Unidade 1: Introdução à Linguagem Java

Variáveis e Operadores

Prof. Daniel Caetano

Objetivo: Revisar os conceitos da linguagem Java.

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

INTRODUÇÃO

Todo curso baseado na linguagem Java exige conhecimentos mínimos iniciais do aluno. O objetivo desta aula é proporcionar uma uniformização do conhecimento básico, para apoio de todas as aulas subsequentes. Esta aula está dividida nas seguintes seções:

1) Histórico da Linguagem Java
2) Ambientes Java
3) A Linguagem Java e as Máquinas Virtuais
4) Um programa mínimo em Java
5) Variáveis e seus tipos
6) Operadores básicos
7) Controle de Fluxo
8) Métodos
9) Tratamento de Erros

1. 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.

1.1. 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. AMBIENTES JAVA

Como dito anteriormente, como as funcionalidades exigidas por uma aplicação depende de seu tipo e finalidade, a linguagem Java foi dividida em três grandes pacotes, que englobam as principais áreas de utilização da linugagem Java: J2SE, J2ME e J2EE.

J2SE: Java 2 Standard Edition - O J2SE é, por assim dizer, um pacote básico do Java, voltado à construção de aplicações tradicionais, isto é, que são executadas em um computador com boa capacidade de processamento e memória, e executam integralmente (ou quase) na máquina do usuário.

J2ME: Java 2 Micro Edition - O J2ME é uma versão bastante reduzida do Java, com bibliotecas relativamente simplificadas - não existem tipos float e double, por exemplo -, voltada para a construção de aplicações pequenas, isto é, executadas usualmente em dispositivos móveis, como celulares e palmtops, com pouca capacidade de processamento e memória, sendo normalmente executadas integralmente no equipamento do usuário.

J2EE: Java 2 Enterprise Edition - O J2EE é uma versão bastante ampliada do J2SE, incluindo todos os recursos necessários para o uso da linguagem em ambiente de rede e, em especial, a Web, incluindo recursos de persistência de dados, gerenciamento de transações e uma série de outros recursos que facilitam o desenvolvimento de aplicações complexas, com partes executadas em diferentes equipamentos. Essas aplicações usualmente devem interagir com outros sistemas já em funcionamento e são executadas em computadores com grande capacidade de processamento e memória.

2.1. Versões do Java

A nomenclatura do Java traz alguma confusão para os iniciantes. A função desta seção é elucidar algumas destas questões.

Primeiramente, **Java Runtime Environment (JRE)** é um pacote que inclui tudo que se precisa para rodar um programa Java tradicional. Este pacote inclui a **Java Virtual Machine (JVM)** e todo o conjunto de bibliotecas e pacotes da linguagem Java.

O outro pacote disponível, chamado **Java Development Kit (JDK)**, é um pacote mais completo, que inclui o suporte básico ao desenvolvimento Java. Este pacote **inclui** tudo que o JRE inclui, **adicionando** os componentes necessários para gerar programas Java.

Ambos os pacotes existem em sabores SE, EE e ME, referindo-se aos pacotes com componentes J2SE, J2EE e J2ME, respectivamente.

Com relação ao **número** de versão, é preciso entender que, até hoje, o Java não saiu da versão 1.x, isto é, a primeira versão. Por questões comerciais, a Sun/Oracle adotaram nomes que sugerem versões mais avançadas, mas isso só traz confusão aos desenvolvedores. Abaixo segue uma lista com as principais versões de Java:

Versão Real	Nome	Descrição
1.0 a 1.1	Java	Versões iniciais do Java
1.2 a 1.4	Java 2	Adição de um conjunto enrome de componentes básicos
1.5	Java 5	Mais pacotes básicos acrescentados
1.6	Java 6	Otimização para melhoria de desempenho do Java 5

3. COMO FUNCIONA O JAVA

Já foi discutida a capacidade de um programa Java poder ser executado em qualquer lugar, mas como isso ocorre? Como um código feito para um "computador que não existe" consegue rodar em qualquer lugar?

Na verdade, o funcionamento é muito similar ao dos populares emuladores de videogames, que permitem a execução de jogos de PlayStation, GameCube, DreamCast e outros no seu PC. É como se Java fosse a linguagem de um computador antigo e existisse um "emulador" para executar os programas desse computador no PC. Esse "emulador" chama-se **Interpretador Java** ou **Java Virtual Machine (JVM)** e, uma vez reescrito para um novo equipamento, todos os programas Java passam a executar neste 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++.

4. PROGRAMANDO EM JAVA

Quando começamos a aprender uma linguagem, é sempre útil examinar um programa mínimo nesta linguagem. Em Java, o programa mínimo é composto de uma única classe com um único método **main**, armazenado em um pacote. Criando-se o programa no NetBeans, com o nome de projeto de **MeuPrimeiroPrograma**, o código será similar ao que segue:

Main.java

```
// Programa mínimo em Java
package meuprimeiroprograma;

// Classe Principal
public class Main {

   // Método Principal
   public static void main( String args[] ) {
      System.out.println( "Bem vindo ao Java!" );
   }

}
```

Este texto, gravado no arquivo *Main.java*, pode ser compilado clicando no botão que tem um triângulo verde no NetBeans. Apesar de ser um programa mínimo, a linuguagem Java exige que ele contenha todos os elementos de um sistema em Java:

1) Um pacote
2) Uma classe
3) Um método

Sempre que criarmos um projeto no NetBeans, ele criará esses três elementos. Em termos de **organização**, eles funcionam da seguinte forma: Um projeto (de sistema) pode ter vários pacotes (subsistemas). Cada pacote, pode conter várias classes. Finalmente, cada classe pode conter varios métodos.

Não se preocupe, por hora, com o significado preciso do termo "classe". Por enquanto, considere que uma classe é um pequeno programa; assim, um pacote pode conter vários pequenos programas e cada um deles pode conter vários métodos, chamados de "funções" nas disciplinas anteriores.

Todos estes conjuntos fornecem um ótimo meio de organizar o sistema, sob o ponto de vista do programador. No entanto, isso cria um problema para o Java: quando mandarmos o NetBeans executar o nosso código, por onde ele começa a execução? **Qual trecho de código** ele deve executar?

A linguagem Java estabelece que, se nada específico for dito, uma classe é sempre executada pelo seu método principal, o método **main**. Entretanto, isso só resolve o nosso problema se existir apenas uma classe no programa. Neste caso específico, isso é verdade... mas usualmente não será essa a situação!

Para que resolver esta dificuldade, foi estabelecido um padrão: todo programa Java do NetBeans deve ter, no mínimo, uma classe chamada **Main** - e ela deve ser única para todo o projeto. Assim, quando mandarmos o NetBeans executar um programa, ele sempre começará a execução a partir do método **main** da classe **Main**. Por essa razão, o NetBeans **sempre** inicia um projeto criando um pacote com o nome do projeto e, dentro dele, cria uma classe **Main** com um método **main**.

4.1. Regras Gerais de Nomenclatura do Java

Você deve ter reparado que ao longo das explicações são usadas diferentes composições de letras para nomes das coisas em Java. Por exemplo: os nomes das classes começam sempre com letras maiúsculas e os nomes dos métodos sempre com letras minúsculas. Essa é uma convenção adotada por todos os profissionais Java por facilitar muito a leitura e a compreensão do código; adicionalmente, esta convenção é obrigatória nos exames de certificação da Sun/Oracle.

Assim, segue abaixo um breve resumo da convenção:

1) Cada arquivo *.java* pode conter **apenas** uma classe pública, ou seja, uma única classe que pode ser usada por um programa maior. Se, por algum motivo, um arquivo *.java* contiver mais que uma classe, todas as classes excedentes devem ser declaradas como *private*, ficando indisponíveis para classes em outros arquivos.

2) Um arquivo .java deve ter **um nome exatamente igual** ao de sua classe pública. Assim, se a classe pública chama-se **BemVindo**, o nome do arquivo será **BemVindo.java**.

3) Convenção: nomes de pacotes/pastas, arquivos, classes, métodos e campos só podem conter caracteres alfanuméricos (A-Z, 0-9) e underline/underscore (_) e **nunca** devem começar com números. **Nunca** use espaços nem caracteres acentuados e/ou especiais.

4) Convenção: nomes de classes devem sempre começar com letra maiúscula. Nomes de métodos e atributos/variáveis devem sempre começar com letras minúsculas. Se o nome da classe, método ou atributo/variável tiver mais de uma palavra, como "meu campo" ou "minha classe", os espaços devem ser eliminados e a primeira letra de cada palavra deve ser feita maiúscula, como em **meuCampo** e **MinhaClasse**.

5) Convenção: use sempre nomes de pastas, arquivos, classes, métodos e campos que descrevam bem seu significado, mas sempre tão curtos quanto possível.

4.2. Comentários de Código

Os comentários são fundamentais nos códigos Java. A linguagem Java permite comentários de 3 formas básicas: // , /* */ e /** */

1) Comentários de linha (//): o compilador ignorará tudo que estetiver na mesma linha a partir da indicação.

2) Comentários multi-linhas (/* */): o compilador ignorará tudo que estiver entre o marcador /* e o marcador */.

3) Comentários tipo "JavaDoc" (/** */): o compilador ignorará tudo que estiver entre o marcador /** e o marcador */, de forma similar aos comentários multilinhas. Entretanto, é possível colocar algumas "instruções" dentro destes comentários (todas iniciadas com "@"), como @author, que são processadas por um programa especial chamado "JavaDoc", para gerar documentação do seu programa automaticamente.

5. ATRIBUTOS E VARIÁVEIS

O Java é um tipo de linguagem conhecido como "fortemente tipada". Isso significa que ele só consegue trabalhar com variáveis e atributos se souber qual é o tipo desta variável ou atributo. A forma de declarar uma variável é (dados entre colchetes [] são opcionais; dados entre menor/maior <> são obrigatórios):

<tipo da variável> <nome da variável> [= valor] ;

Os tipos básicos existentes são:

boolean	true/false
byte	número de 8 bits, com sinal (-128 a 127)
char	número de 16 bits, sem sinal (0 a 65535)
int	número de 32 bits, com sinal (-2.147484E+09 a +2.147484E+09)
long	número de 64 bits, com sinal (-9.223372E+18 a +9.223372E+18)
float	número de 32 bits, ponto flutuante
double	número de 64 bits, ponto flutuante

Note que números inteiros longos devem ser sufixados com a letra **L** e números de ponto flutuante float devem ser sufixados com a letra **F**. Uma construção como a feita abaixo vai gerar um erro de compilação:

long meuNumero = 10000000000;

A forma correta de especificar é:

long meuNumero = 10000000000L;

O mesmo valento para um número como:

float meuFloat = 3.4F

Neste caso, esquecer o **F** não vai causar erro na compilação, mas pode causar erros de arredondamento.

Abaixo, o código adaptado para apresentar alguns dos tipos de valores:

Main.java

```
// Programa não tão mínimo em Java
package meuprimeiroprograma;

// Classe Principal
public class Main {

   // Método Principal
   public static void main( String args[] ) {
      byte umByte = 120;
      int umInteiro = 10000;
      long umLongo = 10000000000L;
      float umFlutuante = 10.37F;
      double umDuplo = 10.37;

      System.out.println( "Bem vindo ao Java!" );
      System.out.println( "Byte: " + umByte);
      System.out.println( "Inteiro: " + umInteiro );
      System.out.println( "Longo: " + umLongo );
      System.out.println( "Flutuante: " + umFlutuante );
      System.out.println( "Duplo: " + umDuplo );
   }

}
```

5.1. Classes Básicas (Wrappers)

Como a linguagem Java é orientada a objetos, foi considerado conveniente criar classes que envolvessem os tipos de dados básicos. As classes possuem os mesmos nomes dos tipos básicos, mas com a primeira letra em maiúscula:

Boolean	true/false
Byte	número de 8 bits, com sinal (-128 a 127)
Char	número de 16 bits, sem sinal (0 a 65535)
Integer	número de 32 bits, com sinal (-2.147484E+09 a +2.147484E+09)
Long	número de 64 bits, com sinal (-9.223372E+18 a +9.223372E+18)
Float	número de 32 bits, ponto flutuante
Double	número de 64 bits, ponto flutuante

O uso é idêntico ao visto anteriormente:

Byte umByte = 100;

ou

Integer i = 5;

5.2. Tipos de Dados Complexos (OPCIONAL)

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 vetores. Por exemplo: se queremos armazenar uma seqüência de números, fazemos o seguinte:

```
char meuVetor1[] = { 1, 2, 3, 4, 5, 6, 7 };
```

Note que os colchetes definem que é um vetor. Podemos também criar matrizes (vetores bidimensionais), usando uma seqüência dupla de colchetes, como indicado a seguir. Note que podem ser criadas matrizes com mais dimensões, bastando aumentar o número de colchetes.

```
char meuVetor2[][] = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9} };
```

Um outro exemplo: se quisermos armazenar um texto de até 15 caracteres, podemos definir uma variável como um vetor de 16 posições (15 caracteres mais uma indicando o fim do texto):

```
public char meuVetor[16];
```

Note os colchetes e o número interno. Os colchetes indicam que é um vetor, ou seja, uma seqüência de dados do mesmo tipo, e o número representa o máximo de dados qeu vamos guardar neste vetor.

Se declararmos, dentro de um método:

```
char meuVetor[] = {'T', 'e', 'x', 't', 'o'};
```

O conteúdo do vetor será:

meuVetor[0]	meuVetor[1]	meuVetor[2]	meuVetor[3]	meuVetor[4]
T	e	x	t	o

Repare que usamos o número para definir qual posição do vetor queremos acessar. Entretanto, observe que esta forma de definir um vetor não é muito pratica para lidar com texto. Se cada vez que fossemos definir um texto precisássemos definí-lo letra por letra, seria bastante trabalhoso. Mas, como o Java permite que usemos uma **classe** como tipo de dado, a Sun já criou a classe **String**, que podemos usar como tipo de dado para texto. A mesma declaração de texto em um método usando o tipo de classe String seria:

String minhaString = "Texto";

A classe String "encapsula" todas as complexidades envolvidas ao lidar com textos e nos fornece várias facilidades, como veremos mais adiante nas aulas práticas. Como um exemplo final nesta aula, nosso programa mínimo poderia ser reescrito para mostrar um pouco do que foi visto nesta aula:

6. OPERAÇÕES EM JAVA

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

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

Exemplo de uso:

Main.java

```
// Programa não tão mínimo em Java
package meuprimeiroprograma;

// Classe Principal
public class Main {

// Método Principal
public static void main( String args[] ) {
   Integer idade;
   Integer idadeMaisUm;

   idade = 18;
   idadeMaisUm = idade + 1;

   System.out.println(idade);
   System.out.println(idadeMaisUm);
   }

}
```

7. CONTROLE DE FLUXO

Bem, até agora foi falado muito sobre as estruturas do Java, mas muito pouco sobre como construir a lógica dos programas, que são as partes que realmente executam trabalho. Como em qualquer outra linguagem clássica, segue-se uma lógica estruturada seqüencial, que pode ser construída basicamente por estruturas de sequência, seleção e repetição.

Na linguagem Java, a estrutura de sequência é automática: basta colocarmos as instruções que realizam trabalho na ordem correta que o Java automaticamente as executará em sequência. Entretanto, para as estruturas de seleção e repetição existem, de fato, instruções explícitas.

Instruções de Seleção

O Java fornece uma número suficiente de instruções de seleção, que podem ser **aninhadas**, isto é, uma colocada dentro da outra. Estas instruções são as de construção do tipo **if ~ else** e as construções do tipo **switch ~ case**. Muitas vezes é possível substituir uma delas pela outra, mas há casos em que é mais cômodo usar uma ou outra, em favor da legibilidade.

Estrutura if ~ else

A estrutura if ~ else serve quando temos uma decisão simples a fazer, ou seja: existem duas possibilidades e nunca ambas ocorrem ao mesmo tempo: ou isso, ou aquilo. 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
       }
```

As estruturas if~else podem ser aninhadas, ou seja, podemos ter ifs dentro de ifs. Por exemplo: se o aluno passasse caso tivesse nota de prova menor que 50 mas tivesse 70 ou mais nos trabalhos, a estrutura ficaria assim:

```
if (notaDeProva >= 50) {          // se nota de prova maior ou igual a 50...
       aprovado = 1;              // aluno aprovado
       }
else   {                          // caso contrário... (nota de prova < 50)
       if (notaDeTrabalho >= 70) {// se nota trabalho maior ou igual a 70...
              aprovado = 1;       // aluno aprovado
              }
       else {                     // caso contrario (prova<50 e trabalho<70)
              aprovado = 0;       // aluno não aprovado
              }
       }
```

Também é possível executar operações lógicas mais complexas dentro da seleção, usando as lógicas E (&&), OU (||) e NÃO (!) dentro do if. Por exemplo, o if anterior pode ser escrito assim:

```
if (notaDeProva >= 50 || notaDeTrabalho >= 70) { // se nota de prova >= a 50
       aprovado = 1;                             // OU a de trabalho >= 70...
       }                                         // aluno aprovado
else   {                                         // caso contrário... (prova<50
       aprovado = 0;                             // E trabalho < 70)
       }                                         // aluno não 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

Estrutura switch ~ case

A estrutura do tipo switch~case existe para situações em que temos um número finito de tarefas a executar, dependendo de um único valor. Por exemplo, se o programa imprime o estado atual de um MP3 player e os estados possíveis são: 0- parado, 1- tocando, 2- pausa e 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.

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.

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

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.

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.

8. MÉTODOS

Antes de mais nada, é importante ressaltar o **que é** e **para que serve** um método. Um método é a descrição de algo que pode ser feito, uma sequência de instruções para o computador.

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:

```
<tipoDeRetorno> <nomeDoMetodo> ([tipoDaVar <nomeDaVar>][, ...]) {
   }
```

Por exemplo:

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

O campo **tipoDeRetorno** indica o que esse método "responde" a quem o chamou. No caso do exemplo, o método "responde" um número inteiro.

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.

9. TRATAMENTO DE ERROS

Em muitas situações da programação, uma sequência de tarefas pode ser interrompida pela ocorrência de algum tipo de erro. Por exemplo: se solicitarmos ao usuário que digite um número, para que possamos fazer uma conta, e o usuário digitar um texto, haverá um problema para realizar a conta. Para este tipo de situação, existe uma estrutura de tratamento de erros denominada *try ~ catch ~ finally*, que é muito similar às diretivas de controle de fluxo, como *if ~ else*.

A idéia é colocar dentro de um bloco "try" toda a sequência de instruções propensas a erro e, para cada tipo de erro, acrescentar um bloco "catch". O bloco finally existe caso desejemos que algumas operações sejam executadas sempre, ocorra um erro ou não; caso típico é desfazer a conexão com o banco de dados.

Isso significa que, se nenhuma das instruções do bloco "try" retornar erro, o bloco "finally" será executado e o programa terá continuidade. Por outro lado, se alguma das instruções do bloco "try" retornar erro, a execução do bloco "try" será interrompida no ponto em que o erro ocorreu, e o bloco "catch" correspondente ao erro será executado. Ao final do bloco "catch", a execução continuará no bloco "finally" e, finalmente, o programa terá continuidade.

Os blocos "catch" são, por exemplo, ótimas oportunidades de mostrar janelas de erro ao usuário. O código abaixo mostra um exemplo de uso de try-catch-finally:

Main.java

```
// Programa não tão mínimo em Java
package meuprimeiroprograma;

// Classe Principal
public class Main {

// Método Principal
public static void main( String args[] ) {

   try {
      String numeroDigitado = "5";
      double numero = Double.parseDouble(numeroDigitado);
      double resultado = numero / 2.0;
      System.out.println(resultado);
      }
   catch (NumberFormatException ex) {
      System.out.println("Número inválido!");
      }
   finally {
      System.out.println("Programa encerrado!");
      }
   }

}
```

Execute o programa e veja o resultado. Depois disso, modifique o valor inicial de "numeroDigitado" para "5A", ao invés de "5". Execute novamente o programa e veja o que ocorre.

10. PACOTES IMPORTANTES (OPCIONAL)

Muitas vezes foi falado, desde o início do curso, sobre reuso de código e de como isso é facilitado pela orientação a objetos. Em realidade, a maior parte destes benefícios vêm da existência de pacotes de classes prontas com o compilador Java.

É fácil ver que isso é verdade. Um exemplo clássico é a escrita e leitura de dados na tela. O processo para realizar estas atividades é extremamente complexo, mas já vem totalmente pronto na classe **System**, que tem um objeto estático chamado **out** que, por sua vez tem um método chamado **println**, que usamos para imprimir um texto. Por isso podemos simplesmente executar:

```
System.out.println ("Alo mundo");
```

E nosso texto será impresso na tela. Toda a complexidade envolvida em imprimir este texto está escondida dentro desta classe **System** e um objeto chamado **out**. Isso é muito prático, pois se esta classe não existisse, seria necessário que o próprio programador desenvolvesse sua classe de impressão na tela.

Esta classe e várias outras estão definidas num pacote chamado **java.lang**, um pacote básico do Java e cujas classes estão sempre disponíveis para serem utilizadas pelos programadores em qualquer aplicação, sem que nenhum outro passo seja necessário.

Entretanto, existem diversos outros pacotes no Java, que não são automaticamente disponibilizados para o programador, mas que executam funções também muito importantes. Alguns destes pacotes estão relacionados abaixo, com uma breve descrição de seu conteúdo:

Pacote	**Descrição**
java.lang	Classes gerais (Object, String, Exception, Error, etc)
java.util	Clases de utilitários (HashTable, Vector, Enumeration)
java.io	Classes de acesso ao sistema de arquivos (Fluxo de entrada e saída)
java.net	Classes de rede (TCP/IP, Sockets)
java.awt	Elementos básicos de interface gráfica (Abstract Window Toolkit)
java.swing	Suporte avançado a elementos de interface (FrameWindow, etc)
java.applet	Suporte a programas "embutidos", como os usados em navegadores

Excluindo-se o pacote **java.lang** (que é sempre disponível automaticamente), sempre que se desejar usar qualquer classe de um destes pacotes deve-se **importar** tais classes. Pensando no conceito que as classes definidas nestes pacotes são *estrangeiras* com relação ao nosso programa, fica fácil entender porque essas classes precisam ser importados.

Por exemplo, uma das classes comumente importadas em programas modo texto é a classe **Scanner**, do pacote **java.util**. Esta classe permite que um dispositivo (teclado, por exemplo) seja lido (para receber entradas do usuário, por exemplo). Para que seja possível criar um objeto desta classe e posteriormente seus métodos possam ser utilizados, é preciso indicar ao nosso programa que esta classe será importada e de onde ela será importada:

import <pacote>.<nomeDaClasse>

No caso da classe Scanner, que está no pacote java.util, podemos indicar isso da seguinte forma, no início do código do arquivo de uma classe, antes de qualquer outra coisa (fora da classe, portanto):

import java.util.Scanner

A partir de então, a classe Scanner pode ser usada no programa como se ela realmente fosse parte do programa.

Entretanto, quando muitas classes de um mesmo pacote são usadas, pode se tornar incômodo importar todas as classes, uma a uma. Por esta razão, é possível indicar ao Java para importar todas as classes de um pacote, sem distinção. Isso é feito com um * no lugar do nome da classe, na declaração do import. Assim, se for necessário incluir muitas classes do pacote java.util, é possível utilizar a seguinte indicação:

import java.util.*

11. ATIVIDADES

A) Pegue o código Aula1Ex1, coloque na pasta dos projetos NetBeans como indicado pelo professor. Modifique o programa para que realize uma operação de divisão, ao invés de uma operação de multiplicação. Observe o resultado. Ele está correto?

B) Ainda com o código do item A, modifique-o para que opere com os tipos de dados double e Double e veja os resultados. Qual a diferença?

C) Experimente digitar os códigos das páginas 6, 9, 11 e 17. Tente entendê-los e faça modificações para verificar se você realmente entendeu o que eles fazem e como funcionam.

12. 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: Revisando a Programação Swing com o NetBeans

Prof. Daniel Caetano

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

INDRODUÇÃO

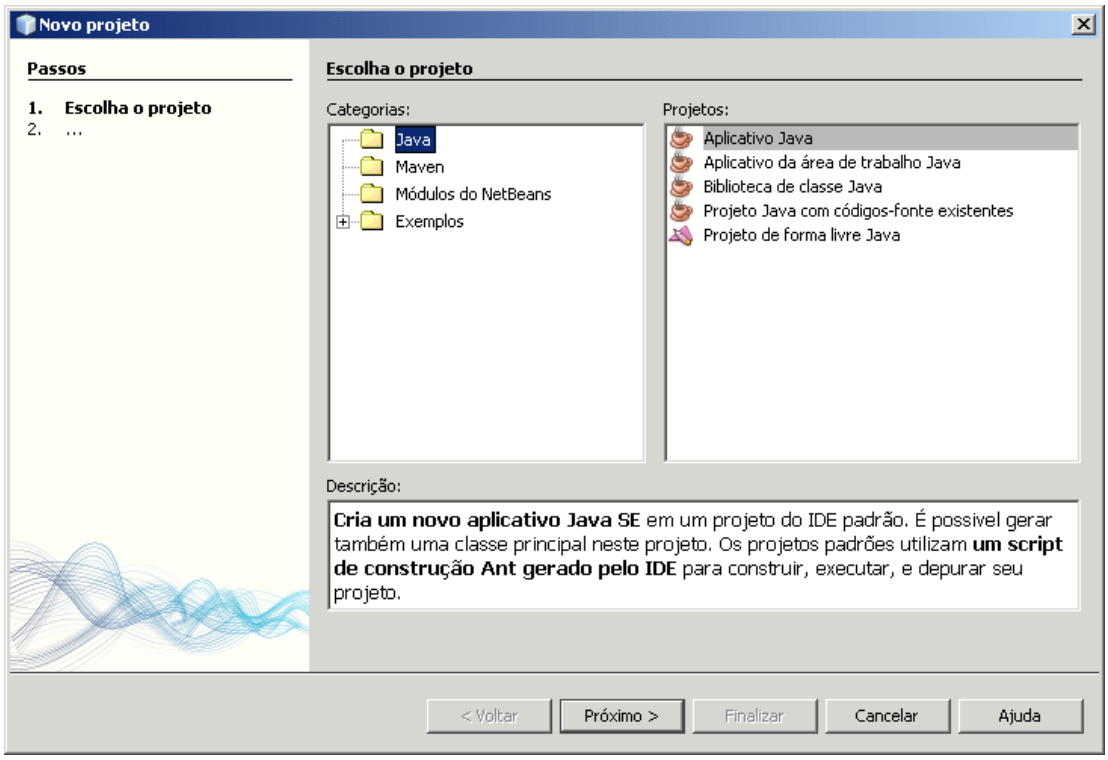
Na aula passada revisamos os principais conceitos do Java. Agora iremos relembrar o uso do NetBeans para criar uma aplicação Swing, usando a classe JDialog, ideal para janelas de configuração das diversas funções de um programa.

Neste exemlo, veremos como construir um aplicativo que realiza algumas modificações em um texto e grava este texto em um arquivo.

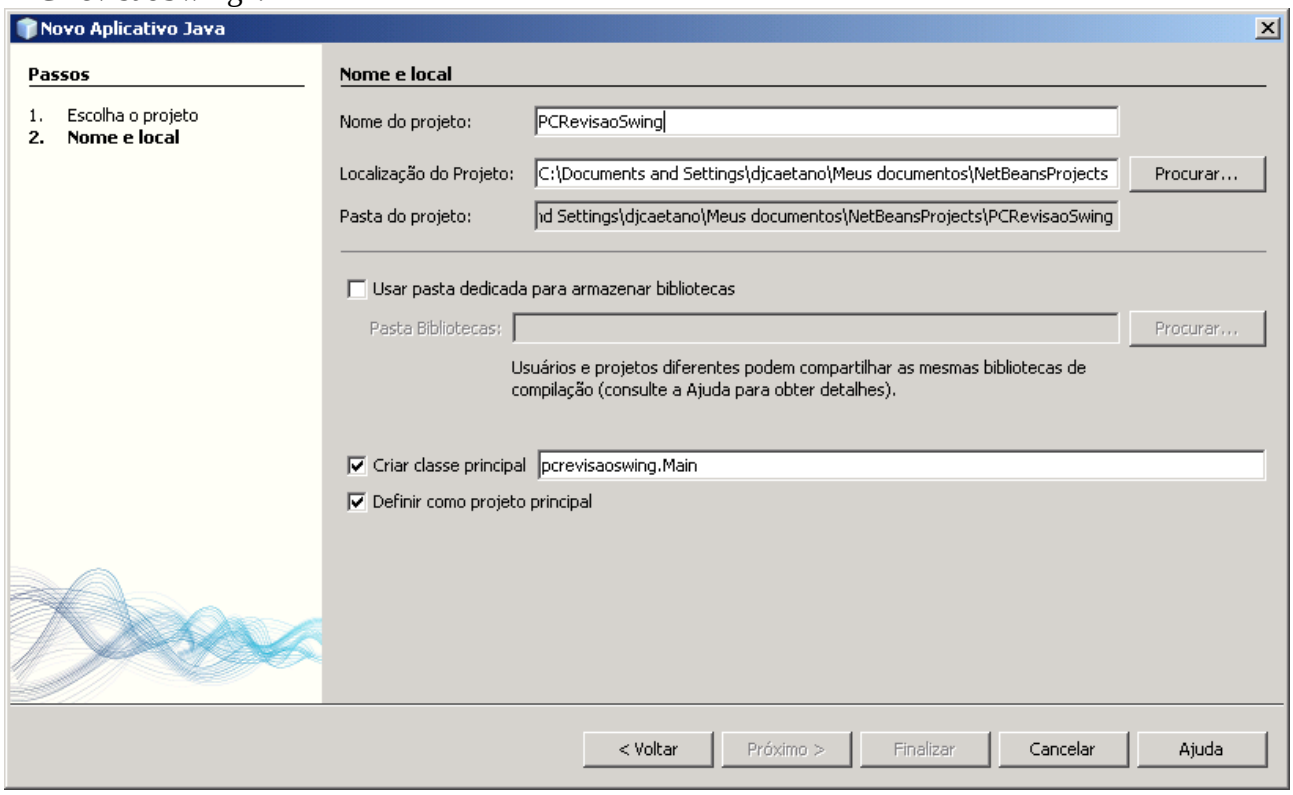
1. APLICATIVO EDITOR

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:

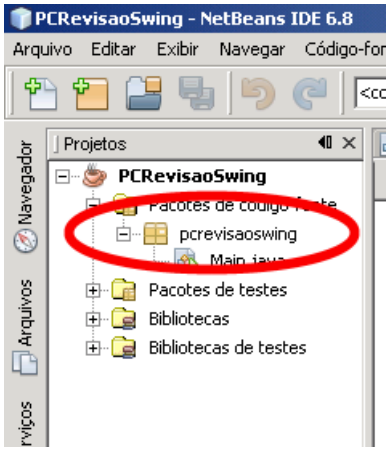
PASSO 2 – Agora é necessário escolher o tipo de projeto. Escolha a categoria “Java” e o tipo de projeto como “Aplicativo Java”.



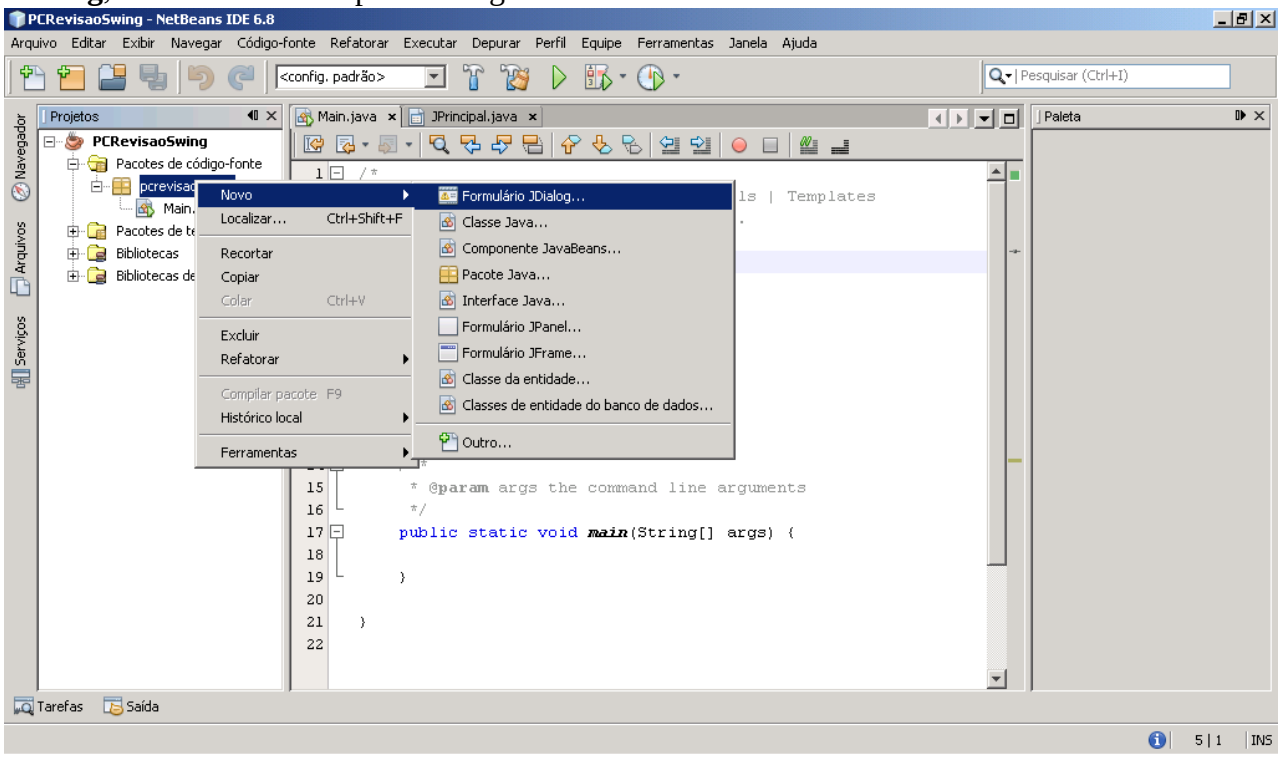
PASSO 3 – Agora é necessário dar um nome ao projeto. Lembre-se: NÃO use caracteres especiais, NÃO use acentos, NÃO use espaços. No exemplo abaixo, usamos o nome “PCRevisaoSwing”.



