Unidade 1: Introdução à Linguagem Java

Variáveis e Operadores

Prof. Daniel Caetano

Objetivo: Apresentar a linguagem Java, suas principais regras, declarações de variáveis e os principais operadores.

Bibliografia: DEITEL, 2005; CAELUM, 2010; HOFF, 1996.

INTRODUÇÃO

O curso de Linguagem de Programação tem o objetivo de iniciar o aluno na área da programação e prepará-lo para o desenvolvimento de aplicações básicas em Java, além de prepará-lo para compreender códigos e identificar falhas. Nesta primeira unidade, serão conhecidos os fundamentos da linguagem Java, desde suas origens e objetivos até a sintaxe básica, tipos de variáveis e operadores.

1. O QUE É UMA LINGUAGEM DE PROGRAMAÇÃO ?

Muitas pessoas podem pensar que uma linguagem de programação seja algo complexo. Embora não sejam de aprendizado automático, as linguagens de programação são linguagens bastante simples, com poucos elementos e grande parte das pessoas são capazes de aprendê-las em poucos dias.

O aluno, neste instante, pode não ter muita certeza disso mas, para ajudá-lo a compreender a razão pela qual uma linguagem de computador é bastante simples, é útil entender o conceito de linguagem ou língua. O que é uma língua?

Uma língua é um conjunto de símbolos e regras com as quais estes símbolos são combinados, para que seja possível transmitir uma idéia. Na língua portuguesa, por exemplo, teríamos as palavras, sinais de pontuações, ortografia e a gramática.

A parte mais complicada da linguagem de programação talvez seja, então, a programação, e não a linguagem. Mas o que é programação? Programação é a definição de uma sequência de passos para cumprir um determinado objetivo. Um exemplo clássico seria uma agenda de um projeto ou uma receita de bolo.

Assim, a linguagem de programação é uma linguagem que serve para descrever passos para se atingir um determinado objetivo. Usar essa linguagem não é difícil, como veremos; a parte mais complexa é, muitas vezes, definir qual é a sequência de passos que precisa ser realizada.

1.1. Definindo um Programa

Como apresentado anteriormente, uma linguagem de programação serve para descrever uma sequência de passos para cumprir um objetivo. Tal sequência de passos pode ser chamada de um **programa**.

Os programas estão em nosso dia-a-dia, ainda que nem sempre pensemos nisso. São programas uma lista de compras, uma lista de afazeres, instruções de uso de aparelhos... Entretanto, nem sempre estes programas são completos, isto é, descrições completas do que precisa ser feito. Uma lista de compras não costuma indicar onde comprar os produtos, ou a data e hora em que isso deve ser feito.

O mesmo ocorre com manuais de instruções. Um manual de DVD pode descrever a seguinte sequência para seu uso:

a) Ligue os cabos
b) Ligue o aparelho de TV e o DVD
c) Insira um DVD

Observe que são instruções genéricas e uma porção de dúvidas surgem: ligar quais cabos e onde? Ligar qual aparelho de TV? Inserir um DVD onde? O que é uma TV e um DVD? O que é um cabo?

Está claro, assim, que a linguagem em si - no caso, o português - não é suficiente para que as instruções sejam compreendidas. O que está faltando?

Basicamente, o que está faltando é definir as instruções de forma não ambígua, isto é, sem nenhum tipo de dúvida acerca do que estas instruções significam. Adicionalmente, talvez seja necessário detalhar um pouco mais estas instruções.

Esta costuma ser a maior dificuldade em se **programar um computador**: estamos acostumados a criar programas; só que estamos acostumados a criar programas que são interpretados por pessoas, que "adivinham" o que queremos com este programa e o cumprem - às vezes a contento, às vezes não.

Quando vamos programar um computador, não podemos agir desta maneira, assim como não aceitaremos se "às vezes ele agir a contento e às vezes não". Ao programar um computador, precisamos ser **específicos**, isto é, **dizer exatamente** o que queremos, e o computador seguirá à risca.

1.2. Usando a Enciclopédia

Obviamente ter que explicar **tudo** em detalhe para um computador pode ser uma tarefa bastante chata e, em alguns casos, árdua. Até mesmo um tarefa simples, como imprimir uma letra na tela, pode exigir centenas e centenas de linhas de código.

Como imprimir um caractere é uma tarefa muito comum - assim como diversas outras, as linguagens de programação costumam vir acompanhadas de um pacote contendo muitas e muitas destas tarefas pré-programadas, cada uma delas com um nome. Desta forma, para imprimir um caractere, basta pedir à linguagem que use o código pronto que imprime o caractere. Isso é feito pelo **nome** da tarefa, que pode ser algo simples como "**imprimir**".

Estes pacotes de tarefas prontas, que podem ser encarados como mini-programas, são chamados de **bibliotecas** e sua utilidade é enorme, pois eles reduzem em muito o nosso trabalho como desenvolvedores.

2. A LINGUAGEM JAVA

Existem diversas linguagens de programação disponíveis no mercado. Praticamente todas elas possuem características semelhantes, como possibilitarem a descrição precisa dos passos que uma tarefa precisa para ser concluída e disponibilizarem uma biblioteca com tarefas complexas pré-programadas.

Os símbolos e regras são bastante similares na maioria destas linguagens, estando a maior diferença entre elas justamente nas bibliotecas. Dependendo do tipo de uso que foi imaginado para uma linguagem - isto é, se ela é para a web, para banco de dados, para cálculos matemáticos etc., sua biblioteca englobará tarefas diferentes.

Assim, antes de nos aprofundarmos no estudo das linguagens de programação com o uso da linguagem Java, vamos conhecer um pouco do histórico da linguagem Java.

2.1. Histórico do Java

A linguagem Java foi concebida como uma linguagem para desenvolvimento de produtos eletrônicos de consumo (eletrodomésticos e eletro-eletrônicos), com software embarcado. Entretanto, ela acabou se popularizando apenas com o advento da World Wide Web e apenas recentemente vem voltando à sua vocação incial.

Origens

No início da década de 1990 estavam se popularizando os equipamentos eletro-eletrônicos programáveis/programados, indo desde televisores até fornos de microondas e geladeiras. Embora muitas empresas tivessem notado que as linguagens

existentes traziam problemas para o desenvolvimento destes equipamentos, foi a Sun Microsystems quem primeiro propôs uma solução.

Antes de entendermos qualquer tipo de solução, é importante entendermos qual era o problema, que talvez não seja óbvio para aqueles que nunca trabalharam com projeto de equipamentos eletro-eletrônicos.

Sempre que um projeto é realizado, uma decisão importante que deve ser feita é a definição de quais serão os componentes do equipamento que está sendo projetado. No caso de um equipamento eletrônico, componentes importantes são os eletrônicos, em especial os circuitos integrados e, no caso dos eletro-eletrônicos programáveis (ou programados), os microprocessadores.

Via de regra, o processador selecionado é aquele que tiver o menor custo, dado que atende às características básicas do projeto. Entretanto, um eletro-eletrônico pode continuar sendo produzido e vendido por vários anos; por outro lado, o preço dos processadores não é estático ao longo destes mesmos anos, fazendo com que o "processador mais barato que atenda às necessidades" possa mudar com o tempo. Nestas situações, em geral os equipamentos voltam para a prancheta e são redesenhados para acomodar um novo processador, por exemplo.

É importante ressaltar que uma economia de alguns reais em cada unidade pode levar a grandes lucros para a empresa, visto que dezenas de milhares de unidades daquele eletro-eletrônico são produzidas ao longo de um ano: um aumento de lucro que as empresas em geral não desprezam. Exemplos, em casos de video-games (SMS1/2/3/Compact, MD1/2/3, PS/PSOne, PS2/PS2Slim, PS3/PS3Slim, XBox/XBox360, GameCube/NintendoWii...)

Entretanto, a troca de um processador muitas vezes implica em troca de todo o software, já que usualmente processadores distintos têm linguagens de máquina distintas. O problema então surge: a necessidade de se re-compilar e, muitas vezes, reescrever um software para o novo processador... acaba com grande parte do lucro obtido com a troca do processador. E, mesmo quando isso não ocorria, muitas vezes significava novos "bugs" e problemas, algo bastante indesejável.

De olho nisso, em 1990, James Gosling começou a trabalhar em uma linguagem que funcionasse de tal forma que os programas raramente precisassem ser reescritos quando a plataforma onde são executados fosse substituída, desde que ambas oferecessem recursos similares. Essa linguagem acabou por ficar conhecida como Linguagem Java.

Projetos Iniciais

Raramente uma linguagem baseada apenas em teoria e sem experimentação prática consegue ter sucesso. Por esta razão, os técnicos da Sun Microsystems, durante o desenvolvimento do Java desenvolveram projetos em Java, para testar suas funcionalidades.

O primeiro destes projetos foi o Projeto Green, que visava a criação de uma nova interface com o usuário para o equipamento "*7" (Star Seven), que tinha o objetivo de controlar os eletrodomésticos de uma casa através de ícones animados e uma touch screen. Um outro projeto foi o de VoD (Video On Demand), com uma função similar ao que hoje se chama de TV Interativa.

Entretanto, foi com o surgimento da Web que a nova linguagem realmente apareceu a público: os navegadores web estavam em franca evolução quando a Sun apresentou o WebRunner, mais tarde renomeado para HotJava. A principal característica destes browsers não era exatamente a renderização HTML (o que eles faziam de forma similar aos já existentes Mosaic e Netscape), mas sim o fato de terem capacidade de executar applets java, pequenos programas que rodavam no computador do usuário, fosse esse computador IBM PC ou Apple MacIntosh.

A inovação fez tanto sucesso que em poucas semanas a Netscape lançava sua primeira versão capaz de executar a Java Virtual Machine da Sun como plugin e, com isso, executar também applets java. Mais tarde foi incorporado no browser da Netscape também o JavaScript e, rapidamente, ambos se tornaram padrões tão importantes que é quase impossível navegar hoje sem os mesmos instalados, juntamente com o Macromedia Flash.

O Java Hoje

O tempo foi passando e mostrou que a Sun Microsystems, de alguma forma, estava adiante de seu tempo. Com o surgimento dos PDAs (Personal Data Assistants, os "PALMs") e telefones celulares capazes de executar aplicativos, tornou-se bastante atrativa uma tecnologia que permitisse que um programa pudesse ser executado em máquinas diferentes: afinal de contas, não só os recursos disponíveis nestes equipamentos, como também seus processadores e arquiteturas podem ser bastante diferentes até mesmo de um modelo para outro!

Assim, hoje o Java voltou a ter sua vocação inicial: desenvolvimento de software embaracado em eletro-eletrônicos. Ainda não é muito comum, mas vem crescendo o número de equipamentos como Set-Top-Boxes (HDTV), modems ADSL, computadores portáteis, DVD players, TVs e outros equipamentos que se utilizam de programas escritos na linguagem Java para permitir que o usuário se comunique com o equipamento.

2.2. Como Funciona o Java

Como um programa em Java consegue rodar em qualquer lugar? Como um código feito para "máquina nenhuma" consegue rodar em qualquer lugar? Na verdade, o segredo está no **Interpretador Java**, também conhecido como **Java Virtual Machine (JVM)**, que é um programa que precisa ser reescrito para cada processador e equipamento.

A JVM exerce o papel de um "tradutor simultâneo". É ela quem lê o programa Java e diz para um computador específico o que deve ser feito para realizar aquela tarefa. Ela

funciona como um intermediário. É como um intérprete de um técnico de futebol que não fala a língua dos jogadores:

Nome do Técnico	*Língua do Técnico*	*Conversão*	*Língua dos Jogadores*
Luis Felipe	Português	Intérprete P/A	Árabe
Luis Felipe	Português	Intérprete P/I	Inglês
Luis Felipe	Português	Intérprete P/J	Japonês

Nome do Programa	*Línguagem do Programa*	*Conversão*	*Línguagem do Processador*
MeuPrograma	Java	JVM J/P4	Pentium IV ASM
MeuPrograma	Java	JVM J/PPC	PowerPC ASM
MeuPrograma	Java	JVM J/A7	ARM7 ASM

Perceba que ao trocar a língua do time, não é preciso trocar o técnico nem a língua que ele fala, pois existe um intérprete que faz as traduções. Se trocar o time e mantiver o técnico, basta trocar o intérprete. No caso do programa em Java, ocorre o mesmo: não é preciso trocar o programa nem a linguagem dele quando se troca de processador: basta trocar a JVM.

Como existe um passo a mais de tradução, isso tem influência direta no desempenho das aplicações Java. Apesar de aplicações Java possuírem um desempenho bastante superior ao de linguagens script normais, seu desempenho pode ser bastante mais lento que uma linguagem compilada como C. Entretanto, os fabricantes não têm se mostrado muito preocupados com esse "problema", dado que os equipamentos têm poder de processamento cada vez maior a custos cada vez menores: preservar o investimento em software desenvolvido acaba sendo muito mais importante quando se visa lucro em alguns mercados (como o dos celulares).

Nas versões mais recentes, a Sun se empenhou em resolver o problema "desempenho", sempre associado à linguagem Java. Para isso criaram um sistema chamado de "hotspots", com o uso da tecnologia JIT (Just-in-Time), que compilam o código à medida em que ele é executado, com grande otimização, permitindo que, em muitos casos, programas em Java de versão recente sejam executados em velocidade similar a programas em C ou C++.

2.3. O Java é bom para o Programador?

Além de todas estas vantagens, o Java ainda tem uma porção de vantagens para os programadores, pois o interpretador Java cuida de uma série de aspectos, como gerenciamento de memória, e a programação é feita de tal maneira que ajuda o programador a organizar melhor o seu programa e, para finalizar, possui uma vasta biblioteca com milhares de tarefas prontas.

2.4. Ok, Mas QUAL Java?

O Java passou por muitas evoluções ao longo dos anos e a nomenclatura não foi feita de uma maneira organizada. Por esta razão, os programadores novatos normalmente se confundem com a mesma. Vejamos quais foram as versões:

Nome Popular	Versão	Mudanças
Java	1.0/1.1	Versão original
Java2	1.2, 1.3 e 1.4	Muitos novos recursos
Java 5	1.5	Muitos novos recursos
Java 6	1.6	Melhorias de performance

Existem também algumas siglas usadas, que são importantes:

- JVM: Java Virtual Machine
- JRE: Java Runtime Machine (A JVM e bibliotecas básicas para execução)
- JDK: Java Development Kit (JRE + tudo que é necessário para o desenvolvimento)

3. PRIMEIRO PROGRAMA EM JAVA

Usando o NotePad++ ou a interface do NetBeans, digite o seguinte programa e grave com o nome EXATAMENTE como indicado:

UmPrograma.java

```
class UmPrograma {
   public static void main(String[] args) {
      System.out.println("Ola, mundo!");
      }
   }
```

Se digitou o programa, aperte o botão de compilação do NetBeans ou, se usou o NotePad++, digite, no prompt, o seguinte comando:

javac UmPrograma.java

Isso irá gerar um arquivo chamado **UmPrograma.class**, que é o seu programa já na pronto para ser executado pelo Java!

Execute-o com o seguinte comando:

java UmPrograma

O resultado será:

```
Olá, mundo!
```

Antes de continuarmos, vamos entender um pouco o que aconteceu. Vamos entender o programa linha por linha.

Inicialmente, encontramos essa linha:

UmPrograma.java

```
class UmPrograma {
   public static void main(String[] args) {
      System.out.println("Ola, mundo!");
      }
   }
```

Antes de mais nada, é preciso entender que todo programa Java está organizado em blocos chamados Classes, que esturademos melhor no futuro. Cada classe principal deve ter seu próprio arquivo, que deve ter o nome EXATAMENTE igual ao da classe (incluindo maiúsculas e minúsculas, não use caracteres especiais e acentos). Convenciona-se que todas as palavras que compõem um nome de classe sempre começam com uma letra maiúscula e as outras são minúsculas. Daí o nome da nossa classe: UmPrograma.

Como uma classe é um bloco, é necessário usar indicadores para marcar onde ela começa e onde ela acaba. O java usa as chaves como marcadores, usando o abre chaves { para indicar o início de um bloco e o fecha chaves } para indicar o fim de um bloco.

E tudo que estiver entre o abre e o fecha chaves será colocado pelo Java dentro do bloco "UmPrograma". Dentro deste bloco, a primeira linha é:

UmPrograma.java

```
class UmPrograma {
   public static void main(String[] args) {
      System.out.println("Ola, mundo!");
      }
   }
```

Esta linha indica que um bloco de tarefa será definido (note o abre chaves). Um bloco de tarefas é chamado de **método**.

Uma classe pode conter vários métodos e, em geral, possui pelo menos um. Sendo assim, é útil dar um **nome** a cada método, de maneira que, quando precisarmos cumprir aquela tarefa, possamos pedir para o Java executá-lo simplesmente mencionando o nome.

O método representado no programa chama-se "main" (principal) e é um nome especial de método: sempre que uma classe tiver um método main ele será chamado automaticamente quando executarmos o arquivo da classe. As palavras antes de "main" são informações sobre a função, que estudaremos no futuro.

Um nome de método sempre vem seguido de parênteses ( ) e os valores dentro destes parênteses também possuem um significado, que veremos mais adiante no curso.

Em nosso programa, dentro do método main, temos apenas uma instrução:

UmPrograma.java

```
class UmPrograma {
   public static void main(String[] args) {
      System.out.println("Ola, mundo!");
      }
   }
```

Agora ficará claro a importância de darmos nome para as classes e funções:

System.out

É o nome de uma classe que agrega métodos de impressão na tela.

println

É o nome de um método da classe System.out que **imprime uma linha** na tela. Assim, quando indicamos:

System.out.println("...");

Significa que estamos pedindo que o Java execute o método **println** da classe **System.out**, imprimindo o texto que estiver indicado dentro dos parênteses. A razão para precisar indicar o nome do método e o nome da classe é que outras classes podem ter métodos com o mesmo nome **println**, como, por exemplo, uma classe que imprima na impressora.

É importante observar que após indicar o método existe o sinal de ; (ponto-e-vírgula). Isso é importante porque isso indica que a **instrução** que estamos dando **acabou**. Ou seja: a instrução pode ser executada pelo Java.

3.1. Mas o Java Entende Isso que Eu Escrevi?

Sim, entende tudo que escrevemos, como a execução do código mostrou. Entretanto, a JVM não entende a **língua** em que escrevemos. Para isso, é necessário "traduzir" ou, em termos técnicos, "compilar" esse texto, para a língua que a JVM realmente entende, chamada **bytecode**. Isso foi feito com o programa JavaC (Java Compiler), usado no item anterior. As extensões do arquivo nos indicam qual a língua que está sendo usada dentro do arquivo:

.java Programa legível por seres humanos, como o escrevemos
.class Programa legível pela JVM, em bytecode

4. VARIÁVEIS EM JAVA

O programa que escrevemos anteriormente é bastante didático porque permite apresentar algumas das principais partes de um programa em Java. Entretanto, o programa que escrevemos não é muito útil, ele apenas imprime uma frase que nós mesmo escrevemos.

Para tornar um programa mais útil, é interessante usarmos as **variáveis**. Variáveis são locais na memória do computador em que guardamos dados para que possamos realizar operações com eles. Como é muito complicado lidar diretamente com posições de memória de um computador, deixaremos que o Java faça isso por nós. É como se tivéssemos um empregado responsável por cuidar de nossas roupas, pegá-las e guardá-las no armário sempre que precisamos.

Para que o Java possa gerenciar nossas variáveis, ou seja, gerenciar o uso da memória do computador, precisamos informá-lo que **tipo de dado** queremos guardar na memória e, mais importante, dar um **nome** para este lugar da memória já que, sem um nome, não teríamos como recuperar aquele valor no futuro. Essa informação é feita através de uma **declaração de variável**. O formato geral é o seguinte:

```
tipoDeVariavel nomeDaVariavel;
```

Uma variável pode ser deeclarada em qualquer lugar em um programa, mas ela só tem validade dentro do bloco em que foi declarada; em outras palavras, fora daquele bloco em que ela foi criada, é como se ela não existisse.

Há varios tipos de variáveis possíveis, como veremos mais adiante. O tipo mais comum, e o tipo "número inteiro", que é indicado para o Java como **int**. Assim, se quisermos pedir para o Java arrumar um espaço para guardarmos uma idade, que é um número inteiro, podemos indicar da seguinte forma:

OutroPrograma.java

```
class OutroPrograma {
   public static void main(String[] args) {
      int idade;
      }
   }
```

Uma variável, entretanto, só é útil quando colocamos algum valor dentro dela. Isso pode ser feito facilmente usando o sinal de igualdade, em uma instrução como a indicada abaixo:

OutroPrograma.java

```
class OutroPrograma {
   public static void main(String[] args) {
      int idade;

      idade = 18;
      }
   }
```

Observe que só podemos guardar informações dentro de uma variável **depois** que ela for declarada; se não for respeitada esta ordem, ocorrerá um erro ao compilar ou executar o código.

Podemos imprimir o valor de uma variável, como é indicado a seguir:

OutroPrograma.java

```
class OutroPrograma {
   public static void main(String[] args) {
      int idade;

      idade = 18;
      System.out.println( idade );
      }
   }
```

É possível criar outras variáveis e colocar o valor de uma delas na outra, como indica o código seguinte:

OutroPrograma.java

```
class OutroPrograma {
   public static void main(String[] args) {
      int idade;
      int outraIdade;

      idade = 18;
      outraIdade = 20;

      idade = outraIdade;
      System.out.println( idade );
      }
   }
```

O resultado agora será diferente, pois o valor de idade não é mais 18 quando o seu valor é impresso: ele recebeu o valor que estava na variável outraidade, que era 20.

Mas isso continua não sendo muito útil. Para tornar isso realmente útil, precisaremos realizar **operações** com estes números!

5. OPERAÇÕES EM JAVA

As operações básicas em Java são feitas com os símbolos clássicos:

+	soma
-	subtração
*	multiplicação
/	divisão
%	resto de divisão

Um exemplo de uso segue abaixo:

Operacoes.java

```
class Operacoes {
   public static void main(String[] args) {
      int idade;
      int idadeMaisUm;

      idade = 18;
      idadeMaisUm = idade + 1;
      System.out.println( idade );
      System.out.println( idadeMaisUm );
      }
   }
```

Qualquer cálculo pode ser feito. Por exemplo, para saber aproximadamente o número de semanas que uma pessoa viveu, podemos multiplicar a idade dela em anos pelo número de semanas de um ano, que é 52:

Operacoes.java

```
class Operacoes {
   public static void main(String[] args) {
      int idade;
      int idadeEmSemanas;

      idade = 18;
      idadeEmSemanas = idade * 52;
      System.out.println( idadeEmSemanas );
      }
   }
```

Substitua o valor "18" pela sua idade e descubra, aproximadamente, quantas semanas você viveu do dia em que nasceu até o seu último aniversário.

Um cáclulo mais complexo pode ser feito, combinando várias operações, por exemplo:

NovasOperacoes.java

```
class NovasOperacoes {
   public static void main(String[] args) {
      int x;
      int y;

      y = 12;
      x = 2*y + 37;

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

6. TIPOS DE VARIÁVEIS EM JAVA

Como foi dito anteriormente, existem vários tipos básicos de variáveis em Java. Estes tipos estão indicados a seguir, com a faixa de valores que podem ser armazenados nessas variáveis e também com o uso comum.

Tipo	Tamanho	Uso
boolean	1 bit	Valores do tipo "falso" ou "verdadeiro"
byte	1 byte	Valores inteiros de -127 a 128
short	2 bytes	Valores inteiros de -32767 a 32768
char	2 bytes	Códigos de caracteres
int	4 bytes	Valores inteiros de aprox. -2.000.000 a +2.000.000
float	4 bytes	Valores de ponto flutuante de simples precisão
long	8 bytes	Valores inteiros muito grandes
double	8 bytes	Valores de ponto flutuante de dupla precisão

O programa abaixo mostra como usar uma variável float para multiplicar um valor inteiro por 1.5. Observe, porém, que o resultado desta operação será um número de ponto flutuante, isto é, com casas decimais e, portanto, ele precisa ser armazenado em uma variável de ponto flutuante.

OperacoesTipos.java

```
class OperacoesTipos {
   public static void main(String[] args) {
      int idade;
      double indice;
      double resultado;

      idade = 18;
      indice = 1.5;
      resultado = 18 * indice;
      System.out.println( resultado );
      }
   }
```

6.1. Casting (opcional)

Caso tentemos armazenar um valor double ou float em uma variável inteira, seja ela de qual tipo for, pode ocorrer "perda de precisão", isto é, os valores depois da vírgula serão perdidos. Assim, se multiplicarmos 1.75 por 10 e armazenarmos o resultado em uma variável do tipo inteira, o valor armazenado será 17, e não 17,5.

Entretanto, se simplesmente escrevermos isso:

int x
x = 10 * 1.75;

O Java irá reclamar, dizendo que isso está errado. Se quisermos fazer isso, precisamos usar um recurso de "casting". Casting significa dizer para o Java "converte do jeito que eu estou mandando, eu sei que vai ter perda de precisão e você não precisa reclamar". O casting é feito com o nome do tipo "destino" entre parênteses, como indicado no código a seguir:

Casting.java

```
class Casting {
   public static void main(String[] args) {
      int resultado;

      resultado = (int) 1.75 * 10;
      System.out.println( resultado );
      }
   }
```

A indicação (int) feita força o Java a converter o resultado para int **antes** de colocar o resultado na variável resultado.

É comum termos de usar casting de uma outra maneira, quando vamos criar um número do tipo float. Por exemplo:

CastingFloat.java

```
class CastingFloat {
   public static void main(String[] args) {
      float umaVariavel;

      umaVariavel = 0.5;
      System.out.println( umaVariavel );
      }
   }
```

Se tentar compilar esse código, você receberá um erro na linha:

```
      umaVariavel = 0.5;
```

Mas o que há de errado nesta linha? Simples: por padrão, o java considera que o valor numérico 0.5 digitado é do tipo **double**, e ao colocar um valor double em uma variável float, há perda de precisão.

Para contornar esse erro sem ter que usar o casting **(float)** em um caso tão simples, existe uma forma alternativa de indicar o número 0.5, acrescentando um **f** na frente, indicando se tratar de um número float, não double, como indicado no código a seguir.

CastingFloat.java

```
class CastingFloat {
   public static void main(String[] args) {
      float umaVariavel;

      umaVariavel = 0.5f;
      System.out.println( umaVariavel );
      }
   }
```

Este código irá compilar normalmente. Esta mesma regra vale para variáveis do tipo **long**, que precisarão ser associadas com um número seguido da letra **l**, como indicado a seguir.

CastingLong.java

```
class CastingFloat {
   public static void main(String[] args) {
      long umaVariavel;

      umaVariavel = 1000000000l;
      System.out.println( umaVariavel );
      }
   }
```

7. EXERCÍCIOS

Observe o programa abaixo:

UmPrograma.java

```
class UmPrograma {
   public static void main(String[] args) {
      System.out.println("Ola, mundo!");
      }
   }
```

1) Altere o programa para imprimir uma mensagem diferente

2) Altere o programa para imprimir DUAS mensagens

3) Sabendo que o código

\n

serve para "quebrar linha", escreve as mesmas duas linhas do item 2 com um único System.out.println().

Você saberia dizer se a forma do item 3 é melhor ou pior do que a do item 2? E saberia dizer o porquê?

4) Na empresa onde trabalhamos, há tabelas com o quanto foi gasto em cada mês. Para fechar o balanço do primeiro trimestre, precisamos somar o gasto total. Sabendo que, em Janeiro, foram gastos 15000 reais, em Fevereiro, 23000 reais e em Março, 17000 reais, faça um programa que calcule e imprima o gasto total no trimestre. Siga esses passos:

a) Crie uma classe chamada BalancoTrimestral com um bloco main, como nos exemplos anteriores;

b) Dentro do main (o miolo do programa), declare uma variável inteira chamada gastosJaneiro e inicialize-a com 15000;

c) Crie também as variáveis gastosFevereiro e gastosMarco, inicializando-as com 23000 e 17000, respectivamente, utilize uma linha para cada declaração;

d) Crie uma variável chamada gastosTrimestre e inicialize-a com a soma das outras 3 variáveis:

```
int gastosTrimestre = gastosJaneiro + gastosFevereiro + gastosMarco;
```

e) Imprima a variável gastosTrimestre.

Extras

4) Se conhecer alguém que está trabalhando em um projeto Java, pergunte a ele o que ele pensa sobre a linguagem e por que o Java foi escolhido para este projeto.

8. BIBLIOGRAFIA

CAELUM, FJ-11: Java e Orientação a Objetos. Acessado em: 10/01/2010. Disponível em: < http://www.caelum.com.br/curso/fj-11-java-orientacao-objetos/ >

DEITEL, H.M; DEITEL, P.J. **Java: como programar** - Sexta edição. São Paulo: Pearson-Prentice Hall, 2005.

HOFF, A; SHAIO, S; STARBUCK, O. **Ligado em Java**. São Paulo: Makron Books, 1996.

Unidade 2: Estruturas de Controle

Parte 1 - Scanner e Estrutura IF

Prof. Daniel Caetano

Objetivo: Apresentar a classe Scanner e as principais estruturas de controle da linguagem Java.

Bibliografia: DEITEL, 2005; CAELUM, 2010; HOFF, 1996.

INTRODUÇÃO

Na primeira unidade a Linugagem Java foi apresentada, juntamente com os conceitos de variáveis e operadores. Estes são conceitos básicos e muito úteis, mas de pouca aplicabilidade se não formos capazes de receber dados digitados pelo usuário.

Os dados digitados pelo usuário podem precisar de algumas verificações antes de serem utilizados e, neste caso, entram em cena as estruturas de controle, que nos permitirão escolher qual parte de um programa será executada de acordo com o valor digitado pelo usuário.

Estes serão os tópicos apresentados nesta aula.

1. LENDO DADOS DIGITADOS PELO USUÁRIO

Na aula anterior, vimos um programa muito simples, reproduzido abaixo, que simplesmente imprimia uma mensagem na tela.

UmPrograma.java

```
class UmPrograma {
   public static void main(String[] args) {
      System.out.println("Ola, mundo!");
      }
   }
```

Vimos que **System.out** era uma das classes do Java, e que **println** é um de seus métodos, que serve para imprimir um texto na tela.

Nesta primeira parte da aula, vamos aprender como receber dados digitados pelo usuário, o que exigirá alguma preparação de nosso código, sendo a primeira delas a necessidade de existir uma variável para receber o valor digitado, como é indicado no código a seguir.

UmPrograma.java

```
class UmPrograma {
   public static void main(String[] args) {
      int valor;

      System.out.println("Ola, mundo!");
      }
   }
```

Infelizmente, a leitura de valores digitados pelo usuário é um processo um pouco mais complexo, não bastando usar a classe **System.in**, embora ela faça parte do processo, já que **System.in** é uma classe que apresenta métodos relativos ao teclado.

Tanto a classe **System.out** quanto a classe **System.in** são classes do pacote básico e são prontas para uso (o Java tem várias delas). No caso da leitura de teclado, não temos uma classe pronta para isso; temos apenas uma classe que serve, genericamente, para a leitura de dispositivos, a classe **Scanner**.

Como a classe Scanner não faz parte do pacote principal, faz parte do pacote ***java.util***, precisamos indicar ao Java que iremos utilizar esse pacote, o que pode ser feito através da instrução **import**, como indicado no código a seguir:

UmPrograma.java

```
import java.util.*;

class UmPrograma {
   public static void main(String[] args) {
      int valor;

      System.out.println("Ola, mundo!");
      }
   }
```

Se não fizermos isso e tentarmos usar a classe Scanner, o Java reportará um erro e não teremos um programa funcional.

> Para saber em qual pacote uma classe está e também quais são seus métodos, basta consultar o site de referência da Sun Microsystems: http://java.sun.com/j2se/1.5.0/docs/api/

Agora já podemos usar a classe Scanner. Entretanto, como a classe Scanner não é específica para ler o teclado, precisaremos solicitar que o Java crie uma **instância** desta classe, adaptada para a leitura do teclado. Essa **instância** é chamada **objeto**.

Mais adiante no curso estudaremos com maior detalhe o que são classes e objetos; por hora, a definição dada é considerada suficiente.

Para gerar um objeto, precisamos primeiro definir um **nome** para ele. Isso pode ser feito da mesma maneira com que se define uma variável:

```
NomeDaClasse   nomeDoObjeto;
```

Por exemplo, podemos criar um objeto a partir da classe **Scanner** e chamá-lo de **teclado**.

