Introdução à Programação de Servidores com Java

Prof. Daniel Caetano

Objetivo: Apresentar os conceitos fundamentais da programação de servidores e da linguagem Java, bem como sua história e nomenclatura.

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

INTRODUÇÃO

O objetivo desta aula é proporcionar uma uniformização do conhecimento básico sobre o que é a programação para servidores, o que é a linguagem Java, para apoio de todas as aulas subsequentes. Esta aula está dividida nas seguintes seções:

1) O que é Programação para Servidores?
2) Histórico da Linguagem Java
3) Ambientes Java
4) A Linguagem Java e as Máquinas Virtuais

1. PROGRAMAÇÃO PARA SERVIDORES

A maioria dos alunos da área de TI já está familiarizada com o conceito de software: praticamente qualquer pessoa do mundo moderno já tomou contato com um programa de computador, em especial aquelas que escolhem um curso voltado ao desenvolvimento deste tipo de produto.

A maioria dos programas aos quais as pessoas estão acostumadas - como o Word, o Excel, os jogos e tantos outros - são aqueles programas que são executados no próprio computador do usuário. Entretanto, desde o surgimento da Internet, os usuários se acostumaram com um outro tipo de recurso: aqueles disponíveis através da rede. Alguns usuários se perguntam: onde fica a internet? Onde fica uma página de internet? Onde fica o Google? Onde fica o Facebook?

Antes de uma análise mais apurada sobre o local no qual estes sistemas estão disponíveis, convém responder primeiramente **o que** são esses sistemas. O que é o Google? O que é o Facebook? O que é o gMail?

A resposta é muito simples: eles são **serviços.**

É notório que estes serviços não estão na máquina do usuário: quando a rede cai, o usuário perde acesso a eles. Esses serviços ficam em algum outro lugar, que não sabemos

exatamente qual é. A resposta mais comum para a localização destes serviços nos tempos atuais é... "**na nuvem**". Mas você sabe o que é "a nuvem"?

A "nuvem" é um termo que foi criado para designar a rede formada pela internet, incluindo todos os computadores do mundo - e isso vale para o seu também. Quando se diz que um documento, um site ou um serviço está "na nuvem", na verdade significa que ele está em um computador idêntico ao seu, embora muitas vezes façamos a menor idéia do local em que esse computador se encontra!

Os computadores que executam serviços como o Google, o Facebook e o gMail, exatamente por fornecerem serviços, são chamados de **servidores**. Há diversos tipos de serviços que um servidor pode fornecer, como e-mail, msn, páginas web, aplicações...

Os computadores que, por outro lado, usam esses serviços, são chamados de computadores **clientes**. Quando você navega na internet, o seu computador está atuando como **cliente** de diversos **servidores web**.

Para que você veja uma página qualquer, o seu computador (cliente) deve enviar uma **requisição** para o servidor, que enviará a resposta solicitada. Nosso computador, na realidade, não sabe como encontrar o servidor. Ele sabe apenas o **nome** do servidor (frequentemente indicado como uma URL ou um número IP).

Assim, para solicitar alguma coisa ao servidor, o computador cliente constrói um documento de requisição chamado **REQUEST**, que inclui, no mínimo, o **nome do servidor, nome da origem** e o **texto da requisição**. Este arquivo é enviado pele rede (nuvem), que se responsabiliza por entregar a requisição ao servidor cujo nome foi informado.

Ao chegar neste servidor, a requisição é processada e, então, o servidor cria um documento de resposta, chamado **RESPONSE**, que irá conter todo o texto da resposta daquela requisição. A resposta também contém um endereço de destino (nome do cliente) e o nome da origem (o servidor). Assim que a resposta está pronta, o servidor a envia através da rede que, mais uma vez, se responsabiliza pelo encaminhamento da mesma de volta até o computador do cliente.

No caso de uma página web, essa resposta é, usualmente, um arquivo HTML, que será usado pelo navegador para desenhar a página web para o usuário. Genericamente, todo aplicativo que aguarda requisições e envia respostas às requisições recebidas é chamado de **aplicativo servidor**. Um equipamento que executa um aplicativo servidor é, assim, um **equipamento servidor**.

Neste curso iremos desenvolver aplicativos servidores. Como a linguagem C/C++ não é exatamente prática para este tipo de desenvolvimento, iremos utilizar, neste curso, a Linguagem Java. A Linguagem Java tem algumas facilidades e vantagens, como uma grande infraestrutura pronta para o desenvolvimento de aplicativos servidores, incluindo uma vasta biblioteca com esta finalidade, além de uma enorme semelhança com a linguagem C/C++, o que facilita o aprendizado da mesma por programadores que já conhecem a linguagem C.

2. A LINGUAGEM JAVA

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

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

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

2.1. Histórico do Java

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

Origens

No início da década de 1990 estavam se popularizando os equipamentos eletro-eletrônicos programáveis/programados, indo desde televisores até fornos de microondas e geladeiras. Embora muitas empresas tivessem notado que as linguagens existentes traziam problemas para o desenvolvimento destes equipamentos, foi a Sun Microsystems quem primeiro propôs uma solução.

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

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

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

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

Entretanto, a troca de um processador muitas vezes implica em troca de todo o software, já que usualmente processadores distintos têm linguagens de máquina distintas. O problema então surge: a necessidade de se re-compilar e, muitas vezes, reescrever um software para o novo processador... acaba com grande parte do lucro obtido com a troca do processador. E, mesmo quando isso não ocorria, muitas vezes significava novos "bugs" e problemas, algo bastante indesejável. De olho nisso, em 1990, James Gosling começou a trabalhar em uma linguagem que funcionasse de tal forma que os programas raramente precisassem ser reescritos quando a plataforma onde são executados fosse substituída, desde que ambas oferecessem recursos similares. Essa linguagem acabou por ficar conhecida como Linguagem Java.

Projetos Iniciais

Raramente uma linguagem baseada apenas em teoria e sem experimentação prática consegue ter sucesso. Por esta razão, os técnicos da Sun Microsystems, durante o desenvolvimento do Java desenvolveram projetos em Java, para testar suas funcionalidades. O primeiro destes projetos foi o Projeto Green, que visava a criação de uma nova interface com o usuário para o equipamento "*7" (Star Seven), que tinha o objetivo de controlar os eletrodomésticos de uma casa através de ícones animados e uma touch screen. Um outro projeto foi o de VoD (Video On Demand), com uma função similar ao que hoje se chama de TV Interativa.

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

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

O Java Hoje

O tempo foi passando e mostrou que a Sun Microsystems, de alguma forma, estava adiante de seu tempo. Com o surgimento dos PDAs (Personal Data Assistants, os "PALMs") e telefones celulares capazes de executar aplicativos, tornou-se bastante atrativa uma tecnologia que permitisse que um programa pudesse ser executado em máquinas diferentes:

afinal de contas, não só os recursos disponíveis nestes equipamentos, como também seus processadores e arquiteturas podem ser bastante diferentes até mesmo de um modelo para outro!

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

3. 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: Java SE, Java ME e Java EE.

Java SE: Java Standard Edition - O Java SE 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.

Java ME: Java Micro Edition - O Java ME 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.

Java EE: Java Enterprise Edition - O Java EE é uma versão voltada ao desenvolvimento de aplicações que são executadas em ambiente servidor, incluindo todos os recursos necessários para seu uso 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 deste tipo de aplicação.

3.1. 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 Java SE, Java EE e Java ME, 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
1.7	Java 7	Simplificação de sintaxe e otimização de desempenho

4. 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/C++.

13. 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 1: Linguagem Java Para Programadores C

Variáveis, Operadores, Estruturas

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; a linguagem Java, entretanto, é suficientemente parecida com a linguagem C para que este aprendizado mínimo seja obtido rapidamente. 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) Um programa mínimo em Java
2) Variáveis e seus tipos
3) Operadores básicos
4) Controle de Fluxo
5) Tipos Não Nativos
6) Regras de Nomenclatura
7) Classes e Objetos
8) Tratamento de Erros

1. PROGRAMANDO EM JAVA

Uma vez que um programa em Java é muito parecido com um programa em C, será apresentado inicialmente um programa mínimo em Java, de maneira a explicar algumas das principais diferenças de estruturação.

