

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

YANK, K. *Build your own database driven web site using PHP*. New York: O'Reilly & 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.

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.

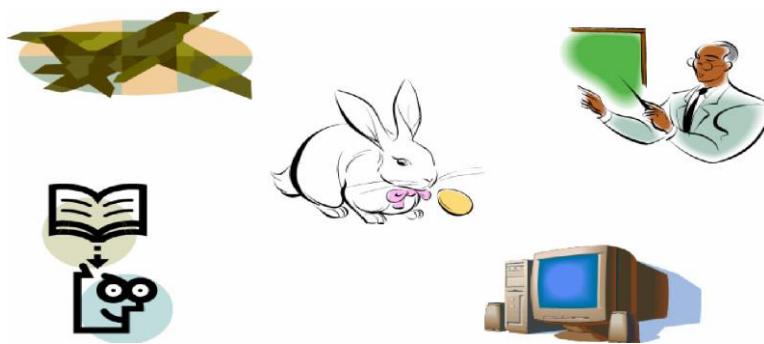
1. Introdução

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

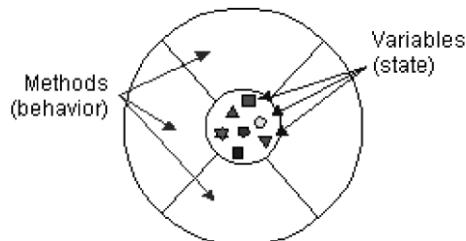
representação da solução para o problema é característica da linguagem usada, tornando a escrita difícil.

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.

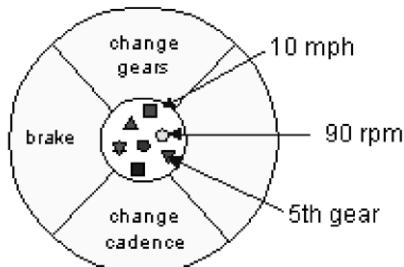
Olhando à sua volta é possível perceber muitos exemplos de objetos do mundo real: seu cachorro, sua mesa, sua televisão, sua bicicleta. Esses objetos têm duas características básicas, que são o estado e o comportamento. Por exemplo, os cachorros tem nome, cor, raça (estados) e eles balançam o rabo, comem, 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 podem fazer (comportamento ou métodos) está contido no próprio objeto. No exemplo da bicicleta, poderemos representar o objeto como no exemplo a seguir:



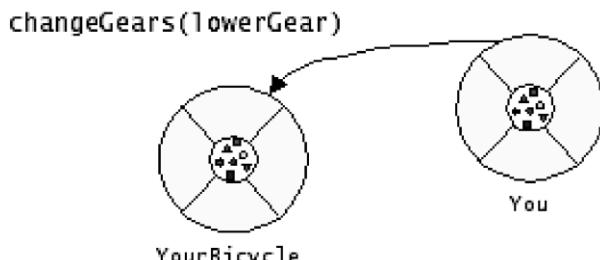
Temos métodos para mudar a marcha, a cadência das pedaladas, e para brecar. A utilização desses métodos altera os valores dos estados. Ao brecar, a velocidade diminui. Ao mudar a marcha, a cadência é alterada. Note que os diagramas mostram que as variáveis do objeto estão no centro do mesmo. Os métodos 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.



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.

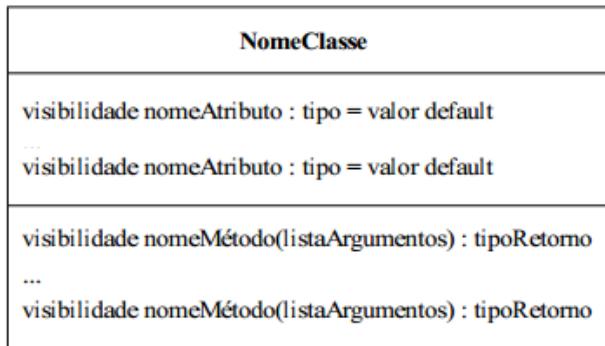
Além da especificação de atributos, a definição de uma classe descreve também qual o comportamento de objetos da classe, ou seja, que funcionalidades podem ser aplicadas a objetos da classe.

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.



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.

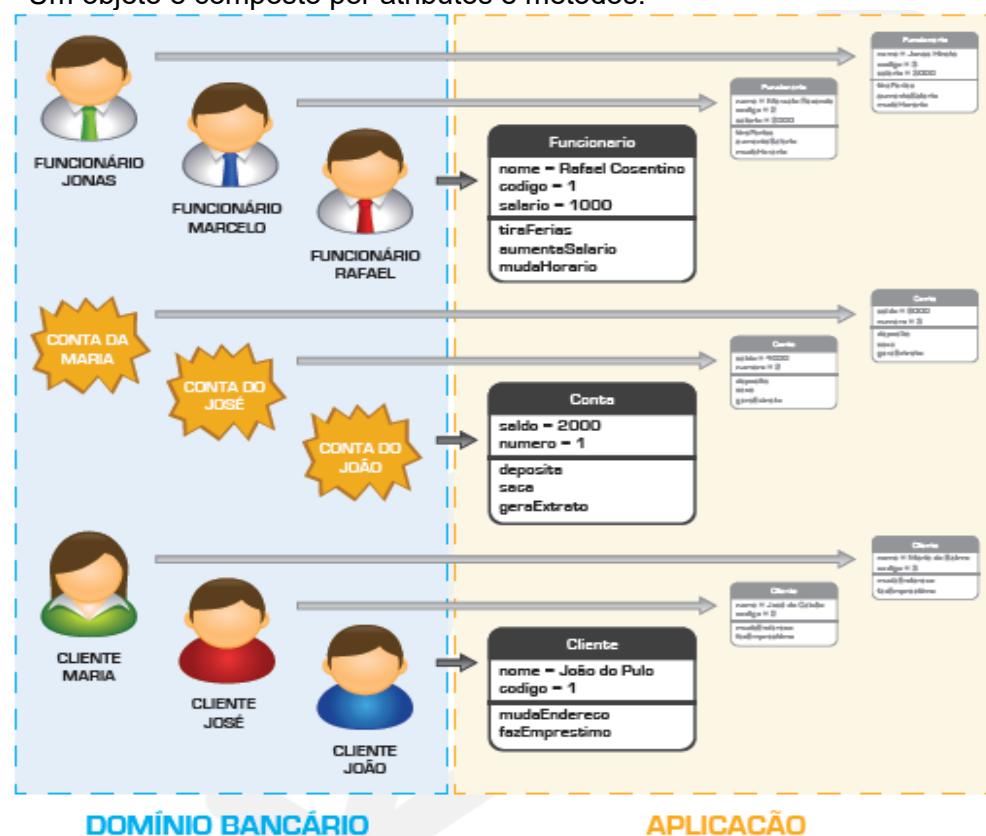
O próprio objeto deve realizar operações de consulta ou alteração dos valores de seus atributos.

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. 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 Orientada à Objetos e multiplataforma....
???????

7. 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”.

```

1 class Conta {
2     double saldo;
3     double limite;
4     int numero;
5 }
```

8. 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 class TestaConta {
2     public static void main(String[] args) {
3         // criando um objeto
4         new Conta();
5     }
6 }
```

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 class TestaConta {
2     public static void main(String[] args) {
3         // criando três objetos
4         new Conta();
5         new Conta();
6         new Conta();
7     }
8 }
```

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.

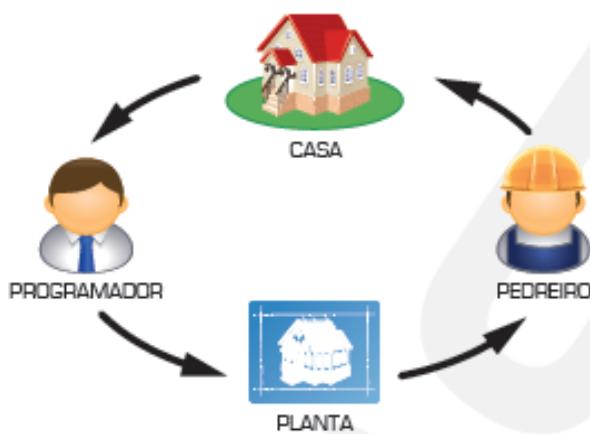
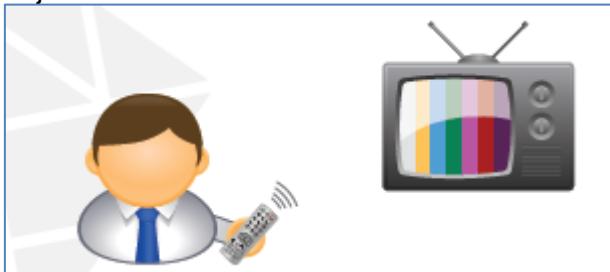


Figura 3.8: Construindo casas

9. 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.



10. 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 Conta referencia = new Conta();
```

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.

11. 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 Conta referencia = new Conta();
2
3 referencia.saldo = 1000.0;
4 referencia.limite = 500.0;
5 referencia.numero = 1;
6
7 System.out.println(referencia.saldo);
8 System.out.println(referencia.limite);
9 System.out.println(referencia.numero);
```

No código acima, o atributo saldo recebe o valor 1000.0. O atributo limite recebe o valor 500 e o numero recebe o valor 1. Depois, os valores são impressos na tela através do comando System.out.println.

12. 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 class Conta {
2     double limite;
3 }
```

```
1 class TestaConta {
2     public static void main(String[] args) {
3         Conta conta = new Conta();
4
5         // imprime 0
6         System.out.println(conta.limite);
7     }
8 }
```

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 class Conta {
2     double limite = 500;
3 }
```

```
1 class TestaConta {
2     public static void main(String[] args) {
3         Conta conta = new Conta();
4
5         // imprime 500
6         System.out.println(conta.limite);
7     }
8 }
```

13. Exercícios de Fixação

1 – Iremos fazer juntos esse projeto, para isso crie um novo projeto chamado **orientacao-a-objetos** para os arquivos desenvolvidos neste exercício.

2 - Implemente uma classe para definir os objetos que representarão os clientes de um banco. Essa classe deve declarar dois atributos: um para os nomes e outro para os códigos dos clientes. Adicione o seguinte arquivo na pasta **orientacao-a-objetos**.

```
1 class Cliente {
2     String nome;
3     int codigo;
4 }
```

3 - Faça um teste criando dois objetos da classe Cliente. Adicione o seguinte arquivo na pasta **orientacao-a-objetos**.

```
1 class TestaCliente {
2     public static void main(String[] args) {
3         Cliente c1 = new Cliente();
4         c1.nome = "Rafael Cosentino";
5         c1.codigo = 1;
6
7         Cliente c2 = new Cliente();
8         c2.nome = "Jonas Hirata";
9         c2.codigo = 2;
10
11         System.out.println(c1.nome);
12         System.out.println(c1.codigo);
13
14         System.out.println(c2.nome);
15         System.out.println(c2.codigo);
16     }
17 }
```

Execute a classe TestaCliente.

4 - 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 para modelar os objetos que representarão os cartões de crédito. Adicione o seguinte arquivo na pasta **orientacao-a-objetos**.

```
1 class CartaoDeCredito {  
2     int numero;  
3     String dataDeValidade;  
4 }
```

5 - Faça um teste criando dois objetos da classe CartaoDeCredito. Altere e imprima os atributos desses objetos. Adicione o seguinte arquivo na pasta **orientacao-a-objetos**.

```
1 class TestaCartaoDeCredito {  
2     public static void main(String[] args) {  
3         CartaoDeCredito cdc1 = new CartaoDeCredito();  
4         cdc1.numero = 111111;  
5         cdc1.dataDeValidade = "01/01/2013";  
6  
7         CartaoDeCredito cdc2 = new CartaoDeCredito();  
8         cdc2.numero = 222222;  
9         cdc2.dataDeValidade = "01/01/2014";  
10  
11         System.out.println(cdc1.numero);  
12         System.out.println(cdc1.dataDeValidade);  
13  
14         System.out.println(cdc2.numero);  
15         System.out.println(cdc2.dataDeValidade);  
16     }  
17 }
```

Execute a classe TestaCartaoDeCredito.

6 - As agências do banco possuem número. Crie uma classe para definir os objetos que representarão as agências.

```
1 class Agencia {  
2     int numero;  
3 }
```

7 - Faça um teste criando dois objetos da classe Agencia. Altere e imprima os atributos desses objetos. Adicione o seguinte arquivo na pasta **orientacao-a-objetos**.

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

Execute a classe TestaAgencia.

8 - As contas do banco possuem número, saldo e limite. Crie uma classe para definir os objetos que representarão as contas.

```

1 class Conta {
2     int numero;
3     double saldo;
4     double limite;
5 }
```

9 - Faça um teste criando dois objetos da classe Conta. Altere e imprima os atributos desses objetos. Adicione o seguinte arquivo na pasta orientacao-a-objetos.

```

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

Execute a classe TestaConta.

10 - Faça um teste que imprima os atributos de um objeto da classe Conta logo após a sua criação.

```

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

Execute a classe TestaValoresPadrao

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

```

1 class Conta {
2     int numero;
3     double saldo;
4     double limite = 100;
5 }
```

Execute a classe TestaValoresPadrao.

14. 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 }
```

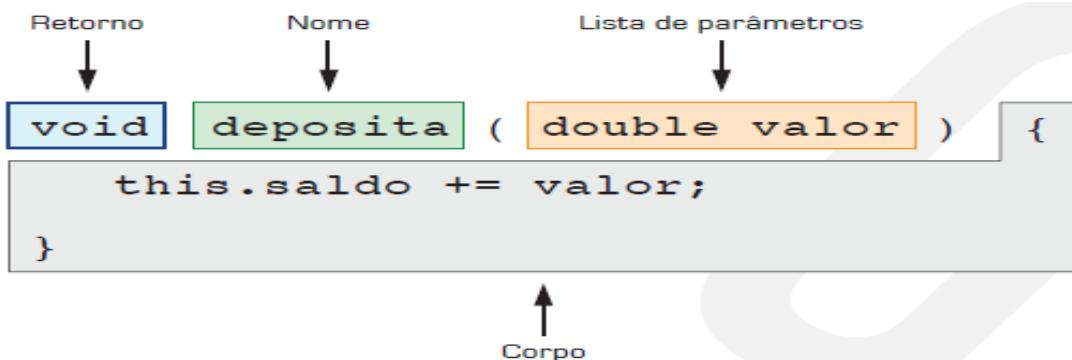
Podemos dividir um método em quatro partes:

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 com a 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.

```
1 // Referência de um objeto
2 Conta c = new Conta();
3
4 // Chamando o método deposita()
5 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`.

```
1 void deposita(double valor) {
2     this.saldo += valor;
3 }
```

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.

```

1 double consultaSaldoDisponivel() {
2     return this.saldo + this.limite;
3 }
```

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

```

1 Conta c = new Conta();
2 c.deposita(1000);
3
4 // Armazenando a resposta de um método em uma variável
5 double saldoDisponivel = c.consultaSaldoDisponivel();
6
7 System.out.println("Saldo Disponível: " + this.saldoDisponivel);
```

15. Exercícios de Fixação

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

```

1 class Conta {
2     int numero;
3     double saldo;
4     double limite = 100;
5     Agencia agencia;
6
7     // ADICIONE OS MÉTODOS ABAIXO
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`.

```

1 class TestaMetodosConta {
2     public static void main(String[] args) {
3         Conta c = new Conta();
4
5         c.deposita(1000);
6         c.imprimeExtrato();
7
8         c.saca(100);
9         c.imprimeExtrato();
10
11        double saldoDisponivel = c.consultaSaldoDisponivel();
12        System.out.println("SALDO DISPONÍVEL: " + saldoDisponivel);
13    }
14 }
```

16. Exercícios Complementares

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.

Atenção: Desenvolva os exercícios propostos e verificaremos em aula.

17. Sobrecarga (Overloading)

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 class Conta {
2     double saldo;
3     double limite;
4
5     void imprimeExtrato(int dias){
6         // extrato
7     }
8 }
```

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 class Conta {
2     double saldo;
3     double limite;
4
5     void imprimeExtrato(){
6         // extrato dos últimos 15 dias
7     }
8
9     void imprimeExtrato(int dias){
10        // extrato
11    }
12 }
```

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.

```

1 class Conta {
2
3     void imprimeExtrato(){
4         this.imprimeExtrato(15);
5     }
6
7     void imprimeExtrato(int dias){
8         // extrato
9     }
10 }
```

16.1. Exercícios de Fixação

1-) Crie uma classe 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.

```

1 class Gerente {
2     String nome;
3     double salario;
4
5     void aumentaSalario() {
6         this.aumentaSalario(0.1);
7     }
8
9     void aumentaSalario(double taxa) {
10        this.salario += this.salario * taxa;
11    }
12 }
```

Teste os métodos de aumento salarial definidos na classe Gerente.

```

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

Compile e execute a classe TestaGerente.

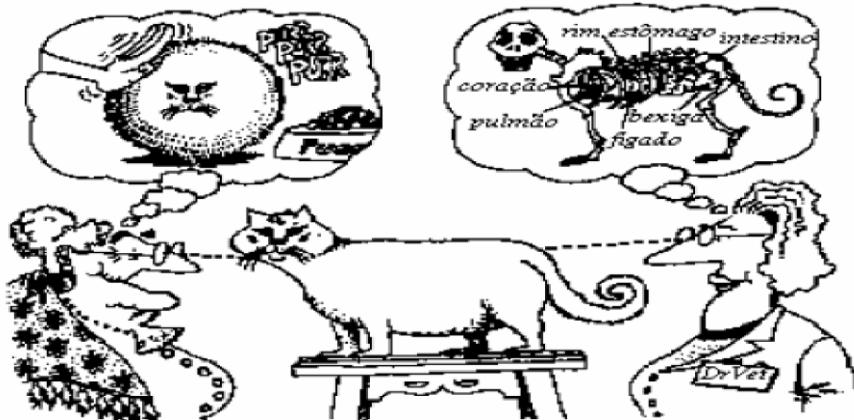
18. Abstração

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.

19. Atributos Privados

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

```
class Funcionario {
    double salario;
}
```

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.

```
class Funcionario {
    private double salario;

    void aumentaSalario(double aumento) {
        // lógica para aumentar o salário
    }
}
```

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.

18.1. 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.

```
class Conta {
    private double saldo;

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

    void saca(double valor) {
        this.saldo -= valor;
        this.descontaTarifa();
    }

    void descontaTarifa() {
        this.saldo -= 0.1;
    }
}
```

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.