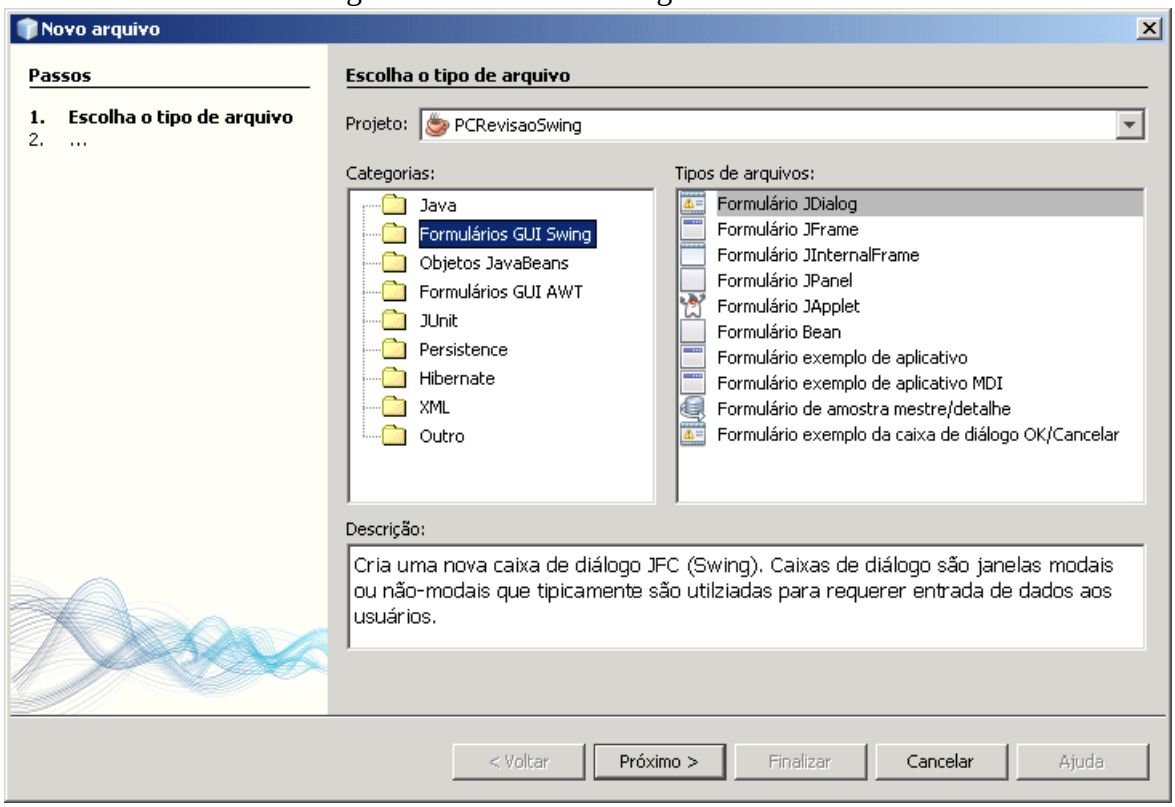
PASSO 4 – O próximo passo é criar a classe da janela do tipo JDialog. Clique com o botão direito do mouse no ícone do PACOTE com o nome **pcrevisaoswing**, conforme indicado abaixo:



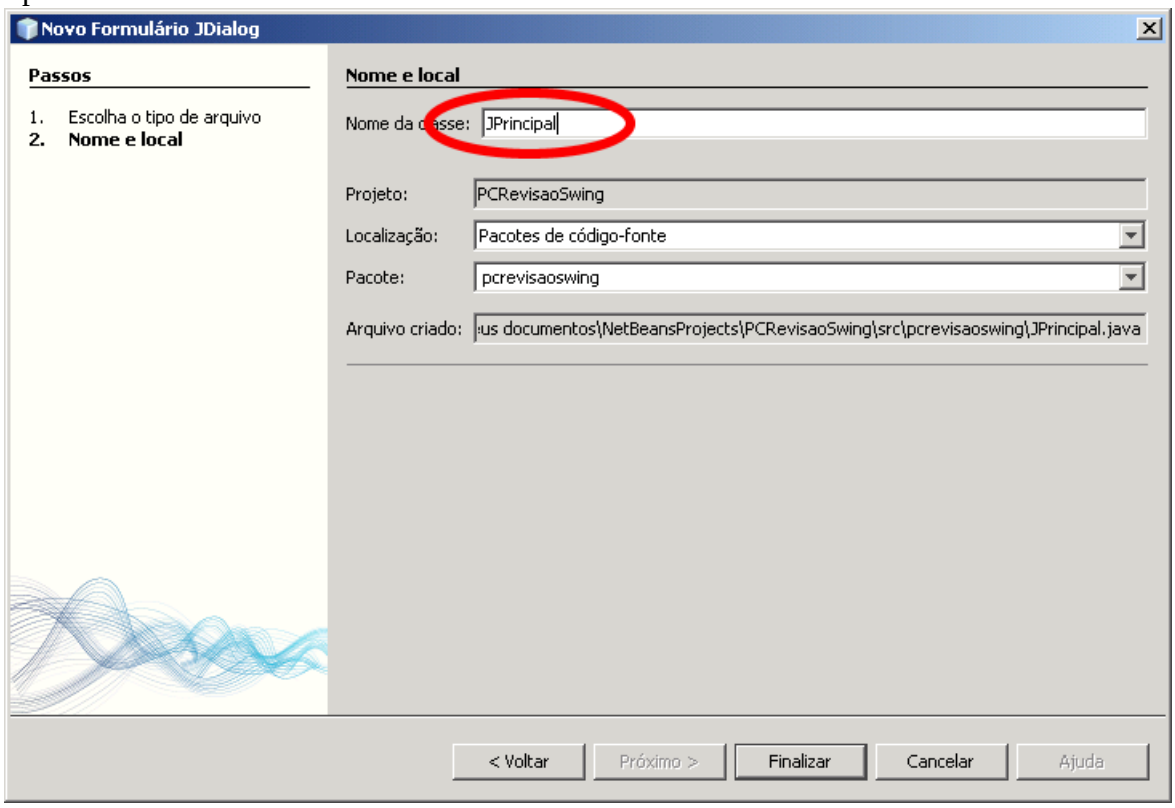
PASSO 5 – Ao clicar com o botão direito neste pacote, um menu irá aparecer com diversas opções. Selecione a opção **Novo** e um outro menu vai aparecer. Escolha a opção **Formulário JDialog**, como indicado na próxima figura:



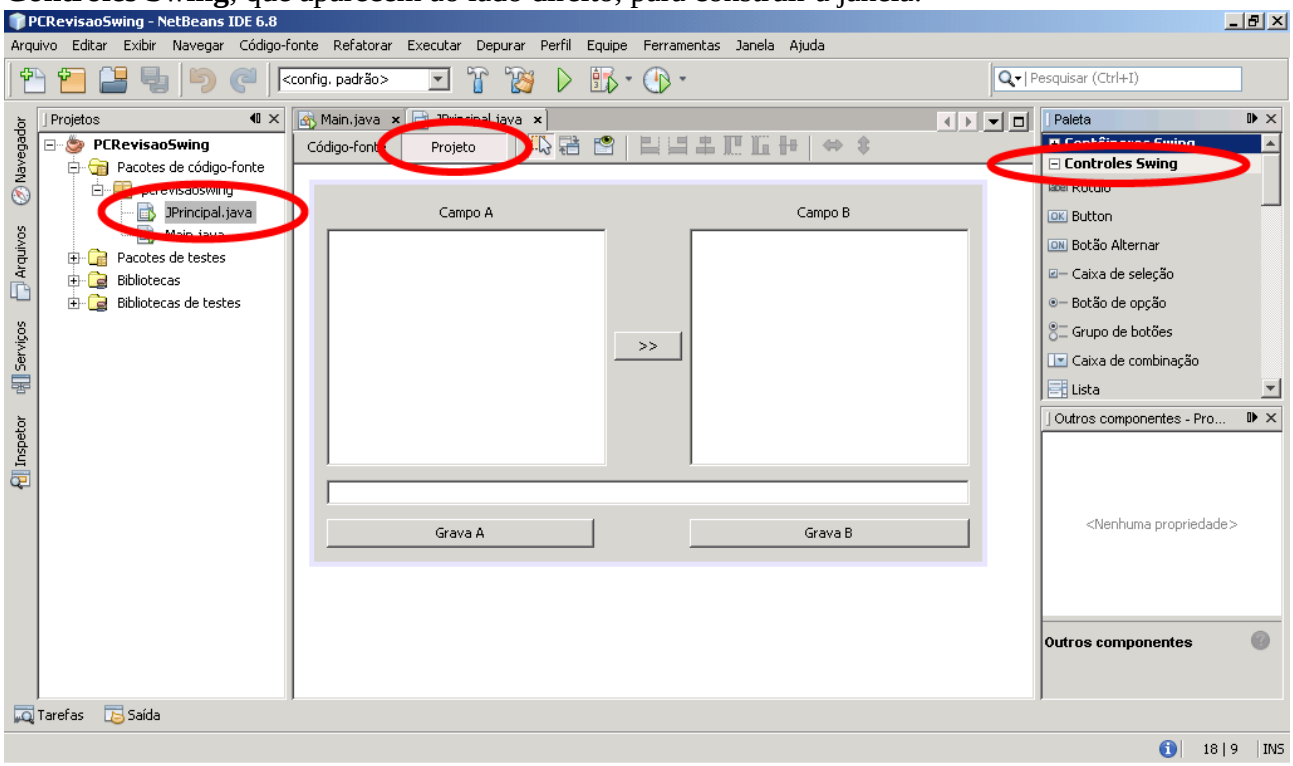
Caso essa opção não esteja disponível, clique na opção **Outro...**, que fará aparecer a janela abaixo. Você encontra o JDialog no local indicado na figura abaixo.



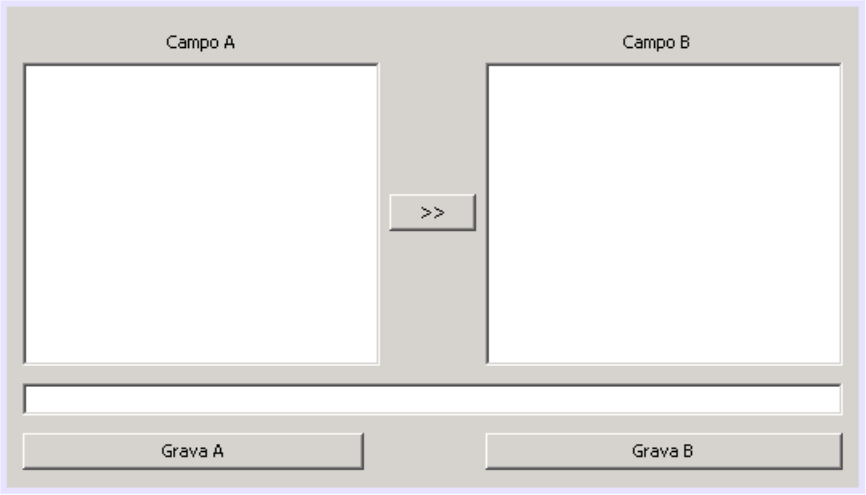
PASSO 6 – Após selecionar **Formulário JDialog**, uma nova janela se abrirá, onde devemos indicar o nome da classe a ser criada. Criaremos a nossa com o nome **JPrincipal** – de Janel Principal – conforme indicado abaixo.



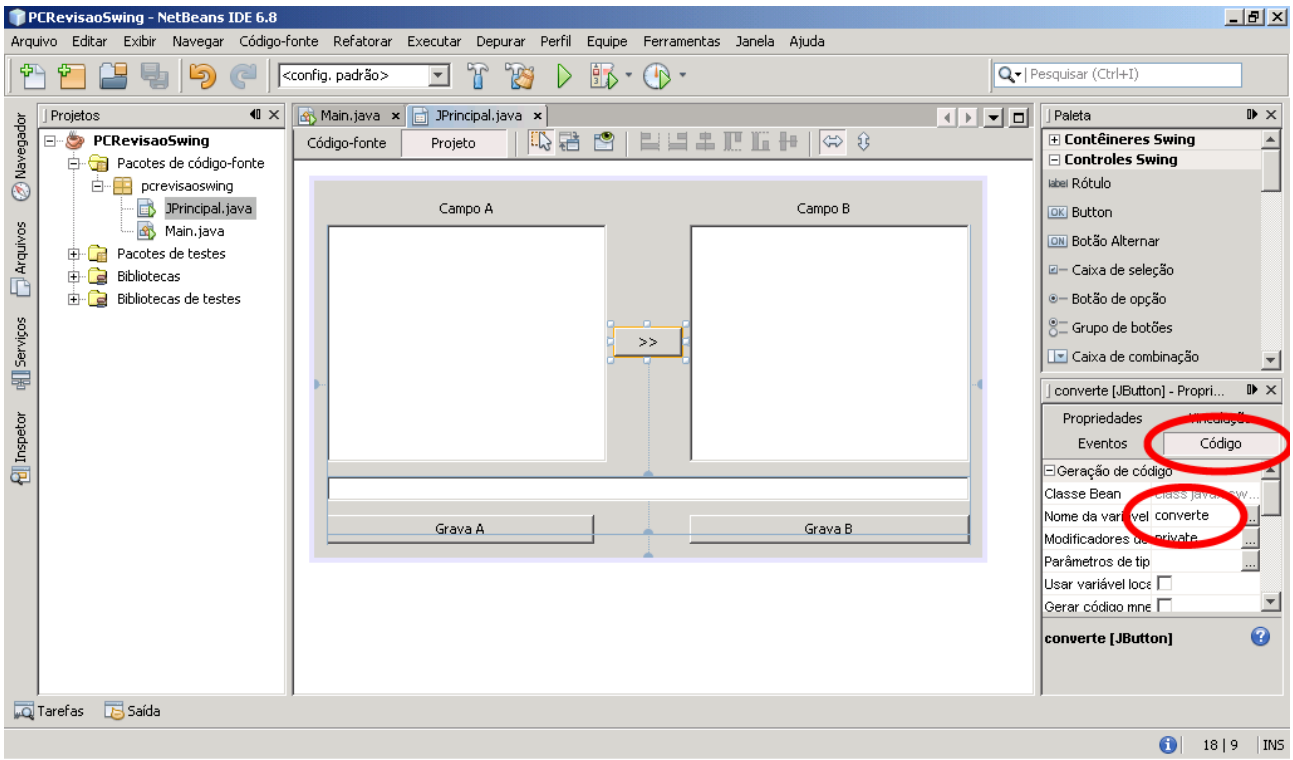
PASSO 7 – Selecionando a classe JPrincipal do lado esquerdo e o botão "Projeto" na parte superior central, a área para construção da janela será apresentada no centro da tela. Use os **Controles Swing**, que aparecem ao lado direito, para construir a janela.



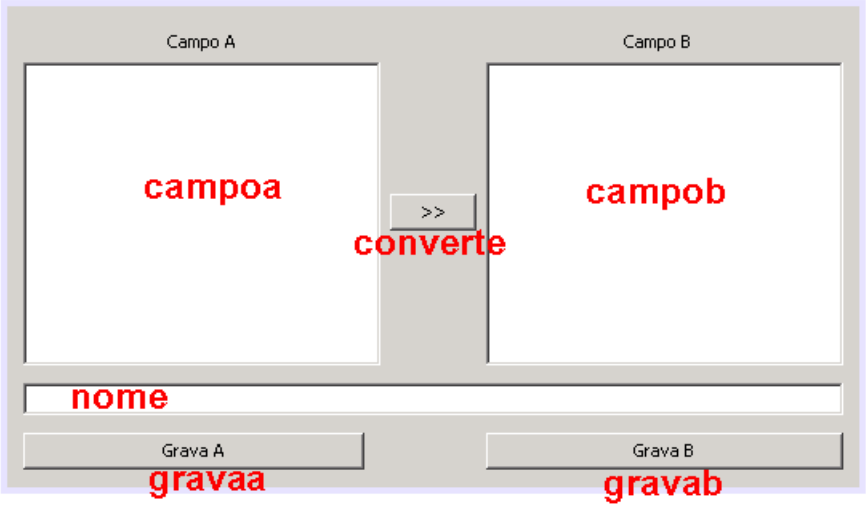
PASSO 8 – Observe a janela na figura abaixo. Reproduza-a o melhor que puder.



PASSO 9 – Agora é preciso nomear os elementos, para que possamos acessá-los de nosso código. Para isso, selecione um dos elementos e clique no botão **Código** que aparece do lado direito da tela, conforme indicado na janela abaixo. Procure pela opção **Nome da Variável**.



PASSO 10 – Dê os nomes aos elementos conforme indicado na figura abaixo.



PASSO 11 – Agora iremos adicionar algum código. Dê um duplo clique no botão ao qual você deu o nome **converte**, representado na janela com o símbolo “>>”. O NetBeans irá mudar para o modo “**Código-Fonte**”, indicado a seguinte região de código:

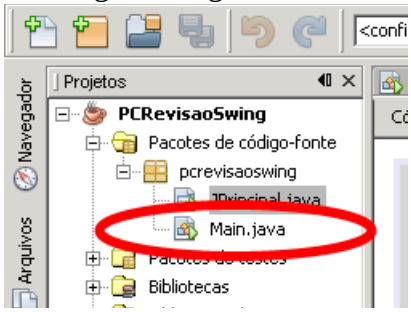
```
private void converteActionPerformed(java.awt.event.ActionEvent evt) {
        // Digite seu código aqui
}
```

Vamos inicialmente fazer um teste. Substitua o comentário de maneira que o método fique com a seguinte cara:

```
private void converteActionPerformed(java.awt.event.ActionEvent evt) {
        String texto = campoa.getText();
        campob.setText(texto);
}
```

Clique no botão com o triângulo verde, indicado abaixo, para executar o programa.

PASSO 12 – Se você fez tudo direitinho, não deve ter funcionado. Por quê? Lembre-se da lição fundamental. O Java, no NetBeans, sempre começa executando o método **main** da classe **Main**. Para corrigir oproblema, clique duas vezes no nome da classe Main.java, indicada no lado esquerdo da tela, conforme indicado na figura a seguir.



PASSO 13 – Procure agora o método **main**, indicado abaixo.

```
public static void main(String[] args) {
        // Digite seu código aqui
}
```

E modifique-o para que ele fique com a seguinte aparência:

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

Esse código simplesmente instrui o Java para que ele execute o método **main** da classe **JPrincipal**, que é exatamente a nossa janela principal. Experimente apertar o botão de executar (o triângulo verde) e veja o que ocorre agora.

PASSO 14 – Experimente digitar um texto no campo de texto do lado esquerdo e clique no botão >> e veja o que acontece! Feche o programa em seguida e volte para a janela do NetBeans.

PASSO 15 – Vamos agora incrementar um pouco este código. Clique duas vezes no nome do arquivo **JPrincipal.java**, do lado esquerdo da janela e, depois, selecione o botão **Projeto** do topo da janela do NetBeans. Clique duas vezes no botão "<<" para voltar a editar seu código, que deve estar como o deixamos:

```
private void converteActionPerformed(java.awt.event.ActionEvent evt) {
        String texto = campoa.getText();
        campob.setText(texto);
}
```

PASSO 16 – Vamos modificá-lo para que todos os caracteres espaço do texto sejam eliminados. Modifique o código para que fique como indicado abaixo. Atenção ao código "replaceAll": o primeiro parâmetro deve ser um **espaço** entre aspas e o segundo deve ser duas aspas sem qualquer caractere entre elas!

```
private void converteActionPerformed(java.awt.event.ActionEvent evt) {
        String texto = campoa.getText();
        texto = texto.replaceAll(" ","");  // substitui os espaços por "nada"
        campob.setText(texto);
}
```

Execute o programa e, após digitar um frase (por exemplo: "Esta é uma frase de teste!") e veja o que acontece quando ela é copiada para o lado direito.

PASSO 17 – Depois de executar o código acima e testá-lo para ver o que ocorre, iremos

adicionar código nos botões **gravaa** e **gravab**, para que eles gravem o conteúdo do campo em um arquivo.

O nome do arquivo deverá estar indicado no campo que chamamos de **nome**. Entretanto, vamos começar mais simples: vamos fazer primeiramente o Java imprimir no console o texto que escrevemos no **campoa**. Para isso, no modo de edição de **Projeto** da classe JPrincipal, clique duas vezes no botão "Grava A", e modifique o código para que fique com a seguinte cara:

```
private void gravaaActionPerformed(java.awt.event.ActionEvent evt) {
        System.out.println("Nome: " + nome.getText() );
        System.out.println("Texto: " + campoa.getText() );
        }
```

Execute o programa e veja o que acontece quando é apertado o botão "Grava A".

PASSO 18 – Ainda no modo "Código-Fonte", edite-o com as seguintes modificações:

```
private void gravaaActionPerformed(java.awt.event.ActionEvent evt) {
        grava( nome.getText() , campoa.getText() );
        }

private void grava(String umNome, String umTexto) {
        System.out.println("Nome: " +  umNome );
        System.out.println("Texto: " + umTexto );
        }
```

Execute novamente e veja que o resultado deve ter sido exatamente o mesmo!

PASSO 19 – Volte para o modo **Projeto** e dê um duplo-clique no botão "Grava B", e edite o código fonte de maneira que ele fique como indicado abaixo:

```
private void gravabActionPerformed(java.awt.event.ActionEvent evt) {
        grava( nome.getText() , campob.getText() );
        }
```

Execute novamente o programa e observe que o botão Grava B tem o mesmo comportamento que o botã Grava A, mas imprimindo o texto do campo do lado direito da janela.

PASSO 20 – Agora que os botões já estão funcionais, ainda no modo "Código-Fonte", procure o antigo método **grava** que criamos há pouco, que deve estar como o criamos:

```
private void grava(String umNome, String umTexto) {
        System.out.println("Nome: " +  umNome );
        System.out.println("Texto: " + umTexto );
        }
```

PASSO 21 – A primeira modificação será verificar se o nome do arquivo está preenchido, já que não se pode criar um arquivo sem nome. Para isso, faça as modificações indicadas no código, conforme apresentado a seguir.

```
private void grava(String umNome, String umTexto) {
        // Verifica se o comprimento do nome é zero...
        if ( umNome.length() == 0 ) {
                // Se for zero, abre uma janela reclamando para o usuário
                JOptionPane.showMessageDialog( this,
                        "Digite um nome de arquivo!",
                        "Erro!", JOptionPane.ERROR_MESSAGE);
                // E vai embora sem gravar nada
                return;
                }
        System.out.println("Nome: " +  umNome );
        System.out.println("Texto: " + umTexto );
        }
```

Ao tentar executar o programa, um erro será indicado. Isso ocorre porque estamos tentado usar uma classe da Swing – JOptionPane – que o NetBeans não sabe onde encontrar. Para ajudá-lo, procure o seguinte trecho de código, no início deste mesmo arquivo JPrincipal.java:

```
/*
 * JPrincipal.java
 *
 * Created on xx/xx/20xx, xx:xx:xx
 */

package pcrevisaoswing;
```

E acrescente a seguinte linha de código:

```
/*
 * JPrincipal.java
 *
 * Created on xx/xx/20xx, xx:xx:xx
 */

package pcrevisaoswing;
import javax.swing.*;
```

Isso deve resolver o problema. Executando o programa, você observará que se apertar "Grava A" ou "Grava B", uma janela informará que um nome de arquivo deve ser digitado.

PASSO 22 – Vamos agora fazer com que o botão de transformação realize algo mais elaborado: além de eliminar os espaços, vamos também pegar letra por letra do texto, fazendo com que as letras **pares** fiquem maiúsculas e as letras **ímpares** fiquem minúsculas. Para isso, acrescente o código abaixo.

```
private void converteActionPerformed(java.awt.event.ActionEvent evt) {
        String texto = campoa.getText();
        String saida;  // Local onde guardaremos o texto já no formato final
        String letra;  // Local onde guardaremos a letra atual
        int i;         // Contador de letras
        texto = texto.replaceAll(" ",""); // substitui os espaços por "nada"

        // Originalmente o texto de saida é vazio
        saida = "";

        // O contador começa na posição 0 e, enquanto for menor que o comprimento
        // do texto, será incrementado em uma unidade a cada execução do laço
        for (i = 0; i < texto.length(); i = i + 1) {
                // Pega a letra da posição i do texto original
                letra = texto.substring( i, i+1);
                // Se a posição do texto é divisível por 2 (resto igual a zero)
                if ( i%2 == 0 ) {
                        // Transforma em letra maiúscula
                        letra = letra.toUpperCase();
                        }
                // Caso contrário...
                else {
                        // Transforma em letra minúscula
                        letra = letra.toLowerCase();
                        }
                // Adiciona letra no texto de saída
                saida = saida + letra;
        }

        // Coloca texto de saída no campob
        campob.setText(saida);
}
```

OBSERVE que o método **substring** tem como primeiro parâmetro a posição inicial do trecho e como segundo parâmetro a posição final, sendo que a posição final **não** fará parte da substring. Experimente trocar o segundo parâmetro por **i+2** e veja o que ocorre.

CUIDADO com erros comuns como acrescentar **;** (ponto-e-vírgula) após o **for** e usar i++ como segundo parâmetro do **substring**.

LEMBRE-SE que o resto de divisão por 2, obtido aqui com o **i%2**, é usado para selecionar apenas as posições **pares**. **if (i%2)** … pode ser lido pelo programador como **if (i==par)** ...

PASSO 23 – Agora voltemos ao código do método **grava**, onde iremos criar um arquivo de impressão, acrescentado o seguinte código:

```
private void grava(String umNome, String umTexto) {
        // Verifica se o comprimento do nome é zero...
        if ( umNome.length() == 0 ) {
                // Se for zero, abre uma janela reclamando para o usuário
                JOptionPane.showMessageDialog( this,
                        "Digite um nome de arquivo!",
                        "Erro!", JOptionPane.ERROR_MESSAGE);
                // E vai embora sem gravar nada
                return;
                }
        // Indica o arquivo a ser criado / aberto,  com o nome "umNome"
        File arquivo = new File(umNome);
        // Tenta manipular o arquivo...
        try {
                // Configura um arquivo de impressão
                PrintStream saida = new PrintStream(arquivo);
                }
        catch (FileNotFoundException e) {
                // Se algum erro ocorreu, aqui é o lugar de mostrar a mensagem!
                JOptionPane.showMessageDialog( this,
                        "Problema com o nome do arquivo!",
                        "Erro!", JOptionPane.ERROR_MESSAGE);
                }
        }
```

Esse código ainda não grava nada no arquivo, mas as linhas que adicionamos são suficientes para que o Java crie o arquivo quando escrevermos ele. Adicionalmente, já programamos a rotina que apresenta o erro, caso haja algum problema ao criar o arquivo.

Entretanto, se você tentar executar, vai perceber que há um "problema" nesse código; o Java parece não conhecer nem **File** e muito menos **PrintStream**. Corrigir isso é fácil: é só dizer a ele onde encontrar essas coisas: no pacote **java.io.*** , o que pode ser feito lá no início do arquivo:

```
package pcrevisaoswing;
import javax.swing.*;
import java.io.*;
```

Fazendo isso você poderá executar o programa, mas nada será gravado ainda. Para isso, mais algumas linhas terão de ser adicionadas em nosso código.

PASSO 24 – Adicionemos agora sa linhas que faltam para que nosso programa imprima no arquivo o texto do campo selecionado.

```
private void grava(String umNome, String umTexto) {
        // Verifica se o comprimento do nome é zero...
        if ( umNome.length() == 0 ) {
                // Se for zero, abre uma janela reclamando para o usuário
                JOptionPane.showMessageDialog( this,
                        "Digite um nome de arquivo!",
                        "Erro!", JOptionPane.ERROR_MESSAGE);
                // E vai embora sem gravar nada
                return;
                }
        // Indica o arquivo a ser criado / aberto,  com o nome "umNome"
        File arquivo = new File(umNome);
        // Tenta manipular o arquivo...
        try {
                // Configura um arquivo de impressão
                PrintStream saida = new PrintStream(arquivo);
                // Imprime o texto no arquivo...
                saida.println(umTexto);
                // Finaliza a impressão
                saida.close();
                }
        catch (FileNotFoundException e) {
                // Se algum erro ocorreu, aqui é o lugar de mostrar a mensagem!
                JOptionPane.showMessageDialog( this,
                        "Problema com o nome do arquivo!",
                        "Erro!", JOptionPane.ERROR_MESSAGE);
                }
        }
```

Com isso o programa está finalizado, e já é capaz de gravar arquivos!

PASSO EXTRA – Você saberia como imprimir arquivos para Excel? Experimente imprimir palavras separadas por TAB (use o código **\t** no meio do texto) e grave o arquivo com extensão **.CSV**. Abra este arquivo no Excel e veja o que acontece!

2. ATIVIDADE (PARA NOTA!)

- **Em TRIOS;**
- **Vale nota AV1a;**
- **Entrega pelo e-mail: daniel@caetano.eng.br**
- **Entrega conforme datas apresentadas pelo professor!**
- **A data que vale é a do e-mail!**

A) Modifique o código do botão "converte" para inverter o texto. Por exemplo: se for digitado "Uma bola azul" no campoa, ao apertar o botão o texto deverá aparecer no campob como "luza alob amU". DICA: pegue letra por letra do texto do campoa, começando pela última, e vá acrescentando ao texto de saída (3,0 pontos).

B) Modifique o código do botão "converte" para que ele substitua as letras acentuadas pelos seus equivalentes sem acento. Por exemplo: se for digitado "ãéíôú" no campo a, o resultado no campob, ao apertar o botão "converte" deve sr "aeiou". DICA; use o método replaceAll (4,0 pontos).

C) Faça um programa com uma janela contendo um único campo de texto e um único botão e que, quando o usuário clicar no botão, o texto seja invertido e gravado em um arquivo de nome "**teste.txt**" (3,0 pontos).

EXTRA D) Modifique o código do botão do item C (ou da janela original, se você não conseguiu fazer o item C) para que ele, ao invés de inverter o texto, elimine os espaços e coloque a primeira letra de cada palavra em maiúsculas, e o resto em minúsculas (2,5 pontos). Por exemplo;

"Uma MENSAGEM de TeXtO" => "UmaMensagemDeTexto"

DICA:

- Considere que a primeira letra de cada palavra sempre vem depois de um espaco.
- Ajuste as letras maiúsculas e minúsculas ANTES de eliminar os espaços.

Unidade 2: Revisao Swing: Construção de Janela JFrame

Prof. Daniel Caetano

Objetivo: Capacitar o aluno para construir a janela principal de um aplicativo.

INTRODUÇÃO

Praticamente todo aplicativo que é desenvolvido para o usuário final possui ao menos uma janela principal que lhe forneça acesso a todos os principais comandos do sistema - em especial, costumam fornecer acesso a todos os principais casos de uso implementados por um determinado sistema.

O objetivo desta aula é apresentar a construção de uma janela principal de aplicativo, que é representada por uma classe especial no Java: JFrame (Janela Moldura). Adicionalmente serão apresentados os elementos de menu e também como acrescentar alguns elementos visuais como imagem de fundo e ícone na janela.

1. PREPARAÇÃO DO APLICATIVO BASE

PASSO 1: A primeira tarefa é criar um novo projeto no NetBeans, clicando no ícone de criação de projeto. Selecione o tipo de aplicação: **Java** na lista da esquerda e **Aplicativo Java** na lista da direita. Clique no botão **Próximo**. Dê para este aplicativo o nome que iremos usar por todo o restante do curso: **SisCli** (Sistema de Clientes). Com isso o NetBeans irá criar todos os arquivos básicos, incluindo a classe **Main.java**, dentro do pacote **siscli**.

PASSO 2: Clique com o botão direito no **pacote siscli** e selecione **Novo > Formulário Jframe** e dê o nome de **JPrincipal** a ela. Isso criará uma nova classe chamada **JPrincipal.java**, que estará automaticamente aberta no editor, no modo "Projeto".

> NOTA: Caso "**Formulário JFrame**" não esteja disponível, clique em **Novo > Outro...** e depois selecione **Formulários GUI Swing** na área de "Categorias" e **Formulário JFrame** na área "Tipos de arquivos".