```
Scanner   teclado;
```

No código isso fica da seguinte forma:

UmPrograma.java

```
import java.util.*;

class UmPrograma {
   public static void main(String[] args) {
      int valor;
      Scanner teclado;

      System.out.println("Ola, mundo!");
      }
   }
```

A questão é que, ao contrário do que ocorre com variáveis, o Java não separa espaço de memória para objetos de maneira automática. Assim, depois que demos um nome ao objeto, precisamos pedir que o Java **aloque a memória** e **construa** este objeto para nós. Isso é feito através da instrução **new** (novo). A instrução **new** tem a seguinte "cara":

```
variavelObjeto = new NomeDaClasse(parâmetros);
```

Os parâmetros, no caso, definem as configurações específicas que queremos para o objeto criado. No nosso caso, por exemplo, queremos um objeto da **classe Scanner**, associado ao teclado, que é representado pela classe **System.in**. Isso pode ser indicado da seguinte forma:

```
teclado = new Scanner(System.in);
```

No código isso fica da seguinte forma:

UmPrograma.java

```
import java.util.*;

class UmPrograma {
   public static void main(String[] args) {
      int valor;
      Scanner teclado;

      teclado = new Scanner(System.in);

      System.out.println("Ola, mundo!");
      }
   }
```

Para simplificar o código, podemos colocar a definição do nome da variável do objeto e a sua criação na mesma linha, como indicado abaixo.

UmPrograma.java

```
import java.util.*;

class UmPrograma {
   public static void main(String[] args) {
      int valor;
      Scanner teclado = new Scanner(System.in);

      System.out.println("Ola, mundo!");
      }
   }
```

Tudo isso para criarmos uma **instância** da classe Scanner para leitura de teclado, ou seja, um objeto, ao qual demos o nome de **teclado**. Observe como o nome dado à variável tem relação direta com sua função.

Agora que o objeto para ler o teclado está pronto, precisamos pedir a ele que recolha informações a serem digitadas pelo usuário. Neste primeiro exemplo, vamos pedir que o usuário digite um número inteiro. Isso pode ser feito com o método **nextInt()** da classe Scanner.

> Sempre que criamos um objeto específico a partir de uma classe, este objeto também contera os métodos daquela classe.

Assim, usaremos o método **nextInt()** do objeto **teclado** para ler um número inteiro digitado pelo usuário, e o resultado será colocado na variável **valor**, o que pode ser feito da seguinte forma:

```
valor = teclado.nextInt();
```

No nosso código, isso fica da seguinte forma:

UmPrograma.java

```
import java.util.*;

class UmPrograma {
   public static void main(String[] args) {
      int valor;
      Scanner teclado = new Scanner(System.in);

      valor = teclado.nextInt();

      System.out.println("Ola, mundo!");
      }
   }
```

Para finalizar, vamos alterar o código do System.out para que ele imprima o número digitado, ao invés do texto "Ola, mundo!":

UmPrograma.java

```
import java.util.*;

class UmPrograma {
   public static void main(String[] args) {
      int valor;
      Scanner teclado = new Scanner(System.in);

      valor = teclado.nextInt();
      System.out.println(valor);
      }
   }
```

Compilando e executando este programa, temos uma surpresa: a tela fica preta, sem pedir nada... se digitarmos um número, o programa re-imprime o número; porém, o programa não pede nada. Precisamos pedir explicitamente, usando o método System.out.println, como indicado abaixo:

UmPrograma.java

```
import java.util.*;

class UmPrograma {
   public static void main(String[] args) {
      int valor;
      Scanner teclado = new Scanner(System.in);
```

```
      System.out.println("Digite um número inteiro");
      valor = teclado.nextInt();
      System.out.println(valor);
      }
   }
```

E também podemos aproveitar e melhorar a mensagem de resposta:

UmPrograma.java

```
import java.util.*;

class UmPrograma {
   public static void main(String[] args) {
      int valor;
      Scanner teclado = new Scanner(System.in);


      System.out.println("Digite um número inteiro");
      valor = teclado.nextInt();
      System.out.println("Valor digitado: " + valor);
      }
   }
```

Observe que basta usar a operação "+" para acrescentar o conteúdo da variável **valor** ao texto que será impresso!

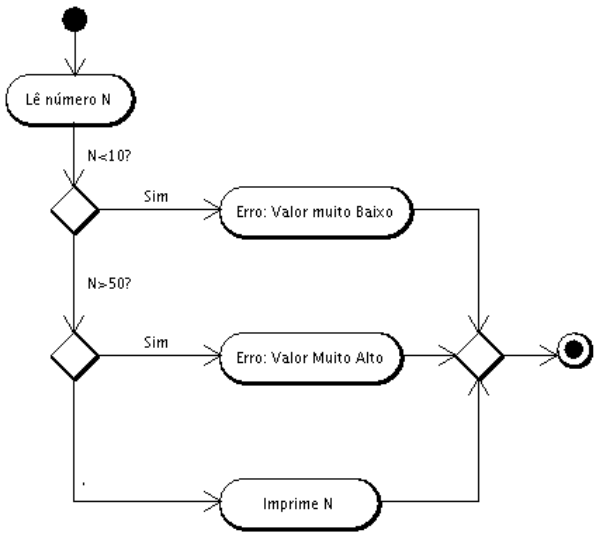
Com isso, já somos capazes de coletar dados digitados. Vejamos agora como tomar decisões com base nas informações digitadas pelo usuário!

2. ESTRUTURAS DE CONTROLE

Estruturas de Controle são instruções da linguagem de programação que permitem decidir, através dos valores das variáveis, qual trecho de código será executado. Por exemplo, imaginemos o seguinte programa:

Faça um programa que leia um número de 10 a 50 (inclusive estes), imprimindo o valor digitado. Caso o número digitado seja muito pequeno ou muito grande, uma mensagem de erro específica deve ser impressa: "Valor muito pequeno" ou "Valor muito alto", respectivamente.

Para simplificar nossa tarefa de escrever o programa, vamos trabalhar com o fluxograma que representa o processo, que é indicado a seguir.



Para programar esse algoritmo, iremos começar com um programa vazio.

EstruturaIf.java

```
import java.util.*;

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

      }
   }
```

Seguindo o diagrama, a primeira coisa a ser feita é ler o número "N". Vamos então criar um objeto da classe Scanner para a leitura, chamado teclado, e uma variável do tipo int chamada N, para podermos realizar esta tarefa:

EstruturaIf.java

```
import java.util.*;

class EstruturaIf {
   public static void main(String[] args) {
      int N;
      Scanner teclado = new Scanner(System.in);
      System.out.println ("Digite um número inteiro");
      N = teclado.nextInt();
      }
   }
```

Vamos colocar um pequeno comentário para nos ajudar a entender cada bloco, conforme o nome das elipses do diagrama.

EstruturaIf.java

```
import java.util.*;

class EstruturaIf {
   public static void main(String[] args) {
      int N;
      Scanner teclado = new Scanner(System.in);

      // Lê número N
      System.out.println ("Digite um número inteiro");
      N = teclado.nextInt();
      }
   }
```

Seguindo o diagrama, temos uma decisão a ser feita: se N for menor que 10, imprimimos um erro... caso contrário, continuamos seguindo a outra setinha. Isso pode ser feito com a estrutura IF, que funciona assim:

```
if (comparação) {
   // código a executar se a comparação for verdadeira
   }
else {
   // código a executar se a comparação for falsa
   }
```

No nosso caso, isso pode ser adaptado assim:

EstruturaIf.java

```
import java.util.*;

class EstruturaIf {
   public static void main(String[] args) {
      int N;
      Scanner teclado = new Scanner(System.in);

      // Lê número N
      System.out.println ("Digite um número inteiro");
      N = teclado.nextInt();

      // Primeira decisão: verifica se número é baixo
      if ( N<10 ) {
         // Imprime Erro: Valor muito baixo
         }
      else {
         // Executa se valor não for muito baixo
         }
      }
   }
```

Ora, imprimir o erro é fácil, basta usar um System.out.println:

```
import java.util.*;

class EstruturaIf {
   public static void main(String[] args) {
      int N;
      Scanner teclado = new Scanner(System.in);

      // Lê número N
      System.out.println ("Digite um número inteiro");
      N = teclado.nextInt();

      // Primeira decisão: verifica se número é baixo
      if ( N<10 ) {
         // Imprime Erro: Valor muito baixo
         System.out.println("Valor muito baixo!");
         }
      else {
         // Executa se valor não for muito baixo
         }
      }
   }
```

Agora escrevemos o código da outra "setinha", que toma a segunda decisão:

```
import java.util.*;

class EstruturaIf {
   public static void main(String[] args) {
      int N;
      Scanner teclado = new Scanner(System.in);

      // Lê número N
      System.out.println ("Digite um número inteiro");
      N = teclado.nextInt();

      // Primeira decisão: verifica se número é baixo
      if ( N<10 ) {
         // Imprime Erro: Valor muito baixo
         System.out.println("Valor muito baixo!");
         }
      else {
         // Segunda decisão: verifica se número é alto
         if ( N>50 ) {
            // Imprime Erro: Valor muito baixo
            System.out.println("Valor muito alto!");
            }
         else {
            // Executa se o valor estiver na faixa esperada!
            }
        }
     }
  }
```

Agora só falta imprimir o valor digitado, no segundo **else**, ou seja, quando as duas verificações foram satisfeitas:

EstruturaIf.java

```
import java.util.*;

class EstruturaIf {
   public static void main(String[] args) {
      int N;
      Scanner teclado = new Scanner(System.in);

      // Lê número N
      System.out.println ("Digite um número inteiro");
      N = teclado.nextInt();

      // Primeira decisão: verifica se número é baixo
      if ( N<10 ) {
         // Imprime Erro: Valor muito baixo
         System.out.println("Valor muito baixo!");
         }
      else {
         // Segunda decisão: verifica se número é alto
         if ( N>50 ) {
            // Imprime Erro: Valor muito baixo
            System.out.println("Valor muito alto!");
            }
         else {
            // Executa se o valor estiver na faixa esperada!
            System.out.println ("Valor digitado: " + N);
            }
         }
      }
   }
```

Isso finaliza o nosso programa!

Na próxima aula veremos como usar estruturas de **repetição**!

3. EXERCÍCIO

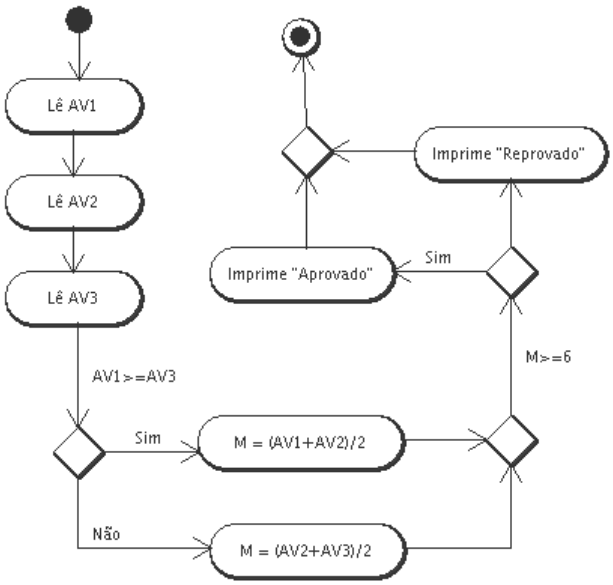
1. Implemente um código que calcule a média do aluno, seguindo as novas regras da Estácio Radial, e indique se o aluno está aprovado ou reprovado. Regra:

Se AV1 >= AV3 ==> M = (AV1 + AV2) / 2
Se AV1 < AV3 ==> M = (AV3 + AV2) / 2

M >= 6 ==> Aprovado M < 6 ==> Reprovado

IGNORE a regra de que as notas precisam ser maiores ou iguais a 4,0!

O diagrama está indicado a seguir:



2. Tente acrescentar ao diagrama as verificações que seriam necessárias para considerar que as notas escolhidas não podem ser menores que 4,0.

3. Implemente no código, em Java, essas verificações.

4. EXTRA: Tente desenhar o diagrama e implementar o algoritmo para o antigo sistema de notas da UniRadial:

M = (AD1 + AD2 + AC + PI) / 4

Sendo que M >= 7,0 para aprovação, M < 2,0 para reprovação e se M >=2,0 e M < 7,0, o programa deve perguntar a nota R, e, nesta segunda etapa:

MR = (M + R) / 2

Sendo que MR >= 6,0 para aprovação e MR < 6,0 para reprovação.

4. BIBLIOGRAFIA

CAELUM, FJ-11: Java e Orientação a Objetos. Acessado em: 10/01/2010. Disponível em: < http://www.caelum.com.br/curso/fj-11-java-orientacao-objetos/ >
DEITEL, H.M; DEITEL, P.J. **Java: como programar** - Sexta edição. São Paulo: Pearson-Prentice Hall, 2005.
HOFF, A; SHAIO, S; STARBUCK, O. **Ligado em Java**. São Paulo: Makron Books, 1996.

Unidade 3: Estruturas de Controle

Parte 2 - Lógica, SWITCH, FOR, WHILE e DO WHILE

Prof. Daniel Caetano

Objetivo: Apresentar a composição lógica em estruturas de decisão e as estruturas FOR, WHILE e DO WHILE.

Bibliografia: DEITEL, 2005; CAELUM, 2010; HOFF, 1996.

INTRODUÇÃO

Nas aulas anteriores foram observadas as estruturas básicas de um programa, incluindo a estrutura de decisão com a instrução IF. Esta instrução foi apresentada com apenas uma comparação, sua forma mais comum. Nesta aula será apresentada a sintaxe que permite realizar diversas comparações com apenas uma estrutura IF. Para casos mais complexos, será apresentada a estrutura SWITCH.

Adicionalmente, serão apresentadas as estruturas de loop, isto é, de repetição, isto é, as estruturas FOR, WHILE e DO WHILE.

1. COMPARAÇÕES MÚLTIPLAS COM IF

Já vimos como fazer seleção de código com comparações múltiplas **aninhadas**, isto é, uma colocada dentro da outra. Para este efeito, foi usada a estrutura do tipo **if ~ else**. Resumidamente, a estrutura if ~ else serve para situações em que uma decisão simples é necessária, ou seja: existem duas possibilidades e nunca ambas ocorrem ao mesmo tempo: se um trecho de código executar, o outro não será executado.

Por exemplo: um aluno é aprovado se tiver média maior ou igual a 50. Então, a verificação é: "Se nota é maior ou igual a 50, aluno aprovado. Caso contrário, aluno reprovado". Em Java isso poderia ser expresso da seguinte forma:

```
if (notaDeProva >= 50) {   // se nota de prova maior ou igual a 50...
      aprovado = 1;        // aluno aprovado
      }
else  {                    // caso contrário...
      aprovado = 0;        // aluno não aprovado
      }
```

Como vimos, as estruturas **if~else** podem ser aninhadas, ou seja, podemos ter ifs dentro de ifs. Por exemplo: se o aluno passasse APENAS se tiver nota de prova maior que 50 e frequencia maior que 75, a estrutura ficaria assim:

```
if (notaDeProva < 50) {          // se nota de prova menor que 50...
      aprovado = 0;              // aluno reprovado
      }
else{                            // caso contrário... (nota de prova >= 50)
      if (frequencia >= 75) {    // se a frequencia maior ou igual a 75...
            aprovado = 1;        // aluno aprovado
            }
      else {                     // caso contrario (prova>=50 e freq<75)
            aprovado = 0;        // aluno não aprovado
            }
      }
```

Um outro jeito de conseguir o mesmo efeito é com associação lógica de comparações, usando as operações lógicas clássicas E (&&), OU (||) e NÃO (!) dentro do IF. Por exemplo, o código anterior pode ser simplificado assim:

```
// se nota de prova menor que 50 ou frequencia menor que 75...
if (notaDeProva < 50 || frequencia < 75 ) {
      aprovado = 0;              // aluno reprovado
      }
else{                            // caso contrário...
      aprovado = 1;       // aluno aprovado
      }
```

Embora um pouco mais complexa à primeira vista, esta notação facilita a leitura, reduz o código e pode tornar o programa mais eficiente. Lembrando a tabela verdade das operações lógicas:

A	Operação	B	Resultado
FALSO	*OU (||)*	FALSO	FALSO
FALSO	*OU (||)*	VERDADEIRO	VERDADEIRO
VERDADEIRO	*OU (||)*	FALSO	VERDADEIRO
VERDADEIRO	*OU (||)*	VERDADEIRO	VERDADEIRO
FALSO	*E (&&)*	FALSO	FALSO
FALSO	*E (&&)*	VERDADEIRO	FALSO
VERDADEIRO	*E (&&)*	FALSO	FALSO
VERDADEIRO	*E (&&)*	VERDADEIRO	VERDADEIRO
-	*NÃO (!)*	FALSO	VERDADEIRO
-	*NÃO (!)*	VERDADEIRO	FALSO

2. SELEÇÃO COM SWITCH~CASE

A estrutura do tipo switch~case existe para situações em que temos um número finito de possibilidades de tarefas a executar, dependendo de um único valor. Por exemplo, se um programa imprime o estado atual de um MP3 player e os estados possíveis são:

0- parado
1- tocando,
2- pausa
3- erro

Isso poderia ser feito com um switch na seguinte forma:

```
switch(estadoDoMP3) {
      case 0:
            System.out.println("MP3 parado");
            break;
      case 1:
            System.out.println("MP3 tocando");
            break;
      case 2:
            System.out.println("MP3 em pausa");
                  break;
      case 3:
            System.out.println("Erro na leitura do MP3");
            break;
      default:
            System.out.println("Erro desconhecido");
            break;
      }
```

Note que para cada valor de estado, um determinado "case" será executado. A instrução break faz com que a execução pule para a primeira linha após os delimitadores { } do switch. Caso não se use a instrução break, a execução continua com o caso seguinte, até encontrar um break.

Note também o caso especial "default". Embora nem sempre estritamente necessário, é uma boa prática de programação usá-lo, pois ajuda a diagnosticar problemas de lógica no software. No exemplo acima, qualquer estadoDoMP3 diferente de 0 a 3 causará a execução do caso "default". Como não é possível fazer a menor idéia do que isso seja (já que o valor do estadoDoMP3 só deveria ser de 0 a 3) este caso especial imprime uma mensagem dizendo que um erro desconhecido ocorreu.

3. INSTRUÇÕES DE REPETIÇÃO

O Java fornece uma número mais que suficiente de instruções de repetição, que podem ser **aninhadas**, isto é, uma colocada dentro da outra. Estas instruções são as de construção do tipo **for**, **while** e **do ~ while**. Via de regra é possível substituir uma delas pela outra, mas há casos em que é mais cômodo usar uma ou outra, em favor da legibilidade.

3.1. Estrutura FOR

A instrução for é usada quando é necessário realizar uma tarefa por um número determinado de vezes. Ela compreende 4 partes: uma inicialização, um teste de finalização, uma descrição de incremento e, finalmente, o trecho que deve ser repetido.

Por exemplo: se for necessário imprimir um determinado texto 3 vezes, podemos usar uma estrutura for da seguinte forma:

```
for (int i = 0; i < 3; i = i + 1) {
      System.out.printf ("Texto número %d \n", i);
      }
```

O Java lê este trecho de código como:

Repita o trecho de código entre {} respeitando as seguintes regras:

1) Faça i (o contador) valer 0
2) Verifique se i é menor que 3. Se sim, execute passo 3. Se não, termine o for.
3) Execute o trecho entre { }
4) Faça i = i + 1 (ou seja, some 1 ao contador)
5) Volta ao passo 2.

O resultado desta estrutura (freqüentemente chamada de *loop* ou *laço*) é:

Texto número 0
Texto número 1
Texto número 2

3.2. Estrutura WHILE

A instrução while é usada quando queremos que um dado conjunto de instruções seja executado até que uma dada situação ocorra. A instrução while compreende 2 partes: um teste de finalização e o trecho que deve ser repetido.

Por exemplo: se for necessário imprimir um determinado texto até que o usuário digite 0 como entrada, podemos usar uma estrutura while da seguinte forma:

```
Scanner input = new Scanner(System.in);
int dado = -1;

while (dado != 0) {
      System.out.println ("Digite o número zero!");
      dado = input.nextInt();
      }
```

O Java lê este trecho de código como "Repita o trecho de código entre {} respeitando as seguintes regras:"

1) Verifique se dado é diferente de 0. Se sim, execute passo 2. Se não, fim do while.
2) Execute o trecho entre { }
3) Volta ao passo 1.

Note que agora a inicialização da variável e a atualização da mesma, ao contrário do que normalmente acontece com o for, **não é** responsabilidade da estrutura while, e sim do código que está antes do *loop* e do código do *loop* respectivamente.

Note também que, se a variável **dado** for inicializada com o valor zero (ao invés de -1, como foi no exemplo), o trecho de código entre { } no while não será executado nenhuma vez, pois o teste é feito antes de qualquer coisa ser executado.

3.3. Estrutura DO~WHILE

A instrução do~while é usada quando é desejado que um dado conjunto de instruções seja executado até que uma dada situação ocorra, mas é necessário garantir que o conjunto de instruções seja executado **pelo menos uma vez**. A instrução do~while compreende 2 partes: um trecho que deve ser repetido e um teste de finalização.

Usando o mesmo exemplo da instrução while, se for necessário imprimir um determinado texto até que o usuário digite 0 como entrada, podemos usar uma estrutura do~while da seguinte forma:

```
Scanner input = new Scanner(System.in);

do {
      System.out.println ("Digite o número zero!");
      dado = input.nextInt();
      } while (dado != 0);
```

O Java lê este trecho de código como: "Repita o trecho de código entre {} respeitando as seguintes regras:"

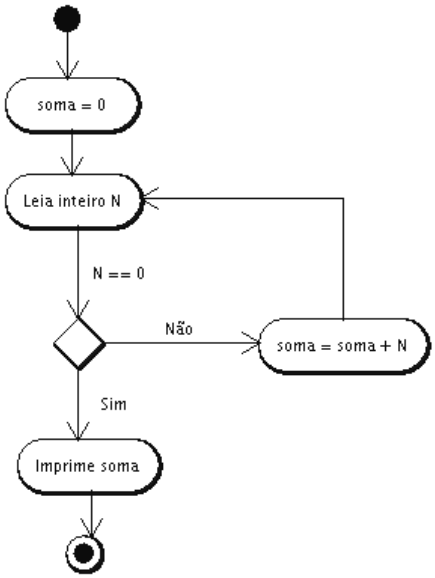
1) Execute o trecho entre { }
2) Verifique se dado é diferente de 0. Se sim, volte ao passo 1. Se não, termine o do-while.

Note que neste caso não foi necessário inicializar a variável, já que ela é lida dentro do *loop* antes de ser testada. De qualquer forma, assim como no caso do while, no do~while a inicialização da variável e a atualização da mesma, **não é** responsabilidade da estrutura do~while, e sim do código que está antes do *loop* e do código do *loop* respectivamente.

Note também que, independente do valor inicial da variável **dado**, o trecho de código entre { } no do~while será executado **pelo menos uma vez** já que o teste só é feito depois que o conteúdo do *loop* tiver sido executado uma vez.

4. IMPLEMENTANDO CÓDIGO

Vamos realizar um programa que leia números inteiros do teclado até que o número zero seja digitado. A cada número digitado, o resultado é armazenado na variável **soma**. O diagrama está a seguir:



While.java

```
import java.util.*;

class While {
   public static void main(String[] args) {
      int N;
      int soma;
      Scanner teclado = new Scanner(System.in);
      soma = 0;

      do {
         // Lê número N
         System.out.print("Digite um número inteiro ");
         System.out.print("ou 0 para finalizar: ");
         N = teclado.nextInt();
         if (N != 0) {
            soma = soma + N;
            }
         } while (N != 0);

      // Executa se o valor estiver na faixa esperada!
      System.out.println ("A soma final é: " + soma);
      }
   }
```

5. EXERCÍCIOS

A) Imprima todos os números de 100 a 220.
B) Imprima a soma dos números de 1 a 300.
C) Imprima todos os múltiplos de 7 entre 1 e 200.
D) Imprima os fatoriais dos números 1 a 10.

N! = N*(N-1)!
Ex: 1! = 1
2! = 2*1 = 2
3! = 3*2*1 = 6
4! = 4*3*2*1 = 24
...

E) Se quisermos fazer o fatorial de 1 a 40, o que acontece com a velocidade? E com os resultados? Altere o programa com variáveis do tipo **long** para corrigir os resultados.

6. BIBLIOGRAFIA

CAELUM, FJ-11: Java e Orientação a Objetos. Acessado em: 10/01/2010. Disponível em: < http://www.caelum.com.br/curso/fj-11-java-orientacao-objetos/ >

DEITEL, H.M; DEITEL, P.J. **Java: como programar** - Sexta edição. São Paulo: Pearson-Prentice Hall, 2005.

HOFF, A; SHAIO, S; STARBUCK, O. **Ligado em Java**. São Paulo: Makron Books, 1996.

Unidade 4: Métodos e Funções

Prof. Daniel Caetano

Objetivo: Introduzir o conceito de métodos e funções.

Bibliografia: DEITEL, 2005; CAELUM, 2010; HOFF, 1996.

INTRODUÇÃO

Nas aulas anteriores foram observadas as estruturas fundamentais de seleção e repetição da linguagem Java. Tais estruturas, juntamente com o as variáveis, permitem elaborar algoritmos para praticamente qualquer finalidade.

Entretanto, para que os softwares possam ser desenvolvidos de maneira organizada e seu código fique mais compreensível, é interessante organizar o código em *blocos funcionais*, isto é, blocos que realizam tarefas específicas.

Estes blocos são chamaos de **métodos** ou, mais genericamente, de **funções**. Um método é um bloco de instruções que executam uma tarefa específica. Cada método possui um nome específico, de maneira que sempre que se desejar executar aquela sequência de instruções, bastará usar o nome do método.

Nesta aula serão apresentados os conceitos envolvidos na criação de um método, bem como alguns exemplos.

1. O QUE É UM MÉTODO

Antes de mais nada, é importante ressaltar o **que é** e **para que serve** um método. Um método é um trecho de código que faz alguma tarefa bem definida e que, usualmente, pode ser reutilizada em diversas partes do software.

Os métodos só existem dentro das classes (o objetos, como será visto depois), não sendo possível criar um método sem uma classe. Assim, pode-se dizer que um método é a descrição de algo que uma classe (ou objeto) pode fazer.

E para que serve o método? Bem, o método serve para ser executado quando necssário, o que pode ser feito através de seu nome. Assim, é quase sempre obrigatório que um método tenha um nome.

2. DECLARANDO UM MÉTODO

Como já foi dito, os métodos são sempre definidos **dentro** de uma classe. Assim, uma declaração de método fora dos demarcadores de classe { } causam erro durante a compilação. A declaração de um método é, de maneira geral, feita da seguinte forma:

```
[escopo] <tipoDeRetorno> <nomeDoMetodo> ( [tipoDaVar <nomeDaVar>] [, ...] ) {
  // O código do método vai aqui
  }
```

Por exemplo:

```
public int multiplicaPorDois (int valor) {
  // ... operações executadas pelo método entram aqui
  }
```

O campo **tipoDeRetorno** indica o que esse método "responde" quando ele for executado, ou seja, que tipo de dado ele *retorna* a quem o executou. No caso do exemplo, o método "responde" um número inteiro (int). Pode ser qualquer tipo válido, um nome de classe ou até mesmo o tipo **void**, que indica "nenhum retorno".

O **nomeDoMetodo** deve ser um nome curto e, ao mesmo tempo, descritivo do que o método faz, qual a função que ele executa. O nome do exemplo não é muito bom: algo como multPor2 seria algo melhor, embora de leitura um pouco mais complexa.

O **tipoDaVar** dentro dos parênteses indica o tipo do dado **nomeDaVar**, que é um **parâmetro** do método, ou seja, um valor que o método vai receber para que ele possa fazer seu processamento e dar uma resposta.

Agora, vejamos o campo que ficou sem dicussão: o **escopo**. O escopo define quem pode executar esse método. Alguns dos valores possíveis só ficarão mais claros quando estivermos falando sobre orientação a objetos, mais adiante no curso.

Escopo	Significado
public	método pode ser chamado a partir de qualquer parte do programa
private	método só pode ser chamado por membros da classe
---	
final	método não pode ser modificado em classes especializadas
protected	método só pode ser chamado por membros da classe ou descendentes
static	método é da classe, e não do objeto.

Existem outras definições de escopo, mas essas são as mais comuns. Os usos dos escopos ficam claros pelo seu significado na tabela acima, a não ser pelo escopo **static**. A função desta definição de escopo é melhor detalhada abaixo.

Como já vimos rapidamente, uma "cópia" de uma classe, com características específicas, pode ser criada. Esta "cópia" ou "instância" é chamada **objeto**. Quando criamos uma instância de uma classe, tudo ocorre como se cada objeto passasse a ter o seu próprio conjunto de métodos.

Quando queremos que apenas uma "cópia" do método exista (podendo ser compartilhada por todos os objetos, inclusive), dizemos que o método é **da classe**. Para indicar que um método é **da classe** e não dos objetos, usa-se o escopo do tipo **static**.

Métodos do tipo **static** podem ser executados **sem** a criação e um objeto de uma dada classe. Métodos que não tenham sido declarados como static não podem ser executados sem a criação de pelo menos um objeto da classe.

2.1. Retornando um Valor

Quando o método precisar devolver algum valor para quem solicitou sua execução, deve ser usada a instrução **return** para isso, conforme o exempo abaixo:

```
public int multiplicaPorDois (int valor) {
   int resultado;
   // ... operações executadas entram aqui, calculando o valor em resutlado
   return resultado;
   }
```

2.2. Escopo de Variáveis

Até o momento tínhamos apenas uma classe com um método e, como sempre declarávamos nossas variáveis no nosso único método, não havia nenhum tipo de complicação.

A partir de agora, entretanto, temos mais de um método e é possível perceber um comportamento bastante peculiar: as variáveis de um método só são válidas dentro daquele método. Assim, não é possível que um método acesse diretamente variáveis de outro método. Tenha isso em mente quando estiver programando.

3. CHAMANDO UM MÉTODO

Os métodos normais, não static, são executados da seguinte forma:

nomeDoObjeto.nomeDoMetodo();

Os métodos static são executados da seguinte forma:

NomeDaClasse.nomeDoMetodo();

É importante ressaltar que apenas métodos públicos (**public**) de uma classe ficam disponíveis para outras classes. Em nossos programas estamos ainda trabalhando com apenas uma classe. Futuramente, quando trabalharmos com mais classes, este fato será importante.

A razão para tornar um método privativo (**private**) também será discutida no futuro.

4. IMPLEMENTANDO CÓDIGO

Vamos realizar um programa que imprima os fatoriais de 1 a 10, de maneira que exista um método para calcular o fatorial. Como estamos trabalhando apenas com a classe, este método deve ter escopo do tipo **public static**.

O código está apresentado a seguir:

MetodoFatorial.java

```
class MetodoFatorial {

   public static long fatorial(int numero) {
      long resultado;
      int i;
      resultado = 1;
      for (i = numero; i > 0; i = i - 1) {
         resultado = resultado * i;
         }
      return resultado;
      }

   public static void main(String[] args) {
      long resultado;
      int i;

      for (i = 1; i <= 10; i = i + 1) {
         resultado = fatorial(i);
         System.out.println( i + "! = " + resultado);
         }
      }
   }
```

5. EXERCÍCIOS

A) Crie um método que calcule a potência de um número. O método deve ter a seguinte apresentação:

public static int potencia(short base, short exp)

E a operação a ser realizada é:

$base^{exp}$

B) Use esse método para calcular as potências de 2^1 a 2^{16}

C) Crie um método que receba as notas AV1, AV2, AV3 e FREQ e responda 0 se o aluno está reprovado e 1 se está aprovado.

D) Use o método acima para testar a aprovação dos alunos abaixo:

	AV1	AV2	AV3	FREQ
Aluno 1:	6,0	6,5	0,0	80
Aluno 2:	5,0	5,0	7,0	75
Aluno 3:	3,0	10,0	3,5	90
Aluno 4:	10,0	3,0	10,0	95
Aluno 5:	10,0	10,0	10,0	70

O resultado apresentado pelo software deve ser:

Aluno 1: Aprovado
Aluno 2: Aprovado
Aluno 3: Reprovado
Aluno 4: Reprovado
Aluno 5: Reprovado

6. BIBLIOGRAFIA

CAELUM, FJ-11: Java e Orientação a Objetos. Acessado em: 10/01/2010. Disponível em: < http://www.caelum.com.br/curso/fj-11-java-orientacao-objetos/ >

DEITEL, H.M; DEITEL, P.J. **Java: como programar** - Sexta edição. São Paulo: Pearson-Prentice Hall, 2005.

HOFF, A; SHAIO, S; STARBUCK, O. **Ligado em Java**. São Paulo: Makron Books, 1996.

Unidade 5: Projeto de Programas

Noções de Concepção e Programação

Prof. Daniel Caetano

Objetivo: Apresentar e aplicar alguns conceitos da análise funcional.

Bibliografia: YOURDON, 1990.

INTRODUÇÃO

Nas aulas anteriores foram apresentados os conceitos fundamentais da linguagem Java, conceitos que, de maneira geral, podem ser aplicados a qualquer linguagem de programação estruturada.

Na última aula foram apresentados os conceitos de métodos e funções, que têm um papel muito relevante no desenvolvimento e organização de sistemas de todos os tipos de tamanhos. Entretanto, a divisão funcional das atividades de um sistema não é algo novo, remontando às origens das linguagens de programação.

Nesta aula reveremos brevemente alguns conceitos relativos à análise de fluxo de dados e à análise funcional, para então exercitar sua aplicação no desenvolvimento de um pequeno software.

1. BREVE HISTÓRICO DAS LINGUAGENS

Início: Linguagem de Máquina - Wireup e Digitação
Depois: Assembly: palavras que são traduzidas em códigos de máquina. Assemblada.
1954: FORTRAN: primeira linguagem de alto nível completa. Código Spaghetti. Compilada.
1960: ALGOL60: primeira linguagem de alto nível estruturada em blocos. Compilada.
1970: Pascal: popular linguagem de alto nível estruturada. Compilada.
1970: Smalltalk: primeira linguagem orientada a objetos de sucesso. Compilada.
1971: C: popular linguagem de alto nível estruturada. Compilada.
1975: BASIC: popular linguagem de alto nível "spaghetti". Interpretada.
1983: C++: popular linguagem orientada a objetos. Compilada
1986: Object Pascal (Mac) / Actor (Windows): linguagens orientadas a objetos já com libs voltadas ao desenvolvimento de GUIs. Compiladas.
1991: Visual BASIC: implementação voltada a GUI do velho BASIC. Interpretada/VBRUN
1991: Oak: embrião da linguagem Java. Interpretada.
1991: Python: Linguagem de programação multi-paradigma, interpretada.
1995: PHP: primeira linguagem real voltada ao desenvolvimento Web. Interpretada.
1995: Java: popular linguagem orientada a objetos multiplataforma. Interpretada.
1995: Ruby: Linguagem de programação multi-paradigma, interpretada.
2001: C#: derivada de C++ e Java, orientad a objetos, especifica da plataforma .Net.

A evolução das linguagens, em geral, acontece antes da evolução de paradigma de projeto; na verdade, o aparecimento de linguagens que seguem novas filosofias, em geral, promovem mudanças de paradigma de projeto.

Se, no início, imperava o "caos", este foi substituído pela análise estruturada e, recentemente, pela análise orientada a objetos. Recentemente surgiu um novo paradigma, a orientação a aspectos, mas este ainda está em fase embrionária.

2. VISÃO GERAL SOBRE ANÁLISE E PROJETO ESTRUTURADO

A análise e projeto nem sempre existiram na forma como conhecemos. Quanto mais voltarmos no passado, mais comum era a atitude de "sentar e programar". Entretanto, com o passar dos anos, essa atitude foi se tornando cada vez mais problemática, à medida em que os softwares foram crescendo.

O "sentar e programar" limitava a capacidade de previsão do desenvolvedor, dificultando a tarefa de criar um software que fosse adaptável a situações futuras. Da mesma forma, a manutenção se tornava difícil, uma vez que os softwares desenvolvidos normalmente não contavam com uma modularização coerente. O trabalho em equipe tornava-se um pesadelo.

Destas dificuldades, surgiu a questão: como implementar um sistema?

Objetivo: resolver com sucesso um problema do mundo real.
Passo 1: Compreensão do domínio do problema, cada um de seus detalhes, com definições mais precisas dos objetivos: esse processo foi chamado de Análise de Sistemas.
- Fundamental para atender as necessidades do usuário.
- Fundamental para prever necessidades futuras (expansibilidade, difícil prever)

Passo 2: Propor um modelo simplificado de implementação
Passo 3: Propor um modelo detalhado de implementação
Passo 4: Implementar
Passo 5: Testar
Passo 6: Implantar
Passo 7: Período de Manutenção

2.1. Análise de Sistemas "Clássica"

Linguagem de Máquina, Assembly, linguagens não estruturadas:
- Fluxogramas (Diagramas de Blocos, uma entrada, múltiplas saídas).

Programação Estruturada

- Duas vertentes básicas:
  => Fluxo de dados
  - Módulos definidos pelos dados que são processados.
  => Decomposição funcional
  - Módulos definidos pelas funções, agrupadas por finalidade

Pensemos num Sistema de Informações genérico. Deve permitir que o usuário processe informações (adição/consulta).

Um exemplo de fluxo de dados (DFD):