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

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

18.2. Métodos Públicos

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

```
class Conta {
    private double saldo;

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

    public void saca(double valor) {
        this.saldo -= valor;
        this.descontaTarifa();
    }

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

18.2.1. 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.

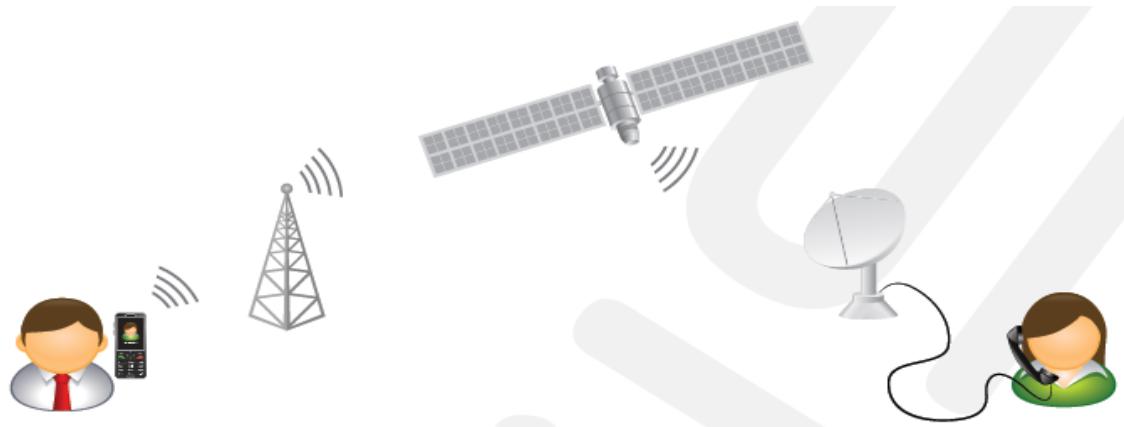
O conceito de encapsulamento pode ser identificado em diversos exemplos do cotidiano. Mostraremos alguns desses exemplos para esclarecer melhor a ideia.

18.2.2. 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.



18.2.3. 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).

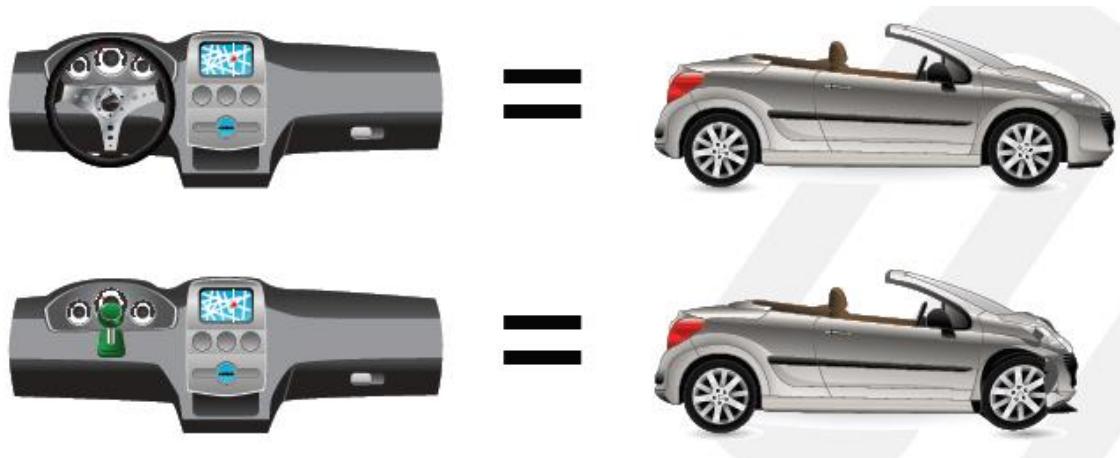
A implementação do carro é composta pelos dispositivos internos (motor, caixa de câmbio, radiador, sistema de injeção eletrônica ou carburador, etc) e pelos processos realizados internamente por esses dispositivos.

Nos carros mais抗igos, 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.



18.2.4. 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.



18.2.4. 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.

```
class Cliente {
    private String nome;

    public String consultaNome() {
        return this.nome;
    }
}
```

```
class Cliente {
    private String nome;

    public void alteraNome(String nome){
        this.nome = nome;
    }
}
```

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, consequentemente, 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.

19. 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.

```
class Cliente {  
    private String nome;  
  
    public String getName() {  
        return this.nome;  
    }  
  
    public void setName(String nome) {  
        this.nome = nome;  
    }  
}
```

19.1. 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.

```
public class Funcionario {  
    double salario;  
    String nome;  
}
```

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

```
class Teste {  
    public static void main(String[] args) {  
        Funcionario f = new Funcionario();  
  
        f.nome = "Rafael Cosentino";  
        f.salario = 2000;  
  
        System.out.println(f.nome);  
        System.out.println(f.salario);  
    }  
}
```

- 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.

```
class Funcionario {  
    private double salario;  
    private String nome;  
}
```

- 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.

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

```

class Funcionario {
    private double salario;
    private String nome;

    public double getSalario() {
        return this.salario;
    }

    public String getName() {
        return this.nome;
    }

    public void setSalario(double salario) {
        this.salario = salario;
    }

    public void setName(String nome) {
        this.nome = nome;
    }
}

```

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.

```

class Teste {
    public static void main(String[] args) {
        Funcionario f = new Funcionario();

        f.setName("Rafael Cosentino");
        f.setSalario(2000);

        System.out.println(f.getName());
        System.out.println(f.getSalario());
    }
}

```

20. HERANÇA

20.1. 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.

Uma classe para todos os serviços

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

```

1 class Servico {
2     private Cliente contratante;
3     private Funcionario responsavel;
4     private String dataDeContratacao;
5
6     // métodos
7 }

```

20.2. 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.

```
1 class Servico {  
2     // GERAL  
3     private Cliente contratante;  
4     private Funcionario responsavel;  
5     private String dataDeContratacao;  
6  
7     // EMPRÉSTIMO  
8     private double valor;  
9     private double taxa;  
10  
11    // métodos  
12}
```

20.3. 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.

```
1 class Servico {  
2     // GERAL  
3     private Cliente contratante;  
4     private Funcionario responsavel;  
5     private String dataDeContratacao;  
6  
7     // EMPRÉSTIMO  
8     private double valor;  
9     private double taxa;  
10  
11    // SEGURO DE VEÍCULO  
12    private Veiculo veiculo;  
13    private double valorDoSeguroDeVeiculo;  
14    private double franquia;  
15  
16    // métodos  
17}
```

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.

20.4. 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.

```

1 class SeguroDeVeiculo {
2     // GERAL
3     private Cliente contratante;
4     private Funcionario responsavel;
5     private String dataDeContratacao;
6
7     // SEGURO DE VEICULO
8     private Veiculo veiculo;
9     private double valorDoSeguroDeVeiculo;
10    private double franquia;
11
12    // métodos
13 }

```

```

1 class Emprestimo {
2     // GERAL
3     private Cliente contratante;
4     private Funcionario responsavel;
5     private String dataDeContratacao;
6
7     // EMPRÉSTIMO
8     private double valor;
9     private double taxa;
10
11    // métodos
12 }

```

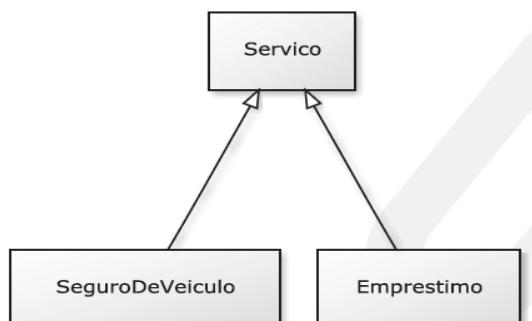
Criar uma classe para cada serviço torna o sistema mais flexível, pois qualquer alteração em um determinado serviço não causará efeitos colaterais nos outros. Mas, por outro lado, essas classes teriam bastante código repetido, contrariando a ideia do DRY. Além disso, qualquer alteração que deva ser realizada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

20.5. 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 abaixo.



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

```

1 Emprestimo e = new Emprestimo();
2
3 // Chamando um método da classe Servico
4 e.setDataDeContratacao("10/10/2010");
5
6 // Chamando um método da classe Emprestimo
7 e.setValor(10000);

```

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

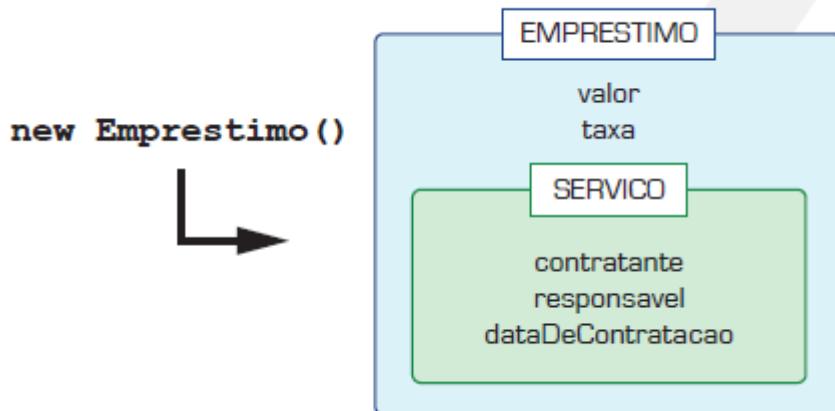
```
1 class Servico {
2     private Cliente contratante;
3     private Funcionario responsavel;
4     private String dataDeContratacao;
5 }
```

```
1 class Emprestimo extends Servico {
2     private double valor;
3     private double taxa;
4 }
```

```
1 class SeguroDeVeiculo extends Servico {
2     private Veiculo veiculo;
3     private double valorDoSeguroDeVeiculo;
4     private double franquia;
5 }
```

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 herdam da classe Servico.

```

1 class Servico {
2     // ATRIBUTOS
3
4     public double calculaTaxa() {
5         return 10;
6     }
7 }
```

```

1 Emprestimo e = new Emprestimo();
2
3 SeguroDeVeiculo sdv = new SeguroDeVeiculo();
4
5 System.out.println("Emprestimo: " + e.calculaTaxa());
6
7 System.out.println("SeguroDeVeiculo: " + sdv.calculaTaxa());
```

21. 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 Emprestimo.

```

1 class Emprestimo extends Servico {
2     // ATRIBUTOS
3
4     public double calculaTaxaDeEmprestimo() {
5         return this.valor * 0.1;
6     }
7 }
```

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().

```

1 class Emprestimo extends Servico {
2     // ATRIBUTOS
3
4     public double calculaTaxa() {
5         return 5 + this.valor * 0.1;
6     }
7 }
1 class SeguroDeVeiculo extends Servico {
2     // ATRIBUTOS
3
4     public double calculaTaxa() {
5         return 5 + this.veiculo.getTaxa() * 0.05;
6     }
7 }
```

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.

```

1 class Servico {
2     public double calculaTaxa() {
3         return 5 ;
4     }
5 }
```

```

1 class Emprestimo extends Servico {
2     // ATRIBUTOS
3
4     public double calculaTaxa() {
5         return super.calculaTaxa() + this.valor * 0.1;
6     }
7 }
```

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

21.2. 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 e da classe Servico devem ser executados. Além disso, os construtores das classes mais genéricas são chamados antes dos construtores das classes específicas.

```

1 class Servico {
2     // ATRIBUTOS
3
4     public Servico() {
5         System.out.println("Servico");
6     }
7 }
```

```

1 class Emprestimo extends Servico {
2     // ATRIBUTOS
3
4     public Emprestimo() {
5         System.out.println("Emprestimo");
6     }
7 }
```

21.3. 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 class Funcionario {
2     private String nome;
3     private double salario;
4
5     // GETTERS AND SETTERS
6 }
```

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 class Gerente extends Funcionario {
2     private String usuario;
3     private String senha;
4
5     // GETTERS AND SETTERS
6 }
```

```

1 class Telefonista extends Funcionario {
2     private int estacaoDeTrabalho;
3
4     // GETTERS AND SETTERS
5 }
```

```

1 class Secretaria extends Funcionario {
2     private int ramal;
3
4     // GETTERS AND SETTERS
5 }
```

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

```

1 class TestaFuncionarios {
2     public static void main(String[] args) {
3         Gerente g = new Gerente();
4         g.setNome("Rafael Cosentino");
5         g.setSalario(2000);
6         g.setUsuario("rafael.cosentino");
7         g.setSenha("12345");
8
9         Telefonista t = new Telefonista();
10        t.setNome("Carolina Mello");
11        t.setSalario(1000);
12        t.setEstacaoDeTrabalho(13);
13
14         Secretaria s = new Secretaria();
15         s.setNome("Tatiane Andrade");
16         s.setSalario(1500);
17         s.setRamal(198);
18
19         System.out.println("GERENTE");
20         System.out.println("Nome: " + g.getNome());
21         System.out.println("Salário: " + g.getSalario());
22         System.out.println("Usuário: " + g.getUsuario());
23         System.out.println("Senha: " + g.getSenha());
24
25         System.out.println("TELEFONISTA");
26         System.out.println("Nome: " + t.getNome());
27         System.out.println("Salário: " + t.getSalario());
28         System.out.println("Estacao de trabalho: " + t.getEstacaoDeTrabalho());
29
30         System.out.println("SECRETARIA");
31         System.out.println("Nome: " + s.getNome());
32         System.out.println("Salário: " + s.getSalario());
33         System.out.println("Ramal: " + s.getRamal());
34     }
35 }
```

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 class Funcionario {  
2     private String nome;  
3     private double salario;  
4  
5     public double calculaBonificacao() {  
6         return this.salario * 0.1;  
7     }  
8  
9     // GETTERS AND SETTERS  
10}
```

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 class TestaFuncionarios {  
2     public static void main(String[] args) {  
3         Gerente g = new Gerente();  
4         g.setNome("Rafael Cosentino");  
5         g.setSalario(2000);  
6         g.setUsuario("rafael.cosentino");  
7         g.setSenha("12345");  
8  
9         Telefonista t = new Telefonista();  
10        t.setNome("Carolina Mello");  
11        t.setSalario(1000);  
12        t.setEstacaoDeTrabalho(13);  
13  
14        Secretaria s = new Secretaria();  
15        s.setNome("Tatiane Andrade");  
16        s.setSalario(1500);  
17        s.setRamal(198);  
18  
19        System.out.println("GERENTE");  
20        System.out.println("Nome: " + g.getNome());  
21        System.out.println("Salário: " + g.getSalario());  
22        System.out.println("Usuário: " + g.getUsuario());  
23        System.out.println("Senha: " + g.getSenha());  
24        System.out.println("Bonificação: " + g.calculaBonificacao());  
25  
26        System.out.println("TELEFONISTA");  
27        System.out.println("Nome: " + t.getNome());  
28        System.out.println("Salário: " + t.getSalario());  
29        System.out.println("Estação de trabalho: " + t.getEstacaoDeTrabalho());  
30        System.out.println("Bonificação: " + t.calculaBonificacao());  
31  
32        System.out.println("SECRETARIA");  
33        System.out.println("Nome: " + s.getNome());  
34        System.out.println("Salário: " + s.getSalario());  
35        System.out.println("Ramal: " + s.getRamal());  
36        System.out.println("Bonificação: " + s.calculaBonificacao());  
37    }  
38}
```

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 class Gerente extends Funcionario {
2     private String usuario;
3     private String senha;
4
5     public double calculaBonificacao() {
6         return this.getSalario() * 0.6 + 100;
7     }
8
9     // GETTERS AND SETTERS
10}

```

21.4. 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().

21.5. 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, podem realizar em duplas).

Vocês terão a aula de hoje para implementar e entregar na semana dia 12/09/2024.

22. Prática do começo ao final

Nesse momento iremos implementar um projeto de login simples, onde abordaremos a Orientação a Objetos com banco de dados e tela, tudo isso confeccionado em Java, e fazendo um *CRUD* (*Create, Read, Update and Delete*) em uma aplicação.

22.1. Banco de dados

Como sempre disse, nosso sistema precisa começar pelo levantamento de requisitos e posteriormente, antes de programarmos qualquer coisa, precisamos implementar nosso banco de dados, iremos utilizar o MySQL, e iremos fazer somente uma tabela, que será o usuário de nosso sistema. Vejam abaixo:

```

1      #CRIAÇÃO DO BANCO DE DADOS BD_LOGIN
2 •    create database bd_login;
3
4      #SELECIONANDO O BANCO DE DADOS
5 •    use bd_login;
6
7      #CRIANDO A TABELA DE LOGIN
8 •    create table usuario(
9          usuario varchar(10) primary key,
10         nome   varchar(30) default '',
11         senha  char(32) default ''
12     );

```

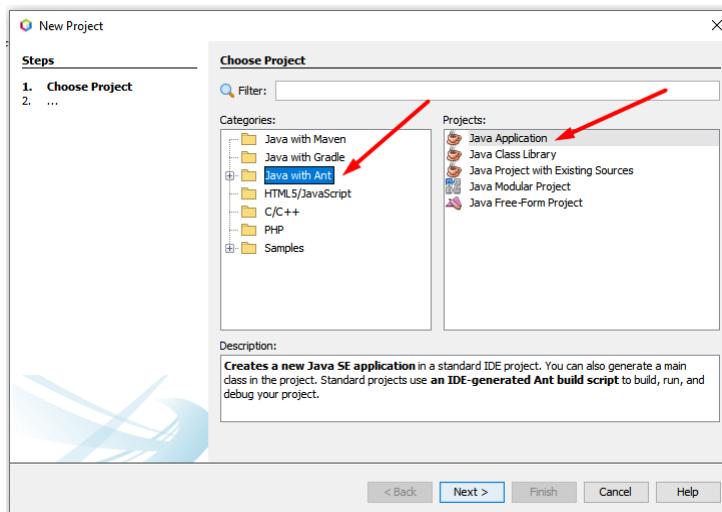
Vocês viram que usamos para inserir o usuário em nossa tabela uma função MD5, correto? Mas o que ela significa?

O MD5 é um dos algoritmos de criptografia mais conhecidos que funciona através de chamada de função como fizemos acima, e seu funcionamento consiste na geração de um valor hexadecimal de 32 dígitos a partir de uma *String*, por isso que criamos o campo senha com tamanho de 32. Esse processo, no entanto, é unidirecional, ou seja, uma vez aplicada, essa função criptográfica não pode ser revertida a fim de recuperar o valor original, ou seja, se o usuário esquecer essa senha, nem nós como administradores do banco de dados conseguiremos passar a senha para o usuário, o que força fazer o *reset* da mesma.

Isso nos ajuda nos quesitos de segurança (principalmente para salvar dados sensíveis como a senha do usuário), já que mesmo que os dados criptografados sejam hackeados, o invasor não terá como descriptografá-los. Para aplicar esse algoritmo no MySQL, dispomos da função MD5, que recebe como único parâmetro o texto a ser criptografado e retorna o valor mascarado. Assim, podemos utilizá-la diretamente ao inserir um registro na tabela, como fizemos anteriormente.

22.2. Projeto Login

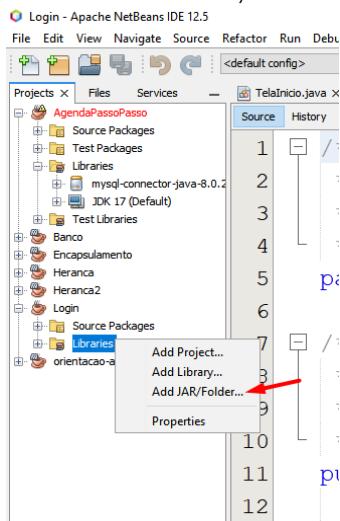
Nesse momento no *NetBeans* iremos criar um novo projeto que chamaremos de Login.



22.3. Conectando nossa aplicação ao banco de dados

Nesse momento precisamos anexar uma biblioteca do MySQL ao nosso projeto para que aconteça a integração entre banco de dados e projeto.

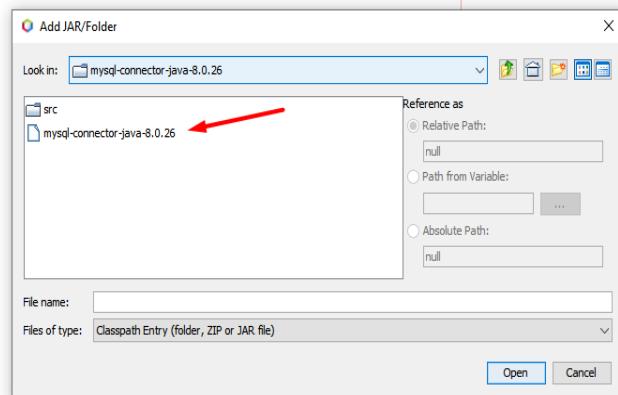
Para fazermos tal, vamos olhar a pasta *Libraries* (biblioteca) dentro de nosso projeto:



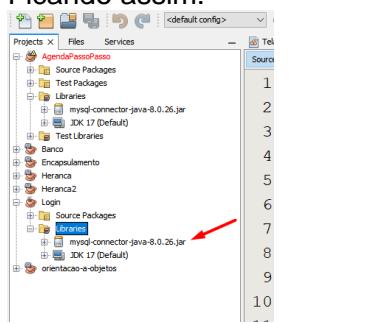
Selecionamos **mysql-connector-java**, se vocês não tiverem esse arquivo eu disponibilizo o mesmo, e coloquem este arquivo na mesma pasta do seu projeto.

Este arquivo é um JAR (Java ARchive) é um arquivo compactado no formato ZIP que contém um conjunto de classes (arquivos ".class") e arquivos de configuração. O formato JAR é utilizado para permitir a distribuição de bibliotecas ou, até mesmo, sistemas inteiros em Java (uma vez que é possível montar arquivos JAR formados por centenas de classes).

As principais bibliotecas Java desenvolvidas por terceiros – ou seja, bibliotecas que não fazem parte da JDK - são estruturadas e distribuídas em arquivos JAR.



