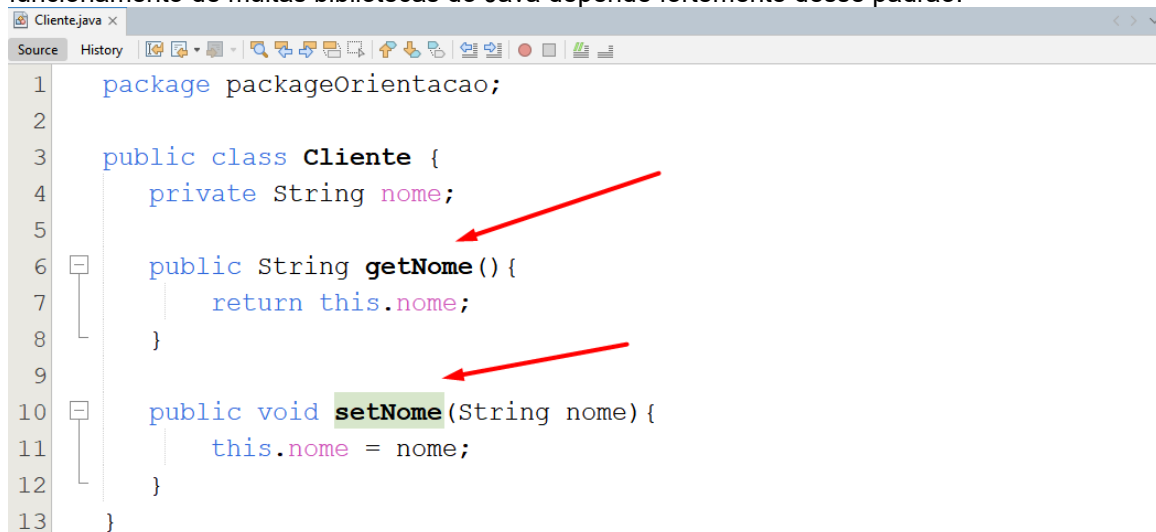
Acessar ou modificar as propriedades de um objeto manipulando diretamente os seus atributos é uma abordagem que normalmente gera problemas. Por isso, é mais seguro para a integridade dos objetos e, conseqüentemente, para a integridade da aplicação, que esse acesso ou essa modificação sejam realizados através de métodos do objeto. Utilizando métodos, podemos controlar como as alterações e as consultas são realizadas. Ou seja, temos um controle maior.

28. Getters e Setters

Na linguagem Java, há uma convenção de nomenclatura para os métodos que têm como finalidade acessar ou alterar as propriedades de um objeto.

Segundo essa convenção, os nomes dos métodos que permitem a consulta das propriedades de um objeto devem possuir o prefixo **get**. Analogamente, os nomes dos métodos que permitem a alteração das propriedades de um objeto devem possuir o prefixo **set**.

Na maioria dos casos, é muito conveniente seguir essa convenção, pois os desenvolvedores Java já estão acostumados com essas regras de nomenclatura e o funcionamento de muitas bibliotecas do Java depende fortemente desse padrão.



```

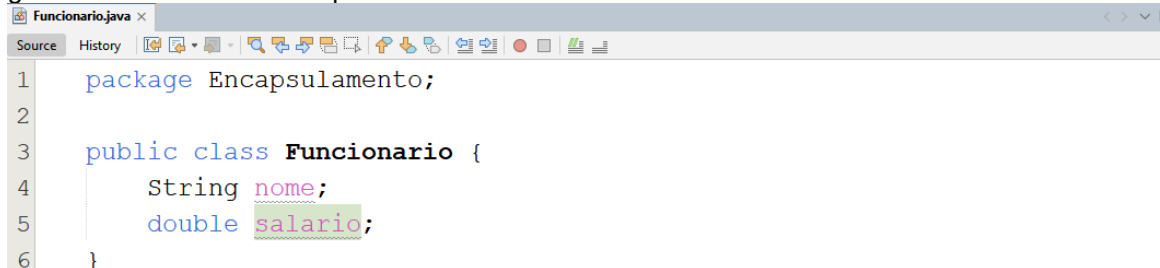
1 package packageOrientacao;
2
3 public class Cliente {
4     private String nome;
5
6     public String getNome() {
7         return this.nome;
8     }
9
10    public void setNome(String nome) {
11        this.nome = nome;
12    }
13 }

```

29. Exercícios de Fixação

1-) Crie um projeto no NetBeans chamado **Encapsulamento**.