```
                 +-------------------------------------------
                 |  Sistema  +------------+
+---------+      |           |  Consulta  |<--------+
|         |<----------------+ Informação |         |
| Usuário |      |           |            |   +-----+----------
|         +----------+      +------------+   |DB|| Informações
+---------+      |     |     +------------+   +-----^----------
                 |     |     |  Atualiza  |         |
                 |     +---->|    Dados   +---------+
                 |           +------------+
                 +-------------------------------------------
```

Um exemplo de blocos funcionais:

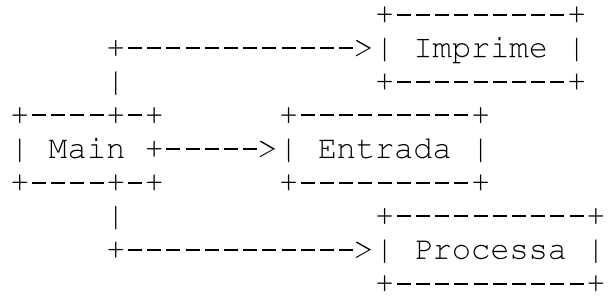
```
                      +---------+
         +------------>| Display |
         |             +---------+
    +----+----+         +-------+
    | Control +-------->| Input |
    +----+----+         +-------+
         |             +---------+
         +------------>| Process |
                       +---------+
```

Não havia consenso sobre qual das metodologias usar e o resultado era uma documentação pouco coerente, causando sérios problemas em projetos grandes, com muitas equipes. Apesar dos partidários do DFD, em geral projetistas das basses de dados, costumeiramente "ganharem a briga", isso também não era necessariamente algo bom: em geral o diagrama de blocos funcionais se adapta muito melhor à programação estruturada funcional.

Enquanto ainda se discutia qual dos dois modelos era mais adequado, em meados da década de 60 surgiam as primeiras linguagens orientadas a objeto, já no início da década de 70 ganhando alguma força com o SmallTalk. Mas isso só complicou as coisas: apesar das vantagens evidentes das linguagens orientadas a objetos, nenhum dos modelos existentes e predominantes aderia bem a elas. Em outras palavras: os modelos existentes eram, semanticamente, muito distantes da implementação em linguagem OO.

3. IMPLEMENTANDO EM CÓDIGO

Vamos construir um programa que colete as notas de um aluno e calcule sua média usando um paradigma de divisão funcional simples, como o apresentado no diagrama anterior:



Iniciaremos especificando cada uma dessas funções.

A) imprime()

O método imprime deve receber como prâmetro a informação a ser impressa, na forma e texto. Ela não retorna nada. Sua assinatura deve ser:

public static void imprime(String texto)

B) entrada()

O método entrada deve receber um dado do usuário, no formato double. Para isso, ele deve ter como parâmetros a mensagem que irá apresentar ao usuário e deverá retornar um número no formato double. Sua assinatura deve ser:

public static double entrada(String texto)

C) processa()

O método processa deve receber os dados AV1, AV2, AV3 e FREQ e retornar o boolean false se o aluno está reprovado e true se ele estiver aprovado. Sua assinatura deve ser esta:

public static boolean(double av1, double av2, double av3, double freq)

Este é o primeiro nível da especificação. Vamos agora detalhar o código de cada trecho, testando-os um a um. Para isso, começaremos implementando um metodo **main** simplesmente para teste.

3.1. Implementação do Método imprime()

O código está apresentado a seguir:

ControleNotas.java

```
class ControleNotas {

   public static void imprime(String texto) {
      System.out.println(texto);
      }

   public static void main(String[] args) {
      imprime("Um texto de teste!");
      }
   }
```

3.2. Implementação do Método entrada()

O código está apresentado a seguir:

ControleNotas.java

```
class ControleNotas {

   public static void imprime(String texto) {
      System.out.println(texto);
      }

   public static double entrada(String texto) {
      Scanner teclado = new Scanner(System.in);
      System.out.print(texto);
      return (teclado.nextDouble());
      }

   public static void main(String[] args) {
      double teste;
      teste = entrada("Digite um numero com virgula e pressione enter: ");
      imprime("Dado digitado: " + teste);
      }
   }
```

3.3. Implementação do Método processa()

O código está apresentado a seguir:

ControleNotas.java

```
class ControleNotas {

   public static void imprime(String texto) {
      System.out.println(texto);
      }

   public static double entrada(String texto) {
      Scanner teclado = new Scanner(System.in);
      System.out.print(texto);
      return (teclado.nextDouble());
      }
```

```
    public static boolean processa(double av1, double av2, double av3, double freq) {
       double media;
       if (av2 < 4.0 || freq < 75.0) return false;
       if (av1 >= av3) {
          if (av1 < 4.0) return false;
          else {
             media = (av1 + av2) / 2;
             if (media >= 6.0) return true;
             else return false;
             }
          }
       else { // av3 > av1
          if (av3 < 4.0) return false;
          else {
             media = (av3 + av2) / 2;
             if (media >= 6.0) return true;
             else return false;
             }
          }
       }

    public static void main(String[] args) {
       if (processa(3,4,8,75)==true) imprime("Ok!");
       if (processa(3,3,9,75)==false) imprime("Ok!");
       if (processa(3,4,8,70)==false) imprime("Ok!");
       if (processa(3,10,3,80)==false) imprime("Ok!");
       if (processa(10,3,10,100)==false) imprime("Ok!");
       }
    }
```

O resultado deve ser a impressão de vários "Ok!", indicando que todas as verificações foram consideradas corretas.

3.4. Implementação do Método main()

O código está apresentado a seguir:

ControleNotas.java

```
class ControleNotas {

    public static void imprime(String texto) {
       System.out.println(texto);
       }

    public static double entrada(String texto) {
       Scanner teclado = new Scanner(System.in);
       System.out.print(texto);
       return (teclado.nextDouble());
       }

    public static boolean processa(double av1, double av2, double av3, double freq) {
       double media;
       if (av2 < 4.0 || freq < 75.0) return false;
       if (av1 >= av3) {
          if (av1 < 4.0) return false;
          else {
             media = (av1 + av2) / 2;
             if (media >= 6.0) return true;
             else return false;
             }
          }
```

```
      else { // av3 > av1
         if (av3 < 4.0) return false;
         else {
            media = (av3 + av2) / 2;
            if (media >= 6.0) return true;
            else return false;
            }
         }
      }

   public static void main(String[] args) {
      double av1, av2, av3, freq;
      av1 = entrada("Digite a nota AV1: ");
      av2 = entrada("Digite a nota AV2: ");
      av3 = entrada("Digite a nota AV3: ");
      freq = entrada("Digite a frequência: ");

      if (processa(av1,av2,av3,freq)) imprime("Aluno Aprovado!");
      else imprime("Aluno reprovado!");
      }
   }
```

4. EXERCÍCIOS

A) Use como base o programa criado nesta aula para coletar a nota de vários alunos e dizer se cada um deles está aprovado ou não.

Sugestão: faça um do~while que, antes de repetir, pergunte se o usuário quer continuar (digitando 1 para prosseguir e 0 para sair).

Sugestão 2: provavelmente você precisará criar novos métodos.

B) Adapte o programa acima para, assim que a pessoa optar por sair, ou seja, não consultar mais alunos, o sistema responda a média de todos os alunos cujas notas foram digitadas.

5. BIBLIOGRAFIA

YOURDON, E. **Análise estruturada moderna.** Editora Campus, 1990.
(Just Enough Structured Analysis, http://www.yourdon.com/strucanalysis/)

Unidade 6: Teste de Software

E Uma Introdução às GUIs

Prof. Daniel Caetano

Objetivo: Apresentar e aplicar alguns conceitos da análise funcional.

Bibliografia: MOLINARI, 2005; MOREIRA FILHO, 2002.

INTRODUÇÃO

Até o momento já vimos conceitos suficientes para desenvolver uma aplicação bastante simples, mas completa do ponto de vista estrutural e conceitual.

Enquanto apresentamos mais da biblioteca do Java e incrementamos o software já desenvolvido, nesta aula aproveitaremos para introduzir os conceitos de teste de software.

1. O QUE É TESTE DE SOFTWARE

Teste de Software é o processo pelo qual se verifica o quão correto, completo e seguro está um software, propiciando uma forma de avaliar a qualidade do software.

O ato de testar é a realização de uma comparação crítica entre o que o software faz e o que ele deveria fazer. Assim, o teste ideal seria aquele que fosse capaz de avaliar todas as possibilidades e combinações de funções de um software. Entretanto, é matematicamente comprovado que há tipos de faltas impossíveis de serem identificados por processos automáticos e, portanto, a identificação destas ficam delegadas à inspeção humana. Como tal inspeção também pode ser falha, a grande maioria dos autores sugere que não existe software isento de faltas.

Corrobora para esta teoria o fato de que, costumeiramente, considera-se que só é possível dizer que um software funciona por um determinado número de horas sem falhas se ele já houver sido testado pelo menos por aquele número de horas. Assim, para que um software que tem objetivo de funcionar "para sempre", exigiria um período de testes também "para sempre", o que certamente é impossível.

Por esta razão, é comum diferenciar os termos "falta do software" e "falha do software". "Falta do software" é um defeito que não foi descoberto, mas cuja probabilidade de existência é alta. "Falha de software", por outro lado, é um defeito já identificado e para o qual é necessário uma solução. A etapa de teste consiste em encontrar o maior número de "faltas", quando então tornam-se "falhas", para que elas possam ser corrigidas e o software que chega às mãos do usuário seja o mais livre possível de problemas.

1.1. Por que Realizar um Teste de Software

A atividade de teste pode ser uma das mais aborrecidas de um software e, sob certo aspecto, até mesmo é uma etapa custosa. Entretanto, muito mais custosa para uma empresa pode ser a divulgação ou o uso de um software que tenha passado por uma fase de teste ineficaz. Produtos defeituosos podem arruinar o nome de uma empresa e, dependendo de quão crítica é a aplicação do software, danos irreparáveis podem ser causados, tais como a morte de seres humanos.

Estima-se que o custo de correção de um defeito é tanto maior quando mais avançado o projeto/implementação estiver. Assim, se não por outras razões, esta é uma razão para se iniciar a etapa de testes o mais cedo possível dentro de um ciclo de desenvolvimento de software.

2. TESTES AO LONGO DO DESENVOLVIMENTO

Os testes podem ser planejados e realizados desde as primeiras etapas do desenvolvimento de um software.

Já na Análise de Requisitos é importante identificar, dentre os principais requisitos e restrições, algumas das situações críticas que poderão vir a ocorrer durante o tempo de vida do software.

Na etapa de Análise de Sistema devem ser identificados com mais precisão os aspectos que são passíveis de teste, considerando o software como um todo. Juntamente com a definição dos "aspectos testáveis" devem ser identificados sob que condições estes testes devem ser aplicados.

Na fase de Projeto de Sistema devem ser criadas estratégias de teste, se serão aplicados os testes automáticos ou manuais, quais os componentes críticos. Nos estágios mais avançados do Projeto devem ser desenvolvidos pequenos aplicativos de teste de componentes, bem como teste global, formando os "test-suites", conjuntos de testes automatizados que serão usados, a cada atualização de código, se o software sendo desenvolvido está se comportando como deveria.

Tais "test-suites" são muito úteis na verificação do software após mudanças no código, para verificar se a correção de um dado problema ou a inserção de uma nova característica não introduziu nenhum novo problema. Este tipo de verificação é chamada de "teste regressivo".

Assim que os primeiros componentes do software ficarem prontos, inicia-se a etapa de testes propriamente dita, quando a grande maioria dos testes é aplicada componente a componente, certificando que seu funcionamento está de acordo com as especificações.

Com base nos testes realizados e defeitos encontrados é que é possível estabelecer um padrão de qualidade, indicando se o software está pronto para ser distribuído ao público ou não.

2.1. Testes Caixa-Branca e Caixa-Preta

Os termos teste "Caixa-Branca" e "Caixa-Preta" referem-se a quanto acesso tem o indivíduo que aplica o teste em relação ao código sendo testado.

Quando o aplicador dos testes tem acesso a todo o código sendo testado, permitindo que ele realize alterações de forma a facilitar seu teste, além de poder testar o código de formas não usuais, este tipo de teste é denominado teste "Caixa-Branca".

Por outro lado, quando o aplicador dos testes tem acesso ao componente ou software como um todo, sem poder saber o que ocorre internamente no componente, apenas utilizando-o segundo as especificações e pontos de entrada fornecidos pelo fabricante, este é então chamado de teste "Caixa-Preta".

Os testes do tipo "Caixa-Branca" são normalmente realizados nos primeiros estágios de teste (alpha testing), embora algumas vezes continuem sendo usados posteriormente. Entretanto, os testes do tipo "Caixa-Branca" são, em geral, realizados apenas pelos próprios desenvolvedores.

Os testes do tipo "Caixa-Preta" são realizados por todo o tempo de desenvolvimento, mas se intensificam à medida em que os componentes são finalizados e as primeiras versões completas do sistema são produzidas. Muitas vezes o teste "Caixa-Preta" é realizado por um grupo seleto de usuários que nada têm a ver com a equipe de desenvolvimento (beta testing).

Recentemente tem-se falado nos testes "Caixa-Cinza", em situações em que o aplicador dos testes não tem acesso ao código, também estando preso à documentação do fabricante com relação à interface do componente ou sistema. Entretanto, no que se denomina "Caixa-Cinza" é possível acessar o estado interno do componente sendo testado, após cada operação (e não apenas avaliar as saídas com base nas entradas, como seria numa "Caixa-Preta" comum).

2.2. Fases de Teste

A primeira etapa dos testes, normalmente quando o software sequer está desenvolvido e muitos de seus componentes ainda estão pela metade, é chamada de "fase alpha". Nesta etapa, que se estende dos primeiros dias até o momento em que o software é liberado para o

público (seja ele restrito ou não), os aplicadores de testes são apenas aqueles que fazem parte da equipe de desenvolvimento, direta ou indiretamente, usando técnicas de "Caixa-Branca".

Mais para o fim desta primeira etapa, uma equipe de testes já deve existir junto à equipe de desenvolvimento, que passa a usar técnicas de "Caixa-Cinza" e "Caixa-Preta" para a realização de testes mais completos.

A segunda etapa dos testes, quando ele é liberado para um pequeno público externo à equipe de desenvolvimento, é chamada "fase beta". Nesta fase usuários comuns estarão utilizando o software em situações de dia-a-dia, usando técnicas de "Caixa-Preta", avaliando se encontram novas falhas no sistema e, caso as encontrem, devem informá-las à equipe de desenvolvimento.

Quando a equipe de desenvolvimento recebe um relatório de defeito destes (bug-report), a primeira atitude é tentar reproduzi-lo e, uma vez reproduzido, tentar identificar a origem e corrigi-lo.

As fases alpha e beta servem para garantir que quando um software chegue à terceira etapa (chamado "nível GA", de "General Availability", ou "Disponibilidade Geral"), que é quando o software é liberado ao grande público (muitas vezes na forma de produto comercial), ele esteja praticamente isento de defeitos.

Alguns autores chamam etapa das versões oficiais de "fase gamma", mas este termo tem sido usado de forma pejorativa, sugerindo que pela quantidade de defeitos que os software comerciais atuais possuem, essa seria não uma fase de produto final, mas sim uma terceira fase de testes.

2.3. Níveis de Teste

Há basicamente três níveis de teste: o nível funcional, o nível de componente e o nível de sistema, além do "teste regressivo" que sempre deve ser realizado.

O nível funcional (ou unitário) é aquele que testa funções ou métodos de um objeto, identificando se, para entradas comuns e incomuns o comportamento da saída é adequado. Este nível visa verificar se, num nível mais básico, o software desenvolvido se comporta como o esperado.

O nível de componente é o nível em que as bibliotecas e os objetos são testados como um todo, verificando se os comportamentos de suas diversas funções e métodos são coerentes entre si e produzem resultados satisfatórios. Este nível visa verificar se, num nível de atividades, o software desenvolvido se comporta como o esperado.

O nível de sistema é o nível mais alto, em que a integração entre diversas bibliotecas e objetos é testado, verificando se o comportamento de cada componente frente aos outros são coerentes e iguais aos esperados, se não há problemas na interface entre estes componentes.

Este nível visa verificar se o sistema está cumprindo as funções propostas no Documento de Especificação de Requisitos.

2.4. Testes de Cobertura

Dado que a identificação de um defeito em um trecho de código é diretamente proporcional ao número de vezes que tal trecho de código é executado, há regiões do código que são executadas muitas vezes e facilmente podem ter suas faltas reveladas. Por outro lado, há trechos de código que são raramente executados (como regiões de tratamento de erros, por exemplo) e, portanto, dificilmente terão suas faltas reveladas.

Por esta razão, é preciso ter um cuidado adicional com regiões do código que são pouco executadas. Mas para tomar este cuidado adicional, é preciso, antes de mais nada, identificar tais regiões.

Existem software específicos que podem ser linkados em conjunto com seu software, de forma a realizar um "profiling" e identificar as regiões do código que foram alvo dos menores tempos de processamento (e, portanto, foram executadas menos vezes). Estas regiões são as mais propensas a possuírem erros que não serão identificados.

Uma vez que tais regiões tenham sido identificadas, podem ser adotadas duas estratégias (separadamente ou conjuntamente): inspeção rigorosa do texto e planejamento de testes específicos que provoquem a execução daqueles trechos de código, de forma a tentar encontrar falhas nos mesmos.

3. IMPLEMENTANDO EM CÓDIGO

Esta aula tem dois objetivos: produzir testes simples para os nossos métodos e utilizar a classe JOptionPane para converter a aplicação de avaliação de alunos da aula passada em uma aplicação com janelas.

A estrutura era essa:

```
                    +---------+
     +------------>| Imprime |
     |              +---------+
+----+-+       +---------+
| Main +----->| Entrada |
+----+-+       +---------+
     |              +----------+
     +------------>| Processa |
                    +----------+
```

O código final da aula passada é apresentado abaixo:

Main.java

```
package Notas;                  // Use essa linha APENAS no NetBeans
import java.util.*;

public class Main {

   public static void imprime(String texto) {
      System.out.println(texto);
      }

   public static double entrada(String texto) {
      Scanner teclado = new Scanner(System.in);
      System.out.print(texto);
      return (teclado.nextDouble());
      }

   public static boolean processa(double av1, double av2, double av3, double freq) {
      double media;
      if (av2 < 4.0 || freq < 75.0) return false;
      if (av1 < 4.0 && av3 < 4.0) return false;
      if (av1 >= av3) media = (av1 + av2) / 2;
      else media = (av3 + av2) / 2;
      if (media >= 6.0) return true;
      return false;
      }

   public static void main(String[] args) {
      double av1, av2, av3, freq;
      av1 = entrada("Digite a nota AV1: ");
      av2 = entrada("Digite a nota AV2: ");
      av3 = entrada("Digite a nota AV3: ");
      freq = entrada("Digite a frequência: ");

      if (processa(av1,av2,av3,freq)) imprime("Aluno Aprovado!");
      else imprime("Aluno reprovado!");
      }
   }
```

Antes de mais nada, modifiquemos o programa para que fique gráfico. Para isso, iremos substituir o método imprime por este:

```
   public static void imprime(String texto) {
      JOptionPane.showMessageDialog(null,texto,"Mensagem!", JOptionPane.PLAIN_MESSAGE);
      }
```

Lembrando de inserir, no início do programa, a diretiva:

```
import javax.swing.*;
```

E o método entrada por este:

```
   public static double entrada(String texto) {
      String valor = JOptionPane.showInputDialog(texto);
      return (Double.parseDouble(valor));
      }
```

O código resultante está abaixo:

Main.java

```
package Notas;                  // Use essa linha APENAS no NetBeans
import javax.swing.*;

public class Main {

   public static void imprime(String texto) {
      JOptionPane.showMessageDialog(null,texto,"Mensagem!", JOptionPane.PLAIN_MESSAGE);
      }

   public static double entrada(String texto) {
      String valor = JOptionPane.showInputDialog(texto);
      return (Double.parseDouble(valor));
      }

   public static boolean processa(double av1, double av2, double av3, double freq) {
      double media;
      if (av2 < 4.0 || freq < 75.0) return false;
      if (av1 < 4.0 && av3 < 4.0) return false;
      if (av1 >= av3) media = (av1 + av2) / 2;
      else media = (av3 + av2) / 2;
      if (media >= 6.0) return true;
      return false;
      }

   public static void main(String[] args) {
      double av1, av2, av3, freq;
      av1 = entrada("Digite a nota AV1: ");
      av2 = entrada("Digite a nota AV2: ");
      av3 = entrada("Digite a nota AV3: ");
      freq = entrada("Digite a frequência: ");

      if (processa(av1,av2,av3,freq)) imprime("Aluno Aprovado!");
      else imprime("Aluno reprovado!");
      }
   }
```

Repare que a máquina virtual java "trava" após completá-lo. Para forçar o programa a sair, é necessário usar o método **System.exit**:

Main.java

```
package Notas;                  // Use essa linha APENAS no NetBeans
import javax.swing.*;

public class Main {

   public static void imprime(String texto) {
      JOptionPane.showMessageDialog(null,texto,"Mensagem!", JOptionPane.PLAIN_MESSAGE);
      }

   public static double entrada(String texto) {
      String valor = JOptionPane.showInputDialog(texto);
      return (Double.parseDouble(valor));
      }

   public static boolean processa(double av1, double av2, double av3, double freq) {
      double media;
      if (av2 < 4.0 || freq < 75.0) return false;
      if (av1 < 4.0 && av3 < 4.0) return false;
      if (av1 >= av3) media = (av1 + av2) / 2;
      else media = (av3 + av2) / 2;
```

```
      if (media >= 6.0) return true;
      return false;
      }

   public static void main(String[] args) {
      double av1, av2, av3, freq;
      av1 = entrada("Digite a nota AV1: ");
      av2 = entrada("Digite a nota AV2: ");
      av3 = entrada("Digite a nota AV3: ");
      freq = entrada("Digite a frequência: ");

      if (processa(av1,av2,av3,freq)) imprime("Aluno Aprovado!");
      else imprime("Aluno reprovado!");

      System.exit(0);
      }
   }
```

Agora vamos realizar alguns testes básicos no método de processamento: tudo zero, notas negativas, frequencia negativa, além dos casos já vistos na aula anteior para verificar se está tudo ok.

Main.java

```
package Notas;                  // Use essa linha APENAS no NetBeans
import javax.swing.*;

public class Main {

   public static void imprime(String texto) {
      JOptionPane.showMessageDialog(null,texto,"Mensagem!", JOptionPane.PLAIN_MESSAGE);
      }

   public static double entrada(String texto) {
      String valor = JOptionPane.showInputDialog(texto);
      return (Double.parseDouble(valor));
      }

   public static boolean processa(double av1, double av2, double av3, double freq) {
      double media;
      if (av2 < 4.0 || freq < 75.0) return false;
      if (av1 < 4.0 && av3 < 4.0) return false;
      if (av1 >= av3) media = (av1 + av2) / 2;
      else media = (av3 + av2) / 2;
      if (media >= 6.0) return true;
      return false;
      }

   public static void main(String[] args) {
      double av1, av2, av3, freq;

      if (processa(0,0,0,0)==false) imprime("Ok!");
      else imprime("Erro!");
      if (processa(-1,-1,-1,75)==false) imprime("Ok!");
      else imprime("Erro!");
      if (processa(10,10,10,-10)==false) imprime("Ok!");
      else imprime("Erro!");
      if (processa(3,4,8,75)==true) imprime("Ok!");
      else imprime("Erro!");
      if (processa(3,3,9,75)==false) imprime("Ok!");
      else imprime("Erro!");
      if (processa(3,4,8,70)==false) imprime("Ok!");
      else imprime("Erro!");
      if (processa(3,10,3,80)==false) imprime("Ok!");
      else imprime("Erro!");
      if (processa(10,3,10,100)==false) imprime("Ok!");
```

```
        else imprime("Erro!");


        av1 = entrada("Digite a nota AV1: ");
        av2 = entrada("Digite a nota AV2: ");
        av3 = entrada("Digite a nota AV3: ");
        freq = entrada("Digite a frequência: ");

        if (processa(av1,av2,av3,freq)) imprime("Aluno Aprovado!");
        else imprime("Aluno reprovado!");

        System.exit(0);
        }
    }
```

4. EXERCÍCIOS

A) Modifique o programa, criando um método imprimeErro para imprimir o resultado dos testes no prompt, usando o método System.out.println.

B) Tente adaptar o programa da SOMA realizado no item 4 da unidade 3 para usar as janelas, como este.

5. BIBLIOGRAFIA

MOLINARI, L. **Testes de Software: Produzindo Sistemas Melhores e Mais Confiáveis**. Editora Érica, 2005.

MOREIRA FILHO, T. **Projeto e Engenharia de Software: Teste de Software**. Alta Books, 2002.

Unidade 7: Documentação de Código

A Importância dos Comentários

Prof. Daniel Caetano

Objetivo: Desenvolver a habilidade de comentar código adequadamente

INTRODUÇÃO

Até o momento a preocupação do curso tem sido com o aprendizado da linguagem Java e o desenvolvimento de código em si. Estes tópicos são extremamente importantes, mas não estão completos sem que os códigos gerados sejam corretamente documentados.

A documentação de código pode ser feita em documentos à parte ou no próprio código. A documentação à parte pode facilitar a consulta para referência no projeto de mudanças futuras e a que fica no código é fundamental para a rápida compreensão do funcionamento do código durante sua leitura, seja porque ela ocorre muitos anos após a criação do código, seja porque a pessoa que o está lendo não foi a pessoa que o criou.

Assim, esta aula se debruça sobre a necessidade de documentar o código no próprio código e, de quebra, trata como fazê-lo de maneira que a documentação à parte possa ser gerada a partir do próprio código.

1. DOCUMENTAR CÓDIGO?

Uma vez que a função do código é explicar para o computador quais são as tarefas que ele deve cumprir, é natural que o código nem sempre seja facilmente compreendido pelos seres humanos, em especial aqueles que não estão totalmente "por dentro" do funcionamento do programa.

Tendo isto em mente, documentar um código é a técnica/arte de inserir comentários no código de maneira que ele seja mais facilmente compreendido por quem quer que o venha a ler. Trata-se de técnica porque existem algumas regras básicas a se seguir; trata-se também de uma arte, pois não há regras para definir tudo que precisa ser comentado, sendo essa uma tarefa deixada para o bom senso do desenvolvedor.

Os comentários, em java, podem ser especificados de três formas:

1) Comentários de uma linha: usa-se duas barras no início da linha:

```
// Este é um comentário de uma linha
```

2) Comentários de várias linhas: usa-se /* para iniciar e */ para finalizá-lo:

```
/* Esta é a primeira linha
   de um comentário de múltiplas
   linhas. Simples não?
 */
```

Por questões estéticas, é comum que os programadores coloquem alguns asteriscos a mais:

```
/* Esta é a primeira linha
 * de um comentário de múltiplas
 * linhas. Simples não?
 */
```

3) Comentários do tipo JavaDoc: usa-se /** para iniciar e */ para finalizá-lo:

```
/** Esta é a primeira linha
 *  de um comentário de múltiplas
 *  linhas em formato JavaDoc. Simples não?
 */
```

NOTA: A razão para se usar comentários do tipo JavaDoc será vista mais adiante.

Agora que já foram apresentadas as formas de se inserir comentários em um programa Java, precisamos entender quais são os tipos de comentários que precisaremos fazer.

Existem, basicamente, dois tipos de comentário em um código:

A) Comentários que descrevem O QUE o código faz (sempre necessários)
B) Comentários que descrevem COMO o código faz (nem sempre necessários).

Examinemos cada um deles com maior profundidade.

1.1 Comentários do tipo "O QUÊ"

Os comentários do tipo "o que" são aqueles que explicam em linhas gerais o que um trecho de código faz. É comum ter comentários deste tipo para as classes e para cada um de seus métodos, descrevendo em detalhes para que essa classe/método serve e como devem ser utilizado em um programa, como mostra o exemplo a seguir.

Main.java

```
/*  Esta classe Main é a principal do programa.
 *  Ela é responsável por inicializar o programa
 *  E executar suas tarefas mais básicas.
 *  Esta classe não depende de nenhuma outra.
 *
 *  Criada em: 10/04/2010
 *  Autor: Daniel Caetano (daniel@caetano.eng.br)
 */
public class Main {

   /*  Este método imprime um texto em uma janela.
    *  Este método depende do parâmetro "texto", do tipo String,
    *  que é o texto que será impresso na janela.
    */
   public static void imprime(String texto) {
      JOptionPane.showMessageDialog(null,texto,"Mensagem!", JOptionPane.PLAIN_MESSAGE);
      }

   /*  Este método abre uma janela e pede que o usuário digite um número.
    *  Este método depende do parâmetro "texto", do tipo String,
    *  que é a pergunta que o usuário deverá responder no campo.
    *  Este método retorna o número digitado pelo usuário, como um valor
    *  do tipo "double".
    */
   public static double entrada(String texto) {
      String valor = JOptionPane.showInputDialog(texto);
      return (Double.parseDouble(valor));
      }
   }
```

Estes comentários são obrigatórios e são os mais importantes para que nosso código possa ser USADO por alguém que não o conhece perfeitamente ou não se lembra como se usava o código. Faça com cuidado, o usuário da documentação pode ser você mesmo, no futuro, e os minutos gastos documentando o código poderão lhe poupar horas de aborrecimento futuro.

1.2 Comentários do tipo "COMO"

Os comentários do tipo "como" são aqueles que explicam em detalhe o que alguns trechos do código mais complexos fazem. Como não existe uma definição clara de o que é um "trecho complexo", a criação desse tipo de comentário acaba sendo um pouco de "arte", já deve-se evitar comentar demais - o que polui o código, assim como se deve evitar comentários de menos - o que não ajuda ninguém. Este tipo de comentário é mostrado no exemplo a seguir.

```
   public static boolean processa(double av1, double av2, double av3, double freq) {

      double media; // usada como variável auxliar para o cálculo da média

      // Alunos com AV2 menor que 4 e frequencia menor que 75% são reprovados
      if (av2 < 4.0 || freq < 75.0) return false;
      // Para o aluno ser aprovado, pelo menos a AV1 ou AV3 precisa ser >= 4
      if (av1 < 4.0 && av3 < 4.0) return false;
      // Se AV1 é a maior, ela que entra na média com AV2
```

```
if (av1 >= av3) media = (av1 + av2) / 2;
// Caso contrário, usa-se AV3 para compor a média com AV2
else media = (av3 + av2) / 2;
// Finalmente... se a média for menor que 6, aluno reprovado!
if (media < 6.0) return false;
// Se todos os critérios forem atendidos, aluno aprovado!
return true;
}
```

2. COMENTÁRIOS JAVADOC

Como é muito complicado manter as duas documentações - a do código e a em papel - ao mesmo tempo, seria interessante se pudéssemos fazer uma documentação única e, dela, extrair a outra. A Sun Microsystems pensou nisso e criou a aplicação JavaDoc, que usa os comentários do tipo "o que" para gerar a documentação externa.

O JavaDoc é um programa que lê o código das classes que escrevemos e gera um arquivo HTML par cada uma delas, resumindo todas as informações importantes que colocamos nos comentários de nosso código. O JavaDoc é uma ferramente muito versátil e permite gerar diferentes tipos de documentação "externa" com base em nosso código:

Só Públicos: com o parâmetro **-public** , o JavaDoc documenta apenas as classes, métodos e atributos públicos de um programa.
Públicos e Protegidos: com o parâmetro **-protected** , o JavaDoc documenta as classes, métodos e atributos públicos e protegidos de um programa. Este é o comportamento padrão.
Públicos, Protegidos e Pacotes: com o parâmetro **-package** , o JavaDoc documenta as classes, métodos e atributos públicos e protegidos de um programa, além de especificar os pacotes.
Tudo: com o parâmetro **-private** , o JavaDoc documenta as classes integralmente, incluindo métodos e atributos públicos, protegidos e privados de um programa, além de especificar os pacotes.

Mas, como devemos especificar os comentários para que o aplicativo JavaDoc os compreenda e possa gerar a documentação externa para nós?

2.1. Sintaxe JavaDoc

Como já foi visto, os comentários JavaDoc tem uma especificação levemente diferente dos comentários de múltiplas linhas, começando com o sinal /** e terminando com o sinal */. Este comentário só será reconhecido pelo JavaDoc se vier imediatamente ANTES da classe, interface, construtor, método ou campo/atributo (daqui em diante chamados apenas de **entidades**).

A primeira linha de um comentário JavaDoc deve ser sempre uma descrição clara e concisa do que a entidade faz, pois esta linha será usada como referência. Um ponto final ou um "tab" indica o "fim" dessa linha para o JavaDoc.

Observe o exemplo:

Main.java

```
/**  Classe principal, responsável pela inicialização e gerenciamento.
 *  Esta classe é responsável por inicializar o programa
 *  E executar suas tarefas mais básicas.
 *  Esta classe não depende de nenhuma outra.
 */
public class Main {

   /** Este método imprime um texto em uma janela.