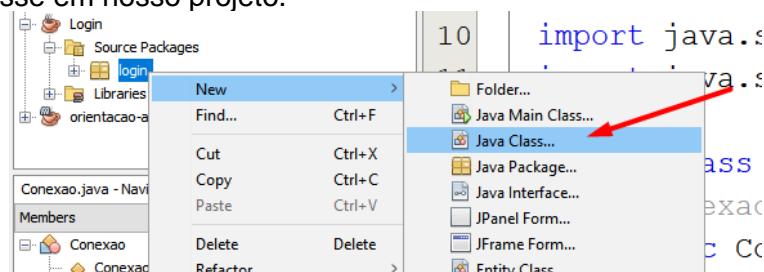
Ficando assim:

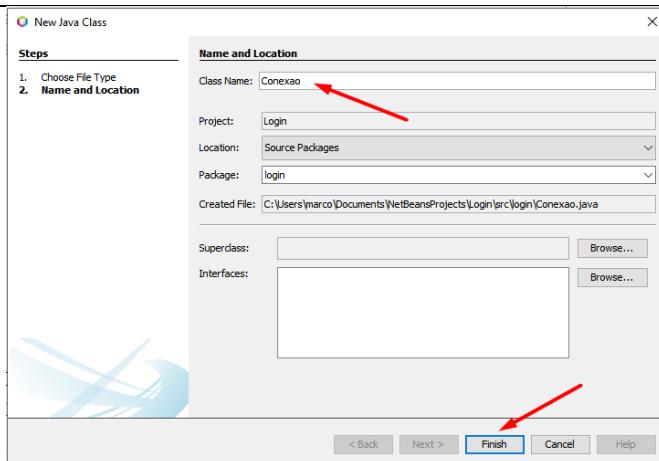


Pronto, nosso projeto já possui a biblioteca MySQL para fazermos a conexão com um banco de dados.

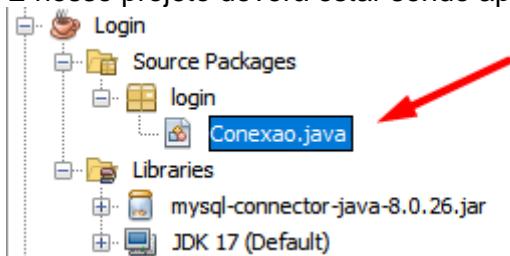
22.3.1. Classe Conexao

Agora iremos codificar nossa classe de conexão, para isso vamos adicionar uma nova classe em nosso projeto.





E nosso projeto deverá estar sendo apresentado assim:



Iremos agora codificar a classe Conexao.java, nela faremos a importação de bibliotecas (conceito que vimos em DS I) de SQL (Banco de Dados) essas bibliotecas nos auxiliarão a trabalhar com comandos, drivers, parâmetros e resultados da base de dados que criamos, vejam:

```

package Login;
/*
 * @author Professor Marcos Costa
 */
/*
 Bibliotecas para trabalharem com as funções
 de banco de dados
 */
import java.sql.Connection; //Coneção com o banco de dados
import java.sql.DriverManager; //Driver de conexão, em nosso caso MySQL
import java.sql.ResultSet; //Resultados das operações em banco de dados
import java.sql.SQLException;
import java.sql.Statement; // Interpretação dos comandos SQL - CRUD

public class Conexao {
    //Atributos de conexao ligados as bibliotecas importadas
    public Connection con = null;
    public Statement stmt = null;
    public ResultSet resultset = null;

    //Atributos de conexão
    private final String servidor = "jdbc:mysql://127.0.0.1:3308/login";
    //Servidor de banco de dados
    private final String usuario = "root"; //Usuário do banco de dados
    private final String senha = ""; //Senha do banco de dados
    private final String driver = "com.mysql.cj.jdbc.Driver"; //Driver de conexão
}

```

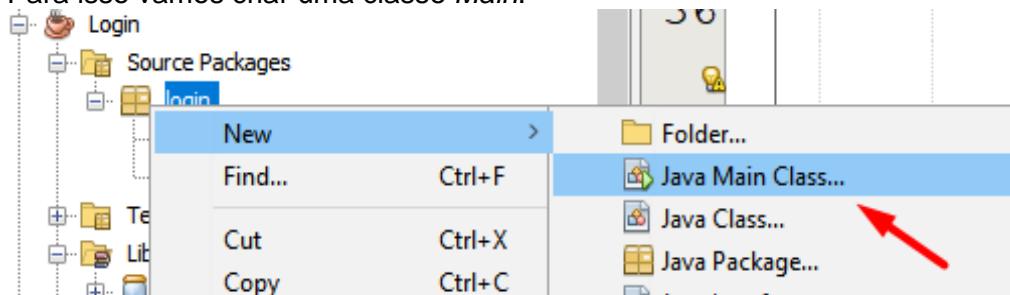
```

29     //Método de abertura da conexão com banco de dados
30     public Connection abrirConexao() {
31         try {
32             Class.forName(driver); //Driver de utilização
33             //Atributos de conexão
34             con = DriverManager.getConnection(servidor, usuario, senha);
35             stmt = con.createStatement(); //Fazendo conexão com o banco de dados
36             //Mensagem informando que a conexão foi realizada com sucesso
37             System.out.println("Conexão aberta com sucesso");
38         } catch ( ClassNotFoundException | SQLException e) {
39             //Mensagem de saída caso haja erro
40             System.out.println("Erro ao acessar banco de dados, verifique! " + e.getMessage());
41         }
42         return con; //Retorna a conexão
43     }
44
45     //Método de fechamento da conexão com o banco de dados
46     public void fecharConexao() {
47         try {
48             con.close(); //Fechamento da conexão com o banco de dados
49             //Mensagem que fechou a conexão
50             System.out.println("Conexão finalizada com sucesso");
51         } catch ( SQLException e ) {
52             //Mensagem de erro ao fechar conexão
53             System.out.println("Erro ao encerrar conexão " + e.getMessage());
54         }
55     }
56 }
```

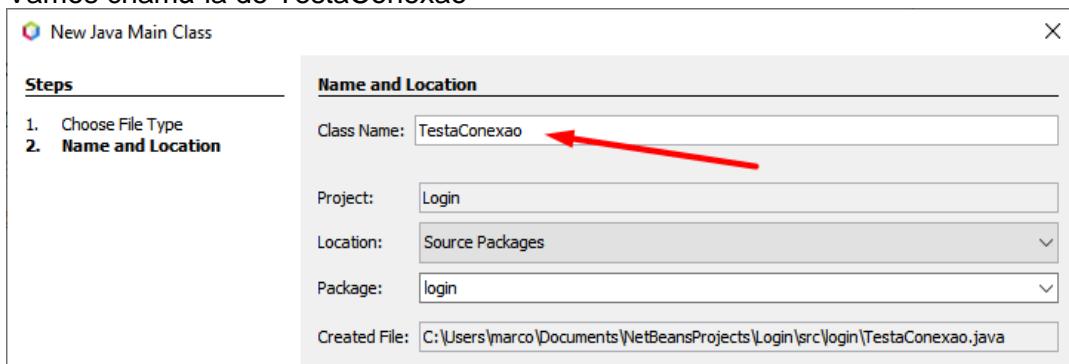
Um bloco **try** é chamado de bloco protegido porque, caso ocorra algum problema com os comandos dentro do bloco, a execução desviará para os blocos **catch** correspondentes, conforme fizemos acima, ou seja, se der algum erro nesse trecho de código, ele “escapa” para o **catch** e mostra o erro.

Muito bem, assim finalizamos nossa classe de conexão para banco de dados, vamos testar? Eu já respondo, é claro!!!

Para isso vamos criar uma classe **Main**:



Vamos chamá-la de **TestaConexao**



Ficando assim:



Agora vamos codificar a mesma fazendo o código abaixo:

```

4  /*
5   * Teste para verificarmos se o Banco conecta e desconecta
6   */
7   public class Testaconexao {
8       /*
9        * Teste para verificarmos se o Banco conecta e desconecta
10       */
11      public static void main(String[] args) {
12          Conexao c = new Conexao(); // Instanciamos o objeto c de Conexao
13          c.abrirConexao(); //Chamamos o método de abertura da conexão do BD.
14
15          try {
16              Thread.sleep(4000); //Fazemos uma pausa de 4 segundos
17              c.fecharConexao(); //Fechamos a conexão
18          } catch (InterruptedException ex) {
19              //Mensagem de saída caso haja erro
20              System.out.println("Houve algum problema no teste de conexão. " + ex.getMessage());
21      }
22  }

```

Feito isso, podemos executar essa classe, a mesma fará a abertura da conexão com o banco de dados, aguardará 4 segundos e fechará a mesma, veja o resultado abaixo:

```

run:
Conexao aberta com sucesso
Conexao finalizada com sucesso
BUILD SUCCESSFUL (total time: 4 seconds)

```

22.3.2. Método construtor e *Toolkits*

Antes de continuarmos vamos ver algumas definições importantes no Java, vamos ver um conceito sobre Método Construtor.

Em Java, o método construtor é definido como um método cujo nome deve ser o mesmo nome da classe e sem indicação do tipo de retorno -- nem mesmo *void*. O construtor é unicamente invocado no momento da criação do objeto através do operador *new*. O retorno do operador *new* é uma referência para o objeto recém criado.

Outros dois elementos importantes a aprender é o AWT e o SWING, eles são *toolkits* (kits de ferramentas) que torna possível a criação de ambientes gráficos (janelas do Windows) no Java, quando ouvimos falar em GUI (*Graphical User Interface*).

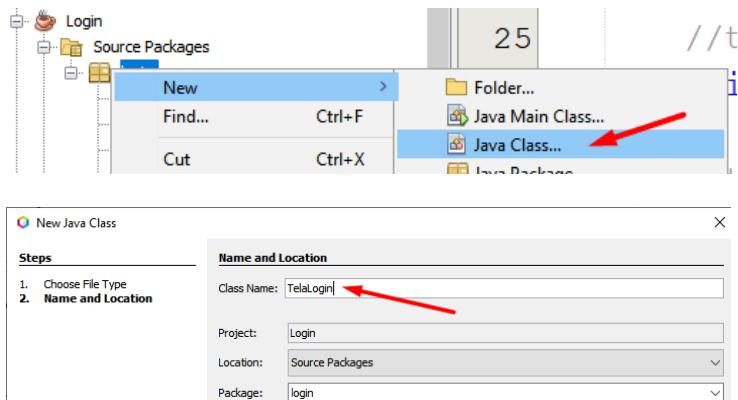
AWT significa *Abstract Window Toolkit*. É uma API dependente de plataforma para desenvolver GUI ou aplicativos baseados em janela em Java. Foi desenvolvido pela *Sun Microsystems* em 1995. É de uso pesado porque é gerado pelo sistema operacional host do sistema. Ele contém um grande número de classes e métodos, que são usados para criar e gerenciar **GUI**.

Já o **Swing** é uma interface gráfica de usuário (GUI) Java leve que é usada para criar vários aplicativos. O **Swing** possui componentes que são independentes de plataforma. Ele permite ao usuário criar botões e barras de rolagem. O **Swing** inclui

pacotes para a criação de aplicativos de desktop em Java. Os componentes do Swing são escritos na linguagem Java. É uma parte do *Java Foundation Classes* (JFC).

22.3.3. Classe TelaLogin

Agora iremos implementar uma classe que fará o desenho da tela de nosso projeto, vamos inserir uma nova classe, a TelaLogin.java.



Vamos codificar essa classe.

```

1 package Login;
2 /**
3 *
4 * @author Professor Marcos Costa
5 */
6 /*
7 Importações de todas as bibliotecas que estou utilizando na classe
8 */
9 import java.awt.Font; //Trabalhar com fontes
10 import java.awt.SystemColor; //Trabalhar com cores
11 import javax.swing.JButton; //Trabalhar com botões
12 import javax.swing.JFrame; //Trabalhar com frames
13 import javax.swing.JLabel; //Trabalhar com labels
14 import javax.swing.JPanel; //Trabalhar com painéis
15 import javax.swing.JPasswordField; //Trabalhar com campos de senha
16 import javax.swing.JTextField; //Trabalhar com campos de texto
17
18 public class TelaLogin extends JFrame{
19
20     //tela Objeto JPanel (tela em si)
21     private final JPanel panelTela;
22
23     //txtusuario Objeto JTextField (campo na tela)
24     private final JTextField txtUsuario;
25
26     //pswSenha Objeto PasswordField (campo na tela)
27     private final JPasswordField pswSenha;
28
29     //Método construtor
30     public TelaLogin() {
31         //coloca o objeto na referencia do centro da tela
32         setLocationRelativeTo(null);
33
34         //não permite que o objeto seja expandido
35         setResizable(false);

```

```

36
37     //coloca título na caixa JFrame
38     setTitle("Login - Senac");
39
40     //Quando clicado no X eu encerro todo o programa
41     setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
42
43     //define posicionamento e tamanho
44     //      x      y      width height
45     setBounds(500, 200, 426, 212);
46
47     //Crio um objeto JPanel e atribuo ele a variável tela
48     panelTela = new JPanel();
49
50     //Define a cor de fundo do JPanel (tela)
51     panelTela.setBackground(SystemColor.gray);
52     setContentPane(panelTela);
53
54     //Vou utilizar o meu panel sem utilizar o padrão
55     panelTela.setLayout(null);
56
57     //Adicionando elementos na tela:
58     // Criando um objeto do tipo JLabel e atribuindo o valor ao atributo
59     JLabel lblIdentificacao = new JLabel("IDENTIFICAÇÃO");
60
61     //Localização na tela
62     lblIdentificacao.setBounds(144, 0, 160, 39);
63
64     //Definindo a Fonte
65     lblIdentificacao.setFont(new Font("Arial", 3, 19));
66
67     //Adicionando o meu label ao meu Panel
68     panelTela.add(lblIdentificacao);
69
70     //Identificação e Posicionamento dos labels
71     JLabel lblUsuario = new JLabel("Usuário");
72     lblUsuario.setBounds(24, 65, 70, 15);
73     panelTela.add(lblUsuario);
74
75     JLabel lblSenha = new JLabel("Senha");
76     lblSenha.setBounds(24, 92, 70, 15);
77     panelTela.add(lblSenha);
78
79     //Identificação e Posicionamento dos texts fields
80     txtUsuario = new JTextField();
81     txtUsuario.setBounds(112, 63, 219, 19);
82     panelTela.add(txtUsuario);
83     txtUsuario.setColumns(10);
84
85     pswSenha = new JPasswordField();
86     pswSenha.setBounds(112, 90, 219, 19);
87     panelTela.add(pswSenha);
88
89     //Identificação e Posicionamento dos botões
90     JButton btnEntrar = new JButton("Entrar");
91     btnEntrar.setBounds(200, 136, 117, 25);
92     panelTela.add(btnEntrar);
93
94     JButton btnCadastrar = new JButton("Cadastrar");
95     btnCadastrar.setBounds(50, 136, 117, 25);
96     panelTela.add(btnCadastrar);
97 }
```

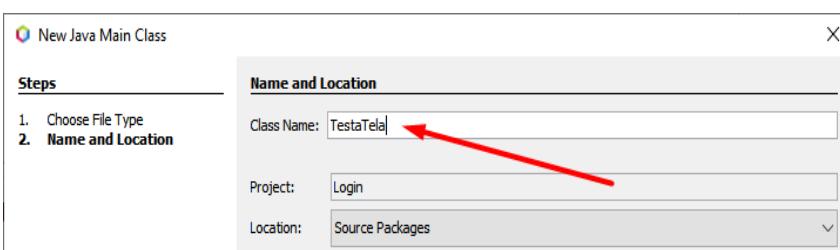
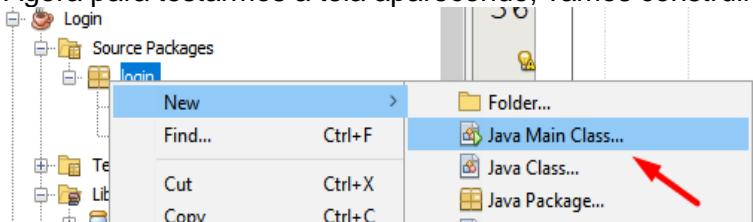
```

98
99     public void abreTela() {
100         TelaLogin tela = new TelaLogin();
101         tela.setVisible(true);
102     }

```

22.3.4. Classe TestaTela

Agora para testarmos a tela aparecendo, vamos construir uma classe *Main*, assim:



Abaixo codificamos a classe *TestaTela.java*:

```

1 package Login;
2
3 import javax.swing.JFrame;
4
5 /**
6 * @author Professor Marcos Costa
7 */
8
9 public class TestaTela extends JFrame {
10     public static void main(String[] args) {
11         //Instância da classe TelaLogin
12         TelaLogin t = new TelaLogin();
13
14         //Método de chamada do método construtor e abertura de tela
15         t.abreTela();
16     }
17 }

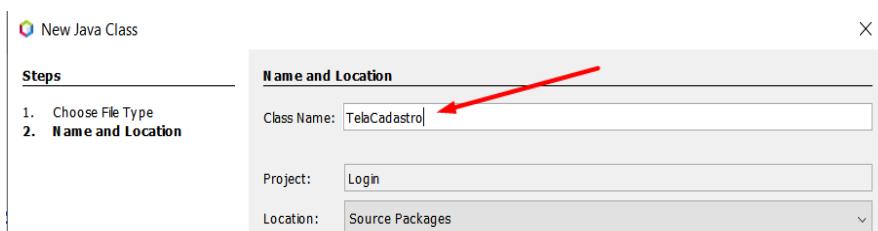
```

Executando a classe *Main* *TestaTela*, veremos:



22.3.5. Classe TelaCadastro

Nesse momento iremos criar a tela de cadastro, para isso vamos construir uma classe TelaCadastro, assim:



Abaixo codificamos a classe TelaCadastro.java:

```

1 package Login;
2
3 /**
4 * @author Professor Marcos Costa
5 */
6 /*
7 Importações de todas as bibliotecas que estou utilizando na classe
8 */
9
10 import java.awt.Font;
11 import java.awt.SystemColor;
12 import javax.swing.JButton;
13 import javax.swing.JFrame;
14 import javax.swing.JPanel;
15 import javax.swing.JLabel;
16 import javax.swing.JPasswordField;
17 import javax.swing.JTextField;
18
19 public class TelaCadastro extends JFrame{
20     //Declaração dos atributos de tela
21     private final JPanel tela;
22     private final JTextField txtNome;
23     private final JTextField txtUsuario;
24     private final JPasswordField passSenha;
25     private final JPasswordField passConfSenha;
26
27     //Método construtor da classe
28     public TelaCadastro() {
29         setLocationRelativeTo(null);
30         setResizable(false);
31         setTitle("Cadastro");
32         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
33         setBounds(500, 200, 426, 230);
34
35         tela = new JPanel();
36         tela.setBackground(SystemColor.gray);

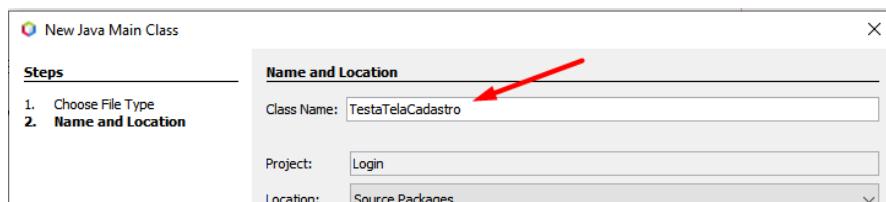
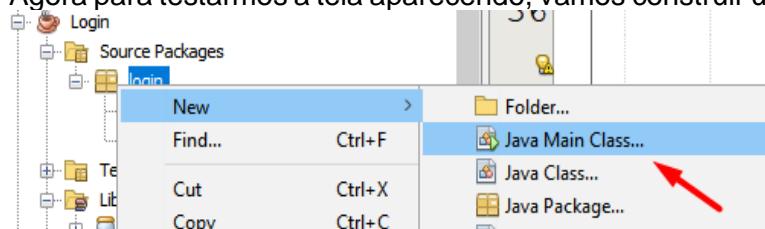
```

```

37 |         setContentPane(tela);
38 |         tela.setLayout(null);
39 |
40 |         //Adicionando elementos na tela:
41 |         JLabel lblIdentificacao = new JLabel("Informar campos para cadastro");
42 |         lblIdentificacao.setBounds(60, 0, 500, 39);
43 |         lblIdentificacao.setFont(new Font("Arial", 3, 19));
44 |         tela.add(lblIdentificacao);    //adiciono o meu label ao meu Panel
45 |
46 |         JLabel lblNome = new JLabel("Nome");
47 |         lblNome.setBounds(24, 50, 70, 15);
48 |         tela.add(lblNome);
49 |
50 |         txtNome = new JTextField();
51 |         txtNome.setBounds(120, 50, 219, 19);
52 |         tela.add(txtNome);
53 |         txtNome.setColumns(10);
54 |
55 |         JLabel lblUsuario = new JLabel("Usuario");
56 |         lblUsuario.setBounds(24, 75, 70, 15);
57 |         tela.add(lblUsuario);
58 |
59 |         txtUsuario = new JTextField();
60 |         txtUsuario.setBounds(120, 75, 219, 19);
61 |         tela.add(txtUsuario);
62 |         txtUsuario.setColumns(10);
63 |
64 |         JLabel lblSenha = new JLabel("Senha");
65 |         lblSenha.setBounds(24, 100, 70, 15);
66 |         tela.add(lblSenha);
67 |
68 |         passSenha = new JPasswordField();
69 |         passSenha.setBounds(120, 100, 219, 19);
70 |         tela.add(passSenha);
71 |
72 |         JLabel lblconfSenha = new JLabel("Confirmar Senha");
73 |         lblconfSenha.setBounds(24, 125, 100, 15);
74 |         tela.add(lblconfSenha);
75 |
76 |         passConfSenha = new JPasswordField();
77 |         passConfSenha.setBounds(120, 125, 219, 19);
78 |         tela.add(passConfSenha);
79 |
80 |         JButton btnCadastrar = new JButton("Cadastrar");
81 |         btnCadastrar.setBounds(200, 156, 117, 25);
82 |         tela.add(btnCadastrar);
83 |
84 |         JButton btnCancelar = new JButton("Cancelar");
85 |         btnCancelar.setBounds(50, 156, 117, 25);
86 |         tela.add(btnCancelar);
87 |
88 }
89
90 [- public void abreTela(){
91 |         TelaCadastro panelCadastro = new TelaCadastro();
92 |         panelCadastro.setVisible(true);
93 |     }
94 }
```