Em Java, o programa (ou **projeto**) mínimo é composto de um único pacote, com uma única classe com um único método **main**. Criando-se o programa no NetBeans, com o nome de projeto de **MeuPrimeiroPrograma**, o código será similar ao que segue:

MeuPrimeiroPrograma.java

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

// Classe Principal
public class MeuPrimeiroPrograma {

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

}
```

Este texto, gravado no arquivo *MeuPrimeiroPrograma.java*, pode ser compilado e executado clicando no botão que tem um triângulo verde no NetBeans. Por ser um programa mínimo, este programa mostra o que minimamente é necessário à construção de um projeto Java organizado:

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

Sempre que criamos um projeto no NetBeans, ele cria esses três elementos. Em termos de **organização**, eles funcionam da seguinte forma: Um projeto pode ter vários pacotes. Cada pacote, por sua vez, pode conter várias classes. Finalmente, cada classe pode conter varios métodos.

Uma analogia útil é comparar esta hierarquia com aquela do Microsoft Office.

Projeto:	Microsoft Office
Pacote:	Microsoft Word
Classe:	Documento de Texto
Método:	Corrigir Ortografia

Uma vez que um projeto pode ter um grande número de pacotes, classes e métodos, o NetBeans precisa saber onde é que o programa começa, isto é, **qual código** ele deve executar.

A linguagem Java estabelece que, por padrão, uma classe é sempre executada pelo seu método principal, chamado obrigatoriamente de **main**. Entretanto, isso só resolve o nosso problema se o programa tiver apenas uma classe... mas isso não é o que normalmente ocorre!

Para que resolver esta dificuldade, foi estabelecido um padrão: todo programa Java do NetBeans tem sempre um pacote com o mesmo nome do projeto e, dentro dele, uma classe **com o mesmo nome do projeto**. Quando mandarmos o NetBeans executar um programa, ele sempre começará a execução a partir do método **main** desta classe específica.

1.1. Impressão de Informações

Enquanto no C/C++ usam-se as instruções **cout** e **printf**, em Java pode-se usar:

No Lugar de:	**Use:**
cout << "Texto";	System.out.print("Texto");
cout << var;	System.out.print(var);
cout << "Texto" << endl;	System.out.println("Texto");
cout << "Texto" << var;	System.out.print("Texto" + var);
printf ("Texto %d\n",var);	System.out. printf ("Texto %d\n",var);

1.2. Comentários de Código

Os comentários são fundamentais nos códigos Java, em especial para quem ainda está aprenendo a programar. 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.

2. VARIÁVEIS NATIVAS

As variáveis do Java funcionam, basicamente, da mesma maneira que as da linguagem C... ou seja, é preciso declará-las com um tipo. 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] ;

Exemplo:

```
int var;
int var = 5;
```

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, c/ sinal (-2.147484E+09 a +2.147484E+09)
long	número de 64 bits, c/ 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. Aaixo, o código adaptado para apresentar alguns dos tipos de valores:

MeuPrimeiroPrograma.java

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

// Classe Principal
public class MeuPrimeiroPrograma {

   // 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 );
   }
}
```

3. OPERAÇÕES EM JAVA

As operações básicas em Java são as mesmas do C:

+	soma	==	Comparação de igualdade
-	subtração	!=	Comparação de diferença
*	multiplicação	<=	Comparação de menor ou igual
/	divisão	>=	Comparação de maior ou igual
%	resto de divisão	<	Comparação menor que
		>	Comparação maior que

Exemplo de uso:

MeuPrimeiroPrograma.java

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

// Classe Principal
public class MeuPrimeiroPrograma {

// 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);
   }
}
```

4. 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. Em especial, estas estruturas são idênticas àquelas encontradas na linguagem C/C++.

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.

4.1. 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.

4.2. Instruções de Repetição

Como o C e o C++, 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); // Não se importe com a função disso!
int dado = -1;

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

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); // Não se importe com a função disso!

do {
System.out.println ("Digite o número zero!");
dado = input.nextInt(); // Lê um dado do teclado
} 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.

5. TIPOS NÃO NATIVOS (Classes Wrappers)

Tipos de dados não nativos são tipos de dados que são criados por meio de classes. A utilidade disso é criar tipos de dados que além de armazenar informação, também sejam capazes de realizar tarefas específicas com essa informação.

Vejamos o caso dos textos. Em C é bastante enfadonho trabalhar com textos. Devemos declarar um vetor de caracteres:

```
char texto[30] = "Meu texto";
```

E se quisermos saber quantas letras possui esse texto, precisamos usar uma função do C:

```
int tamanho = strlen(texto);
```

Em Java esse processo é mais simples. Vejamos como declarar uma variável texto:

```
String texto = "Meu texto";
```

Note que não é preciso se preocupar com o tamanho máximo do texto. Se eu quiser saber quantas letras tem esse texto, basta chamar a função **length** do próprio texto:

```
int tamanho = texto.length();
```

A classe String "encapsula" todas as complexidades envolvidas ao lidar com textos e nos fornece várias facilidades. Por não fazer parte da **linguagem Java**, mas sim da biblioteca Java, chamamos estes tipos de dados de "tipos não nativos".

Alguns outros tipos de dados não nativos foram criados, para representar os tipos nativos:

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

Observe que eles se inicial **todos** com letra maiúscula: todos os nomes de classe em Java devem ser grafados com a primeira letra maiúscula.

O uso é idêntico ao visto anteriormente:

Byte umByte = 100;

ou

Integer i = 5;

A vantagem com relação aos tipos nativos, é que estes tipos não nativos são capazes de, por exemplo, converter uma String para um número.

6. REGRAS DE NOMENCLATURA

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.

7. 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.

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

Exemplos de objetos do mundo real:

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

7.2. Classes x Objetos

É importante, neste momento, diferenciar uma classe de 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");
```

7.3. Chamando Métodos

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]);
```

7.4. 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.

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.

HERANÇA

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

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

Exemplo:

classe: Pessoa ação: dirige **classe: Veículo**
objeto: joao
objeto: carla

classe: Veículo
subclasse: Carro (é um Veículo)
objeto: carroAzul
subclasse: Caminhonete (é um Veículo)
objeto: caminhoneteVermelha

Se objetos da classe Pessoa conhecem a interface da classe Veículo, conhecem também a interface 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.

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.

8. 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*.

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.

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.

9. 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** - 6a ed. São Paulo: Pearson, 2005.

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

Unidade 2: Introdução à Tecnologia Servlets

Como Construir um Servlet Básico
Prof. Daniel Caetano

Objetivo: Preparar o aluno para construir Servlets para compor aplicações Web.
Bibliografia: QIAN, 2007; DEITEL, 2005.

INTRODUÇÃO

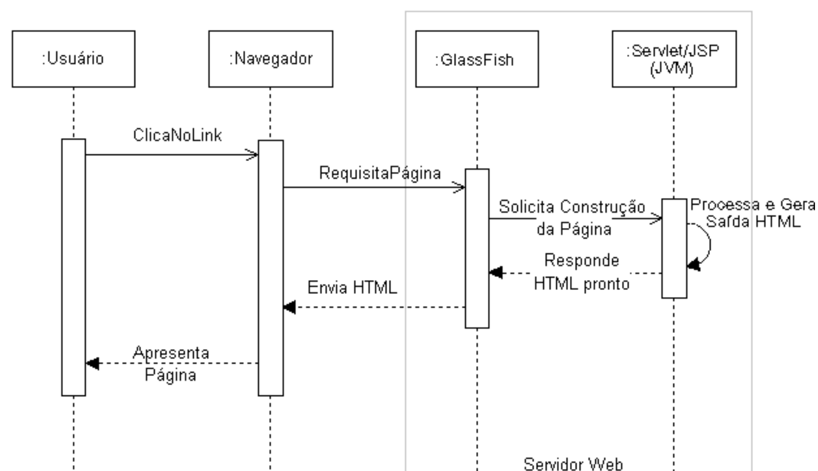
Como este curso tem o objetivo de apresentar brevemente a linguagem Java em um contexto de desenvolvimento para a Web, faz-se necessário estudar um pouco dos componentes que tornam essa apresentação possível.

Neste curso iremos utilizar um ambiente baseado no servidor de aplicações GlassFish, que fornece recursos similares ao uso combinado do TomCat com o JBoss, isto é, combina um servidor de páginas Web básico com os recursos avançados do Java Enterprise Edition 5 e 6 (JEE5 e JEE6). Nesta aula serão apresentados:

1) O que é um servlet, como ele funciona e seu ciclo de vida
2) Configurando uma Web Application com o NetBeans
3) Construindo uma Servlet com o NetBeans
4) Adicionando código à Servlet

1. O QUE É UM SERVLET

- O que é um Servlet.
- Como o Servlet é acionado:



- Ciclo de Vida de um Servlet:
    Invocação => init() => service() => destroy()
- O Init pode ocorrer logo que o servidor de aplicações se inicia...
    - Ou apenas quando o Servlet for executado a primeira vez!
- O Servlet fica a maior parte do tempo em modo "service"

2. CONFIGFURANDO UMA WEB APPLICATION

- Clicar em Criar Projeto
- Selecionar "Java Web" e "Aplicação Web"
- Clicar em Próximo.
- Dar nome ao projeto "WProjeto1"
- Clicar em Próximo.
- Clicar em Finalizar.

- web.xml : arquivo de configuração do GlassFish
    - Configuração pela Interface
    - Configuração pelo XML

3. CONSTRUINDO UMA SERVLET

<title>Cálculo de Índice de Massa Corporal</title>
<h1>Digite seus dados:</h1>
- Construir formulário HTML com:
    - action para "Imc" método "post"
    - peso e altura, dentro de tabela
    - botão submit
- Executar: erro => não existe o servlet!

- Clicar com botão direito em "Pacotes de Código Fonte"
- Selecionar **Novo > Pacote Java**
- Criar pacote com nome **imc**, clicando depois em Finalizar.
- Clicar com botão direito no pacote "imc".
- Selecionar **Novo > Servlet** (se não existir, procure na opção "Outros...")
- Dê o nome **Imc** para o servlet e clique em **Próximo**.
- Marque "**Adicionar informação ao descritor de implementação (web.xml)**" e, depois, clique em **Finalizar**.

- web.xml : arquivo de configuração das servlets disponíveis
    - Aba "servlets" permite configurar descrições e ordem de inicialização
    - Inicializar na criação do GlassFish / Na Execução
    - Edição direto no XML

- Executar: funciona => mas não faz nada (não programamos!)
- Mudar o conteúdo do texto de saída da Servlet para:

    <title>Cálculo de Índice de Massa Corporal: Resultados</title>
    <h1>Índice de Massa Corporal: </h1>

- Adicionar atributo "double imc = 18.0;"
- Adicionar a impressão do atributo no texto de resultado.

4. ADICIONANDO CÓDIGO À SERVLET

- Dentro do processRequest:
    - Pegar os parâmetros com
        String pesoT = request.getParameter("peso");
        String alturaT = request.getParameter("altura");
    - Formato String x formato numérico: conversão
        double peso = Double.valueOf(pesoT);
        double altura = Double.valueOf(alturaT);

    - Calcular imc com:
        imc = peso / (altura * altura);

- Executar: funciona! => Erro em números com vírgula!

- Corrigir conversão:
    pesoT = pesoT.replaceAll(",",".");

- Impressão do número está ruim! Formatar saída:
    out.printf com uso do parâmetro **%3.1f**

- Acrescentar **if**s para mostrar as descrições:

imc < 18.5	=> Cuidado! Você está muito abaixo do peso!
18.5 <= imc < 25.0	=> Parabéns! Você está em seu peso ideal!
25.0 <= imc < 30.0	=> Atenção! Você está acima de seu peso ideal!
30.0 <= imc < 35.0	=> Atenção! Obesidade grau 1!
35.0 <= imc < 40.0	=> Cuidado! Obesidade grau 2!
40.0 <= imc	=> Cuidado! Obesidade grau 3!

5. BIBLIOGRAFIA

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

QIAN, K; ALLEN, R; GAN, M; BROWN, R. **Desenvolvimento Web Java**. Rio de Janeiro: LTC, 2007.

Unidade 3: Recursos Adicionais dos Servlets

Como Construir um com Duas Partes

Prof. Daniel Caetano

Objetivo: Preparar o aluno para construir Servlets para compor aplicações Web.

Bibliografia: QIAN, 2007; DEITEL, 2005.

INTRODUÇÃO

Nas aulas anteriores foram apresentados os conceitos da linguagem Java e também como construir um Servlet simples. Nesta aula apresentaremos mais algumas das capacidades de um Servlet, de maneira que seja possível transferir informações de um Servlet a outro.

Nesta aula serão apresentados:

1) Métodos doGet e doPost
2) Redirect e Forward
3) Alterando os dados da requisição

1. PARÂMETROS PARA UM SERVLET

Para colher os dados do usuário, é preciso aprender um pouco mais sobre o protocolo **http** e sobre os métodos das Servlets.

O protocolo HTTP tem duas formas principais de mandar uma requisição:

Requisição do tipo POST: Serve para enviar dados
- Enviadas, normalmente, através de formulários

Requisição do tipo GET: Serve para solicitar dados
- Enviadas, normalmente, através de links ou da barra de endereços

Em ambos os casos é possível indicar parâmetros. Estes parâmetros poderão ser lidos no servlet através do método getParameter:

```
String valorLido = request.getParameter("nomeDoParametro");
```

Cada um dos casos será apresentados a seguir.

1.1. Parâmetros POST

Os parâmetros POST são passados por meio de formulários XHTML. O nome do **servlet** é indicado através do parâmetro "action" da tag "form".

Cada campo de entrada - <**input** />, por exemplo - possui um parâmetro chamado "name". Tudo que o usuário digita no campo fica associado àquele nome de parâmetro. Por exemplo:

```
<form action="Imc" method="post">
        <p>Peso: <input type="text" name="peso" />kg</p>
        <p>Altura: <input type="text" name="altura" />m</p>
        <p><input type="submit" value="enviar"></p>
</form>
```

Neste caso, o valor digitado no campo "peso" vai estar associado ao parâmetro "peso" e o valor digitado no campo "altura" vai estar associado ao parâmetro "altura".

1.2. Parâmetros GET

Parâmetros GET são passados pela URL, similar ao que acontece quando passamos parâmetros para um programa pela linha de comando do Windows, Linux ou DOS.

Por exemplo, considere o endereço abaixo, que indica o caminho de execução de um servlet chamado **Imc**:

http://localhost:8080/Imc

... temos que ***Imc* é o nome do servlet** e, para passar parâmetros para esse programa, podemos usar o caractere "**?**" para separar o que é nome do programa do que são os parâmetros.

Por exemplo, se quisermos definir que um parâmetro chamado "**peso**" seja igual a **70**, basta indicar da seguinte forma:

http://localhost:8080/Imc?peso=70

Mas... agora uma outra dúvida pode surgir: e se quisermos passar ***mais de um*** parâmetro? Por exemplo, e se quisermos indicar, além do parâmetro **peso**=**70**, quisermos indicar também o parâmetro **altura**=**1.70**?

Existe uma solução simples e elegante para isso: iremos indicar todos os parâmetros separados pelo símbolo "**&**", como apresentado abaixo:

http://localhost:8080/Imc?peso=70&altura=1.70

2. REDIRECT E FORWARD

Quando um servlet é executado, eventualmente podemos querer redirecionar o navegador ou a execução para um outro endereço ou servlet. Neste caso, usaremos os comandos **redirect** ou **forward**.

2.1. Redirect

Eventualmente um Servlet pode querer redirecionar o usuário para uma outra página / JSP ou mesmo outro Servlet. Isso pode ser conseguido com a seguinte instrução:

response.sendRedirect("url")

Por exemplo, para redirecionar para a página do google, usa-se:

response.sendRedirect("http://www.google.com/");

2.2. Forward

Algumas vezes, um serviço pode ser executado por uma combinação de Servlets. Por exemplo: já existe um servlet que calcula um determinado valor em função do tamanho de um armazém. Este Servlet, porém, exige os dados de entrada em "metros" e, segundo os requisitos do cliente, é necessário que os dados de entrada estejam em "pés". Assim, podemos construir um servlet intermediário, que recebe os dados em "pés", converte para "metros" e, então, aciona o outro servlet já existente.

Para acionar um outro servlet repassando os dados da requisição, usa-se a seguinte sequência:

```
// Pega o "despachador de requisições" para a url desejada
RequestDispatcher rd = request.getRequestDispatcher("url");
// Envia dados de requisição e resposta
rd.forward(request,response);
```

Lembrando que os dados da requisição podem ser alterados antes que esse despacho seja feito.

3. ALTERANDO OS DADOS DE UMA REQUISIÇÃO

Quando encaminhamos uma requisição para outro servlet, com o comando **forward**, é comum queremos inserir alguns dados na requisição.

Esses dados ganham o nome de atributos e **sempre** devem ser OBJETOS. Por exemplo: para colocar uma String na requisição, fazemos o seguinte:

```
Sting nome = "Fulano de Tal";
request.setAttribute("cliente", nome);
```

Para guardar um número, fazemos o seguinte:

```
Integer anos = 50;
request.setAttribute("idade", anos)/
```

Isso irá armazenar os dados na requisição com os nomes de "cliente" e "idade", respectivamente.

Quando essa requisição for repassada a outro servlet, o novo servlet poderá obter os dados da seguinte forma:

```
String nome = (String) request.getAttribute("cliente");
Integer idade = (Integer) request.getAttribute("idade");
```

4. BIBLIOGRAFIA

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

QIAN, K; ALLEN, R; GAN, M; BROWN, R. **Desenvolvimento Web Java**. Rio de Janeiro: LTC, 2007.

Unidade 4: Java Server Pages

Prof. Daniel Caetano

Objetivo: Capacitar o aluno para produzir páginas usando a tecnologia JSP.

INTRODUÇÃO

Como foi observado na aula anterior, podemos dividir a tarefa de nossos Servlets entre aqueles que processam e aqueles que apresentam resultados, chamados respectivamente de Servlets de Processamento e Servlets de Apresentação.

Embora os Servlets desempenhem com ótimo resultado ambos os papéis, a construção de Servlets de Apresentação é um tanto desajeitada. Isso ocorre porque os Servlets de Apresentação têm a tarefa primordial de construir uma página web, em html, e a tecnologia Servlets em si não traz nenhuma facilidade para isso, tornando o processo tedioso e bastante propenso a erros.

Para mitigar este problema, a Sun Microsystems/Oracle criou a tecnologia **Java Server Pages** (ou, em bom português, Páginas em Servidor Java) - **JSP**, para os íntimos. Nesta aula veremos os fundamentos da tecnologia JSP e também transformaremos os servlets de aulas anteriores em JSPs.

1. O QUE É UM JSP?

Versão curta: se um servlet é um programa em Java com um pouco de script HTML...
... um JSP é um script HTML com um pouco de programa em Java.

Versão longa: JSP é uma forma alternativa de escrever um Servlet que facilita a vida do desenvolvedor quando a função principal do servlet é gerar uma página ou um formulário HTML. Uma forma alternativa de escrever um Servlet? Como assim?

Antes de mais nada, vamos ver o que é, qual a aparência, de uma página em JSP.

Uma página JSP tem a seguinte "cara":

index.jsp

```
<%--
    Document   : index
    Created on : 23/08/2011, 10:38:32
    Author     : djcaetano
--%>

<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Teste</title>
    </head>
    <body>
        <h1>Olá mundo!</h1>
    </body>
</html>
```

Um desenvolvedor desavisado tomaria isso por um código HTML mas... observe que existem algumas tags estranhas, começando com **<%** e finalizando com **%>**. Essas tags servem para indicar um código JSP.

Apesar da aparência, entretanto, não se engane: estamos escrevendo um Servlet! Todo o código que funcionaria em um Servlet vai funcionar se for escrito dentro de <% ... %>!

Por outro lado, o servidor GlassFish ou TomCat **não** é capaz de executar um JSP; sempre que carregarmos um JSP pela primeira vez, o servidor irá transformar o JSP para um Servlet e irá, a partir de então, usar esse "Servlet gerado automaticamente" sempre que o JSP for solicitado.

2. FORMAS DE USAR CÓDIGO JAVA EM JSP

Existem várias formas de usar código Java em JSPs. Não faz parte do escopo do curso apresentar todas elas; contudo, iremos apresentar algumas mais importantes.

Basicamente existem formas para:

a) Inserir um código completo em Java, como feito no Servlet
b) Imprimir o valor de uma variável ou expressão
c) Chamar comandos específicos dos Servlets
d) Declarar variáveis e métodos
e) Diretivas

Cada uma destas formas será apresentada a seguir.

2.1. Inserindo Código Java no JSP (Scriplet Tag)

Primeiramente, vejamos como inserir código Java tradicional. Esse é o mais fácil: basta usar a tag <% ... %> para inserir código em qualquer lugar da página:

```
<%--
    Document   : index
    Created on : 04/03/2011, 10:13:15
    Author     : djcaetano
--%>

<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>

<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Teste</title>
    </head>
    <body>
        <h1>Olá mundo!</h1>
        <%
            int a = 2;
            int b = 3;
            int c = a + b;
            out.println("<p>Resultado: " + c + "</p>");
        %>
    </body>
</html>
```

É importante ressaltar que vários **objetos** já estão pré-definidos nesse modo:

out	Objeto de impressão
request	Objeto de requisição
response	Objeto de resposta
session	Objeto de sessão (será visto posteriormente)

Existem outros dentre outros menos usados.

Qualquer código que quisermos executar em Java, basta colocar aí dentro. A única pegadinha é que, dentro de um scriplet, não podemos usar as palavrinhas "import" para indicar "onde estão as classes que estamos usando" e, então, temos que fazer isso toda vez que as utilizarmos. Por exemplo, para usar a classe **Date**, que está no pacote **java.util**, temos que escrever o seguinte no programa:

```
<%--
    Document   : index
    Created on : 04/03/2011, 10:13:15
    Author     : djcaetano
--%>

<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>

<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Teste</title>
    </head>
    <body>
        <h1>Olá mundo!</h1>
        <%
            java.util.Date data = new java.util.Date();
            out.println("<p>Hoje é: " + data + "</p>");
        %>
    </body>
</html>
```

Mais adiante veremos uma forma mais simples de resolver isso.

2.2. Imprimindo o Valor de uma Variável ou Expressão (Expression Tag)

Em algumas situações, como foi visto no exemplo anterior, queremos apenas imprimir um valor ou resultado, tornando excessivo o uso de tanto código. Para evitar isso, foi criada uma *tag* específica para a impressão de resultados: **<%=** ... **%>**.

Esta *tag* funciona da seguinte forma:

```
<%--
    Document   : index
    Created on : 04/03/2011, 10:13:15
    Author     : djcaetano
--%>

<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>

<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Teste</title>
    </head>
    <body>
        <h1>Olá mundo!</h1>
        <%= new java.util.Date() %>
    </body>
</html>
```

O **<%=** inicial implica que haverá apenas uma instrução e o resultado dela deverá ser impresso. Não é possível inserir múltiplas instruções separando-as por ponto-e-vírgula dentro dessa seção. Na verdade, **não finalize a instrução por ponto-e-vírgula!**

Se quisermos pegar um valor da requisição e imprimí-lo, podemos fazer como é indicado no código a seguir:

```
<%--
    Document   : index
    Created on : 04/03/2011, 10:13:15
    Author     : djcaetano
--%>

<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>

<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Teste</title>
    </head>
    <body>
        <h1>Olá mundo!</h1>
        <%= request.getParameter("nome") %>
    </body>
</html>
```

E se precisarmos fazer algum cálculo complexo antes de imprimir os resultados? Basta realizar os cálculos ANTES!

```
<%--
    Document   : index
    Created on : 04/03/2011, 10:13:15
    Author     : djcaetano
--%>

<%    int a = 10;
      int b = 30;
      int c = (a*b)+a;
      String peso = request.getParameter("peso");
      String texto = "Os resultados são: " + peso + " e " + c;
%>

<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>

<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Teste</title>
    </head>
    <body>
        <h1>Olá mundo!</h1>
        <%= texto %>
    </body>
</html>
```

2.3. Executando Recursos de Servlets Direto do JSP (Action Tag)

Alguns recursos dos Servlets estão acessíveis diretamente a partir dos JSPs, com o uso de tags especiais iniciadas por **jsp:** . Por exemplo, para realizar a transferência da requisição de um JSP para um Servlet (ou mesmo para outro JSP), pode-se usar o **forward** como visto na aula passada... ou sua forma mais simples do JSP! Como é isso?

FORA de uma seção <% ... %> indica-se o forward com:

<jsp:forward page="/NomeDoServlet" />

Exemplo:

```
<%--
    Document   : index
    Created on : 04/03/2011, 10:13:15
    Author     : djcaetano
--%>

<jsp:forward page="outro.jsp" />

<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>

<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Teste</title>
    </head>
    <body>
        <h1>Olá mundo!</h1>
    </body>
</html>
```

NOTA: só funciona com páginas e servlets no próprio servidor!

Um recurso insteressante é incluir um JSP dentro de outro, o que pode ser feito com o comando **include**:

```
<jsp:include page="outro.jsp"  flush="true" />
```

Caso seja necessário definir parâmetros na chamada, usa-se:

```
<jsp:include page="/Outro.jsp"  flush="true"/>
        <jsp:param name="nomeDoParametro" value="valorDoParametro" />
</jsp:include>
```

Observe o exemplo a seguir

index.jsp

```
<%--
    Document   : index
    Created on : 04/03/2011, 10:13:15
    Author     : djcaetano
--%>

<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>

<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Teste</title>
    </head>
    <body>
        <jsp:include page="conteudo.jsp" flush="true" />
    </body>
</html>
```

conteudo.jsp

```
<%--
    Document   : index
    Created on : 04/03/2011, 10:13:15
    Author     : djcaetano
--%>

    <h1>Olá Mundo!</h1>
```

2.4. Declarando Variveis e Métodos (Declaration Tag)

Se tudo que precisamos em um momento é declarar uma variável com um certo valor ou mesmo um método, isso pode ser feito com a tag de declaração: **<%!   ...   %>**

Observe o exemplo:

```
<%--
    Document   : index
    Created on : 04/03/2011, 10:13:15
    Author     : djcaetano
--%>

<%! private int contador = 1345; %>
```

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Teste</title>
    </head>
    <body>
        <h1><%= contador %></h1>
   </body>
</html>
```

2.5. Diretivas (Directive Tag)

Tags do tipo **<%@ ... %>** servem para indicar informações adicionais ao Java, como definir o tipo de codificação a ser usada na geração do código HTML:

<%@page contentType="text/html" pageEncoding="UTF-8" %>

Um atributo importante para a diretiva @page é o **import**, que premite declarar bibliotecas Java adicionais.

<%@page import="java.util.*" %>

```
<%--
    Document   : index
    Created on : 04/03/2011, 10:13:15
    Author     : djcaetano
--%>


<%@page contentType="text/html" pageEncoding="UTF-8" import="java.util.*"%>
<!DOCTYPE html>

<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Teste</title>
    </head>
    <body>
        <h1>Olá mundo!</h1>
        <%
            Date data = new Date();
            out.println("<p>Hoje é: " + data + "</p>");
        %>
    </body>
</html>
```

Uma outra diretiva útil é o **include** (é, existem duas formas de fazer um include!):

<%@include file="pagina.jsp" %>

A diferença desta **@include** "diretiva" para a **jsp:include** "action" é que a **jsp:include** ocorre sempre que o usuário solicitar a página novamente; a **@include** ocorre apenas no momento em que o JSP é convertido para Servlet na primeira vez.

3. ALTERANDO SISTEMAS COM SERVLETS PARA USAR JSPs

Basicamente, o processo é esse:

PASSO 1: Criar um JSP que substitua o Servlet de apresentação
Exemplo: para o servlet **Calcula.java**, crie o **Calcula.jsp**

PASSO 2: Ir até o descritor dos servlets (**Páginas Web > WEB-INF > web.xml**) e vá até a aba **Servlets**

PASSO 3: Procure o nome de seu servlet (por exemplo **Calcula -> /Calcula**) e, desmarque "**Classe do servlet**", ativando a opção "**Arquivo JSP**". Ao lado do "Arquivo JSP", indique o nome do seu JSP. Por exemplo: **Calcula.jsp** .

4. BIBLIOGRAFIA

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

QIAN, K; ALLEN, R; GAN, M; BROWN, R. **Desenvolvimento Web Java.** Rio de Janeiro: LTC, 2010.

Unidade 7: Middleware JDBC e Java DB

Prof. Daniel Caetano (Fonte: Tutorial Oficial do NetBeans)

Objetivo: Capacitar o aluno para criar bancos de dados usando Java DB.

INTRODUÇÃO

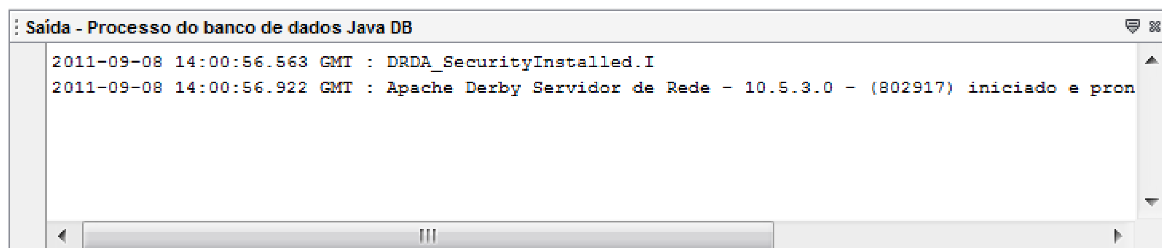
Todo software fica bem mais poderoso se tivermos acesso a um banco de dados. O NetBeans traz consigo um Banco de Dados completo, que iremos utilizar para construir nossas aplicações Java EE. A forma de usar este banco de dados é similar à de usar qualquer outro banco de dados, mas a criação difere um pouco, por usar a interface do NetBeans.