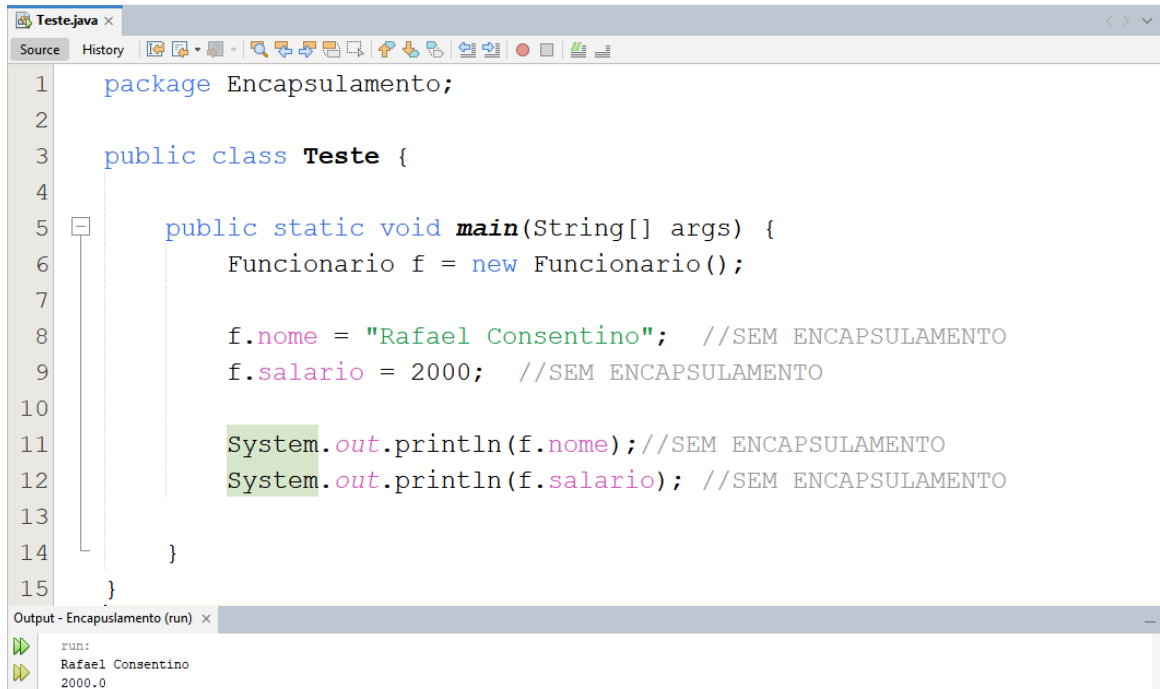
2-) Defina uma classe para representar os funcionários do banco com um atributo para guardar os salários e outro para os nomes.



```

1 package Encapsulamento;
2
3 public class Funcionario {
4     String nome;
5     double salario;
6 }
    
```

3-) Teste a classe Funcionario criando um objeto e manipulando diretamente os seus atributos.



```

1 package Encapsulamento;
2
3 public class Teste {
4
5     public static void main(String[] args) {
6         Funcionario f = new Funcionario();
7
8         f.nome = "Rafael Consentino"; //SEM ENCAPSULAMENTO
9         f.salario = 2000; //SEM ENCAPSULAMENTO
10
11         System.out.println(f.nome); //SEM ENCAPSULAMENTO
12         System.out.println(f.salario); //SEM ENCAPSULAMENTO
13
14     }
15 }
    
```

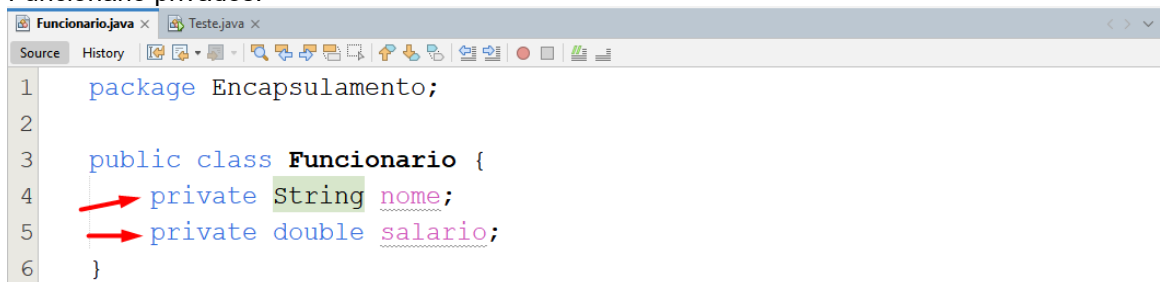
Output - Encapsulamento (run) x

```

run:
Rafael Consentino
2000.0
    
```

4-) Compile a classe Teste e perceba que ela pode acessar ou modificar os valores dos atributos de um objeto da classe Funcionario. Execute o teste e observe o console.

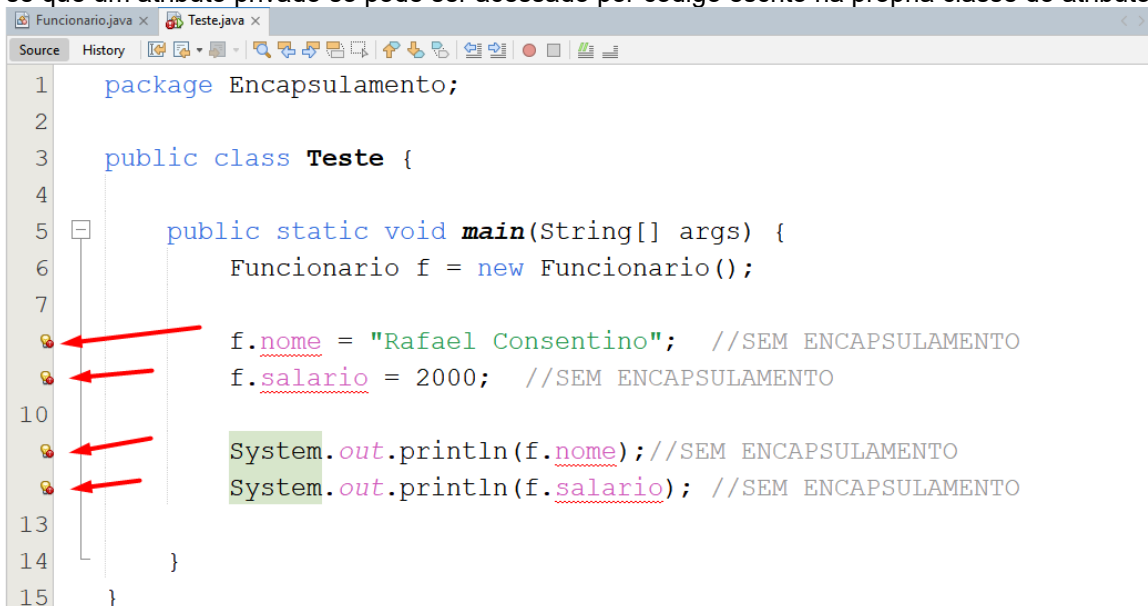
5-) Aplique a ideia do encapsulamento tornando os atributos definidos na classe Funcionario privados.



```

1 package Encapsulamento;
2
3 public class Funcionario {
4     private String nome;
5     private double salario;
6 }
    
```