PASSO 3: Se executarmos, veremos que a janela principal não abre. Na verdade, nada acontece porque não programamos nada no método **main** da classe **Main**. Vamos modificar a classe Main para que ela **crie um objeto** do tipo JPrincipal.

Sendo assim, na área de "Projetos", no lado esquerdo da tela, clique duas vezes no arquivo **Main.java**, para abrí-lo na área de edição. Modifique-o para que o código tenha a aparência indicada na próxima figura, lembrando sempre que as linhas em cinza indicam linhas que **já existem** no arquivo.

Main.java

```
package siscli;

public class Main {

    public static void main(String[] args) {
        JPrincipal aplicativo = new JPrincipal();
    }
}
```

PASSO 4: Executando o código anterior, aparentemente nada acontece. Na verdade, a janela foi criada, mas faltou indicar um comando para que ela seja apresentada ao usuário. Há várias formas de fazer isso; nesta aula, iremos obter este resultado modificando o contrutor da JPrincipal. Assim, clique duas vezes no nome da classe **JPrincipal.java**, e depois clique no botão "**Código-Fonte**" para editarmos o código da janela diretamente. Procure pelo método "**public JPrincipal()**" e modifique-o como indicado abaixo.

JPrincipal.java (método JPrincipal)

```
    /** Creates new form JPrincipal */
    public JPrincipal() {
        initComponents();

        setVisible(true);
    }
```

O método **setVisible** é um método herdado da classe JFrame, e ele recebe um parâmetro que indica se queremos que a janela seja desenhada para o usuário. Se o parâmetro for **false** a janela não será desenhada; se, por outro lado, o parâmetro for **true**, a janel a será desenhada. Experimente executar novamente o código e verifique que a janela aparece corretamente, embora apareça vazia (porque ainda não colocamos nenhum elemento nela!).

2. CRIANDO UM MENU NA JANELA PRINCIPAL

PASSO 5: A primeira coisa que faremos é acrescentar um menu à janela principal. Clique duas vezes no nome da classe **JPrincipal.java** e selecione o botão **Projeto**, para visualizar a edição da janela. Na Paleta, procure a categoria **Menus Swing** (é a terceira, depois de Contêineres Swing e Controles Swing). Arraste o elemento **Barra de Menu** para a janela. Ele deverá ser criado automaticamente com dois menus principais: File e Edit, sem nenhuma opção.

PASSO 6: Vamos agora editar estes menus. Clique duas vezes na opção "File" e modifique-a para "Arquivo". Clique duas vezes na opção "Edit" e modifique-a para "Ajuda". Repare que estes elementos são entradas para submenus, e não itens de menu: isso significa que, em geral, não iremos adicionar códigos quando um deles for executado.

PASSO 7: Vamos agora adicionar alguns itens de menu nos menus criados. Primeiramente, procure na Paleta o elemento **Item de menu** e arraste-o para cima do menu

Arquivo. O elemento será criado com o nome pouco criativo de "jMenuItem1". Clique duas vezes neste item de menu, para modificar seu nome para "Sair".

PASSO 8: Vamos adicionar uma tecla de atalho para esta opção. Clique na área à direta do nome do menu (Sair), onde está escrito "atalho" em cinza. Uma janela irá abrir. Clique no campo cujo nome é "**Curso da tecla**:". Pressione as teclas **ALT** e **X** simultaneamente. Clique no botão **Ok**. Se quiser adicionar um ícone para a opção, clique no quadradinho em branco à esquerda da opção (Sair).

PASSO 9: Para que este menu fique pronto, selecione-o (um retângulo laranja vai aparecer ao redor da opção Sair) e, na região de Propriedades, abaixo da Paleta, selecione o botão **Código**. Na opção **Nome da variável**, substitua o nome **jMenuItem1** por **mArquivoSair** .

PASSO 10: Vamos agora adicionar um item ao menu Ajuda. Procure na Paleta o elemento **Item de menu** e arraste-o para cima do menu Ajuda. O elemento será criado com o nome pouco criativo de "jMenuItem1". Clique duas vezes neste item de menu, para modificar seu nome para "Sobre".

PASSO 11: Para que este menu fique pronto, selecione-o (um retângulo laranja vai aparecer ao redor da opção Sobre) e, na região de Propriedades, abaixo da Paleta, selecione o botão **Código**. Na opção **Nome da variável**, substitua o nome **jMenuItem1** por **mAjudaSobre** .

Execute o aplicativo e veja se os menus aparecem corretamente. Observe que eles não irão fazer nada, pois não associamos código a nenhum deles.

3. ADICIONANDO FUNCIONALIDADE AOS MENUS

PASSO 12: Vamos agora adicionar alguma funcionalidade aos nossos menus. Primeiramente, selecione o menu "Arquivo > Sair". Um retângulo laranja deve aparecer ao redor da opção "Sair". Na área de Propriedades (abaixo da paleta), clique no botão "Eventos". Na opção **Action Performed**, selecione **mArquivoSairActionPerformed**. Isso irá, automaticamente, fazer com que o NetBeans crie a base do código que será executado quando o menu for selecionado.

PASSO 13: Modifique o método mArquivoSairActionPerformed conforme indicado a seguir. Isso adiciona a funcionalidade de sair do programa.

JPrincipal.java (método mArquivoSairActionPerformed)

```
    private void mArquivoSairActionPerformed(java.awt.event.ActionEvent evt) {
        dispose();
        System.exit(0);
    }
```

Neste código, a primeira linha em preto indica para o Java que a janela já pode ser destruída (o que faz com que ela desapareça visualmente, mas continua existindo na memória). A segunda linha em preto indica para o Java que o aplicativo já pode ser encerrado e toda a memória ainda alocada pode ser liberada.

Execute o programa e veja se funciona! Teste a tecla de atalho!

PASSO 14: Vamos agora adicionar funcionalidade ao menu Ajuda > Sobre. Primeiramente, clique no botão "Projeto" para visualizar a janela e selecione o menu "Ajuda > Sobre". Um retângulo laranja deve aparecer ao redor da opção "Sobre". Na área de Propriedades (abaixo da paleta), clique no botão "Eventos". Na opção **Action Performed**, selecione **mAjudaSobreActionPerformed**. Isso irá, automaticamente, fazer com que o NetBeans crie a base do código que será executado quando o menu for selecionado.

PASSO 15: Modifique o método mAjudaSobreActionPerformed conforme indicado a seguir. Isso indica ao Java que mostre uma janela de informações.

JPrincipal.java (método mAjudaSobreActionPerformed)

```
    private void mAjudaSobreActionPerformed(java.awt.event.ActionEvent evt) {
        JOptionPane.showMessageDialog(this,
            "Sistema de Clientes SisCli v0.1.\nEquipe Estácio-Radial.",
            "Sobre o SisCli",
            JOptionPane.INFORMATION_MESSAGE);
    }
```

Os termos JOptionPane estarão marcados pelo NetBeans porque o uso da classe JOptionPane exige o uso do pacote **javax.swing.***. Podemos colocar o **import** manualmente ou podemos clicar com o botão direito sobre o texto **JOptionPane** marcado com vermelho e selecionar a opção **Corrigir importações** no menu.

Neste código, o primeiro parâmetro do método showMessageDialog (this) indica que a janela de mensagens a ser aberta deve ser centralizada na JPrincipal (como estamos no código de JPrincipal, **this** é uma referência que, traduzindo, significa "esta janela"). O segundo parâmetro é o texto que aparece na janela. O terceiro parâmetro indica o título da janela (barra de título) e o quarto parâmetro indica o tipo de mensagem.

4. TORNANDO A APLICAÇÃO MAIS PROFISSIONAL

PASSO 16: Quase todo aplicativo profissional tem uma janela perguntando ao usuário se ele realmente quer sair. No Java, em especial, na classe JOptionPane, já existe uma janela prontinha para este tipo de pergunta: o diálogo do tipo Confirmation Dialog, que é criado com o método **showConfirmationDialog**.

Este método retorna o botão precionado: YES_OPTION para SIM e NO_OPTION para NÃO. Assim, podemos usar um **if** para verificar se o usuário realmente quer sair. Observe o código a seguir.

JPrincipal.java (método mArquivoSairActionPerformed)

```
    private void mArquivoSairActionPerformed(java.awt.event.ActionEvent evt) {
        int opcao;
        opcao = JOptionPane.showConfirmDialog(this,
                "Voce deseja mesmo sair?", "Saindo do SisCli",
                JOptionPane.YES_NO_OPTION);
        // Se NAO quer sair, retorna (sem fechar o aplicativo!)
        if (opcao == JOptionPane.NO_OPTION) return;
        dispose();
        System.exit(0);
    }
```

É criada uma variável chamada **opcao** para guardar a opção selecionada pelo usuário e, em seguida, é aberto o diálogo com showConfirmDialog. Os primeiros três parâmetros são idênticos ao do showMessageDialog, visto anteriormente; o último, porém, indica os botões que devem estar disponíveis. No caso, foi adatada a opção com botão SIM e botão NÃO (YES_NO_OPTION).

A janela de diálogo interrompe a execução desse código até que o usuário selecione uma das alternativas. Quando a seleção for feita, o número do botão selecionado será armazenado na variável **opcao**. Em seguida, o código verifica se a opção selecionada foi NÃO (NO_OPTION) e, se sim, retorna sem fechar o aplicativo. Caso não tenha sido o botão NÃO a ser pressionado, o aplicativo será fechado, pois o dispose e o exit serão processados.

PASSO 17: Usualmente queremos colocar uma marca da empresa no software, um logotipo visual. A forma mais simples de fazer isso é com o uso de um rótulo. Selecione para edição o arquivo JPrincipal.java e clique no botão "Projeto". Na Paleta, na categoria "Controles Swing", selecione **Rótulo** e arraste-o para o meio da janela, na área de edição.

PASSO 18: Selecione o rótulo jLabel1 que apareceu (um retângulo laranja deverá aparecer ao redor do mesmo) e, na região de propriedades, selecione o botão **Propriedades**. Procure pela opção **icon** e clique no botão com "**...**". Surgirá uma janela para a seleção de imagem. Selecione a opção **Imagem Externa** e use o botão "**...**" para selecionar um arquivo de imagem e, finalmente, selecione o botão **Ok**.

NOTA: Se não possuir uma imagem de fácil acesso, use o endereço web **http://www.caetano.eng.br/main/images/aflogo_horiz_transp.gif**

PASSO 19: Agora clique duas vezes no rótulo, que já deve aparecer com a imagem, e apague o texto do mesmo. Feito isso, posicione a imagem como desejar.

PASSO 20: Agora vamos adicionar um nome à Janela principal. Selecione o fundo da janela Jprincipal e, na área de propriedades, selecione o botão **Propriedades**. Procure a opção **title** e altere seu valor para **SisCli - Sistema de Clientes** . Grave e execute para testar.

PASSO 21: O último passo é a inserção de um ícone à janela. Primeiramente devemos adicionar um arquivo de imagem de ícone ao projeto. Isso é fácil: faça o download

de um arquivo no formato **gif** ou **png** com um tamanho limitado a 32x32 pixels (por exemplo, este: http://www.caetano.eng.br/main/images/icone.gif). Com o download feito, **arraste** o ícone para dentro do pacote **siscli**. Ele aparecerá como um arquivo de imagem.

PASSO 22: Agora é preciso indicar ao Java que mostre esse arquivo, diretamente no código da aplicação. Assim, selecione o arquivo **JPrincipal.java** e clique no botão **Código-fonte**. Produre pelo método JPrincipal e modifique-o como indicado abaixo:

```
/** Creates new form JPrincipal */
public JPrincipal() {
    initComponents();

    ImageIcon icon = new ImageIcon(getClass().getResource("icone.gif"));
    setIconImage(icon.getImage());
    setVisible(true);
}
```

A primeira linha em preto lê o arquivo de imagem na memória e a segunda linha em preto modifica o ícone da janela para o ícone lido. Observe que você deve ajustar o nome da imagem (no caso, **icone.gif**) ao nome do arquivo que você adicionou ao seu projeto.

PASSO 23: Está quase pronto, com um único problema: caso o Java não encontre o arquivo de imagem do ícone, uma porção de erros serão indicados na região de mensagens e o programa não irá funcionar (experimente, indicando um nome de arquivo que não exista). Para evitar que esse tipo de problema, vamos usar uma região **try~catch** adequada, para capturar o erro **NullPointerException**, que é o erro que temos quando a imagem não foi encontrada.

```
/** Creates new form JPrincipal */
public JPrincipal() {
    initComponents();

    try {
        ImageIcon icon = new ImageIcon(getClass().getResource("icone.gif"));
        setIconImage(icon.getImage());
        }
    catch (NullPointerException ex) {
        System.out.println("Ícone de aplicativo não encontrado!");
        }
    setVisible(true);
}
```

Unidade 3: Orientação a Objetos e UML

Noções de Orientação a Objetos

Prof. Daniel Caetano

Objetivo: Revisar os conceitos de Orientação a Objetos e uma breve visão de um Diagramas de Classes.

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

INTRODUÇÃO

Apesar de nem sempre ser tão clara, segundo Bezerra (2007), a necessidade de realizar a modelagem e projeto de software é das mais importantes conforme aumenta sua complexidade. O exemplo usado por Bezerra (2007) é o da casa de cachorro. Provavelmente ninguém precisa de um projeto para construí-la. Basta "algumas ripas de madeira, alguns pregos, uma caixa de ferramenta e certa dose de amor por seu cachorro". Entretanto, quando se pretende construir uma casa ou um edifício, as coisas se complicam o suficiente para ninguém pensar em construí-las sem um projeto; na realidade, é até mesmo proibido realizar tais obras sem um projeto.

O objetivo principal do projeto é gerenciar a complexidade do problema reduzindo-os a modelos, usando o princípio das *abstrações*, que removem todas as características "não importantes" do problema, evidenciando apenas os dados relevantes. Os modelos auxiliam a comunicação de todos envolvidos no desenvolvimento, possibilitando que todos enxerguem o objeto a ser construído sob a mesma óptica.

Além disso, a realização do projeto permite antever problemas antes de partir para o trabalho de implementação propriamente dito, permitindo que problemas sejam corrigidos sem desperdício de recursos. Em outras palavras, o projeto evita que se construa uma parede para depois ter que derrubá-la.

Neste curso deve ser produzido um projeto baseado no paradigma orientado a objetos, usando a Unified Modeling Language para especificação. Esta aula introduz alguns dos conceitos necessários.

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*.

Inicialmente estaremos trabalhando com "objetos de análise". Isso significa que esboçaremos o que objeto faz e o que ele armazena, mas não "como faz" ou "como armazena". Observe que estaremos respondento, então, à pergunta "O quê?" e não à pergunta "Como?". Deixemos o "Como" para a etapa de projeto.

Exemplos de objetos do mundo real:

TV	- Liga	- Canal
	- Desliga	- Volume
	- Muda canal	- Estado(ligada/desligada)
Carro	- Liga	- Cor
	- Desliga	- Velocidade
	- Acelera	- Quilometragem (odômetro)
	- Breca	- Portas

1.2. Conceitos da Modelagem e da Programação Orientadas a Objetos

Além dos objetos, os modelos orientados a objetos também se baseiam em outros conceitos, como os de classes, mensagens e associações, além das propriedades dos objetos.

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.

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. Este estímulo é chamado de *mensagem*. Em outras palavras, uma mensagem é a forma como um objeto se comunica com outro (ou estimula a outro). Essas mensagens normalmente são padronizadas, constituindo uma *interface*. Pense em uma língua comum que os objetos precisam saber para poder se comunicar. Para um ser humano entender um pedido de outro (ou seja, receber uma mensagem), é necessária uma *interface* comum.

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.*

1.3. Classes x Objetos

É importante, neste momento, reforçar a diferença entre uma classe e um objeto. Uma classe é um conceito genérico: pessoa, carro, cliente. Um objeto, por outro lado, é um conceito específico: Alberto, Mustang Azul, O Comprador do Mustang Azul do Alberto. Uma classe pode ser considerada como um "molde" de um objeto.

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

Quando programamos, nós programamos **classes**. Para realizar uma tarefa, podemos precisar de objetos específicos e, então, solicitamos ao Java que construa um objeto com base em uma classe específica. Um objeto é criado da seguinte forma:

```
ClasseDoObjeto nomeDoObjeto = new ClasseDoObjeto(parâmetrosDeCriação);
```

Por exemplo, para criar a Pessoa chamada Alberto, uma possibilidade seria essa:

```
Pessoa alberto = new Pessoa("Alberto");
```

Observe que o nome do objeto não precisa ter relação nenhuma com os atributos do objeto. O código a seguir, por exemplo, realiza a mesma tarefa que o código anterior:

```
Pessoa umaPessoa = new Pessoa("Alberto");
```

1.4. Chamando Métodos (Ou Enviando Mensagens)

Uma vez que um objeto tem atributos e métodos, pode ser que desejemos solicitar a algum objeto que ele nos realize uma tarefa específica. Por exemplo, podemos querer solicitar que um personagem de um jogo se desenhe na tela. Isso poderia ser feito conforme indicado abaixo:

```
personagem.desenhar(tela);
```

Observe que primeiramente foi indicado o nome do objeto (personagem), seguido do método (desenhar) e, como parâmetro, foi passado o nome de outro objeto (tela).

De maneira geral, uma chamada a um método segue o seguinte formato:

[dono_do_metodo.]<nome_do_metodo>([parametros_do_metodo]);

1.5. Principais Propriedades da Orientação a Objetos

As principais caracterísitas das classes de objetos constituem também as fundações do modelo orientado a objetos. Estas características são: encapsulamento, 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.

Note que essa propriedade permite que pensemos em termos de "classe de análise" antes de pensarmos em "classe de projeto". Nas "classes de análise" são definidas, basicamente, as interfaces dos objetos. Posteriormente, nas "classes de projeto", é que existirá a preocupação em como fazer tais objetos funcionarem a partir da interface estabelecida.

HERANÇA

Herança é a propriedade que nos permite criar uma nova classe que contenha todos os elementos de uma classe pré-existente, complementando-a com novos atributos e métodos.

De forma mais rigorosa, podemos dizer que herança é a propriedade que permite que, ao estender 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*.

Como os objetos da nova classe possuem todos os elementos da classe mais antiga, podemos dizer que o objeto da nova classe **também** é objeto da classe antiga. Por exemplo, se temos a classe "Pessoa", podemos criar a classe "Trabalhador" que a estenda e, assim, dizer que *Trabalhador é uma Pessoa.*

Isso é razoável, afinal, qualquer objeto da classe Trabalhador tem todos os atributos e métodos que um objeto da classe Pessoa teria!

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*, 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 operar tanto o carro quanto a caminhonete, mesmo que o objeto caminhonete tenha sido inventado muito tempo depois da criação do objeto João.

O polimorfismo pode ser implementado por Herança, como no exemplo a seguir:

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

Se objetos da classe Pessoa sabem manipular objetos da classe Veículo, sabem manipular também objetos das classes Carro e Caminhonete, que são classes **especializadas** da classe Carro original. Assim, se *joao* e *carla* são objetos da classe Pessoa e *carroAzul* é um objeto da classe Carro e *caminhoneteVermelha* é um objeto da classe Caminhonete, então tanto objeto *joao* quanto o objeto *carla* podem interagir com carroAzul e caminhoneteVermelha da mesma forma com que o fariam com objetos da classe Veiculo.

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

2. OBSERVAÇÕES SOBRE PROGRAMAÇÃO ORIENTADA A OBJETOS

1) Não é preciso usar uma linguagem orientada a objetos para programar de forma orientada a objetos. É importante lembrar que tudo que é feito no computador é convertido pelo compilador em código de máquina. Assim, em essência, é possível realizar Programação Orientada a Objetos até mesmo em Assembly.

2) Um programa orientado a objetos não tem necessariamente uma interface com o usuário (UI) orientada a objetos. O método de programação e a interface com o usuário são duas coisas bastante distintas. Para que a interface seja OO, é necessário que o programador implemente todas as características citadas anteiormente para os elementos da interface, o que não é tão simples. Como um exemplo, podemos citar a interface do Windows, que é programada em linguagem OO, mas sua operação não respeita os princípios da OO.

Por outro lado, apesar disso, é possível dizer que o uso de uma linguagem orientada a objetos facilita a implementação de um projeto orientado a objetos, e a razão para isso é a proximidade semântica de modelos.

Com proximidade do modelo e o código, é mais fácil também definir uma mudança no sistema fazendo uma análise do projeto e, a partir dele, ir diretamente ao código daquele componente ou módulo para realizar as mudanças necessárias.

Nem tudo são flores, entretanto. O uso de Orientação a Objetos tem implicações de desempenho, seja em termos de memória ou de processamento.

2.1. Eficiência em Softwares Orientados a Objetos

1) Em geral, os programas OO são menos eficientes que programas bem estruturados, sob o ponto de vista de custo computacional, ou seja, na sua velocidade de execução e consumo de memória. Por outro lado, nestas condições, os códigos estruturados tendem a ser relativamente mais complexos.

2) É possível dizer também, em geral, que os programas OO são muito mais eficientes do que programas estruturados, sob o ponto de vista de tempo de desenvolvimento. Isto é uma conseqüência da maior abstração e proximidade do modelos de projeto e implementação com relação à realidade.

3) Em programas OO a correção de bugs é mais simples, pois é mais fácil testar componentes (que quase não dependem do "mundo exterior") do que testar funções (que podem ser altamente dependentes do "mundo exterior"). Entenda "mundo exterior" por *variáveis globais* e outros elementos similares.

4) Os programas OO são modulares por natureza, já que os próprios objetos acabam por constituir módulos atômicos. Por esta razão, o reaproveitamento de código é facilitado.

3. UM DIAGRAMA DE CLASSES UML SIMPLIFICADO

Para rememorar os principais elementos de um diagrama de classes UML vamos analisar um diagrama simplificado.

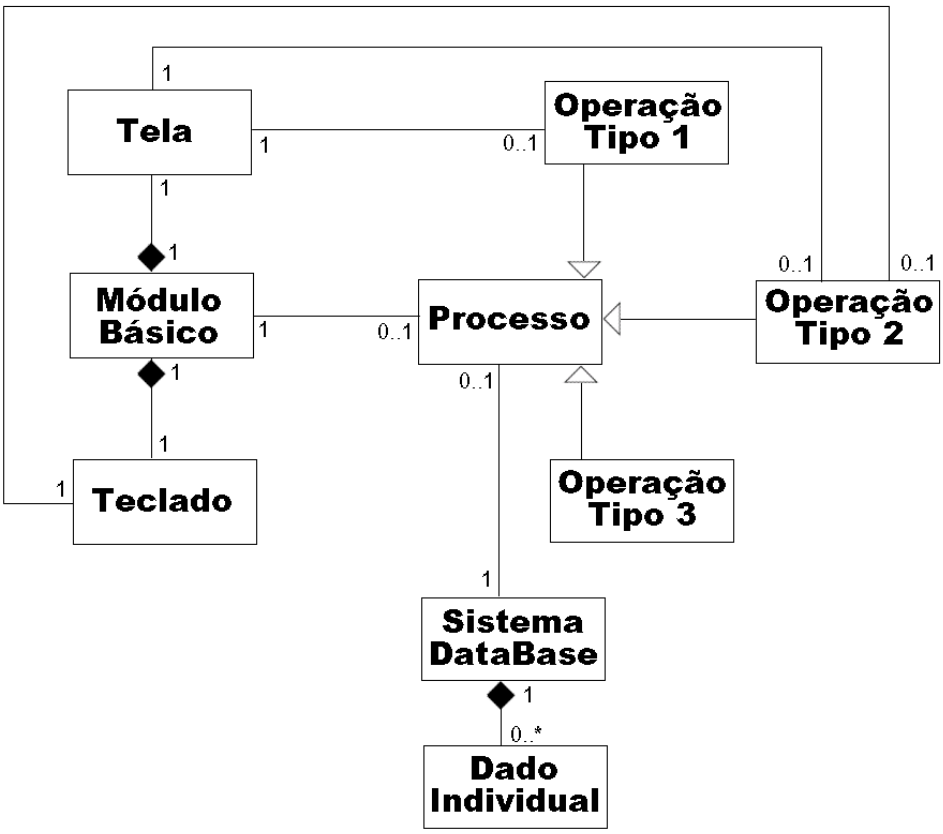


Figura 1: Diagrama de Classes Simplificado

Este é um diagrama básico de classes de um Sistema de Informações genérico. Neste diagrama temos uma arquitetura do tipo MVC (Modelo/Visão/Controle), onde a interação com o usuário (Visão) é separada das classes que implementam os processos do negócio (Controle) e ambas são separadas das classes que manipulam o banco de dados (Modelo).

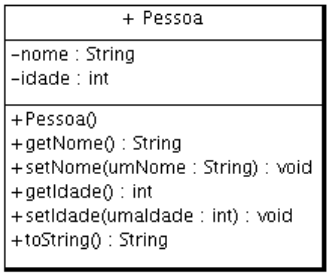
Nesta implementação a camada chamada **visão** abrange o Módulo Básico, o Teclado e a Tela. A camada chamada **modelo** abrange o Sistema Database e o Dado Individual. A camada **controle** abrange o Processo, a Operação Tipo 1, a Operação Tipo 2 e a Operação Tipo 3.

A simbologia do UML indica que o Módulo Básico é composto por Teclado e Tela, e o Sistema DataBase é composto por vários Dado Individual. Além disso, as Operações (Tipo 1, 2 e 3) são especializações da classe Processo.

4. EXERCÍCIOS

O objetivo deste exercício é exercitar a criação de classes e objetos, exercitando e rememorando alguns dos conceitos de criação de classes.

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



2. Não se esqueça de comentar adequadamente o código.

3. 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  <<== 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
      }
   }
```

4. Modifique o construtor para que ele receba as informações sobre o objeto, como nome e idade, e modifique os valores internos do objeto.

Nota: O procedimento apresentado será modificado posteriormente, para um formato mais adequado e de acordo com as premissas da orientação a objetos.

```
/**
 * 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(String nome, int idade) { // <= siga sempre uma ordem lógica!
      // Inicializa variáveis de instância
      this.nome = nome;
      this.idade = idade;
      }
   }
```

5. Vamos usar o método **main** para criar um objeto desta classe. O método **main** é um dos melhores locais para realizar testes unitários de uma classe e, durante o desenvolvimento, praticamente sempre o utilizaremos com esse intuito.

```
/**
 * 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(String nome, int idade) { // <= siga sempre uma ordem lógica!
      // Inicializa variáveis de instância
      this.nome = nome;
      this.idade = idade;
      }

   /**
    * 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");
      }
   }
```

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

4.1. Um pouco de Boas Práticas

7. 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  <<== Versão é muito importante!
 */

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

   /**
    * Construtor para objetos da classe Pessoa
    * @param      nome   o nome da pessoa
    * @param      idade  a idade da pessoa, deve ser entre 0 e 100
    * @return     não há valor de retorno
    */
   public Pessoa(String nome, int idade) { // <= siga sempre uma ordem lógica!
      // Inicializa variáveis de instância
      this.nome = nome;
      this.idade = idade;
      }

   /**
    * 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");
      }
   }
```

8. Observe o diagrama apresentado: existem alguns sinais de "+" e "-" nele representados. Os métodos e atributos com um "-" devem ser privados e apenas os marcados com "+" devem ser públicos. Isso é importante porque declarar atributos como públicos é, em geral, desaconselhável, por permitir um desrespeito aos critérios de encapsulamento.

Entretanto, se simplesmente eles forem modificados para que sejam **private**, haverá um problema para manipular estes valores fora dela. Para isso, criaremos dois métodos públicos: getIdade() e getNome().

getIdade() deve simplesmente retornar um inteiro com o valor do atributo idade e, getNome(), por sua vez, deve retornar uma String com o valor do atributo nome. Ambas estão indicadas no código 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
   private String nome;
   private int idade;

   /**
    * Construtor para objetos da classe Pessoa
    * @param       nome   o nome da pessoa
    * @param       idade  a idade da pessoa, deve ser entre 0 e 100
    * @return      não há valor de retorno
    */
   public Pessoa(String nome, int idade) { // <= siga sempre uma ordem lógica!
      // Inicializa variáveis de instância
      this.nome = nome;
      this.idade = idade;
      }

   /**
    * Retorna o nome da pessoa
    * @return      String contendo o nome
    */
   public String getNome() {
      return (nome);
      }

   /**
    * Retorna a idade da pessoa
    * @return      int contendo a idade
    */
   public int getIdade() {
      return (idade);
      }

   /**
    * 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.getNome()+"\nIdade: "+joao.getIdade()+"\n");
      System.out.println("Nome: "+maria.getNome()+"\nIdade: "+maria.getIdade()+"\n");
      }
   }
```

9. A boa prática diz que é interessante possuir métodos "setters", ou seja, métodos para modificar o valor dos atributos. Neste caso, estes métodos podem ser privados, já que não há necessidade de modificar os valores por um meio externo.

Muitas vezes - como é o caso - nem mesmo faz sentido que alguns atributos sejam alterados externamente. O uso de "setters" mesmo dentro da classe permite que limites (como idade entre 0 e 100) sejam testados e respeitados. Veja no código 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
   private String nome;
   private int idade;

   /**
    * Construtor para objetos da classe Pessoa
    * @param       nome   o nome da pessoa
    * @param       idade  a idade da pessoa, deve ser entre 0 e 100
    * @return      não há valor de retorno
    */
   public Pessoa(String nome, int idade) { // <= siga sempre uma ordem lógica!
      // Inicializa variáveis de instância
      setNome(nome);
      setIdade(idade);
      }

   /**
    * Retorna o nome da pessoa
    * @return      String contendo o nome
    */
   public String getNome() {
      return (nome);
      }

   /**
    * Retorna a idade da pessoa
    * @return      int contendo a idade
    */
   public int getIdade() {
      return (idade);
      }

   /**
    * Muda o nome da pessoa
    * @param       nome String contendo o nome
    */
   public void setNome(String novoNome) {
      nome = novoNome;
      }

   /**
    * Muda a idade da pessoa
    * @param       int contendo a idade (0 a 100)
    */
   public void setIdade(int novaIdade) {
      if (novaIdade < 0) novaIdade = 0;
      else if (novaIdade > 100) novaIdade = 100;
      idade = novaIdade;
      }

   /**
    * 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.getNome()+"\nIdade: "+joao.getIdade()+"\n");
      System.out.println("Nome: "+maria.getNome()+"\nIdade: "+maria.getIdade()+"\n");
      }
   }
```

10. Uma última modificação interessante é usar um padrão do Java quanto a imprimir os dados de um objeto qualquer. No exemplo, em nossa rotina de teste **main**, "invadimos" o conteúdo do objeto para imprimir suas características. Não seria muito mais simples se pudéssemos simplesmente escrever:

```
System.out.println(joao);
System.out.println(maria);
```

Para imprimir os objetos? E de fato, podemos. Para isso, basta criar um método chamado toString, como indicado a seguir:

```
public String toString() {
   String tmpString;
   tmpString = "Nome: " + getNome() + "\nIdade: " + getIdade() + "\n";
   return(tmpString);
   }
```

Esse código, inserido no programa final, já com as alterações feitas, fica assim:

```
/**
 * 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
   private String nome;
   private int idade;

   /**
    * Construtor para objetos da classe Pessoa
    * @param      nome   o nome da pessoa
    * @param      idade  a idade da pessoa, deve ser entre 0 e 100
    * @return     não há valor de retorno
    */
   public Pessoa(String nome, int idade) { // <= siga sempre uma ordem lógica!
      // Inicializa variáveis de instância
      setNome(nome);
      setIdade(idade);
      }

   /**
    * Retorna um texto que descreve o objeto
    * @return     String contendo a descrição da pessoa
    */
   public String toString() {
      String tmpString;
      tmpString = "Nome: " + getNome() + "\nIdade: " + getIdade() + "\n";
      return(tmpString);
      }

   /**
    * Retorna o nome da pessoa
    * @return     String contendo o nome
    */
   public String getNome() {
      return (nome);
      }
```

```
    /**
     * Retorna a idade da pessoa
     * @return      int contendo a idade
     */
    public int getIdade() {
       return (idade);
       }

    /**
     * Muda o nome da pessoa
     * @param       nome String contendo o nome
     */
    public void setNome(String novoNome) {
       nome = novoNome;
       }

    /**
     * Muda a idade da pessoa
     * @param       int contendo a idade (0 a 100)
     */
    public void setIdade(int novaIdade) {
       if (novaIdade < 0) novaIdade = 0;
       else if (novaIdade > 100) novaIdade = 100;
       idade = novaIdade;
       }

    /**
     * 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(joao);
       System.out.println(maria);
       }
    }
```

E assim, finalmente a nossa classe simples está completa e dentro dos padrões de bons costumes de programação.

5. 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 4: Construção de Classe de Entidade

Prof. Daniel Caetano

Objetivo: Capacitar o aluno para construir classes de entidade básicas.

INTRODUÇÃO

Uma classe de entidade é uma classe cuja principal finalidade no sistema é armazenar informações, isto é, em um sistema orientado a objetos, ela cumpre a função que seria de uma "linha de tabela de banco de dados".