22.3.6. Classe TestaTelaCadastro

Agora para testarmos a tela aparecendo, vamos construir uma classe *Main*, assim:

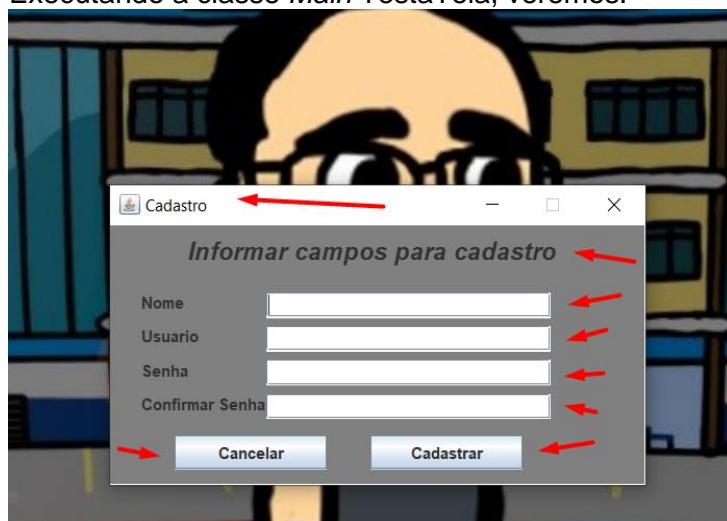


Abaixo codificamos a classe *TestaTelaCadastro.java*:

```

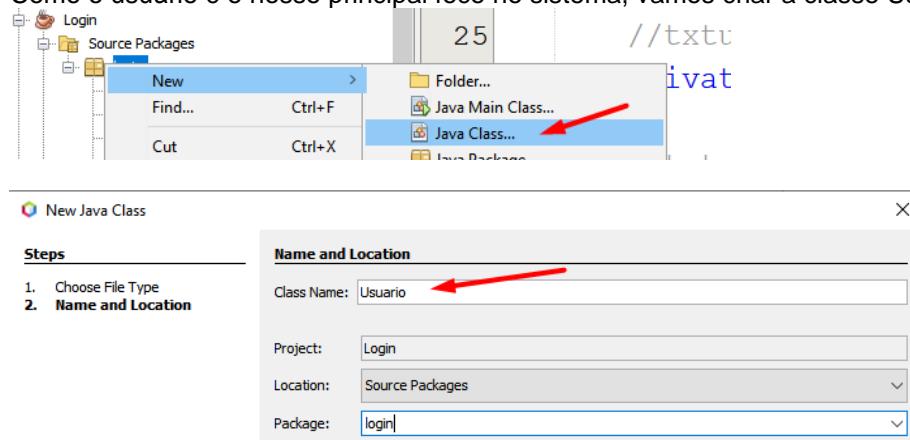
1 package Login;
2
3 /**
4 * @author Professor Marcos Costa
5 */
6 public class TestaTelaCadastro {
7
8     public static void main(String[] args) {
9         //Instância da classe TelaCadastro
10        TelaCadastro t = new TelaCadastro();
11
12        //Método de chamada do método construtor e abertura de tela
13        t.abreTela();
14    }
15 }
16
17 }
```

Executando a classe *Main* *TestaTela*, veremos:



22.3.7. Classe de Usuario

Como o usuário é o nosso principal foco no sistema, vamos criar a classe Usuario, assim:



E iremos codificar assim:

```

1 package Login;
2
3 import java.sql.SQLException; //Tratar as exceções no banco de dados
4
5 /**
6 * @author Professor Marcos Costa
7 */
8
9 public class Usuario {
10     //Criação dos atributos privados da classe
11     private String usuario;
12     private String nome;
13     private String senha;
14
15     //Atributo que armazenará o retorno do banco de dados
16     private boolean resultUsuario;
17
18     //Criação dos getters e setters
19     public String getUsuario() {
20         return usuario;
21     }
22
23     public void setUsuario(String usuario) {
24         this.usuario = usuario;
25     }
26
27     public String getNome() {
28         return nome;
29     }
30
31     public void setNome(String nome) {
32         this.nome = nome;
33     }
34
35

```

```

36     public String getSenha() {
37         return senha;
38     }
39
40     public void setSenha(String senha) {
41         this.senha = senha;
42     }
43
44     //Método de verificação da autenticidade do usuário
45     public boolean verificaUsuario(String usuario, String senha){
46         //Fazer a instância da conexão com o banco de dados
47         Conexao banco = new Conexao();
48
49         try{
50             //Abro a conexão com o banco de dados
51             banco.abrirConexao();
52
53             //Criando parâmetro de retorno
54             banco.stmt= banco.con.createStatement();
55
56             //Executando a consulta no banco de dados
57             banco.resultset =
58                 banco.stmt.executeQuery("SELECT * FROM usuario "
59                                 + "WHERE usuario = '" + usuario + "'"
60                                 + " AND senha = md5('" + senha + "')");
61
62             //Verificando se existe retorno de dados no banco
63             if (banco.resultset.next()){
64                 //Caso tenha
65                 resultUsuario = true;
66             }else{
67                 //Caso não tenha
68                 resultUsuario = false;
69             }
70
71             banco.fecharConexao(); // fecha nossa conexão com o banco de dados
72
73         }catch (SQLException ec) {
74             System.out.println("Erro ao consultar usuário " + ec.getMessage());
75         }
76
77         return resultUsuario;
78     }
79 }
```

22.3.8. Colocando validações na tela de Login no botão Entrar

Para melhorarmos mais nosso projeto iremos programar o botão entrar, mas antes precisamos pensar em validar os campos Usuário e Senha, ou seja, só iremos validar a entrada se os campos estiverem preenchidos.

Iremos implementar mais algumas linhas de código na classe **TelaLogin**, iremos acrescentar a importação da classe **JOptionPane**, mas o que ela faz?

JOptionPane é uma classe que possibilita a criação de uma caixa de diálogo padrão que ou solicita um valor para o usuário ou retorna uma informação. Abaixo encontra-se alguns métodos e parâmetros mais utilizados quando se opta pelo **JOptionPane**.

Métodos

Método	Descrição
showConfirmDialog	Solicita uma confirmação como(YES, NO, CANCEL)
showInputDialog	Solicita algum valor

showMessageDialog	Informa ao usuário sobre algo
showOptionDialog	Unificação dos três acima

Parâmetros

Parâmetros	Descrição
parentComponent	Define a caixa de diálogo onde irá aparecer todo o conteúdo. Há duas maneiras de definir a caixa de diálogo a primeira você mesmo cria utilizando os conceitos da classe JFrame. A segunda, você define esse parâmetro como null e o Java irá gerar uma caixa de diálogo padrão.
message	É a mensagem que o usuário deve ler. Esta mensagem pode ser uma simples String ou um conjunto de objetos.
messageType	Define o estilo da mensagem. Ele pode expor a caixa de diálogo de formas diferentes, dependendo deste valor, pode fornecer um ícone padrão. Exemplos: ERROR_MESSAGE INFORMATION_MESSAGE WARNING_MESSAGE QUESTION_MESSAGE PLAIN_MESSAGE
optionType	Define o conjunto de botões que irá aparecer na parte inferior da caixa de diálogo. Exemplos: DEFAULT_OPTION YES_NO_OPTION YES_NO_CANCEL_OPTION OK_CANCEL_OPTION

Exemplos:

1. Mostra um diálogo de erro que exibe a mensagem “alerta”:

```

1
2     JOptionPane.showMessageDialog(null, "alerta",
3             "alerta", JOptionPane.ERROR_MESSAGE);
4
5

```



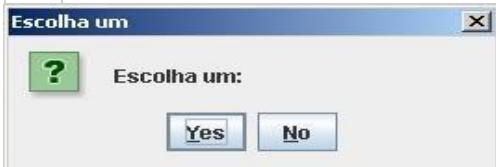
2. Mostra um painel de informação com as opções Sim/Não e exibe a mensagem: ‘Escolha um:’

```

1
2     JOptionPane.showConfirmDialog(null,"Escolha um:",
3             "Escolha um",JOptionPane.YES_NO_OPTION);

```

4



3. Mostrar uma janela de aviso com as opções OK, CANCELAR, o texto 'Aviso' no título e a mensagem 'Clique em OK para continuar':

1
2
3
4
5
6

```
Object[] options = { "OK", "CANCELAR" };
JOptionPane.showOptionDialog(null, "Clique OK para continuar", "Aviso",
    JOptionPane.DEFAULT_OPTION, JOptionPane.WARNING_MESSAGE,
    null, options, options[0]);
```



Note que se eu adicionar mais um item no vetor options, automaticamente adicionar mais um botão de opção com o nome que eu colocar.

1
2
3
4
5
6

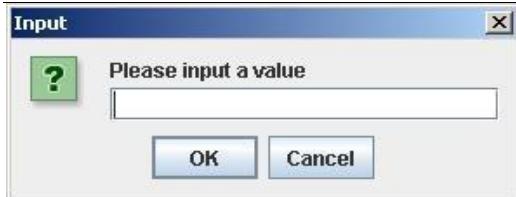
```
Object[] options = { "OK", "CANCELAR", "VOLTAR" };
JOptionPane.showOptionDialog(null, "Clique OK para continuar", "Aviso",
    JOptionPane.DEFAULT_OPTION, JOptionPane.WARNING_MESSAGE,
    null, options, options[0]);
```



4. Mostra uma caixa de diálogo solicitando que o usuário digite uma string:

1
2
3
4

```
String inputValue = JOptionPane.showInputDialog("Please input a value");
```

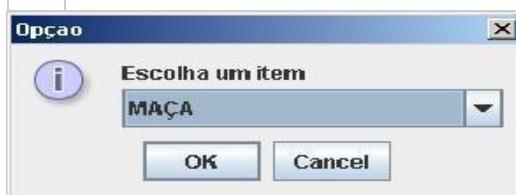


5. Mostra uma caixa de diálogo solicitando que o usuário selecione uma item:

```

1 Object[] itens = { "MAÇA", "PERA", "BANANA" };
2     Object selectedValue = JOptionPane.showInputDialog(null,
3             "Escolha um item", "Opção",
4             JOptionPane.INFORMATION_MESSAGE, null,
5             itens, itens [0]); //
6
7
8

```



No código ficará assim, teremos que importar mais duas bibliotecas a **ActionEvent** para trabalharmos com eventos, em nosso caso, será o click do botão, e o **JOptionPane**, que tratamos anteriormente:

TelaLogin.java

```

Source History | 
1 package Login;
2 /**
3 * 
4 * @author Professor Marcos Costa
5 */
6 /*
7 Importações de todas as bibliotecas que estou utilizando na classe
8 */
9 import java.awt.Font; //Trabalhar com fontes
10 import java.awt.SystemColor; //Trabalhar com cores
11 import java.awt.event.ActionEvent; //Trabalhar com evento
12 import javax.swing.JButton; //Trabalhar com botões
13 import javax.swing.JFrame; //Trabalhar com frames
14 import javax.swing.JLabel; //Trabalhar com labels
15 import javax.swing.JOptionPane; //Trabalhar com mensagens
16 import javax.swing.JPanel; //Trabalhar com painéis
17 import javax.swing.JPasswordField; //Trabalhar com campos de senha
18 import javax.swing.JTextField; //Trabalhar com campos de texto
19

```

Acrescentem também um atributo para validar o retorno do usuário cadastrado no banco de dados.

```

31     //validar se o usuario é correto
32     private boolean usuarioValido;

```

Após isso iremos implementar a ação no botão Entrar, **btnEntrar**, conforme abaixo, se atentem a partir da linha 104, vejam:



```

99     JButton btnCadastrar = new JButton("Cadastrar");
100    btnCadastrar.setBounds(50, 136, 117, 25);
101    panelTela.add(btnCadastrar);
102
103    //Ação no botão de entrar no sistema
104    btnEntrar.addActionListener((ActionEvent e) -> {
105        //Instancio a classe usuario
106        Usuario usu = new Usuario();
107
108        //Utilizo o setter de usuario e senha
109        usu.setUsuario(txtUsuario.getText());
110        usu.setSenha(pswSenha.getText());
111
112        if ("".equals(txtUsuario.getText())){
113            //Vamos dar uma mensagem na tela
114            JOptionPane.showMessageDialog(null,
115                "Campo usuário precisa ser informado!",
116                "Atenção",
117                JOptionPane.ERROR_MESSAGE);
118            //Voltar o cursor para o campo txtUsuario
119            txtUsuario.grabFocus();
120        }else if("".equals(pswSenha.getText())) {
121
122            //Vamos dar uma mensagem na tela
123            JOptionPane.showMessageDialog(null,
124                "Campo senha precisa ser informado!",
125                "Atenção",
126                JOptionPane.ERROR_MESSAGE);
127            //Voltar o cursor para o campo txtUsuario
128            pswSenha.grabFocus();
129        }else{
130            //Verifico se o usuário consta no banco de dados
131            usuarioValido = usu.verificaUsuario(usu.getUsuario(),
132                usu.getSenha());
133
134            if (usuarioValido == true){
135                //Usuario e senha bateram no banco estão corretos
136                JOptionPane.showMessageDialog(null,
137                    "Usuário válido em nossa base de dados",
138                    "Atenção",
139                    JOptionPane.INFORMATION_MESSAGE);
140
141            }else{
142                //Usuario e senha bateram no banco estão incorretos
143                JOptionPane.showMessageDialog(null,
144                    "Usuário inválido ou inexistente",
145                    "Atenção",
146                    JOptionPane.ERROR_MESSAGE);
147
148                //Método para limpar os texto
149                limpaText();
150
151                //Mando o foco para o campo usuário
152                txtUsuario.grabFocus();
153            }
154        }
    
```

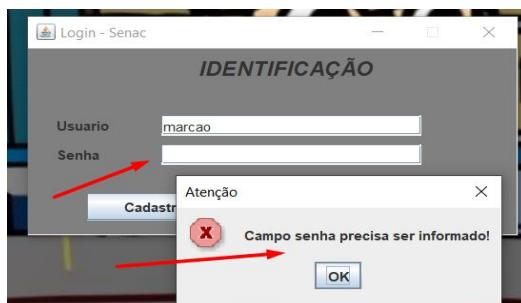
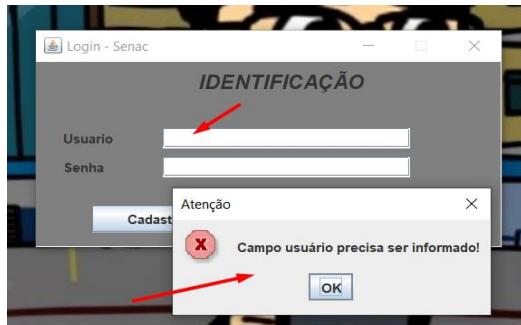
```

154     ...
155     ...
156 }
157
158     public void abreTela() {
159         TelaLogin tela = new TelaLogin();
160         tela.setVisible(true);
161     }
162
163     public void limpaText() {
164         txtUsuario.setText("");
165         pswSenha.setText("");
166     }
167 ...

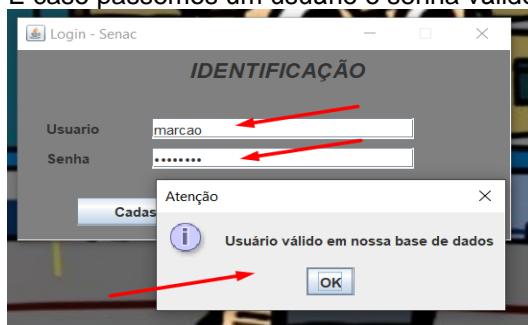
```

Percebam que os dois retângulos vermelhos representam respectivamente a chamada do método **limpaText**, que servirá para limpar os campos usuário e senha da tela.

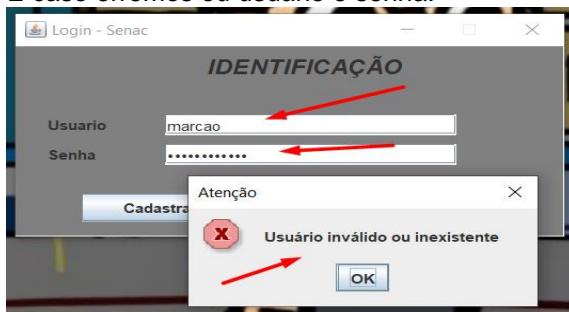
Feito isso vamos testar as funcionalidades de não passar usuário e senha, vejamos:



E caso passemos um usuário e senha válidos:



E caso erremos ou usuário e senha:



22.3.9. Cadastrar o usuário

Agora iremos codificar o cadastro do usuário, ou seja, fazer o **insert** na base de dados, iremos programar o botão cadastrar, mas antes, precisamos pensar na validação dos campos **nome**, **usuário**, **senha** e **confirmar senha**, ou seja, só efetivaremos a entrada dos dados se os mesmos estiverem preenchidos, e se os campos **senha** e **confirmar senha** estiverem com o mesmo conteúdo.