Esta aula apresenta, em específico, um tutorial de criação de bancos de dados com Java DB.

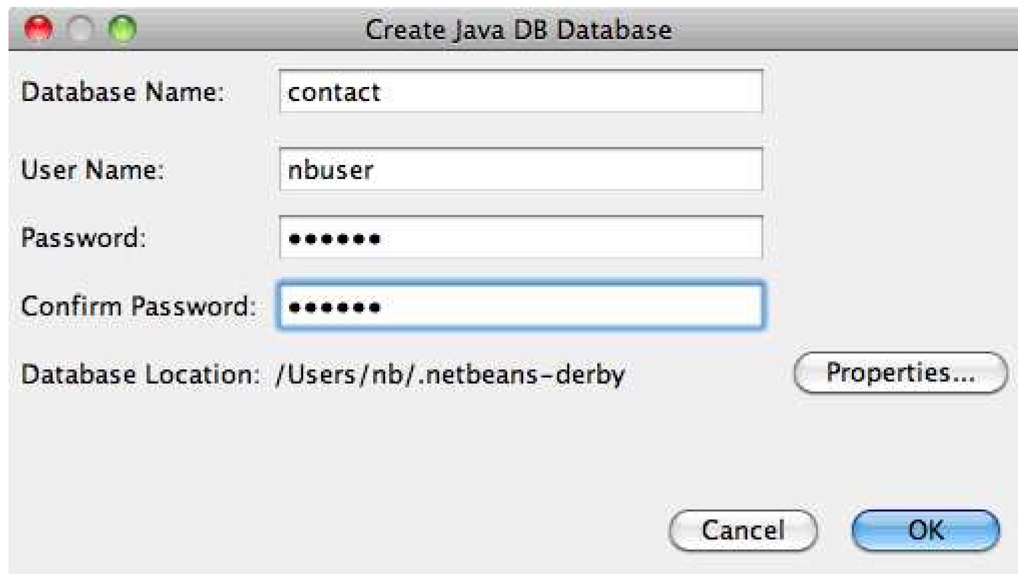
1. INICIANDO O SERVIDOR E CRIANDO UM BANCO DE DADOS

As opções de menu do banco de dados JavaDB são exibidas quando você clica com o botão direito do mouse no nó Java DB na janela Serviços. Os itens desse menu contextual permitem o início e a interrupção do servidor de banco de dados, a criação de uma nova instância de banco de dados e o registro de servidores de bancos de dados no IDE (como demonstrado na etapa anterior). Para iniciar o servidor de banco de dados:

PASSO 1. Na janela Serviços, clique com o botão direito do mouse no nó JavaDB e escolha Iniciar servidor. Observe a saída a seguir na janela de Saída, indicando que o servidor foi iniciado:



PASSO 2: Clique com o botão direito no nó JavaDB e escolha Criar banco de dados. A caixa de diálogo Criar JavaDB se abre.



PASSO 3: No campo de texto Nome do banco de dados, digite `contact`. Também defina o nome de usuário e senha para `nbuser`. Observe que Local do banco de dados é o local padrão definido durante a instalação do Java DB do GlassFish. Se já foi instalado o Java DB separadamente, essa localização deve diferir. Clique em OK.

2. CONEXÃO AO BANCO DE DADOS

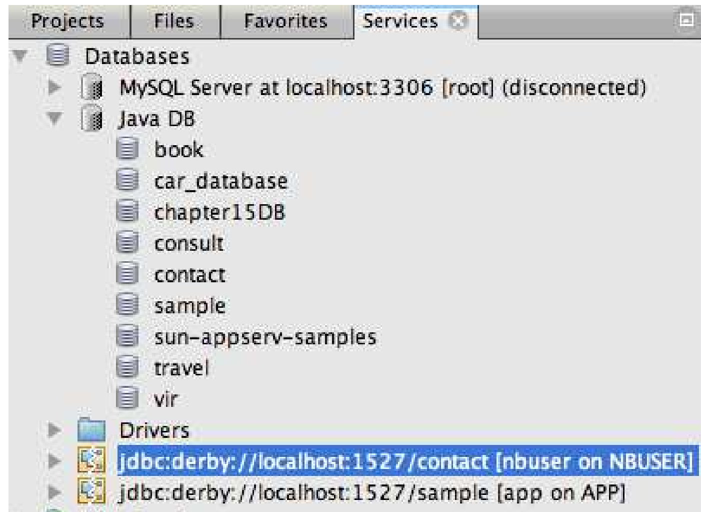
Até agora, você iniciou com êxito o servidor de banco de dados e criou uma instância de banco de dados denominada `contact` no IDE. O banco de dados Explorer do NetBeans IDE, disponível a partir da janela Serviços, fornece funcionalidade para tarefas comuns em estruturas de bancos de dados. Para começar a trabalhar com o banco de dados `contact`, você precisa criar uma conexão com o mesmo. Para se conectar a `contact`:

PASSO 4: Expanda o banco de dados Explorer na janela Serviços e localiza o novo banco de dados.

Observe que `amostra` [app on APP] é o esquema de banco de dados padrão.

PASSO 5: Clique com o botão direito do mouse no nó da conexão do banco de dados (`jdbc:derby://localhost:1527/contact [nbuser em NBUSER]`) e escolha Conectar. O ícone do nó da conexão agora aparece por inteiro ( ), o que significa que a conexão foi bem sucedida.

PASSO 6: Crie um nome de exibição conveniente para o banco de dados. Clique com o botão direito do mouse no nó da conexão do banco de dados (`jdbc:derby://localhost:1527/contact [nbuser em NBUSER]`) e escolha Propriedades.



PASSO 7: Clique no botão elipse (...) próximo ao nome de exibição e insira `Contato DB` no campo de texto. Agora o banco de dados tem um nome de exibição mais conveniente no IDE.

3. CRIANDO TABELAS

O banco de dados `contact` recém-criado está vazio no momento. Ele não contém ainda tabelas ou dados. No NetBeans IDE, você pode adicionar uma tabela de banco de dados usando a caixa de diálogo Criar tabela ou inserindo uma instrução SQL e executando-a diretamente do Editor SQL.

PASSO 8: Expanda o nó de conexão `contact` e observe que existem vários esquemas de subnós. O esquema app é o único esquema que se aplica a este tutorial. Clique com o botão direito do mouse no nó APP e escolha Definir como esquema padrão.

PASSO 9: Expanda o nó APP e observe que existem três subpastas: Tabelas, Visualização e Procedimentos. Clique com o botão direito do mouse no nó Tabelas e escolha Criar tabela. A caixa de diálogo Criar tabela é aberta.

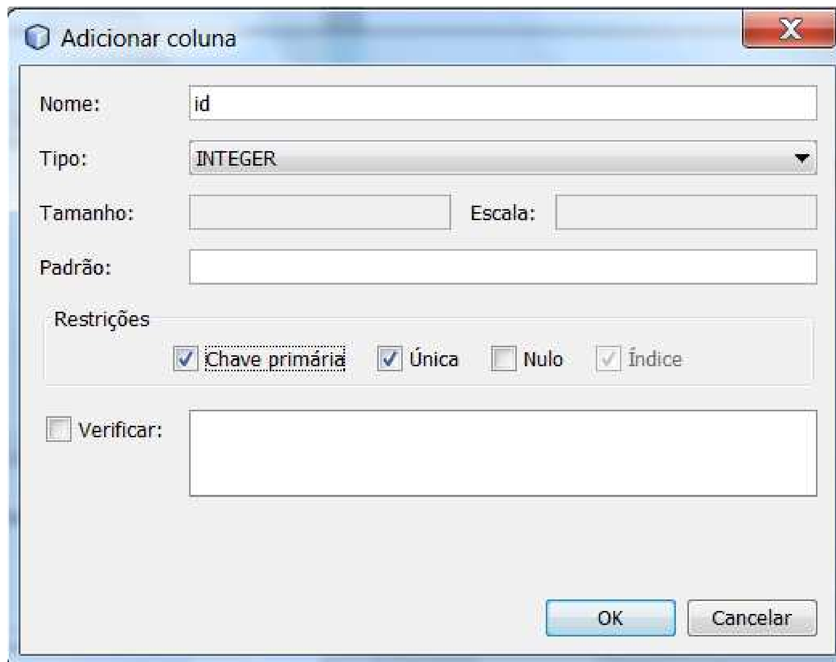
PASSO 10: No campo de texto Nome da tabela, digite `AMIGOS`.

PASSO 11: Clique em Adicionar coluna. A caixa de diálogo Adicionar coluna aparece.

PASSO 12: Para Nome de coluna, insira `id`. Para Tipo de dados, selecione `INTEIRO` da lista suspensa.

PASSO 13: Em Restrições, selecione caixa de verificação Chave primária para especificar que essa coluna é a chave primária para a tabela. Todas as tabelas de bancos de

dados relacionais devem conter uma chave primária. Observe que quando você marca a caixa de verificação Chave primária, as caixas de verificação Índice e Exclusivo são automaticamente marcadas e a caixa de verificação Nulo é desmarcada. Isso ocorre porque as chaves primárias são usadas para identificar uma linha exclusiva no banco de dados e por padrão são usadas como o índice da tabela. Como todas as linhas devem ser identificadas, as chaves primárias não podem conter um valor `Nulo`.