As classes de entidade, entretanto, são responsáveis também pela **validação** dos dados, isto é, elas são responsáveis por garantir que as informações armazenadas sejam válidas. Por exemplo: o cpf de um cliente não deve poder ser um texto qualquer, nem ter um número de dígitos diferentes de 11.

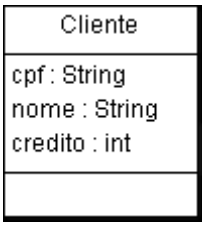
Assim, o objetivo desta aula é iniciar a implementação de nosso cadastro de clientes, através da implementação de uma classe de entidade básica denominada Cliente.

1. CRIAÇÃO DA CLASSE CLIENTE BASE

PASSO 1: Abra o NetBeans e o projeto SisCli, iniciado anteriormente.

PASSO 2: Clique com o botão direito no **pacote siscli** e selecione **Novo > Classe Java** e dê o nome de **Cliente** a ela. Isso criará um novo arquivo de classe chamado **Cliente.java**, que estará automaticamente aberta no editor.

PASSO 3: Iremos agora configurar a classe para que tenha 3 atributos: **cpf**, **nome** e **credito**, conforme indicado no diagrama abaixo:



Todos estes atributos serão privados. O atributo **credito** foi adotado como inteiro porque este valor será armazenado em centavos (R$100,00 será armazenado como 10000). Assim, devemos inserir as seguintes linhas na classe Cliente:

Cliente.java

```
package siscli;

public class Cliente {
    // Atributos Privados
    private String cpf;
    private String nome;
    private int credito;
}
```

PASSO 4: Agora podemos criar um construtor. É possível fazer isso manualmente, mas vamos usar os recursos do NetBeans para acelerar o trabalho. Clique com o botão direito no código e selecione a opção **Inserir Código** que aparece no menu e, em seguida, selecione **Construtor**, o que irá abrir uma janela. Selecione **todos os atributos** e clique em **Gerar**.

PASSO 5: Como se trata de um objeto de entidade, teremos *getters* e *setters* para todos os atributos (cpf, nome e crédito). Vamos criá-los usando os recursos do NetBeans para acelerar o trabalho. Clique com o botão direito no código e selecione a opção **Inserir Código** que aparece no menu e, em seguida, selecione **Getter e setter**, o que irá abrir uma janela. Selecione **todos os atributos** da classe e clique em **Gerar**.

PASSO 6: Para que possamos testar, vamos criar um método **toString** que imprima nosso objeto na forma **nome (cpf)** . Para isso, clique com o botão direito no código e selecione a opção **Inserir Código** que aparece no menu e, em seguida, selecione **toString**, o que irá abrir uma janela. Selecione **cpf** e **nome** e clique em **Gerar**. Isso irá criar um código que você deve modificar para que fique como indicado abaixo.

Cliente.java (método toString)

```
    @Override
    public String toString() {
        return getNome() + " (" + getCpf() + ")";
    }
```

PASSO 7: Vamos agora testar o que foi feito. Edite a classe **Main.java** para que fique como indicado a seguir. Esse código irá criar um objeto da classe Cliente com alguns valores e irá imprimí-lo, para que verifiquemos se tudo está ocorrendo como previsto. Teste e veja o que ocorre.

Main.java

```
package siscli;

public class Main {

    public static void main(String[] args) {

        JPrincipal aplicativo = new Jprincipal();

        Cliente c1 = new Cliente("01234567890","Um Cliente", 100000);
        System.out.println(c1);
    }
}
```

PASSO 8: Para teste, pode ser interessante ter um método que imprima **todos** os atributos. Neste exemplo, usaremos o nome **debug** para este método, e ele deve acessar os valores diretamente, não através dos **getters**, como o toString. Implemente-o como indicado abaixo, na classe Cliente.

Cliente.java (método debug)

```
    public String debug() {
        return cpf + " : " + nome + " : " + credito;
    }
```

PASSO 9: Vamos agora testar o método **debug**. Edite a classe **Main.java** para que fique como indicado a seguir. Esse código irá imprimir o objeto cliente criado anteriormente usando também o médoto debug, para que verifiquemos se tudo está ocorrendo como previsto. Teste e veja o que acontece.

Main.java

```
package siscli;

public class Main {
    public static void main(String[] args) {
        JPrincipal aplicativo = new Jprincipal();

        Cliente c1 = new Cliente("01234567890","Um Cliente", 100000);
        System.out.println(c1);
        System.out.println(c1.debug());
    }
}
```

2. IMPLEMENTANDO VALIDAÇÃO

PASSO 10: Modifique a classe Main conforme indicado abaixo. Execute e observe.

Main.java

```
package siscli;

public class Main {
    public static void main(String[] args) {
        JPrincipal aplicativo = new Jprincipal();

        Cliente c1 = new Cliente("01234567890","Um Cliente", 100000);
        System.out.println(c1);
        System.out.println(c1.debug());

        Cliente c2 = new Cliente(null,"Um Cliente", 100000);
        System.out.println(c2);
    }
}
```

PASSO 11: Para evitar esse tipo de "bobagem" na impressão, é comum fazer com que os getters que devolvem objetos (no caso, getNome e getCpf devolvem Strings) verifiquem se o objeto existe (é difernte de *null*) antes de devolvê-lo. Se ele não existir, no caso de textos, devolvemos uma String vazia. Para isso, modifique os métodos getNome e getCpf:

Cliente.java (método getCpf)

```
    public String getCpf() {
        if (cpf == null) return "";
        return cpf;
```

Cliente.java (método getNome)

```
    public String getNome() {
        if (nome == null) return "";
        return nome;
```

PASSO 12: Modifique a classe Main conforme indicado abaixo. Execute e observe.

Main.java

```
package siscli;

public class Main {
    public static void main(String[] args) {
        JPrincipal aplicativo = new Jprincipal();

        Cliente c1 = new Cliente("01234567890","Um Cliente", 100000);
        System.out.println(c1);
        System.out.println(c1.debug());
        c1.setCpf("aeiou");
        System.out.println(c1);
        System.out.println(c1.debug());
    }
}
```

Você acha que o objeto da classe Cliente deveria deixar que isso ocorresse? É claro que não! O objeto é responsável pela validação dos dados! Assim, vamos validar os dados de crédito, nome e cpf.

PASSO 13: Iniciemos com a validação mais simples: a do crédito. Como o credito é uma variável inteira, ela nunca vale nulo. Assim, só precisamos verificar se o novo crédito não é um valor negativo, já que crédito negativo não faz sentido. Assim, devemos modificar o método setCredito da seguinte forma:

Cliente.java (método setCredito)

```
    public boolean setCredito(int credito) {
        // Se crédito for negativo, vai embora com false.
        if (credito < 0) return false;
        this.credito = credito;
        return true;
    }
```

Observe que o tipo de dado de retorno foi alterado para **boolean**, ou seja, esse método agora responde **false** se houve algum problema na alteração do dado... E responde **true** se a alteração ocorreu com sucesso.

PASSO 14: Vamos agora validar o nome. Como o nome é um objeto String, primeiramente vamos verificar se o valor do novo nome não é **null**, que obviamente será rejeitado. Adicionalmente, vamos verificar se o tamanho do novo nome é menor que 5 caracteres e, nesse caso, também vamos rejeitá-lo. Assim, devemos modificar o método setNome da seguinte forma:

Cliente.java (método setNome)

```
public boolean setNome(String nome) {
    // Se nome muito curto, vai embora com false.
    if (nome == null || nome.length() < 5) return false;
    this.nome = nome;
    return true;
}
```

Observe que é muito importante verificar se o nome é **null** antes de executar o método **length**. A razão é simples: **null nunca poderá executar o método length** e tentar fazê-lo irá causar erro na execução do programa.

PASSO 15: Por último, o mais complicado: validar o CPF. Não faremos uma validação completa, mas seguiremos o seguinte procedimento: primeiro verificaremos se o novo CPF não é null; se for, será rejeitado. Depois, limparemos espaços e caracteres especiais e verificaremos se o comprimento é 11 caracteres; se não for, será rejeitado. Finalmente, verificaremos se cada um dos caracteres é um dígito (numérico); se algum deles não o for, rejeitaremos o CPF. O código que faz isso é apresentado a seguir.

Cliente.java (método setCpf)

```
public boolean setCpf(String cpf) {
    // Se nenhum CPF fornecido, vai embora com erro.
    if (cpf == null) return false;
    // Limpa espaços, pontos e traços
    cpf = cpf.trim();
    cpf = cpf.replaceAll(" ","");
    cpf = cpf.replaceAll("[.-]","");
    // Pega o comprimento do cpf já limpo.
    int cpflen = cpf.length();
    // Se não tiver exatos 11 dígitos, rejeita.
    if (cpflen != 11) return false;
    // Precisa ser composto apenas por números
    for (int i=0; i<cpflen; i++ ) {
        // Se algum dos caracteres não for um dígito numérico,
        // vai embora com erro.
        if (Character.isDigit(cpf.charAt(i)) == false) return false;
    }
    // No caso real, é necessário testar o dígito de verificação!
    // Se chegou aqui, todas as validações foram feitas com sucesso!
    this.cpf = cpf;
    return true;
}
```

PASSO 16: Teste novamente o programa e veja que, agora, o setCpf não permite mais a mudança do valor de CPF para valores claramente inválidos (como "aeiou", por exemplo).

PASSO 17: Modifique, agora, a classe Main como indicado a seguir, execute o programa e veja o que ocorre.

Main.java

```
package siscli;

public class Main {
    public static void main(String[] args) {
        JPrincipal aplicativo = new Jprincipal();

        Cliente c1 = new Cliente("01234567890","Um Cliente", 100000);
        System.out.println(c1);
        System.out.println(c1.debug());
        c1.setCpf("aeiou");
        System.out.println(c1);
        System.out.println(c1.debug());

        Cliente c2 = new Cliente("aeiou","Um Cliente", 100000);
        System.out.println(c2);
        System.out.println(c2.debug());
    }
}
```

PASSO 18: Parece que a nossa classe Cliente ainda não está à prova de balas! Não seria interessante que o construtor da classe cliente **também validasse** os dados fornecidos? Claro que sim! Para isso, modifique o construtor da classe Cliente como indicado.

Cliente.java (método Cliente)

```
    public Cliente(String cpf, String nome, int credito) {
        setCpf(cpf);
        setNome(nome);
        setCredito(credito);
    }
```

PASSO 19: Isso torna a nossa classe mais protegida, mas o NetBeans avisa que o uso de métodos no contrutor da classe é temerário, pois eles podem ser substituídos em classe que extendam a classe Cliente, causando problemas. Para **impedir** isso, modifique os métodos setCpf, setNome e setCredito com a palavra **final**, como indicado abaixo. Isso **proíbe** que esses métodos sejam substituídos (soberscritos) em classes que extendam Cliente.

public **final** boolean setCpf(String cpf)
public **final** boolean setCredito(int credito)
public **final** boolean setNome(String nome)

3. EXCEÇÕES: QUANDO A CRIAÇÃO DO OBJETO FALHA

PASSO 20: Na implementação feita até o momento, quando ocorre uma falha na definição de um dos atributos pelo construtor, aquele atributo simplesmente fica sem um valor. Isso não é exatamente adequado. Poderíamos adotar uma abordagem de definir um

valor padrão, fazendo **if (setNome(nome) == false) setNome("Mané");** . Outra abordagem, a que adotaremos, é a de devolver um código de erro... Mas como, se construtor não pode retornar erros? Bem, usaremos uma EXCEÇÃO para isso.

O primeiro passo é programar uma classe que represente um objeto de erro. Essa classe deve simplesmente guardar o código (ou identificação) do erro e fornecer um método para que este valor seja analisado. Como estaremos programando a exceção para a classe Cliente, chamaremos essa exceção de **ClienteEx**.

Para criá-la, clique com o botão direito no **pacote siscli** e selecione **Novo > Classe Java** e dê o nome de **ClienteEx** a ela. Isso criará um novo arquivo de classe chamado **ClienteEx.java**, que estará automaticamente aberta no editor. Modifique este código de acordo com o indicado a seguir.

ClienteEx.java

```
package siscli;

// Classe que especifica exeções (erros) na criação de objetos Cliente
public class ClienteEx extends Exception {
    // Enumera os tipos de erro válidos
    public static enum Id { ERRO_CPF, ERRO_NOME, ERRO_CREDITO };

    // Atributo que armazenará o código de erro (inicialmente nulo)
    private Id erro = null;

    // Construtor: recebe um código válido e o armazena no atributo
    public ClienteEx(Id umErro) { erro = umErro; }

    // Getter: retorna o código de erro a quem solicitar
    Id getErro() { return erro; }
}
```

Toda exceção deve extender a classe **Exception** do Java. Fora isso, tudo que essa classe faz é guardar um código de erro (que é armazenado no atributo pelo construtor do objeto de erro) e informar esse código de erro quando alguém o solicita, pelo método **getErro**. A única novidade é o tipo de dado **enum**.

Uma **enumeração** é uma sequência de valores inteiros que não definimos explicitamente, apenas indicamos um conjunto de "rótulos". Sempre que precisarmos nos referir a um desses erros, usaremos o rótulo correspondente. Se quisermos criar um erro com um valor **diferente** dos especificados nos rótulos, o **Java não irá permitir**.

PASSO 21: Ok, já temos uma classe para o objeto de erro. Como usá-la no construtor da classe Cliente? Bem, primeiramente precisamos indicar, no constutor da classe Cliente, que ele **joga uma exceção do tipo ClienteEx**. Joga, em inglês, é a palavra **throws** . Assim, o construtor da classe Cliente será, inicialmente, modificado como descrito no código a seguir. Observe.

Cliente.java (método Cliente)

```
    public Cliente(String cpf, String nome, int credito)
            throws ClienteEx {
        setCpf(cpf);
        setNome(nome);
        setCredito(credito);

    }
```

PASSO 22: Agora, para cada **set**, será preciso verificar o resultado e, caso não tenha sido possível executar a alteração (ou seja, o set retornou **false**), devemos **jogar uma nova exceção**. Jogar, em inglês, é throw. Observe como pode ser feito, para o CPF, no código apresentado a seguir.

Cliente.java (método Cliente)

```
    public Cliente(String cpf, String nome, int credito)
            throws ClienteEx {
        if (setCpf(cpf) == false)
            throw (new ClienteEx(ClienteEx.Id.ERRO_CPF));
        setNome(nome);
        setCredito(credito);

    }
```

Observe que quando declaramos o método, dizemos que ele **joga** (**throws**) uma exceção, mas no momento em que vou jogá-la, no if, a instrução é **jogar** (**throw**, sem s). Observe que o que é jogado é, na verdade, um objeto do tipo ClienteEx, construído no exato momento de jogá-lo, com **new ClienteEx(...)**. O parâmetro do construtor deve ser um dos códigos de erro admissíveis; no caso, ERRO_CPF. Por quê preciso indicar **ClienteEx.Id** antes do código ERRO_CPF? Por causa da estrutura de dados que definimos com a classe; ERRO_CPF é um valor que foi definido dentro da enumeração chamada **Id**, que por sua vez foi definida dentro da classe **ClienteEx**. Se não especificarmos o caminho completo, o Java não saberá de que valor estamos falando.

PASSO 23: Podemos, agora, repetir o processo para os outros **sets**, conforme indicado no código a seguir.

Cliente.java (método Cliente)

```
    public Cliente(String cpf, String nome, int credito)
            throws ClienteEx {

        if (setCpf(cpf) == false)
            throw (new ClienteEx(ClienteEx.Id.ERRO_CPF));

        if (setNome(nome)== false)
            throw (new ClienteEx(ClienteEx.Id.ERRO_NOME));

        if (setCredito(credito)== false)
            throw (new ClienteEx(ClienteEx.Id.ERRO_CREDITO));

    }
```

PASSO 24: Ao gravar esse código, descobrimos que agora "surgiram" erros na classe Main. Isso ocorre porque, ao declararmos que o construtor de Cliente **joga uma exceção**, o Java nos **obriga a tratá-la**, algo que não está sendo feito.

O que precisamos é colocar o código de criação dos objetos Cliente dentro de um bloco **try~catch**. Modifique a classe Main como indicado abaixo e verifique os resultados.

Main.java

```
package siscli;

public class Main {
    public static void main(String[] args) {
        JPrincipal aplicativo = new Jprincipal();
        try {
           Cliente c1 = new Cliente("01234567890","Um Cliente", 100000);
           System.out.println(c1);
           System.out.println(c1.debug());
           c1.setCpf("aeiou");
           System.out.println(c1);
           System.out.println(c1.debug());
        } catch (Exception ex) {
           System.out.println("Erro na criação 1: " + ex);
        }

        try {
            Cliente c2 = new Cliente("aeiou","Um Cliente", 100000);
            System.out.println(c2);
            System.out.println(c2.debug());
        } catch (Exception ex) {
            System.out.println("Erro na criação 2: " + ex);
        }
    }
}
```

PASSO 25: O uso da captura da exceção genérica (Exception) permite detectar que um erro ocorreu, mas não exatamente qual foi o erro. Experimenteo modificar o código da classe Main conforme indicado a seguir, isto é, capturando a exceção ClienteEx.

Main.java

```
package siscli;

public class Main {
    public static void main(String[] args) {
        JPrincipal aplicativo = new Jprincipal();

        try {
           Cliente c1 = new Cliente("01234567890","Um Cliente", 100000);
           System.out.println(c1);
           System.out.println(c1.debug());
           c1.setCpf("aeiou");
           System.out.println(c1);
           System.out.println(c1.debug());
        } catch (Exception ex) {
           System.out.println("Erro na criação 1: " + ex);
        }
```

```
        try {
            Cliente c2 = new Cliente("aeiou","Um Cliente", 100000);
            System.out.println(c2);
            System.out.println(c2.debug());
        } catch (ClinteEx ex) {
            System.out.println("Erro na criação 2: " + ex);
            if (ex.getErro() == ClienteEx.Id.ERRO_CPF)
                System.out.println("CPF inválido!");
            else if (ex.getErro() == ClienteEx.Id.ERRO_NOME)
                System.out.println("Nome inválido!");
            if (ex.getErro() == ClienteEx.Id.ERRO_CREDITO)
                System.out.println("Crédito inválido!");
        }
    }
}
```

Com isso finalizamos a criação de nossa classe de entidade básica; para adicionar novos atributos, é necessário modificar a classe de exceção com novos códigos de erro na enumeração, além de modificar o construtor, debug e, claro, criar os getters e setters com as respectivas verificações.

4. ATIVIDADE (NOTA E REPOSIÇÃO) - INDIVIDUAL

a) Entregar IMPRESSO
b) Entregar na próxima aula (impreterivelmente!)
c) INDIVIDUAL
d) INDIVIDUAL :)

Considerando a classe Cliente desta aula:

i) Acrescente, no cliente, o atributo **sexo** do tipo **int**. O valor 1 deve significar se tratar de homem e o valor 2 deve indicar se tratar de mulher.

ii) Crie o erro do tipo ERRO_SEXO na exceção de cliente.

iii) Crie o *getter* e o *setter* para esse novo atributo, fazendo a validação (apenas valores 1 e 2 são aceitos).

iv) Modifique o construtor de maneira que o sexo seja configurado na criação do objeto.

v) Modifique o método "debug" de maneira que ele apresente também o sexo do cliente.

Unidade 5: Edição de Classe de Entidade

Prof. Daniel Caetano

Objetivo: Capacitar o aluno para construir janelas de edição do tipo JDialog.

INTRODUÇÃO

Na aula passada vimos como construir uma classe de entidade. Entretanto, a edição do objeto só podia ser feita no código, isto é, se quiséssemos modificar um objeto, éramos obrigados a modificar o código e executar o programa novamente. Certamente não é assim que os programas reais funcionam, não é?

Bem, para que possamos modificar um objeto de entidade qualquer, incluindo a nossa classe cliente, devemos criar uma **janela de diálogo** com o usuário. Uma janela de diálogo é uma janela que **conversa** com o usuário para obter informações necessárias para realizar uma modificação em uma classe de entidade (ou configurar um processamento).

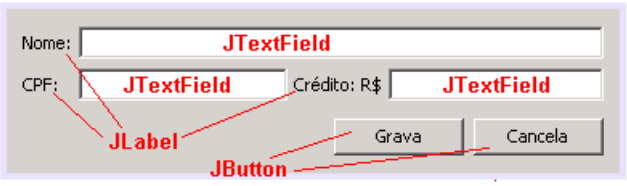
Nessa aula nós iremos, então, construir uma janela de diálogo de edição para nosso objeto cliente. O nome que daremos a esta janela é **ClienteDialog** e ela será uma especialização de uma classe que já existe no Java, chamada **JDialog**.

1. CRIAÇÃO DA JANELA DE DIÁLOGO

PASSO 1: Abra o NetBeans e o projeto SisCli, iniciado anteriormente.

PASSO 2: Clique com o botão direito no **pacote siscli** e selecione **Novo > Formulário JDialog** e dê o nome de **ClienteDialog** a ela. Isso criará um novo arquivo de classe chamado **ClienteDialog.java**, que estará automaticamente aberta no editor, na forma de Projeto (edição visual).

PASSO 3: Construa a seguinte janela no seu editor (os textos em vermelho indicam os tipos de elementos, todos presentes na palheta em **Controles Swing**):



PASSO 4: Selecione cada um dos elementos e configure, na área de **Propriedades - Código** (canto inferior direito da janela do NetBeans) o **Nome da Variável** de cada um deles para os valores indicados na imagem abaixo:

PASSO 5: Entre, agora, no modo de Código-Fonte e observe o código criado pelo NetBeans. Ele é um bocado complicado. Vamos simplificá-lo. Primeiramente, procure o método **main** da classe ClienteDialog e modifique-a como indicado abaixo, lembrando que você deve apagar todas as linhas riscadas no código.

ClienteDialog.java (método main)

```
/**
 * @param args the command line arguments
 */
public static void main(String args[]) {
    java.awt.EventQueue.invokeLater(new Runnable() {
        public void run() {
            ClienteDialog dialog = new ClienteDialog(new javax.swing.JFrame(), true);
            dialog.addWindowListener(new java.awt.event.WindowAdapter() {
                public void windowClosing(java.awt.event.WindowEvent e) {
                    System.exit(0);
                }
            });
            dialog.setVisible(true);
        }
    });
}
```

PASSO 5: Agora vamos modificar o construtor. O construtor original recebe como parâmetros a janela pai (uma referência para a janela principal do nosso aplicativo) e um parâmetro que indica se a janela é modal ou não. No nosso caso, a janela **sempre será modal**, de maneira que não precisaremos desse parâmetro. Adicionalmente, devemos indicar que a janela deve ser centralizada e ficar visível após a inicialização dos componentes:

ClienteDialog.java (método ClienteDialog)

```
/**
 * Janela de Edição de Clientes.
 * Permite a edição de todos os atributos de um cliente.
 * @author djcaetano
 */
public ClienteDialog(java.awt.Frame parent) {
    // Iniciliza JDialog original
    super(parent, modal);
    // Cria botões a campos na tela
    initComponents();

    // Centraliza Janela na Janela Pai
    setLocationRelativeTo(parent)
    // Torna janela visível
    setVisible(true)
}
```

PASSO 6: Bem, ocorre que a finalidade da nossa janela é editar um cliente, e ela precisa saber qual é o cliente em edição no momento. Então, precisamos criar um atributo na janela (lembre-se que ela é um objeto como outro qualquer!) que armazene um objeto do tipo cliente. Observe no código abaixo:

ClienteDialog.java (declaração da classe)

```
package siscli;

/**
 * Janela de Edição de Clientes.
 * Permite a edição de todos os atributos de um cliente.
 * @author djcaetano
 */
public class ClienteDialog extends javax.swing.JDialog {

    // Atributos
    private Cliente clienteEmEdicao; // Atributo para armazenar o cliente sendo editado

   /**
    * Construtor da janela de edição de cliente.
    * @param parent Janela principal da aplicação.
    */
    public ClienteDialog(java.awt.Frame parent) {
        // Inicializa a janela JDialog
        super(parent, true);
```

PASSO 7: O cliente a ser editado deve ser passado para a janela quando ela é construída. Assim, o construtor deve receber um cliente a ser editado no construtor, e guardá-lo no atributo:

ClienteDialog.java (método ClienteDialog)

```
   /**
    * Construtor da janela de edição de cliente.
    * @param parent Janela principal da aplicação.
    * @param umCliente Cliente a ser editado pela janela.
    */
    public ClienteDialog(java.awt.Frame parent, Cliente umCliente) {
        // Inicializa a janela JDialog
        super(parent, true);
        // Cria botões a campos na tela
        initComponents();

        // Se objeto cliente for null, vai embora fechando a janela
        if (umCliente == null) {
            dispose();
            return;
            }
        // Só chega aqui se o cliente existe
        clienteEmEdicao = umCliente;

        // Centraliza Janela na Janela Pai
        setLocationRelativeTo(parent)
        // Torna janela visível
        setVisible(true)
        }
```

PASSO 8: Agora precisamos preencher os campos da janela com as informações que vieram de dentro de nosso objeto cliente. Isso é simples: usaremos os métodos **getters** do objeto para pegar os valores e o método **setText** de cada campo para modificar seu valor. Observe o código a seguir:

ClienteDialog.java (método ClienteDialog)

```
/**
 * Construtor da janela de edição de cliente.
 * @param parent Janela principal da aplicação.
 * @param umCliente Cliente a ser editado pela janela.
 */
public ClienteDialog(java.awt.Frame parent, Cliente umCliente) {
    // Inicializa a janela JDialog
    super(parent, true);
    // Cria botões a campos na tela
    initComponents();

    // Se objeto cliente for null, vai embora fechando a janela
    if (umCliente == null) {
        dispose();
        return;
        }
    // Só chega aqui se o cliente existe
    clienteEmEdicao = umCliente;

    // Ajusta o valor dos campos nome e CPF
    cNome.setText(clienteEmEdicao.getNome());
    cCpf.setText(clienteEmEdicao.getCpf());
    // Ajusta o valor do campo crédito: observe a divisão por 100,
    // Para que na janela o valor apareça em reais, ao invés de centavos.
    double creditoEmCentavos = clienteEmEdicao.getCredito();
    cCredito.setValue(creditoEmCentavos/100);

    // Centraliza Janela na Janela Pai
    setLocationRelativeTo(parent)
    // Torna janela visível
    setVisible(true)
    }
```

PASSO 9: O próximo passo é inserir código no botão Cancela. O botão cancela deve simplesmente fechar a janela sem fazer nada. Volte à interface de "Projeto" e clique duas vezes no botão "Cancela". Isso irá criar o método **bCancelaActionPerformed**. Edite-o inicialmente da seguinte forma:

ClienteDialog.java (método bCancelaActionPerformed)

```
/**
 * Ação do botão Cancela.
 * @param evt Evento de ação.
 */
private void bCancelaActionPerformed(java.awt.event.ActionEvent evt) {
    // Solicita o fechamento da janela.
     dispose();
}
```

PASSO 10: O próximo passo é inserir código no botão Grava. Volte à interface de "Projeto" e clique duas vezes no botão "Grava". Isso irá criar o método **bGravaActionPerformed**. Sua função é pegar as informações de cada campo e inseri-las de volta no objeto de entrada, verificando se houve algum erro.

Cada um dos valores (nome, cpf e crédito) será testado com relação à modificação (retorno em true ou false) e uma mensagem de erro é mostrada de acordo... além de selecionar o campo em questão. No caso do valor do crédito, também é preciso verificar se não houve

erro na conversão de texto para número. Para conseguir isso tudo, edite o método da seguinte forma:

ClienteDialog.java (método bGravaActionPerformed)

```
    /**
     * Ação do botão Gravar.
     * @param evt Evento de ação.
     */
    private void bGravaActionPerformed(java.awt.event.ActionEvent evt) {
       // Se o nome digitado no campo for rejeitado pelo objeto cliente...
        if (clienteEmEdicao.setNome(cNome.getText()) == false) {
            // Indica erro...
            JOptionPane.showMessageDialog(this,"Nome de cliente inv ílido",
                    "Erro no preenchimento!",JOptionPane.ERROR_MESSAGE);
            // E coloca o foco no campo de nome.
            cNome.requestFocusInWindow();
            return;
        }
        // Aqui se a modificação do nome foi processada com sucesso...
        // Se o CPF for válido, troca o CPF do cliente pelo novo.
        if ( clienteEmEdicao.setCpf(cCpf.getText()) == false) {
            // Se cpf inválido... Mostra mensagem de erro...
            JOptionPane.showMessageDialog(this,"CPF inválido",
                    "Erro no preenchimento!",JOptionPane.ERROR_MESSAGE);
            // E joga o foco para o campo do CPF.
            cCpf.requestFocusInWindow();
            return;
        }
        // Se chegou aqui, é porque o CPF foi alterado com sucesso.
        // Finalmente, tenta converter o campo crédito para um número
        try {
            // Se conseguir converter, altera o crédito do cliente.
            int credito = Integer.parseInt(cCredito.getTextLimpo());
            // Se o valor for inválido, provoca exceção
            if ( clienteEmEdicao.setCredito(credito) == false) {
                throw new NumberFormatException("Valor inválido.");
            }
        }
        // Se o valor do crédito era inválido...
        catch (NumberFormatException ex) {
            // Mostra mensagem de erro...
            JOptionPane.showMessageDialog(this,"Valor de crédito inválido",
                    "Erro no preenchimento!",JOptionPane.ERROR_MESSAGE);
            // E joga o foco no campo de crédito.
            cCredito.requestFocusInWindow();
            return;
        }
        // Tudo finalizado, solicitamos o fechamento da janela.
        dispose();
    }
```

PASSO 11: Com isso a janela já funciona! Vamos testar? Modifique o método **main** na classe **Main** como se segue e veja!

Main.java (método main)

```
  public static void main(String[] args) {
        JPrincipal aplicativo = new JPrincipal();
        try {
            // Cria o objeto cliente
            Cliente c1 = new Cliente("01234567890", "Um Cliente", 0);
            System.out.println(c1);

            // Cria a janela
            ClienteDialog janela = new ClienteDialog(aplicativo,c1);
```

```
            // Imprime cliente depois da edição
            System.out.println(c1);

        } catch (ClienteEx ex) {
            if (ex.getErro() == ClienteEx.Id.ERRO_CPF)
                System.out.println("CPF Inválido");
            if (ex.getErro() == ClienteEx.Id.ERRO_NOME)
                System.out.println("Nome Inválido");
            if (ex.getErro() == ClienteEx.Id.ERRO_CREDITO)
                System.out.println("Credito Inválido");
        }
    }
```

2. TORNANDO A JANELA MAIS PROFISSIONAL

PASSO 12: Para deixar nossa janela mais completa, precisamos apenas definir alguns detalhes. O primeiro deles tornar o comportamento da janela mais profissional, fazendo com que o texto de um campo seja totalmente selecionado quando ele recebe o foco (clique do mouse). Para isso, voltemos ao modo de "Projeto".

Pemeiramente selecione o campo de texto **cNome**. Agora, na área de **Propriedades - Eventos** procure pelo evento **focusGained**. Nesta opção, clique com o mouse na setinha para baixo da combo box e selecione a opção **cNomeFocusGained**. Isso irá criar um codigo que será executado **sempre** que o campo cNome receber o foco (seja por clique do mouse, seja por tabulação). Nesse método, escreva o seguinte código:

ClienteDialog.java (método cNomeFocusGained)

```
    /**
     * Quando o campo de Nome ganhar foco, seu texto será selecionado.
     * @param evt Evento de foco.
     */
    private void cNomeFocusGained(java.awt.event.FocusEvent evt) {
        cNome.selectAll();
    }
```

Repita o procedimento para os campos cCpf e cCredito (você precisa criar o evento de um em um, usando a interface de projeto! Caso contrário, **não irá funcionar**!):

ClienteDialog.java (método cCpfFocusGained)

```
    /**
     * Quando o campo de Cpf ganhar foco, seu texto será selecionado.
     * @param evt Evento de foco.
     */
    private void cCpfFocusGained(java.awt.event.FocusEvent evt) {
        cCpf.selectAll();
    }
```

ClienteDialog.java (método cCreditoFocusGained)

```
    /**
     * Quando o campo de Credito ganhar foco, seu texto será selecionado.
     * @param evt Evento de foco.
     */
    private void cCreditoFocusGained(java.awt.event.FocusEvent evt) {
        cCredito.selectAll();
    }
```

PASSO 13: Finalmente, só falta um toque final: muitas vezes o programador desejará saber se o usuário editou ou não o cliente, de acordo com o botão que foi apertado na janela. Para conseguir isso, criaremos um atributo chamado **editado** na classe da janela, isto é, na classe ClienteDialog, que iniciará com o valor padrão **false** (que indica que a edição **não** ocorreu):

ClienteDialog.java (declaração da classe)

```
package siscli;

/**
 * Janela de Edição de Clientes.
 * Permite a edição de todos os atributos de um cliente.
 * @author djcaetano
 */
public class ClienteDialog extends javax.swing.JDialog {

    // Atributos
    private Cliente clienteEmEdicao; // Atributo para armazenar o cliente sendo editado
    private boolean editado = false; // Resultado da edição (false por padrão)
```

PASSO 14: Agora vamos criar o *getter* pra esse atributo. Logo depois do final do método construtor, clique com o botão direito no editor e selecione **Inserir Código... > Getter**. Na janela que vai aparecer, selecione **apenas** a variável **editado : boolean**. O NetBeans vai criar um método chamado **isEditado()**, que é o nome padrão para getters de variaveis boolean (ao invés de getVariável, usa-se isVariável).