    */
   public static void imprime(String texto) {
      JOptionPane.showMessageDialog(null,texto,"Mensagem!", JOptionPane.PLAIN_MESSAGE);
      }

   /** Este método abre uma janela e pede que o usuário digite um número.
    */
   public static double entrada(String texto) {
      String valor = JOptionPane.showInputDialog(texto);
      return (Double.parseDouble(valor));
      }
   }
```

Dentro destes comentários pode-se usar tags HTML se considerado interessante (<B>, <EM>, <I>...). Evite usar tags estruturadores, como <P>, <H1>, <HR> e outros.

Depois da primeira linha, pode-se fazer uma explicação mais extensa sobre a entidade sendo documentada. Observe o exemplo:

Main.java

```
/**  Classe principal, responsável pela inicialização e gerenciamento.
 */
public class Main {

   /** Este método imprime um texto em uma janela.
    */
   public static void imprime(String texto) {
      JOptionPane.showMessageDialog(null,texto,"Mensagem!", JOptionPane.PLAIN_MESSAGE);
      }

   /** Este método abre uma janela e pede que o usuário digite um número.
    */
   public static double entrada(String texto) {
      String valor = JOptionPane.showInputDialog(texto);
      return (Double.parseDouble(valor));
      }
   }
```

Existem, ainda, diversos indicadores que podemos e alguns que devemos usar dentro dos comentários. Estes indicadores são feitos com o uso de *tags* especiais, que devem vir no início da linha do comentário. Eles estão descritos a seguir:

@author
@deprecated
@exception
{@link}
@param
@return
@see
@serial / @serialData / @serialField
@since
@throws
@version

É interessante que todos os *tags* de um mesmo tipo venham agrupados, pois isso facilita o trabalho do aplicativo JavaDoc. Por exemplo, se um método ou classe tem mais de um autor, cada um deles deve ser especificado em uma linha iniciando com @author, mas todas essas linhas devem ser agrupadas.

Foge ao escopo deste curso estudar em profundidade todas as tags do JavaDoc, mas você pode encontrar informações sobre elas na Internet. Aqui iremos falar das mais comuns e importantes neste ponto do curso: @author, @deprecated, @param, @return, @version.

@author serve para identificar o autor de um trecho de código. Usa-se assim:

@author Nome do Autor

@deprecated serve para identificar uma classe/método que existe por compatibilidade (e, portanto, não deve ser usada em nada novo). Usa-se assim:

@deprecated Evite usar esta classe. Use a classe XXXXX no lugar desta.

@param serve para identificar para que serve um dos parâmetros de um método. Usa-se assim:

```
/** Este método imprime um texto em uma janela.
 *  @param texto Indica o texto que deve ser impresso na janela de mensagem.
 */
public static void imprime(String texto) {
   JOptionPane.showMessageDialog(null,texto,"Mensagem!", JOptionPane.PLAIN_MESSAGE);
   }
```

Observe que o "nome" depois do @param deve ser o mesmo que aparece na declaração do parâmetro do método!

@return serve para indicar o que um método retorna. Usa-se assim:

```
/** Este método abre uma janela e pede que o usuário digite um número.
 *  @return O número digitado pelo usuário.
 */
public static double entrada(String texto) {
   String valor = JOptionPane.showInputDialog(texto);
   return (Double.parseDouble(valor));
   }
```

@version serve para indicar a versão de uma classe, pacote ou método. Usa-se assim:

@version 1.10.17

3. EXERCÍCIOS

A) Comente o código abaixo, usando a sintaxe vista para o JavaDoc:

```
import javax.swing.*;

public class ContraCheque {
    public static void imprime(String texto) {
        JOptionPane.showMessageDialog(null,texto,"Atenção!",JOptionPane.PLAIN_MESSAGE);
    }

    public static double entrada(String texto) {
       return (Double.parseDouble(JOptionPane.showInputDialog(texto)));
    }

    public static double calculaImpostoRetido(double salario) {
        if (salario <= 1499.15) return 0;
        else if (salario <= 2246.75) return arDinheiro((salario-112.43)*0.075);
        else if (salario <= 2995.70) return arDinheiro((salario-280.94)*0.150);
        else if (salario <= 3743.19) return arDinheiro((salario-505.62)*0.225);
        else return arDinheiro((salario-692.78)*0.275);
    }

    public static double calculaInssRetido(double salario) {
        if (salario <= 1024.197) return arDinheiro(0.08*salario);
        else if (salario <= 1708.27) return arDinheiro(0.09*salario);
        else if (salario <= 3416.54) return arDinheiro(0.11*salario);
        else return 375.82;
    }

    public static double arDinheiro(double valor) {
        return (Math.rint(valor*100)/100);
    }

    public static void processa() {
        double bruto, inss, irrf;
        String saida;
        bruto = entrada("Digite o Salário Bruto");
        inss = calculaInssRetido(bruto);
        irrf = calculaImpostoRetido(bruto-inss);
        saida  = "Salário Bruto:   R$ " + bruto +"\n";
        saida += "Desconto INSS:   R$ " + inss +"\n";
        saida += "Salário-IRRF:   R$ " + arDinheiro(bruto-inss) +"\n";
        saida += "Desconto IRRF:   R$ " + irrf +"\n";
        saida += "Salário Líquido:   R$ " + arDinheiro(bruto-inss-irrf);
        imprime(saida);
    }

    public static void main(String[] args) {
        processa();
        System.exit(0);
    }
}
```

Unidade 8: Sobrecarga de Funções

e Vetores e Matrizes (Arrays)

Prof. Daniel Caetano

Objetivo: Uso de sobrecarga de funções para criação de código intuitivo e uso de arrays para melhorar a codificação.

INTRODUÇÃO

Agora que a linguagem Java já é conhecida em seus aspectos mais básicos, serão apresentados alguns aspectos um pouco mais avançados da linguagem. Estes aspectos, apesar de serem mais complexos conceitualmente, são de uso bastante simples e podem facilitar em muito a vida do programador.

1. SOBRECARGA DE FUNÇÕES E MÉTODOS

A grande maioria dos métodos que criamos precisam de algum tipo de parâmetro, isto é, dados de entrada para que o método possa fazer seu trabalho. São as notas de um aluno para o qual se quer calcular a média, o salário de um funcionário para o qual se quer calcular os descontos, o nome do cliente para que ele seja cadastrado... enfim, em geral, os métodos precisam de parâmetros para trabalharem. Por exemplo:

```
public static void imprime(String texto) {
   JOptionPane.showMessageDialog(null,texto,"Titulo",JOptionPane.PLAIN_MESSAGE);
   }
```

Neste código, o método imprime precisa do parâmetro "texto", que é uma String qualquer, para realizar seu trabalho.

Na linguagem Java, quando definimos um ou mais parâmetros na assinatura e um método, assinamos um contrato: o método será sempre chamado com aquele determinado número de parâmetros, que devem ser todos do tipo correto. Por exemplo, o método com a assinatura abaixo:

public static void imprime(String texto)

Ele deve ser executado, no programa, da seguinte forma:

imprime("Algum texto qualquer"); <=== Correto!

Ou seja, só podemos ter um parâmetro e ele **tem** que ser uma String. Qualquer tentativa de fazer diferente, como as indicadas abaixo, irão causar erros diversos:

imprime("Um texto", "Outro texto"); <=== Errado! Dois parâmetros!
imprime(1); <=== Errado! Parâmetro int!

Entretanto, vez ou outra queremos que um mesmo método possa ser chamado com diferentes parâmetros ou diferentes números de parâmetros. Por exemplo, poderíamos querer o método imprime funcionasse em todos os casos exemplificados anteriormente.

Como conseguir isso?

É fácil! Basta definir o método várias vezes, indicando parâmetros diferentes:

```
public static void imprime(String texto) {
   JOptionPane.showMessageDialog(null,texto,"Titulo",JOptionPane.PLAIN_MESSAGE);
   }

public static void imprime(String texto, String titulo) {
   JOptionPane.showMessageDialog(null,texto,titulo,JOptionPane.PLAIN_MESSAGE);
   }

public static void imprime(int numero) {
   String texto = "Número " + numero;
   JOptionPane.showMessageDialog(null,texto,"Titulo",JOptionPane.PLAIN_MESSAGE);
   }
```

Ou seja: em Java, nós **podemos** ter mais de um método com o mesmo nome, na mesma parte do programa, desde que:

a) O número de parâmetros seja diferente

E/OU

b) O tipo dos parâmetros seja diferente

O ato de criar múltiplos métodos com o mesmo nome e parâmetros diferentes recebe o nome de **sobrecarregar funções** (ou métodos).

Esse recurso é **muito** usado e é **muito útil**.

1.1. Otimizando um Pouco o Código

É claro que o código apresentado anteriormente é a versão mais simples. Entretanto, existe um grande número de repetições do código do método (JOptionPane....). Para evitar isso, selecionamos o método mais genérico deles (aquele com dois parâmetros) para ser o principal, como indicado no exemplo a seguir.

```
public static void imprime(String texto, String titulo) {
   JOptionPane.showMessageDialog(null,texto,titulo,JOptionPane.PLAIN_MESSAGE);
   }

public static void imprime(String texto) {
   // ... Código Aqui
   }


public static void imprime(int numero) {
   // ... Código Aqui
   }
```

Agora, reescreveremos as outras duas versões do método para usar a versão principal, eliminando a repetição de chamadas a JOptionPane...

```
public static void imprime(String texto, String titulo) {
   JOptionPane.showMessageDialog(null,texto,titulo,JOptionPane.PLAIN_MESSAGE);
   }

public static void imprime(String texto) {
   imprime(texto,"Título");
   }

public static void imprime(int numero) {
   String texto = "Número: " + numero;
   imprime(texto);
   }
```

Observe como essa representação é mais compacta e, de quebra, ela facilita a manutenção do código, já que mudanças na forma de exibição precisam ser feitas **apenas** no método principal, que é o que chama o JOptionPane.showMessageDialog.

2. EXERCÍCIOS

Para cada exercício, crie o método solicitado e modifique o método **main** para demonstrar o uso.

A) (1,0 pontos) Crie um método para imprimir uma mensagem em uma janela, com a seguinte assinatura:

public static void imprime(String texto, String titulo)

B) (1,0 pontos) Crie outro método que imprima um número do tipo **double** em uma janela, sempre com o título de "Resultado", usando o método do item A, com a seguinte assinatura:

public static void imprime(double numero)

C) (2,0 pontos) Crie um método que tire a média de 2 números double, com a assinatura:

public static double media(double n1, double n2)

D) (2,0 pontos) Crie um método que tire a média de 3 números double, com a assinatura:

public static double media(double n1, double n2, double n3)

E) (2,0 pontos) Crie um método que tire a média de dois números inteiros, com a assinatura:

public static double media(int n1, int n2)

F) (2,0 pontos) Crie um método que receba duas Strings, cada uma contendo um número, e tire a média dos dois, com a assinatura:

public static double media(String n1, String n2)

3. ARRAYS (MATRIZES E VETORES)

Além dos tipos de dados básicos já definidos (boolean, byte, char, int, long, float e double) existe ainda dois outros tipos de dados que o Java compreende: classes e arrays. As classes serão estudadas num momento futuro, mas você já conhece bem uma delas: a classe String.

Vamos focar, no momento, nos **arrays**. Um array é uma matriz (ou uma tabela, se preferir) que guarda apenas valores de um mesmo tipo. Assim, ter um array de inteiros é como ter uma tabela que só guarda números inteiros. Observe a tabela abaixo:

1	2	3	4	5	6	7

Se eu quiser criar a tabela de inteiros acima, em Java, eu indico da seguinte forma:

```
int umaTabela[] = { 1, 2, 3, 4, 5, 6, 7 };
```

Observe a indicação de [e] em frente ao nome dado à variável. É exatamente essa indicação que faz com que o Java entenda que se trata de um **array** de **inteiros** sendo criado, e não apenas um número inteiro.

Os valores associados inicialmente são indicados separados por vírgula, dentro das chaves, do lado direito do igual.

Para recuperar os valores armazenados, usamos o número da posição. Observe na tabela abaixo a indicação do número de cada posição:

Posição	0	1	2	3	4	5	6
Valor	1	2	3	4	5	6	7

IMPORTANTE: Observe que a numeração das posições começa SEMPRE em zero!

Assim, se eu quiser ler o valor da posição 4, basta usar a seguinte expressão:

umaTabela[4]

Por exemplo:

```
x = umaTabela[4];
```

Isso colocará o valor 5 em X (observe, na tabela, que o valor guardado na posição 4 é o número 5):

Posição	0	1	2	3	4	5	6
Valor	1	2	3	4	5	6	7

O uso de tabelas facilita bastante a vida do programador. Suponhamos que precisamos guardar as notas de 30 alunos e, depois, imprimi-las separadamente. O código abaixo faz o serviço (considerando que os métodos **entrada** e **imprime** sejam definidos como anteriormente):

```
int i;
// Se não vamos inicializar um array, precisamos pedir pro Java alocar espaço para ele,
// Com a instrução new.
double notas[] = new double[30];

for (i = 0; i<30; i=i+1) {
      // Guarda na posição i a nota do aluno i
      notas[i] = entrada("Digite nota do aluno " + i);
      }
for (i = 0; i<30; i=i+1) {
      imprime("A nota do aluno " + i + " é: " + notas[i] + "\n");
      }
```

No exemplo acima, como o vetor notas[] não foi inicializado no código, precisamos pedir para o Java reservar um espaço na memória para todos os 30 números (afinal, só com "notas[]" o Java não tem como adivinhar o espaço necessário!). Para fazer isso, foi usada a instrução **new**, como indicado abaixo:

double notas[] = **new double[30];**

A parte em negrito pede a reserva do espaço de 30 números do tipo double.

3.1. Arrays Multidimensionais (OPCIONAL)

É claro que a tabela não precisa ser unidimensional. Ela pode ter duas ou mais dimensões. Por exemplo, considere a tabela abaixo (as linhas e colunas em cinza marcam a numeração de linhas e colunas, não fazem parte da tabela em si):

	0	1	2
0	30	50	70
1	40	60	80

Esta tabela poderia ser declarada, em Java, da seguinte forma:

```
int outraTabela[][] = { {30, 50, 70} , {40, 60, 80} };
```

Observe que a declaração tem um par de colchetes a mais e que os números de cada linha da tabela são declarados agrupados.

4. EXERCÍCIOS

A) (1,0 ponto) Crie outro método que imprima um número do tipo **double** em uma janela, com a seguinte assinatura:

public static void imprime(double numero)

B) (1,0 ponto) Crie um método que leia um número digitado pelo usuário, através de uma janela, com a seguinte assinatura:

public static double entrada(String mensagem)

C) (2,0 pontos) Crie um método **main** que leia as médias de 10 alunos, armazenando-os em um **array** do tipo double.

DICA:

- Use um **for** para repetir a pergunta.
- Use o método desenvolvido no item B para ler as notas.

D) (2,0 pontos) Ajuste o programa desenvolvido no item C para que ele calcule a média das notas digitadas e guarde em uma variável chamda **media**.

DICA:

- Use um **for** para somar as notas.
- Para calcular a média, divida a soma total das notas pelo número de alunos.

E) (1,0 ponto) Imprima o resultado da média usando o método do item A.

F) (1,5 pontos) Ajuste o programa para que ele arredonde a nota média final para 2 casas decimais antes de imprimir.

DICA:

- Use a fórmula: valor = (Math.rint(valor*100))/100;

G) (0,5 ponto) Ajuste o método **main** para que ele também imprima a **DIFERENÇA** entre a nota média antes de arredondar e a nota média depois do arredondamento.

H) (0,5 ponto) Ajuste o método **main** para que, no final de tudo, ele imprima a menor nota da turma.

DICA:

- Use um **for** para testar todas as notas e sempre armazene a menor encontrada.

I) (0,5 ponto) Ajuste o método **main** para que, no final de tudo, ele imprima a maior nota da turma.

DICA:

- Use um **for** para testar todas as notas e sempre armazene a maior encontrada.

Unidade 9: Noções de Orientação a Objetos

Prof. Daniel Caetano

Objetivo: Apresentar os conceitos inciais de Orientação a Objetos.

Bibliografia: BEZERRA, 2007; JACOBSON, 1992; COAD, 1992.

INTRODUÇÃO

Em geral, quanto maior o software, mais complexo é seu desenvolvimento, devido às muitas partes que o compõem e o inter-relacionamento entre elas. Adicionalmente, a execução de testes e a solução de problemas frequentemente se torna um pesadelo em sistemas de software mal-elaborados.

Uma razão frequente para as dificuldades de implementação, testes e manutenção é que, em geral, quando se segue uma lógica de projeto de software baseado em decomposição funcional, como vimos anteriormente, **o sistema é desenvolvido estruturado de acordo com o que ele faz**, sem uma preocupação em representar no software o processo de uma maneira similar ao que ocorre na realidade.

O problema dessa abordagem é que, de tempos em tempos, as empresas mudam seus processos e procedimentos. Isso significa que, talvez, o que o software faz mude também... e se ele foi todo estruturado de acordo com o que ele precisava fazer antes, pode ser que agora ele precise ser totalmente reestruturado. Em outras palavras, pode ser que ele precise ser refeito (isto é, se não quisermos fazer "gambiarras").

Entretanto, alguns desenvolvedores pensaram: "*Se as funções de uma empresa e de um software mudam com muita frequência, eu não posso usá-las para basear a organização de meu programa*". Essa foi a primeira grande conclusão dos fatos acima. A pergunta que levou à segunda grande conclusão foi:

"*Mas o que é que, em uma empresa e em seus processos, raramente muda*?"

Por sorte alguém conseguiu perceber coisas que raramente mudam: **as coisas**!

Como assim, as coisas? Simples: tudo aquilo que é físico. Existe uma grande constância no uso de formulários, na produção de um determinado produto, nos funcionários envolvidos em um procedimento... as **entidades** são muito constantes... ou, em outras palavras, os **objetos** envolvidos na realização dos processos são, quase sempre, relativamente constantes. E disso surgiu a segunda grande conclusão:

"*Vamos **basear a estrutura do software nos objetos** envolvidos em seus processos*".

1. ORIENTAÇÃO A OBJETOS

Orientação a objetos é um conceito de representação da realidade em que os componentes são objetos e não funções ou estruturas de dados. Em outras palavras, se nos modelos estruturados os componentes eram definidos de acordo com características intrínsecas à implementação, nos modelos orientados a objetos estes componentes se baseiam objetos (entidades) do mundo real.

- O mundo real é composto de objetos que interagem entre si.
- Um modelo orientado a objetos é composto de objetos que interagem entre si.

Da teoria de sistemas, temos que um sistema é um conjunto de entidades que interagem entre si a fim de produzir um resultado comum. Assim, é natural o uso de "objetos programa" a fim de compor um sistema computacional.

1.1. Como São os Objetos?

No mundo real, objetos podem ser animados ou inanimados, mas qualquer um deles possui carcterísticas que podem ser classificadas como atributos ou comportamentos.

Exemplos de objetos: átomos, veículos, vias, pessoas...

Isso faz com que exista uma diferença semântica muito pequena entre modelo e a realidade que ele representa, proporcionando maior clareza.

Vantagens principais:

- **Concepção do sistema mais simples**: a transição da *realidade* para o *modelo* é facilitada.
- **Compreensão do modelo é simples**: como o *modelo* é mais próximo da *realidade*, a compreensão do modelo por quem compreende o problema real é quase automática.
- **Gerenciamento do sistema mais simples**: assim como na realidade, os objetos são estáveis na solução de um problema, ou seja, os objetos mudam muito pouco; quando é necessário resolver problemas ligeiramente diferentes, modificamos a forma com que os objetos interagem e não os objetos em si.

Mas afinal, o que são objetos em programação?

Em programação (e, de certa forma também na vida real), um objeto é um ente caracterizado por um conjunto de operações e um estado, caracterizados por *métodos* e *campos*, podendo ainda ser compostos por outros objetos.

Note que a propriedade de um objeto poder ser composto de outros objetos também atende à teoria de sistemas, já que uma entidade que faz parte de um sistema pode ser ela mesma um *subsistema*. Da mesma forma, é uma característica que está em perfeito acordo com a realidade, visto que usualmente um objeto é composto de outros objetos (ex.: uma geladeira é composta de porta, prateleiras, caixa, motor, fios...). Os próprios seres vivos são compostos de elementos chamados *células*.

Note que um objeto é uma estrutura similar à uma "estrutura de dados"; porém, além de "dados", um objeto pode armazenar também "funções". Em um objeto os dados são chamados de *atributos* e as funções são chamadas de *métodos*.

Exemplos de objetos do mundo real:

Objeto	**Métodos**	**Atributos**
TV	Liga	Canal
	Desliga	Volume
	Muda canal	Estado(ligada/desligada)

Objeto	**Métodos**	**Atributos**
Carro	Liga	Cor
	Desliga	Velocidade
	Acelera	Quilometragem (odômetro)
	Breca	Portas

1.2. Conceitos da Programação Orientadas a Objetos

Além dos objetos, a oritentação a objetos também se baseia em outros conceitos, como os de classes, mensagens e associações. Vejamos cada um destes conceitos a seguir.

CLASSES

Uma classe pode ser considerada como um "molde" de um objeto, sendo uma descrição de como um objeto pode ser criado. Uma forma interessante de explicar é que uma classe está para um objeto assim como a planta de uma casa está para a casa. Uma outra maneira de explicar é que se o objeto é um bolo, então a classe seria uma combinação entre a forma e a receita do bolo.

Exemplo:
Classe: Carro
Objetos: Carro vermelho, Carro azul, Ferrari do Daniel etc.

MENSAGENS

Objetos são capazes de executar operações. Entretanto, estas operações não são ativadas de maneira aleatória. É preciso que um objeto receba um estímulo para executar uma operação, ou seja, é preciso que alguma coisa **solicite que o objeto faça algo**.

Essa solicitação, esse estímulo para que o objeto realize uma tarefa, é chamado de *mensagem*. Em outras palavras, uma mensagem é a forma como um objeto se comunica com outro (ou estimula a outro). As mensagens que um objeto entende, em geral, são os métodos definidos em sua classe. Às mensagensagens que um objeto entende é dado o nome de *interface*.

ASSOCIAÇÕES

Como já foi visto anteriormente, um objeto pode ser composto de outros objetos diferentes. Quando objetos mais simples se unem para formar um objeto mais complexo, dizems que houve uma *associação de objetos.*

Exemplo: Carro = chassi + motor + acessórios + etc.

1.3. Principais Propriedades da Orientação a Objetos

As principais caracterísitas das classes de objetos constituem também as fundações da programação orientada a objetos. Estas características são: encapsulamento, herança e polimorfismo e a herança.

ENCAPSULAMENTO

É a propriedade que permite que um objeto seja tratado como uma "caixa preta". O interior do objeto, ou seja, "como" ele realiza as tarefas é invisível para os clientes daquele objeto. Os clientes só podem se comunicar com um objeto através da *interface* deste objeto, sendo que a interface de um objeto nada mais é do que a definição de quais mensagens ele "sabe" responder.

Exemplo: o carro é um objeto que pode ser tratado como uma caixa preta; uma pessoa pode dirigir sem saber como funciona o motor do carro, basta conhecer o uso de sua interface, isto é, como girar a direção, pisar nos pedais e trocar de marcha com o câmbio.

HERANÇA

Herança é a propriedade que nos permite criar uma nova classe especificando que ela "é uma" outra classe também. Por exemplo, se temos a classe "Pessoa", podemos criar a classe "Trabalhador" dizendo que *Trabalhador é uma Pessoa.* Assim, um objeto da classe Trabalhador vai também possuir todos os atributos e métodos de um objeto da classe Pessoa (como *nome*, por exemplo).

Assim, se existe uma relação de que a ***Classe B*** "é uma" ***Classe A***, isso significa que a Classe B é capaz de fazer tudo que a Classe A faz, ou seja, a Classe B **herda** os atributos e métodos da Classe A. Neste caso, pode-se dizer:

1) A é a ***superclasse*** de B;
2) A é a ***generalização*** de B;
3) B é uma ***subclasse*** de A;
4) B é uma ***especialização*** de A.

5) B respeita a mesma interface que A.

POLIMORFISMO

Polimorfismo é uma propriedade que permite que um objeto que conheça uma determinada *interface*, pode se comunicar, isto é, trocar mensagens com qualquer outro objeto que respeite aquela *interface* (ou seja, que tenha os mesmos atributos e métodos), independentemente de qual seja o tipo do objeto com quem está se comunicado. Em outras palavras, dois objetos que conheçam uma mesma *interface* podem se comunicar, independentemente de quais sejam suas classes.

Exemplo: Se Carro Azul e Caminhonete Vermelha possuem a mesma interface, que é conhecida por João, então:

Objeto	Ação	Objeto		Objeto	Ação	Objeto
João	Dirige	Carro azul	=>	João	Dirige	Caminhonete Vermelha

Trocando em miúdos, se carro e caminhonete possuem a mesma interface de operação, que é conhecida por João, então João saberá usar/dirigir tanto o carro quanto a caminhonete, mesmo que o objeto caminhonete tenha sido inventado muito tempo depois da criação do objeto João.

RELAÇÃO ENTRE HERANÇA E POLIMORFISMO

De uma forma rigorosa, podemos dizer que herança é a propriedade que permite que, ao especializar uma classe, os objetos da nova classe preservem todos os comportamentos e atributos dos objetos da classe original, ou seja, os comportamentos e atributos são *herdados*. Em outras palavras, a nova classe (mais especializada) continua a respeitar a *interface* estabelecida pela classe original.

Exemplo:

classe: Pessoa	ação: dirige	**classe: Veículo**
objeto: joao		***subclasse: Carro*** (é um Veículo)
objeto: carla		*objeto: carroAzul*
		subclasse: Caminhonete (é um Veículo)
		objeto: caminhoneteVermelha

Se objetos da classe Pessoa conhecem a interface que lhe permite dirigir um objeto da da classe Veículo, então eles conhecerão também como dirigir um objeto das classes Carro e Caminhonete.

Isso ocorre porque Carro e Caminhonete são classes **especializadas** da classe Veículo original. Assim, se *joao* e *carla* são objetos da classe Pessoa e *carroAzul* é um objeto da classe Carro (que é um Veículo) e *caminhoneteVermelha* é um objeto da classe Caminhonete (que é um Veículo), então tanto o objeto ***joao*** quanto o objeto ***carla*** podem interagir com carroAzul e caminhoneteVermelha.

Muitas linguagens, incluindo C++ e Java, utilizam a propriedade da Herança para implementar diversos tipos de Polimorfismo. O Java inclui também o tipo "interface" para esta finalidade.

2. EXERCÍCIOS

O objetivo deste exercício é exercitar a criação de classes e objetos, apresentando os conceitos ao aluno.

1. Usando JCreator, NetBeans, Eclipse ou qualquer outro programa de sua preferência, inicie um projeto com uma classe **Pessoa**, conforme descrita abaixo, comentando adequadamente o código:

Nome:
Pessoa
Atributos:
- nome (String)
- idade (int)

O código inicial deve ficar mais ou menos como é descrito abaixo.

```
/**
 * A classe Pessoa define características importantes de uma pessoa.
 * @author (seu nome)
 * @version 20100227_001
 */
public class Pessoa {
   // Variáveis de Instância
   public String nome;
   public int idade;
   }
```

2. Toda classe deve ter um método construtor, responsável por definir os valores iniciais dos atributos sempre que um objeto é criado. O contrutor deve ter EXATAMENTE o mesmo nome que a classe, conforme mostrado a seguir:

```
/**
 * A classe Pessoa define características importantes de uma pessoa.
 * @author (seu nome)
 * @version 20100227_001  <<== Versão é muito importante!
 */
public class Pessoa {
   // Variáveis de Instância
   public String nome;
   public int idade;

   /**
    * Construtor for objects of class Pessoa
    */
   public Pessoa() {
      // Inicializa variáveis de instância
      }

   }
```

3. Vamos modificar o construturo para que ele receba informações sobre o objeto, como nome e idade, e modifique os valores internos do objeto.

```
/**
 * A classe Pessoa define características importantes de uma pessoa.
 * @author (seu nome)
 * @version 20100227_001
 */
public class Pessoa {
   // Variáveis de Instância
   public String nome;
   public int idade;

   /**
    * Construtor for objects of class Pessoa
    */
   public Pessoa(String umNome, int umaIdade) {
     // Inicializa variáveis de instância
      nome = umNome;
      idade = umaIdade;
      }
   }
```

4. Vamos criar/modificar o método **main** para criar um objeto desta classe.

```
/**
 * A classe Pessoa define características importantes de uma pessoa.
 * @author (seu nome)
 * @version 20100227_001
 */

public class Pessoa {
   // Variáveis de Instância
   public String nome;
   public int idade;

   /**
    * Construtor for objects of class Pessoa
    */
   public Pessoa(String umNome, int umaIdade) {
      // Inicializa variáveis de instância
      nome = umNome;
      idade = umaIdade;
      }

   /**
    * Rotina de Teste da Classe
    */
   public static void main(String[] args) {
      Pessoa joao = new Pessoa ("João Alves", 18);
      Pessoa maria = new Pessoa ("Maria Alves", 17);
      System.out.println("Nome: " + joao.nome + "\nIdade: " + joao.idade + "\n");
      System.out.println("Nome: " + maria.nome + "\nIdade: " + maria.idade + "\n");
      }
   }
```

5. Compile e execute este programa e observe os resultados.

Um pouco de Boas Práticas

6. A classe (muito simples) criada não segue alguns padrões de boas práticas. O primeiro deles é a falta de comentário adequado no construtor.

```
/**
 * A classe Pessoa define características importantes de uma pessoa.
 * @author (seu nome)
 * @version 20100227_001
*/

public class Pessoa {
   // Variáveis de Instância
   public String nome;
   public int idade;

   /**
    * Construtor para objetos da classe Pessoa
    * @param      umNome       o nome da pessoa
    * @param      umaIdade     a idade da pessoa
    * @return     não há valor de retorno
    */
   public Pessoa(String umNome, int umaIdade) {
      // Inicializa variáveis de instância
      nome = umNome;
      idade = umaIdade;
      }