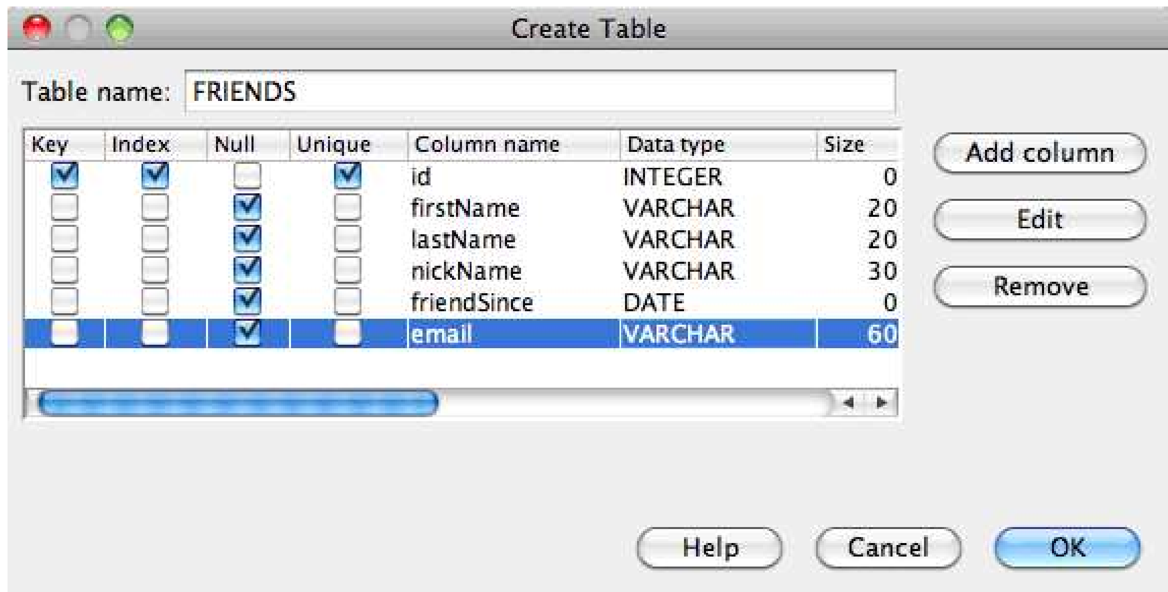


PASSO 14: Repita este procedimento agora especificando campos conforme exibido na tabela abaixo:

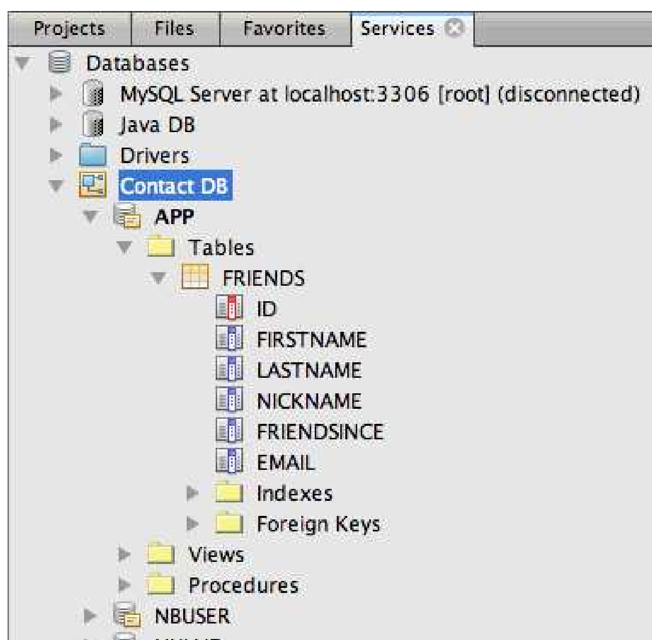
Chave	Índice	Nulo	Exclusiva	Nome da coluna	Tipo de dados	Tamanho
[marcada]	[marcada]		[marcada]	id	INTEIRO	0
		[marcada]		firstName	VARCHAR	20
		[marcada]		lastName	VARCHAR	20
		[marcada]		nickName	VARCHAR	30
		[marcada]		friendSince	DATA	0
		[marcada]		email	VARCHAR	60

PASSO 15: Você está criando uma tabela chamada `AMIGOS` que contém os seguintes dados para cada registro de contato:

- **Nome**
- **Sobrenome**
- **Apelido**
- **Amigo desde**
- **Endereço de e-mail**



PASSO 16: Quando tiver certeza que a caixa de diálogo Criar tabela contém as mesmas especificações que as exibidas acima, clique em OK. O IDE gera a tabela `AMIGOS` no banco de dados e você pode ver o nó da nova tabela `AMIGOS` ( ) exibido em Tabelas no banco de dados Explorer. Abaixo do nó da tabela, as colunas (campos) são listadas, começando pela chave primária.



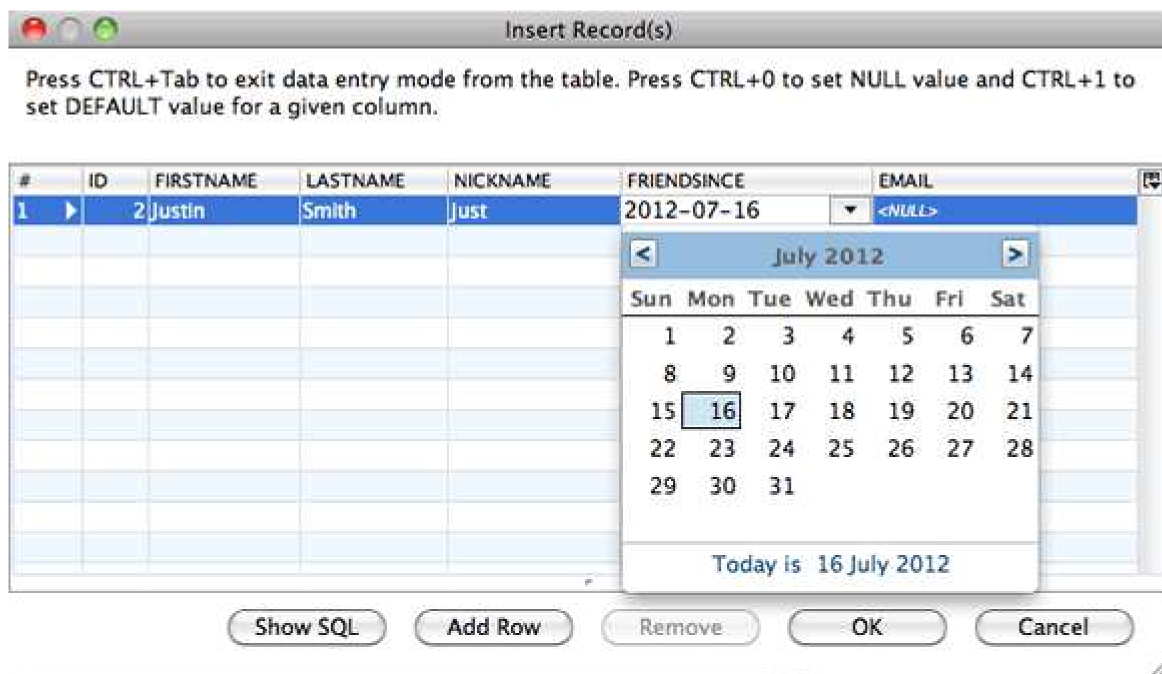
4. ADIÇÃO DE DADOS NA TABELA

Agora que você criou uma ou mais tabelas no banco de dados `contact`, pode começar a preenchê-lo com dados.

PASSO 17: Clique com o botão direito no nó da tabela `AMIGOS` e escolha Visualizar dados (se não tiver feito isso na última etapa da seção anterior).

PASSO 18: Clique no botão Inserir registro(s) (`Alt-I`) para adicionar uma linha. A caixa de diálogo Inserir registros aparece.

PASSO 19: Clique em cada célula e insira registros. Observe que para células com tipo de dados Data, é possível escolher uma data do calendário. Clique em OK quando tiver Acabado.



No editor SQL, é possível ordenar os resultados clicando na linha de cabeçalho, modifique e exclua registros existentes e veja o script SQL para ações que estão sendo feitas no editor (o comando Exibir script SQL do menu pop-up).

5. EXCLUINDO TABELAS

Para excluir uma tabela de banco de dados:

PASSO 20: Clique com o botão direito do mouse no nó do banco de dados Explorer e escolha Excluir. Observe que o nó da tabela é imediatamente removido do Explorer do banco de dados sem confirmação.

Unidade 6: Padrão MVC e DAO

Prof. Daniel Caetano

Objetivo: Apresentar a teoria por trás dos padrões na construção de aplicações Web.

INTRODUÇÃO

Nas aulas anteriores apresentamos a idéia de separar as aplicações em pelo menos dois componentes: o de apresentação e o de processamento. Seguindo esse princípio, foram construídas pequenas Aplicações Web compostas por um Servlet de processamento e um Servlet (JSP) de apresentação.

Essa divisão não é aleatória: ela foi feita segundo algumas regras específicas. Mas que regras são essas? Essas são as únicas separações que faremos em um projeto? Quais são as vantagens? E as desvantagens?