PASSO 15: Por enquanto, o método isEditado está retornando sempre **false**, uma vez que o atributo **editado** foi criado com valor inicial false e, em nenhuma situação, esse valor foi alterado. O único lugar onde o valor de **editado** deve mudar para **true** é no final do método **bGravaActionPerformed**, visto que só chegando naquele lugar o objeto terá sido corretamente editado. Observe o código a seguir:

ClienteDialog.java (método bGravaActionPerformed)

```
    /**
     * Ação do botão Gravar.
     * @param evt Evento de ação.
     */
    private void bGravaActionPerformed(java.awt.event.ActionEvent evt) {
       // Se o nome digitado no campo for rejeitado pelo objeto cliente...
        if (clienteEmEdicao.setNome(cNome.getText()) == false) {
            // Indica erro...
            JOptionPane.showMessageDialog(this,"Nome de cliente inv ílido",
                    "Erro no preenchimento!",JOptionPane.ERROR_MESSAGE);
            // E coloca o foco no campo de nome.
            cNome.requestFocusInWindow();
            return;
        }
        // Aqui se a modificação do nome foi processada com sucesso...
        // Se o CPF for válido, troca o CPF do cliente pelo novo.
        if ( clienteEmEdicao.setCpf(cCpf.getText()) == false) {
            // Se cpf inválido... Mostra mensagem de erro...
            JOptionPane.showMessageDialog(this,"CPF inválido",
                    "Erro no preenchimento!",JOptionPane.ERROR_MESSAGE);
            // E joga o foco para o campo do CPF.
            cCpf.requestFocusInWindow();
            return;
        }
```

```
        // Se chegou aqui, é porque o CPF foi alterado com sucesso.
        // Finalmente, tenta converter o campo crédito para um número
        try {
            // Se conseguir converter, altera o crédito do cliente.
            int credito = Integer.parseInt(cCredito.getTextLimpo());
            // Se o valor for inválido, provoca exceção
            if ( clienteEmEdicao.setCredito(credito) == false) {
                throw new NumberFormatException("Valor inválido.");
            }
        }
        // Se o valor do crédito era inválido...
        catch (NumberFormatException ex) {
            // Mostra mensagem de erro...
            JOptionPane.showMessageDialog(this,"Valor de crédito inválido",
                    "Erro no preenchimento!",JOptionPane.ERROR_MESSAGE);
            // E joga o foco no campo de crédito.
            cCredito.requestFocusInWindow();
            return;
        }
        // Finalmente, chegando aqui, significa que todos os atributos foram
        // atualizados com sucesso e, assim, indica-se TRUE como sendo o
        // novo resultado da operação...
        editado = true;
        // E solicitamos o fechamento da janela.
        dispose();
    }
```

PASSO 16: Vamos testar? Modifique o método **main** na classe **Main** como se segue e veja!

Main.java (método main)

```
  public static void main(String[] args) {
        JPrincipal aplicativo = new JPrincipal();
        try {
            // Cria o objeto cliente
            Cliente c1 = new Cliente("01234567890", "Um Cliente", 0);
            System.out.println(c1);

            // Cria a janela
            ClienteDialog janela = new ClienteDialog(aplicativo,c1);
            // Se o resultado for true, cliente foi editado
            if ( janela.isEditado() == true) {
                System.out.println("Cliente editado com sucesso!");
            }
            else {
                System.out.println("Edição cancelada!");
            }

            // Imprime cliente depois da edição
            System.out.println(c1);

        } catch (ClienteEx ex) {
            if (ex.getErro() == ClienteEx.Id.ERRO_CPF)
                System.out.println("CPF Inválido");
            if (ex.getErro() == ClienteEx.Id.ERRO_NOME)
                System.out.println("Nome Inválido");
            if (ex.getErro() == ClienteEx.Id.ERRO_CREDITO)
                System.out.println("Credito Inválido");
        }
    }
```

Com isso, está finalizada nossa janela de edição de objetos de entidade!

3. ATIVIDADE (NOTA) - EM TRIO

a) Entregar pelo e-mail daniel@caetano.eng.br
b) Entregar daqui DUAS aulas (daqui 14 dias!)

A classe dessa aula considerou uma classe Cliente simples, sem o atributo de **sexo**. Na aula anterior, foi deixada uma atividade para que fosse acrescentado este atributo na classe Cliente.

Pois bem, considerando a classe Cliente modificada, já com a inclusão do atributo sexo, modifique a janela de edição para que considere também o sexo!

i) Procure na Internet como manipular campos JRadioButton no NetBeans (DICA: consulte estes sites, para começar: http://met.al/md1 e http://met.al/md2)
ii) No modo de Projeto da janela, crie campos JRadioButton (Caixa de Opção), de maneira que se chamem, no código, **cMasculino** e **cFeminino**.
iii) Modifique o construtor para que ele inicialize os campos cMasculino e cFeminino de maneira correta.
iv) Modifique o método bGravaActionPerformed de maneira que ele altere o sexo do objeto cliente de acordo com a seleção das JRadioBox.

Unidade 6: Programação de Banco de Dados com Java

Prof. Daniel Caetano

Objetivo: Adicionar suporte a banco de dados em uma aplicação Java.

INTRODUÇÃO

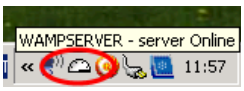
Nas aulas anteriores apresentamos os passos iniciais de nosso sistema, incluindo a criação de uma janela principal, de uma classe de entidade Cliente e de uma janela de edição de entidades do tipo Cliente. Agora veremos a parte que complementa esse conjunto, que é o acesso ao banco de dados, onde os objetos de entidade são armazenados para futuro uso.

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 MySQL para atingir o propósito específico da disciplina.

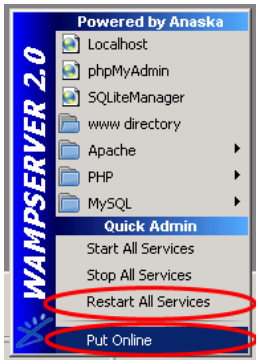
1. CONSTRUINDO O BANCO DE DADOS DA APLICAÇÃO NO MySQL

Existem diversas formas de realizar este mesmo procedimento. Este tutorial apresenta uma delas.

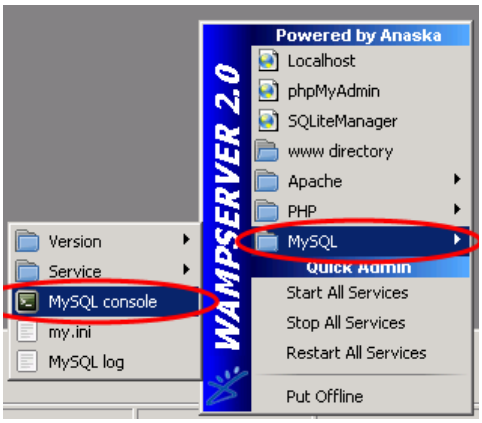
PASSO 1: Inicie o aplicativo **Start WampServer**, disponível em ***Iniciar* > *Todos os Programas* > *WampServer* > *Start WampServer***. Se tudo estiver corretamente configurado, no canto inferior direito da tela, próximo ao relógio do sistema, aparecerá mais um ícone, que parece ser um velocímetro, como o destacado na figura a seguir.



Caso ele apareça com uma parte vermelha ou amarela, clique com o botão esquerdo no ícone do velocímetro e use as opções "Restart All Services" e, depois, "Put Online", como indicado na figura a seguir.



PASSO 2: Já com o WampServer online - o que significa que o MySQL também estará online -, vamos iniciar uma janela do MySQL. Clicando novamente com o botão esquerdo no ícone do velocímetro, selecione a opção ***MySQL > MySQL console***, como indicado na figura a seguir.



Isso abrirá uma janela preta solicitando o password do administrador (root). Na instalação padrão, esse password é vazio, então **simplesmente aperte a tecla ENTER** e você estará no prompt de comandos do MySQL.

PASSO 3: Agora é o momento de criar um banco de dados específico para nossa aplicação. Vamos chamá-lo de **siscli**. Para conseguir isso, digite o comando abaixo no prompt do MySQL:

```
CREATE DATABASE siscli;
```

PASSO 4: Agora, vamos autorizar um usuário chamado **siscli** a usar este banco de dados, com permissão total (ALL PRIVILEGES), acessando pelo computador local (localhost), com password também **siscli**. Para isso, digite o comando abaixo no prompt do MySQL:

```
GRANT ALL PRIVILEGES ON siscli.* TO 'sisclir'@'localhost' IDENTIFIED BY 'siscli';
```

PASSO 5: Vamos agora indicar ao MySQL que os próximos comandos se referem especificamente ao banco de dados **siscli**. Para isso usamos o comando USE, conforme indicado a seguir. Digite-o no prompt do MySQL exatamente como especificado.

```
USE siscli;
```

PASSO 6: Chegou o momento de criarmos a tabela **clientes**, que irá armazenar os dados de nossos objetos. Como nossos objetos possuem três atributos (cpf, nome e credito), teremos três colunas nesta tabela, conforme indicado a seguir.

cpf : PK, CHAR(11)	nome : VARCHAR(200)	credito : INT
...	...	...

Para conseguir criar essa tabela, iremos usar o comando CREATE TABLE do SQL, conforme indicado a seguir. Digite o comando no prompt do MySQL exatamente como especificado.

```
CREATE TABLE clientes (cpf CHAR(11) NOT NULL, nome VARCHAR(200), credito INT, PRIMARY KEY(cpf));
```

PASSO 7: Nosso serviço está finalizado, mas antes de começarmos a modificar a aplicação, vamos nos certificar de que o banco de dados está funcionando. Para isso, vamos inserir um cliente na tabela:

```
INSERT INTO clientes VALUES ("01234567890", "Zé da Silva", 100000);
```

PASSO 8: Verifiquemos se o cliente foi armazenado na tabela:

```
SELECT * FROM clientes;
```

PASSO 9: Tendo ele aparecido, vamos apagá-lo:

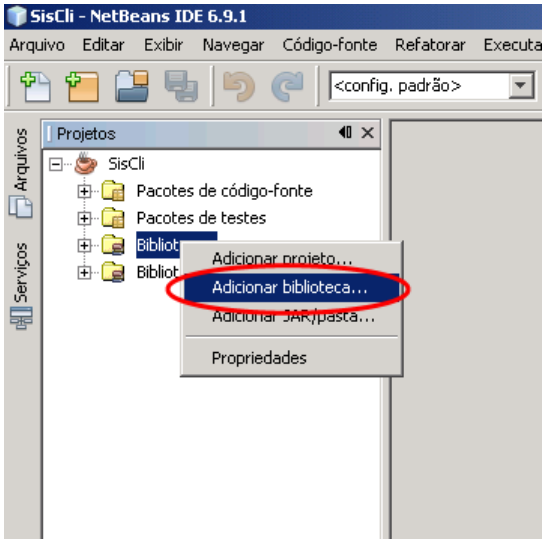
```
DELETE FROM clientes;
```

ATENÇÃO: NÃO FECHE A JANELA DO MySQL AINDA! Iremos usá-la daqui alguns instantes!

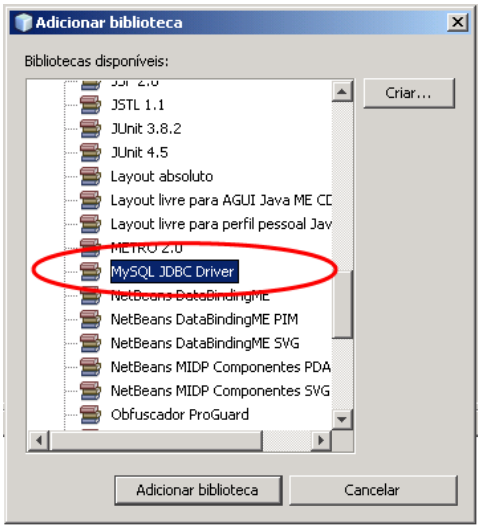
2. AJUSTANDO O PROJETO JAVA PARA ENTENDER O MySQL

PASSO 10: Abra o projeto SisCli da última aula.

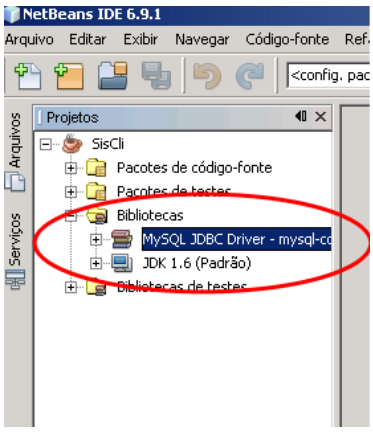
PASSO 11: Como iremos conectar no banco de dados MySQL, precisamos acrescentar em nosso projeto a biblioteca de acesso ao MySQL em nosso ambiente de desenvolvimento. Por sorte, essa biblioteca já está disponível no NetBeans, então sua adição é bastante simples: na área de projeto, clique com o botão direito no item **Bibliotecas**, e selecione a opção **Adicionar Biblioteca...** no menu. Observe na figura a seguir.



PASSO 12: Na janela que se abrirá, selecione a opção "MySQL JDBC Driver", conforme indicado na figura a seguir, e clique no botão "Adicionar Biblioteca".



PASSO 13: Como resultado, a pastinha **Bibliotecas** do projeto deve ter, agora, o seguinte conteúdo:



3. ACESSO AO BANCO DE DADOS

Independente do banco de dados e da linguagem, o acesso ao banco de dados envolve sempre três etapas: **conexão**, **busca** e **desconexão**. Vejamos como executar essas tarefas

3.1. Criando uma Classe de Conexão

PASSO 14: Como a realização da conexão envolve nomes de usuário, password, endereços etc... é conveniente que a realização da conexão em si fique centralizada em um único lugar. Assim, vamos criar uma classe chamada Conexao, cuja finalidade é, simplesmente, conectar ao banco de dados, devolvendo uma referência para um objeto do tipo Connection do Java.

Sendo assim, clique com o botão direito no pacote **siscli** e selecione Nova > **Classe Java**. Dê o nome de **Conexao** a esta classe.

PASSO 15: Observe, entretanto, que essa classe que iremos criar tem uma única função: conectar ao banco. Não faz sentido, portanto, criar objetos dessa classe. Sendo assim, iremos configurar essa classe com seu construtor como **private**, impedindo que objetos sejam criados. Isso pode ser feito como se segue:

Conexao.java

```
package siscli;

/**
 * Classe base para criação de conexões com o banco de dados
 * @author djcaetano
 */
public class Conexao {

   // Construtor privado
    private Conexao() {
    }
}
```

PASSO 16: Agora precisamos criar os atributos nos quais serão armazenados os valores de conexão (nome, password etc.). A razão para criá-los é facilitar mudanças futuras. Observe o código a seguir:

Conexao.java

```
package siscli;

/**
 * Classe base para criação de conexões com o banco de dados
 * @author djcaetano
 */
public class Conexao {
    // Endereço (URL) do servidor de banco de dados.
    private static String url = "localhost";
    // Nome do database a ser utilizado.
    private static String db = "siscli";
```

```
    // Nome do usuário para login.
    private static String user = "siscli";
    // Senha do usuário para login.
    private static String pass = "siscli";

   // Construtor privado
    private Conexao() {
    }
}
```

NOTA: todos os atributos precisam ser precedidos da palavra "static", para indicar que eles podem ser usados **mesmo sem a criação de objetos**. A palavra **static** significa que estamos criando atributos que pertencem à classe e que, portanto, não exigem a existência de objetos para serem usados!

PASSO 17: Agora precisamos criar o método que realiza a conexão em si. Basicamente, ele poderia ser definido como se segue:

Conexao.java

```
package siscli;
import java.sql.*;

/**
 * Classe base para criação de conexões com o banco de dados
 * @author djcaetano
 */
public class Conexao {
    // Endereço (URL) do servidor de banco de dados.
    private static String url = "localhost";
    // Nome do database a ser utilizado.
    private static String db = "siscli";
    // Nome do usuário para login.
    private static String user = "siscli";
    // Senha do usuário para login.
    private static String pass = "siscli";

   // Construtor privado
    private Conexao() {
    }

    /**
     * Realiza a conexão com banco de dados
     * Se houver problemas, indica mensagem de erro no prompt.
     * @return Link de conexao. Null se não for possível conectar.
     */
    public static Connection conecta() {
        // verifica existência de driver.
        Class.forName("com.mysql.jdbc.Driver");
        // Realiza conexão
        String endereco = "jdbc:mysql://" + url + "/" + db;
        return DriverManager.getConnection(endereco,user,pass);
    }
}
```

PASSO 18: Entretanto, o Java nos obriga a tratar erros que podem ocorrer ao criar a conexão, que são: driver inexistente (erro referente à linha Class.forName...) e erro de

conexão propriamente dito (erro da linha DriverManager.getConnection...). Assim, o código acima deve ser modificado da seguinte forma:

Conexao.java

```
package siscli;
import java.sql.*;

/**
 * Classe base para criação de conexões com o banco de dados
 * @author djcaetano
 */
public class Conexao {
    // Endereço (URL) do servidor de banco de dados.
    private static String url = "localhost";
    // Nome do database a ser utilizado.
    private static String db = "siscli";
    // Nome do usuário para login.
    private static String user = "siscli";
    // Senha do usuário para login.
    private static String pass = "siscli";

   // Construtor privado
    private Conexao() {
    }

    /**
     * Realiza a conexão com banco de dados
     * Se houver problemas, indica mensagem de erro no prompt.
     * @return Link de conexao. Null se não for possível conectar.
     */
    public static Connection conecta() {
        try {
            // verifica existência de driver.
            Class.forName("com.mysql.jdbc.Driver");
            // Realiza conexão
            String endereco = "jdbc:mysql://" + url + "/" + db;
            return DriverManager.getConnection(endereco,user,pass);
        }
        // Se não tem driver correto, informa erro.
        catch (ClassNotFoundException ex) {
            System.err.println("Driver do BD não encontrado.");
            System.err.println("Buscando Driver: MySQL");
        }
        // Se não coneguiu conectar, informa erro.
        catch (SQLException ex) {
            System.err.println("Não foi possível conectar ao BD.");
            System.err.println("Driver:   MySQL");
            System.err.println("Servidor: " + url);
            System.err.println("Database: " + db);
            System.err.println("Usuário:  " + user);
            System.err.println("Password: ********");
            }
        return null;
    }
}
```

3.2. Criando uma Classe de Persistência para o Cliente

A imlementação de buscas no Banco de Dados será feita segundo um padrão criado pela Microsoft, denominado Data Access Objects (DAO), com algumas simplificações. Esse padrão propõe que, para cada classe de entidade, exista uma classe específica para acesso ao banco de dados. No caso de nossa classe Cliente, a classe DAO relacionada será a classe ClienteDAO.

PASSO 19: Clique com o botão direito no pacote **siscli** e selecione Nova > **Classe Java**. Dê o nome de **ClienteDAO** a esta classe.

PASSO 20: Observe, entretanto, que também essa classe realiza funções que não exigem a criação de objetos. Suas funções são:

a) Armazenar um objeto Cliente no banco de dados.

b) Buscar objetos cliente no banco de dados. Sendo assim, iremos configurar também essa classe com seu construtor como **private**, impedindo que objetos sejam criados. Isso pode ser feito como se segue:

ClienteDAO.java

```
package siscli;

/**
 * Responsável por armazenar e recuperar clientes do banco de dados.
 * Essa classe torna o uso do BD "transparente" para o resto do sistema.
 * Seus métodos devem ser construídos usando linguagem SQL padrão, para
 * que seja compatível com qualquer banco de dados existente.
 * @author djcaetano
 */
public class ClienteDAO {

    // Construtor Privado
    private ClienteDAO() {
    }
}
```

PASSO 21: Agora vamos criar o método **adiciona**, que recebe um objeto cliente para ser armazenado e retorna true se a adição ocorreu com sucesso. Inicialmente criaremos o código assim:

ClienteDAO.java

```
package siscli;

/**
 * Responsável por armazenar e recuperar clientes do banco de dados.
 * Essa classe torna o uso do BD "transparente" para o resto do sistema.
 * Seus métodos devem ser construídos usando linguagem SQL padrão, para
 * que seja compatível com qualquer banco de dados existente.
 * @author djcaetano
 */
public class ClienteDAO {
```

```
    // Construtor Privado
    private ClienteDAO() {
    }

    /**
     * Adiciona/Atualiza um cliente no banco de dados.
     * @param umCliente Objeto de cliente a ser armazenado.
     * @return True se cliente foi armazenado com sucesso.
     */
    public static boolean adiciona(Cliente umCliente) {

        return false;
    }
}
```

PASSO 22: Agora, vamos adicionar o código da conexão e desconexão:

ClienteDAO.java

```
package siscli;
import java.sql.*;

/**
 * Responsável por armazenar e recuperar clientes do banco de dados.
 * Essa classe torna o uso do BD "transparente" para o resto do sistema.
 * Seus métodos devem ser construídos usando linguagem SQL padrão, para
 * que seja compatível com qualquer banco de dados existente.
 * @author djcaetano
 */
public class ClienteDAO {

    // Construtor Privado
    private ClienteDAO() {
    }

    /**
     * Adiciona/Atualiza um cliente no banco de dados.
     * @param umCliente Objeto de cliente a ser armazenado.
     * @return True se cliente foi armazenado com sucesso.
     */
    public static boolean adiciona(Cliente umCliente) {
        // Pega a conexao atual.
        Connection conexao = Conexao.conecta();
        // Se não existe conexão... retorna false.
        if (conexao == null) return false;


        // Se chegou aqui, não foi possível realizar a atualização.
        return false;
    }
```

PASSO 23: Entre a conexão e a desconexão precisaremos agora fazer duas coisas: a primeira é tentar atualizar o banco de dados, com UPDATE, caso o cliente já exista no banco. A segunda coisa é realizar um INSERT, caso o UPDATE falhe (o que indica que o cliente é novo). Vamos realizar essa tarefa por partes. Primeiramente, iremos tentar atualizar um cliente existente:

ClienteDAO.java (método adiciona)

```
/**
 * Adiciona/Atualiza um cliente no banco de dados.
 * @param umCliente Objeto de cliente a ser armazenado.
 * @return True se cliente foi armazenado com sucesso.
 */
public static boolean adiciona(Cliente umCliente) {
    // Pega a conexao atual.
    Connection conexao = Conexao.conecta();
    // Se não existe conexão... retorna false.
    if (conexao == null) return false;
    // Garante que textos a serem inseridos não contêm aspas
    // para evitar SQL Injection.
    String cpf = umCliente.getCpf();
    cpf = cpf.replaceAll("'","");
    String nome = umCliente.getNome();
    nome = nome.replaceAll("'","");
    // Primeiramente, tenta fazer um update!
    try {
        // Cria a transação
        Statement transacao = conexao.createStatement();
        // Cria a query de update...
        String query  = "UPDATE clientes SET ";
        query += "nome = '" + nome + "', ";
        query += "credito = " + umCliente.getCredito();
        query += " WHERE ";
        query += "cpf LIKE '" + cpf + "'";
        // Excute update retorna número de linhas modificadas;
        transacao.executeUpdate(query);
        // Finaliza transação
        transacao.close();
        return true;
    }
    // se houve algum erro nas transações de SQL...
    catch (SQLException ex) {
        // Não faz nada... código apenas para debug
        //System.err.println(ex);
    }
    // Se chegou aqui, não foi possível realizar a atualização.
    return false;
}
```

PASSO 24: O código anterior pressupõe que o UPDATE ocorrerá sempre com sucesso... **mas isso não é verdade**. Se verificarmos o retorno de **transacao.executeUpdate()**, podemos verificar se houve falha (o resultado seria 0, indicando 0 linhas modificadas no BD). Nesse caso, precisamos adicionar um código para tentar realizar um INSERT no banco de dados, conforme indicado no próximo código.

ClienteDAO.java (método adiciona)

```
/**
 * Adiciona/Atualiza um cliente no banco de dados.
 * @param umCliente Objeto de cliente a ser armazenado.
 * @return True se cliente foi armazenado com sucesso.
 */
public static boolean adiciona(Cliente umCliente) {
    // Pega a conexao atual.
    Connection conexao = Conexao.conecta();
    // Se não existe conexão... retorna false.
```

```
        if (conexao == null) return false;
        // Garante que textos a serem inseridos não contêm aspas
        // para evitar SQL Injection.
        String cpf = umCliente.getCpf();
        cpf = cpf.replaceAll("'","");
        String nome = umCliente.getNome();
        nome = nome.replaceAll("'","");
        // Primeiramente, tenta fazer um update!
        try {
            // Cria a transação
            Statement transacao = conexao.createStatement();
            // Cria a query de update...
            String query  = "UPDATE clientes SET ";
            query += "nome = '" + nome + "', ";
            query += "credito = " + umCliente.getCredito();
            query += " WHERE ";
            query += "cpf LIKE '" + cpf + "'";
            // Excute update retorna número de linhas modificadas;
            // Se retornar 0, é porque não existia o cliente!
            // Então... vamos fazer um insert!
            if ( transacao.executeUpdate(query) == 0 ) {
                // Cria query de insert...
                query  = "INSERT INTO clientes ";
                query += "VALUES ('" + cpf + "', ";
                query += "'" + nome + "', " + umCliente.getCredito();
                query += ")";
                // Se nenhuma linha for modificada com o insert, é
                // porque houve erro... então retorna false
                if ( transacao.executeUpdate(query) == 0 ) {
                    transacao.close();
                    return false;
                }
            }
            // Se chegou aqui, adição/update ocorreu com sucesso.
            transacao.close();
            return true;
        }
        // se houve algum erro nas transações de SQL...
        catch (SQLException ex) {
            // Não faz nada... código apenas para debug
            //System.err.println(ex);
        }
        // Se chegou aqui, não foi possível realizar a atualização.
        return false;
    }
```

PASSO 25: Depois de tanto código sem testar, será interessante ver se tudo isso funcionou, não é? Por bem. Editemos o método **main** da classe **Main** para usar esta tal de ClienteDAO. Para isso, edite a **Main.java** antiga como se segue:

Main.java

```
package siscli;

public class Main {

    public static void main(String[] args) {

        JPrincipal aplicativo = new JPrincipal();

        try {
            Cliente c1 = new Cliente("01234567890", "Um Cliente", 0);
```

```
            System.out.println(c1);
            ClienteDialog janela = new ClienteDialog(aplicativo,c1);
            if ( janela.isEditado() == true) {
                System.out.println("Cliente editado com sucesso!");
            } else {
                System.out.println("Edição cancelada!");
            }
            System.out.println(c1);
            // Armazena cliente
            if ( ClienteDAO.adiciona(c1) == true ) {
               System.out.println("Cliente armazenado!");
            } else {
                System.out.println("Erro no armazenamento!");
            }
        }
        catch (ClienteEx ex) {
            if (ex.getErro() == ClienteEx.Id.ERRO_CPF)
                System.out.println("CPF Inválido");
            if (ex.getErro() == ClienteEx.Id.ERRO_NOME)
                System.out.println("Nome Inválido");
            if (ex.getErro() == ClienteEx.Id.ERRO_CREDITO)
                System.out.println("Credito Inválido");
        }

    }
}
```

Execute e veja o que acontece! Se tudo correu bem, vá até a janela do prompt do MySQL e digite:

```
SELECT * FROM clientes;
```

O cliente editado pela janela do aplicativo deve aparecer no Banco de Dados, agora!

Definitivamente a construção da classe DAO não é uma das tarefas mais simples do mundo; por outro lado, uma vez que a classe DAO está criada, seu uso é extremamente simples, como pode ser visto pelas poucas linhas adicionadas na classe Main.

Adicionalmente, toda a vinculação do programa feito com o MySQL está na classe **Conexao**. Para modificar esse sistema para acessar outros sistemas de banco de dados, basta modificar adequadamente a classe Conexao.

NOTA: uma dúvida frequente sobre o método **adiciona** é a razão pela qual não se usa o REPLACE ao invés do par UPDATE/INSERT. A razão para isso é que o REPLACE não é um comando SQL 100% padronizado e, em cada sistema de banco de dados ele tem uma sintaxe ligeiramente diferente. Sendo assim, o autor opta por usar a dupla UPDATE/INSERT, como uma garantia de que o código execute a contento em qualquer SGBD.

NOTA2: nas queries SQL, as Strings devem ser circundadas por aspas simples ', não as aspas duplas ". A razão para isso é que o SQL Server "não gosta" das aspas duplas e, portanto, usar as aspas simples é uma forma de ampliar a compatibilidade de nossas classes DAO.

4. EXERCÍCIOS (EM TRIO, PARA NOTA)

- Entrega até as 23:59 da véspera da prova AV1.

1) Crie um banco de dados alternativo em que a tabela clientes tenha um coluna adicional, do tipo INT, que sirva para indicar o **sexo** do cliente, compatibilizando-o com a classe Cliente criada no exercício das aulas anteriores.

2) Ajuste a classe ClienteDAO para armazenar a informação de sexo do cliente.

5. BIBLIOGRAFIA

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

Unidade 7: Herança e Polimorfismo

Prof. Daniel Caetano

Objetivo: Introduzir explicitamente os conceitos de herança e seu uso para obter polimorfismo.

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

INTRODUÇÃO

Até o presente momento, nos utilizamos de algumas das vantagens da orientação a objetos sem termos plena consciência disso. Criamos janelas e acessamos ao banco dados, por exemplo, sem que fosse dada a devida atenção ao fato de não precisarmos programar detalhadamente todas essas ações.

A propriedade específica que nos permite aproveitar código de uma maneira rápida e simples é a **herança**, que será melhor examinada adiante. A herança, entretanto, não é usada apenas para reaproveitar código, mas também para permitir um efeito chamado **polimorfismo**, que ficará claro até o final deste capítulo.

1. HERANÇA

No mundo real, chama-se de herança tudo aquilo que é transmitido dos pais para os filhos. Isso significa que, se um pai deixa uma casa de herança para um de seus filhos, esse filho poderá desfrutar dessa casa sem ter que construí-la.

Os projetistas de linguagens orientadas a objetos julgaram que esse era um comportamento interessante de um código, isto é, se uma classe de objetos pudesse **herdar características e comportamentos** de outra classe, já que isso seria uma forma bastante prática de **reaproveitar código**.

Ao analisar as possibilidades, perceberam que esse recurso de **herança** tem inúmeras vantagens, mas que em alguns casos podia tornar a compreensão do sistema confuso; sendo assim, estabeleceram regras para o uso de herança:

a) A classe que vai "dar" seus métodos e atributos para outra é chamada **classe pai** ou **superclasse**.

b) A classe que "herda" (recebe) métodos e atributos de outra é chamada **classe filha** ou **subclasse**.

c) Sempre que uma classe filha **herda** características e comportamentos da classe pai, ela deve herdar **todos os atributos e métodos públicos** da classe pai. Isso significa que a classe filha será capaz de realizar todas as tarefas da classe pai.

d) Uma classe filha pode **adicionar** funcionalidades e modificar o comportamento de funcionalidades já existentes, mas **jamais pode remover** funcionalidades (isto é: um método público que foi herdado não pode ser removido ou tornado privado).

e) Só deve ser considerada a herança entre elementos fisicamente correlacionados, isto é, de uma classe pai **mais genérica** para uma classe filha **mais especializada**.

Como consequência dos itens a) a d), chama-se o processo de herança na codificação de **estensão** ou **especialização**. Como uma consequência de todos estes itens, incluindo o e), apenas classes semanticamente relacionadas devem estar relacionadas por herança.

Por exemplo, uma classe Cliente (classe filha) pode herdar as características de uma classe Pessoa (classe pai), já que existe uma relação clara entre elas: Cliente é um tipo mais específico de pessoa. Por outro lado, Pessoa nunca pode estender Cachorro, já que Pessoa não é um tipo mais específico de Cachorro - ainda que eles tenham algumas partes em comum.

NOTA: Neste segundo caso, para permitir o reaproveitamente de código, haveria a necessidade de criar uma classe mais genérica para acomodar o que ambos - Pessoa e Cachorro - possuem em comum. Por exemplo: se for criada a classe Animal, é possível dizer que Pessoa estende Animal... assim como Cachorro estende Animal.

2. EXEMPLO PRÁTICO DE HERANÇA

Em uma das primeiras aulas, construimos uma classe Pessoa, indicada abaixo:

Pessoa.java

```
/**
 * 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
   private String nome;
   private int idade;

   /**
    * Construtor para objetos da classe Pessoa
    * @param       nome   o nome da pessoa
    * @param       idade  a idade da pessoa, deve ser entre 0 e 100
    * @return      não há valor de retorno
    */
   public Pessoa(String nome, int idade) {
      // Inicializa variáveis de instância
      setNome(nome);
      setIdade(idade);
      }
```

```
    /**
     * Retorna um texto que descreve o objeto
     * @return      String contendo a descrição da pessoa
     */
    public String toString() {
       String tmpString;
       tmpString = "Nome: " + getNome() + "\nIdade: " + getIdade() + "\n";
       return(tmpString);
       }

    /**
     * Retorna o nome da pessoa
     * @return      String contendo o nome
     */
    public String getNome() {
       return (nome);
       }

    /**
     * Retorna a idade da pessoa
     * @return      int contendo a idade
     */
    public int getIdade() {
       return (idade);
       }

    /**
     * Muda o nome da pessoa
     * @param       nome String contendo o nome
     */
    public void setNome(String novoNome) {
       nome = novoNome;
       }

    /**
     * Muda a idade da pessoa
     * @param       int contendo a idade (0 a 100)
     */
    public void setIdade(int novaIdade) {
       if (novaIdade < 0) novaIdade = 0;
       else if (novaIdade > 100) novaIdade = 100;
       idade = novaIdade;
       }

    }
```

Se quiséssemos, agora, construir uma classe Cliente que tenha o atributo crédito, mas reaproveitando o código da classe Pessoa acima... como poderíamos fazer isso?