6-) Tente compilar novamente a classe Teste. Observe os erros de compilação. Lembre-se que um atributo privado só pode ser acessado por código escrito na própria classe do atributo.

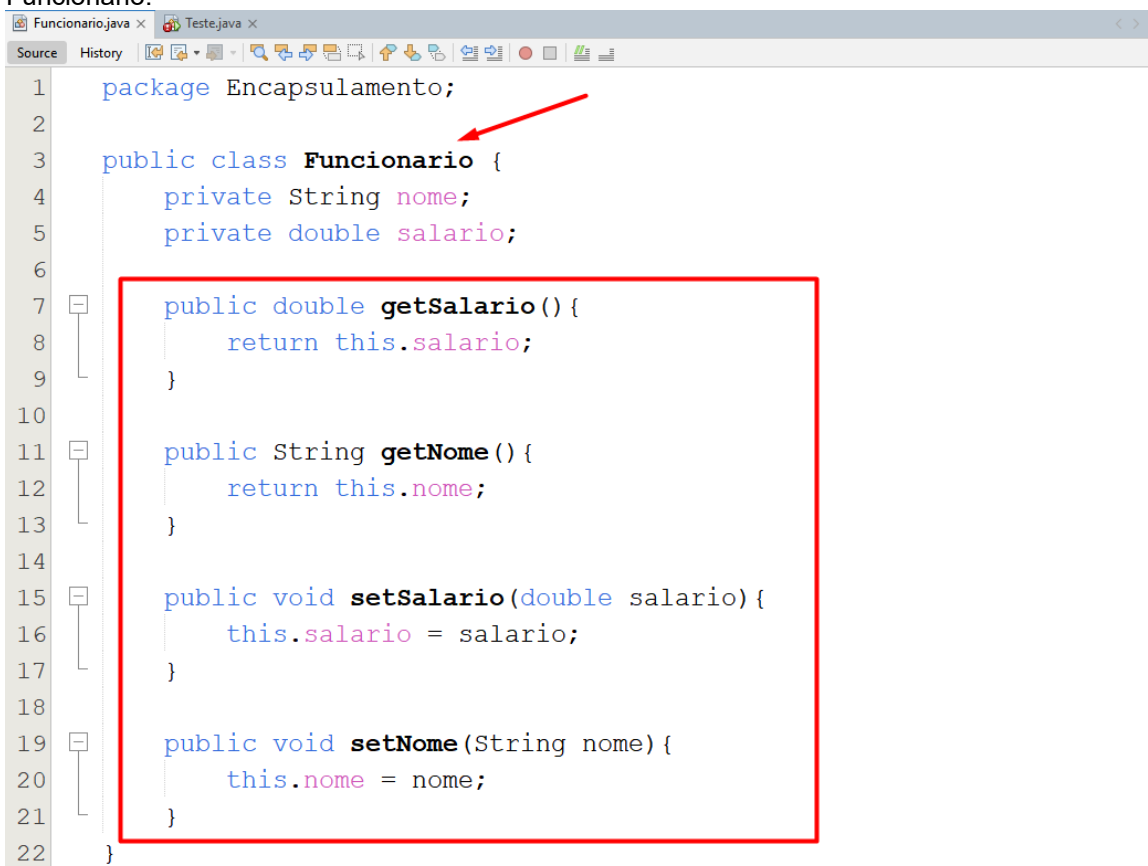


```

1 package Encapsulamento;
2
3 public class Teste {
4
5     public static void main(String[] args) {
6         Funcionario f = new Funcionario();
7
8         f.nome = "Rafael Consentino"; //SEM ENCAPSULAMENTO
9         f.salario = 2000; //SEM ENCAPSULAMENTO
10
11         System.out.println(f.nome); //SEM ENCAPSULAMENTO
12         System.out.println(f.salario); //SEM ENCAPSULAMENTO
13
14     }
15 }

```

7-) Crie métodos de acesso com nomes padronizados para os atributos definidos na classe Funcionario.



```

1 package Encapsulamento;
2
3 public class Funcionario {
4     private String nome;
5     private double salario;
6
7     public double getSalario() {
8         return this.salario;
9     }
10
11     public String getNome() {
12         return this.nome;
13     }
14
15     public void setSalario(double salario) {
16         this.salario = salario;
17     }
18
19     public void setNome(String nome) {
20         this.nome = nome;
21     }
22 }