O objetivo desta aula é apresentar, de forma explícita e simplificada, alguns dos mais importantes padrões seguidos em Aplicações Web, independentemente de qual seja a linguagem de programação utilizada.

1. O QUE É UM PADRÃO DE DESENVOLVIMENTO

Sempre que estamos desenvolvendo um sistema, um dos primeiros passos é dividí-lo em partes cada vez menores, até que tenhamos partes pequenas e simples o suficiente para que possamos implementá-las.

Cada solução que um programador dá para cada problema simples - como, por exemplo, escolher o tipo de dado que será usado para armazenar um CPF - tem consequências que podem tornar a manutenção de um sistema desde extremamente simples até algo próximo de impossível.

Assim, é possviel dizer que a forma com que dividimos um sistema, assim como a forma com que implementamos cada uma das pequenas tarefas, tem consequências diretas sobre o nosso sistema, como por exemplo:

a) Sua manutibilidade (flexibilidade e extensibilidade);
b) Seus custos;
c) Reusabilidade de suas partes.

No entanto, grande parte das tarefas que temos de executar em um sistema tradicional já foram exaustivamente implementadas ao longo dos últimos 50 anos, de maneira que, para diversas dessas tarefas, já foram definidas **"melhores práticas"**. Isso significa que, para um grande número de casos, a melhor forma de implementar já é conhecida e documentada.

Essas "melhores práticas" de implementação são conhecidas como **padrões de desenvolvimento**.

2. PADRÃO DE PROJETO

Entre os padrões de desenvolvimento, existem aqueles que são denominados padrões de projeto, que são relacionados à melhor maneira de dividir e organizar um sistema, de maneira a solucionar problemas comuns de maneira "elegante", isto é, permitindo grande facilidade de manutenção e estensão, além de facilitar o reuso de cada uma dessas partes.

Os padrões de projeto existem em diferentes níveis de detalhamento. Alguns são extremamente específicos (como implementar a interação entre um sistema novo e um antigo), mas existem também os padrões mais genéricos, que estabelecem apenas alguns critérios a serem seguidos no projeto.

Nesta aula veremos um de cada tipo, lembrando que estes padrões não são soluções definitivas e imutáveis. São apenas diretrizes que, se bem aplicadas, trazem benefícios para o desenvolvimento.

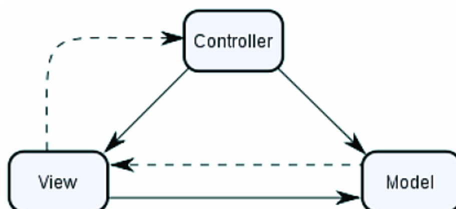
3. PADRÃO MVC

O padrão MVC (*Model-View-Controller* ou, em português, Modelo-Visão-Controle) é um padrão mais macroscópico, digamos assim, do tipo que fornece apenas algumas diretrizes para o projeto. Este tipo de modelo é chamado de **modelo de arquitetura**.

Em essência, o padrão MVC indica que os componentes (servlets, por exemplo) de um sistema devem ser divididos em três categorias distintas:

Modelo: composto pelos componentes de entidade e persistência, esta categoria representa os componentes que representam a modelagem de dados.
Visão: composto pelos componentes de apresentação (janelas, formulários etc.), esta categoria representa os componentes que interagem com o usuário, seja para receber informações, seja para fornecer informações.
Controle: composto pelos componentes de processamento, esta categoria representa os componentes que controlam os processos de negócio, coordenando os outros componentes do sistema para que o resultado final seja aquele esperado pelo usuário.

A relação entre esses componentes é apresentada na figura a seguir:



Nessa representação, as setas "cheias" representam relação direta (direção das requisições) e as setas "tracejadas" indicam relação indireta (direção das respostas).

Pelo diagrama, percebe-se que os componentes do tipo Controle fazem requisições tanto para componetes do tipo Modelo quanto do tipo Visão. Os componentes do tipo Visão, por sua vez, fazem requisições apenas aos componentes do tipo Modelo. Por fim, os componentes do tipo Modelo são passivos, não dando início a requisições para componentes do tipo Controle e Visão.

A razão para essa separação é simples:

a) As mudanças mais frequentes estão na interface com o usuário (Visão) e, normalmente, não exigem mudanças no processo de negócio (componentes de Controle) e na modelagem de dados (componentes de Modelo).
b) Mudanças menores na modelagem de dados (componentes de Modelo) não devem exigir modificações no restante do sistema (componentes de Visão e Controle).
c) Algumas modificações no processo de negócio (componentes de Controle) devem ser possíveis sem exigir alterações nos demais componentes do sistema (Visão e Modelo).

De fato, a parte mais mutante é a interface com o usuário, já que é comum que a empresa queira modificá-la, de tempos em tempos, para ter um "visual mais moderno". A modelagem de dados é modificada com frequência bem menor e, em geral, em razão de modificações na estrutura interna do banco de dados ou para acrescentar informações que nem sempre são visíveis ao usuário, para a geração de estatísticas. Finalmente, o processo de negócio só muda quando também muda a empresa para a qual o sistema foi produzido o sistema. Por exemplo: era uma empresa de vendas e, agora, passa a fazer também locações.

Assim, a separação facilita a manutenção por centralizar algumas dessas mudanças em partes específicas do sistema, não existindo a chance para que alguém "modificando a interface" quebre o funcionamento do processo de negócio.

Mas essas não são as únicas vantagens: a separação em blocos desse tipo permitem uma maior possibilidade de reuso dos componentes desenvolvidos. Ainda que a interface (Visão) de um sistema raramente possa ser aproveitada em outro - ao menos não sem grandes modificações -, os componentes de Controle e, em especial, os de Modelo normalmente podem ser "transplantados" de um sistema para outro praticamente sem alterações significativas, preservando muito o investimento de desenvolvimento.

3.1. O Padrão MVC para a Web

É importante observar, porém, que o este modelo de arquitetura não foi definido para a Web, tendo sido criado na década de 1970, para aplicações desktop, tradicionais, com janelas. Como há diferenças fundamentais entre estes dois tipos de aplicação (desktop x web), não é possível seguí-lo à risca em aplicações Web, mas é possível preservar as linhas gerais e, com isso, preservar grande parte de seus benefícios.

Quando aplicado à Web, os componentes de visão são associados às páginas HTML e seus formulários; os componentes de Controle são associados aos servlets de processamento e, finalmente, os componentes de Modelo são classes de Entidade, que representam os dados (como o Produto e o Livro que foram desenvolvidos nas primeiras aulas do curso), bem como a comunicação com o banco de dados.

NOTA: para maximizar o reaproveitamento, os servlets de processamento podem ser definidos de maneira a apenas repassar tarefas para classes Java de Controle. Por simplicidade, neste curso optamos por integrar a parte de controle aos servlets de processamento.

Como você deve ter percebido, nas aulas anteriores de Banco de Dados não seguimos muito esta especificação: o mesmo Servlet processa, acessa o banco de daods e imprime os resultados. Vamos ver agora como podemos organizar melhor nossa aplicação.

4. PERSISTÊNCIA

Em um sistema orientado a objetos, incluindo aqueles desenvolvidos segundo o MVC, não existe uma preocupação específica com o armazenamento dos dados em um banco de dados. Isso ocorre porque, originalmente, o modelo orientado a objetos prevê um sistema de **memória persistente**, isto é, um sistema em que a memória não se apaga quando o equipamento é desligado.

Atualmente, este esquema de memória persistente só está disponível para algumas linguagens (como SmallTalk) e em algumas categorias de equipamentos (como grande parte dos celulares e alguns palm-tops). Nestes casos o uso de um banco de dados não é apenas inútil, como também totalmente indesejável.

Por outro lado, quando desenvolvemos sistemas para computadores de memóra principal volátil - como o PC -, o que inclui as aplicações Web, é preciso **"simular" a existência de uma memória persistente**, tarefa essa que fica delegada para os **componentes de persistência**.

Simplificadamente, o papel do componente de persistência é, de maneira transparente (ou quase) para o sistema, armazenar e recuperar objetos de entidade em um banco de dados, de maneira que nenhum outro componente do sistema precise entrar em contato direto com o banco de dados. Existem diversos padrões que permitem esse resultado.

4.1. MVC Nível 1 e 2

Uma das primeiras idéias que se tem ao pensar em persistência é a seguinte:

"Ora, se a persistência serve para armazenar e recuperar objetos de entidade de um banco de dados, que tal inserir esse código dentro do próprio objeto de entidade?"

A afirmação faz todo o sentido do mundo e, de fato, essa é uma das formas de fazê-lo; na verdade, foi a primeira forma de fazer a coisa, que ficou conhecida como MVC Nível 1.

Na próxima aula iremos implementar uma classe de entidade seguindo o MVC Nível 1 e verificar como esse tipo de implementação funciona. Entretanto, este modelo leva a

algumas dificuldades, quando se manipula objetos complexos; em especial, quando se quer garantir que apenas uma cópia do objeto permanece na memória.

Para resolver esses problemas, foi proposto o MVC Nível 2, em que a manipulação do Banco de Dados é feito por uma classe externa, usualmente seguindo o padrão DAO.

4.2. O Padrão DAO

O Padrão DAO estabelece alguns critérios para que sejam criadas classes específicas para o acesso ao banco de dados. Este tipo e implementação permite a solução do problema de duplicidade de objetos na memória, e tem uma vantagem adicional: é fácil remover ou substituir o suporte ao banco de dados.