PASSO 1: Primeiramente, criaremos uma classe nova chamada Cliente e a alteraremos com a diretiva **extends**, para indicar que essa nova classe **estende Pessoa**:

Cliente.java

```
/**
 * A classe Cliente define características importantes de um cliente.
 *
 * @author (seu nome)
 * @version 20110406_001  <<== Versão é muito importante!
 */

public class Cliente extends Pessoa {

   }
```

PASSO 2: Agora precisamos definir os novos atributos - no caso, apenas o crédito:

Cliente.java

```
/**
 * A classe Cliente define características importantes de um cliente.
 *
 * @author (seu nome)
 * @version 20110406_001  <<== Versão é muito importante!
 */

public class Cliente extends Pessoa {
   // Atributos de Instância
   private int credito;
   }
```

PASSO 3: Criemos agora os *getters* e *setters* para os atributos:

Cliente.java

```
/**
 * A classe Cliente define características importantes de um cliente.
 *
 * @author (seu nome)
 * @version 20110406_001  <<== Versão é muito importante!
 */

public class Cliente extends Pessoa {
   // Atributos de Instância
   private int credito;

   /**
    * Retorna o crédito do cliente
    * @return      int contendo o crédito em centavos
    */
   public int getCredito() {
      return (credito);
      }

   /**
    * Muda o crédito da pessoa
    * @param       int contendo o crédito (em centavos)
    */
   public void setCredito(int novoCredito) {
      if (novoCredito < 0) return;
      credito = novoCredito;
      }
   }
```

PASSO 4: Agora precisamos criar um construtor. Como Cliente é uma extensão de Pessoa, ele precisa receber todos os parâmetros de uma pessoa qualquer, como nome e idade, mas também precisa receber um novo parâmetros: o crédito.

Entretanto, o código que inicializa os atributos **nome** e **idade** já está pronto no **construtor da classe Pessoa**. Sendo assim, vamos usá-lo para inicializar estes valores e, só depois, vamos inicializar o valor do atributo **credito**. Observe como isso é feito, a seguir.

Cliente.java

```
/**
 * A classe Cliente define características importantes de um cliente.
 *
 * @author (seu nome)
 * @version 20110406_001  <<== Versão é muito importante!
 */

public class Cliente extends Pessoa {
   // Atributos de Instância
   private int credito;

   /**
    * Retorna o crédito do cliente
    * @return      int contendo o crédito em centavos
    */
   public int getCredito() {
      return (credito);
      }

   /**
    * Muda o crédito da pessoa
    * @param       int contendo o crédito (em centavos)
    */
   public void setCredito(int novoCredito) {
      if (novoCredito < 0) return;
      credito = novoCredito;
      }

   /**
    * Construtor para objetos da classe Cliente
    * @param       nome          o nome da pessoa
    * @param       idade         a idade da pessoa, deve ser entre 0 e 100
    * @param       credito       o crédito do cliente (em centavos)
    * @return      não há valor de retorno
    */
   public Cliente(String nome, int idade, int credito) {
      // Inicializa atributos da superclasse (nome e idade)
      super(nome, idade);
      // Inicializa atributos exclusivos da classe cliente
      setCredito(credito);
      }
   }
```

Observe o uso da palavra **super** para executar o construtor da **super**classe. Essa chamada ao método **super()** deve ser **sempre a primeira coisa** feita no construtor de uma **subclasse**.

PASSO 5: Modifique agora o método main a classe Main e observe o resultado:

Main.java

```
// Pacote
package nome_do_pacote;

// Classe Principal do Aplicativo
public class Main {
    // Método de Início da Aplicação
    public static void main(String[] args) {
        Cliente c;
        c = new Cliente("Fulano", 18, 10000);
        System.out.println(c);
    }
}
```

3. HERANÇA COM ELEMENTOS PRONTOS

Uma coisa muito prática é estender elementos que já existem no Java, modificando seu comportamento. Por exemplo, não seria interessante ter um campo de texto (JTextField) específico para CPF? Poderíamos dar o nome de JCPFField para ele, por exemplo. Mas como podemos fazer isso?

Como essa é uma tarefa comum, a Sun já criou duas classes importantes: **JFormattedTextField** e **MaskFormatter**, que serão usadas em conjunto aqui.

JFormattedTextField é simplesmente uma versão "genérica" do JTextField que permite que definamos uma formatação através de um objeto do tipo **MaskFormatter**. Vamos ver como usar estes dois componentes para criarmos um JCPFField, ou seja, um campo formatado para CPF.

PASSO 1. No NetBeans, crie um projeto chamado **ElementosUI** . Isso deve criar automaticamente o pacote **elementosui** e a classe **Main** .

PASSO 2. No pacote **elementosui**, crie agora uma classe java simples e dê a ela o nome de **JCPFField** . O código abaixo será criado:

JCPFField.java

```
package elementosui;

/**
 *
 * @author djcaetano
 */
public class JCPFField {

}
```

PASSO 3. Vamos modificar essa classe para que ela seja uma extensão de JFormattedTextField, com um construtor que irá apenas chamar o contrutor da superclasse:

JCPFField.java

```
package elementosui;
import javax.swing.*;

/**
 *
 * @author djcaetano
 */
public class JCPFField extends JFormattedTextField {

    public JCPFField() {
        super(); // Inicializa a superclasse
    }
}
```

PASSO 4. Agora chegou a hora de criar o objeto do tipo MaskFormatter, que é responsável por definir as características da formatação.

JCPFField.java

```
package elementosui;
import javax.swing.*;
import java.text.*;
import javax.swing.text.*;

/**
 *
 * @author djcaetano
 */
public class JCPFField extends JFormattedTextField {

    public JCPFField() {
        super(); // Inicializa a superclasse
        // Cria a máscara formatadora
        MaskFormatter formato;
        formato = new MaskFormatter("###.###.###-##");
    }
}
```

PASSO 5. Bem, o NetBeans vai indicar um erro na linha de criação do MaskFormatter. Isso ocorre porque pode ocorrer um erro de "formato inválido" na criação da máscara de formatação, e o Java lhe obriga a lidar com esse possível erro (ainda que ele nunca vá ocorrer, se você especificar a máscara corretamente).

Assim, será preciso colocar as operações com o objeto MaskFormatter, incluindo sua criação, dentro de um bloco **try**. Como nunca irá ocorrer o erro em questão (porque estamos definindo correta e estaticamente a máscara), iremos colocar o **catch** apenas para "agradar" o Java, e não iremos inserir nenhum código dentro.

JCPFField.java

```
package elementosui;
import javax.swing.*;
import java.text.*;
import javax.swing.text.*;

/**
 *
 * @author djcaetano
 */
public class JCPFField extends JFormattedTextField {

    public JCPFField() {
        super(); // Inicializa a superclasse
        MaskFormatter formato;
        try {
           // Cria a máscara formatadora
           formato = new MaskFormatter("###.###.###-##");
        } catch (ParseException ex) { }
    }
}
```

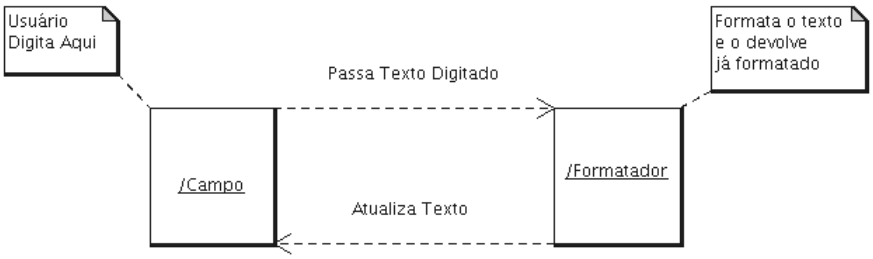
NOTA: os dígitos da máscara podem ser especificados assim:

Dígito	Descrição
#	Qualquer número válido (usa Character.isDigit).
'	Caractere de escape, para indicar formatação especial.
U	Qualquer letra (usa Character.isLetter), apenas em maiúsculas.
L	Qualquer letra (usa Character.isLetter), apenas em minúsculas.
A	Qualquer letra ou número (Character.isLetter ou Character.isDigit)
?	Qualquer letra (Character.isLetter).
*	Qualquer símbolo (incluindo letras e números).
H	Qualquer caractere em hexadecimal (0-9, a-f ou A-F).

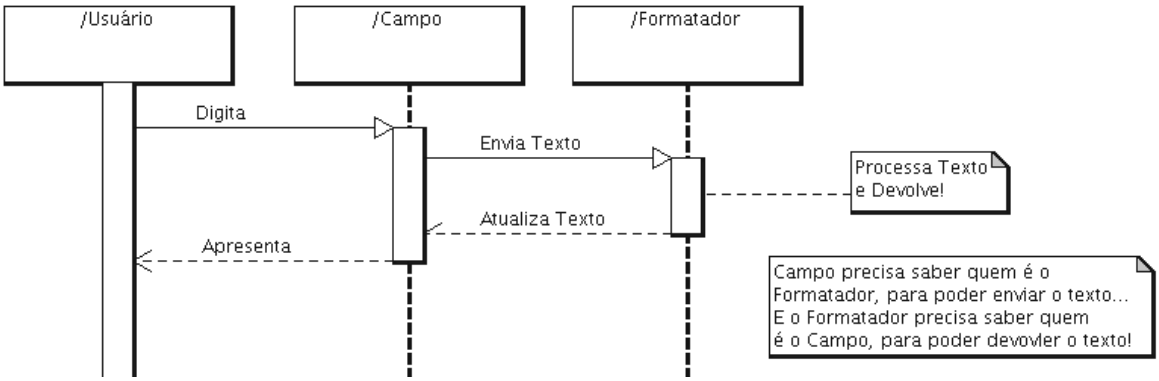
NOTA2: Quando se desejar limitar os dígitos, por exemplo, permitindo que números sejam apenas 0 e 1, deve-se usar setValidCharacters. No exemplo em curso, seria algo assim:

```
formato.setValidCharacters("01");
```

Infelizmente nosso trabalho ainda não está finalizado. Observe no diagrama a seguir como a máscara do tipo MaskFormatter e o campo JFormattedTextField **precisam trabalhar em conjunto**.



A sequência de ações ocorre como indicado abaixo:



Observe que o Campo precisa saber quem é seu Formatador e o Formatador precisa saber quem é o Campo que está sendo formatado.

PASSO 6. Como o MaskFormatter e o campo JFormattedTextField **precisam trabalhar em conjunto**, nós precisamos avisar ao JFormattedTextField (superclasse) qual é o objeto MaskFormatter que ele vai usar, través do método setFormatter:

JCPFField.java

```
package elementosui;
import javax.swing.*;
import java.text.*;
import javax.swing.text.*;

/**
 *
 * @author djcaetano
 */
public class JCPFField extends JFormattedTextField {

    public JCPFField() {
        super(); // Inicializa a superclasse
        MaskFormatter formato;

        try {
           // Cria a máscara formatadora
           formato = new MaskFormatter("###.###.###-##");
           // Instala formatador no JFormattedTextField
           setFormatter(formato);
        } catch (ParseException ex) { }
    }
}
```

PASSO 7. Da mesma forma que o JFormattedTextField precisa saber do formatador, o formatador **também** precisa saber do JFormattedTextField. Isso pode ser avisado ao formatador através do médodo **install**, que define para o formatador **em que objeto ele está sendo instalado**.

JCPFField.java

```
package elementosui;
import javax.swing.*;
import java.text.*;
import javax.swing.text.*;

/**
 *
 * @author djcaetano
 */
public class JCPFField extends JFormattedTextField {

    public JCPFField() {
        super(); // Inicializa a superclasse
        MaskFormatter formato;

        try {
           // Cria a máscara formatadora
           formato = new MaskFormatter("###.###.###-##");
           // Instala formatador no JFormattedTextField
           setFormatter(formato);
            // Instala o JFormattedTextField no MaskFormatter
           formato.install(this);
        } catch (ParseException ex) { }
    }
}
```

PASSO 8. Finalizada a nossa classe, **compile e execute o código**, ainda que nada aconteça. Depois disso, no mesmo pacote elementosui, crie um Formulário JDialog com o nome de **JExemplo**.

PASSO 9. Na área do projeto da janela JExemplo, clique com o botão esquerdo do mouse no nome da classe **JCPFField.java** e a arraste para cima do formulário JDialog que você acabou de criar. Selecione o arquivo **JExemplo.java** na área de projeto e pressione SHIFT+F6, para executar a janela. Observe como o campo se comporta.

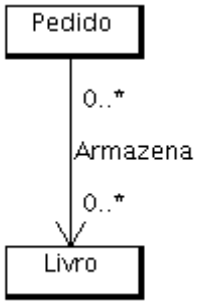
3.1. Adicionando o Elemento à Palheta

É possível adicionar o seu novo elemento visual JCPFField à Paleta. Para isso, selecione o modo de Projeto de uma das janelas e clique com o botão direito em alguma das categorias da Paleta (por exemplo, **Contêineres Swing**) e selecione **Criar Nova Categoria**. Dê o nome à essa categoria (por exemplo: **Estacio**).

Agora, clique com o botão direito sobre o nome da classe JCPFField, selecione a opção **Ferramentas > Adicionar à paleta...** e, em seguida, selecione a palheta desejada (por exemplo, a paleta **Estacio**)! A partir de agora, sempre que desejar um campo CPF, basta arrastá-lo da paleta!

4. ENTENDENDO O POLIMORFISMO

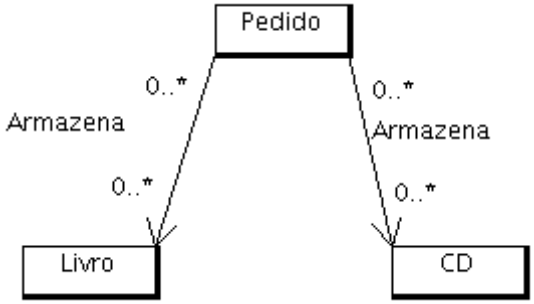
A grande maioria dos sistemas comerciais desenvolvidos envolvem uma classe chamada Pedido. Objetos da classe Pedido são, usualmente, considerados "objetos depósito", isto é, são objetos que armazenam outros objetos. Em uma loja de livros, por exemplo, o objeto da classe Pedido armazenaria vários objetos da classe Livro, indicando os livros que um cliente comprou ou está comprando. Seguindo esta lógica, poderíamos criar um diagrama indicado como se segue:



Como o pedido armazena objetos do tipo livro, é de se esperar que a classe Pedido contenha um método como o seguinte:

```
void adiciona(Livro umLivro)
```

Ou seja, sempre que se desejar, é possível adicionar um novo objeto do tipo livro ao objeto do tipo pedido. Ora, mas e se agora a loja passar a vender também CDs? A primeira solução seria adaptar o software antigo, cujo diagrama ficaria assim:



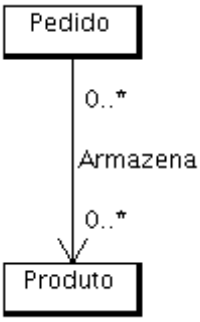
Ocorre que isso implica em acrescentar um método novo a Pedido:

```
void adiciona(CD umCD)
```

Isso é absolutamente indesejável! Será que não existe uma solução que nos permita criar quantos tipos de produto quisermos, **SEM** modificar as classes para que suportem os novos produtos?

A resposta é **SIM** e a solução está exatamente na palavra **produto**. Tanto CDs quanto Livros são **produtos**, o que sugere o uso de polimorfismo através da herança. Hã? Como assim?

Simples: criamos um modelo em que Pedido guarda objetos do tipo Produto!



E pedido precisa ter apenas um método de adição:

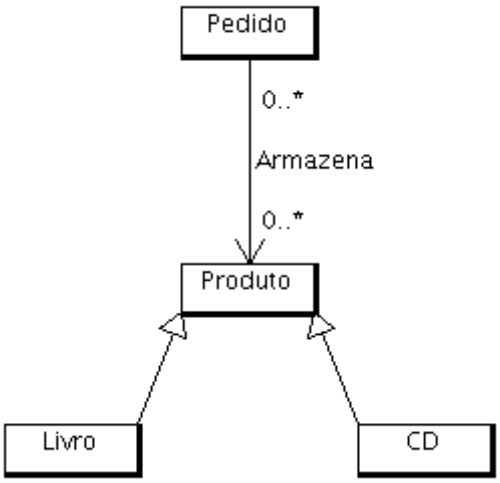
```
void adiciona(Produto umProduto)
```

Mas... então "Produto" vai ter que possuir todos os atributos de qualquer tipo de produto que exista? Por exemplo: vou ter que colocar um atributo "faixas" e outro "paginas" mesmo em produtos que não possuam essas características?

A resposta é **NÃO**. Na classe Produto serão colocados apenas atributos e métodos que sejam genéricos de qualquer produto, como *codigo*, *nome*, *descricao*, *preco*, dentre outros. Os outros atributos, que sejam específicos de Livro, CD, DVD e outras categorias que venham a ser criadas, ficarão em suas respectivas classes!

Isso mesmo: ainda teremos as classes Livro e CD, por exemplo; o segredo é que as programaremos como extensões da classe Produto!

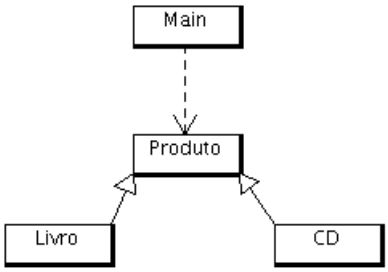
Observe o diagrama abaixo:



Agora, o método **adiciona(Produto umProduto)** aceitará normalmente objetos das classes Livro e CD pois, afinal de contas, objetos das classes Livro e CD **são** produtos!

4.1. Programando com Polimorfismo

Para verificar o comportamento descrito anteriormente, iremos programar um código simples, com um diagrama ligeiramente modificado:



Vamos implementar esse diagrama. Feche todos os projetos no NetBeans e crie um projeto de aplicativo Java simples, dando-lhe o nome **TestePolimorfismo**. A classe Main será criada automaticamente, incluindo um método main. Vamos deixá-los de lado por enquanto.

Por quê vamos deixá-la de lado? Observe as "setas" no diagrama. Na UML, as setas indicam a dependência, isto é, se uma seta liga a classe Main à classe Produto, isso significa que a classe Main depende da classe Produto.

De maneira geral, vamos sempre iniciar a programação pelas classes que são *independentes* de outras classes, isto é, vamos começar por classes que não tenham nenhum seta *saindo* delas. Nesse diagrama, a única classe que não tem setas saindo é a classe Produto e, sendo assim, começaremos por ela.

Quando a classe Produto estiver pronta, poderemos criar as classes que dependam apenas da classe Produto... e assim por diante.

Clique com o botão direito no pacote chamado **testepolimorfismo** (ATENÇÃO: é o que tem o pacote à esquerda, **não** o que tem a xícara!) e selecione Novo > Classe Java. Dê o nome de **Produto** para a classe. Clique duas vezes no arquivo Produto.java recém criado e edite-o para que ele tenha a seguinte aparência:

Produto.java

```
// Pacote
package testepolimorfismo;

// Como Construir Produtos
public class Produto {

    // Todo produto tem uma identificação
    private int id;
    // Todo produto tem um nome
    private String nome;
    // Todo produto tem uma descrição
    private String descricao;

    // Construtor de objetos (preenche os atributos!)
    public Produto(int umaId, String umNome, String umaDescricao) {
        id = umaId;
        nome = umNome;
        descricao = umaDescricao;
    }
}
```

ATENÇÃO! As linhas em cinza **já estão presentes**, foram criadas pelo NetBeans. Neste exemplo os comentários do NetBeans foram apagados e foram acrescentados os comentários que, no exemplo acima, estão indicados em azul.

Executando este código, nada acontece! Por quê?

Porque o Java está começando a execução pelo método **main** da classe **Main**, lembra?

Então, vamos até lá, modificar o código para que ele crie um produto e produza algum resultado! Clique duas vezes no arquivo Main.java para poder editá-lo, e modifique-o para que fique como o descrito abaixo:

Main.java

```
// Pacote
package testepolimorfismo;

// Classe Principal do Aplicativo
public class Main {
    // Método de Início da Aplicação
    public static void main(String[] args) {
        Produto p1;
        p1 = new Produto(0,"O Produto","Um produto qualquer.");
        System.out.println(p1);
    }
}
```

Execute e veja o que contece!

Provavelmente deve ter aparecido algo como:

```
testepolimorfismo.Produto@098a3b
```

Isso ocorre porque mandamos imprimir um objeto sem explicar ao Java como ele deve ser impresso! Para explicar ao Java como imprimir um objeto nosso, devemos criar um método com a seguinte assinatura:

String toString()

E, neste método, devemos retornar uma String (com a instrução **return**) que descreva o objeto atual.

De onde vem aquele texto impresso pelo Java por padrão? Sempre que criarmos uma classe sem informar a classe origem, isto é, sem informar uma classe a ser extendida, o Java assume que estamos estendendo a classe fundamental, chamada **Object**.

Dentre outros métodos, a classe Object implementa uma versão simplificada de método toString que imprime aquele texto "codificado" que vimos acima. Como não declaramos uma classe origem, a nossa classe está estendendo a classe Object e, portanto, esta **herdando** o método toString definido na classe Object.

O código abaixo apresenta a mudança com relação ao código anterior, acrescentando um método toString;

Produto.java

```
// Pacote
package testepolimorfismo;

// Como Construir Produtos
public class Produto {

    // Todo produto tem uma identificação
    private int id;
    // Todo produto tem um nome
    private String nome;
    // Todo produto tem uma descrição
    private String descricao;

    // Construtor de objetos (preenche os atributos!)
    public Produto(int umaId, String umNome, String umaDescricao) {
        id = umaId;
        nome = umNome;
        descricao = umaDescricao;
    }

    public String toString() {
        return  id + ": " + nome + " - " + descricao;
    }
}
```

Execute novamente e veja o resultado!

Agora, vamos criar a classe Livro, que extende a classe Produto! Clique com o botão direito no pacote chamado **testepolimorfismo** e selecione Novo > Classe Java. Dê o nome de **Livro** para a classe. Clique duas vezes no arquivo Livro.java recém criado e edite-o para que ele tenha a seguinte aparência:

Livro.java

```
// Pacote
package testepolimorfismo;

// Como Construir Livros
public class Livro extends Produto {

    // Construtor de objetos (preenche os atributos!)
    public Livro(int umaId, String umNome, String umaDescricao) {
        super(umaId, umNome, umaDescricao);
    }
}
```

Observe duas coisas importantes: a) a expressão "extends Produto" na linha de declaração da classe e, no método construtor, a primeira instrução é uma tal de "super". Sempre que estendemos uma classe - no caso, estamos estendendo a classe Produto -, a primeira instrução do construtor da nova classe - neste caso, o método Livro -, deve ser uma chamada ao construtor da classe origem (neste caso, Produto).

Como o construtor da classe produto exige três parâmetros para construir um objeto produto, a chamada **super** precisa também ocorrer com estes três parâmetros... que obviamente precisam ser recebidos pelo construtor da classe Livro.

Neste ponto, a classe Livro está se comportando exatamente como um Produto, já que nenhuma nova característica foi acrescentada. Vamos, então, acrescentar atributos de números de páginas e número de capítulos na classe Livro, modificando-a como segue:

Livro.java

```
// Pacote
package testepolimorfismo;

// Como Construir Livros
public class Livro extends Produto {

    // Além do que todo produto tem, um livro também tem...
    // Todo livro tem um número de páginas
    private int paginas;
    // Todo livro tem capítulos
    private int capitulos;

    // Construtor de objetos (preenche os atributos!)
    public Livro(int umaId, String umNome, String umaDescricao, int noPaginas, int
noCapitulos) {
        super(umaId, umNome, umaDescricao);
        paginas = noPaginas;
        capitulos = noCapitulos;
    }
}
```

Pronto, agora o Livro já tem características próprias. Vamos alterar o código da classe Main para que um objeto do tipo Livro seja criado:

Main.java

```
// Pacote
package testepolimorfismo;

// Classe Principal do Aplicativo
public class Main {
    // Método de Início da Aplicação
    public static void main(String[] args) {
        Produto p1;
        p1 = new Produto(0,"O Produto","Um produto qualquer.");
        System.out.println(p1);

        Produto p2;
        p2 = new Livro(1,"Duna","Um livro de Frank Herbert.", 678, 3);
        System.out.println(p2);
    }
}
```

Execute e veja... o que aconteceu? Deve ter funcionado, MAS o número de páginas do livro e o número de capítulos não foi mostrado pela impressão, não é? Isso ocorre porque a classe Livro esta **herdando** o método toString da classe Produto, então só as informações básicas estão sendo mostradas.

Vamos modificar a classe livro, com um novo método toString. Entretanto, que tal se pudermos reaproveitar todo o trabalho já feito pelo toString da classe Produto? Afinal de contas, queremos a descrição completa, só queremos acrescentar alguns detalhes.

Obviamente não podemos definir um novo toString como indicado a seguir:

```
    public String toString() {
        return toString() + " " + paginas + " páginas em " + capitulos + " capítulos.";
    }
```

Isso faria com que o Java travasse (experimente!), pois temos um ciclo infinito: um método que chama a si mesmo! Como podemos fazer para explicar ao Java que queremos chamar a versão "antiga" do toString, para podermos "grudar" alguma coisa no resultado dela?

Bem, lembra-se da palavrinha "super"? Ela serve também para que possamos nos referir à classe original, também chamada de **super**classe. Pois bem, o código que faz o "truque" está indicado na listagem a seguir:

Livro.java

```
// Pacote
package testepolimorfismo;

// Como Construir Livros
public class Livro extends Produto {

    // Além do que todo produto tem, um livro também tem...
    // Todo livro tem um número de páginas
    private int paginas;
    // Todo livro tem capítulos
    private int capitulos;

    // Construtor de objetos (preenche os atributos!)
    public Livro(int umaId, String umNome, String umaDescricao, int noPaginas, int
noCapitulos) {
        super(umaId, umNome, umaDescricao);
        paginas = noPaginas;
        capitulos = noCapitulos;
    }

    public String toString() {
        return super.toString() + " " + paginas + " páginas em " + capitulos + "
capítulos.";
    }
}
```

Observe que agora, ao executar, é impresso tanto o número de páginas quanto o número de capítulos do livro! Agora, observe bem o código da classe Main atual:

Main.java

```
// Pacote
package testepolimorfismo;

// Classe Principal do Aplicativo
public class Main {
    // Método de Início da Aplicação
    public static void main(String[] args) {
        Produto p1;
        p1 = new Produto(0,"O Produto","Um produto qualquer.");
        System.out.println(p1);

        Produto p2;
        p2 = new Livro(1,"Duna","Um livro de Frank Herbert.", 678, 3);
        System.out.println(p2);
    }
}
```

Estamos guardando um objeto Livro em uma variável do tipo Produto! E mais, estamos imprimindo a variável do tipo Produto, mas como o objeto que está na variável é um Livro, o texto é impresso segundo o toString da classe Livro!

> Apesar de isso parecer mais confundir do que ajudar, esse é exatamente o comportamento esperado e, com o tempo, isso se torna natural. Dominar esse conceito é fundamental para que um programador passe a programar de maneira eficiente usando orientação a objetos!

Para finalizar e sedimentar os conhecimentos apreendidos, vamos agora criar uma classe chamada CD, para representar produtos do tipo CD. Clique com o botão direito no pacote chamado **testepolimorfismo** e selecione Novo > Classe Java. Dê o nome de **CD** para a classe. Clique duas vezes no arquivo CD.java recém criado e edite-o para que ele tenha a seguinte aparência:

CD.java

```
// Pacote
package testepolimorfismo;

// Como Construir CDs
public class CD extends Produto {

    // Além do que todo produto tem, um CD também tem...
    // Todo CD tem um número de faixas!
    private int faixas;

    // Construtor de objetos (preenche os atributos!)
    public Cd(int umaId, String umNome, String umaDescricao, int noFaixas) {
        super(umaId, umNome, umaDescricao);
        faixas = noFaixas;
    }

    public String toString() {
        return super.toString() + " " + "CD com " + faixas + " faixas.";
    }
}
```

E, finalmente, modifique a classe Main conforme o código abaixo:

Main.java

```
// Pacote
package testepolimorfismo;

// Classe Principal do Aplicativo
public class Main {
    // Método de Início da Aplicação
    public static void main(String[] args) {
        Produto p1;
        p1 = new Produto(0,"O Produto","Um produto qualquer.");
        System.out.println(p1);

        Produto p2;
        p2 = new Livro(1,"Duna","Um livro de Frank Herbert.", 678, 3);
        System.out.println(p2);

        Produto p3;
        p3 = new CD(2,"Death Magnetic","CD do Metallica de 2008.", 10);
        System.out.println(p3);
    }
}
```

Execute e veja os resultados!

```
        p1 = new Cd(1,"Death Magnetic","□ülbum do Metallica, lan□°ado em 2008.", 10);
        p2 = new Livro(2,"Duna","Livro de Fic□°□úo Cient□¡fica de Frank Herbert.", 678,
3);

        System.out.println(p1);
        System.out.println(p2);
    }
}
```

4.1. Exercite!

A) Pegue o código digitado e modifique a classe Produto para que o código do produto apareça entre parênteses, após o nome do produto. A alteração funcionou para os Livros e CDs? Por quê?

B) Tente modificar a classe CD para que ela também armazene a informação de tempo total de música, em minutos. Em que métodos você precisou mexer? Foi preciso alterar outras classes?

C) Crie uma classe DVD que especialize a classe Produto, acrescentando o atributo relativo ao número de DVDs do filme, bem como os *getters*, *setters* e a respectiva alteração no método toString.

5. 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 8: Interfaces e Polimorfismo

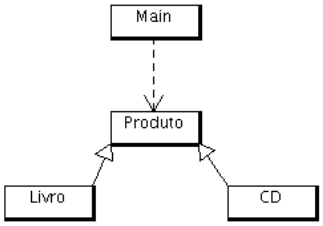
Prof. Daniel Caetano

Objetivo: Revisar e Aplicar os Conceitos de Polimorfismo com uso de Interfaces.

Bibliografia: DEITEL, 2005; HOFF, 1996

INTRODUÇÃO

Na aula anterior vimos como o uso de herança pode simplificar o desenvolvimento, seja pela facilidade de expansão de uma classe, seja pela capacidade de substituir uma classe por outra que seja sua descendente.



A estensão de classes, entretanto, nem sempre é adequada; por esta razão, veremos uma maneira alternativa para implementar o polimorfismo.

1. QUANDO ESTENDER UMA CLASSE NÃO SERVE

- Quando uma classe não é um subtipo da outra

Quando fazemos uma extensão de uma classe, isto é, usamos o conceito de herança, vimos que sempre temos uma relação direta entre a idéia das duas classes, isto é, se a classe B extende a classe A, dizemos que B é uma versão mais específica de A.

Assim, quando falamos na classe Produto e posteriormente que a classe Livro extende a classe produto, estamos dizendo que qualquer objeto do tipo Livro também é um objeto do tipo Produto e, assim, podemos fazer uso do polimorfismo e, de quebra, isso nos permitia aproveitar muito código da classe Produto.

Bem, algumas vezes temos duas classes diferentes que compartilham um grande número de atributos ou métodos mas, **em termos de idéia, uma não tem nada a ver com outra**. Por exemplo:

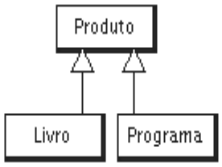
Classe: Retangulo	Classe: Piscina
- Largura	- Largura
- Comprimento	- Comprimento
- Preencher()	- Profundidade
	- Preencher()

Apesar de estas classes compartilharem atributos e métodos - o que ocorre porque, neste caso, a piscina tem a forma plana retangular (um paralelepípedo na verdade), **NÃO** dá para imaginar alguém dizendo que a **Piscina É um Retangulo**.

Assim, dizer que Piscina "extends" Retangulo é inadequado, filosoficamente. Mas a razão para isso não é apenas "filosófica". É prática também: um sistema construído por Classes que se extendem **apenas** para aproveitar código, sem que isso faça um sentido lógico, destrói uma boa parte das vantagens do uso de orientação a objetos, como a facilidade de compreensão de um projeto pronto ou a manutenção do mesmo.

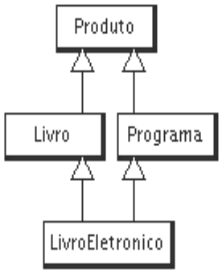
- Quando uma classe é um subtipo de outras duas ou mais classes

Uma outra razão que leva a não ser sempre adequado o uso de extensão é quando precisamos herdar características de duas classes diferentes, simultaneamente. Por exemplo, originalmente temos as seguintes classes:



Nossa loja vende Produtos dos tipos Livro e Programa. Cada um tem características bem específicas. Livro não é um Programa e Programa não é um Livro, mas ambos são um Produto.