   /**
    * Rotina de Teste da Classe
    */
   public static void main(String[] args) {
      Pessoa joao = new Pessoa ("João Alves", 18);
      Pessoa maria = new Pessoa ("Maria Alves", 17);
      System.out.println("Nome: " + joao.nome + "\nIdade: " + joao.idade + "\n");
      System.out.println("Nome: " + maria.nome + "\nIdade: " + maria.idade + "\n");
      }
   }
```

BIBLIOGRAFIA

BEZERRA, E. **Princípios de Análise e Projeto de Sistemas com UML**. 2ed. Rio de Janeiro: Editora Campus Elsevier, 2007.

JACOBSON, I; CHRISTERSON, M; JONSSON, P; ÖVERGAARD, G. **Object-oriented software engineering: a use case driven approach.** Essex, England: Addison-Wesley Longman Ltd, 1992.

COAD, P; YOURDON, E. **Análise baseada em objetos.** Editora Campus, 1992.

Unidade 10: Introdução à Programação de GUIs com Swing

Prof. Daniel Caetano

Objetivo: Apresentar os conceitos inciais de Programação Orientada a Eventos.

Bibliografia: BEZERRA, 2007; JACOBSON, 1992; COAD, 1992.

INTRODUÇÃO

Uma interface gráfica com o usuário (**GUI**) é um meio pelo qual pode ocorrer **comunicação entre um ser humano e um computador**, utilizando-se para isso de uma metáfora de manipulação direta **através de imagens e texto**. Em outras palavras, as GUIs são instrumentos para que o usuário possa operar um computador através de metáforas gráficas.

Existem diversos tipos de GUIs, sendo o mais comum aquele que usa janelas, ícones, menus e um dispositivo apontador (**WIMP: Windows, Icons, Menus e Pointing Device**, tendo sido este um dos modelos pioneiros de interface, desenvolvido pela Xerox, e utilizado até hoje nos computadores.

Um dos grandes **benefícios** que as GUIs trouxeram para o ramo da computação foi uma grande **redução na dificuldade de uso dos computadores**. Isto é conseguido através do uso de metáforas de fácil reconhecimento pelo usuário para a execução de tarefas computacionais (por exemplo: apagar um documento pode ser feito arrastando o ícone do documento em um ícone de lixeira), o que permite que um usuário opere o computador com uma necessidade mínima de aprendizagem.

Além disso, as GUIs **permitem uma grande flexibilidade** na forma com que as tarefas são executadas, possibilitando um projeto de comunicação com o usuário que vise a um aumento de produtividade do mesmo, reduzindo seu cansaço ao máximo.

1. EVOLUÇÃO DAS UIs (OPCIONAL)

Os primeiros computadores desenvolvidos, há cerca de um século, eram equipamentos muito simples, mecânicos, que sequer eram programáveis. A interface de comunicação destes equipamentos, que tinham apenas algumas poucas funções, era feita através alavancas mecânicas e engrenagens.

Os equipamentos foram evoluindo lentamente e já em meados do século XX existiam computadores programáveis, mas sua interface de comunicação ainda era bastante precária, através de conexões elétricas físicas e cartões perfurados.

Durante a segunda metade do século XX os avanços foram incríveis. Surgiram teclados, impressoras e posteriormente monitores, e foram estes equipamentos que trouxeram as interfaces com o usuário para a era visual.

Inicialmente foram desenvolvidos os primeiros sistemas para impressora, que operavam basicamente com estrutura de terminal. O usuário digitava e o conteúdo aparecia na impressora. Com os monitores de vídeo, a interatividade aumentou, mas os primeiros software eram apenas adaptações do que já se utilizava nas impressoras para o vídeo. Estas interfaces estão disponíveis até hoje, em diversos sistemas, e são chamadas **Interfaces de Linha de Comando** (CLIs, Command Line Interfaces).

Durante algum tempo, os sistemas operavam na forma de texto puro, mas à medida em que os usuários se diversificavam (engenheiros, matemáticos, etc... e não apenas especialistas em computadores), começaram a surgir as primeiras **interfaces de menus**, embora ainda em modo texto. Seu uso ficou bastante popular no início da década de 80 e a grande maioria dos software desenvolvidos até 1985 usava este tipo de interface.

Entretanto, já no fim da década de 1970, uma idéia brilhante havia surgido em um dos laboratórios da **Xerox**, o PARC (Palo Alto Research Center), onde havia sido criado o computador **Star**, que era operado através de um ***mouse*** (o primeiro da história) e a comunicação do usuário era feita através de imagens **gráficas**. Este equipamento nunca tornou-se viável devido ao seu alto custo.

Posteriormente foi desenvolvido o padrão **X-Windows** para computadores que executassem o sistema operacional compatível com UNIX. Este padrão tornou-se muito importante e tem muitas características especiais, mas só veio a ter participação no mercado de usuários não especialistas a partir do meio da década de 1990.

Entretanto, já no início da década de 1980, a Apple Computers, naquela época uma pequena empresa, começava a desenvolver um computador pessoal que fosse capaz de executar uma interface daquele tipo, a um preço acessível. O resultado disso foi um computador chamado **Lisa**, que pode ser confundido com uma "versão beta" do **MacIntosh**. O Lisa não fez tanto sucesso, mas a Apple tinha feito o mais importante: tinha criado o *know-how*, ou seja, ela sabia como fazer.

Pouco tempo depois surgia a primeira versão do MacIntosh, a um preço acessível (apesar de alto), que tornou-se o primeiro computador pessoal comercial a oferecer uma interface gráfica com o usuário (GUI). A interface era bastante pobre, mas já era de uso bem mais simples que as tradicionais interfaces de "linha de comando".

A IBM, a grande empresa ocidental dos computadores pessoais da época (no oriente a história é bem mais complexa e pouco conhecida) não quis ficar atrás e contratou com a Microsoft que fizesse uma nova versão do MS/IBM-DOS, com uma interface gráfica e usando os recursos dos mais recentes processadores da Intel (na época, o i80286). Neste mesmo instante a IBM iniciou pesquisas na área de interfaces com usuário, para determinar qual seria a interface perfeita.

Deste convênio surgiram dois software distintos: o Microsoft **Windows** e o MS/IBM **OS/2**. O primeiro deles era apenas uma interface nova para o velho MS/IBM-DOS, já o segundo era um sistema operacional completamente novo, o qual já levava a interface gráfica em consideração em seu projeto. Na aparência, entretanto, os primeiros Windows e os primeiros OS/2 eram absolutamente similares, mas o OS/2 exigia bem mais recursos para executar, uma vez que fazia bem mais coisas que o Windows de então.

Por diversas razões mercadológicas, o Windows acabou se instituindo como padrão e a Microsoft resolveu romper seus contratos com a IBM, no início da década de 1990. A IBM não aceitou essa quebra de contrato pacificamente e, com os primeiros resultados de sua pesquisa sobre interfaces gráficas nas mãos (padrão **CUA-1991**, Common User Access), a IBM resolveu dar um passo ousado e contribuir significativamente para o avanço das interfaces gráficas.

O padrão CUA-1991 especificava diversas características de usabilidade para as interfaces, de forma a minimizar ao máximo a curva de aprendizagem dos usuários. Além disso, o padrão considerava uma implementação totalmente orientada a objetos, permitindo que os software desenvolvidos para a interface se integrassem a ela de forma tão sutil que o usuário nem precisaria ter consciência de quando utilizava uma ferramenta de sistema ou um software de terceiros.

Em 1992 a IBM lançava sua versão 2.0 do OS/2, contendo a WorkPlace Shell, implementando grande parte das características do CUA-1991. A Apple correu e implementou muitas das características também em seu MacOS 7, que na época sequer tinha o nome de MacOS.

Muitos dos recursos do CUA foram mimetizados pela Microsoft no Windows na versão 4.0 do mesmo (conhecido pelo nome de Windows 95). A partir de então os avanços foram muito disputados entre Apple, Microsoft, IBM e um novo jogador: Linus Torvalds que, com o seu Linux, trazia o mundo Unix para a realidade do usuário comum e, com ele, o X-Windows na versão gratuita chamada XFree.

Todas estas interfaces evoluíram e mudaram em muitos aspectos, culminando hoje com as disponíveis no MacOS X, Windows 7 e, no Linux, as diversas como KDE, Gnome, dentre outras. A WorkPlace Shell da IBM, apesar de ter sido a única a realmente implementar completamente orientação a objetos na interface, foi descontinuada em 2005, devido à mudanças na estratégia da empresa desde 1996, que fizeram com que o número de usuários se tornasse bastante reduzido.

2. A ORIENTAÇÃO A EVENTOS DE UMA GUI

Do ponto de vista do usuário, a principal característica das GUIs é, obviamente, que sua operação é gráfica. Entretanto, esta é apenas a diferença mais superficial, no que se chama

de "***look***" (aparência). Existe uma diferença também referente ao "***feel***" (sentimento) que é fundamental e tem uma importância fundamental inclusive na etapa de programação.

Esta diferença se manifesta em "**quem tem o controle**" da operação. Nas antigas **interfaces modo texto**, quem definia "o que acontecerá a seguir" **era sempre o computador**; de uma certa forma, o usuário era refém do software, algumas vezes sem saber como sair de uma dada tela ou mesmo como fechar um programa.

As interfaces modo texto via de regra se baseiam na interação do tipo diálogo: o computador pergunta e o usuário responde. Assim que o usuário responder todas as perguntas, o computador faz algum processamento e posteriormente permite que o usuário escolha alguma outra opção.

As **interfaces gráficas**, por sua vez, operam no modo de "manipulação direta", onde o usuário pode selecionar qualquer elemento visível da tela e comandar uma ação sobre ele, **sem uma ordem pré-estabelecida**. Isto significa que, nas GUIs, em geral é o usuário quem determina "o que vai acontecer em seguida".

Esta característica é fundamental do ponto de vista da programação, dado que ela implica numa mudança completa no conceito base para o projeto da interface. Enquanto nas **interfaces texto o código da interface é meramente seqüencial**, com uma ordem bastante rígida, **nas GUIs este código não tem uma ordem de execução prevista**: dependendo da ação do usuário, um trecho totalmente diferente do código será executado.

Chamamos de "**evento**" cada **ação do usuário** e, se são estes eventos que controlam a execução de um software, este software é dito ***orientado a eventos*** e, portanto, requer uma programação orientada a eventos.

2.1. Programação Orientada a Eventos

A **programação orientada a eventos**, no fundo, não difere muito da programação usual. A diferença reside na **estrutura de controle** do que será executado em cada instante.

Na **programação seqüencial** usual, o **software costuma ter um *loop* principal** que é executado até que o usuário selecione uma opção para sair. Todo o código do software fica dentro deste *loop* e é executado ininterruptamente.

Na **programação orientada a eventos**, o *loop* contém apenas uma chamada ao sistema, verificando se alguma *mensagem de evento* chegou para o programa. Caso nenhuma mensagem tenha chegado, o software fica bloqueado (sem executar) e o sistema (ou outros programas) continua sua operação. Caso alguma **mensagem** tenha chegado, o sistema retorna a mensagem que, através de uma **estrutura do tipo *switch*** escolhe qual trecho de código vai ser executado naquele instante.

Uma má prática de programação é colocar *código útil*, ou seja, o código que realiza a tarefa para a qual o software foi projetado, dentro deste *switch* ou mesmo em outros lugares. Isto não deve ser feito pois, fazendo isso, o programa exigirá a interface gráfica para poder realizar suas tarefas. O ideal é que dentro destas regiões sejam apenas chamadas operações de um outro componente do sistema, este sim executando as atividades do programa. Neste tipo de arquitetura, a MVC, a interface gráfica ocupa a camada *visão* (*view*) e as atividades do sistema ocupam a camada *controle* (*controller*).

3. CONCEITOS DE PROGRAMAÇÃO GUI

Como foi visto, a **programação GUI** é feita através do paradigma denominado "**programação orientada a eventos**". Isso significa que as tarefas do programa não ocorrem em uma ordem pré-definida: as tarefas são executadas quando determinados eventos ocorrem, na ordem em que tais eventos ocorrem.

Entretanto, **é necessária toda uma preparação** para que um software possa agir quando algum evento ocorre, além de ser preciso criar as janelas da aplicação e cuidar de uma série de detalhes que não são necessários em uma programação usual para modo texto (CLI).

Assim como existem diversas formas de resolver um problema usando programação, existem também diversas formas de programar uma GUI, não existindo uma forma explicitamente "correta" ou "incorreta". Em geral convenciona-se chamar de "correta" uma implementação extensível e de "incorretas" todas as outras formas de resolver o mesmo problema.

Nesta aula o foco está em aprender os conceitos básicos da programação orientada a eventos, então não serão focados os aspectos de extensibilidade. Serão, entretanto, apresentadas duas formas de executar uma mesma tarefa: a execução de uma soma com o uso de janelas GUI para receber os dados e apresentar a resposta. A primeira delas é a mais simples programacionalmente, mas a menos extensível e a menos prática. A segunda delas é um pouco mais flexível, mas também é de programação mais complexa.

3.1. Um Aplicativo de Soma com GUI Básica

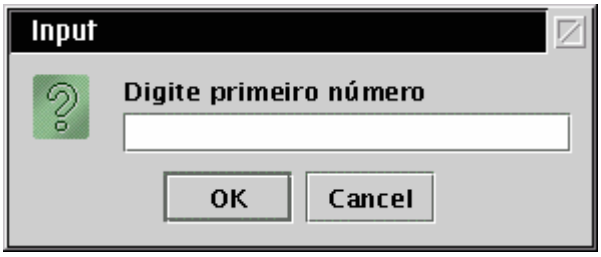
A primeira implementação segue conceitos já vistos em aulas anteriores, apenas para que fique bem clara a proposta da atividade. O projeto terá apenas uma classe, a classe Main.

Esta classe fará todas as funções da adição, criando janelas para pedir informações e janelas para apresentar os resultados. Serão usadas janelas prontas da biblioteca Swing do Java. Acompanhe os detalhes da implementação pelos comentários no código:

No NetBeans, crie um projeto chamado "calculadora" e defina a classe principal dele como "calculadora.Main"

```
/* Primeiramente definimos o pacote da aplicação */
package calculadora

/* Agora especificamos que vamos usar a biblioteca
   JOptionPane, do pacote javax.swing */
import javax.swing.JOptionPane;

/* Em seguida criamos a classe de nossa aplicação */
public class Main {
    /* E é definido o método main (principal) */
    public static void main ( String args[] ) {
        /* Declaramos algumas variáveis temporárias: alphaNum1 e
           alphaNum2 serão usadas para guardar os valores digitados
           pelo usuário no formato String (as janelas sempre retornam
           valores digitados pelo usuário como strings). num1 e num2
           serão usadas para colocar o valor digitado pelo usuário já
           convertidos para números e soma será usado para guardar o
           resultado da soma. */
        String alphaNum1, alphaNum2;
        int num1, num2, soma;

        /* Usamos o método estático showInputDialog (mostraJanelaDeEntrada)
           da classe JOptionPane (JPainelDeOpções) para colher entradas do
           usuário. A primeira chamada serve para pegar o primeiro valor da
           soma e a segunda para pegar o segundo valor da soma. Note que o
           valor é guardado na variável do tipo String */
        alphaNum1 = JOptionPane.showInputDialog ("Digite primeiro número");
        alphaNum2 = JOptionPane.showInputDialog ("Digite segundo número");

        /* Agora o método estático parseInt (traduzInteiro) da classe
           Integer (Inteiro) é chamado para traduzir o texto digitado pelo
           usuário em números, armazenados nas variáveis num1 e num2.
           Note que seria necessário realizar uma verificação, caso o
           valor digitado não fosse um número.  */
        num1 = Integer.parseInt( alphaNum1 );
        num2 = Integer.parseInt( alphaNum2 );

        /* A soma é realizada, e o resultado é armazenado em soma. */
        soma = num1 + num2;

        /* Usamos o método estático showMessageDialog (mostraJanelaDeMensagem)
           da classe JOptionPane (JPainelDeOpções) para mostrar o resultado para
           o usuário. */
        JOptionPane.showMessageDialog(null,"Soma: " + soma,
                "Soma de dois inteiros", JOptionPane.PLAIN_MESSAGE );
        }
     }
```

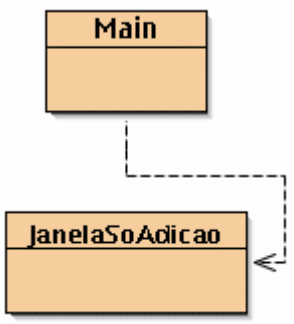
É importante notar que, neste exemplo, como usamos diretamente as janelas fornecidas pelo pacote Swing do Java, não precisamos nem mesmo entrar em contato com as mensagens (embora elas existam, dentro das janelas que abrimos)! Entretanto, este software é extremamente limitado e até mesmo pouco prático, pois usa várias janelas para fazer uma única coisa. Além disso, ele praticamente nem é orientado a objetos. Uma das janelas desta implementação está representada abaixo:



3.2. Um Aplicativo de Soma com GUI mais Avançada

A segunda implementação que será examinada já é um pouco mais complexa. Serão utilizadas três classes (duas delas explícitas e a outra implícita) para realizar a tarefa. Uma delas será a classe principal do aplicativo e a outra será a classe da janela. A execução da tarefa será toda executada dentro da classe de janela, que é única em todo o processo.

Conceitualmente, este é um "mal programa", entretanto é este tipo de programa que se costuma produzir usando os aplicativos de auxílio ao desenvolvimento de interfaces gráficas. Maneiras melhores de desenvolver um software de janela serão examinadas na disciplina Programação de Componentes. O diagrama de classe do novo programa proposto está representado abaixo:



A classe Main é a classe principal do aplicativo e a classe JanelaSoAdicao é a classe que apresenta a janela onde são realizadas as operações. Notem a seta indicando que "Main usa JanelaSoAdicao".

Acompanhe os detalhes da implementação pelos comentários do código, inciando pela classe Main.

No NetBeans, crie um projeto chamado "calculadora2" e defina como classe principal a classe "calculadora2.Main".

```
/* Primeiramente definimos o pacote da aplicação */
package calculadora2;

/* Agora criamos a classe da Aplicação */
public class Main {
    /* E seu método principal */
    public static void main ( String args[] ) {
        /* Declaramos uma referência para a janela que será criada */
        JanelaSoAdicao umaJanela;

        /* E é feita uma solicitacao que o Java nos crie a janela */
        umaJanela = new JanelaSoAdicao();
        }
    }
```

A classe principal é muito simples: ela apenas cria um objeto do tipo JanelaSoAdicao e o deixa realizar seu "trabalho". O código inicial da classe JanelaSoAdicao segue abaixo.

No NetBeans, clique com o botão direito no pacote "calculadora2" e selecione "Novo > Classe Java", dando o nome de "JanelaSoAdicao" à nova classe.

```
/* Primeiramente definimos o pacote da aplicação */
package calculadora2;

/* Agora especificamos que vamos usar várias bibliotecas
   dos pacotes java.awt, java.awt.event e javax.swing */
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/* Em seguida é declarada a classe da janela da aplicação, que
   é uma especialização da classe JFrame */
public class JanelaSoAdicao extends JFrame {
    /* Aqui são declaradas as variáveis onde serão realizadas
       as operações, num1 e num2 para armazenar os valores numéricos
       digitados pelo usuário e soma para armazenar o valor a ser
       apresentado como saída */
    private int num1, num2, soma;

    /* O próximo passo é declarar o construtor da classe JaneaSoAdicao
       que nada faz além de construir a janela */
    public JanelaSoAdicao() {
        /* No caso de janelas derivadas de JFrame e JDialog, a primeira
           instrução deve ser - sempre - uma chamada ao construtor da
           superclasse. Esta chamada está especificando o título da
           janela a ser criada. */
        super("Programa de Adicao");

        /* Configurações Finais da Janela */
        /* Indica que, ao apertar o botão fechar, o aplicativo deve sair */
        setDefaultCloseOperation ( JFrame.EXIT_ON_CLOSE );
        /* Ajusta o tamanho da janela */
        setSize ( 350, 80 );
        /* Indica que a janela não pode ser redimensionada */
        setResizable ( false );
        /* Indica para a janela aparecer */
        setVisible ( true );
        }
    }
```

Isso apresenta a janela, mas ela ainda está vazia, não tem elemento nenhum. Para que possamos adicionar elementos na janela, precisamos entender que a janela é composta por duas partes: **o frame** (ou moldura) e **a área cliente**. Normalmente não alteramos o frame, que é a moldura da janela, mas é comum desejarmos modificar a área cliente, ou seja, o interior da janela.

No Swing essa região é um objeto do tipo Conteiner, porque ele armazena todos os elementos da janela, ou seja, ele contém os elementos da janela. Assim, precisamos pegar uma referência, do tipo conteiner, para a região cliente, que vamos chamar de **painel**.

```
package calculadora2;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
```

```
public class JanelaSoAdicao extends JFrame {
    private int num1, num2, soma;

    public JanelaSoAdicao() {
        super("Programa de Adicao");

        /* Agora é declarada uma referência para o painel da janela
           criada, de forma que possamos adicionar coisas neste painel. */
        Container painel;
        /* E é pedido para a janela o valor da referência do painel */
        painel = getContentPane();

        setDefaultCloseOperation ( JFrame.EXIT_ON_CLOSE );
        setSize ( 350, 80 );
        setResizable ( false );
        setVisible ( true );
        }
    }
```

Agora precisamos criar referências para os objetos que serão colocados na janela: dois campos de texto para entrada (tipo JTextField), um botão (tipo JButton) e um campo texto de saída (tipo JTextField)... e depois criar estes objetos:

```
package calculadora2;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class JanelaSoAdicao extends JFrame {
    /* Aqui são declaradas as referências para os objetos que
       estarão presentes na janela: dois campos de texto para a
       entrada (entrada1 e entrada2), um botão para executar a
       soma (botaoSoma) e um outro campo de texto para a saída
       do resultado (saida). */
    private JTextField entrada1, entrada2;
    private JButton botaoSoma;
    private JTextField saida;

    private int num1, num2, soma;

    public JanelaSoAdicao() {
        super("Programa de Adicao");

        Container painel;
        painel = getContentPane();

        /* Cria-se o primeiro campo de entrada (com 5 posições)... */
        entrada1 = new JTextField(5);
        /* Cria-se o segundo campo de entrada (com 5 posições)... */
        entrada2 = new JTextField(5);
        /* Cria-se o botão com o texto "Soma!" */
        botaoSoma = new JButton("Soma!");
        /* Cria-se o campo de saída (com 10 posições) */
        saida = new JTextField(10);

        setDefaultCloseOperation ( JFrame.EXIT_ON_CLOSE );
        setSize ( 350, 80 );
        setResizable ( false );
        setVisible ( true );
        }
    }
```

Agora que temos os elementos criados, precisamos adicioná-los ao painel. Entretanto, para que o painel aceite estes elementos, devemos indicar **como eles devem ser organizados**. Existem vários tipos de organização (procure na documentação da classe Conteiner, o método setLayout, para verificar as possibilidades). Neste caso usaremos um layout fluido (tipo **FlowLayout**), isto é, que não é fixo, posicionando um elemento ao lado do outro.

Quando o tipo de layout estiver definido para o painel, basta adicionar elementos nele:

```
package calculadora2;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class JanelaSoAdicao extends JFrame {
    private JTextField entrada1, entrada2;
    private JButton botaoSoma;
    private JTextField saida;

    private int num1, num2, soma;

    public JanelaSoAdicao() {
        super("Programa de Adicao");

        Container painel;
        painel = getContentPane();

        /* Indica-se o layout como FlowLayout, ou seja, uma disposição
           em que os elementos serão colocados lado a lado. */
        painel.setLayout ( new FlowLayout() );

        entrada1 = new JTextField(5);
        /* Coloca-se o campo de entrada no painel. */
        painel.add(entrada1);

        entrada2 = new JTextField(5);
        /* Coloca-se o campo de entrada no painel. */
        painel.add(entrada2);

        botaoSoma = new JButton("Soma!");
        /* Coloca-se o botão no painel. */
        painel.add(botaoSoma);

        saida = new JTextField(10);
        /* Coloca-se o campo de saída no painel. */
        painel.add(saida);

        setDefaultCloseOperation ( JFrame.EXIT_ON_CLOSE );
        setSize ( 350, 80 );
        setResizable ( false );
        setVisible ( true );
        }
    }
```

A classe já está ok... só falta agora fazer duas coisas: primeiro, marcar o campo de saída de texto como "Não Editável"; segundo, criar uma classe/método (tipo **ActionListener**) para executar a função do botão.

Observe atentamente o código a seguir e seus comentários:

```
package calculadora2;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class JanelaSoAdicao extends JFrame {
    private JTextField entrada1, entrada2;
    private JButton botaoSoma;
    private JTextField saida;

    private int num1, num2, soma;

    public JanelaSoAdicao() {
        super("Programa de Adicao");
```

```
        Container painel;
        painel = getContentPane();
        painel.setLayout ( new FlowLayout() );

        entrada1 = new JTextField(5);
        painel.add(entrada1);

        entrada2 = new JTextField(5);
        painel.add(entrada2);

        botaoSoma = new JButton("Soma!");
        painel.add(botaoSoma);

        saida = new JTextField(10);
        /* Marca-se este campo como "não editável" */
        saida.setEditable(false);
        painel.add(saida);

        /* Agora será associada uma classe implícita, implementando
           a interface ActionListener, como a executora da ação do
           botão criado anteriormente. */
        botaoSoma.addActionListener( new ActionListener() {
            /* Quando o botão for clicado, o método actionPerformed
               desta classe será chamada. Deve-se, então, redefinir
               esse método para que ele realize as atividades desejadas */
           public void actionPerformed( ActionEvent event ) {
                /* entrada1.getText() retorna o texto que o usuário
                   digitou no campo entrada1. Este texto já é repassado
                   para o método Integer.parseInt, que o traduz em um
                   número e o armazena na variável num1. O procedimento
                   para obter o segundo valor é o mesmo, colhendo a
                   informação em entrada2 e armazenando o resultado em num2. */
                num1 = Integer.parseInt( entrada1.getText() );
                num2 = Integer.parseInt( entrada2.getText() );

                /* A soma é realizada normalmente */
                soma = num1 + num2;

                /* E o texto do objeto saida é alterado para a resposta calculada,
                   usando o método saida.setText. */
                saida.setText("Soma: "+soma);
                }
           });

        setDefaultCloseOperation ( JFrame.EXIT_ON_CLOSE );
        setSize ( 350, 80 );
        setResizable ( false );
        setVisible ( true );
        }
    }
```

Assim, temos um aplicativo que realiza todas as operações em uma única janela, como a apresentada abaixo:

O código completo, com comentários, está a seguir:

```
/* Primeiramente definimos o pacote da aplicação */
package calculadora2;

/* Agora especificamos que vamos usar várias bibliotecas
```

```
   dos pacotes java.awt, java.awt.event e javax.swing */
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/* Em seguida é declarada a classe da janela da aplicação, que
   é uma especialização da classe JFrame */
public class JanelaSoAdicao extends JFrame {
    /* Aqui são declaradas as referências para os objetos que
       estarão presentes na janela: dois campos de texto para a
       entrada (entrada1 e entrada2), um botão para executar a
       soma (botaoSoma) e um outro campo de texto para a saída
       do resultado (saida). */
    private JTextField entrada1, entrada2;
    private JButton botaoSoma;
    private JTextField saida;

    /* Também são declaradas as variáveis onde serão realizadas
       as operações, num1 e num2 para armazenar os valores numéricos
       digitados pelo usuário e soma para armazenar o valor a ser
       apresentado como saída */
    private int num1, num2, soma;

    /* O próximo passo é declarar o construtor da classe JaneaSoAdicao
       que nada faz além de construir a */
    public JanelaSoAdicao() {
        /* No caso de janelas derivadas de JFrame e JDialog, a primeira
           instrução deve ser - sempre - uma chamada ao construtor da
           superclasse. Esta chamada está especificando o título da
           janela a ser criada. */
        super("Programa de Adicao");

        /* Agora é declarada uma referência para o painel da janela
           criada, de forma que possamos adicionar coisas neste painel. */
        Container painel;
        /* E é pedido para a janela o valor da referência do painel */
        painel = getContentPane();
        /* Indica-se o layout como FlowLayout, ou seja, uma disposição
           em que os elementos serão colocados lado a lado. */
        painel.setLayout ( new FlowLayout() );

        /* Cria-se o primeiro campo de entrada (com 5 posições)... */
        entrada1 = new JTextField(5);
        /* E coloca-se o mesmo no painel. */
        painel.add(entrada1);

        /* Cria-se o segundo campo de entrada (com 5 posições)... */
        entrada2 = new JTextField(5);
        /* E coloca-se o mesmo no painel. */
        painel.add(entrada2);

        /* Cria-se o botão com o texto "Soma!" */
        botaoSoma = new JButton("Soma!");
        /* E coloca-se o mesmo no painel. */
        painel.add(botaoSoma);

        /* Cria-se o campo de saída (com 10 posições) */
        saida = new JTextField(10);
        /* Marca-se este campo como "não editável" */
        saida.setEditable(false);
        /* E coloca-se o mesmo no painel. */
        painel.add(saida);

        /* Agora será associada uma classe implícita, implementando
           a interface ActionListener, como a executora da ação do
           botão criado anteriormente. */
        botaoSoma.addActionListener( new ActionListener() {
            /* Quando o botão for clicado, o método actionPerformed
               desta classe será chamada. Deve-se, então, redefinir
               esse método para que ele realize as atividades desejadas */
           public void actionPerformed( ActionEvent event ) {
                /* entrada1.getText() retorna o texto que o usuário
                   digitou no campo entrada1. Este texto já é repassado
                   para o método Integer.parseInt, que o traduz em um
                   número e o armazena na variável num1. O procedimento
                   para obter o segundo valor é o mesmo, colhendo a
                   informação em entrada2 e armazenando o resultado em num2. */
                num1 = Integer.parseInt( entrada1.getText() );
                num2 = Integer.parseInt( entrada2.getText() );
```

```
                /* A soma é realizada normalmente */
                soma = num1 + num2;

                /* E o texto do objeto saida é alterado para a resposta calculada,
                   usando o método saida.setText. */
                saida.setText("Soma: "+soma);
                }
            });

        /* Terminada a definição da classe que interpreta o evento do botão,
           termina-se de definir as propriedades da janela */