Vamos implementar mais algumas linhas de código na classe TelaCadastro, com os **imports** das bibliotecas **HeadlessException**, **JOptionPane** e **ActionEvent**:

```

8  Importações de todas as bibliotecas que estou utilizando na classe
9  */
10 import java.awt.Font;
11 import java.awt.HeadlessException;
12 import java.awt.SystemColor;
13 import java.awt.event.ActionEvent;
14 import javax.swing.JButton;
15 import javax.swing.JFrame;
16 import javax.swing.JPanel;
17 import javax.swing.JLabel;
18 import javax.swing.JOptionPane;
19 import javax.swing.JPasswordField;
20 import javax.swing.JTextField;
--
```

Iremos acrescentar os atributos **usuarioValido** e **cadastroValido**, ambos como booleano para verificarmos se o usuário já existe na base de dados para não duplicarmos o mesmo, e para verificarmos a efetivação do cadastro, e também os atributos **mensagemJOptionPane** e **mensagemTipo** para armazenarmos as mensagens e tipo de cada verificação que será realizada, no caso desta última, para uma melhor codificação, ao invés de colocarmos os **JOptionPane.ERROR_MESSAGE** ou **JOptionPane.INFORMATION_MESSAGE**, vamos colocar respectivamente 0 (**ERROR**) e 1 (**INFORMATION**).

```

29 //Validações de usuário e cadastro corretos
30 private boolean usuarioValido;
31 private boolean cadastroValido;
32
33 //String de mensagem
34 private String mensagemJOptionPane;
35 private int mensagemTipo = 0;
```

Vamos antes de mais nada, fazer a implementação do botão Cancelar, que deverá retornar para a tela de Login.

```

167 btnCancelar.addActionListener((ActionEvent e) -> {
168     TelaLogin tLogin = new TelaLogin();
169     tLogin.abreTela();
170     dispose();
171});
```

Agora faremos a implementação do botão cadastrar, onde faremos as validações de preenchimento dos campos e verificando se as senhas estão iguais.

```

100      //Ação do botão cadastrar usuário na base de dados
101     btnCadastrar.addActionListener(ActionEvent e) -> {
102         try{
103             //Instancio o objeto Usuario
104             Usuario usu = new Usuario();
105
106             //Realizando os setters dos dados de tela
107             usu.setNome(txtNome.getText());
108             usu.setUsuario(txtUsuario.getText());
109             usu.setSenha(passSenha.getText());
110
111             //Validações de preenchimento dos dados
112             if("".equals(usu.getNome())){
113                 mensagemJOptionPane = "Campo nome do usuário precisa ser informado!";
114                 mensagemTipo = 0;
115             }else if("".equals(usu.getUsuario())){
116                 mensagemJOptionPane = "Campo usuário precisa ser informado!";
117             }else if("".equals(usu.getSenha())){
118                 mensagemJOptionPane = "Campo senha precisa ser informado!";
119                 mensagemTipo = 0;
120             }else if(!usu.getSenha().equals(passConfSenha.getText())){
121                 mensagemJOptionPane = "Campos senha e confirmação de senha não coincidem!";
122                 mensagemTipo = 0;
123             }else{
124                 //Verifico se somente o usuário consta no banco,
125                 //neste caso, faremos uma sobrecarga de método
126                 usuarioValido = usu.verificaUsuario(usu.getUsuario());
127
128             if (usuarioValido == true){
129                 //Caso exista, não pode ser colocado na base
130                 mensagemJOptionPane = "Usuário já existente na base da dados";
131                 mensagemTipo = 0;
132             }else{
133                 cadastroValido = usu.cadastraUsuario(usu.getNome(),
134                                         usu.getUsuario(),
135                                         usu.getSenha());
136
137                 if (cadastroValido == true){
138                     //Usuário cadastrado na base de dados
139                     mensagemJOptionPane = "Usuário cadastrado corretamente!";
140                     mensagemTipo = 1;
141
142                 }else{
143                     //Algum erro aconteceu
144                     mensagemJOptionPane = "Problemas ao inserir o usuário!";
145                     mensagemTipo = 0;
146                 }
147             }
148         }
149
150         //Mostrar a mensagem referida
151         JOptionPane.showMessageDialog(null,
152             mensagemJOptionPane, "Atenção", mensagemTipo );
153         if (mensagemTipo == 1){
154             //Voltamos para a tela de login
155             TelaLogin tLogin = new TelaLogin();
156             tLogin.abreTela();
157
158             //Fecho a tela de cadastro
159             dispose();
160         }
161     }catch (HeadlessException ec) {
162         System.out.println("Erro no cadastro do usuário "
163                         + ec.getMessage());
164     }
165 };
166

```

Vamos nos atentar que na linha 126 fazemos a chamada do método **verificaUsuario()**, mas observe que só passamos um parâmetro, o usuário, diferente do

que fizemos no Login, onde utilizamos esse mesmo método, mas passando o usuário e a senha, nesse esse caso, estamos fazendo uma **sobrecarga de método**, onde agora só nos interessa verificar o usuário, para que não haja duplicidade em nosso banco de dados, para isso precisaremos na classe **Usuário** acrescentar esse método, portanto, agora teremos dois métodos **verificaUsuario()**, só que as assinaturas são diferentes, veja:

```
51 //Método de verificação da autenticidade do usuário
52 public boolean verificaUsuario(String usuario, String senha){
53     //Fazer a instância da conexão com o banco de dados
54     Conexao banco = new Conexao();
55
56     try{
57         //Abro a conexão com o banco de dados
58         banco.abrirConexao();                                Este método já existe em
59
60         //Criando parâmetro de retorno nossa classe, NÃO MEXER.
61         banco.stmt= banco.con.createStatement();
62
63         //Executando a consulta no banco de dados
64         banco.resultset =
65             banco.stmt.executeQuery("SELECT * FROM usuario "
66                         + "WHERE usuario = '" + usuario + "'"
67                         + " AND senha = md5('" + senha + "')");
68
69         //Verificando se existe retorno de dados no banco
70         if (banco.resultset.next()){
71             //Caso tenha
72             resultUsuario = true;
73
74             //Setters em Nome e Usuário
75             setNome(banco.resultset.getString(1));
76             setUsuario(banco.resultset.getString(2));
77
78             //Nos atributos estáticos, realizo as atribuições
79             nomeUsuario = this.getNome();
80             usuarioSistema = this.getUsuario();
81         }else{
82             //Caso não tenha
83             resultUsuario = false;
84         }
85
86         banco.fecharConexao(); // fecha nossa conexão com o banco de dados
87
88     }catch (SQLException ec) {
89         System.out.println("Erro ao consultar usuário " + ec.getMessage());
90     }
91
92     return resultUsuario;
93 }
94
95 //Método de verificação do usuário, estamos aqui fazendo uma
96 //sobrecarga de método
97 public boolean verificaUsuario(String usuario){                                Este método vamos acrescentar
98     //Fazer a instância da conexão com o banco de dados
99     Conexao banco = new Conexao();                                            agora.
100    try{                                                              
101        //Abro a conexão com o banco de dados
102        banco.abrirConexao();
103
104        //Criando parâmetro de retorno
105        banco.stmt= banco.con.createStatement();
106
107        //Executando a consulta no banco de dados
108        banco.resultset =
```

```

110         banco.createStatement("SELECT * FROM usuario "
111                                         + "WHERE usuario = '" + usuario + "'");
112
113         //Verificando se existe retorno de dados no banco
114         if (banco.getResultSet().next()){
115             //Caso tenha
116             resultUsuario = true;
117         }else{
118             //Caso não tenha
119             resultUsuario = false;
120         }
121
122         banco.fecharConexao(); // fecha nossa conexão com o banco de dados
123
124     }catch (SQLException ec) {
125         System.out.println("Erro ao consultar usuário " + ec.getMessage());
126     }
127
128     return resultUsuario;
129 }
```

A diferença além da assinatura, é na *query*, pois não validamos a senha, somente o usuário.

Não havendo duplicidade de usuário na linha 124, conforme programamos há pouco, fazemos a chamada do método **cadastraUsuario()** que está na classe de **Usuário**, e implementamos uma atributo para retornar o resultado do cadastro o **resultCadastro**, assim:

```

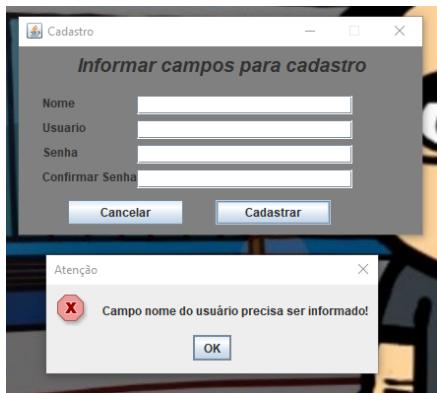
16     //Atributo que armazenará o retorno do banco de dados
17     private boolean resultUsuario;
18     private boolean resultCadastro; ←
```

Agora a implementação do método **cadastraUsuario()**:

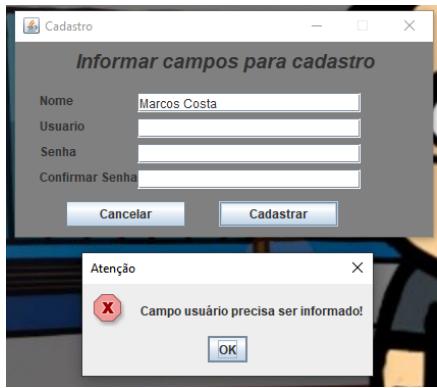
```

131     //Método para cadastro de usuário em nosso sistema
132     public boolean cadastraUsuario(String nome, String usuario, String senha){
133         //Fazer a instância da conexão com o banco de dados
134         Conexao banco = new Conexao();
135
136         try{
137             //Abertura da conexão com o banco de dados
138             banco.abrirConexao();
139
140             //Criando parâmetro de retorno
141             banco.setStatement(banco.con.createStatement());
142
143             //Executando a inserção dos dados
144             banco.setStatement("INSERT INTO usuario (nome, usuario, senha)"
145                             + " VALUES ('" + nome + "','" + usuario + "', md5('"
146                             + senha + "')");
147
148             resultCadastro = true;
149
150         }catch(SQLException ec){
151             System.out.println("Erro ao inserir o usuário " + ec.getMessage());
152             resultCadastro = false;
153         }
154
155         return resultCadastro;
156     }
```

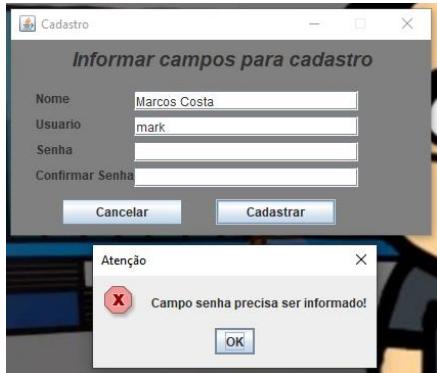
Assim feito, temos a inserção e validações de usuário implementadas em nosso projeto, testando ficaria assim:



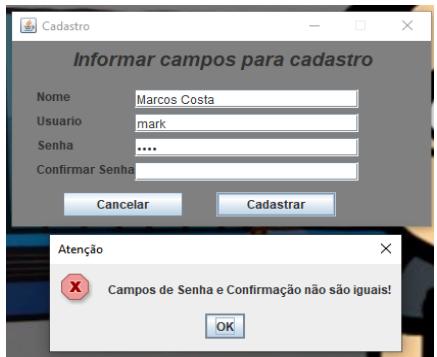
Nome do usuário precisa ser informado



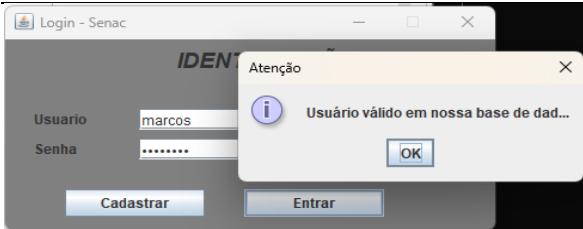
Usuário precisa ser informado



Senha precisa ser informada



Campos senha e confirmar senha precisam ser iguais



Usuário e senha corretos.

Após a inserção o sistema será direcionado para a tela de login.

22.3.10. Tela Inicio

Agora iremos codificar uma tela para início, ou seja, após o login, será mostrada uma tela home, nesta tela teremos uma opção para alteração e uma exclusão de nosso usuário, percebam que ao codificar a chamada do método de exclusão, apresentará um erro, mas não se preocupem, isso acontece porque o método ainda não foi codificado, e nas linhas 34, 78 e 90, iremos trabalhar com atributos estáticos (`nomeUsuario` e `usuarioSistema`), que abordaremos logo que fizermos estas alterações, este erro acertaremos, vejam:

```

1 package Login;
2
3
4 import java.awt.HeadlessException; ←
5 import java.awt.SystemColor;
6 import java.awt.event.ActionEvent;
7 import javax.swing.JButton;
8 import javax.swing.JFrame;
9 import javax.swing.JLabel;
10 import javax.swing.JOptionPane;
11 import javax.swing.JPanel;
12
13 public class TelaInicio extends JFrame{
14     private final JPanel tela;
15
16     //Atributo que faremos para validar a Exclusão
17     private boolean exclusaoValida;
18
19     public TelaInicio() {
20
21         setLocationRelativeTo(null);
22         setResizable(false);
23         setTitle("Menu - Senac");
24
25         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
26         setBounds(500, 200, 426, 212);
27
28         tela = new JPanel();
29         tela.setBackground(SystemColor.gray);
30         setContentPane(tela);
31         tela.setLayout(null);
32
33         //Concateno o Label com o atributo global nome da classe Usuario
34         JLabel lblUsuario = new JLabel("Bem vindo " + Usuario.nomeUsuario); →
35         lblUsuario.setBounds(24, 65, 200, 15);
36         tela.add(lblUsuario);
37
38         JButton btnExcluir = new JButton("Excluir");
39         btnExcluir.setBounds(2, 130, 117, 25);
40         tela.add(btnExcluir);
41
42         JButton btnAlterar = new JButton("Alterar Dados");
43         btnAlterar.setBounds(150, 130, 117, 25);
44         tela.add(btnAlterar);
45
46         JButton btnVoltar = new JButton("Voltar");
47         btnVoltar.setBounds(300, 130, 117, 25);
48         tela.add(btnVoltar);

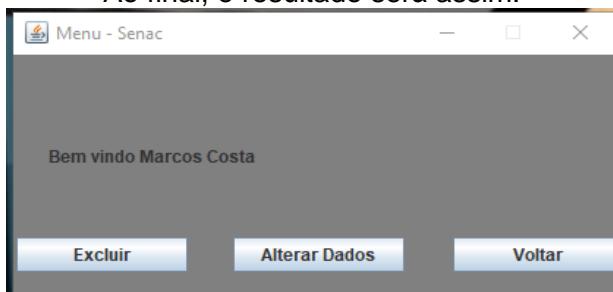
```



```

111     }
112
113     }catch (HeadlessException ec) {
114         System.out.println("Erro ao excluir usuário "
115                         + ec.getMessage());
116         "Erro ao excluir usuário "
117     });
118 }
119
120 public void abreTela(){
121     TelaInicio telaInicio = new TelaInicio();
122     telaInicio.setVisible(true);
123 }
124 }
```

Ao final, o resultado será assim:



Tela inicial

Percebam que na tela temos boas-vindas, carregando o nome do usuário, com botões de exclusão, alteração de dados e voltar ao login. Iremos ter que realizar algumas alterações em nosso sistema para que isso aconteça, pois, como vimos acima, tem alguns erros que aparecem, mas o motivo é porque não implementamos algumas alterações que farão necessárias agora, vejam:

22.3.11. Mudança na classe TelaLogin

Para contemplarmos após o acerto do usuário e da senha na tela de login a ida para o menu ou tela home, precisamos acrescentar as seguintes linhas no método **actionPerformed** do Botão Entrar:

```

151
152             if (usuarioValido == true){
153                 //Usuario e senha bateram no banco estão corretos
154                 JOptionPane.showMessageDialog(null,
155                     "Usuário válido em nossa base de dados",
156                     "Atenção",
157                     JOptionPane.INFORMATION_MESSAGE);
158
159             TelaInicio telaInicio = new TelaInicio();
160             telaInicio.AbreTela();
161
162             dispose();
```

Também precisamos criar o método no botão Cadastrar para abrimos esta tela.

```

162             //Ação no botão para cadastrar Usuário
163             btnCadastrar.addActionListener((ActionEvent e) -> {
164                 //Instancio a classe TelaCadastro
165                 TelaCadastro tCadastro = new TelaCadastro();
166                 tCadastro.abreTela();
167                 dispose();
168             });
169 }
```

22.3.12. Mudança na classe Usuario

Para nossas implementações iremos acrescentar algumas implementações na classe de usuário, para isso vamos utilizar o *Static*, que serve para referenciar todos aqueles atributos/métodos de classe, ou seja, que podem ser acessados diretamente da definição da classe, sem precisar instanciar nenhum objeto, com isso, estes atributos serão vistos em outras classes que precisaremos utilizar.

```
22     //Atributos Estáticos do sistema
23     static String nomeUsuario;
24     static String usuarioSistema;
```

Iremos acrescentar os atributos de retorno para a Alteração e Exclusão.

```
19     private boolean resultAlteracao;
20     private boolean resultExclusao;
```

No método **verificaUsuario()** que valida usuário e senha precisamos atribuir com *gets* o nome e o usuário aos atributos estáticos.

```
65             banco.stmt.executeQuery("SELECT * FROM usuario "
66                                         + "WHERE usuario = '" + usuario + "'"
67                                         + " AND senha = md5('" + senha + "')");
68
69         //Verificando se existe retorno de dados no banco
70         if (banco.resultset.next()){
71             //Caso tenha
72             resultUsuario = true;
73
74             //Setters em Nome e Usuario
75             setUsuario(banco.resultset.getString(1));
76             setNome(banco.resultset.getString(2));
77
78             //Nos atributos estáticos, realizo as atribuições
79             nomeUsuario = getNome();
80             usuarioSistema = getUsuario();
```

Iremos acrescentar o método de alteração do usuário.

```
158     //Método para alteração dos dados em nosso sistema
159     public boolean alteraUsuario(String nome, String usuario, String senha){
160         //Fazer a instância da conexão com o banco de dados
161
162         Conexao banco = new Conexao();
163
164         try{
165             //Abro a conexão com banco de dados
166             banco.abrirConexao();
167
168             //Criando o parâmetro de retorno
169             banco.stmt = banco.con.createStatement();
170
171             //Executando a alteração no banco de dados
172             banco.stmt.execute("UPDATE usuario SET nome = '" + nome +
173                               ", senha = '" + senha + "' WHERE usuario = '" +
174                               usuario + "'");
175         }catch (SQLException ec){
176             System.out.println("Erro ao atualizar usuário " + ec.getMessage());
177             resultAlteracao = false;
178         }
179
180         banco.fecharConexao();
181
182         return resultAlteracao;
183     }
```

Iremos acrescentar o método de exclusão do usuário.

```
185     //Método para exclusão do Usuário do sistema
186     public boolean excluiUsuario(String usuario){
187         //Fazer a instância da conexão com o banco de dados
188
189         Conexao banco = new Conexao();
190
191         try{
192             //Abro a conexão com banco de dados
193             banco.abrirConexao();
194
195             //Criando o parâmetro de retorno
196             banco.stmt = banco.con.createStatement();
197
198             //Executando a alteração no banco de dados
199             banco.stmt.execute("DELETE FROM usuario WHERE usuario = ''"
200                               + usuario + "'");
201
202             //Caso exclua
203             resultExclusao = true;
204
205         }catch (SQLException ec){
206             System.out.println("Erro ao excluir usuário " + ec.getMessage());
207             resultExclusao = false;
208         }
209
210         banco.fecharConexao();
211
212         return resultExclusao;
213     }
```