Quando a loja passa a querer vender Livros Eletrônicos, o projetista/programador conclui que as características deste novo produto levam à idéia de que LivroEletronico deve ser uma classe que extenda, simultaneamente, um Livro **E** um Programa, como indicado a seguir.



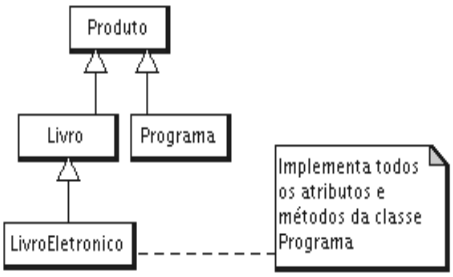
Porém, o programador vai ler a documentação e descobre que o Java não permite que uma classe herde características de várias outras, o que é chamado de **herança múltipla**. As

razões detalhadas pelas quais o java não permite isso estão além do escopo deste curso, mas, simplificadamente, a razão para esta limitação é que o uso de herança múltipla, em geral, traz mais dores de cabeça do que resolve problemas.

O programador pensa, então, em duas situações disintas:

A) Implementar LivroEletronico como extensão de Livro
B) Implementar LivroEletronico como extensão de Programa

Nenhuma destas duas soluções é boa; vamos analisar o primeiro caso, que o programador extende a classe Livro e implementa, na classe LivroEletronico, todos os atributos e métodos da classe Programa:



Apesar de o programador ter inserido no LivroEletronico **todas** as características (atributos e métodos) de um Programa (sem reaproveitamento de código), o Java **não tem como saber disso**. Como consequência, as partes do software que souberem manipular um objeto da classe Programa **não poderão usar polimorfismo** com objetos da classe LivroEletronico.

Como na situação B o problema é análogo e, em nenhum dos casos é possível reaproveitar diretamente totalmente o código, chegamos a um beco sem saída? Na verdade, não.

2. AS SALVADORAS INTERFACES

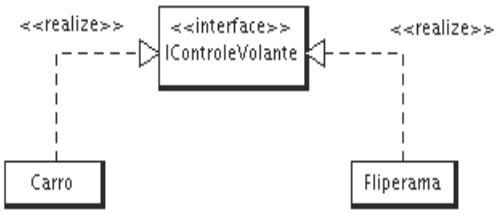
Bem, é claro que os desenvolvedores da Sun (agora Oracle) não são obtusos e, assim, eles criaram um jeito de avisar ao Java quando um objeto de uma classe é capaz de fazer as mesmas coisas que um objeto de outra classe, permitindo o polimorfismo mesmo quando uma classe não é extensão de outra. O nome mais natural que descobriram para esse recurso foi o nome **interface**.

Ora, esse nome faz total sentido, se pensarmos que, no mundo real, só precisamos que conhecer a interface de um objeto, para que saibamos operá-lo. Precisamos saber **qual é o tipo de interface** que ele tem.

Mas, qual é o intuito das interfaces? Em poucas palavras, o objetivo das interface é permitir o polimorfismo sem o uso de herança, exatamente para os casos onde a herança não se justifica ou para os casos em que seria necessária a herança múltipla.

- Quando a herança não faz sentido

O exemplo a seguir permite visualizar o uso de interfaces em situações em que a herança não faz o menor sentido.

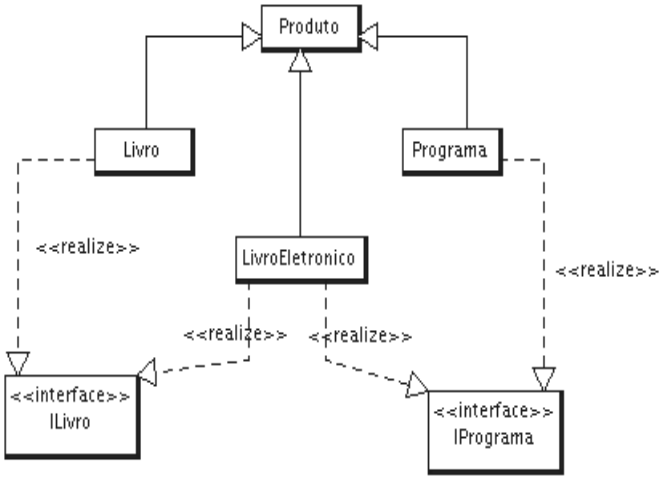


Todos já viram fliperamas de corrida que simulam a cabine de um carro. Ora, embora Carro e Fliperama sejam classes completamente distintas, isto é, não faria o menor sentido dizer "Carro é um Fliperama" ou vice-versa, é possível dizer que ambos **implementam** (*realize*) uma mesma interface, à qual foi dado o nome de **IControleVolante**.

Assim, se em algum ponto do meu sistema eu tiver algum código que saiba usar a interface IControleVolante, ele saberá usar, ao mesmo tempo, qualquer classe que implemente a interface IControleVolante; no exemplo, são as classes Carro e Fliperama. E, observe, não foi necessário o uso de herança!

- Quando a herança faz sentido, mas é necessária uma herança múltipla

O exemplo a seguir permite que tiremos todo o proveido necessário de polimorfismo, sem a necessidade do uso de herança múltipla:



Agora, para que se possa usar polimorfismo genérico, usa-se a classe Produto; para usar um polimorfismo entre classes que possuem a Interface ILivro (como Livro e LivroEletronico) usa-se a **Interface ILivro**. O mesmo vale para a Interface IPrograma.

Observe duas coisas importantes: está explícito que LivroEletronico implementa (*realize*) tanto a interface ILivro quanto a interface IPrograma, o que leva à segunda observação: **uma classe pode implementar várias interfaces**.

Mas como implementar uma interface?

3. AS INTERFACES NO CÓDIGO

Uma interface é declarada diretamente da seguinte forma:

[escopo] interface <nome da interface> [extends <outra interface>] {
    // Métodos da interface
    }

Exemplo:

IVolante.java

```
public interface IVolante {
      public void virarDireita();
      public void virarEsquerda();
      }
```

Essa declaração diz **O QUE** um objeto que implemente esta interface deve fazer; observe, entretanto, que interfaces **NÃO DEFINEM ATRIBUTOS** e também **NÃO IMPLEMENTAM CÓDIGO DOS MÉTODOS**. As interfaces servem apenas para que o Java saiba que "toda classe que implementar aquela interface, tem essa lista de métodos implementados".

Quando quisermos implementar uma interface, devemos fazê-lo na declaração da classe, de maneira muito parecida com a herança:

[escopo] class <nome da classe> [extends <outra classe>] [implements <interface>] {
    // definições internas da classe
    // tudo que estiver aqui *faz parte* da classe
    }

Observe atentamente os códigos a seguir, dados como exemplo.

Carro.java

```
public class Carro implements IVolante {
      // Atributos do carro
      private String modelo;

      // Métodos do carro
      public String getModelo() { return modelo; }
      public void setModelo(String desc) { modelo = desc; }

      // Métodos da IControleVolante
      public void virarEsquerda() { System.out.println("Esquerda!"); }
      public void virarDireita() { System.out.println("Direita!"); }
      }
```

Outro exemplo:

Fliperama.java

```
public class Fliperama implements IVolante {
      // Atributos do fliperama
      private String nome;

      // Métodos do fliperama
      public String getNomeDoJogo() { return nome; }
      public void setNomeDoJogo(String desc) { nome = desc; }

      // Métodos da IControleVolante
      public void virarEsquerda() { System.out.println("Esquerda!"); }
      public void virarDireita() { System.out.println("Direita!"); }
      }
```

E como isso é usado pelo programa? Simples!

3.1. Usando as Interfaces

Suponhamos que temos a nossa classe pessoa, e que ela tenha um método chamado "dirigir", definido da seguinte forma:

Pessoa.java

```
public class Pessoa {
      public void dirigir(Carro umCarro) {
            umCarro.virarDireita();
            umCarro.virarEsquerda();
            }
      }
```

Com essa implementação, um objeto pessoa só pode dirigir um objeto do tipo Carro. Observe o código principal a seguir.

Main.java

```
public class Main {
      public static void main(String[] args) {
            Carro meuCarro;
            Pessoa eu;
            eu = new Pessoa();
            meuCarro = new Carro();

            eu.dirigir(meuCarro);
            }
      }
```

Essse código deve funcionar perfeitamente e imprimir as mensagens "Esquerda" e "Direita". Entretanto, o proximo código, onde indicamos para dirigir um Fliperama, não irá funcionar!

Main.java

```
public class Main {
      public static void main(String[] args) {
            Carro meuCarro;
            Fliperama meuFliperama;
            Pessoa eu;
            eu = new Pessoa();
            meuCarro = new Carro();
            meuFliperama = new Fliperama();

            eu.dirigir(meuCarro);
            eu.dirigir(meuFliperama);
            }
      }
```

Observe que um erro é indicado na última linha deste código do método main da classe Main. Isso ocorre porque o método Pessoa.dirigir está declarado assim:

Pessoa.java

```
public class Pessoa {
      public void dirigir(Carro umCarro) {
            umCarro.virarDireita();
            umCarro.virarEsquerda();
            }
      }
```

Ou seja: ele só sabe usar objetos do tipo Carro, ainda que só sejam usados métodos da IControleVolante (virarDireita e virarEsquerda). Para tornar nosso código mais genérico, podemos mudar a classe Pessoa da seguinte forma:

Pessoa.java

```
public class Pessoa {
      public void dirigir(IVolante equipamento) {
            equipamento.virarDireita();
            equipamento.virarEsquerda();
            }
      }
```

Agora não temos mais o erro no código principal, porque tanto Carro quanto Fliperama implementam a interface IControleVolante. Observe, porém, que agora NÃO PODEMOS acessar métodos específicos das classes Carro e/ou Fliperama dentro do método dirigir. Assim, por exemplo, a classe Carro possui o método setModelo() mas, se eu usar o código abaixo, vou ter um erro:

Pessoa.java

```
public class Pessoa {
      public void dirigir(IVolante equipamento) {
            equipamento.virarDireita();
            equipamento.virarEsquerda();
            equipamento.setModelo("Vectra");
            }
      }
```

Isso ocorre porque quando dissemos que este método poe usar QUALQUER objeto que implemente a interface IControleVolante, nos comprometemos a usar métodos APENAS desta interface. Como o método setModelo **não está** na interface IControleVolante, eu não posso usá-lo. Isso não me impede, entretanto, de usá-lo na classe principal, onde eu sei exatamente quem é Carro e quem é Fliperama:

Main.java

```
public class Main {
      public static void main(String[] args) {
            Carro meuCarro;
            Fliperama meuFliperama;
            Pessoa eu;
            eu = new Pessoa();
            meuCarro = new Carro();
            meuCarro.setModelo("Omega");
            meuFliperama = new Fliperama();
            meuFliperama.setNomeDoJogo("Super Mário 57");

            System.out.println(meuCarro.getModelo());
            eu.dirigir(meuCarro);
            System.out.println(meuFliperama.getNomeDoJogo());
            eu.dirigir(meuFliperama);
            }
      }
```

NOTA: SEMPRE que for indicado que uma classe implementa uma interface, TODOS os métodos daquela interface precisam ser implementados. A falha em fazê-lo irá causar erros na hora de gerar o programa.

Existe uma forma de contornar esse problema parcialmente: declarar a classe em questão como abstrata:

```
public abstract class MinhaClasse implements IMinhaInterface {
        }
```

O uso da palavra "abstract" na declaração da classe impede que o java reclame de que algum método não tenha sido implementado; por outro lado, NÃO É POSSÍVEL CRIAR OBJETOS de classe abstrata.

Para que ela serve? Para ser extendida por outra, onde os métodos da Interface faltantes deverão ser implementados.

3.2. Atributos em Interfaces (OPCIONAL)

Em princípio, não se deve indicar atributos nas interfaces; caso eles sejam indicados, só poderão ser do tipo "**static final**", isto é, valores que serão os mesmos em qualquer uma das classes que implementem a interface e que não podem ser mudados.

Em geral esse tipo de recurso é usado para indicar a versão da classe, como mostrado no código a seguir:

IControleVolante.java

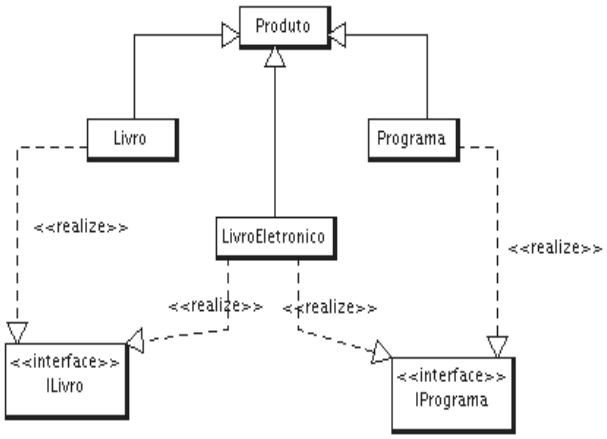
```
public interface IVolante {
      public static final int version = 17;

      public void virarDireita();
      public void virarEsquerda();
      }
```

Observe, porém, que nem todo tipo de dado pode ser usado; em especial, os tipos comumente definidos desta forma são os inteiros (int) e os booleanos (boolean).

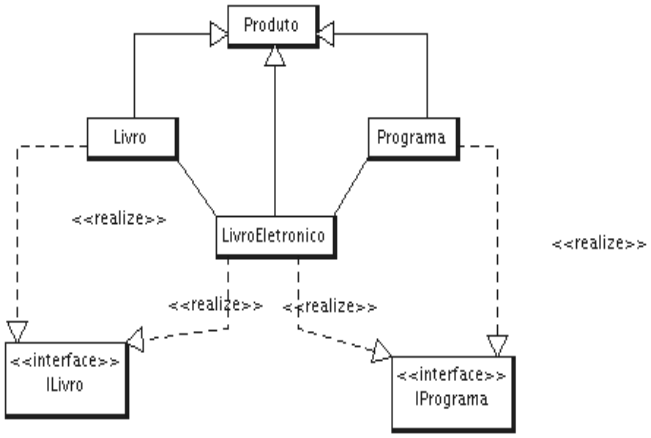
3.3. Usando Interfaces e Reaproveitando Código (OPCIONAL)

Como deve estar claro, até pelo exemplo citado, o uso de interfaces não permite reaproveitamento direto de código... mas isso não significa que não possamos "trapacear".

A idéia é usar uma "classe casca" e, para mostrar como isso é feito, usaremos o nosso exemplo anterior, da livraria. Para refrescar a memória, o exemplo está reproduzido a seguir.



Neste exemplo, embora seja possível o uso do polimorfismo entre objetos do tipo Livro e LivroEletronico (através da interface ILivro) e entre objetos do tipo Programa e LivroEletronico (através da interface IPrograma), os códigos referentes aos métodos destas interfaces precisam estar repetidos em todas as classes que as implementam. Em alguns casos é possível "trapacear" da seguinte forma:



Isso significa que LivroEletronico agora "tem" um Livro e "tem" um Programa, e eu vou usá-los para executar algumas tarefas, ao invés de re-implementá-las. Por exemplo: imagine que a ILivro seja declarada assim:

ILivro.java

```
public interface ILivro {
      public int getNumPaginas();
      }
```

E, a interface IPrograma, seja definida assim:

IPrograma.java

```
public interface ILivro {
      public int getNumBytes();
      }
```

Bem, as classes Livro e Programa poderiam ser definidas assim:

Livro.java

```
public class Livro implements ILivro {
      // Atributos do livro
      private int pags;

      // Construtor
      public Livro(int numPags) { pags = numPags; }

      // Métodos da ILivro
      public int getNumPaginas() { return pags; }
      }
```

Programa.java

```
public class Programa implements IPrograma {
      // Atributos do programa
      private int bytes;

      // Construtor
      public Programa(int numBytes) { bytes = numBytes; }

      // Métodos da IPrograma
      public int getNumBytes() { return bytes; }
      }
```

Agora, para podermos criar a classe LivroEletronico **sem** ter que reprogramar os métodos getNumPaginas() e getNumBytes() - que num programa real podem ser bastante complexos, eu posso construir a minha classe LivroEletronico assim:

LivroEletronico.java

```
public class LivroEletronico implements ILivro, IPrograma {
      // Atributos do LivroEletronico
      private Livro umLivro;
      private Programa umPrograma;

      // Construtor
      public LivroEletronico(int numPags, int numBytes) {
            umLivro = new Livro(numPags);
            umProgroama = new Programa(numBytes);
            }

      // Métodos da ILivro e IPrograma
      public int getNumPaginas() { return umLivro.getNumPaginas; }
      public int getNumBytes() { return umPrograma.getNumBytes; }
      }
```

Observe que ao invés de reimplementar os métodos getNumPaginas() e getNumBytes(), eu implementei métodos que apenas "redirecionam" a responsabilidade para outros objetos.

4. ATIVIDADES DE FIXAÇÃO (NÃO VALEM NOTA!)

Relembrando, uma interface pode ser declarada da seguinte forma:

```
public inferface nome {
        // definição dos métodos da interface
}
```

Por exmeplo:

IPessoa.java

```
/**
 * Interface Pessoa
 *
 * @author (seu nome)
 * @version 20100306_001  <<== Versão é muito importante!
 */

public interface IPessoa {
   public String toString();
   public String getNome();
   public int getIdade();
   public void setNome(String novoNome);
   public void setIdade(int novaIdade);
   }
```

A classe Pessoa, por exemplo, pode ser definida como "implementando" a classe InterfacePessoa:

Pessoa.java

```
/**
 * A classe Pessoa define características importantes de uma pessoa.
 *
 * @author (seu nome)
 * @version 20100306_001  <<== Versão é muito importante!
 */

public class Pessoa implements IPessoa {
   //...
```

1. Estude o código disponível no site, no arquivo pc_ap08_code.zip, e modifique as classes Homem e Mulher nele implementadas para que implementem a interface IPessoa SEM extender a classe Pessoa.

2. Essa abordagem foi sábia? Por quê?

BIBLIOGRAFIA

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.

SUN MICROSYSTEMS: http://java.sun.com/j2se/5.0/docs/api/index.html

Unidade 9: Arquitetura de Componentes

Prof. Daniel Caetano

Objetivo: Produzir e compreender documentações de componentes.

Bibliografia: DEITEL, 2005; HOFF, 1996.

INTRODUÇÃO

Como já comentado em aula, componentes são blocos autônomos que compõem um sistema. O funcionamento de um componente pode ser simples ou complexo, envolvendo uma ou até centenas ou milhares de classes; o uso de um componente, entretanto, deve ser mantido o mais simples possível.

Na aula passada foram apresentados os primeiros detalhes que permitem a especificação da arquitetura de componentes de uma aplicação:

a) As interfaces
b) Classes que forma os componentes da aplicação
c) Suas relações estruturais
d) As dependências entre eles

Nesta aula veremos com detalhe como deve ser especificado um componente.

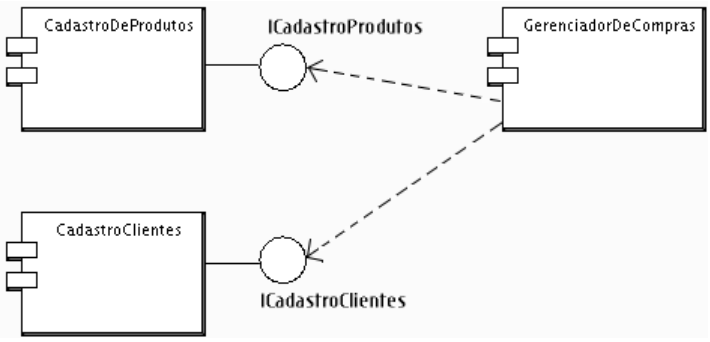
1. PROJETO DE ARQUITETURA

Um projeto de arquitetura de componentes de um sistema deve responder basicamente a três perguntas:

a) Quais são as partes do sistema?
b) Como combiná-las?
c) Qual o impacto das mudanças que possam ocorrer?

Em grande parte, os diagramas de classes vistos já apresentam estes elementos, em especial os diagramas que envolviam as interfaces, como ILivro e IArquivo, que pode ser consideradas as interface de componentes muito simples.

Muitas vezes, entretanto, é preciso representar a interação entre diversos componentes. Esta representação de interação é feita conforme o diagrama a seguir, chamado de Diagrama de Componentes.



Neste diagrama temos uma representação simplificada dos componentes, na notação UML: uma caixa com o nome do componente. O componente que é utilizado por outros componentes usualmente possui uma interface, indicada pela bolinha com o nome da interface.

Um componente que use outros componentes é ligado à interface daquele componente por uma seta de dependência, indicando que mudanças na interface do componente usado pode obrigar mudanças no componente que o utiliza.

Este tipo de especificação indica que o elemento principal de um componente é, em geral, uma classe que implemente a interface de comunicação deste cliente, o que é chamado de **realização.**

Quando se define uma arquitetura de componentes, define-se também - **obrigatoriamente** - as interfaces destes componentes, para que um objeto ou outro componente possa interagir com esses componentes. A interface de um componente é altamente influenciada pelos **usos** que serão dados ao componente, ou seja, é bastante importante a especificação dos **casos de uso**.

A definição da interface define dois importantes contratos:

A) O contrato de uso
B) O contrato de realização

1.1. Contrato de Uso

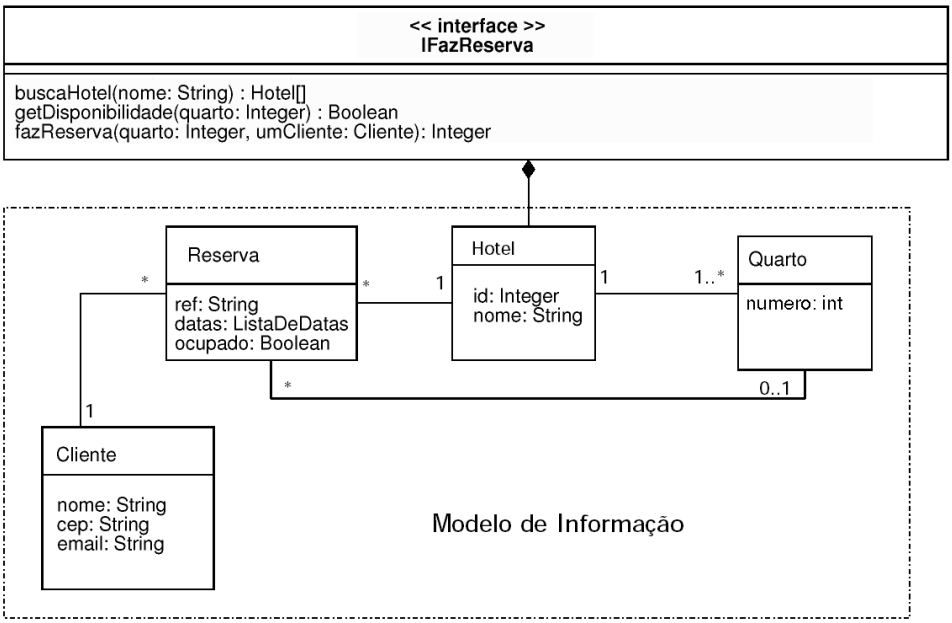
O Contrato de Uso é o que define como um componente é usado, isto é, o que ele faz e como ele pode ser usado para obter os resultados desejados.

Existem dois tipos de componentes: aqueles que o componente sempre reage de maneira igual, isto é, um componente "sem memória", que chamamos de "Stateless Components". Existem, porém, componentes mais complexos, que o resultado de uma operação depende do que aconteceu anteriormente, isto é, são componentes "com memória", que são chamaods "Statefull Components".

1.1.1. Stateless Components

Em alguns casos, os mais desejáveis, o uso do componente é simples: ele não guarda estado e, portanto, seu comportamento é sempre previsível. Neste caso, basta definir as **assinaturas das operações** e o **modelo de informação** (diagrama de classes de entidade).

As assinaturas são simplesmente os métodos públicos da interface: já vimos isso, é fácil. O modelo de informação é um pouco mais complexo, porque ele é composto de objetos de entidade, como especificado na figura a seguir (fonte: UML Components: Arquitetura de Componentes, de Patrícia Machado, UFCG, 2003).



Os objetos de entidade representados sob o retângulo "modelo de informação" indicam como os dados estão armazenados, isto é, o que é a informação do tipo Hotel e Cliente mencionadas na interface e a relação entre elas.

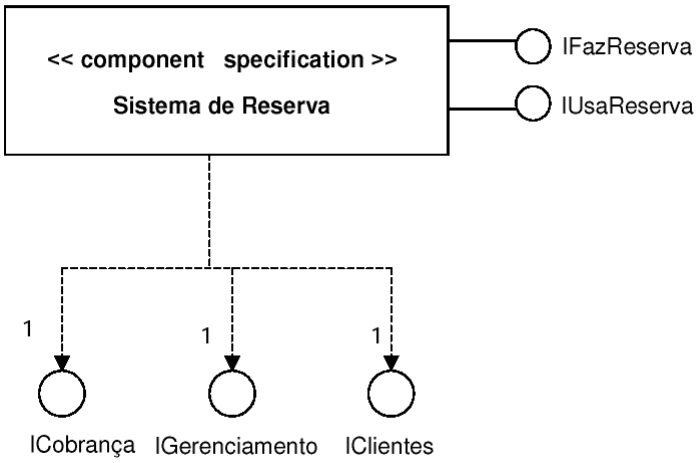
1.1.2. Statefull Components

Quando os componentes possuem estados, isto é, o resultado de uma operação pode variar drasticamente a depender das operações anteriores - como, por exemplo, a execução de uma instrução em um componente que emule um processador de computador - é preciso também especificar as **pré-condições** e as **pós-condições**.

As pré-condições e pós-condições explicitam o que é necessário fazer para que o componente se comporte de uma maneira esperada e o que é necessário fazer após algumas operações, para que ele continue se comportando de maneira esperada. A falha em seguir estas pré e pós-condições pode implicar em resultados imprevisíveis. Por exemplo: é preciso que o usuário faça o "logon" no sistema para que uma determinada opção funcione.

1.2. Contrato de Realização

O contrato de realização é basicamente a indicação de quais interfaces são oferecidas pelo componente e quais outras interfaces este componente usa, isto é, com que outros componentes ele interagem. A figura a seguir mostra este tipo de especificação, que deve ser seguida à risca pelo programador.



O componente, no caso, fornece as interfaces IFazReserva e IUsaReserva, e usa outros componentes, através das interfaces ICobranca, IGerenciamento e ICliente.

2. GERANDO DOCUMENTAÇÃO USANDO JAVA

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;
}
```

3. 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. Basicamente serve para documentar **componentes** que serão distribuídos a outros programadores que irão utilizar este componente.

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 e serve para documentar classes e componentes que serão extendidos por outros programadores.

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. É uma versão mais completa de Públicos e Protegidos.

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. Serve para documentar os componentes e classes para que possa ser feita sua manutenção futura.

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

3.1. 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:

Tag	**Significado Resumido**
@author	Indica o autor
@deprecated	Indica que não deve ser usado
@exception	Indica exceções
{@link}	Link pra outro documento
@param	Indica atributos de entrada
@return	Indica resultados de retorno
@see	Indicação para outros docs.
@serial / @serialData / @serialField	Possibilidade de serialização
@since	Em que versão foi criado
@throws	Indica lançamento de exceções
@version	Versão atual

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

4. CRIANDO UM COMPONENTE

Nessa aula criaremos o componente CadCli, que deve implementar a interface ICadCli. Esta interface é a mais simples possível, e define os seguintes métodos:

Cliente **novo**()
boolean **edita**(Cliente umCliente)
Cliente **busca**()
boolean **edita**()

PASSO 1: Vamos, primeiramente, criar um pacote para nosso cadastro de clientes. Clique com o botão direito em "Pacotes de Código Fonte" e selecione **Novo > Criar novo pacote java**. Dê o nome de **CadCli** para esse novo pacote.

PASSO 2: Mova (refatorando) para este novo pacote as seguintes classe: Cliente, ClienteEx, ClienteDialog, ClienteDAO.

PASSO 3: Vamos, agora, criar a interface do componente. Clique com o botão direito no pacote CadCli e selecione a opção **Novo > Interface java**. Dê o nome de **ICadCli** para esta interface.

PASSO 4: Modifique o código para indicar os métodos previamente especificados.

ICadCli.java

```
package cadcli;

/**
 * Interface para o componente CadCli
 * @author djcaetano
 */
public interface ICadCli {
    /**
     * Cria novo cliente, abrindo janela de edição.
     * @return objeto cliente criado
     */
    Cliente novo();

    /**
     * Edita um objeto de cliente.
     * @param umCliente Objeto cliente a ser editado.
     * @return True se cliente foi editado com sucesso.
     */
    boolean edita(Cliente umCliente);

    /**
     * Busca genérica com janela
     * @return Objeto cliente selecionado; Null se nenhum selecionado.
     */
    Cliente busca();

    /**
     * Busca e edita um cliente.
     * @return True se cliente foi editado com sucesso.
     */
    boolean edita();
}
```

PASSO 5: Crie, dentro do pacote CadCli, a classe **CadCli**. Para isso, clique com o botão direito no pacote CadCli e seleciona **Novo > Classe Java** e dê o nome de **CadCli** para ela.

PASSO 6: Modifique a classe, indicando que ela "implementa" a interface ICadCli:

CadCli.java

```
package cadcli;

/**
 * Controlador do Componete CadCli
 * @author djcaetano
 */
public class CadCli implements ICadCli {

}
```

PASSO 7: Acrescente os métodos de ICadCli, que inicialmente não farão nada, mas são necessários para que o Java compile o nosso código. Observe o código seguinte:

CadCli.java

```
package cadcli;

/**
 * Controlador do Componete CadCli
 * @author djcaetano
 */
public class CadCli implements ICadCli {

    Cliente novo() {
       return null;
       }
```

```
    boolean edita(Cliente umCliente) {
       return false;
       }

    Cliente busca() {
       return null;
       }

    boolean edita() {
       return false;
       }
}
```

PASSO 8: Vamos, agora, acrescentar um elemento fundamental para o CadCli: uma referência para a janela da aplicação. Esse elemento é fundamental porque o CadCli abrirá janelas de diálogo e, quando o fizer, precisará da referência da janela principal.

CadCli.java

```
package cadcli;

/**
 * Controlador do Componete CadCli
 * @author djcaetano
 */
public class CadCli implements ICadCli {

    private JFrame janela;

    Cliente novo() {
       return null;
       }

    boolean edita(Cliente umCliente) {
       return false;
       }

    Cliente busca() {
       return null;
       }

    boolean edita() {
       return false;
       }
}
```

PASSO 9: Agora precisamos de um construtor. Você pode usar o contrutor automático do NetBeans:

CadCli.java

```
package cadcli;

/**
 * Controlador do Componete CadCli
 * @author djcaetano
 */
public class CadCli implements ICadCli {

    private JFrame janela;

    public CadCli(JFrame umaJanela) {
       janela = umaJanela;
       }

    Cliente novo() {
       return null;
       }

    boolean edita(Cliente umCliente) {
       return false;
       }
```

```
    Cliente busca() {
       return null;
       }

    boolean edita() {
       return false;
       }
}
```

PASSO 10: Modifique agora a classe JPrincipal para que ela tenha uma referência para o cadastro de clientes, chamada **cadastro**, do tipo ICadCli e, em seu construtor, crie um objeto CadCli, associando-o à referência **cadastro**, passando ela própria como janela principal.

PASSO 11: Usando os recursos do NetBeans, crie um menu **Cliente** e, dentro dele, o item de menu **Cadastrar**.

PASSO 12: Crie um código associado ao evento de ação de clique no menu. Acrescente a esse código uma chamada à **cadastro.novo()**.

PASSO 13: Vamos, agora, implementar o código no método novo() da classe CadCli, de maneira que ele realize o processo necessário: crie um objeto Cliente e o edite com o método **edita(Cliente umCliente)**.

PASSO 14: Agora modifique o método **edita(Cliente umCliente)** para que ele crie uma janela ClienteDialog para edição do objeto **umCliente** recebido.

PASSO 15: Se o cliente for editado com sucesso no passo anteiror, esse método deve, então, armazenar o objeto Cliente no banco de dados, usando **ClienteDAO.adiciona(umCliente)**.

5. BIBLIOGRAFIA

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.

SUN MICROSYSTEMS: http://java.sun.com/j2se/5.0/docs/api/index.html

*** Criando a Busca ***

1. No modo de Projeto da classe JPrincipal, no pacote SisCli, arraste um Item de Menu para o menu "Clientes" e dê o nome de "Busca".

2. Mude o texto deste item de menu para "Buscar" e mude o nome da variável do item para mClientesBuscar.

3. Clique com o botão direito no item de menu "Buscar" e selecione "Eventos > Action > actionPerformed", para criar a região de código apropriada.

4. O código deste menu deve simplesmente executar o método "busca" do cadastro de clientes. Isso pode ser feito assim:

JPrincipal.java (método mClientesBuscaActionPerformed)