```

8-) Altere a classe Teste para que ela utilize os métodos de acesso ao invés de manipular os atributos do objeto da classe Funcionario diretamente.

```
Teste.java x
Source History
1 package Encapsulamento;
2
3 public class Teste {
4
5     public static void main(String[] args) {
6         Funcionario f = new Funcionario();
7
8         //f.nome = "Rafael Consentino"; //SEM ENCAPSULAMENTO
9         f.setName("Rafael Consentino"); //COM ENCAPSULAMENTO
10
11        //f.salario = 2000; //SEM ENCAPSULAMENTO
12        f.setSalario(20000); //COM ENCAPSULAMENTO
13
14        //System.out.println(f.nome); //SEM ENCAPSULAMENTO
15        System.out.println(f.getName()); //COM ENCAPSULAMENTO
16
17        //System.out.println(f.salario); //SEM ENCAPSULAMENTO
18        System.out.println(f.getSalario()); //COM ENCAPSULAMENTO
19    }
20 }
```

Compile e execute o teste!

30. HERANÇA

30.1. Reutilização de Código

Um banco oferece diversos serviços que podem ser contratados individualmente pelos clientes. Quando um serviço é contratado, o sistema do banco deve registrar quem foi o cliente que contratou o serviço, quem foi o funcionário responsável pelo atendimento ao cliente e a data de contratação.

Com o intuito de ser produtivo, a modelagem dos serviços do banco deve diminuir a repetição de código. A ideia é reaproveitar o máximo do código já criado. Essa ideia está diretamente relacionada ao conceito *Don't Repeat Yourself*. Em outras palavras, devemos minimizar ao máximo a utilização do "copiar e colar". O aumento da produtividade e a diminuição do custo de manutenção são as principais motivações do DRY.

Em seguida, vamos discutir algumas modelagens possíveis para os serviços do banco.

Buscaremos seguir a ideia do DRY na criação dessas modelagens.

30.2. Uma classe para todos os serviços

Poderíamos definir apenas uma classe para modelar todos os tipos de serviços que o banco oferece.

```
3 public class Servico {
4     private Cliente contratante;
5     private Funcionario responsavel;
6     private String dataDeContratacao;
7
8     //métodos
9 }
```

30.3. Empréstimo

O empréstimo é um dos serviços que o banco oferece. Quando um cliente contrata esse serviço, são definidos o valor e a taxa de juros mensal do empréstimo. Devemos acrescentar dois atributos na classe `Servico`: um para o valor e outro para a taxa de juros do serviço de empréstimo.

```
3 public class Emprestimo {
4     //Geral
5     private Cliente contratante;
6     private Funcionario responsavel;
7     private String dataDeContratacao;
8
9     //Empréstimo
10    private double valor;
11    private double taxa;
12
13    //métodos
14 }
```

30.4. Seguro de veículos

Outro serviço oferecido pelo banco é o seguro de veículos. Para esse serviço devem ser definidas as seguintes informações: veículo segurado, valor do seguro e a franquia. Devemos adicionar três atributos na classe `Servico`.

```
3 public class SeguroDeVeiculo {
4     //Geral
5     private Cliente contratante;
6     private Funcionario responsavel;
7     private String dataDeContratacao;
8
9     //Seguro de Veículos
10    private Emprestimo veiculo;
11    private double valorDoSeguroDeVeiculo;
12    private double franquia;
13
14    //métodos
15 }
```

Apesar de seguir a ideia do DRY, modelar todos os serviços com apenas uma classe pode dificultar o desenvolvimento. Supondo que dois ou mais desenvolvedores são responsáveis pela implementação dos serviços, eles provavelmente modificariam a mesma classe concorrentemente. Além disso, os desenvolvedores, principalmente os recém chegados no projeto do banco, ficariam confusos com o código extenso da classe `Servico`.

Outro problema é que um objeto da classe `Servico` possui atributos para todos os serviços que o banco oferece. Na verdade, ele deveria possuir apenas os atributos relacionados a um serviço. Do ponto de vista de performance, essa abordagem causaria um consumo desnecessário de memória.

30.5. Uma classe para cada serviço

Para modelar melhor os serviços, evitando uma quantidade grande de atributos e métodos desnecessários, criaremos uma classe para cada serviço.

```
3 public class SeguroDeVeiculo {
4     //Geral
5     private Cliente contratante;
6     private Funcionario responsavel;
7     private String dataDeContratacao;
8
9     //Seguro de Veículos
10    private Emprestimo veiculo;
11    private double valorDoSeguroDeVeiculo;
12    private double franquia;
13
14    //métodos
15 }
```

E:

```
3 public class Emprestimo {
4     //Geral
5     private Cliente contratante;
6     private Funcionario responsavel;
7     private String dataDeContratacao;
8
9     //Empréstimo
10    private double valor;
11    private double taxa;
12
13    //métodos
14 }
```

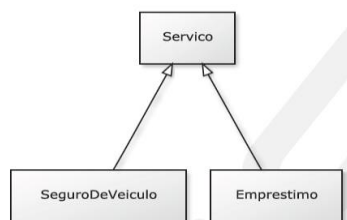
Criar uma classe para cada serviço torna o sistema mais flexível, pois qualquer alteração em um determinado serviço não causará efeitos colaterais nos outros. Mas, por outro lado, essas classes teriam bastante código repetido, contrariando a ideia do DRY. Além disso, qualquer alteração que deva ser realizada em todos os serviços precisa ser implementada em cada uma das classes.