        /* Indica que, ao apertar o botão fechar, o aplicativo deve sair */
        setDefaultCloseOperation ( JFrame.EXIT_ON_CLOSE );
        /* Ajusta o tamanho da janela */
        setSize ( 350, 80 );
        /* Indica que a janela não pode ser redimensionada */
        setResizable ( false );
        /* Indica para a janela aparecer */
        setVisible ( true );
        }
    }
```

BIBLIOGRAFIA

DEITEL, H.M; DEITEL, P.J. **Java: como programar** - Sexta edição. São Paulo: Pearson-Prentice Hall, 2005.

Bibliografia Complementar:

DIX, A; FINLAY, J; ABOWD, G; BEALE, R. **Human-Computer Interaction.** Edinburg, England: Prentice-Hall, 1997.

SHNEIDERMAN, B. **Designing the User Interface.** 3rd. Ed. Addison-Wesley. Reading, Ma. 1998.

Unidade 11: Programando Swing com o NetBeans

Prof. Daniel Caetano

Objetivo: Construir uma aplicação baseada na classe Jdialog do Java Swing.

INTRODUÇÃO

Na aula passada vimos como construir uma interface simples, do tipo JFrame, usando apenas instruções da linguagem Java. Este tipo de janela é ideal para a janela principal de aplicativos, mas existem muitas outras situações em que construir uma janela da maneira vista pode ser bastante entendiante e ineficiente.

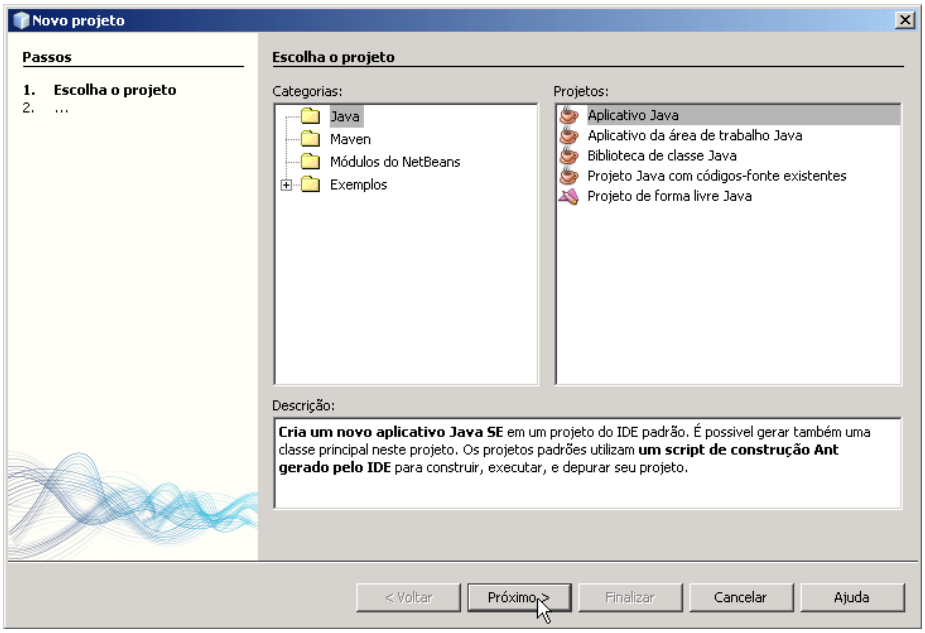
Assim, nesta aula veremos como utilizar o NetBeans para nos ajudar a construir uma aplicação Swing simples, utilizando a classe JDialog, ideal para janelas de configuração das diversas funções de um programa.

Neste exemplo, veremos como construir facilmente uma calculadora com funções bastante básicas. Fica como exercício para o aluno implementar funções mais complexas.

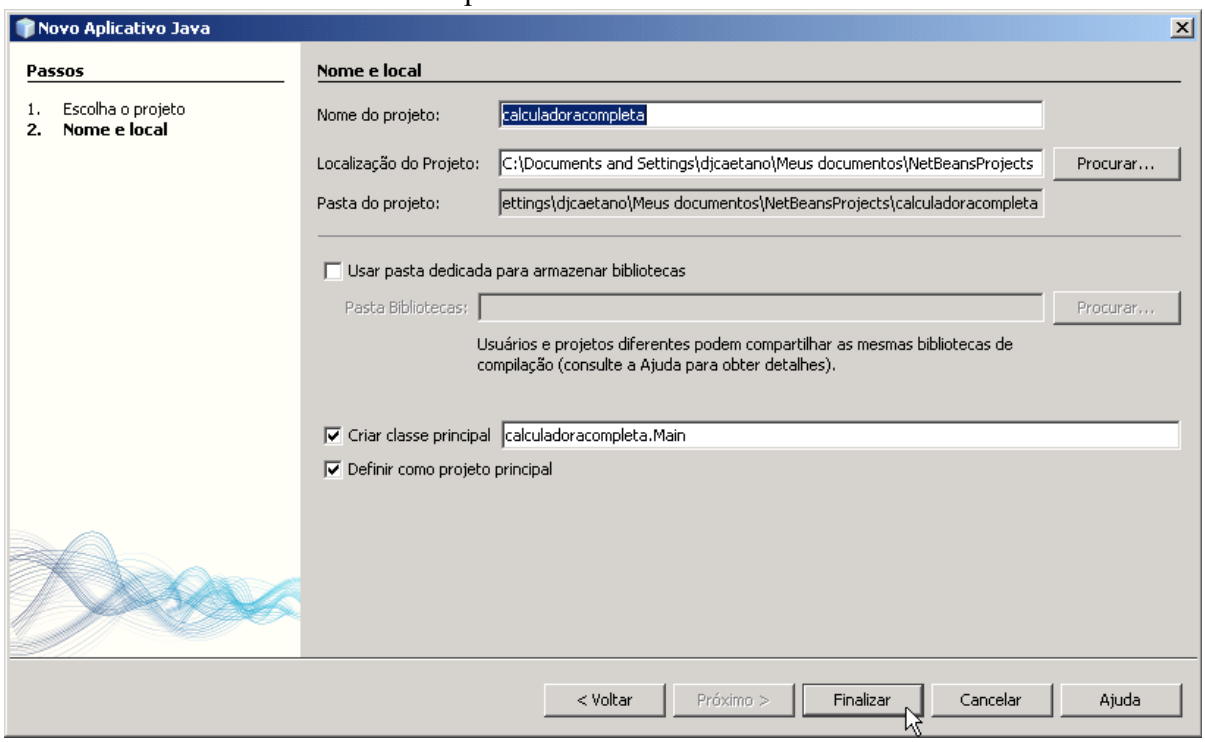
1. CALCULADORA USANDO SWING E NETBEANS

PASSO 1 - A primeira coisa a se fazer é criar um novo projeto. Você pode ir pelo menu (Arquivo > Novo > Projeto) ou clicar no ícone indicado abaixo:

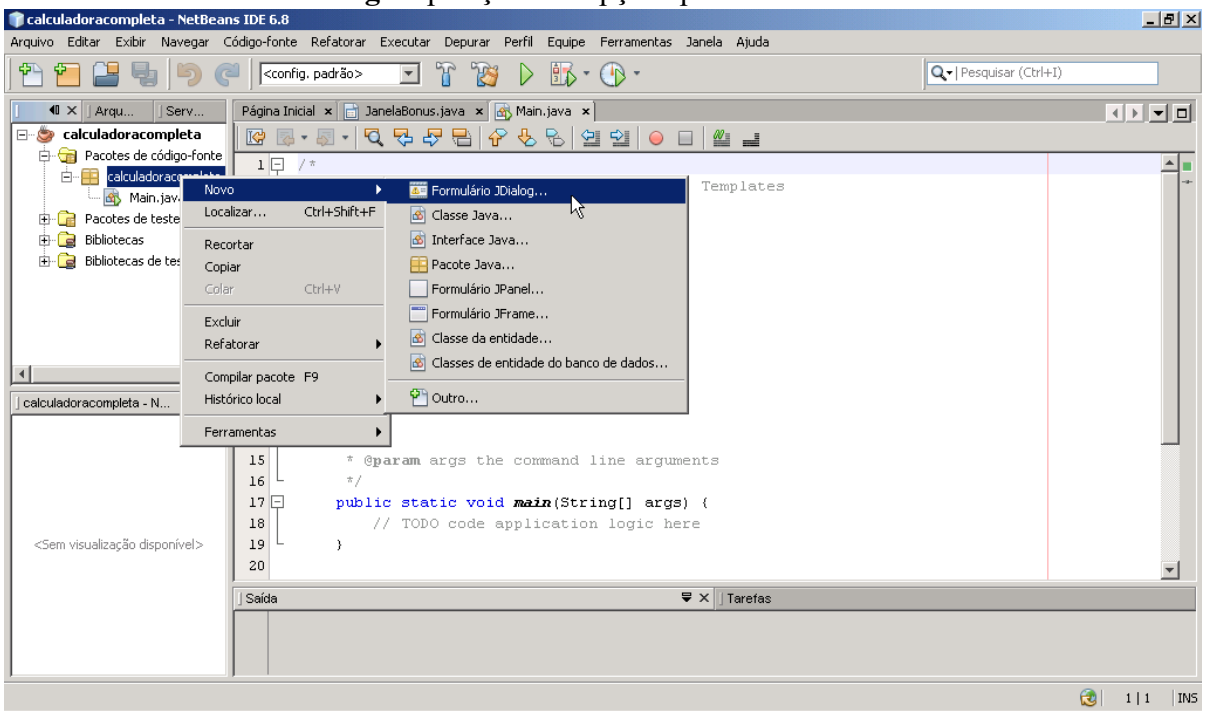
PASSO 2 - Agora é necessário escolher o tipo de projeto. Escolha a categoria "Java" e o tipo de projeto como "Aplicativo Java".



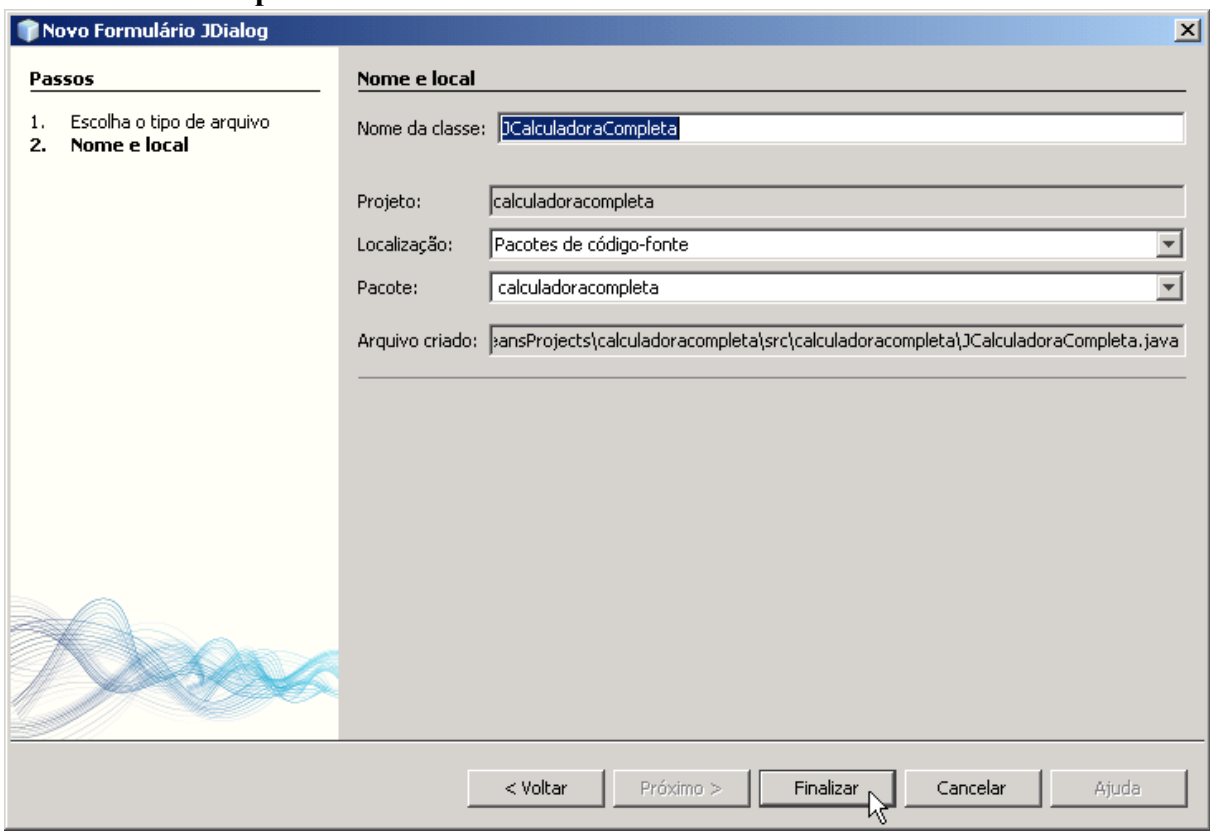
PASSO 3 - Agora é necessário dar um nome ao projeto. Lembre-se: NÃO use caracteres especiais, NÃO use letras maiúsculas, NÃO use espaços. No exemplo abaixo, usamos o nome de "calculadoracompleta".



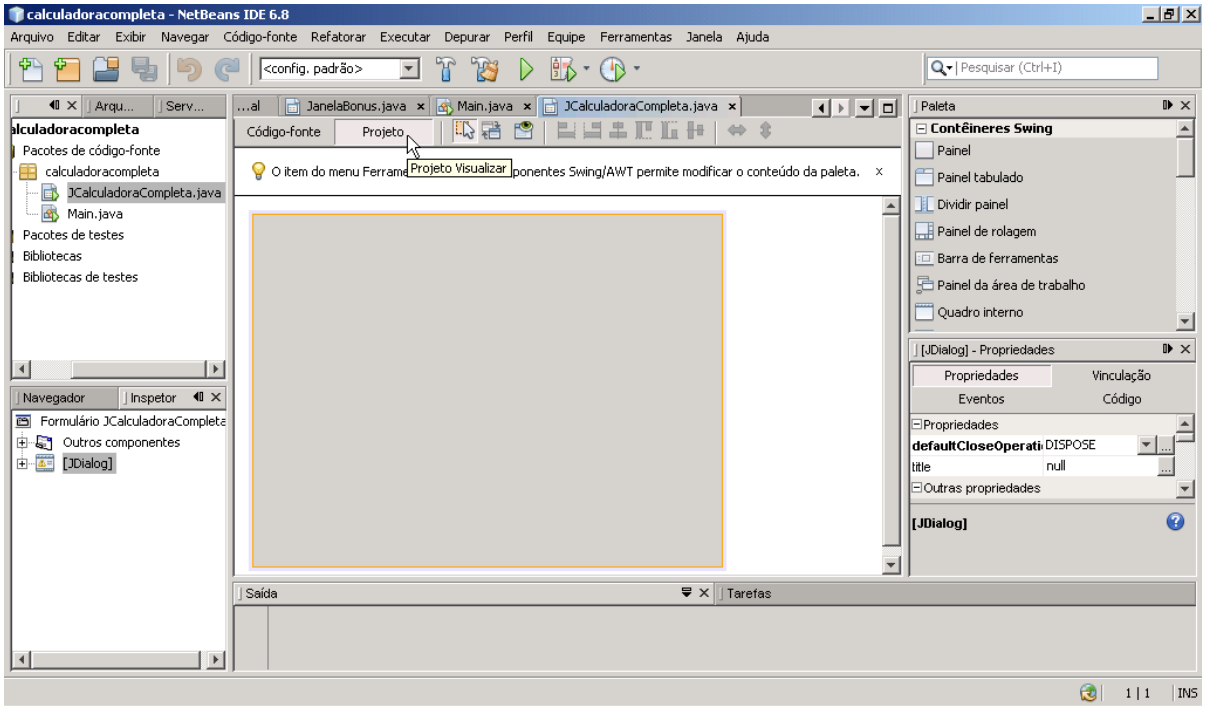
PASSO 4 - A próxima coisa a fazer é criar a classe da janela do tipo JDialog. Clique com o botão direito do mouse no ícone do PACOTE com o nome **calculadoracompleta** (é o que tem o pacote do lado, **não é** o que tem a xícara de café do lado). Selecione as opções **Novo > Formulário JDialog**. A posição das opções pode ser diferente no seu NetBeans.



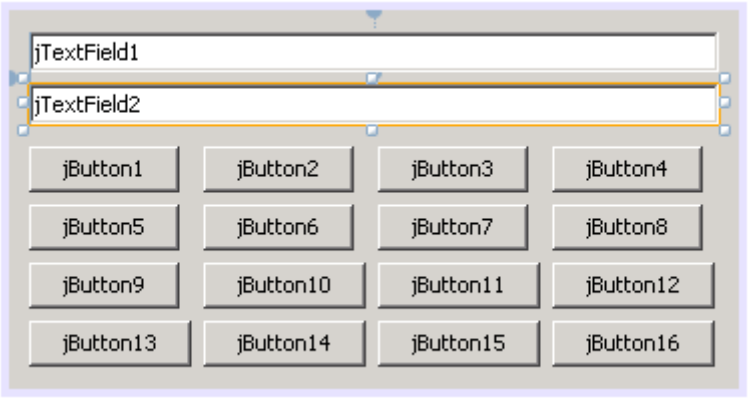
PASSO 5 - Precisamos dar um nome para a janela de nossa aplicação. Como é um nome de classe, lembre-se de começa-lo com letra maiúscula. No exemplo, o nome dado foi **JCalculadoraCompleta**.



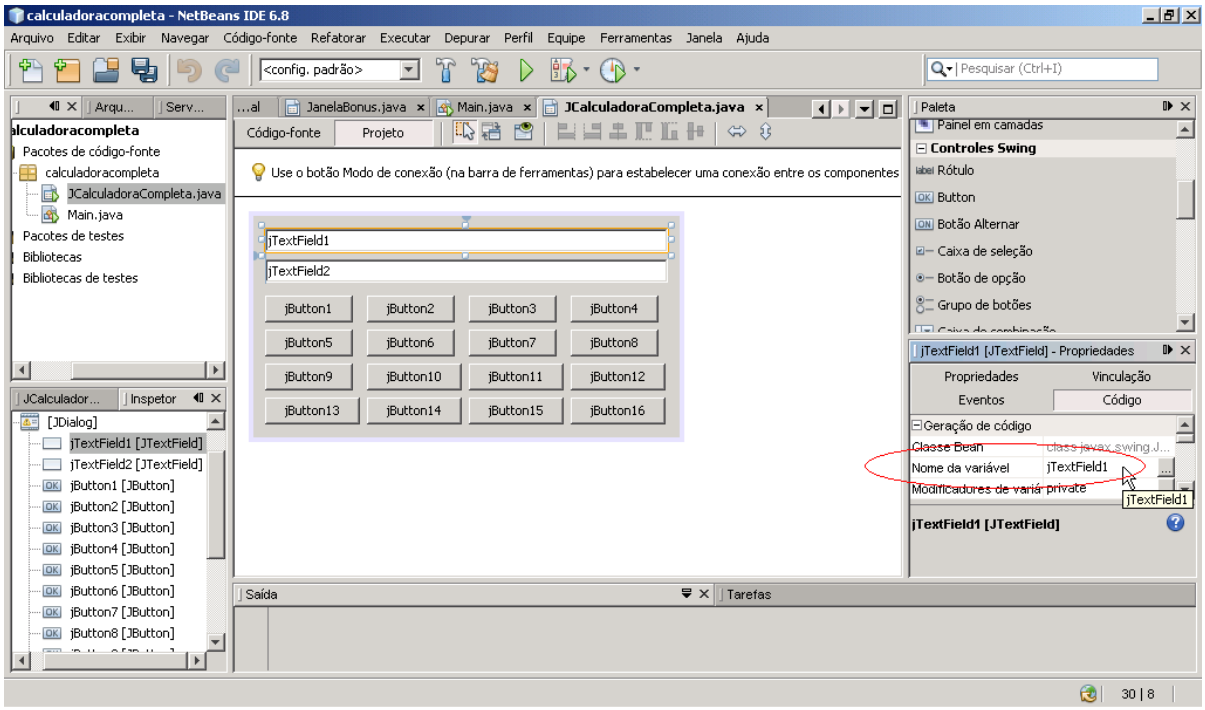
PASSO 6 - Selecionando o botão "Projeto" indicado na figura abaixo, podemos editar o layout da janela, arrastando os itens (botões, caixas de texto etc) da lista da direita para a área da janela.



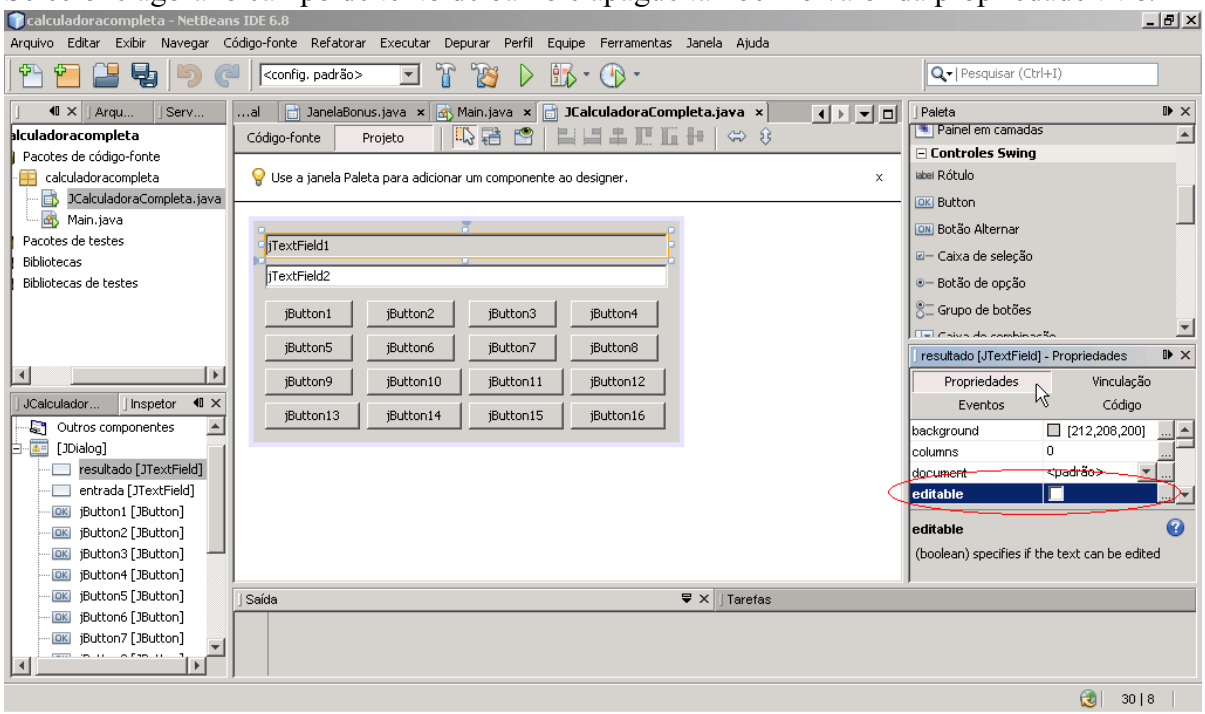
PASSO 7 - Arraste dois campos de texto (JTextField) e 16 botões (JButton) e organize-os conforme indicado abaixo.



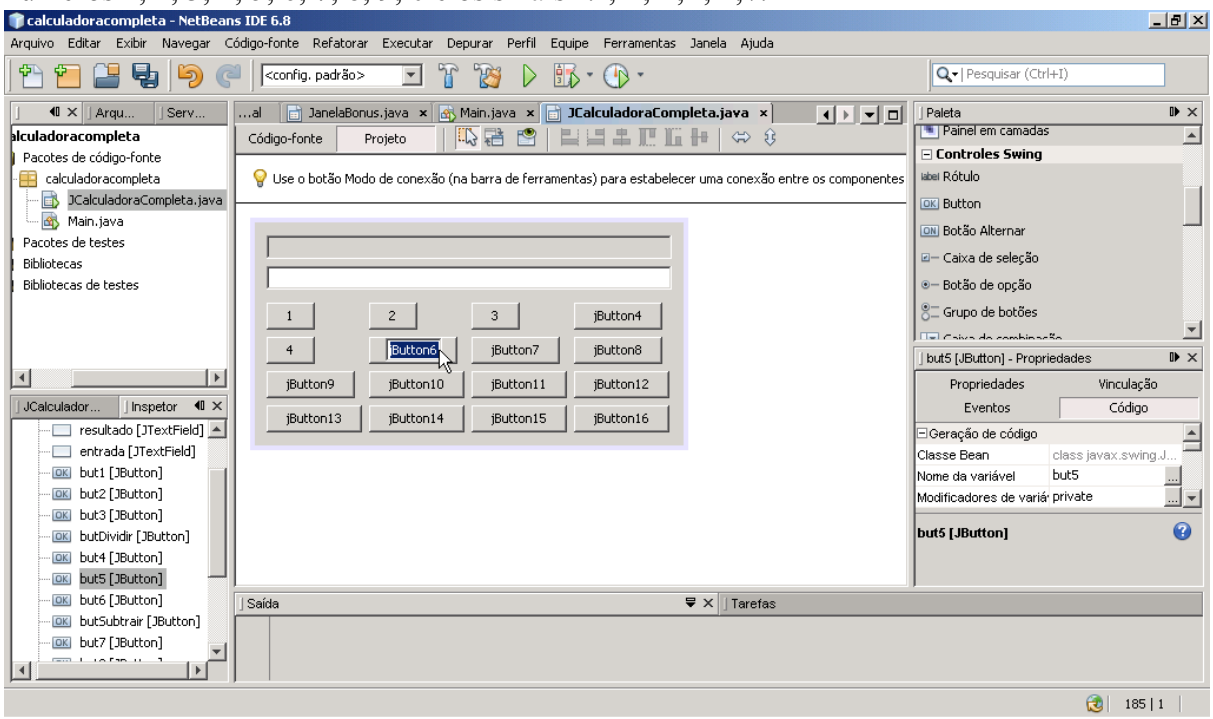
PASSO 8 - Selecione o campo de texto (JTextField) mais do alto (no exemplo, aquele que está com o text "jTextField1"). Clicando no botão "Código" no lado direito, encontre a opção **Nome da variável** e modifique o valor dela para **resultado**. Repita o processo para o jTextField2 e mude o **Nome da variável** para **entrada**.



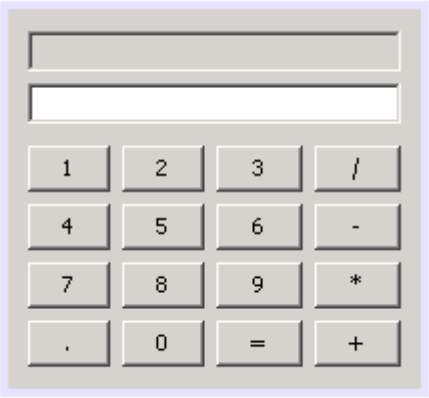
PASSO 9 - Clique agora no botão "Propriedades". Selecione o campo de texto de cima e, na opção **title**, **apague** o texto, deixando a caixa de texto vazia. Procure a opção **editable** e **desmarque-a**, tornando a caixa de texto desligada para edição do usuário. Selecione agora o campo de texto de baixo e apague também o valor da propriedade **title**.



PASSO 10 - Agora dê dois cliques bem lentamente (com uma pausa maior entre os dois cliques) em cada botão para editar seu texto. Modifique-os para que contenham os números 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 e os sinais ".", =, +, -, *, /.



O resultado deve ter a aparência indicada na figura a seguir. Agora, selecione o botão com o número "1" e, no lado direito da janela, clique em "Código" para visualizar as propriedades do botão. Mude o **nome da variável** para **but1**.

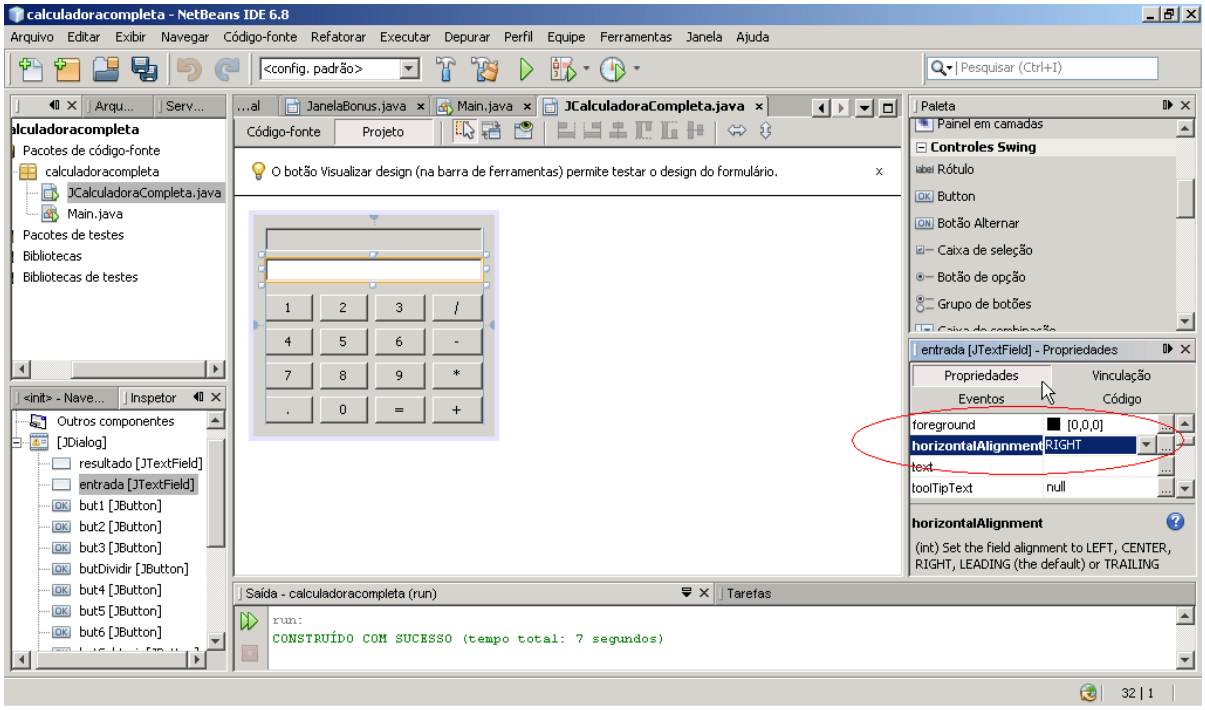


Selecione agora o botão 2 e mude o **nome da variável** para **but2**. Repita esse procedimento para todos os botões. A tabela abaixo indica o nome que deve ser dado a cada botão da janela:

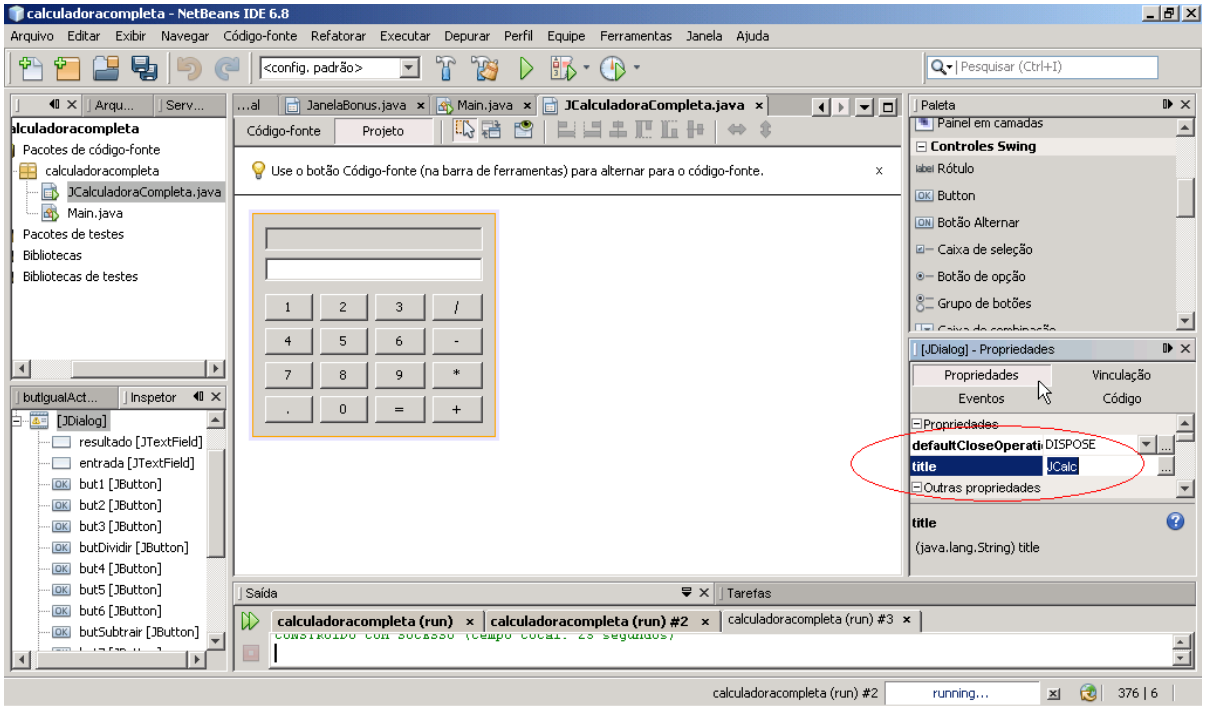
Botão	**Nome da Variável**
1	but1
2	but2
3	but3
4	but4
5	but5
6	but6
7	but7
8	but8
9	but9
0	but0
.	butPonto
=	butIgual
/	butDividir
-	butSubtrair
*	butMultiplicar
+	butSomar

Estes nomes são importantes para que possamos acessar estes elementos a partir de nosso código.

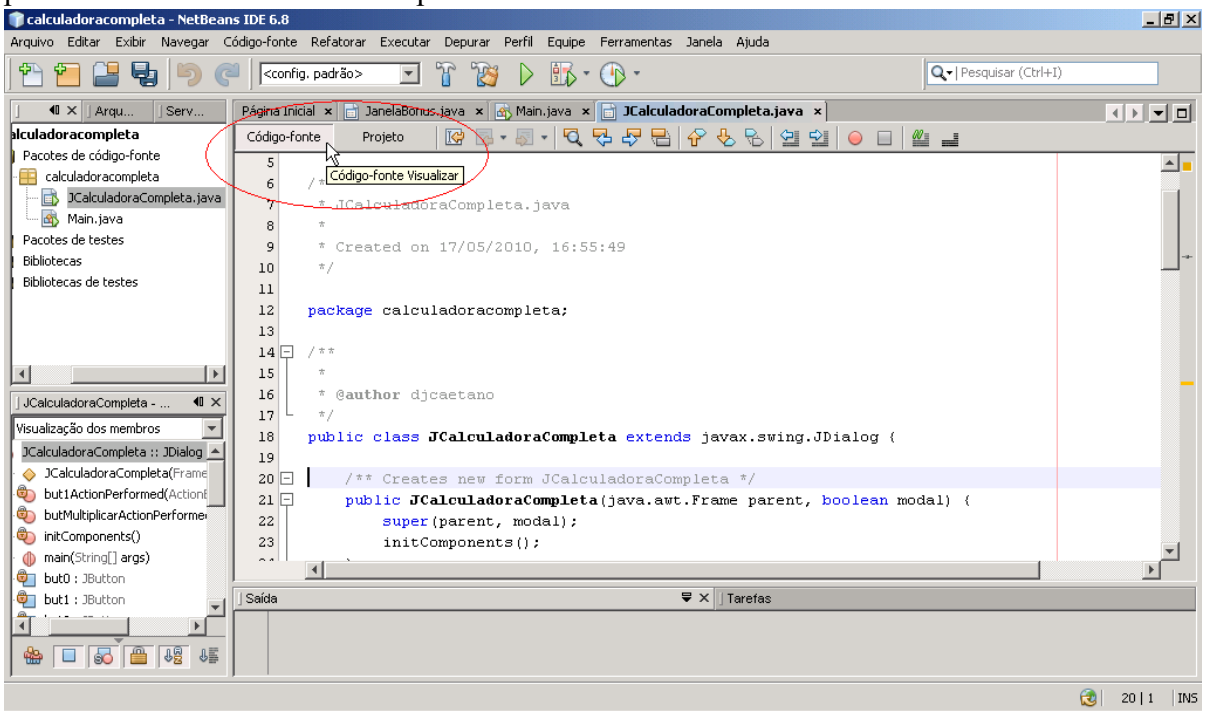
PASSO 11 - Vamos indicar alinhamento à esquerda para os dois campos texto. Selecione o campo texto de cima e, em suas propriedades, procure por **horizontalAlignment** e altere seu valor para "RIGHT", como indicado abaixo. Repita o procedimento para o campo de texto de baixo.



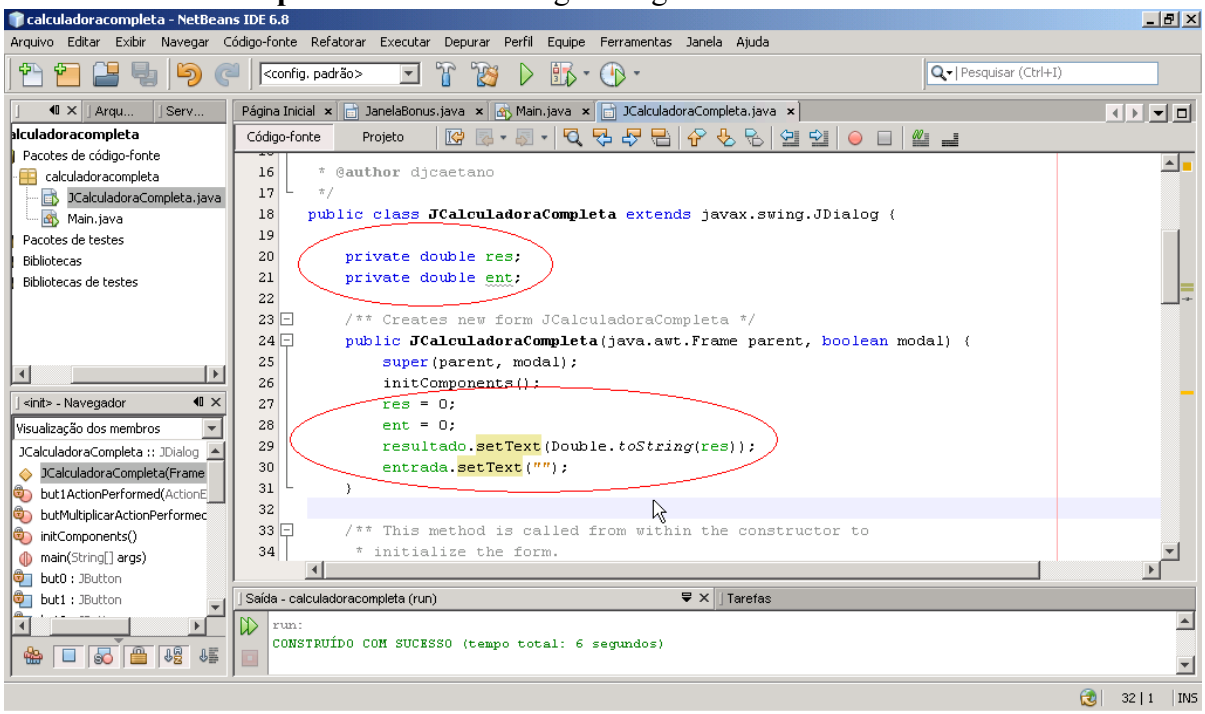
PASSO 12 - Vamos dar um nome para nossa janela agora. Clique na parte do fundo da área cinza da janela. Isso fará aparecer um contorno em toda a janela. Nas propriedades dela, localize **title** e mude seu valor para o nome da aplicação. No exemplo, usamos o nome **JCalc**.



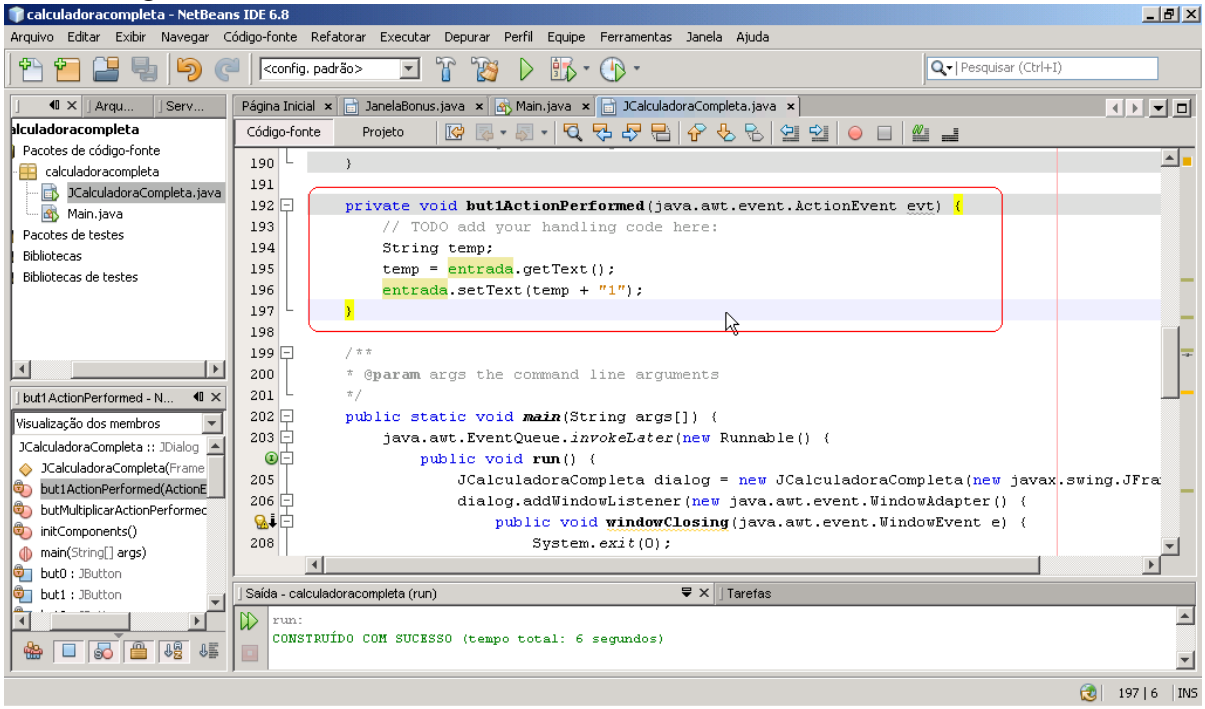
PASSO 13 - Se selecionarmos o botão "Código-fonte" como indicado abaixo, veremos o código de nossa classe **JanelaCalculadoraCompleta**. Observe que o NetBeans já preencheu um bocado de coisas para nós.



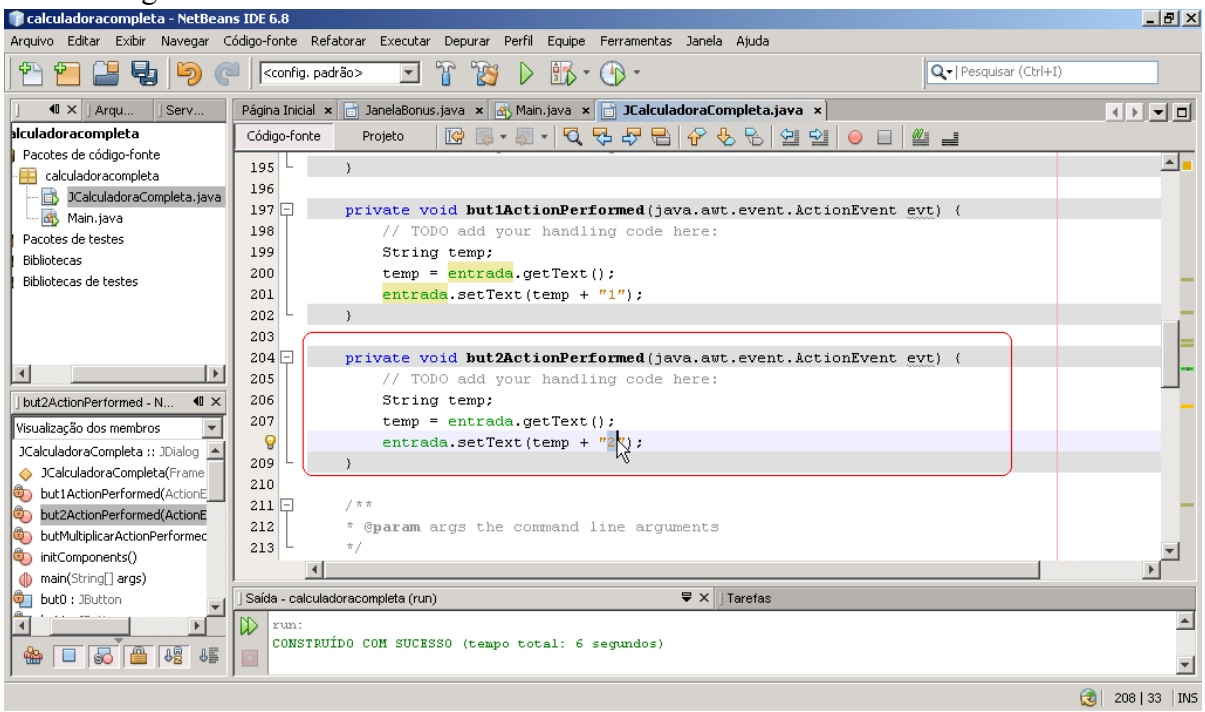
PASSO 14 - Vamos agora criar as variáveis de trabalho de nossa classe. Precisaremos de duas variáveis double: um para guardar os valores digitados pelo usuário (**ent**) e outra para guardar o resultado anterior (**res**). Declare-as como indicado na figura abaixo. Dentro do método construtor, **JanelaCalculadoraCompleta()**, inicialize as variáveis de trabalho com zero e preencha os textos iniciais dos campos: **resultado** deve receber o valor de **res** e **entrada** deve ser **limpado**. Observe o código na figura abaixo.



PASSO 15 - Agora vamos adicionar os códigos dos botões. Para isso, precisamos entrar no modo projeto (botão acima da área de código), onde aparece o layout da nossa janela. Agora, dê um **duplo-clique no botão com o número 1**. Isso vai mudar para o modo de "Código-fonte" já com um método para o evento clique no botão 1, chamado **but1ActionPerformed()**. Insira o código abaixo, que simplesmente pega o texto do campo entrada e "gruda" o número 1 nele.



PASSO 16 - Volte para o modo **Projeto**, e dê um **duplo-clique no botão com o número 2**. Isso vai mudar para o modo de "Código-fonte", criando o método **but2ActionPerformed()**. Insira o código abaixo, que simplesmente pega o texto do campo entrada e "gruda" o número 2 nele.



PASSO 17 - Repita o procedimento para os botões de 3 a 9.

PASSO 18 - Para o botão 0, o código é um pouco diferente: o clique deve ser ignorado se não houver nada preenchido no campo entrada. Observe o código abaixo:

```
279     private void but8ActionPerformed(java.awt.event.ActionEvent evt) {
280         String temp;
281         temp = entrada.getText();
282         entrada.setText(temp + "8");
283     }
284
285     private void but9ActionPerformed(java.awt.event.ActionEvent evt) {
286         String temp;
287         temp = entrada.getText();
288         entrada.setText(temp + "9");
289     }
290
291     private void but0ActionPerformed(java.awt.event.ActionEvent evt) {
292         String temp;
293         temp = entrada.getText();
294         if (temp.isEmpty() == false) entrada.setText(temp + "0");
295     }
296
297     /**
```

PASSO 19 - No caso do botão "." (ponto), temos de verificar o conteúdo de entrada. Se estiver vazio, acrescentamos "0." ao campo entrada. Se o campo não estiver vazio, precisamos verificar se o texto já contém ponto. Se contiver, devemos ignorar o clique, porque um número não pode ter dois pontos. Caso não exista um ponto ainda, acrescentamos o ponto ao texto do campo entrada. Observe o código abaixo:

```
298         temp = entrada.getText();
299         if (temp.isEmpty() == false) entrada.setText(temp + "0");
300     }
301
302     private void butPontoActionPerformed(java.awt.event.ActionEvent evt) {
303         String temp;
304         temp = entrada.getText();
305         if (temp.isEmpty() == false ) {
306             if (temp.contains(".") == false) {
307                 entrada.setText( temp + ".");
308             }
309         }
310         else {
311             entrada.setText("0.");
312         }
313     }
314
315     /**
316     * @param args the command line arguments
```

PASSO 20 - No caso do botão / (divisão), é preciso primeiramente verificar se o campo **entrada** está vazio. Se estiver, ele deve ser considerado zero. Em seguida, verifica-se se o campo de **entrada** tinha valor zero pois, nesta situação, não é possível realizar a divisão entre o valor do campo **resultado** e o valor do campo **entrada** e, portanto, deve-se indicar "Erro!" no campo **resultado**. Caso o valor do campo **entrada** seja aceitável, a divisão será executada, como indicado abaixo.