22.3.13. Classe TelaAlteracao

Agora para finalizar, iremos implementar a tela de alteração do usuário, vejam:

```
1  package Login;
2
3
4  import java.awt.Font;
5  import java.awt.SystemColor;
6  import java.awt.event.ActionEvent;
7  import javax.swing.JButton;
8  import javax.swing.JFrame;
9  import javax.swing.JLabel;
10 import javax.swing.JOptionPane;
11 import javax.swing.JPanel;
12 import javax.swing.JPasswordField;
13 import javax.swing.JTextField;
14 import static Login.Usuario.usuarioSistema;
15 import java.awt.HeadlessException;
16
17 /**
18 * @author Professor Marcos Costa
19 */
20
21 public class TelaAlteracao extends JFrame{
22     //Criando meus atributos globais
23     private final JPanel tela;
24     private final JTextField txtNome;
25     private final JPasswordField passAtual;
26     private final JPasswordField passSenha;
27     private final JPasswordField confPassSenha;
```

```
28
29     private boolean atualizacaoValida;
30
31     public TelaAlteracao() {
32
33         setLocationRelativeTo(null);
34         setResizable(false);
35         setTitle("Senac - Alteração");
36         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
37         setBounds(500, 200, 426, 212);
38
39         tela = new JPanel();
40         tela.setBackground(SystemColor.gray);
41         setContentPane(tela);
42         tela.setLayout(null);
43
44         //Adicionando elementos na tela:
45         JLabel lblIdentificacao = new JLabel("Informar campos para alteração");
46         lblIdentificacao.setBounds(60, 0, 500, 39);
47         lblIdentificacao.setFont(new Font("Arial", 3, 19));
48         tela.add(lblIdentificacao);
49
50         JLabel lblNome = new JLabel("Nome");
51         lblNome.setBounds(24, 35, 100, 15);
52         tela.add(lblNome);
53
54         txtNome = new JTextField();
55         txtNome.setBounds(120, 35, 218, 20);
56         tela.add(txtNome);
57         txtNome.setColumns(10);
58
59         JLabel lblSenhaAtual = new JLabel("Senha Atual");
60         lblSenhaAtual.setBounds(24, 60, 70, 15);
61         tela.add(lblSenhaAtual);
62
63         passAtual = new JPasswordField();
64         passAtual.setBounds(120, 60, 219, 19);
65         tela.add(passAtual);
66
67         JLabel lblNovasenha = new JLabel("Nova Senha");
68         lblNovasenha.setBounds(24, 85, 70, 15);
69         tela.add(lblNovasenha);
70
71         passSenha = new JPasswordField();
72         passSenha.setBounds(120, 85, 219, 19);
73         tela.add(passSenha);
74
75         JLabel lblConfSenha = new JLabel("Confirmar Senha");
76         lblConfSenha.setBounds(24, 110, 100, 15);
77         tela.add(lblConfSenha);
78
79         confPassSenha = new JPasswordField();
80         confPassSenha.setBounds(120, 110, 219, 19);
81         tela.add(confPassSenha);
82
83         JButton btnAlterar = new JButton("Alterar");
84         btnAlterar.setBounds(200, 136, 117, 25);
85         tela.add(btnAlterar);
86
87         JButton btnCancelar = new JButton("Cancelar");
88         btnCancelar.setBounds(50, 136, 117, 25);
89         tela.add(btnCancelar);
```

```

90
91     btnCancelar.addActionListener(ActionEvent e) -> {
92         TelaInicio telaIni = new TelaInicio();
93         telaIni.setVisible(true);
94         dispose();
95     };
96
97     //Botão de alteração
98     btnAlterar.addActionListener(ActionEvent e) -> {
99         try{
100             //Instancio a classe usuario
101             Usuario usu = new Usuario();
102
103             //Validações antes de efetivar a alteração
104             //setando a senha e usuario
105             usu.setSenha(confPassSenha.getText());
106             usu.setUsuario(usuarioSistema);
107
108             //Nome vazio
109             if ("").equals(usu.getNome())){
110                 //Vamos dar uma mensagem na tela
111                 JOptionPane.showMessageDialog(null,
112                     "Campo nome do usuário precisa ser informado!",
113                     "Atenção",
114                     JOptionPane.ERROR_MESSAGE);
115                 //Voltar o cursor para o campo txtNome
116                 txtNome.grabFocus();
117                 //Senha vazia
118             }else if("".equals(usu.getSenha())){
119                 //Vamos dar uma mensagem na tela
120                 JOptionPane.showMessageDialog(null,
121                     "Campo senha precisa ser informado!",
122                     "Atenção",
123                     JOptionPane.ERROR_MESSAGE);
124                 //Voltar o cursor para o campo passSenha
125                 passSenha.grabFocus(); //*****
126             }else if(usu.verificaUsuario(usu.getUsuario(),
127                 passAtual.getText()) == false){
128                 //Vamos dar uma mensagem na tela
129                 JOptionPane.showMessageDialog(null,
130                     "Senha inválida, verifique!",
131                     "Atenção",
132                     JOptionPane.ERROR_MESSAGE);
133                 //Voltar o cursor para o campo passSenha
134                 passSenha.grabFocus();
135             }else if(!passSenha.getText().equals(confPassSenha.getText())){
136                 //Vamos dar uma mensagem na tela
137                 JOptionPane.showMessageDialog(null,
138                     "Campos de Senha e Confirmação não são iguais!",
139                     "Atenção",
140                     JOptionPane.ERROR_MESSAGE);
141                 //Voltar o cursor para o campo passSenha
142                 passSenha.grabFocus();
143
144             }else{
145
146                 //Efetivo a alteração do usuario
147                 atualizacaoValida =
148                     usu.alteraUsuario(txtNome.getText(),
149                         usu.getUsuario(),
150                         usu.getSenha());
151
152             if (atualizacaoValida == true){
153                 //Usuario cadastrado na base de dados
154                 JOptionPane.showMessageDialog(null,
155                     "Dados do usuário alterado(s) retornaremos "
156                     + "a tela de login.",
157                     "Atenção",
158                     JOptionPane.INFORMATION_MESSAGE);
159

```

```

160 //Abrimos a tela de login novamente
161 TelaLogin tLogin = new TelaLogin();
162 tLogin.abreTela();

163
164 //Fecho a tela de cadastro
165 dispose();

166
167 }else{
168     //Usuario cadastrado na base de dados
169     JOptionPane.showMessageDialog(null,
170         "Problemas ao atualizar o usuário",
171         "Atenção",
172         JOptionPane.ERROR_MESSAGE);
173 }
174
175 }catch (HeadlessException ec) {
176     System.out.println("Erro ao alterar usuário "
177         + ec.getMessage());
178 }
179 });

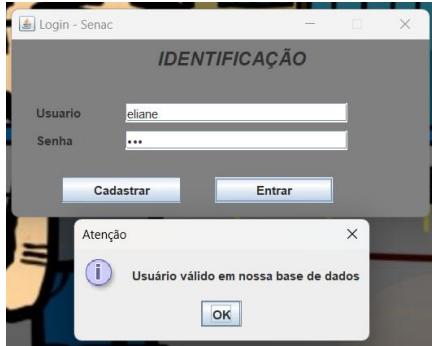
180 //Atribuir o atributo global ao objeto
181 txtNome.setText(Usuario.nomeUsuario);
182 }

183
184 public void abreTela(){
185     TelaAlteracao telaAlteracao = new TelaAlteracao();
186     telaAlteracao.setVisible(true);
187 }
188
189 }

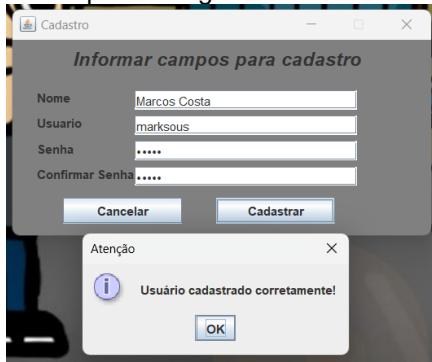
```

Com isso, finalizamos a parte de programação de nosso projeto, abaixo vamos ver os passos para o completo funcionamento do sistema, vejam:

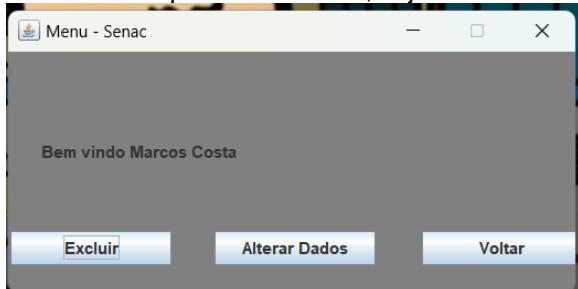
O nosso sistema deve iniciar o funcionamento pela tela de login do mesmo:



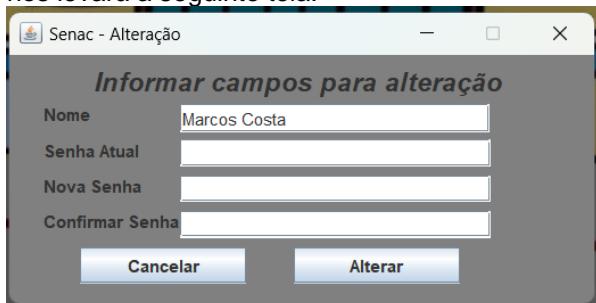
Caso tenhamos o usuário e senha cadastrados, o passo é a autenticação, mas caso não tenhamos, vamos cadastrar um novo usuário clicando no botão cadastrar da tela de login que nos levará para a seguinte tela:



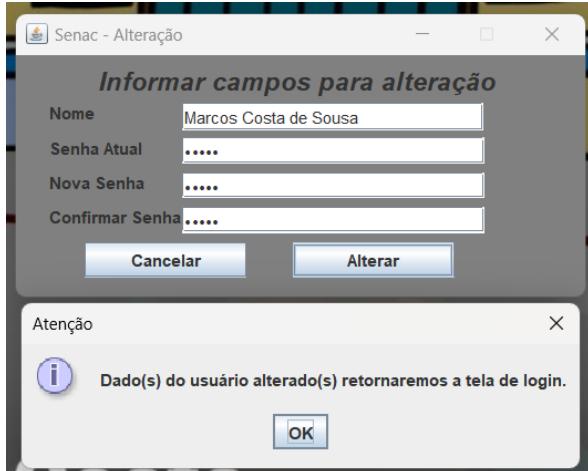
Fazendo o login de acordo com os dados que constam no banco de dados, iremos ser direcionados a uma tela inicial, uma espécie de home de nosso projeto, onde deverá ter boas vidas com o nome que cadastramos, vejam:



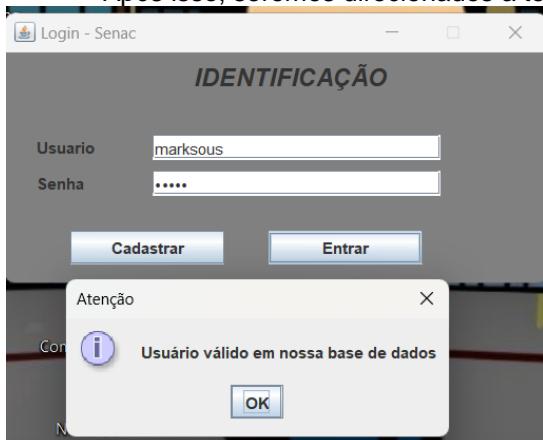
Agora vamos alterar os dados de nosso usuário, clicando no botão **Alterar Dados**, que nos levará a seguinte tela:



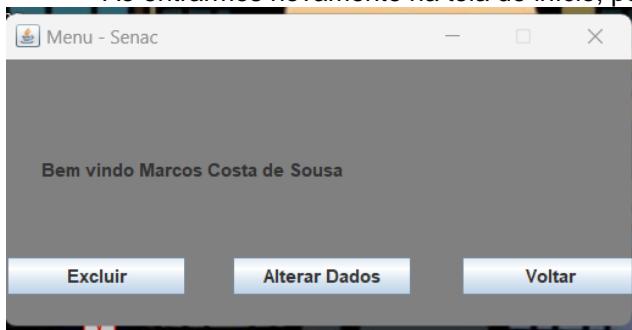
Vamos alterar os dados e clicar em Alterar para efetivar as ações:



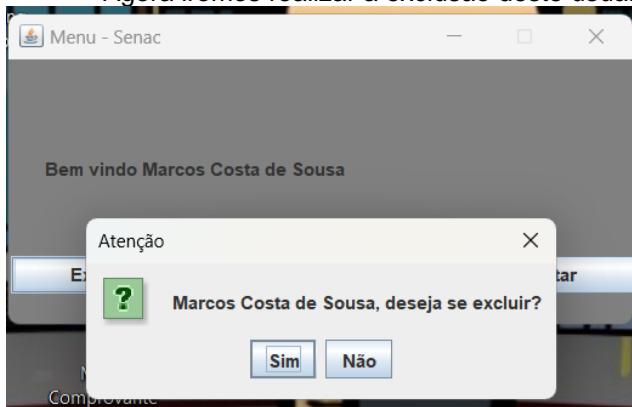
Após isso, seremos direcionados a tela de login novamente, para autenticação:



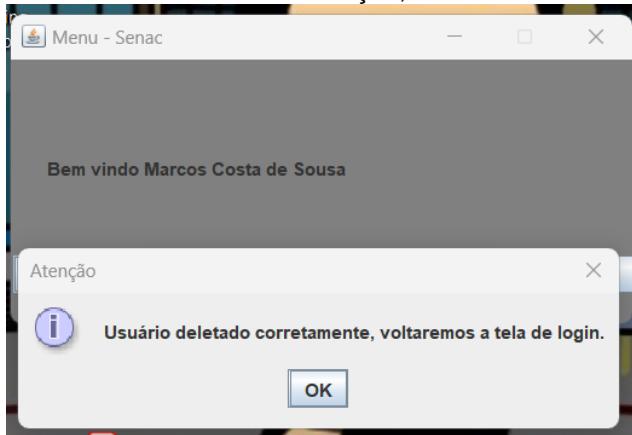
Ao entrarmos novamente na tela de início, podemos perceber que o nome foi alterado:



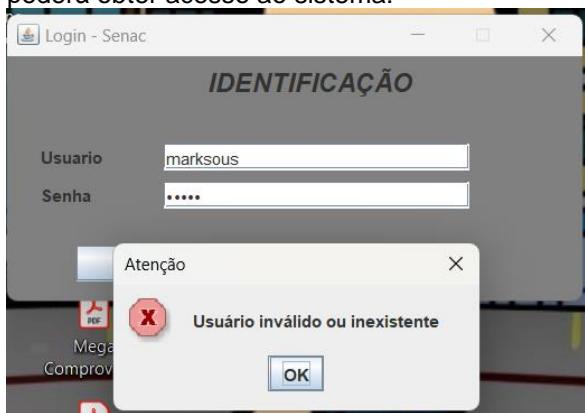
Agora iremos realizar a exclusão deste usuário:



Ao realizarmos esta ação, nos será informado o direcionamento a tela de login novamente:



Agora para finalizar, vamos tentar logar novamente com este usuário e senha, e claro, não poderá obter acesso ao sistema:



Fazendo o **select** no banco de dados, percebemos que o usuário marksous foi deletado:

```
12 •  SELECT * FROM bd_login.usuario;
13
```

Result Grid			Filter Rows:	Edit:	Export:
	usuario	nome	senha		
▶	eliane	Eliane Prado Campos de Sousa	202cb962ac59075b964b07152d234b70		
*	HULL	HULL	HULL		

Com isto, fizemos uma aplicação com um CRUD completo, agora a missão é com vocês, para a avaliação de nossa **P2**, precisamos que apresentem este sistema funcionando com todos estes passos realizados em aula, a data limite para apresentação do mesmo, será dia **31/10/2024**, de forma individual.

23. Introdução

Nessa fase do curso iremos abordar a ideia da programação no lado do servidor, o que podemos chamar de **backend**, para isso, utilizaremos a linguagem PHP, nossa preocupação não será o **frontend**, e sim entendermos o comportamento dessa linguagem dentro da programação Web, com o andamento do curso iremos refinando e melhorando os conceitos.

Mas antes vamos imaginar a internet na qual você pode apenas consumir conteúdos, como se fosse um jornal, uma revista, ou ainda, um programa na televisão. Chato, né? Mas quando se aprende as linguagens da web, como HTML e CSS, podemos montar sites que são como revistas e servem apenas para leitura, sem permitir interação com os internautas.

O segredo da famosa web 2.0 é a capacidade de interação entre as pessoas e os serviços online. Mas, para que esta interação seja possível, é necessário que os sites sejam capazes de receber informações dos internautas e também de exibir conteúdos personalizados para cada um, ou de mudar seu conteúdo automaticamente, sem que o desenvolvedor precise criar um novo HTML para isso.

Estes dois tipos de sites são chamados de estático e dinâmico, respectivamente.

Você já parou para pensar em tudo o que acontece quando você digita um endereço em seu navegador web? A história toda e mais ou menos assim:

- O navegador vai até o servidor que responde no endereço solicitado e pede a página solicitada.
- O servidor verifica se o endereço existe e se a página também existe em seu sistema de arquivos e então retorna o arquivo para o navegador.
- Após receber o arquivo HTML, o navegador começa o trabalho de renderização, para exibir a página para o usuário. É neste momento que o navegador também requisita arquivos de estilos (css), imagens e outros arquivos necessários para a exibição da página.

Quando se desenvolve páginas estáticas, este é basicamente todo o processo necessário para que o navegador exiba a página para o usuário. Chamamos de estáticas as páginas web que não mudam seu conteúdo, mesmo em uma nova requisição ao servidor.

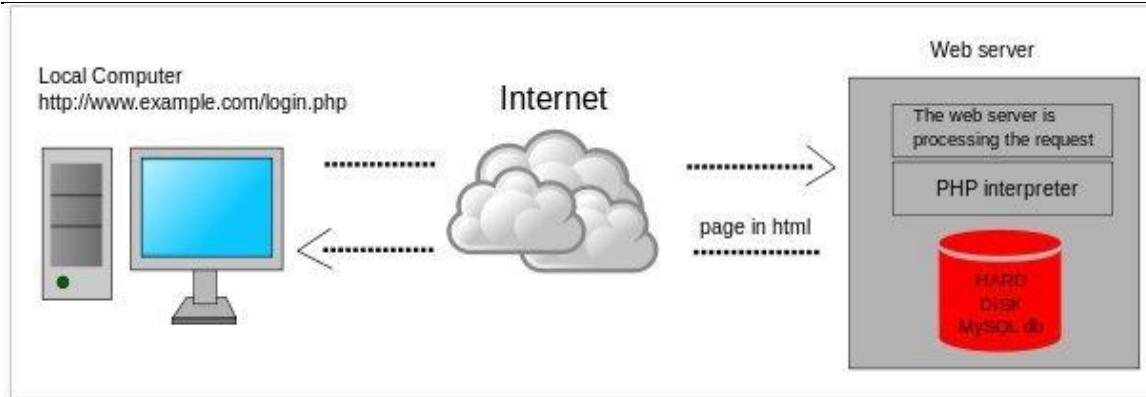
E como funciona uma página dinâmica?

O processo para páginas dinâmicas é muito parecido com o das páginas estáticas. A diferença é que a página será processada **no servidor** antes de ser enviada para o usuário. Este processamento no servidor é usado para alterar dinamicamente o conteúdo de uma página, seja ele HTML, CSS, imagens ou outros formatos.

Pense, por exemplo, em um site de um jornal. Em geral, este tipo de site contém algumas áreas destinadas às notícias de destaque, outras áreas para notícias gerais e ainda outras áreas para outros fins. Quando o navegador solicita a página para o servidor, ele irá montar o conteúdo antes de enviar para o navegador. Este conteúdo pode ser conseguido de algumas fontes, mas a mais comum é um banco de dados, onde, neste caso, as notícias ficam armazenadas para serem exibidas nas páginas quando necessário.

23.2. Scripting no lado do servidor (*server-side*)

A linguagem de servidor, ou *server-side scripting*, é a linguagem que vai rodar, fornecendo a lógica principal da aplicação.



Do lado do servidor significa toda a regra de negócio, acesso a banco de dados, a parte matemática e cálculos estarão armazenadas, enquanto que *local computer* será na máquina do usuário/cliente. Para a segurança e integridade dos dados, do lado do servidor geralmente é melhor, pois o usuário final não será capaz de ver ou manipular os dados que são enviados para o mesmo, e muito menos visualizar a regra de negócio, ou a forma de acesso ao banco de dados.

23.3. Scripting no lado do cliente (*client-side*)

As linguagens *client-side* são linguagens onde apenas o seu NAVEGADOR vai entender. Quem vai processar essa linguagem não é o servidor, mas o seu browser (Chrome, IE, Firefox, etc...). Significa "lado do cliente", ou seja, aplicações que rodam no computador do usuário sem necessidade de processamento de seu servidor (ou host) para efetuar determinada tarefa.

Basicamente, ao se falar de aplicações *client-side* na web, estamos falando de *JavaScript*, e *AJAX* (*Asynchronous Javascript And XML*).

Existem vantagens e desvantagens ao utilizar o *JavaScript* e *AJAX*.

A principal vantagem está na possibilidade de você economizar bandwidth (largura de banda), e dar ao usuário uma resposta mais rápida de sua aplicação por não haver processamento externo.

Outra vantagem ao utilizar, agora o *AJAX*, seria o apelo visual de sua aplicação e rapidez de resposta. O que o *AJAX* faz é o processamento externo (*server-side*) parecendo ser interno (*client-side*). O usuário não percebe que houve um novo carregamento de página, pois ele busca informações no servidor e mostra rapidamente em um local específico da página através do *JavaScript*.

A principal desvantagem do *JavaScript* atualmente é que o usuário pode desativá-lo em seu navegador. Se a sua aplicação basear-se exclusivamente em *JavaScript*, nesse caso, ela simplesmente não vai funcionar.