30.6. Uma classe genérica e várias específicas

Na modelagem dos serviços do banco, podemos aplicar um conceito de orientação a objetos chamado Herança. A ideia é reutilizar o código de uma determinada classe em outras classes.

Aplicando herança, teríamos a classe *Servico* com os atributos e métodos que todos os serviços devem ter e uma classe para cada serviço com os atributos e métodos específicos do determinado serviço.

As classes específicas seriam “ligadas” de alguma forma à classe *Servico* para reaproveitar o código nela definido. Esse relacionamento entre as classes é representado em UML pelo diagrama a seguir.



Os objetos das classes específicas *Emprestimo* e *SeguroDeVeiculo* possuiriam tanto os atributos e métodos definidos nessas classes quanto os definidos na classe *Servico*.

As classes específicas são vinculadas a classe genérica utilizando o comando *extends*. Não é necessário redefinir o conteúdo já declarado na classe genérica.

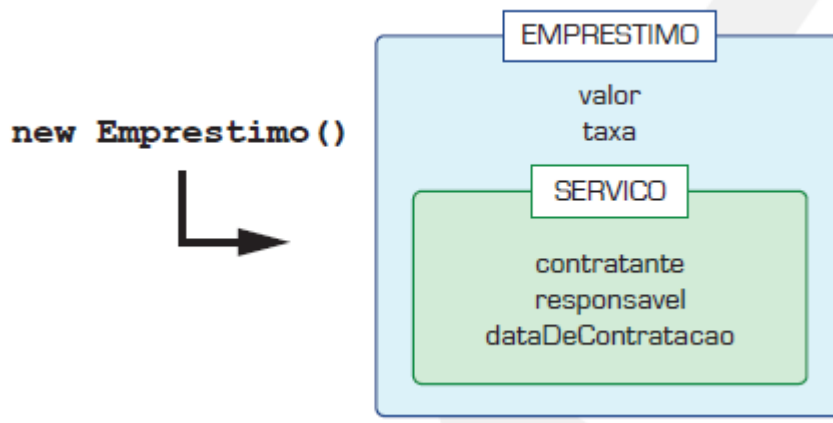
```
3 public class Servico {
4     //Geral
5     private Cliente contratante;
6     private Funcionario responsavel;
7     private String dataDeContratacao;
8 }
```

```
3 public class Emprestimo extends Servico{
4     //Geral
5     private Cliente contratante;
6     private Funcionario responsavel;
7     private String dataDeContratacao;
8
9     //Empréstimo
10    private double valor;
11    private double taxa;
12
13    //métodos
14 }
```

```
3 public class SeguroDeVeiculo extends Servico{
4     //Geral
5     private Cliente contratante;
6     private Funcionario responsavel;
7     private String dataDeContratacao;
8
9     //Seguro de Veículos
10    private Emprestimo veiculo;
11    private double valorDoSeguroDeVeiculo;
12    private double franquia;
13
14    //métodos
15 }
```

A classe genérica é denominada super classe, classe base ou classe mãe. As classes específicas são denominadas sub classes, classes derivadas ou classes filhas.

Quando o operador *new* é aplicado em uma sub classe, o objeto construído possuirá os atributos e métodos definidos na sub classe e na super classe.



Preço Fixo

Suponha que todo serviço do banco possui uma taxa administrativa que deve ser paga pelo cliente que contratar o serviço. Inicialmente, vamos considerar que o valor dessa taxa é igual para todos os serviços do banco.

Neste caso, poderíamos implementar um método na classe Servico para calcular o valor da taxa.

Este método será reaproveitado por todas as classes que herdarem da classe Servico.

```

1 public class Servico {
2     //Geral
3     private Cliente contratante;
4     private Funcionario responsavel;
5     private String dataDeContratacao;
6
7     public double calculaTaxa() {
8         return 10;
9     }
10 }
  
```

```

3 public class TestaServico {
4
5     public static void main(String[] args) {
6         Emprestimo e = new Emprestimo();
7         SeguroDeVeiculo sdv = new SeguroDeVeiculo();
8
9         System.out.println("Emprestimo " + e.calculaTaxa());
10        System.out.println("SeguroDeVeiculo " + sdv.calculaTaxa());
11    }
12 }
13
14
  
```

Output - Run (TestaServico) x

```

--- resources:3.3.1:resources (default-resources) @ ExemplosParaAulas ---
skip non existing resourceDirectory C:\Users\marco\Documents\NetBeansProjects\ExemplosParaAulas\src\main\resources
--- compiler:3.13.0:compile (default-compile) @ ExemplosParaAulas ---
Recompiling the module because of changed source code.
Compiling 9 source files with javac [debug release 22] to target\classes
--- exec:3.1.0:exec (default-cli) @ ExemplosParaAulas ---
Emprestimo 10.0
SeguroDeVeiculo 10.0
  
```

31. Reescrita de Método

Suponha que o valor da taxa administrativa do serviço de empréstimo é diferente dos outros serviços, pois ele é calculado a partir do valor emprestado ao cliente. Como esta lógica é específica para o serviço de empréstimo, devemos acrescentar um método para implementar esse cálculo na classe Empréstimo.

```

3  public class Empréstimo extends Servico{
4      //Geral
5      private Cliente contratante;
6      private Funcionario responsavel;
7      private String dataDeContratacao;
8
9      //Empréstimo
10     private double valor;
11     private double taxa;
12
13     //métodos
14     public double calculaTaxaDeEmpréstimo() {
15         return this.valor * 0.1;
16     }
17 }

```

Os métodos das classes específicas têm prioridade sobre os métodos das classes genéricas. Em outras palavras, se o método chamado existe na classe filha ele será chamado, caso contrário o método será procurado na classe mãe.