```
-----------------------------------
   private void mClientesBuscarActionPerformed (java.awt.event.ActionEvent evt) {
      clientes.busca();
   }
-----------------------------------
```

5. Agora precisamos criar esse código em nosso componente, lá na classe CadCli. O código desse componente, primeiramente, precisa abrir uma janela para a indicação do CPF. Essa janela irá retornar, então, o CPF para que seja processada a busca. Assim, antes de mais nada, precisamos criar uma janela!

6. Clique com o botão direito no pacote "CadCli" e selecione "Novo > Formulário JDialog", e dê o nome "ClienteBuscaDialog" a ele. Nas propriedades, mude o texto título da janela (title) para "Buscar Cliente".

7. Agora vamos arrastar um JTextField para que o usuário possa digitar o CPF nele. Use um JLabel para indicar o que deve ser preenchido no campo (CPF: [ ]). Dê o nome de "cBusca" para a variável que representa o campo de texto de busca.

8. Acrescente agora um JButton "Buscar" e associe o nome "bBuscar" a ele.

9. Agora vamos acrescentar função ao botão "Buscar". Clique com o botão direito sobre o botão e selecione "Eventos > Action > actionPerformed" para criar o código do botão.

10. Bem, precisamos agora fazer duas coisas. Primeiro temos que criar um *atributo* na janela ClienteBuscaDialog, chamado "busca", com valor inicial igual a null:

```
private String busca = null;
```

11. Agora, usando o recurso "Inserir Código" do NetBeans, crie um getter para este atributo.

ClienteBuscaDialog.java (getBusca)

```
   public String getBusca() {
      return busca;
   }
```
---

12. Agora vamos cirar o código para bBuscaActionPerformed:

ClienteBuscaDialog.java (bBuscarActionPerformed)

```
   private void bBuscarActionPerformed(java.awt.event.ActionEvent evt) {
      busca = cBusca.getText();
      dispose();
   }
```

Esse código simplesmente transfere o valor do texto de busca para o atributo e fecha a janela.

13. Vamos, agora, para finalizar a janela, torná-la sempre *modal*, modificando o construtor comos e segue:

ClienteBuscaDialog.java (ClienteBuscaDialog)

```
   public ClienteBuscaDialog(java.awt.Frame parent) {
      super(parent, true);
      initComponents();
      // Centraliza a janela no Frame
      setLocationRelativeTo(parent);
      // Torna Visível
      setVisible(true);
   }
```
---

14. Elimine agora o método *main* da classe ClienteBuscaDialog, para finalizá-la.

15. Agora que já temos uma janela para coletar o CPF de busca, vamos usá-la para programar o método "busca" da classe CadCli. Ele deve pegar o CPF usando a ClienteBuscaDialog e, se for diferente de null, deve realizar a busca com a classe ClienteDAO. O resultado é um objeto cliente que deve ser devolvido pelo método:

CadCli (método busca)

```
   /**
    * Busca um cliente no sistema, com janela.
    * @return Cliente encontrado / selecionado.
    */
   public Cliente busca() {
      ClienteBuscaDialog dialogo = new ClienteBuscaDialog(janela);
      String busca = dialogo.getBusca();
      if (busca == null) return null;
      return ClienteDAO.busca(busca);
   }
```

---

16. O sistema reclamará que não existe o método ClienteDAO.busca(String). Bem, vamos implementá-lo!

ClienteDAO (método busca)

```
   /**
    * Busca Cliente cujo cpf seja igual a "texto".
    * @param texto Texto para realizar a busca.
    * @return Clientes cujo cpf seja igual a "texto".
    */
   public static Cliente busca(String texto) {
      // Cria um cliente, inicialmente vazio.
      Cliente c = null;
      // Realiza conexao.
      Connection conexao = Conexao.conecta();
      // Se não existe conexão... retorna a lista vazia.
      if (conexao == null) return c;

      // Tenta realizar a busca:
      try {
         // Cria transação
         Statement transacao = conexao.createStatement();
         // Define a query de busca em SQL padrão.
         String query  = "SELECT nome, cpf, credito";
         query += " FROM clientes WHERE ";
         query += "cpf LIKE '" + texto + "';";
```

```
        // Executa busca e pega resultados
        ResultSet res = transacao.executeQuery(query);
        // Se houve uma linha encontrada no banco de dados,
        // cria um objeto e associa ao cliente de resposta
        if (res.next()) { // Se há linhas na resposta do BD
          try {
            String cpf = res.getString("cpf"); // pega o CPF
            String nome = res.getString("nome"); // Pega o nome
            int credito = res.getInt("credito"); // Pega o crédito
            // Monta objeto Cliente
            c = new Cliente(cpf, nome, credito);
          }
          // Considerando que o banco está correto, nunca deve ocorrer
          // qualquer erro nessa etapa, daí o catch não ter código.
          catch(ClienteEx ex) { }
        }
        // Finaliza a transação
        transacao.close();
      }
      // se houve algum erro nas transações de SQL...
      catch (SQLException ex) {
        // Não faz nada... código apenas para debug
        //System.err.println(ex);
      }
      // Retorna a lista de clientes encontrados.
      return c;
    }
```
---

17. Isso completa a nossa busca. Entretanto, no JPrincipal nós não fazemos nada com o cliente encontrado e, sendo assim, não é possível testar o código direito! Para poder testar, vamos modificar o código de JPrincipal.buscar para que, após encontrar o cliente, passe a editá-lo:

JPrincipal (método mClientesBuscarActionPerformed)

```
    private void mClientesBuscarActionPerformed(java.awt.event.ActionEvent evt) {
      Cliente c = clientes.busca();
      if (c != null) clientes.edita(c);
    }
```

Teste e veja se funciona!

18. Para finalizar nosso sistema, vamos criar a opção "Edita" no menu! Para isso, no modo de Projeto da classe JPrincipal, no pacote SisCli, arraste um Item de Menu para o menu "Clientes".

19. Mude o texto deste item de menu para "Buscar" e mude o nome da variável do item para mClientesEditar.

20. Clique com o botão direito no item de menu "Editar" e selecione "Eventos > Action > actionPerformed", para criar a região de código apropriada.

21. O código deste menu deve simplesmente executar o método "edita" do cadastro de clientes. Isso pode ser feito assim:

JPrincipal.java (método mClientesEditarActionPerformed)

```
   private void mClientesEditarActionPerformed(java.awt.event.ActionEvent evt) {
      clientes.edita();
   }
```
---

22. Vamos implementar o método edita() da nossa classe CadCli, o único que ainda não foi implementado. Ele deve realizar uma busca e, com o Cliente resultante, deve abrir a janela de edição.

CadCli (método busca)

```
   /**
    * Edita clientes, com busca precedente.
    * @return True se clienet foi editado com sucesso.
    */
   public boolean edita() {
      Cliente c = busca();
      if (c != null) {
         return edita(c);
      }
      return false;
   }
```

Experimente! Veja como funciona.

Na próxima aula veremos como implementar uma busca mais complexa, quando houver múltiplos resultados!

*** Mudando o DAO

1. Vamos começar modificando o nosso método ClienteDAO.busca(). Observe as mudanças abaixo:

```
/**
 * Busca Lista de cliente cujo cpf ou nome contenha "texto".
 * @param texto Texto para realizar a busca.
 * @return Lista de clientes encontrados.
 */
public static ArrayList<Cliente> busca(String texto) {
    // Cria um cliente, inicialmente vazio.
    Cliente c = null;
    // Cria uma lista, inicialmente vazia
    ArrayList<Cliente> lista = new ArrayList<Cliente>();
    // Realiza conexao.
    Connection conexao = Conexao.conecta();
    // Se não existe conexão... retorna a lista vazia.
    if (conexao == null) return lista;

    // Tenta realizar a busca:
    try {
        // Cria transação
        Statement transacao = conexao.createStatement();
        // Define a query de busca em SQL padrão.
        String query  = "SELECT nome, cpf, credito";
        query += " FROM clientes WHERE ";
        query += "cpf LIKE '%" + texto + "%'";
        query += " || ";
        query += "nome LIKE '%" + texto + "%'";
        // Executa busca e pega resultados
        ResultSet res = transacao.executeQuery(query);
        // Se houve uma linha encontrada no banco de dados,
        // cria um objeto e associa ao cliente de resposta
        while (res.next()) { // Se há linhas na resposta do BD
           try {
               String cpf = res.getString("cpf"); // pega o CPF
               String nome = res.getString("nome"); // Pega o nome
               int credito = res.getInt("credito"); // Pega o crédito
               // Monta objeto Cliente
               c = new Cliente(cpf, nome, credito);
               // Guarda cliente na lista
               lista.add(c);
           }
           // Considerando que o banco está correto, nunca deve ocorrer
           // qualquer erro nessa etapa, daí o catch não ter código.
           catch(ClienteEx ex) { }
        }
        // Finaliza a transação
        transacao.close();
    }
    // se houve algum erro nas transações de SQL...
    catch (SQLException ex) {
        // Não faz nada... código apenas para debug
        //System.err.println(ex);
    }
    // Retorna a lista de clientes encontrados.
    return lista;
}
```

2. Agora, precisamos alterar o nosso método CadCli.busca() para lidar com a nova situação. Primeiro ele deve verificar: a) se não houve resultados, retorna null; se houve um único resultado, retorna esse cliente.

```
/**
 * Busca um cliente no sistema, com janela.
 * @return Cliente encontrado / selecionado.
 */
public Cliente busca() {
    ClienteBuscaDialog dialogo = new ClienteBuscaDialog(janela);
    String busca = dialogo.getBusca();
    if (busca == null) return null;
    ArrayList<Cliente> lista = ClienteDAO.busca(busca);
    if (lista.isEmpty()) return null;
    if (lista.size() == 1) return lista.get(0);
    // Temporário, para satisfazer o Java
    return null;
}
```

3. Isso ainda não resolve nossos problemas... O que fazer se mais de um cliente for encontrado? Bem, precisamos criar uma janela que nos permita selecionar um dos clientes. Vamos começar criando um novo diálogo chamado ClienteSelecionaDialog. Clique com o botão direito no pacote **CadCli**, e crie um novo **Formulário JDialog** e dê o nome de **ClienteSelecionaDialog**.

4. Essa janela será sempre modal, e precisará receber a lista de clientes. Sendo assim, modifique o construtor dela da seguinte forma:

```
/** Creates new form ClienteSelecionaDialog */
public ClienteSelecionaDialog(java.awt.Frame parent, ArrayList<Cliente> lista)
{
    super(parent, true);
    initComponents();
    setLocationRelativeTo(parent);
    setVisible(true);
}
```

5. Agora **apague** o método **main** dessa nova janela.

6. Vamos, agora, criar na janela uma **JComboBox** e dê o nome da variável que a representa como **cClienteList**.

7. Procure, nas Propriedades da JComboBox, a propriedade **model** e **apague** o conteúdo.

8. Vamos ver se conseguimos usar essa janela. O primeiro passo é pedir ao método CadCli.busca() que a execute. Faça isso com o código a seguir:

```
/**
  * Busca um cliente no sistema, com janela.
  * @return Cliente encontrado / selecionado.
  */
 public Cliente busca() {
    ClienteBuscaDialog dialogo = new ClienteBuscaDialog(janela);
    String busca = dialogo.getBusca();
    if (busca == null) return null;
    ArrayList<Cliente> lista = ClienteDAO.busca(busca);
    if (lista.isEmpty()) return null;
    if (lista.size() == 1) return lista.get(0);
    ClienteSelecionaDialog dialogo2 = new ClienteSelecionaDialog(janela,lista);
    // Temporário, para satisfazer o Java
    return null;
 }
```

9. Execute o programa, adicione uns 3 clientes com nomes iguais e veja o que ocorre quando busca por esse nome (aproveite e mude a janela ClienteBuscaDialog para que, ao invés de **cpf**, indique **busca**.

10. Você deve ter reparado que a janela apareceu, mas a ComboBox ficou vazia. Vamos "populá-la" com os elementos da lista de clientes. Isso será feito no construtor da janela **ClienteSelecionaDialog**:

```
 /** Creates new form ClienteSelecionaDialog */
public ClienteSelecionaDialog(java.awt.Frame parent, ArrayList<Cliente> lista)
     {
     super(parent, true);
     initComponents();
     // Popula ComboBox
     int ncli = lista.size();
     int i;
     for (i=0; i<ncli; i++) {
         cClienteList.addItem(lista.get(i));
     }
     setLocationRelativeTo(parent);
     setVisible(true);
 }
```

11. Execute e veja o que ocorre. Bem, agora precisamos colocar um botão **JButton**, com o texto "Seleciona!". Dê o nome de **bSeleciona** à variável.

12. O código do botão deve, antes de mais nada, pegar o cliente selecionado na lista e guardá-lo em um atributo e, em seguida, indicar **dispose** na janela. Então, primeiramente, criemos o atributo **cliente** no topo da classe **ClienteSelecionaDialog**:

```
private Cliente cliente;
```

13. Crie o **getter** para esse atributo.

14. Agora vamos criar o método da ação do botão: **bSelecionaActionPerformed** da seguinte forma:

```
private void bSelecionaActionPerformed(java.awt.event.ActionEvent evt) {
    // Guarda cliente selecionado no atributo
    cliente = (Cliente)cClienteList.getSelectedItem();
    // Fecha a janela
    dispose();
}
```

15. Pronto, a janela já funciona! Experimente! Só falta "colhermos" o cliente selecionado! Para isso, finalizemos o método CadCli.busca():

```
/** Busca um cliente no sistema, com janela.
 * @return Cliente encontrado / selecionado.
 */
public Cliente busca() {
    ClienteBuscaDialog dialogo = new ClienteBuscaDialog(janela);
    String busca = dialogo.getBusca();
    if (busca == null) return null;
    ArrayList<Cliente> lista = ClienteDAO.busca(busca);
    if (lista.isEmpty()) return null;
    if (lista.size() == 1) return lista.get(0);
    ClienteSelecionaDialog dialogo2 = new ClienteSelecionaDialog(janela,lista);
    return dialogo2.getCliente();
}
```

*** Criando uma ComboBox Automática ***

PASSO 1. Abra o SisCli.

PASSO 2. Clique com o botão direito no pacote **campos** e selecione **Novo > Classe Java**. Dê o nome de **JEstadoCivilBox**.

PASSO 3. Vamos transformar essa classe em uma JComboBox, dizendo que ela **extends** a classe JComboBox, como indicado a seguir:

```
----------------------------------------
package campos;
import javax.swing.*;

/**
* ComboBox que mostra diferentes estados civis
* @author djcaetano
*/
public class JEstadoCivilBox extends JComboBox {

}
----------------------------------------
```

PASSO 4. Agora vamos criar um Construtor que popule esta ComboBox:

```
----------------------------------------
package campos;
import javax.swing.*;

/**
* ComboBox que mostra diferentes estados civis
* @author djcaetano
*/
public class JEstadoCivilBox extends JComboBox {

   public JEstadoCivilBox() {

       addItem("Casado");
       addItem("Solteiro");

   }
}
-------------------------------------------------------------
```

PASSO 5. Podemos, agora, arrastar esse elemento para a janela de edição do cliente: ClienteDialog. Coloque também um rótulo JLabel, com o texto "**Estado Civil:**". Dê o nome de **cEstadoCivil** à ComboBox.

PASSO 6. Precisamos agora, modificar tudo relacionado ao cliente para que contenha também o estado civil. Vamos começar pelo ClienteEx:

```
-------------------------------------------------------------
package CadCli;

/**
* Objeto de erro de criação de objetos Cliente.
* Esta classe armazena códigos de erro relativos à criação de objetos
* do tipo Cliente.
* @author djcaetano
*/
```

```
public class ClienteEx extends Exception {
   /**
    * Tipos de Erros possíveis na criação de Cliente.
    */
    public static enum Id { ERRO_CPF, ERRO_NOME, ERRO_CREDITO, ERRO_CIVIL };

   /**
    * Atributo que armazena o erro ocorrido, inicialmente  vazio.
    */
   private Id erro = null;

   /**
    * Construtor do objeto de erro.
    * @param erro Código de erro, dentre aquels descritos na enum Erros.
    */
   public ClienteEx(Id umErro) {
       erro = umErro;
   }

   /**
    * Recupera o código de erro do objeto de erro.
    * @return Código de erro, de acordo com os valores descritos na enum Erros.
    */
   public Id getErro() {
       return erro;
   }
}
```

PASSO 7. Agora, vamos editar a classe **Cliente**, acrescentando o atributo, getter, setter e modificando o construtor.

```
package CadCli;
/**
* Classe que representa um cliente do sistema
* @author djcaetano
*/
public class Cliente {
   // Atributos privados
   private String cpf;     // CPF do cliente
   private String nome;    // Nome do cliente
   private int credito;    // Crédito, expresso em centavos
    private String civil;   // Armazena estado civil

   /**
    * Construtor de cliente vazio.
    * Não valida nenhuma informação.
    */
   public Cliente() {
       cpf = "";
       nome = "";
       credito = 0;
        civil = "";
   }

   /**
    * Construtor de Cliente.
    * Constrói um objeto de cliente completo e validado.
    * @param cpf CPF do cliente.
    * @param nome Nome do cliente a ser criado.
    * @param credito Crédito, em centavos, do cliente.
    * @param civil Estado civil do cliente.
    * @throws ClienteEx Indica qual a falha que houve na criação do objeto.
    */
    public Cliente(String cpf, String nome, int credito, String civil)
            throws ClienteEx {
        if (setCpf(cpf) == false)
            throw new ClienteEx(ClienteEx.Id.ERRO_CPF);
       if (setNome(nome) == false)
            throw new ClienteEx(ClienteEx.Id.ERRO_NOME);
       if (setCredito(credito) == false)
            throw new ClienteEx(ClienteEx.Id.ERRO_CREDITO);
        if (setCivil(civil) == false)
            throw new ClienteEx(ClienteEx.Id.ERRO_CIVIL);
    }
```

```java
 /**
 * Retorna o estado civil do cliente.
 * @return Estado civil do cliente.
 */
public String getCivil() {
    return civil;
}

/**
 * Muda o estado civil do cliente.
 * @return true se a mudança ocorreu com sucesso.
 */
public boolean setCivil(String civil) {
    if (civil == null) return false;
    this.civil = civil;
    return true;
}

/**
 * Retorna o objeto CPF do cliente, só números.
 * @return Cpf do cliente, na forma de objeto.
 */
public String getCpf() {
    if (cpf == null) return "";
    return cpf;
}

/**
 * Muda o objeto CPF do cliente.
 * @param cpf Novo CPF para o cliente.
 * @return True se a mudança ocorrer com sucesso.
 */
public final boolean setCpf(String cpf) {
    // Se nenhum CPF fornecido, vai embora com erro.
    if (cpf == null) return false;
    // Limpa espaços, pontos e traços
    cpf = cpf.trim();
    cpf = cpf.replaceAll(" ","");
    cpf = cpf.replaceAll("[.-]","");
    // Pega o comprimento do cpf já limpo.
    int cpflen = cpf.length();
    // Precisa estar preenchido com exatos 11 dígitos
    // Se não estiver, vai embora com erro.
    if (cpflen != 11) return false;
    // Precisa ser composto apenas por números
    for (int i=0; i<cpflen; i++ ) {
        // Se algum dos caracteres não for um dígito numérico, vai embora
        // com erro.
        if (Character.isDigit(cpf.charAt(i)) == false) return false;
    }
    // No caso real, seria necessário testar também o dígito de verificação!
    // Se chegou aqui, é porque todas as validações foram feitas com sucesso,
    // e, assim, podemos guardar o novo CPF no atributo.
    this.cpf = cpf;
    return true;
}

/**
 * Recupera o valor do crédito do cliente.
 * @return O valor do crédito do cliente, em centavos.
 */
public int getCredito() {
    if (credito < 0) return 0;
    return credito;
}

/**
 * Define um novo limite de crédito para o cliente.
 * @param credito Novo limite de crédito, em centavos. Deve ser não negativo.
 * @return True se a mudança ocorreu com sucesso.
 */
public final boolean setCredito(int credito) {
    // Se crédito for negativo, vai embora com false.
    if (credito < 0) return false;
    this.credito = credito;
    return true;
}
```

```
    /**
     * Retorna o nome do cliente.
     * @return Retorna o nome do cliente ou vazio se não houver um nome definido.
     */
    public String getNome() {
        if (nome == null) return "";
        return nome;
    }

    /**
     * Modifica nome do cliente.
     * @param nome Novo nome. Deve ter mais que 4 caracteres.
     * @return True se a mudança ocorreu com sucesso.
     */
    public final boolean setNome(String nome) {
        // Se nome muito curto, vai embora com false.
        if (nome == null || nome.length() < 5) return false;
        this.nome = nome;
        return true;
    }

    /**
     * Transforma objeto na forma textual.
     * @return Texto na forma: Nome (cpf).
     */
    @Override
    public String toString() {
        return getNome() + " (" + getCpf() + ")";
    }

     public String debug() {
        return cpf + " : " + nome + " : " + credito + " : " + civil;
    }
}
```

PASSO 8. Vamos agora modificar a classe **ClienteDAO** para armazenar o estado civil:

```
package CadCli;
import java.sql.*;
import Conexao.Conexao;
import java.util.ArrayList;

/**
* Responsável por criar e armazenar clientes no banco de dados.
* Esta classe torna o uso do banco de dados "transparente" para o resto do
* sistema. Seus métodos devem ser construídos, dentro do possível, usando
* linguagem SQL padrão, para que o código seja compatível com qualquer
* banco de dados existente.
* @author djcaetano
*/
public class ClienteDAO {
    /**
     * Construtor do objeto DAO.
     * Não deve ser usado. Classe Estática
     */
    private ClienteDAO() {
    }

    /**
     * Adiciona/Atualiza um cliente no banco de dados.
     * @param umCliente Objeto de cliente a ser armazenado.
     * @return True se cliente foi armazenado com sucesso.
     */
    public static boolean adiciona(Cliente umCliente) {
        // Pega a conexao atual.
        Connection conexao = Conexao.conecta();
        // Se não existe conexão... retorna false.
        if (conexao == null) return false;
        // Se a conexão ocorreu, cria variáveis para realizar o update/insert.
        // Garante que textos da busca não contem aspas para evitar SQL Injection.
        String cpf = umCliente.getCpf();
        cpf = cpf.replaceAll("'","");
        String nome = umCliente.getNome();
        nome = nome.replaceAll("'","");
        String civil = umCliente.getCivil();
        civil = civil.replaceAll("'","");
```

```
     // Primeiramente, tenta fazer um update!
    try {
        // Cria a transação
        Statement transacao = conexao.createStatement();
        // Cria a query de update...
        String query  = "UPDATE clientes SET ";
        query += "nome = '" + nome + "', ";
        query += "credito = " + umCliente.getCredito();
        query += ", civil = '" + civil + "'";
        query += " WHERE ";
        query += "cpf LIKE '" + cpf + "'";
        // Se update falhar...
        if ( transacao.executeUpdate(query) == 0 ) {
            // Cria query de insert...
            query  = "INSERT INTO clientes ";
            query += "VALUES ('" + cpf + "', ";
            query += "'" + nome + "', " + umCliente.getCredito();
            query += ", '" + civil + "'";
            query += ")";
            // Se insert falhar, retorna erro
            if ( transacao.executeUpdate(query) == 0 ) {
                transacao.close();
                return false;
            }
        }
        // Finaliza a transação e sai.
        transacao.close();
        return true;
    }
    // se houve algum erro nas transações de SQL...
    catch (SQLException ex) {
        // Não faz nada... código apenas para debug
        //System.err.println(ex);
    }
    // Se chegou aqui, não foi possível realizar a atualização.
    return false;
}

/**
 * Busca Lista de cliente cujo cpf ou nome contenha "texto".
 * @param texto Texto para realizar a busca.
 * @return Lista de clientes encontrados.
 */
public static ArrayList<Cliente> busca(String texto) {
    // Cria um cliente, inicialmente vazio.
    Cliente c = null;
    // Cria uma lista, inicialmente vazia
    ArrayList<Cliente> lista = new ArrayList<Cliente>();
    // Realiza conexao.
    Connection conexao = Conexao.conecta();
    // Se não existe conexão... retorna a lista vazia.
    if (conexao == null) return lista;
    // Tenta realizar a busca:
    try {
        // Cria transação
        Statement transacao = conexao.createStatement();
        // Define a query de busca em SQL padrão.
        String query  = "SELECT nome, cpf, credito, civil";
        query += " FROM clientes WHERE ";
        query += "cpf LIKE '%" + texto + "%'";
        query += " || ";
        query += "nome LIKE '%" + texto + "%'";
        // Executa busca e pega resultados
        ResultSet res = transacao.executeQuery(query);
        // Se houve uma linha encontrada no banco de dados,
        // cria um objeto e associa ao cliente de resposta
        while (res.next()) { // Se há linhas na resposta do BD
           try {
               String cpf = res.getString("cpf"); // pega o CPF
               String nome = res.getString("nome"); // Pega o nome
               int credito = res.getInt("credito"); // Pega o crédito
               String civil = res.getString("civil"); // Pega estado civil
               // Monta objeto Cliente
               c = new Cliente(cpf, nome, credito, civil);
               // Guarda cliente na lista
               lista.add(c);
           }
           // Considerando que o banco está correto, nunca deve ocorrer
           // qualquer erro nessa etapa, daí o catch não ter código.
           catch(ClienteEx ex) { }
```

```
            }
            // Finaliza a transação
            transacao.close();
        }
        // se houve algum erro nas transações de SQL...
        catch (SQLException ex) {
            // Não faz nada... código apenas para debug
            //System.err.println(ex);
        }
        // Retorna a lista de clientes encontrados.
        return lista;
    }
}
```

PASSO 9. Antes de continuarmos as modificações, precisamos modificar nosso banco de dados. Entre no prompt do MySQL e use a sequência:

USE siscli;
DROP TABLE clientes;
CREATE TABLE clientes (cpf CHAR(11) NOT NULL, nome VARCHAR(200), credito INT, civil VARCHAR(20), PRIMARY KEY(cpf));

PASSO 10. Vamos, agora, mudar a janela ClienteDialog, de maneira que ela apresente o estado civil correto em sua criação:

ClienteDialog.java (método ClienteDialog)

```
-------------------------------------------------------
    public ClienteDialog(java.awt.Frame parent, Cliente umCliente) {
        // Inicializa a janela JDialog
        super(parent, true);
        // Coloca os componentes (botões, campos etc)
        initComponents();
        // Se não foi passado um objeto de cliente válido, fecha a janela.
        if (umCliente == null) {
            dispose();
            return;
        }
        // Caso o cliente seja válido, armazena ele no atributo
        clienteEmEdicao = umCliente;
        // Ajusta o valor dos campos nome e CPF
        cNome.setText(clienteEmEdicao.getNome());
        cCpf.setText( (clienteEmEdicao.getCpf()) );
        // Ajusta o valor do campo crédito: observe a divisão por 100,
        // Para que na janela o valor apareça em reais, ao invés de centavos.
        cCredito.setValue(((double)clienteEmEdicao.getCredito())/100);
        // Ajuda estado civil
        cEstadoCivil.setSelectedItem(clienteEmEdicao.getCivil());
        // Centraliza a janela no Frame
        setLocationRelativeTo(parent);
        // Torna a janela visível.
        setVisible(true);
    }
-------------------------------------------------------
```

PASSO 11. Agora, precisamos modificar o código do botão de confirmação de edição para que, quando ele for apertado, o dado do estado civil seja armazenado corretamente no cliente:

ClienteDialog.java (método bGravaActionPerformed)

```
-------------------------------------------------------
    private void bGravaActionPerformed(java.awt.event.ActionEvent evt) {

        // Se o nome digitado no campo for rejeitado pelo objeto cliente...
        if (clienteEmEdicao.setNome(cNome.getText()) == false) {
            // Indica erro...
            JOptionPane.showMessageDialog(this,"Nome de cliente inválido",
                    "Erro no preenchimento!",JOptionPane.ERROR_MESSAGE);
            // E coloca o foco no campo de nome.
```

```
            cNome.requestFocusInWindow();
            return;
        }
        // Aqui se a modificação do nome foi processada com sucesso...
        // Se o CPF for válido, troca o CPF do cliente pelo novo.
        if ( clienteEmEdicao.setCpf(cCpf.getText()) == false) {
            // Se cpf inválido... Mostra mensagem de erro...
            JOptionPane.showMessageDialog(this,"CPF inválido",
                    "Erro no preenchimento!",JOptionPane.ERROR_MESSAGE);
            // E joga o foco para o campo do CPF.
            cCpf.requestFocusInWindow();
            return;
        }
        // Se chegou aqui, é porque o CPF foi alterado com sucesso.
        // Finalmente, tenta converter o campo crédito para um número
        try {
            // Se conseguir converter, altera o crédito do cliente.
            int credito = Integer.parseInt(cCredito.getTextLimpo());
            // Se o valor for inválido, provoca exceção
            if ( clienteEmEdicao.setCredito(credito) == false) {
                throw new NumberFormatException("Valor inválido.");
            }
        }
        // Se o valor do crédito era inválido
        catch (NumberFormatException ex) {
            // Mostra mensagem de erro...
            JOptionPane.showMessageDialog(this,"Valor de crédito inválido",
                    "Erro no preenchimento!",JOptionPane.ERROR_MESSAGE);
            // E joga o foco no campo de crédito.
            cCredito.requestFocusInWindow();
            return;
        }

         // Tenta mudar o estado civil.
        if ( clienteEmEdicao.setCivil((String)cEstadoCivil.getSelectedItem()) == false) {
            // Se cpf inválido... Mostra mensagem de erro...
            JOptionPane.showMessageDialog(this,"Estado Civil inválido",
                    "Erro no preenchimento!",JOptionPane.ERROR_MESSAGE);
            // E joga o foco para o campo do CPF.
            cEstadoCivil.requestFocusInWindow();
            return;
        }
        // Finalmente, chegando aqui, significa que todos os atributos foram
        // atualizados com sucesso e, assim, indica-se TRUE como sendo o
        // novo resultado da operação...
        editado = true;
        // E solicitamos o fechamento da janela.
        dispose();
```

PASSO 12. Tente, agora, cadastrar um cliente novo, realizar buscas... e verifique se o estado civil está sendo corretamente armazenado!

PASSO 13. Agora vamos um pouco além. Ao invés de criarmos um componente de estado civil com valores estáticos, vamos fazê-lo ler os estados civis do Banco de Dados. Para isso, vamos ao MySQL para criar uma tabela de estados civis:

```
USE siscli;
CREATE TABLE estadocivil (id INT NOT NULL, texto VARCHAR(20), PRIMARY KEY(id));
INSERT INTO estadocivil VALUES (0, "Solteiro");
INSERT INTO estadocivil VALUES (1, "Casado");
INSERT INTO estadocivil VALUES (2, "Separado");
INSERT INTO estadocivil VALUES (3, "Divorciado");
INSERT INTO estadocivil VALUES (4, "Viúvo");
```

PASSO 14. Vamos, agora, criar uma classe DAO que leia esses elementos. Clique com o botão direito no pacote **campos** e selecione **Novo > Classe Java** e dê o nome de **EstadoCivilDAO** a ela.

PASSO 15. Vamos criar um método busca como o indicado abaixo, que retorne uma lista de Strings:

```
------------------------------------------------------------
package campos;
import Conexao.Conexao;
import java.sql.*;
import java.util.ArrayList;

/**
*
* @author djcaetano
*/
public class EstadoCivilDAO {

   /**
    * Busca Lista de strings de estado civil.
    * @return Lista de strings de estado civil.
    */
   public static ArrayList<String> busca() {
       // Cria uma lista, inicialmente vazia
       ArrayList<String> lista = new ArrayList<String>();
       // Realiza conexao.
       Connection conexao = Conexao.conecta();
       // Se não existe conexão... retorna a lista vazia.
       if (conexao == null) return lista;

       // Tenta realizar a busca:
       try {
           // Cria transação
           Statement transacao = conexao.createStatement();
           // Define a query de busca em SQL padrão.
           String query  = "SELECT texto FROM estadocivil";
           // Executa busca e pega resultados
           ResultSet res = transacao.executeQuery(query);
           // Se houve uma linha encontrada no banco de dados...
           while (res.next()) { // Se há linhas na resposta do BD
               String texto = res.getString("texto");
               lista.add(texto);
              }
           // Finaliza a transação
           transacao.close();
       }
       // se houve algum erro nas transações de SQL...
       catch (SQLException ex) {
           // Não faz nada... código apenas para debug
           //System.err.println(ex);
       }
       // Retorna a lista de strings.
       return lista;
   }
}
------------------------------------------------------------
```

PASSO 16. Vamos, agora, modificar a classe **JCivilBox** para que use o DAO para receber a lista de estados civis!

```
-------------------------------------------------------------
package campos;
import java.util.ArrayList;
import javax.swing.*;

/**
* ComboBox que mostra diferentes estados civis
* @author djcaetano
*/
public class JEstadoCivilBox extends JComboBox {

   public JEstadoCivilBox() {

        ArrayList<String> lista = EstadoCivilDAO.busca();
       int nlista = lista.size();
       int i;
       for (i=0; i<nlista; i++) {
           addItem(lista.get(i));
       }

   }

}
-------------------------------------------------------------
```

PASSO 17. Pronto! Seus sistema está atualizado com uma caixa de estado civil para o cliente!