23.4. A diferença entre o lado do cliente (*client-side*) e do lado do servidor (*server-side*) Scripting

Ao escrever aplicações para a web, você pode colocar os programas, ou scripts, ou no servidor da Web ou no navegador do cliente, a melhor abordagem combina uma mistura cuidadosa dos dois. Endereços de scripts do lado do servidor de gerenciamento e segurança de dados, enquanto que o script do lado do cliente se concentra principalmente na verificação de dados e layout de página.

Um servidor web é um computador separado e software com a sua própria conexão à Internet. Quando o navegador solicita uma página, o servidor recebe o seu pedido e envia o conteúdo do navegador. Um script de programa que executa no servidor web gera uma página com base na lógica do programa e envia para o navegador do usuário. O conteúdo pode ser texto e imagens padrão, ou pode incluir scripts do lado do cliente. Seu navegador executa os scripts do lado do cliente, o que pode animar imagens da página web, solicitar os dados do servidor ou executar outras tarefas.

Para um website ter uma sessão, onde você faz *login*, fazer compras e outros pedidos, o servidor precisa para identificar o computador. Milhares de usuários podem ser registrados em, ao mesmo tempo, o servidor tem de distingui-los. *Scripting* do lado do servidor mantém o controle da identidade de um usuário através de alguns mecanismos diferentes, tais como variáveis de sessão.

Quando você fizer *login*, o script do servidor cria uma identificação de sessão exclusiva para você. O script pode armazenar informações em variáveis que duram o tempo que você ficar conectado.

Muitas páginas da web têm formulários para você preencher com seu nome, endereço e outras informações. Para certificar que os dados vão corretamente, os scripts de validação são capazes de verificar que as datas e códigos postais contêm apenas números e os estados têm certas combinações de duas letras. Este processo é mais eficaz quando o script é executado no lado do cliente. Caso contrário, o servidor tem que receber os dados, verificar-ló, e enviar-lhe uma mensagem de erro. Quando o navegador faz isso, você envia os dados de volta para o servidor somente uma vez.

Quando uma sessão de web envolve peneirar grandes quantidades de dados, um *script* do lado do servidor faz este trabalho melhor. Por exemplo, um banco pode ter um milhão de clientes. Quando você entrar, ele deve buscar o seu registro a partir deste arquivo grande. Ao invés de enviá-lo por toda a sua conexão de Internet para o seu navegador, o servidor web solicita informações de um servidor de dados perto dele. Além de aliviar a Internet do tráfego de dados desnecessários, isso também melhora a segurança.

Podemos encontrar uma maior variedade de linguagens de programação em servidores do que em navegadores. Os programadores fazem mais *scripting* do lado do cliente com a linguagem Javascript. No lado do servidor, você pode escrever em linguagens como PHP, VBScript ou ColdFusion. Enquanto alguns programadores escrever scripts do lado do cliente para executar fora do navegador, isso é arriscado, uma vez que pressupõe que o computador sabe que a linguagem.

23.5. A linguagem PHP

O PHP (um acrônimo recursivo para PHP: Hypertext Preprocessor) é uma linguagem de *script open source* de uso geral, muito utilizada, e especialmente adequada para o desenvolvimento web e que pode ser embutida dentro do HTML, é uma das linguagens mais utilizadas na web. Milhões de sites no mundo inteiro utilizam PHP. A principal diferença em relação às outras linguagens é a capacidade que o PHP tem de interagir com o mundo web, transformando totalmente os websites que possuem páginas estáticas. Imagine, por exemplo, um website que deseja exibir notícias em sua página principal, mostrando a cada dia, ou a cada hora, notícias diferentes. Seria inviável fazer isso utilizando apenas HTML. As páginas seriam estáticas, e a cada notícia nova que aparecesse no site a página deveria ser alterada manualmente, e logo após enviada ao servidor por FTP (*File Transfer Protocol*) para que as novas notícias fossem mostradas no site. Com o PHP, tudo isso poderia ser feito automaticamente. Bastaria criar um banco de dados onde ficariam armazenadas as notícias e criar uma página que mostrasse essas notícias, “puxando-as” do banco de dados. Agora imagine um site que possui cerca de cem páginas. Suponha que no lado esquerdo das páginas há um menu com links para as seções do site. Se alguma seção for incluída ou excluída, o que você faria para atualizar as cem páginas, incluindo ou excluindo esse novo link? Alteraria uma a uma, manualmente? Com certeza, você demoraria horas para alterar todas as páginas. E isso deveria ser feito cada vez que houvesse alteração, inclusão ou exclusão de uma seção no site. Para resolver esse problema utilizando PHP é muito simples. Basta construir um único menu e fazer todas as cem páginas acessarem esse arquivo e mostrá-lo em sua parte da esquerda. Quando alguma alteração for necessária, basta alterar um único arquivo, e as cem páginas serão alteradas automaticamente, já que todas acessam o mesmo menu.

Essas são apenas algumas das inúmeras vantagens das páginas que utilizam PHP. Você acabou de conhecer dois exemplos de sites em que a principal característica é o dinamismo e a praticidade. Automatização de tarefas, economia de tempo e de mão de obra são características evidentes nos dois exemplos citados. Mais adiante, veremos como implementar programas como os que foram citados aqui.

23.5.1 Características do PHP

- É uma linguagem de fácil aprendizado;
- Tem suporte a um grande número de bancos de dados como: dBase, Interbase, mSQL, mySQL, Oracle, Sybase, PostgreSQL e vários outros;
- Tem suporte a outros serviços através de protocolos como IMAP, SNMP, NNTP, POP3 e, logicamente, HTTP;
- É multiplataforma, tendo suporte aos sistemas Operacionais mais utilizados no mercado;

- Seu código é livre, não é preciso pagar por sua utilização e pode ser alterado pelo usuário na medida da necessidade de cada usuário
- Não precisa ser compilado.

23.5.2 Embutido no HTML

Outra característica do PHP é que ele é embutido no HTML. Veremos mais adiante as facilidades que isso pode nos trazer. Uma página que contém programação PHP normalmente possui extensão .php (isso depende da configuração do seu servidor web). Sempre que o servidor web receber solicitações de páginas que possuem essa extensão, ele saberá que essa página possui linhas de programação. Porém, você verá que o HTML e o PHP estão misturados, pois começamos a escrever em PHP, de repente escrevemos um trecho em HTML, depois voltamos para o PHP, e assim por diante.

23.5.3 Baseado no servidor

O *JavaScript* que vocês já viram anteriormente, consiste em scripts que também são colocados nas páginas web, no meio do HTML, mas essa é uma programação que é executada no lado do cliente. Você abre seu browser (navegador) e acessa uma página que possui JavaScript. Essa página é carregada na memória da sua máquina, e o código JavaScript é executado consumindo os recursos de processamento do seu computador. Além disso, a programação escrita em JavaScript pode ser vista e copiada por qualquer pessoa. Para isso, basta escolher Exibir > Código-fonte no menu do navegador. O PHP é exatamente o contrário, pois é executado no servidor. Quando você acessa uma página PHP por meio de seu navegador, todo o código PHP é executado no servidor, e os resultados são enviados para seu navegador.

Portanto, o navegador exibe a página já processada, sem consumir recursos de seu computador. As linhas de programação PHP não podem ser vistas por ninguém, já que elas são executadas no próprio servidor, e o que retorna é apenas o resultado do código executado.

Há um exemplo simples para facilitar a compreensão: você já deve ter visto alguns sites que exibem a data e a hora atual em suas páginas. Se essas informações forem escritas utilizando *JavaScript*, a data e a hora mostradas serão retiradas do relógio do seu computador. Ou seja, para cada pessoa que acessar, a data e a hora mostradas serão diferentes, pois nem todos os computadores marcam exatamente o mesmo horário. Agora, se a data e a hora forem escritas utilizando PHP, essas informações serão retiradas do relógio do servidor, ou seja, há um relógio único, e por isso todos que acessarem o site ao mesmo tempo verão a mesma data e a mesma hora.

23.5.4 Bancos de dados

Diversos bancos de dados são suportados pelo PHP, ou seja, o PHP possui código que executa funções de cada um. Entre eles, temos MySQL, PostgreSQL, Sybase, Oracle, SQL Server e muitos outros. Cada um dos bancos de dados suportados pelo PHP possui uma série de funções que você poderá usar em seus programas para aproveitar todos os recursos. Os bancos de dados não suportados diretamente pelo PHP podem ser acessados via ODBC. Veremos exemplos de utilização do MySQL.

23.6. Iniciando em PHP

Um programa PHP pode ser escrito em qualquer editor de texto, como um bloco de notas, porém existem diversos editores específicos para PHP, que exibem cada elemento (variáveis, textos, palavras reservadas etc.) com cores diferentes, para melhor visualização, podemos destacar o *Sublime Text*, *Visual Studio Code* e o *Atom*. Mas antes precisamos configurar o ambiente para que tudo funcione.

23.7. Obtendo o PHP

Antes da instalação do PHP, devemos obter o software, por meio do download gratuito a partir da página oficial www.php.net. Há a necessidade, apenas, de escolher o formato no qual realizar o download e, então, podemos instalar o PHP no sistema.

Existem diversos pacotes de instalação automatizados, os quais facilitam o processo de instalação, pois recomendamos isso no início de vocês por ser mais fácil, além de oferecer versatilidade em relação ao sistema operacional utilizado e às ferramentas ou aos módulos associados. Entre eles, podemos citar XAMPP (geralmente é o que utilizamos), Wampserver, Easy PHP.

O site oficial do PHP disponibiliza documentação, informações e manuais de utilização, além de todo o código-fonte do PHP e também versões pré-compiladas para os sistemas operacionais mais usados, como Windows e Unix.

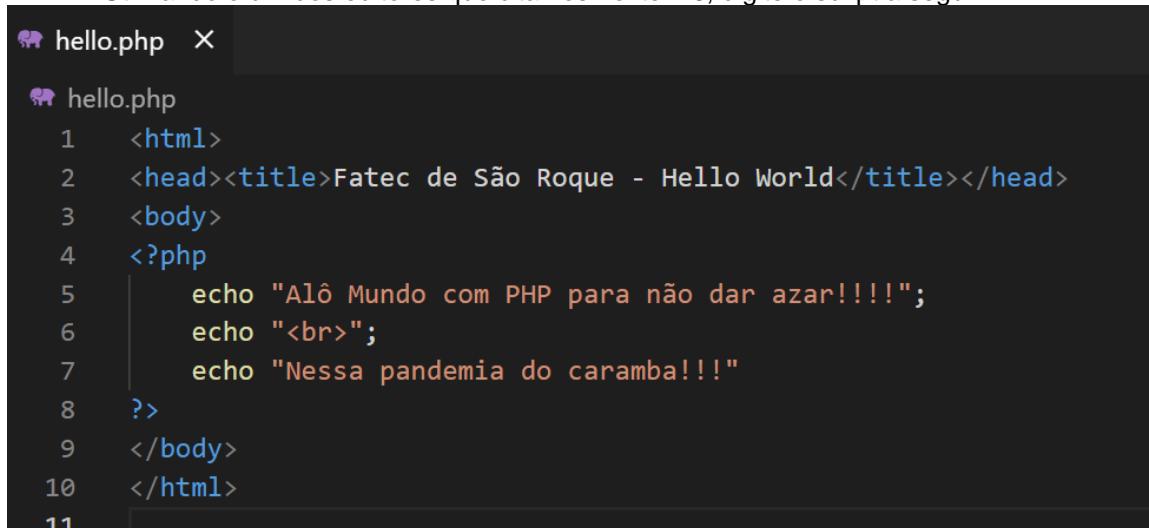
Para a instalação do PHP, os pacotes essenciais são os seguintes:

- Linguagem de programação: PHP;
- Servidor de Internet, sendo o Apache o mais indicado;
- SGBD (Sistema Gerenciador de Banco de Dados), sendo o MySQL o mais indicado.

23.8. Iniciando os trabalhos com o PHP

Vamos criar um pequeno script em PHP para testarmos e verificarmos se o PHP e o servidor Web foram instalados e configurados corretamente no seu computador.

Utilizando o um dos editores que citamos no item 6, digite o script a seguir.



```

hello.php X

hello.php

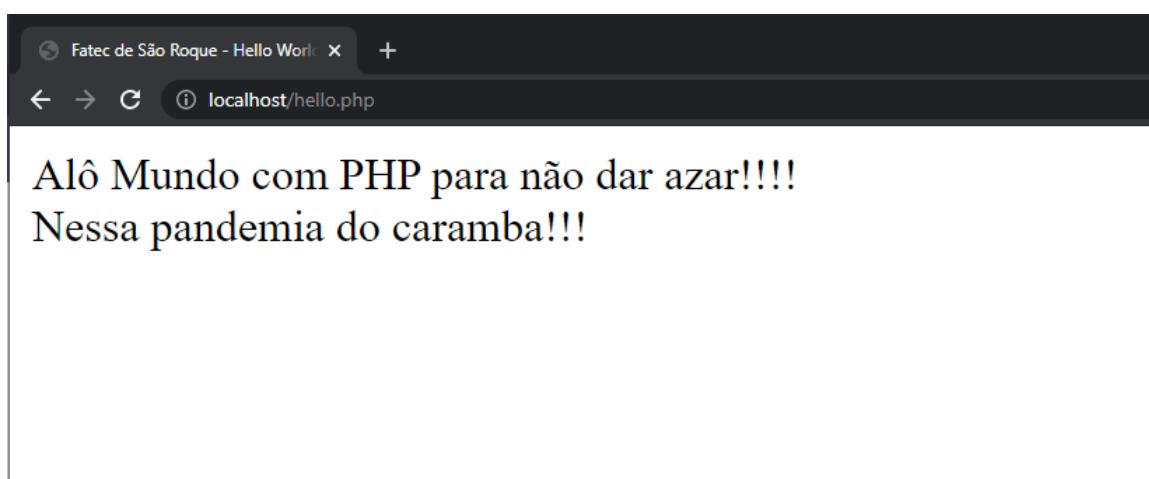
1 <html>
2 <head><title>Fatec de São Roque - Hello World</title></head>
3 <body>
4 <?php
5     echo "Alô Mundo com PHP para não dar azar!!!!";
6     echo "<br>";
7     echo "Nessa pandemia do caramba!!!"
8 ?>
9 </body>
10 </html>
11

```

Salve o arquivo como “hello.php” na pasta C:\XAMPP\htdocs\aula1, esse é o endereço default que o servidor no caso o XAMPP busca os scripts PHP, a pasta **aula1**(você deve criar) é para uma questão de organização dos códigos que iremos trabalhar em nossas aulas, mas fiquem a vontade para modificar se for o caso.

Agora, abra o navegador e, na barra de endereços, digite o seguinte endereço:
<http://localhost/aula1/hello.php>.

Observe que o seu browser exibiu a frase “Alô mundo com PHP para não dar azar!!!”. Sinal de que o PHP e o servidor Web foram instalados e configurados corretamente no seu computador.



23.9. Sintaxe do PHP

23.9.1. Delimitando o código PHP

O código PHP fica embutido no próprio HTML. O interpretador identifica quando um código é PHP pelas seguintes tags:

```
<?php
    comandos;
?>
```

```
<script language="php">
    comandos;
</script>
```

```
<?
    Comandos;
?>
```

```
<%
    comandos;
%>
```

O tipo de tags mais utilizado é o terceiro, que consiste em uma “abreviação” do primeiro. Para utilizá-lo, é necessário habilitar a opção *short-open_tag* na configuração do PHP. O último tipo serve para facilitar o uso por programadores acostumados à sintaxe de ASP. Para utilizá-lo também é necessário habilitá-lo no PHP, através do arquivo de configuração *php.ini*. Em nossas aulas adotarei o primeiro, por uma questão de costume.

23.9.2. Separador de instruções

Entre cada instrução em PHP é preciso utilizar o ponto-e-vírgula, assim como em C, Arduino, Java e outras linguagens mais conhecidas. Na última instrução do bloco de script não é necessário o uso do ponto e vírgula, mas por questões estéticas recomenda-se o uso sempre.

23.9.3. Comentários

Há dois tipos de comentários em código PHP:

Comentários de uma linha:

Marca como comentário até o final da linha ou até o final do bloco de código PHP – o que vier antes. Pode ser delimitado pelo caractere “#” ou por duas barras (//). O delimitador “//”, normalmente, é o mais utilizado.

Exemplo:

```
<?php
    echo "teste"; // este teste é similar ao anterior
    echo "teste"; # este teste é similar ao anterior
    // sql "teste";
?>
```

Comentários de mais de uma linha:

Tem como delimitadores os caracteres “/*” para o início do bloco e “*/” para o final do comentário. Se o delimitador de final de código PHP (?>) estiver dentro de um comentário, não será reconhecido pelo interpretador.

Exemplo:

```
1 <?php
2     echo "teste";
3     /*
4     este é um comentário com mais de uma linha.
5     echo "teste. Este comando echo é ignorado pelo interpretador do PHP por ser um comentário."
6     */
7 ?>
```

23.9.4. Imprimindo código html

Um script php geralmente tem como resultado uma página html, ou algum outro texto. Para gerar esse resultado, deve ser utilizada uma das funções de impressão, echo e print.

Sintaxes:

```
1 <?php
2     print(argumento);
3     echo (argumento1, argumento2, ... );
4     echo argumento;
5 ?>
```

Exemplos:

```
1 <?php
2     $a = 10;
3     $b = 20;
4     $soma = $a + $b;
5 //Imprimindo a soma na tela com o comando print.
6     print ("A soma é: ". $soma."<p>");
7
8 //Imprimindo a soma na tela com o comando echo.
9     echo ("A soma é: ". $soma."<p>");
10    echo "<br>";
11    echo $a." ".$b;
12 ?>
```

23.10 – Variáveis

23.10.1. Nomes de variáveis

Toda variável em PHP tem seu nome composto pelo caractere \$(dólar) e uma string, que deve iniciar por uma letra ou o caractere “_”. O PHP é case sensitive, ou seja, as variáveis \$vivas e \$VIVAS são diferentes.

Por isso é preciso ter muito cuidado ao definir os nomes das variáveis. É bom evitar os nomes em maiúsculas, pois como veremos mais adiante, o PHP já possui algumas variáveis pré-definidas cujos nomes são formados por letras maiúsculas.

Exemplo de variáveis:

- \$teste = nome correto de variável
- teste= nome errado de variável.
- _teste= nome correto de variável

Obs.: Normalmente, a variável é criada utilizando-se, na maioria das vezes, o caracter \$ (dólar).

23.10.2. Tipos Suportados

O PHP suporta os seguintes tipos de dados:

- Inteiro
- Ponto flutuante
- String
- Array
- Objeto

O PHP utiliza checagem de tipos dinâmica, ou seja, uma variável pode conter valores de diferentes tipos em diferentes momentos da execução do script. Por este motivo não é necessário declarar o tipo de uma variável para usá-la. O interpretador PHP decidirá qual o tipo daquela variável, verificando o conteúdo em tempo de execução.

Inteiros (integer ou long)

Uma variável pode conter um valor inteiro com atribuições que sigam as seguintes sintaxes:

- \$vivas = 1234; # inteiro positivo na base decimal;
- \$vivas = -234; # inteiro negativo na base decimal;
- \$vivas = 0234; # inteiro na base octal-simbolizado pelo 0 equivale a 156 decimal;

- \$vivas = 0x34; # inteiro na base hexadecimal(simbolizado pelo 0x) – equivale a 52 decimal.

A diferença entre inteiros simples e long está no número de bytes utilizados para armazenar a variável.

Como a escolha é feita pelo interpretador PHP de maneira transparente para o usuário, podemos afirmar que os tipos são iguais.

Ponto Flutuante (double ou float)

Uma variável pode ter um valor em ponto flutuante com atribuições que sigam as seguintes sintaxes:

- \$vivas = 1.234;
- \$vivas = 23e4; # equivale a 230.000

Strings (Atenção)

Strings podem ser atribuídas de duas maneiras:

a) utilizando aspas simples (' ') – Desta maneira, o valor da variável será exatamente o texto contido entre as aspas (com exceção de \\" e \' – ver exemplo abaixo)

b) utilizando aspas duplas (" ") – Desta maneira, qualquer variável ou caractere de escape será expandido antes de ser atribuído.