Quando definimos um método com a mesma assinatura na classe base e em alguma classe derivada, estamos aplicando o conceito de Reescrita de Método.

Fixo + Específico

Suponha que o preço de um serviço é a soma de um valor fixo mais um valor que depende do tipo do serviço. Por exemplo, o preço do serviço de empréstimo é 5 reais mais uma porcentagem do valor emprestado ao cliente. O preço do serviço de seguro de veículo é 5 reais mais uma porcentagem do valor do veículo segurado. Em cada classe específica, podemos reescrever o método calculaTaxa().

```

3  public class Empréstimo extends Servico{
4      //Geral
5      private Cliente contratante;
6      private Funcionario responsavel;
7      private String dataDeContratacao;
8
9      //Empréstimo
10     private double valor;
11     private double taxa;
12
13     //métodos
14     @Override
15     public double calculaTaxa() {
16         return 5 + this.valor * 0.1;
17     }
18 }

```

```

3  public class SeguroDeVeiculo extends Servico{
4      //Geral
5      private Cliente contratante;
6      private Funcionario responsavel;
7      private String dataDeContratacao;
8
9      //Seguro de Veículos
10     private Emprestimo veiculo;
11     private double valorDoSeguroDeVeiculo;
12     private double franquia;
13
14     //métodos
15     @Override
16     public double calculaTaxa() {
17         return 5+ this.valorDoSeguroDeVeiculo * 0.1;
18     }
19 }

```

Se o valor fixo dos serviços for atualizado, todas as classes específicas devem ser modificadas. Outra alternativa seria criar um método na classe Servico para calcular o valor fixo de todos os serviços e chamá-lo dos métodos reescritos nas classes específicas.

```

3  public class Servico {
4      //Geral
5      private Cliente contratante;
6      private Funcionario responsavel;
7      private String dataDeContratacao;
8
9      public double calculaTaxa() {
10         return 5;
11     }
12 }

```

```

3  public class Emprestimo extends Servico{
4      //Geral
5      private Cliente contratante;
6      private Funcionario responsavel;
7      private String dataDeContratacao;
8
9      //Empréstimo
10     private double valor;
11     private double taxa;
12
13     //métodos
14     @Override
15     public double calculaTaxa() {
16         return super.calculaTaxa() + this.valor * 0.1;
17     }
18 }

```

Dessa forma, quando o valor padrão do preço dos serviços é alterado, basta modificar o método na classe Serviço.

31.1. Construtores e Herança

Quando temos uma hierarquia de classes, as chamadas dos construtores são mais complexas do que o normal. Pelo menos um construtor de cada classe de uma mesma sequência hierárquica deve ser chamado ao instanciar um objeto. Por exemplo, quando um objeto da classe Emprestimo é criado, pelo menos um construtor da própria classe Emprestimo um da classe Serviço devem ser executados. Além disso, os construtores das classes mais genéricas são chamados antes dos construtores das classes específicas.

```
1 public class Serviço {
2     //Geral
3     private Cliente contratante;
4     private Funcionario responsavel;
5     private String dataDeContratacao;
6
7     public Serviço() {
8         System.out.println("Emprestimo");
9     }
10
11     public double calculaTaxa() {
12         return 5;
13     }
14 }
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
```

31.2. Exercícios de Fixação

- 1) Crie um projeto chamado Heranca2.
- 2) Defina uma classe para modelar os funcionários do banco. Sabendo que todo funcionário possui nome e salário, inclua os *getters* e *setters* dos atributos.

```
1 package heranca3;
2
3 public class Funcionario {
4     private String nome;
5     private double salario;
6
7     public String getNome() {
8         return nome;
9     }
10
11    public void setNome(String nome) {
12        this.nome = nome;
13    }
14
15    public double getSalario() {
16        return salario;
17    }
18
19    public void setSalario(double salario) {
20        this.salario = salario;
21    }
22 }
```

3) Crie uma classe para cada tipo específico de funcionário herdando da classe Funcionario. Considere apenas três tipos específicos de funcionários: gerentes, telefonistas e secretarias. Os gerentes possuem um nome de usuário e uma senha para acessar o sistema do banco. As telefonistas possuem um código de estação de trabalho. As secretarias possuem um número de ramal.

```
1 package heranca3;
2
3 public class Gerente extends Funcionario{
4     private String usuario;
5     private String senha;
6
7     public String getUsuario() {
8         return usuario;
9     }
10
11    public void setUsuario(String usuario) {
12        this.usuario = usuario;
13    }
14
15    public String getSenha() {
16        return senha;
17    }
18
19    public void setSenha(String senha) {
20        this.senha = senha;
21    }
22
23 }
24
25
26
27 }
```

```

1 package heranca3;
2
3 public class Telefonista extends Funcionario{
4     private int estacaoDeTrabalho;
5
6     public int getEstacaoDeTrabalho() {
7         return estacaoDeTrabalho;
8     }
9
10    public void setEstacaoDeTrabalho(int estacaoDeTrabalho) {
11        this.estacaoDeTrabalho = estacaoDeTrabalho;
12    }
13 }