```
318            }
319        }
320
321        private void butDividirActionPerformed(java.awt.event.ActionEvent evt) {
322            String temp;
323            temp = entrada.getText();
324            if (temp.isEmpty()) temp = "0";
325            ent = Double.parseDouble(temp);
326            if (ent != 0) {
327               res = res/ent;
328               entrada.setText("");
329               resultado.setText(Double.toString(res));
330            }
331            else {
332                entrada.setText("");
333                resultado.setText("Erro!");
334            }
335        }
336
```

PASSO 21 - No caso do botão - (subtração), deve-se subtrair o valor do campo **entrada** do valor do campo **resultado**, como indicado no código abaixo.

```
333            entrada.setText("");
334            resultado.setText(Double.toString(res));
335         }
336         else {
337             entrada.setText("");
338             resultado.setText("Erro!");
339         }
340     }
341
342     private void butSubtrairActionPerformed(java.awt.event.ActionEvent evt) {
343         ent = Double.parseDouble(entrada.getText());
344         res = res - ent;
345         entrada.setText("");
346         resultado.setText(Double.toString(res));
347     }
348
349     /**
350     * @param args the command line arguments
351     */
```

PASSO 22 - No caso do botão * (multiplicação), deve-se multiplicar o valor do campo **entrada** pelo valor do campo **resultado**, como indicado no código abaixo.

```
/** This method is called from within the constructor to
 * initialize the form.
 * WARNING: Do NOT modify this code. The content of this method is
 * always regenerated by the Form Editor.
 */
@SuppressWarnings("unchecked")
Generated Code

private void butMultiplicarActionPerformed(java.awt.event.ActionEvent evt) {
    ent = Double.parseDouble(entrada.getText());
    res = res * ent;
    entrada.setText("");
    resultado.setText(Double.toString(res));
}

private void but1ActionPerformed(java.awt.event.ActionEvent evt) {
    String temp;
    temp = entrada.getText();
```

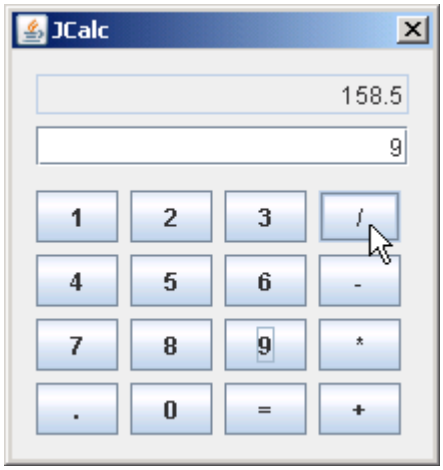
PASSO 23 - No caso do botão + (soma), deve-se somar o valor do campo **entrada** com o valor do campo **resultado**, como indicado no código abaixo.

```
private void butSubtrairActionPerformed(java.awt.event.ActionEvent evt) {
    ent = Double.parseDouble(entrada.getText());
    res = res - ent;
    entrada.setText("");
    resultado.setText(Double.toString(res));
}

private void butSomarActionPerformed(java.awt.event.ActionEvent evt) {
    ent = Double.parseDouble(entrada.getText());
    res = res + ent;
    entrada.setText("");
    resultado.setText(Double.toString(res));
}

/**
 * @param args the command line arguments
 */
public static void main(String args[]) {
```

PASSO 24 - No caso do botão = (igual), deve-se copiar o valor do campo **entrada** para o valor do campo **resultado**, como indicado no código abaixo.

```
entrada.setText("");
resultado.setText(Double.toString(res));
}

private void butIgualActionPerformed(java.awt.event.ActionEvent evt) {
    String temp;
    temp = entrada.getText();
    if (temp.isEmpty()) temp = "0";
    ent = Double.parseDouble(temp);
    res = ent;
    entrada.setText("");
    resultado.setText(Double.toString(res));
}

/**
* @param args the command line arguments
*/
public static void main(String args[]) {
    java.awt.EventQueue.invokeLater(new Runnable() {
```

PASSO 25 - Se tudo foi feito como indicado, o resultado deve ser uma calculadora operante, como a indicada na figura abaixo.



EXERCÍCIOS (OPCIONAIS!)

PASSO 26 - Há várias funções faltantes nesta calculadora, como um botão "+/-", para inverter o sinal do resultado, uma tecla "C" para limpar o resultado, uma tecla "x^2" para calcular o quadrado do resultado, e uma tecla "$x^{1/2}$", para calcular a raiz quadrada do resultado. Será que você consegue implementá-los?

PASSO 27 - Esta calculadora funciona no formato das calculadoras Hewlett Packard, isto é, em notação reversa (digita-se os números e, depois, a operação desejada). Você conseguiria escrever uma que funcione como as calculadoras usuais?

Unidade 12: Programação de Banco de Dados com Java

Prof. Daniel Caetano

Objetivo: Construir uma aplicação Java que interaja com Banco de Dados

INTRODUÇÃO

Nas aulas anteriores vimos um aspecto fundamental em quase todas as aplicações comerciais: a interface gráfica. Agora veremos o outro aspecto presente em 99 entre 100 aplicações comerciais: o banco de dados.

Não é objetivo deste curso apresentar detalhes sobre banco de dados, visto que há disciplinas específicas para isto. O objetivo é apresentar como integrar o Java com um Banco de Dados como o MS SQL e o MySQL, e apresentar estes resultados em um componente visual.

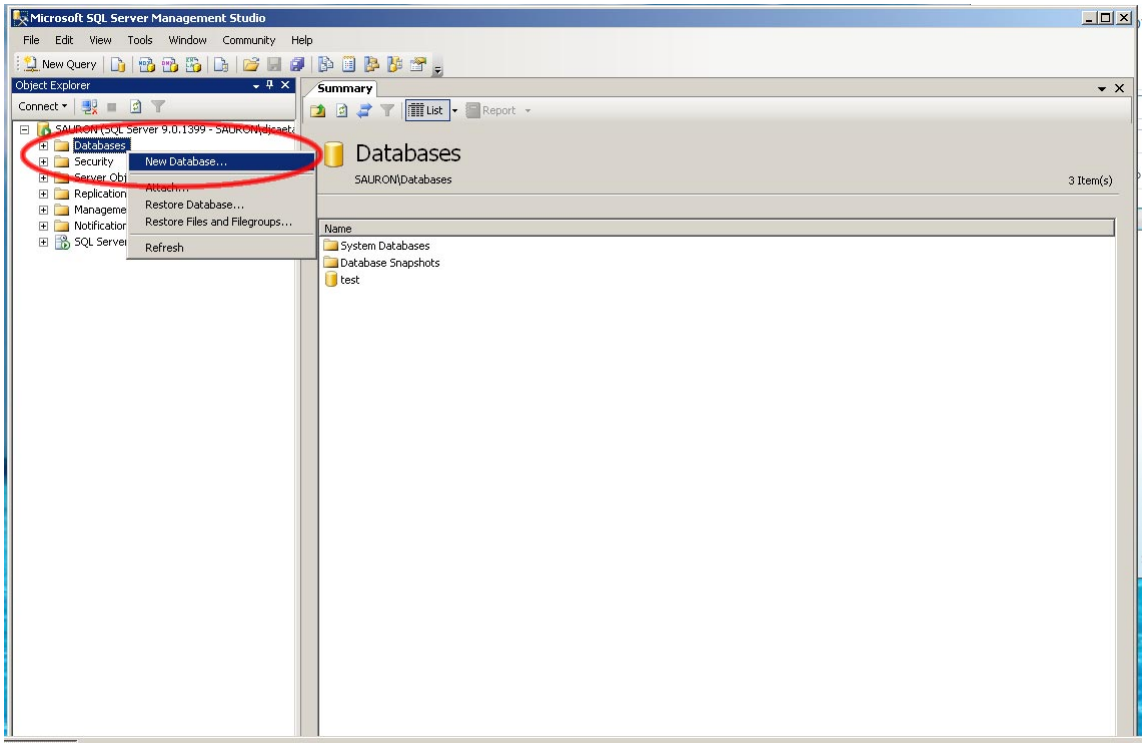
Esta aula será apresentada no formato de tutorial, apresentando primeiramente os passos para a criação de um banco de dados simples no MS SQL Server e, posteriormente, como ler estes dados usando um programa em Java.

1. CONSTRUINDO UM BANCO DE DADOS COM O MS SQL SERVER

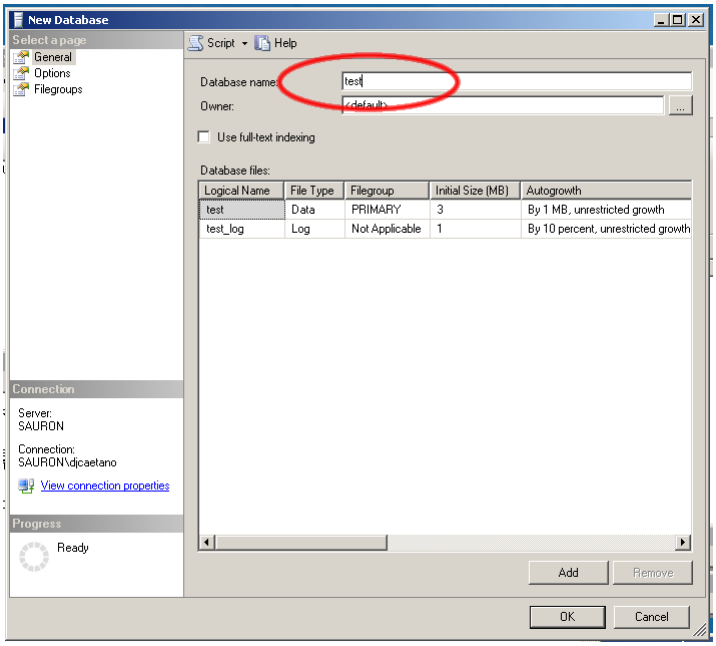
PASSO 1: Abra o aplicativo **SQL Server Management Studio**, disponível em ***Iniciar > Todos os Programas > Microsoft SQL Server > SQL Server Management Studio***. Se tudo estiver corretamente configurado, uma janela similar a apresentada abaixo surgirá. Clique no botão "Conectar".

Caso o sistema exija um password, consulte o administrador do sistema ou o professor.

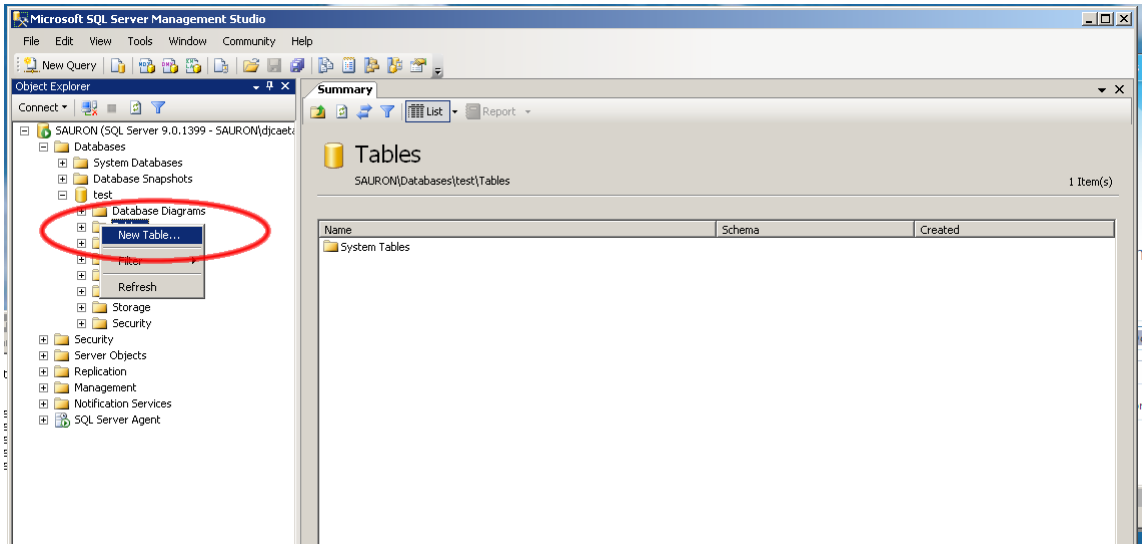
PASSO 2: Uma vez conectado, o primeiro passo é criar um banco de dados. Para isso, clique com o botão direito na opção **Databases** e selecione a opção **New Database...** , conforme indicado abaixo.



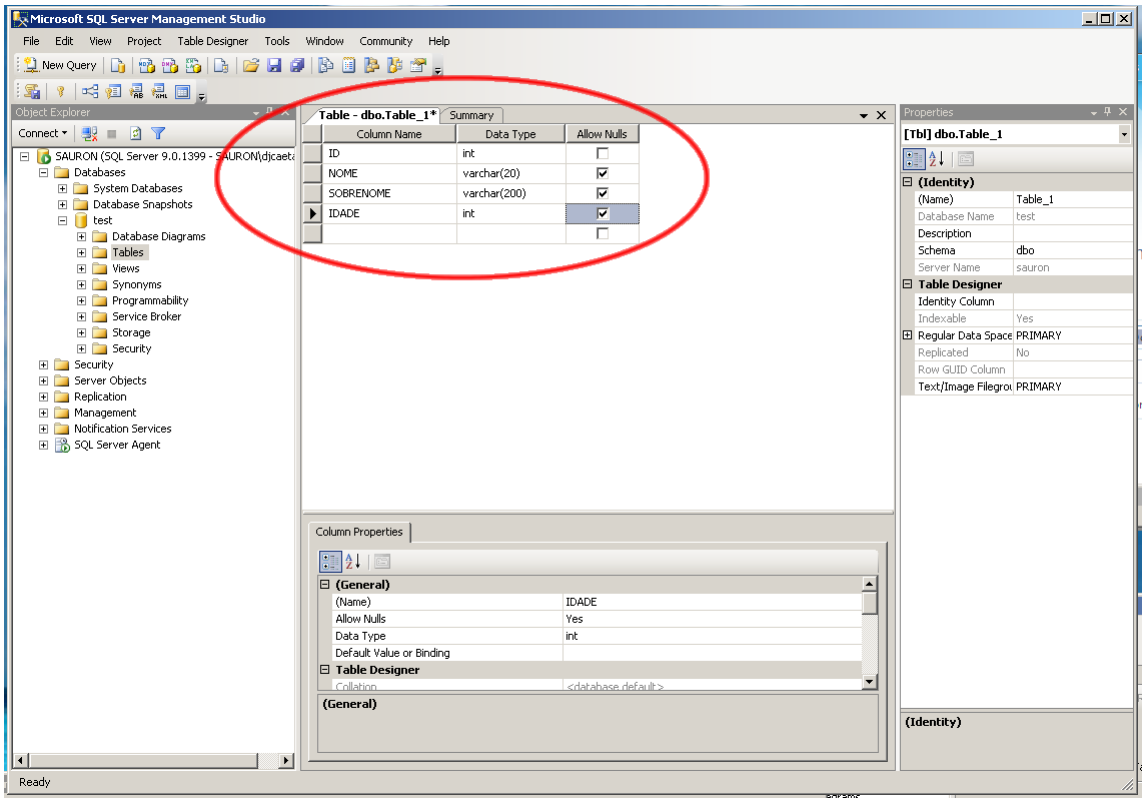
PASSO 3: Na janela que aparece em seguida, dê o nome de **test** para o banco de dados, conforme indicado na figura a seguir. Depois disso, clique no botão **OK**.



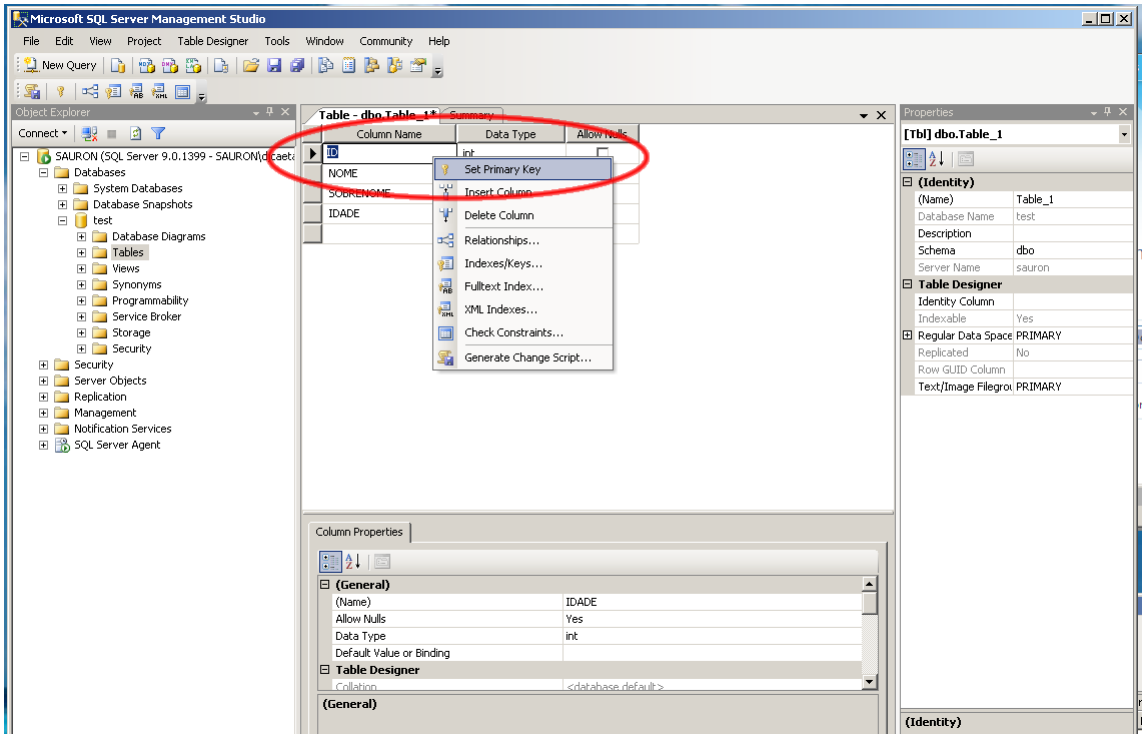
PASSO 4: O próximo passo é a criação de uma tabela (ou relação) neste banco de dados. São as tabelas que armazenam os dados em um banco de dados. Para criar uma nova tabela, clique com o botão direito na opção **Databases > test > <u>Tables</u>** e selecione a opção **New Table...** , como indicado abaixo.



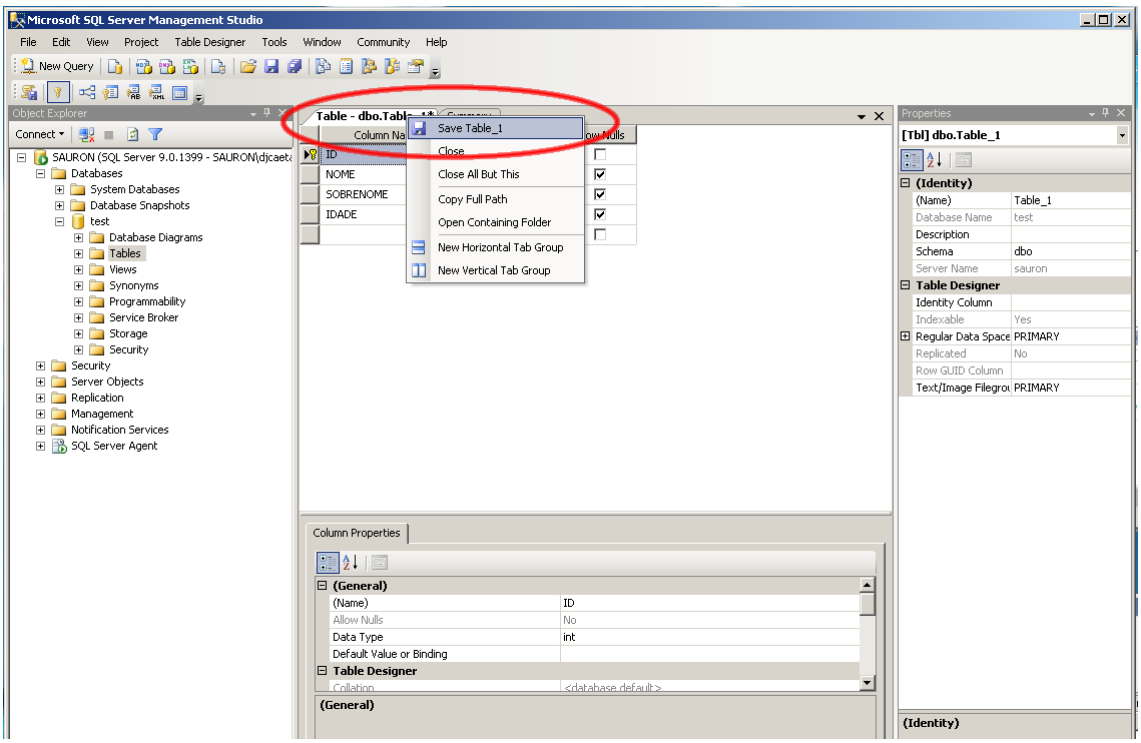
PASSO 5: Vamos agora definir quais dados irão existir em cada uma das colunas da tabela. Configure como indicado na figura abaixo. Serão QUATRO colunas: ID (int, sem Allow Nulls), NOME (varchar(20) com Allow Nulls), SOBRENOME (varchar(200) com Allow Nulls) e IDADE (int com Allow Nulls).



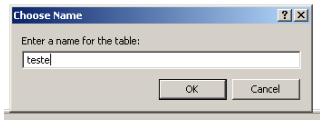
PASSO 6: Vamos agora definir a chave primária de nossa tabela. Clique com o botão direito na linha da coluna chamada ID, e selecione a opção **Set Primary Key**, como indicado abaixo. Uma "chavinha" deverá aparecer ao lado do nome da coluna ID.



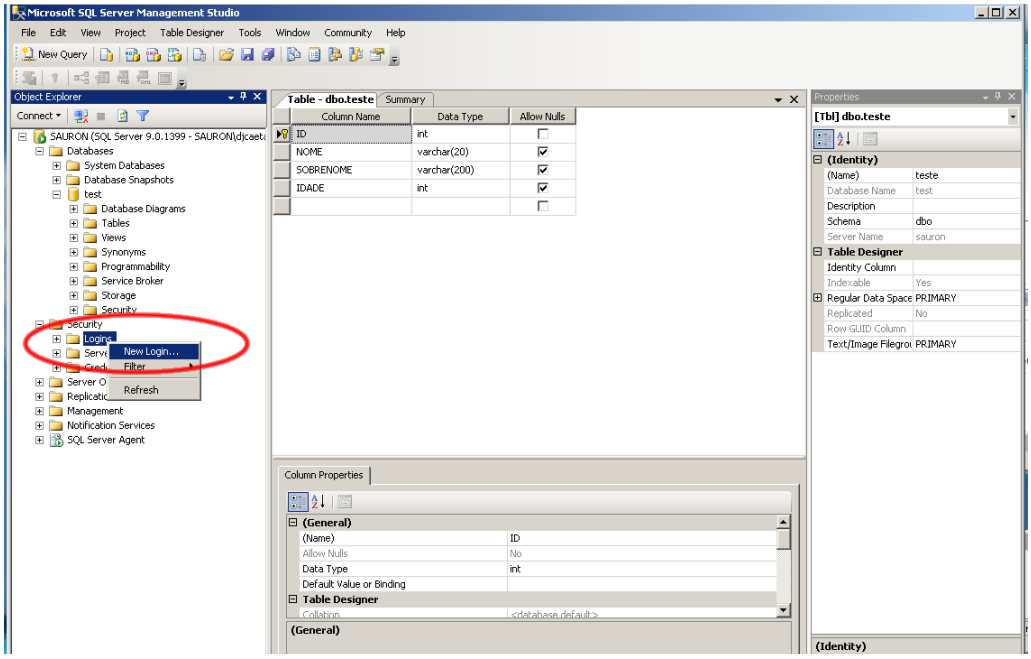
PASSO 7: Agora é o momento de gravarmos a nossa tabela. Clique com o botão direiro do mouse na aba **Table - dbo.Table_1*** e selecione a opção **Save Table_1** conforme indicado na figura abaixo.



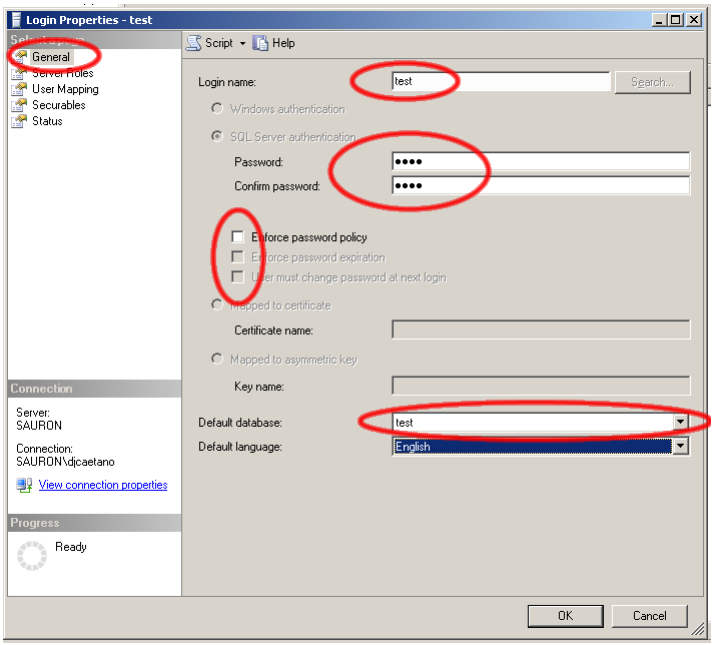
PASSO 8: Na janela que irá aparecer, dê o nome de **teste** para a tabela, conforme indicado na figura a seguir.



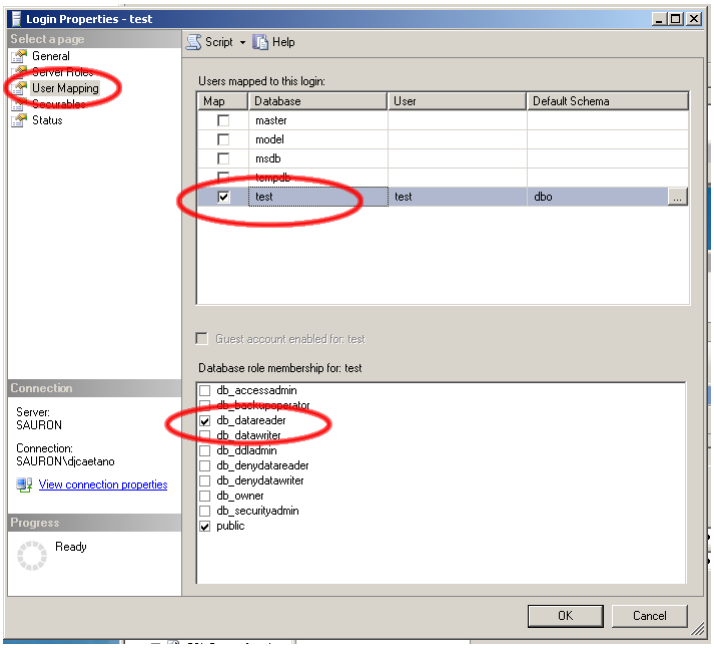
PASSO 9: Agora chegou o momento de criar um usuário para a nossa aplicação, ou seja, um usuário que seja capaz de ler a tabela que já criamos. Clique com o botão direito na opção **Security > Logins** e selecione a opção **New Login...** , como indicado abaixo.



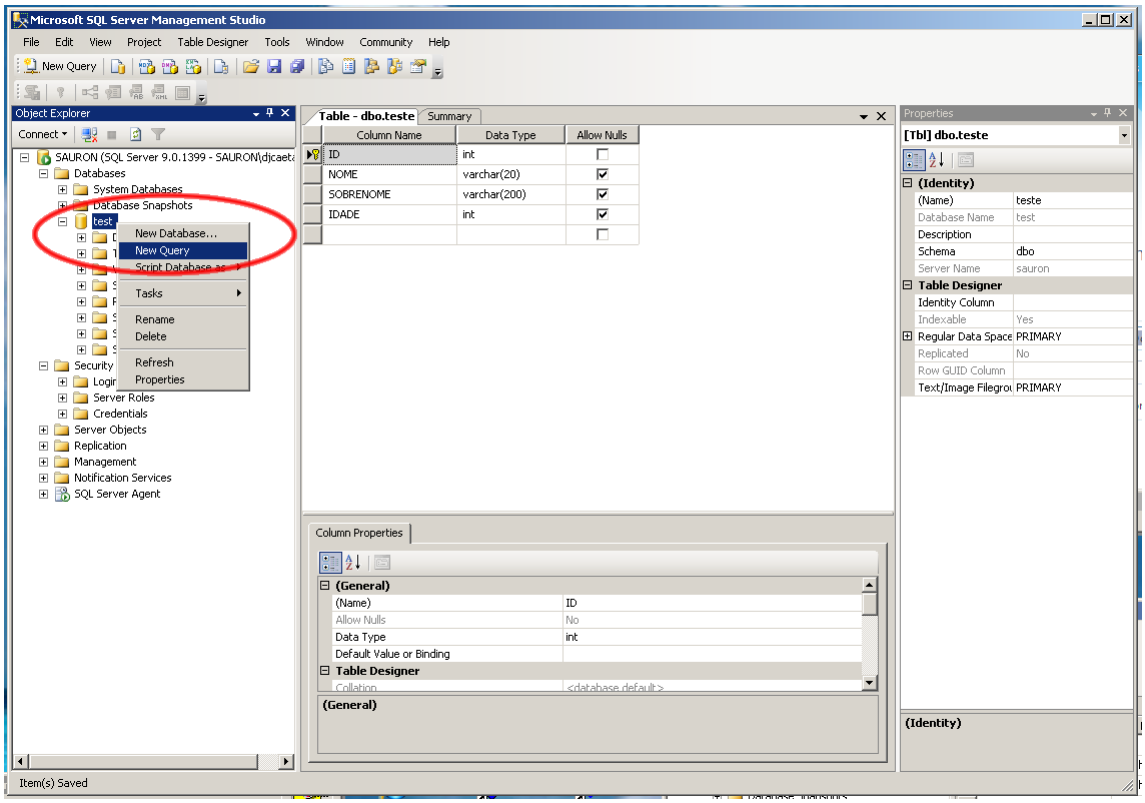
PASSO 10: Na primeira janela que irá aparecer, configuraremos o login do usuário. Dê o nome **test**, configure as duas senhas como **test**, desligue a opção "**Enforce password policy**" e selecione como Default Database o banco de dados **test** que criamos.



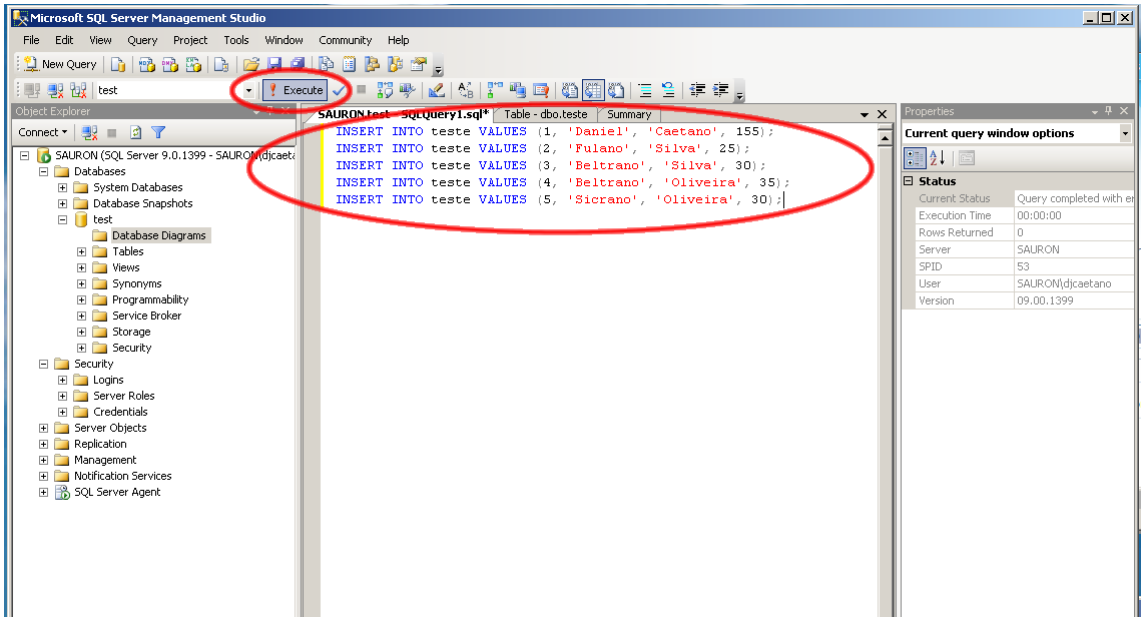
PASSO 11: Selecione agora a opção **User Mapping**, marque o database **test** e, como Database role membership, marque **db_datareader** (e deixe marcado o public). Observe a indicação na janela abaixo.



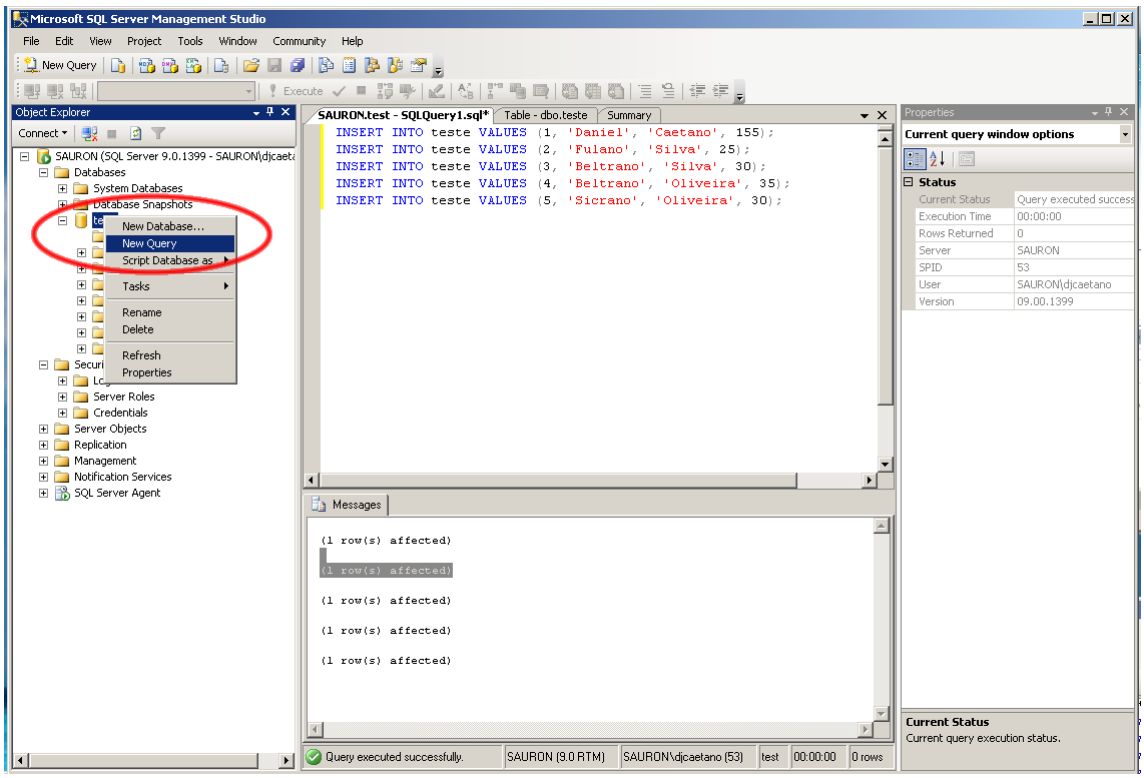
PASSO 12: Agora iremos inserir alguns dados em nossa tabela, para que possamos consultá-los futuramente, pelo programa em Java. Para isso, precisamos criar uma query (busca). Clique com o botão direito no item **Databases > test** e selecione **New Query**.



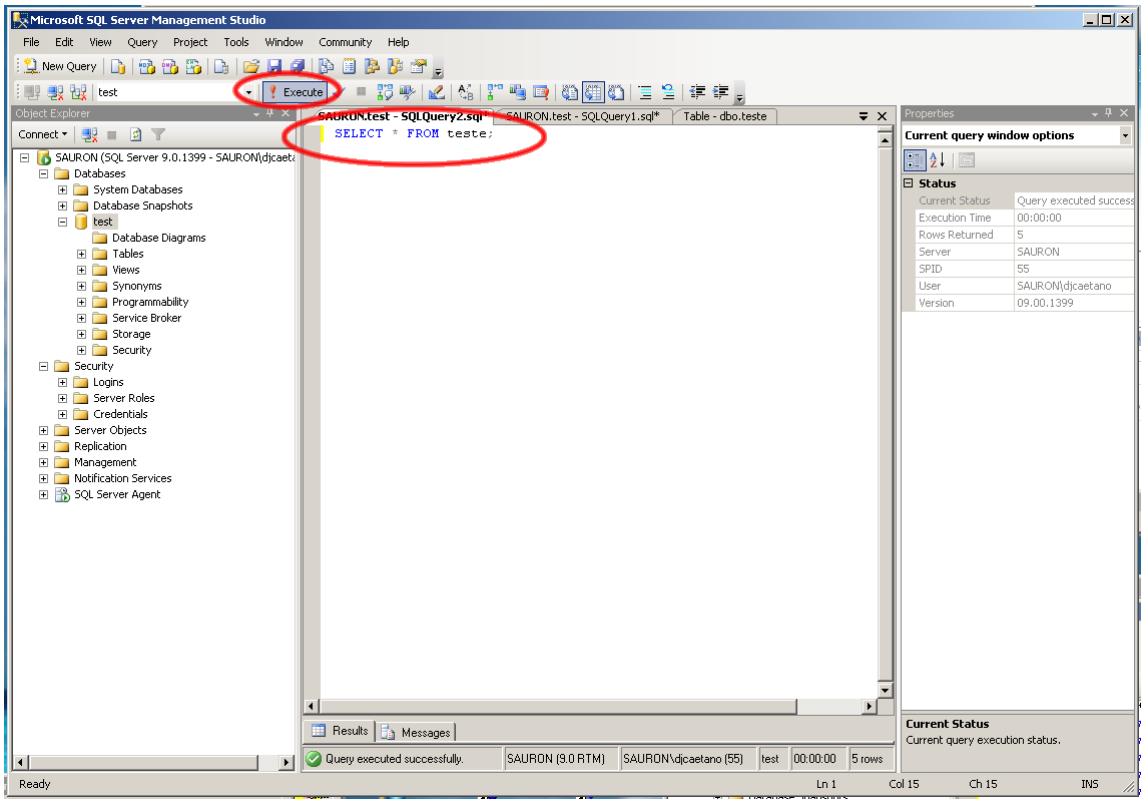
PASSO 13: Na nova tela, iremos inserir 5 linhas em nossa tabela, usando a instrução INSERT. Digite-as como indicado na figura abaixo e, depois, clique na opção **Execute**, conforme indicado.



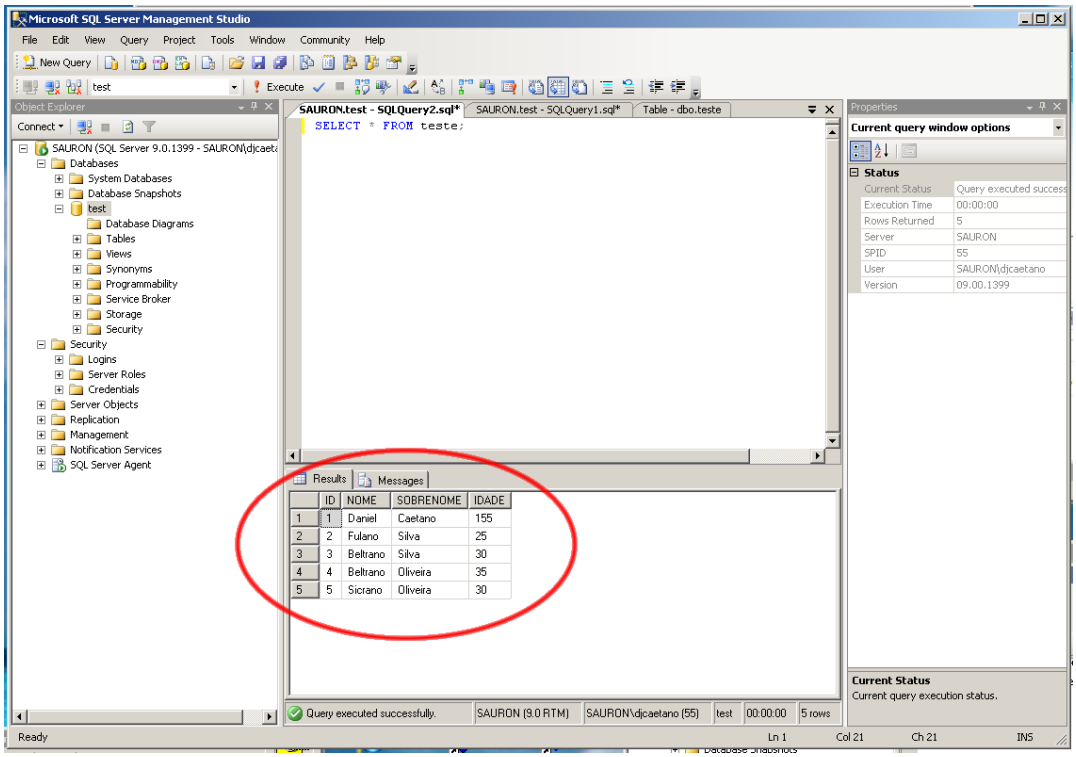
PASSO 14: Se tudo foi feito corretamente e nenhum erro ocorreu, podemos testar a verificar se os dados foram inseridos com sucesso. Para isso, criaremos uma nova query. Clique com o botão direito no item **Databases > test** e selecione **New Query**.



PASSO 15: Na nova tela, iremos realizar uma busca, usando a instrução SELECT. Digite-as como indicado na figura abaixo e clique na opção **Execute**, conforme indicado.



PASSO 16: Como resultado, a janela do SQL Server deve mostrar os dados conforme a figura abaixo.



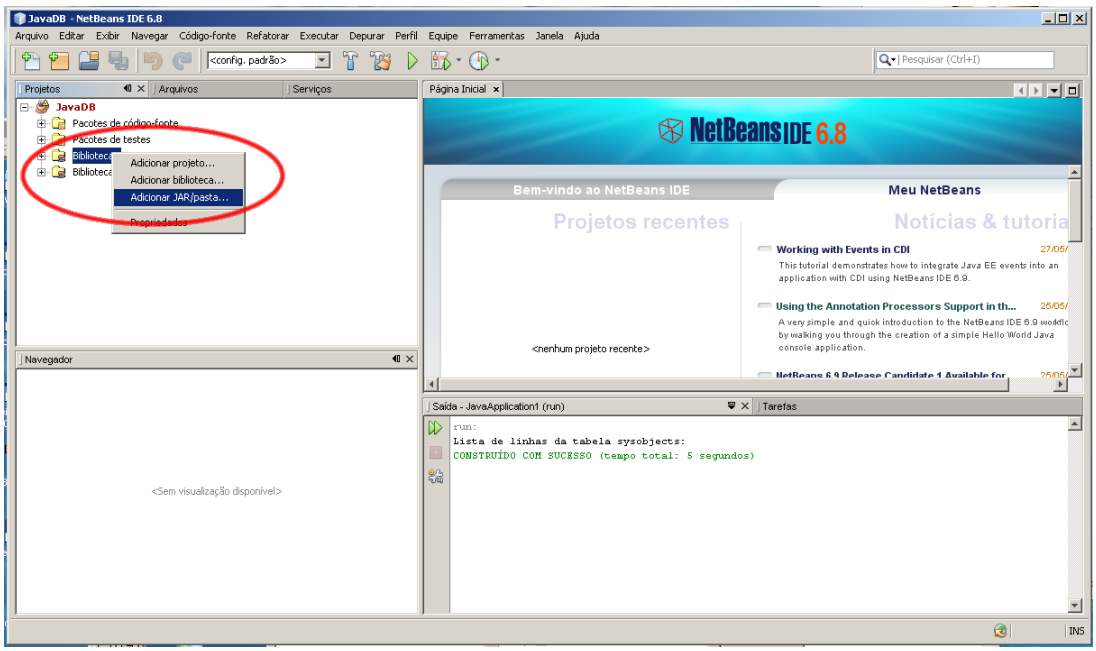
2. CONSTRUINDO UMA APLICAÇÃO JAVA PARA BANCO DE DADOS

PASSO 1: Crie um projeto ***Java*** normal chamado **JanelaDB** no NetBeans.

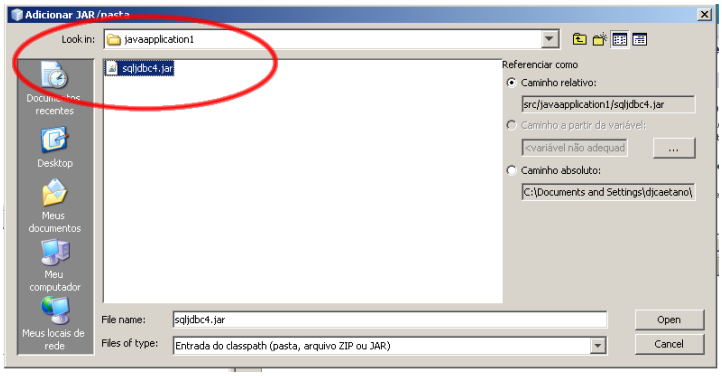
PASSO 2: Como iremos conectar no banco de dados MS SQL Server, precisamos acrescentar em nosso projeto a biblioteca de acesso ao MS SQL Server em nosso ambiente de desenvolvimento. Para isso, primeiramente copie o arquivo **sqljdbcb4.jar**, fornecido pelo professor, para o diretório do seu projeto (provavelmente em **Meus Documentos > My NetBeans Projects > JavaDB**).

> Nota: caso o Java que você esteja usando seja anterior ao Java 1.6, você deverá usar o arquivo **sqljdbc.jar**, ao invés do sqljdbc4.jar.

PASSO 3: Agora iremos associar este arquivo ao projeto. Para isso, na área de projeto, clique com o botão direito no item **JavaDB > Bibliotecas**, e selecione a opção **Adicionar JAR/pasta...** no menu. Observe na figura abaixo.



PASSO 4: E selecione o arquivo **sqljdbcb4.jar**, fornecido pelo professor e colocado no diretório adequado, conforme indicado na figura.



PASSO 5: Agora, clique com o botão direito no pacote **janeladb** e selecione **Novo > Classe Java**. Dê o nome de **JMain** para esta classe.

PASSO 6: Comecemos criando o código para a classe Main. Na área de projeto, dê um duplo clique no arquivo **Main.java**, que deve estar no pacote janeladb. Isso irá apresentar o código fonte atual na janela da direita. Além dos comentários, o código deve ter os seguintes elementos:

Main.java