Exemplo:

```

1 <?php
2     $teste = "Marcos";
3     $vivas = '---$teste--\n';
4     $texto = "2º DSN";
5     echo "$vivas";
6 ?>

```

A saída desse script será "---\$teste--\n".

```

1 <?php
2     $texto = "2º DSN"; Nova linha
3     $teste = "Marcos";
4     $vivas = "---$teste---\n";
5     echo "$vivas";
6 ?>

```

A saída desse script será "---Marcos--" (com uma quebra de linha no final).

Arrays

Arrays em PHP podem ser observados como mapeamentos ou como vetores indexados. Mais precisamente, um valor do tipo array é um dicionário onde os índices são as chaves de acesso. Vale ressaltar que os índices podem ser valores de qualquer tipo e não somente inteiros. Inclusive, se os índices forem todos inteiros, estes não precisam formar um intervalo contínuo. Como a checagem de tipos em PHP é dinâmica, valores de tipos diferentes podem ser usados como índices de array, assim como os valores mapeados também podem ser de diversos tipos.

Exemplo:

```

1 <?php
2     $cor[0] = "amarelo";
3     $cor[1] = "vermelho";
4     $cor[2] = "verde";
5     $cor[3] = "azul";
6     $cor[4] = "anil";
7     $cor["teste"] = 1;
8 ?>

```

Equivalentemente, pode-se escrever:

```

1 <?php
2     $cor = array(0 => "amarelo", 1 => "vermelho", 2 => "verde", 3 => "azul", 4 => "anil", teste => 1);
3 ?>

```

Listas

As listas são utilizadas em PHP para realizar atribuições múltiplas.

Através de listas é possível atribuir valores que estão num array para variáveis. Vejamos o exemplo:

```
1 <?php
2     list($a, $b, $c) = array("a", "b", "c");
3 ?>
```

O comando acima atribui valores às três variáveis simultaneamente. É bom notar que só são atribuídos às variáveis da lista os elementos do array que possuem índices inteiros e não negativos. No exemplo acima as três atribuições foram bem sucedidas porque ao inicializar um array sem especificar os índices eles passam a ser inteiros, a partir do zero.

Booleanos

O PHP não possui um tipo booleano, mas é capaz de avaliar expressões e retornar true ou false, através do tipo integer: é usado o valor 0 (zero) para representar o estado false, e qualquer valor diferente de zero (geralmente 1) para representar o estado true.

23.10.3. Transformação de tipos

A transformação de tipos em PHP pode ser feita das seguintes maneiras:

a) Coerções

Quando ocorrem determinadas operações (“+”, por exemplo) entre dois valores de tipos diferentes, o PHP converte o valor de um deles automaticamente (coerção). É interessante notar que se o operando for uma variável, seu valor não será alterado.

O tipo para o qual os valores dos operandos serão convertidos é determinado da seguinte forma: Se um dos operandos for float, o outro será convertido para float, senão, se um deles for integer, o outro será convertido para integer.

Exemplo:

```
<?php
//declaração da variável vivas
//-----
$vivas = "1"; // $vivas é a string "1"
$vivas = $vivas + 1;// $vivas é o integer 2
$vivas = $vivas + 3.7;// $vivas é o double 5.7
$vivas = 1 + 1.5; // $vivas é o double 2.5
echo "O valor de vivas é: $vivas";

?>
```

Como podemos notar, o PHP converte string para integer ou double mantendo o valor. O sistema utilizado pelo PHP para converter de strings para números é o seguinte:

- É analisado o início da string. Se contiver um número, ele será avaliado. Senão, o valor será 0 (zero);
- O número pode conter um sinal no início (“+” ou “-”);
- Se a string contiver um ponto em sua parte numérica a ser analisada, ele será considerado, e o valor obtido será double;
- Se a string contiver um “e” ou “E” em sua parte numérica a ser analisada, o valor seguinte será considerado como expoente da base 10, e o valor obtido será double;

Exemplos:

```
1 <?php
2     $vivas = 1 + "10.5"; // $vivas == 11.5
3     $vivas = 1 + "-1.3e3"; // $vivas == -1299
4     $vivas = 1 + "teste10.5"; // $vivas == 1
5     $vivas = 1 + "10testes"; // $vivas == 11
6     $vivas = 1 + " 10testes"; // $vivas == 11
7     $vivas = 1 + "+ 10testes"; // $vivas == 1
8     echo "O valor de vivas é: $vivas";
9 ?>
```

b) Transformação explícita de tipos

A sintaxe do typecast de PHP é semelhante ao C: basta escrever o tipo entre parênteses antes do valor.

Exemplo:

```
<?php
    $vivas = 15; // $vivas é integer (15)
    $vivas = (double) $vivas; // $vivas é double (15.0)
    $vivas = 3.9; // $vivas é double (3.9)
    $vivas = (int) $vivas; // $vivas é integer (3)
    // o valor decimal é truncado
    echo "O valor de vivas é: $vivas";
?>
```

Os tipos permitidos na Transformação explícita são:

- (int), (integer) ----- = muda para integer;
- (real), (double), (float) ----- = muda para float;
- (string)----- = muda para string;
- (array)----- = muda para array;
- (object) ----- = muda para objeto.

Com a função settype

A função settype converte uma variável para o tipo especificado, que pode ser “integer”, “double”, “string”, “array” ou “object”.

Sintaxe:

Settype(nomedavariável,novo tipo da variável)

Exemplo:

```
<?php
    $vivas = 15; // $vivas é integer
    settype($vivas,double); // $vivas é double
    $vivas = 15; // $vivas é integer
    settype($vivas,string); // $vivas é string
    echo "O valor de vivas é: $vivas";
?>
```

23.11. Operadores

23.11.1 Operadores Aritméticos

Só podem ser utilizados quando os operandos são números (integer ou float). Se forem de outro tipo, terão seus valores convertidos antes da realização da operação. São eles:

- + adição
- subtração
- * multiplicação
- / divisão
- % módulo

23.11.2. Operadores de Strings

Só há um operador exclusivo para strings:

- . concatenação

Exemplo:

```
1 <?php
2     $nome = "Marcos";
3     $sobrenome = "Costa";
4     echo $nome . ".$sobrenome;
5 ?>
```

A saída do script acima será: Marcos Costa

23.11.3. Operadores de atribuição

Existe um operador básico de atribuição e diversos derivados. Sempre retornam o valor atribuído. No caso dos operadores derivados de atribuição, a operação é feita entre os dois

operandos, sendo atribuído o resultado para o primeiro. A atribuição é sempre por valor, e não por referência.

- = atribuição simples
- += atribuição com adição
- = atribuição com subtração
- *= atribuição com multiplicação
- /= atribuição com divisão
- %= atribuição com módulo
- .= atribuição com concatenação

Exemplo:

```

1 <?php
2     $a = 7;
3     $a += 2; // $a passa a conter o valor 9
4 ?>
5

```

23.11.4. Operadores Lógicos

Utilizados para inteiros representando valores booleanos

- and “e” lógico
- or “ou” lógico
- xor ou exclusivo
- ! não (inversão)
- && “e” lógico
- || “ou” lógico

Existem dois operadores para “e” e para “ou” porque eles têm diferentes posições na ordem de precedência.

23.11.5. Operadores de Comparação

As comparações são feitas entre os valores contidos nas variáveis, e não as referências. Sempre retornam um valor booleano.

- == igual a
- === igual (conteúdo) e também (tipo de dado);
- != diferente de
- < menor que
- > maior que
- <= menor ou igual a
- >= maior ou igual a

23.11.6. Operadores de incremento e decremento

- ++ incremento
- -- decremento

Podem ser utilizados de duas formas: antes ou depois da variável. Quando utilizado antes, retorna o valor da variável antes de incrementá-la ou decrementá-la.

Quando utilizado depois, retorna o valor da variável já incrementado ou decrementado.

Exemplo:

```

1 <?php
2     $a = $b = 10; // $a e $b recebem o valor 10
3     $c = $a++; // $c recebe 10 e $a passa a ter 11
4     $d = ++$b; // $d recebe 11, valor de $b já incrementado
5 ?>

```

23.12. Estruturas de Controle

As estruturas que veremos a seguir são comuns para as linguagens de programação imperativas, bastando, portanto, descrever a sintaxe de cada uma delas, resumindo o funcionamento.

23.12.1. Blocos

Um bloco consiste de vários comandos agrupados com o objetivo de relacioná-los com determinado comando ou função. Em comandos como if, for, while, switch e em declarações de funções blocos podem ser utilizados para permitir que um comando faça parte do contexto desejado. Blocos em PHP são delimitados pelos caracteres “{” e “}”. A utilização dos delimitadores de bloco em uma parte qualquer do código não relacionada com os comandos citados ou funções não produzirá efeito algum, e será tratada normalmente pelo interpretador.

Exemplo:

```
1 <?php
2 if ($x == $y)
3     comando1;
4     comando2;
5 ?>
```

Para que comando2 esteja relacionado ao if é preciso utilizar um bloco:

```
1 <?php
2     if ($x == $y) {
3         comando1;
4         comando2;
5     }
6 ?>
```

23.12.2. Estrutura if / else / elseif

O mais trivial dos comandos condicionais é o if. Ele testa a condição e executa o comando indicado se o resultado for true (valor diferente de zero). Ele possui duas sintaxes:

Sintaxes:

```
if (expressão)
    comando;

if (expressão) {
    comando;
    ...
    comando;
}
```

Para incluir mais de um comando no if da primeira sintaxe, é preciso utilizar um bloco, demarcado por chaves.

Exemplo 1:

```
1 <?php
2     $a = 10;
3     $b = 20;
4     if ($a > $b)
5         echo "o Valor de a é: ".$a + $b;
6 ?>
```

Exemplo 2:

```
1 <?php
2     $a = 30;
3     $b = 20;
4     if ($a >= $b) {
5         $media = ($a + $b) / 2;
6         $resto = $a % $b;//calcula o resto da divisão de A por B
7         echo " o Valor de A é: ".$a."<br>";
8         echo " o Valor de B é: ".$b."<br>";
9         echo " o Valor da média é: ".$media."<br>";
10        echo " O resto da divisão de A por B é: ".$resto;
11    }
12 ?>
```

23.12.2.1 Else

O else é um complemento opcional para o if. Se utilizado, o comando será executado se a expressão retornar o valor false (zero). Suas duas sintaxes são:

Sintaxes:

```
if (expressão)
    comando;
else
    comando;
```

ou

```
if (expressão) {
    comando 1;
    comando n
} else {
    comando 1;
    comando n
}
```

Exemplos:

```
<?
    $a = 10;
    $b = 20;
    if ($a > $b) {
        $maior = $a;
    } else {
        $maior = $b;
    }
?>
```

O exemplo acima coloca em \$maior o maior valor entre \$a e \$b.

Em determinadas situações é necessário fazer mais de um teste, e executar condicionalmente diversos comandos ou blocos de comandos. Para facilitar o entendimento de uma estrutura do tipo:

```
if (expressao1)
    comando1;
else
    if (expressao2)
        comando2;
    else
        if (expressao3)
            comando3;
        else
            comando4;
```

Foi criado o comando, também opcional elseif. Ele tem a mesma função de um else e um if usados sequencialmente, como no exemplo acima. Num mesmo if podem ser utilizados diversos elseif's, ficando essa utilização a critério do programador, que deve zelar pela legibilidade de seu script.

23.12.2.2 Elseif

O comando elseif também pode ser utilizado com dois tipos de sintaxe. Em resumo, a sintaxe geral do comando if fica das seguintes maneiras:

Sintaxe:

```

if (expressao1)
    comando;
elseif (expressao2)
    comando;
else
    comando;
ou
} elseif (expressão 1) {
    comando 1;
    comando n;
}
else if (expressão 2) {
    comando 1;
    comando n;
}
else {
    comando 1;
    comando n;
}

```

Exemplo:

```

<?php
    $nota1 = 6;
    $nota2 = 8;
    $media = ($nota1 + $nota2) / 2;

    if ($media > 7) {
        echo "Média: ".$media."<br>";
        echo "Aluno aprovado.";
    }elseif ($media <7) {
        echo "Média: ".$media."<br>";
        echo "Aluno reprovado.";
    }else {
        echo "Média: ".$media."<br>";
        echo "Aluno em recuperação.";
    }
?>

```

23.12.4. Estrutura Switch Case

O comando switch atua de maneira semelhante a uma série de comandos if na mesma expressão.

Frequentemente o programador pode querer comparar uma variável com diversos valores, e executar um código diferente a depender de qual valor é igual ao da variável. Quando isso for necessário, deve-se usar o comando switch.

Sintaxe:

```

switch ($valor){
    case "____";
        comandos;
        comandos;
        break;
    case "____";
        comandos;
        comandos;
        break;
    case "____";
        comandos;
        comandos;
        break;
    default;
        comandos;
        comandos;
        break;
}

```

O exemplo seguinte mostra dois trechos de código que fazem a mesma coisa, sendo que o primeiro utiliza uma série de if's e o segundo utiliza switch:

```
<?
    $i = 1;
    if ($i == 0){
        print "i é igual a zero";
    }elseif ($i == 1){
        print "i é igual a um";
    }elseif ($i == 2){
        print "i é igual a dois";
    }
?>
```

Exemplo 2: Estrutura SWITCH

```
<?
    $i = 0;
    switch ($i) {
        case 0:
            print "i é igual a zero";
            break;
        case 1:
            print "i é igual a um";
            break;
        case 2:
            print "i é igual a dois";
            break;
    }
?>
```

É importante compreender o funcionamento do switch para não cometer enganos. O comando switch testa linha a linha os cases encontrados, e a partir do momento que encontra um valor igual ao da variável testada, passa a executar todos os comandos seguintes, mesmo os que fazem parte de outro teste, até o fim do bloco.

Por isso usa-se o comando break, quebrando o fluxo e fazendo com que o código seja executado da maneira desejada. Veremos mais sobre o break mais adiante.

Em outras linguagens que implementam o comando switch, ou similar, os valores a serem testados só podem ser do tipo inteiro.

Em PHP é permitido usar valores do tipo String como elementos de teste do comando switch. O exemplo abaixo funciona perfeitamente:

```
<?
    $s = "casa";
    switch ($s) {
        case "casa":
            print "A casa é amarela";
            break;
        case "arvore":
            print "A árvore é bonita";
            break;
    }
?>
```

23.12.5. Estruturas de Repetição

As estruturas de repetição são utilizadas quando o programador precisa, por exemplo, repetir o mesmo comando várias vezes. Vamos ver as estruturas de repetição existentes.

12.5.1. While

O while é o comando de repetição (laço) mais simples. Ele testa uma condição e executa um comando, ou um bloco de comandos, até que a condição testada seja falsa. Assim como o if, o while também possui duas sintaxes alternativas:

Sintaxes:

```
While (condição)
    comando;
```

Ou

```
While (condição) {
    comandos;
    comandos;
}
```

Exemplo:

O exemplo a seguir mostra o uso do while para imprimir os números de 1 a 10:

```
<?php
    $i = 1;
    while ($i <=10){
        print $i++;
        print "<br>";
    }
?>
```

23.12.5.2. Do ... While

O laço do..while funciona de maneira bastante semelhante ao while, com a simples diferença que a expressão é testada ao final do bloco de comandos.

Sintaxe:

```
do {
    comando
    ...
    comando
}
while (<condição>);
```

Exemplo:

```
<?php
    $a = 1;
    do {
        echo $a++;
        echo "<br>";
    }
    while ($a <=10)
?>
```

23.15.5.3. For

O tipo de laço mais complexo é o for. Para os que programam em C, C++ ou Java, a assimilação do funcionamento do for é natural. Mas para aqueles que estão acostumados a linguagens como Pascal, há uma grande mudança para o uso do for.

Sintaxe:

```
for (<inicializacao>;<condicao>;<incremento>) {
    <comando>;
    ...
    <comando>;
}
```

As três expressões que ficam entre parênteses têm as seguintes finalidades:

- Inicialização: comando ou seqüência de comandos a serem realizados antes do inicio do laço. Serve para inicializar variáveis.
- Condição: Expressão booleana que define se os comandos que estão dentro do laço serão executados ou não. Enquanto a expressão for verdadeira (valor diferente de zero) os comandos serão executados.
- Incremento: Comando executado ao final de cada execução do laço.

Um comando for funciona de maneira semelhante a um while escrito da seguinte forma:

Exemplo:

```
<?php
    for ($a = 1; $a <= 10; $a++) {
        echo "O valor de A é: ". $a;
        echo "<br>";
    }
?>
```

23.15.5.3. Foreach

O FOREACH tem o mesmo funcionamento do FOR, porém, ele não precisa de contador. Ou seja, você vai correr o laço sabendo em que posição está. Veja abaixo o primeiro exemplo da sintaxe do FOREACH:

```
1  <?php
2      $frutas = array('maca','banana','melancia','melao',
3                         'abacaxi','laranja');
4      foreach($frutas as $fruta)
5      {
6          echo "A fruta é: ". $fruta . "<br>";
7      }
8 ?>
```

23.16 – Exercícios para fixação com validade da P3:

Criem as soluções em PHP, lembrando que não há necessidade de front end neste momento, façam o que é pedido:

1- Crie um algoritmo que receba um número digitado pelo usuário e verifique se esse valor é positivo, negativo ou igual a zero. A saída deve ser: "Valor Positivo", "Valor Negativo" ou "Igual a Zero".

2 - Faça um algoritmo PHP que receba os valores A e B, imprima-os em ordem crescente em relação aos seus valores. Exemplo, para A=5, B=4. Você deve imprimir na tela: "4 5". Dica, utilizem concatenação.

3 - Crie um algoritmo para calcular a média final de um aluno, para isso, solicite a entrada de três notas e as insira em um array, por fim, calcule a média geral. Caso a média seja maior ou igual a seis, exiba aprovado, caso contrário, exiba reprovado. Exiba também a média final calculada.

Ex: N1 = 5 | N2 = 10 | N3 = 4 | MG = 6,33 [Aprovado].

4 - Ler um número inteiro entre 1 e 12 e escrever o mês correspondente. Caso o número seja fora desse intervalo, informar que não existe mês com este número. Exigência, resolva esse exercício utilizando array.

Observação: Crie as soluções abaixo utilizando a estrutura switch case:

5 - Elabore um algoritmo que leia dois valores do usuário e a operação que ele deseja executar (Operações: soma, subtração, divisão, multiplicação). Execute a operação desejada e mostre o resultado;

6 - Faça um algoritmo que aborde a seguinte situação: Uma loja fornece 10% de desconto para funcionários e 5% de desconto para clientes vips. Faça um programa que calcule o valor total a ser pago por uma pessoa. O script deverá ler o valor total da compra e um código que identifique se o comprador é um cliente comum (1), funcionário (2) ou vip (3);

7 - Faça um algoritmo PHP que calcule e imprima o salário reajustado de um funcionário de acordo com a seguinte regra:

- salários até 300, reajuste de 50%
- salários maiores que 300, reajuste de 30%.

Observação: Crie as soluções abaixo utilizando estruturas de repetição de acordo com sua escolha:

8 - Faça um algoritmo em PHP que receba um valor qualquer e imprima os valores de 0 até o valor recebido, exemplo:

- Valor recebido = 9
- Impressão do programa – 0 1 2 3 4 5 6 7 8 9

9 - Faça um algoritmo PHP que receba um valor qualquer e calcule o seu fatorial (!), sabendo que fatorial de um número é: $7! = 7*6*5*4*3*2*1$ $4! = 4*3*2*1$

10 - Faça um algoritmo PHP que receba dois valores quaisquer e imprime todos os valores intermediários a ele, veja exemplo: Primeiro Valor = 5 Segundo Valor = 15 Imprime: 6 7 8 9 10 11 12 13 14

11- Pesquise. Faça um algoritmo PHP que receba uma string, encontre o número total de caracteres desta e imprima todos os números que existem entre 0 e o número total, exemplo:

```
* string = "Gil Eduardo de Andrade"
* total_caracter = 22
Imprime: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
```

Pessoal, criem os scripts .php, vocês terão até o dia 28/11/2024, para efetivar a entrega, isso acontecerá da seguinte forma: Vocês poderão subir as resoluções no GitHub se preferirem, ou gerar um arquivo .zip e enviar na atividade que abrirei no Microsoft Teams em nossa disciplina, caso optem pelo GitHub, na atividade no Teams, só informem o link do mesmo para que eu possa corrigir.

Na semana do dia 03/12, irei na continuidade destas notas de aula, disponibilizar a resolução destes exercícios.