```

```

1 package heranca3;
2
3 public class Secretaria extends Funcionario{
4     private int ramal;
5
6     public int getRamal() {
7         return ramal;
8     }
9
10    public void setRamal(int ramal) {
11        this.ramal = ramal;
12    }
13 }

```

4) Teste o funcionamento dos três tipos de funcionários criando um objeto de cada uma das classes: Gerente, Telefonista e Secretaria.

```

1 package heranca3;
2
3 public class TestaFuncionarios {
4
5     public static void main(String[] args) {
6         Gerente g = new Gerente();
7         //Classe Funcionario (Mãe)
8         g.setNome("Marcos Costa");
9         g.setSalario(32450.22);
10
11         //Classe Filha (Funcionario)
12         g.setUsuario("marcão");
13         g.setSenha("batatinha");
14
15         Telefonista t = new Telefonista();
16         //Classe Telefonista (Mãe)
17         t.setNome("Eliane Prado");
18         t.setSalario(10780.23);
19
20         //Classe Filha (Telefonista)
21         t.setEstacaoDeTrabalho(12);
22
23         Secretaria s = new Secretaria();

```



```

24         //Classe Funcionario (Mãe)
25         s.setNome("Nilva Prado");
26         s.setSalario(6550.99);
27
28         //Classe Filha (Secretaria)
29         s.setRamal(6677);
30
31         //Mostrar os dados Gerente
32         System.out.println("=====");
33         System.out.println("GERENTE:");
34         System.out.println("Nome: " + g.getNome());
35         System.out.println("Salário: " + g.getSalario());
36         System.out.println("Usuario: " + g.getUsuario());
37         System.out.println("Senha: " + g.getSenha());
38
39         //Mostrar os dados Telefonista
40         System.out.println("=====");
41         System.out.println("TELEFONISTA:");
42         System.out.println("Nome: " + t.getNome());
43         System.out.println("Salário: " + t.getSalario());
44         System.out.println("Estação de Trabalho: " + t.getEstacaoDeTrabalho());
45
46         //Mostrar os dados Secretaria
47         System.out.println("=====");
48         System.out.println("SECRETARIA:");
49         System.out.println("Nome: " + s.getNome());
50         System.out.println("Salário: " + s.getSalario());
51         System.out.println("Ramal: " + s.getRamal());
52     }
53 }

```

Execute o teste!

5) Suponha que todos os funcionários recebam uma bonificação de 10% do salário. Acrescente um método na classe Funcionario para calcular essa bonificação.

```

1  package heranca3;
2
3  @
4  public class Funcionario {
5      private String nome;
6      private double salario;
7
8      public String getNome() {
9          return nome;
10     }
11
12     public void setNome(String nome) {
13         this.nome = nome;
14     }
15
16     public double getSalario() {
17         return salario;
18     }
19
20     public void setSalario(double salario) {
21         this.salario = salario;
22     }
23 }

```

```
24 public double calculaBonificacao() {  
25     return this.salario * 0.1;  
26 }
```

6) Altere a classe TestaFuncionarios para imprimir a bonificação de cada funcionário, além dos dados que já foram impressos. Depois, execute o teste novamente.

```
1 package heranca3;  
2  
3 public class TestaFuncionarios {  
4  
5     public static void main(String[] args) {  
6         Gerente g = new Gerente();  
7         //Classe Funcionario (Mãe)  
8         g.setNome("Marcos Costa");  
9         g.setSalario(32450.22);  
10  
11         //Classe Filha (Funcionario)  
12         g.setUsuario("marcão");  
13         g.setSenha("batatinha");  
14  
15         Telefonista t = new Telefonista();  
16         //Classe Telefonista (Mãe)  
17         t.setNome("Eliane Prado");  
18         t.setSalario(10780.23);  
19  
20         //Classe Filha (Telefonista)  
21         t.setEstacaoDeTrabalho(12);  
22  
23         Secretaria s = new Secretaria();  
24         //Classe Funcionario (Mãe)  
25         s.setNome("Nilva Prado");  
26         s.setSalario(6550.99);  
27  
28         //Classe Filha (Secretaria)  
29         s.setRamal(6677);  
30  
31         //Mostrar os dados Gerente  
32         System.out.println("=====");  
33         System.out.println("GERENTE:");  
34         System.out.println("Nome: " + g.getNome());  
35         System.out.println("Salário: " + g.getSalario());  
36         System.out.println("Usuário: " + g.getUsuario());  
37         System.out.println("Senha: " + g.getSenha());  
38         System.out.println("Bonificação: " + g.calculaBonificacao());  
39  
40         //Mostrar os dados Telefonista  
41         System.out.println("=====");  
42         System.out.println("TELEFONISTA:");  
43         System.out.println("Nome: " + t.getNome());  
44         System.out.println("Salário: " + t.getSalario());  
45         System.out.println("Estação de Trabalho: " + t.getEstacaoDeTrabalho());  
46         System.out.println("Bonificação: " + t.calculaBonificacao());  
47  
48         //Mostrar os dados Secretaria  
49         System.out.println("=====");  
50         System.out.println("SECRETARIA:");
```

```

51     System.out.println("Nome: " + s.getNome());
52     System.out.println("Salário: " + s.getSalario());
53     System.out.println("Ramal: " + s.getRamal());
54     System.out.println("Bonificação: " + s.calculaBonificacao());
55 }
56 }