```
package janeladb;
public class Main {
   public static void main(String[] args) {
      }
   }
```

PASSO 7: O código que iremos acrescentar na classe Main simplesmente irá criar uma janela do tipo JMain, que ainda iremos programar. Modifique o arquivo **Main.java** para que fique como indicado a seguir.

Main.java

```
package janeladb;
public class Main {
   public static void main(String[] args) {
      Jmain janela = new JMain();
      }
   }
```

PASSO 8: O próximo passo será simplesmente a criação da janela JFrame, com um elemento JTable interno. Para isso, vamos editar o arquivo JMain, clicando duas vezes no nome de arquivo **JMain.java**, na área de projeto. O código dele deve ser o indicado abaixo:

JMain.java

```
package janeladb;
public class JMain {
   }
```

PASSO 9: Vamos começar inserindo os *imports* básicos, conforme indicado abaixo.

JMain.java

```
package janeladb;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.sql.*;

public class JMain {
   }
```

PASSO 10: A próxima coisa que precisa ser feita é a definição que a nossa nova classe estende a classe JFrame, para que possamos configurá-la como a janela de nossa aplicação e, assim, definir seus elementos visuais. Para dizer que **JMain estende JFrame**, faça a indicação abaixo.

JMain.java

```
package janeladb;
```

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.sql.*;

public class JMain extends JFrame {
   }
```

PASSO 11: O NetBeans provavelmente vai reclamar, pois agora precisaremos fazer um construtor (método com o mesmo nome da classe) para a nossa janela. O construtor pode ser programado como indicado abaixo: ele cria apenas configura a janela e cria um elemento JTable dentro.

JMain.java

```
package janeladb;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.sql.*;

public class JMain extends JFrame {
   public JMain() {
      super("Consultando SQL!"); // Cria a janela com este título
      Conteiner painel = getContentPane(); // Pega a área de cliente da janela
      painel.setLayout (new FlowLayout()); // Configura area de cliente como flow
      Jtable result = new JTable(6,4); // Cria tabela result com 6 linhas e 4 colunas
      painel.add(result); // Acrescenta tabela na janela

      setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE); // Fecha progr. no botão fechar
      setSize(400,150); // Define tamanho da janela
      setVisible(true); // torna a janela visível
      }
   }
```

PASSO 12: Vamos criar um método para consultar o banco de dados e preencher a tabela. Esse método será chamado no construtor e terá a seguinte assinatura:

public void readData(JTable tabela)

Note que ele deve receber a tabela a ser preenchida como um parâmetro. Observe a implementação básica a seguir.

JMain.java

```
package janeladb;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.sql.*;

public class JMain extends JFrame {
   public JMain() {
      super("Consultando SQL!"); // Cria a janela com este título
      Conteiner painel = getContentPane(); // Pega a área de cliente da janela
      painel.setLayout (new FlowLayout()); // Configura area de cliente como flow
      Jtable result = new JTable(6,4); // Cria tabela result com 6 linhas e 4 colunas
      painel.add(result); // Acrescenta tabela na janela
      readData(result);
      setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE); // Fecha progr. no botão fechar
      setSize(400,150); // Define tamanho da janela
      setVisible(true); // torna a janela visível
      }
   public void readData(JTable tabela) {
      }
   }
```

PASSO 13: Vamos agora acrescentar o código que conecta ao banco de dados.

JMain.java

```
package janeladb;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.sql.*;

public class JMain extends JFrame {
   public JMain() {
      super("Consultando SQL!"); // Cria a janela com este título
      Conteiner painel = getContentPane(); // Pega a área de cliente da janela
      painel.setLayout (new FlowLayout()); // Configura area de cliente como flow
      Jtable result = new JTable(6,4); // Cria tabela result com 6 linhas e 4 colunas
      painel.add(result); // Acrescenta tabela na janela
      readData(result);
      setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE); // Fecha progr. no botão fechar
      setSize(400,150); // Define tamanho da janela
      setVisible(true); // torna a janela visível
      }
   public void readData(JTable tabela) {
      try {
          // Driver SQL instalado?
          Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
        }
      catch(java.lang.ClassNotFoundException e) {
         // Se não achou, mostra erro
         System.err.println("Classe não encontrada: " + e.getMessage());
         }
      try {
         // Define endereço da conexão
         String endereco = "jdbc:sqlserver://localhost:1433;" +
               "databaseName=test;user=test;password=test;";
         // Cria conexação e um objeto de transacao
         Connection conexao = DriverManager.getConnection(endereco);
         Statement transacao = conexao.createStatement();

         // Aqui virá o código de consulta

         // Fim da Conexão
         transacao.close();
         conexao.close();
         }
      catch(SQLException ex) {
         System.err.println("Excessão no SQL: " + ex.getMessage());
         }
      }
   }
```

Neste caso, há dois blocos *try-catch*: o primeiro testa se o driver necessário para a conexão com o banco de dados está instalado; o segundo faz a conexão propriamente dita. Dentro do bloco da conexão, antes de fechá-la, é que serão realizadas as consultas e as atualizações da tabela.

PASSO 14: O próximo passo é realizar uma consulta no banco de dados. Iremos coletar todas as colunas de todas as linhas da tabela, usando a sintaxe

SELECT id, nome, sobrenome, idade FROM teste

O código que implementa esta consulta pode ser visto a seguir.

JMain.java

```
package janeladb;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.sql.*;

public class JMain extends JFrame {
   public JMain() {
      super("Consultando SQL!"); // Cria a janela com este título
      Conteiner painel = getContentPane(); // Pega a área de cliente da janela
      painel.setLayout (new FlowLayout()); // Configura area de cliente como flow
      Jtable result = new JTable(6,4); // Cria tabela result com 6 linhas e 4 colunas
      painel.add(result); // Acrescenta tabela na janela
      readData(result);
      setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE); // Fecha progr. no botão fechar
      setSize(400,150); // Define tamanho da janela
      setVisible(true); // torna a janela visível
      }
   public void readData(JTable tabela) {
      try {
         // Driver SQL instalado?
         Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
         }
      catch(java.lang.ClassNotFoundException e) {
         // Se não achou, mostra erro
         System.err.println("Classe não encontrada: " + e.getMessage());
         }
      try {
         // Define endereço da conexão
         String endereco = "jdbc:sqlserver://localhost:1433;" +
               "databaseName=test;user=test;password=test;";
         // Cria conexação e um objeto de transacao
         Connection conexao = DriverManager.getConnection(endereco);
         Statement transacao = conexao.createStatement();

         // Define a busca
         String query = "SELECT id, nome, sobrenome, idade FROM teste";
         // Executa busca e pega resultado
         ResultSet resultado = transacao.executeQuery(query);
         // Colhe informações sobre resultado (nome das colunas, número de linhas etc)
         ResultSetMetaData infoResultado = resultado.getMetaData();

         // Fim da Conexão
         transacao.close();
         conexao.close();
         }
      catch(SQLException ex) {
         System.err.println("Excessão no SQL: " + ex.getMessage());
         }
      }
   }
```

PASSO 15: Agora iremos colocar a informação dos nomes das colunas na primeira linha da tabela, conforme indicado a seguir

JMain.java

```
package janeladb;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.sql.*;

public class JMain extends JFrame {
   public JMain() {
      super("Consultando SQL!"); // Cria a janela com este título
      Conteiner painel = getContentPane(); // Pega a área de cliente da janela
      painel.setLayout (new FlowLayout()); // Configura area de cliente como flow
      Jtable result = new JTable(6,4); // Cria tabela result com 6 linhas e 4 colunas
      painel.add(result); // Acrescenta tabela na janela
      readData(result);
      setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE); // Fecha progr. no botão fechar
      setSize(400,150); // Define tamanho da janela
```

```
      setVisible(true); // torna a janela visível
      }
   public void readData(JTable tabela) {
      try {
         // Driver SQL instalado?
         Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
         }
      catch(java.lang.ClassNotFoundException e) {
         // Se não achou, mostra erro
         System.err.println("Classe não encontrada: " + e.getMessage());
         }
      try {
         // Define endereço da conexão
         String endereco = "jdbc:sqlserver://localhost:1433;" +
               "databaseName=test;user=test;password=test;";
         // Cria conexação e um objeto de transacao
         Connection conexao = DriverManager.getConnection(endereco);
         Statement transacao = conexao.createStatement();

         // Define a busca
         String query = "SELECT id, nome, sobrenome, idade FROM teste";
         // Executa busca e pega resultado
         ResultSet resultado = transacao.executeQuery(query);
         // Colhe informações sobre resultado (nome das colunas, número de linhas etc)
         ResultSetMetaData infoResultado = resultado.getMetaData();

         /* Imprimindo nomes das colunas em cada coluna */
         // Imprime nome da coluna 1 do resultado na linha 0 / coluna 0 da tabela
         tabela.setValueAt(infoResultado(1),0,0);
         // Imprime nome da coluna 2 do resultado na linha 0 / coluna 1 da tabela
         tabela.setValueAt(infoResultado(2),0,1);
         // Imprime nome da coluna 3 do resultado na linha 0 / coluna 2 da tabela
         tabela.setValueAt(infoResultado(3),0,2);
         // Imprime nome da coluna 4 do resultado na linha 0 / coluna 3 da tabela
         tabela.setValueAt(infoResultado(4),0,3);

         // Fim da Conexão
         transacao.close();
         conexao.close();
         }
      catch(SQLException ex) {
         System.err.println("Excessão no SQL: " + ex.getMessage());
         }
      }
   }
```

PASSO 16: Falta apenas o último passo: acrescentar as linhas de informação, colhidas no banco de dados, na tabela, conforme indicado no código a seguir.

JMain.java

```
package janeladb;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.sql.*;

public class JMain extends JFrame {
   public JMain() {
      super("Consultando SQL!"); // Cria a janela com este título
      Conteiner painel = getContentPane(); // Pega a área de cliente da janela
      painel.setLayout (new FlowLayout()); // Configura area de cliente como flow
      Jtable result = new JTable(6,4); // Cria tabela result com 6 linhas e 4 colunas
      painel.add(result); // Acrescenta tabela na janela
      readData(result);
      setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE); // Fecha progr. no botão fechar
      setSize(400,150); // Define tamanho da janela
      setVisible(true); // torna a janela visível
      }
   public void readData(JTable tabela) {
      try {
         // Driver SQL instalado?
         Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
         }
      catch(java.lang.ClassNotFoundException e) {
```

```
         // Se não achou, mostra erro
         System.err.println("Classe não encontrada: " + e.getMessage());
         }
      try {
         // Define endereço da conexão
         String endereco = "jdbc:sqlserver://localhost:1433;" +
               "databaseName=test;user=test;password=test;";
         // Cria conexação e um objeto de transacao
         Connection conexao = DriverManager.getConnection(endereco);
         Statement transacao = conexao.createStatement();

         // Define a busca
         String query = "SELECT id, nome, sobrenome, idade FROM teste";
         // Executa busca e pega resultado
         ResultSet resultado = transacao.executeQuery(query);
         // Colhe informações sobre resultado (nome das colunas, número de linhas etc)
         ResultSetMetaData infoResultado = resultado.getMetaData();

         /* Imprimindo nomes das colunas em cada coluna */
         // Imprime nome da coluna 1 do resultado na linha 0 / coluna 0 da tabela
         tabela.setValueAt(infoResultado(1),0,0);
         // Imprime nome da coluna 2 do resultado na linha 0 / coluna 1 da tabela
         tabela.setValueAt(infoResultado(2),0,1);
         // Imprime nome da coluna 3 do resultado na linha 0 / coluna 2 da tabela
         tabela.setValueAt(infoResultado(3),0,2);
         // Imprime nome da coluna 4 do resultado na linha 0 / coluna 3 da tabela
         tabela.setValueAt(infoResultado(4),0,3);

         /* Loop que imprime dados na tabela */
         int linecount=1; // contador de linha sendo impressa
         while (resultado.next()) { // enquanto houver mais linhas...
            int i1 = resultado.getInt("id"); // pega coluna "id" como inteiro
            String s1 = resultado.getString("nome"); // pega coluna "nome" como string
            String s2 = resultado.getString("sobrenome"); // pega "sobrenome" como string
            int i2 = resultado.getInt("idade"); // pega coluna "idade" como inteiro

            // Coloca dados nas colunas certas da linha
            tabela.setValueAt(i1, linecount, 0);
            tabela.setValueAt(s1, linecount, 1);
            tabela.setValueAt(s2, linecount, 2);
            tabela.setValueAt(i2, linecount, 3);

            // Atualiza contador de linhas
            linecount = linecount + 1;
            }

         // Fim da Conexão
         transacao.close();
         conexao.close();
         }
      catch(SQLException ex) {
         System.err.println("Excessão no SQL: " + ex.getMessage());
         }
      }
   }
```

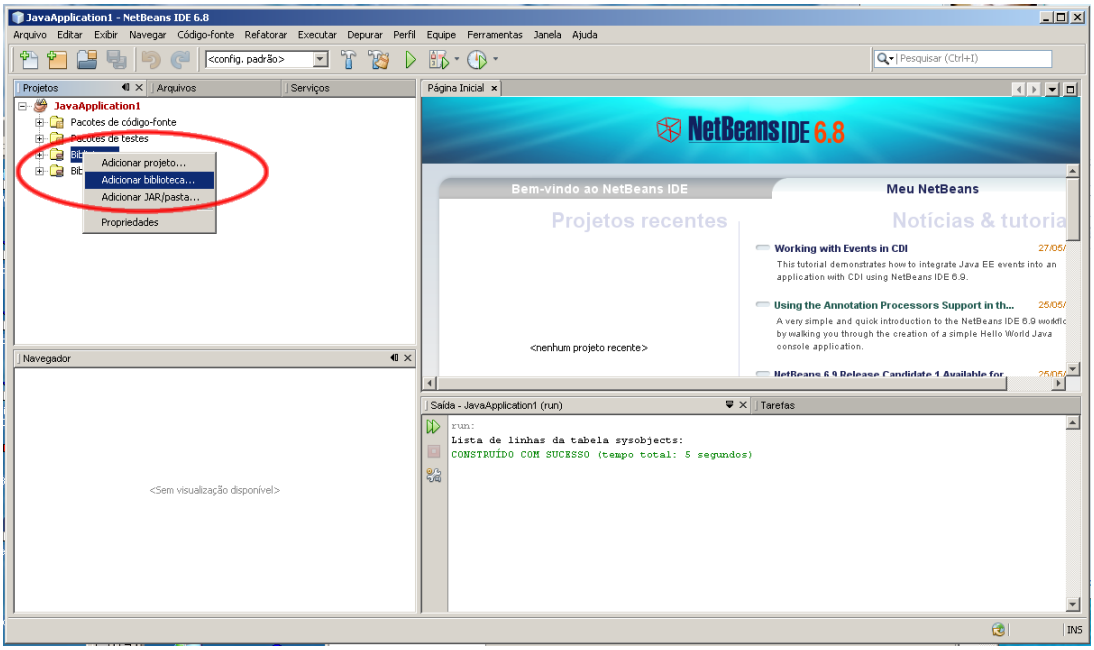
E, com isso, nosso programa está finalizado.

3. CONEXÃO COM O MYSQL

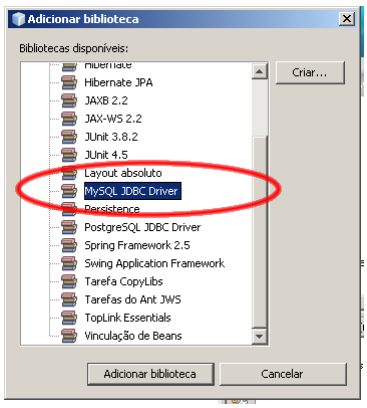
O MySQL é o banco de dados mais acessível, uma vez que é gratuito e seu download é de um tamanho bastante reduzido. Adicionalmente, o driver do MySQL já vem instalado no pacote Java SE + NetBeans, bastando para isso selecioná-lo no momento da criação do projeto. Os primeiros passos para a associação do MySQL ao projeto desenvolvido nesta aula são indicados a seguir.

PASSO 1: Crie um projeto ***Java*** normal chamado **JanelaDB** no NetBeans.

PASSO 2: Como iremos conectar no banco de dados MySQL, precisamos acrescentar em nosso projeto a biblioteca de acesso ao MySQL. Para isso, na área de projeto, clique com o botão direito no item **JavaDB > Bibliotecas**, e selecione a opção **Adicionar Biblioteca...** no menu. Observe na figura abaixo.



PASSO 3: Selecione, agora, o item **MySQL JDBC Driver**, conforme indicado na figura, clicando no botão "Adicionar Biblioteca" em seguida.



PASSO 4: Continue com os passos 5 em diante da seção 2. Ao final do último passo da seção 2, realize as mudanças identificadas no código que está indicado a seguir.

JMain.java

```
package janeladb;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.sql.*;

public class JMain extends JFrame {
   public JMain() {
      super("Consultando SQL!"); // Cria a janela com este título
      Conteiner painel = getContentPane(); // Pega a área de cliente da janela
      painel.setLayout (new FlowLayout()); // Configura area de cliente como flow
```

```
      Jtable result = new JTable(6,4); // Cria tabela result com 6 linhas e 4 colunas
      painel.add(result); // Acrescenta tabela na janela
      readData(result);
      setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE); // Fecha progr. no botão fechar
      setSize(400,150); // Define tamanho da janela
      setVisible(true); // torna a janela visível
      }
   public void readData(JTable tabela) {
      try {
         // Driver SQL instalado?
         Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
         Class.forName("com.mysql.jdbc.Driver");
         }
      catch(java.lang.ClassNotFoundException e) {
         // Se não achou, mostra erro
         System.err.println("Classe não encontrada: " + e.getMessage());
         }
      try {
         // Define endereço da conexão
         String endereco = "jdbc:sqlserver://localhost:1433;" +
               "databaseName=test;user=test;password=test;";
         String endereco = "jdbc:mysql://mysql.caetano.eng.br/estcioalunos";
         // Cria conexação e um objeto de transacao
         Connection conexao = DriverManager.getConnection(endereco);
         Connection conexao = DriverManager.getConnection(endereco,"estacioalunos",
               "estacioalunos");
         Statement transacao = conexao.createStatement();

         // Define a busca
         String query = "SELECT id, nome, sobrenome, idade FROM teste";
         // Executa busca e pega resultado
         ResultSet resultado = transacao.executeQuery(query);
         // Colhe informações sobre resultado (nome das colunas, número de linhas etc)
         ResultSetMetaData infoResultado = resultado.getMetaData();

         /* Imprimindo nomes das colunas em cada coluna */
         // Imprime nome da coluna 1 do resultado na linha 0 / coluna 0 da tabela
         tabela.setValueAt(infoResultado(1),0,0);
         // Imprime nome da coluna 2 do resultado na linha 0 / coluna 1 da tabela
         tabela.setValueAt(infoResultado(2),0,1);
         // Imprime nome da coluna 3 do resultado na linha 0 / coluna 2 da tabela
         tabela.setValueAt(infoResultado(3),0,2);
         // Imprime nome da coluna 4 do resultado na linha 0 / coluna 3 da tabela
         tabela.setValueAt(infoResultado(4),0,3);

         /* Loop que imprime dados na tabela */
         int linecount=1; // contador de linha sendo impressa
         while (resultado.next()) { // enquanto houver mais linhas...
            int i1 = resultado.getInt("id"); // pega coluna "id" como inteiro
            String s1 = resultado.getString("nome"); // pega coluna "nome" como string
            String s2 = resultado.getString("sobrenome"); // pega "sobrenome" como string
            int i2 = resultado.getInt("idade"); // pega coluna "idade" como inteiro

            // Coloca dados nas colunas certas da linha
            tabela.setValueAt(i1, linecount, 0);
            tabela.setValueAt(s1, linecount, 1);
            tabela.setValueAt(s2, linecount, 2);
            tabela.setValueAt(i2, linecount, 3);

            // Atualiza contador de linhas
            linecount = linecount + 1;
            }

         // Fim da Conexão
         transacao.close();
         conexao.close();
         }
      catch(SQLException ex) {
         System.err.println("Excessão no SQL: " + ex.getMessage());
         }
      }
   }
```

BIBLIOGRAFIA

DEITEL, H.M; DEITEL, P.J. **Java: como programar** - Sexta edição. São Paulo: Pearson-Prentice Hall, 2005.