```

7) Suponha que os gerentes recebam uma bonificação maior que os outros funcionários. Reescreva o método `calculaBonificacao()` na classe `Gerente`. Depois, compile e execute o teste novamente.

```

1  package heranca3;
2
3  public class Gerente extends Funcionario{
4      private String usuario;
5      private String senha;
6
7      public String getUsuario() {
8          return usuario;
9      }
10
11     public void setUsuario(String usuario) {
12         this.usuario = usuario;
13     }
14
15     public String getSenha() {
16         return senha;
17     }
18
19     public void setSenha(String senha) {
20         this.senha = senha;
21     }
22
23     @Override
24     public double calculaBonificacao() {
25         return this.getSalario() * 0.6 + 100;
26     }
27 }

```

31.3. Exercícios Complementares

1) Defina na classe `Funcionario` um método para imprimir na tela o nome, salário e bonificação dos funcionários.

2) Reescreva o método que imprime os dados dos funcionários nas classes `Gerente`, `Telefonista` e `Secretaria` para acrescentar a impressão dos dados específicos de cada tipo de funcionário.

3) Modifique a classe `TestaFuncionarios` para utilizar o método `mostraDados()`.

31.4. Correção dos Exercícios Complementares

Primeiramente vamos criar o método de impressão na classe de funcionário.

```

Funcionario.java x
Source History
16         return salario;
17     }
18
19     public void setSalario(double salario) {
20         this.salario = salario;
21     }

```

```

22 public double calculaBonificacao() {
23     return this.salario * 0.1;
24 }
25
26
27 public void mostrarDados() {
28     System.out.println("Nome: " + getNome());
29     System.out.println("Salário: " + getSalario());
30     System.out.println("Bonificação: " + calculaBonificacao());
31 }
32 }

```

Agora vamos trabalhar a classe Gerente.

```

Gerente.java x
Source History
23 @Override
24 public double calculaBonificacao() {
25     return this.getSalario() * 0.6 + 100;
26 }
27
28 @Override
29 public void mostrarDados() {
30     super.mostrarDados();
31     System.out.println("Usuario: " + getUsuario());
32     System.out.println("Senha: " + getSenha());
33     System.out.println("Bonificação: " + calculaBonificacao());
34 }
35 }

```

Agora vamos trabalhar a classe Secretaria

```

Secretaria.java x
Source History
8 }
9
10 public void setRamal(int ramal) {
11     this.ramal = ramal;
12 }
13
14 @Override
15 public void mostrarDados() {
16     super.mostrarDados();
17     System.out.println("Ramal: " + getRamal());
18     System.out.println("Bonificação: " + calculaBonificacao());
19 }
20 }

```

Agora vamos trabalhar a classe Telefonista

```

Telefonista.java x
Source History
8 }
9
10 public void setEstacaoDeTrabalho(int estacaoDeTrabalho) {
11     this.estacaoDeTrabalho = estacaoDeTrabalho;
12 }
13
14 @Override
15 public void mostrarDados() {
16     super.mostrarDados();
17     System.out.println("Estação de trabalho: " + getEstacaoDeTrabalho());
18     System.out.println("Bonificação: " + calculaBonificacao());
19 }
20 }

```

Modificando agora o TestaFuncionarios, que ficará menor.

```

31 //Mostrar os dados Gerente
32 System.out.println("=====");
33 System.out.println("GERENTE:");
34 g.mostrarDados();
35
36 //Mostrar os dados Telefonista
37 System.out.println("=====");
38 System.out.println("TELEFONISTA:");
39 t.mostrarDados();
40
41 //Mostrar os dados Secretaria
42 System.out.println("=====");
43 System.out.println("SECRETARIA:");
44 s.mostrarDados();
45
46
47 }
48 }

```

Teremos este resultado:

```

Output - Heranca3 (run) x
run:
=====
GERENTE:
Nome: Marcos Costa
Salario: 32450.22
Bonificacao: 19570.132
Usuario: marc
Senha: batatinha
Bonificacao: 19570.132
=====
TELEFONISTA:
Nome: Eliane Prado
Salario: 10780.23
Bonificacao: 1078.023
Estadocivil: 12
Bonificacao: 1078.023
=====
SECRETARIA:
Nome: Nilva Prado
Salario: 6550.99
Bonificacao: 655.099
Ramal: 6677
Bonificacao: 655.099
BUILD SUCCESSFUL (total time: 0 seconds)

```

31.5. Exercício final.

Implementem um sistema (livre escolha) com os conceitos abordados em aula (classe, atributo – visibilidade, método, encapsulamento, herança, *getters* e *setters*, sobrecarga de método e reescrita de método), realizar de forma individual.

Vocês devem subir o projeto desenvolvido ao seu GitHub até o dia 25/09/2025, o restante da aula de hoje e a próxima será exclusivamente para esta entrega que será nossa P1.

Deixarei uma atividade no Microsoft Teams, onde o endereço do repositório deverá ser informado.