Por que iríamos desejar remover o suporte ao banco de dados? Bem, cada vez mais, estão disponíveis dispositivos de memória persistente no mercado e, sendo assim, para garantir uma boa capacidade de reuso de código, **é interessante que seja fácil remover o suporte ao banco de dados**, para que nosso sistema seja facilmente adaptado para ambientes naturalmente persistentes.

A forma mais simples de obter esse resultado é isolando a parte de persistência de uma classe de entidade em uma outra classe específica para este fim. Por exemplo: nosso sistema tem a classe **Cliente** e, para as tarefas de persistência, foi criada a classe **ClienteDAO**. A nomenclatura deste jeito não é obrigatória, mas ela facilita a identificação das classes de entidade e sua correspondente classe de persistência.

Nesse caso, a classe ClienteDAO é uma classe de serviço, isto é, ela serve apenas para agregar alguns métodos. Desta forma, não é necessário criar um objeto desta classe para que se possa executar seus serviços. Adicionar um objeto ao banco de dados torna-se simplesmente uma questão de comandar:

```
ClienteDAO.adiciona( nome_do_objeto_do_cliente );
```

Para cada classe de entidade teremos uma classe DAO correspondente. Se houver uma classe de entidade Pedido, teremos uma classe PedidoDAO. Se houver uma classe de entidade Produto, teremos uma classe ProdutoDAO... e assim por diante.

4.3. Outros Padrões de Persistência

O Java EE, em suas versões mais recentes, apresenta uma interface própria de persistência, denominada JPA (*Java Persistence API*). A JPA **não é uma biblioteca**, mas sim uma interface padronizada que pode ser adotada por aqueles que quiserem implementar um framework de persistência. A idéia é que, se diversos frameworks implementarem a JPA, o uso de todos eles será bastante similar. O JPA é baseado na arquitetura do tipo MVC Nível 1 e é usada por diversos frameworks.

Um dos frameworks de persistência mais comuns e conhecidos para Java é **Hibernate**. O Hibernate é integrado ao Java e proporciona uma série de facilidades para a

implementação de persistência em um sistema; por outro lado, o Hibernate exige uma configuração específica para seu funcionamento. O Hibernate padrão, portanto, não segue a JPA; entretanto, foi criada uma versão denominada **Hibernate JPA**, que permite a manipulação do Hibernate seguindo o padrão JPA.

Um outro framework de persistência que implementa o JPA é o TopLink, fornecido pela própria Oracle, junto com o NetBeans. Mais para o final do curso veremos como trabalhar com o TopLink e aprenderemos um pouco mais sobre o JPA.

5. BIBLIOGRAFIA

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

Unidade 7: Middleware JDBC para Criação de Beans

Implementando MVC Nível 1

Prof. Daniel Caetano

Objetivo: Preparar o aluno para construir classes de entidade com capacidade de persistência.

Bibliografia: QIAN, 2007; DEITEL, 2005.

INTRODUÇÃO

Até o momento criamos diversos Servlets, mas sempre trabalhando com tipos nativos ou tipos não nativos pré-existentes no Java. Se quisermos criar uma aplicação mais útil, entretanto, seremos "obrigados" a criar nossos próprios tipos não nativos.

Vimos como criar tipos não nativos logo no início do curso, mas aqueles tipos não nativos tinham uma deficiência do ponto de vista de um sistema prático real: de que adiante criar um objeto de livro ou cliente se, ao desligar o computador, todos os dados estarão perdidos?

Obviamente nenhum sistema real funciona que seja executado em um PC desta forma; sendo assim, iremos verificar nessa aula como implementar uma classe de entidade capaz de **persistir**, isto é, de fazer com que seus dados "permaneçam" mesmo que o equipamento seja desligado e, para isso, usaremos o middleware JDBC.

1. CRIANDO A WEB APPLICATION

A nossa aplicação vai conter vários elementos, incluindo servlets e JSPs. Para criá-la, façamos o seguinte:

- Clicar em Criar Projeto
- Selecionar "Java Web" e "Aplicação Web"
- Clicar em Próximo.
- Dar nome ao projeto "WProjeto5"
- Clicar em Próximo.
- Selecione o uso do GlassFish, dependendo da sua instalação
- Clicar em Finalizar.

2. CRIANDO O BEAN CLIENTE

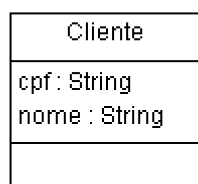
Agora que a aplicação está configurada, vamos criar a nossa classe de entidade, o nosso **bean** Cliente.

- Clique com botão direito em "Pacotes de Código Fonte"
- Selecionar **Novo > Pacote Java**
- Criar pacote com nome **entidades**, clicando depois em Finalizar.

O objeto de entidade Cliente é aquele que armazenará todos os dados de nosso cliente. Para construí-lo, comecemos assim:

PASSO 1: Clique com o botão direito no **pacote entidade**s 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 2: Iremos agora configurar a classe para que tenha 2 atributos: **cpf** e **nome**, conforme indicado no diagrama abaixo:



Todos estes atributos serão privados. Assim, devemos inserir as seguintes linhas na classe Cliente:

Cliente.java

```
package wsiscli;

public class Cliente {

    // Atributos Privados
    private String cpf;
    private String nome;

}
```

PASSO 3: Como se trata de um objeto de entidade, teremos *getters* e *setters* para todos os atributos (cpf e nome). 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 4: 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 a seguir.

Cliente.java (método toString)

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

PASSO 5: Vamos agora testar o que foi feito. Vá ao arquivo index.jps e modifique o conteúdo do corpo da seguinte forma:

```
<%--
    Document   : index
    Created on : 23/08/2011, 10:38:32
    Author     : djcaetano
--%>

<%@page contentType="text/html" pageEncoding="UTF-8" import="entidades.*" %>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Teste</title>
    </head>
    <body>

        <%
              Cliente cli = new Cliente();
              cli.setNome("José da Silva");
              cli.setCpf("01234567890");
              out.println("<p>" + cli + "</p>");
        %>

    </body>
</html>
```

PASSO 6: Execute o programa e veja se, agora, o objeto é impresso corretamente.

PASSO 7: Uma das funções importantes de um objeto de entidade é que seus **getters** e **setters** realizem alguma validação. Nos getters que respondem objetos (como getCpf e getNome, que devolvem objetos **String**) é preciso definir que, caso estes objetos não existam (valor "null"), um texto vazio deve ser devolvido. Para isso, modifique os métodos getNome e getCpf, conforme indicado a seguir.

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 8: Agora falta definir a validação dos **setters**. Os setters são métodos usados para **atualizar** os valores dos atributos. Essa atualização **não deve** ser realizada caso os novos valores sejam inválidos. Assim, precisamos **validar** os dados de nome e cpf, iniciando pelo

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 o retorno do método foi **modificado** para **boolean**, para que seja possível verificar, por quem chamou o método, se o resultado da alteração teve ou não sucesso. Observe, também, 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 9: Por último, algo um pouco mais complicado: validar o CPF. Não faremos uma validação completa, pois excluiremos a validação do dígito de verificação. Será seguido 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;
}
```

Nas futuras disciplinas de Java SE serão apresentados mais detalhes sobre a criação de classes de entidade ainda mais completas. Por hora, trabalharemos com esta.

3. IMPLEMENTANDO A PERSISTÊNCIA

Antes de mais nada, precisamos de um banco de dados. Na aba de "serviços" do NetBeans, inicie o servidor **Java DB** e crie o banco de dados **sisclientes**, usando como usuário o nome **sisclientes** e como senha também **sisclientes**. Mude o nome de visualização para **SisClientes DB**.

Conecte ao banco de dados **sisclientes** e defina **APP** como o esquema padrão. Agora, na parte **APP > TABELAS**, crie uma tabela chamada **cliente** com os campos:

Nome	Tipo	Tamanho	Chave Primária	NULL
cpf	CHAR	11	Sim	Não
nome	VARCHAR	150	Não	Sim

Agora, vamos aproveitar o conhecimento das aulas anteriores e vamos acrescentar, em nossa classe Cliente, o seguinte método chamado **persist**:

Cliente.java (persist)

```
    /**
     * Adiciona/Atualiza um cliente no banco de dados.
     * @return true se cliente foi armazenado/atualizado com sucesso.
     */
    public boolean persist() {
        try {
            // *** CONECTA AO BANCO
            // Linka com driver
            Class.forName("org.apache.derby.jdbc.ClientDriver");
            // Conecta ao banco
            Connection con = DriverManager.getConnection(
                   "jdbc:derby://localhost:1527/sisclientes",
                   "sisclientes", "sisclientes");
            if (con == null) throw new SQLException();
            // Cria a transação
            Statement trans = con.createStatement();

           //*** Executa UPDATE!
            String query  = "UPDATE app.cliente SET ";
            query += "nome = '" + getNome() + "'";
            query += " WHERE ";
            query += "cpf = '" + getCpf() + "'";
            int linhas = transacao.executeUpdate(query);

            // Se NÃO foi possível inserir, tenta atualizar!
            if (linhas == 0) {
                 //*** Executa INSERT
                  String query  = "INSERT INTO app.cliente VALUES(";
                  query += "'" + getCpf() + "'";
                  query += ", ";
                  query += "'" + getNome() + "'";
                  query += ")";
                  linhas = transacao.executeUpdate(query);
            }

            //*** Finaliza transação e conexão
            transacao.close();
            con.close();

            // Se foi possível alterar o banco de dados... Retorna ok.
            if (linhas != 0) return true;
            }
        // se houve algum erro nas transações de SQL...
        catch (SQLException ex) {
            System.err.println(ex); // Código apenas para debug
            }
```

```
        // Se não encontrou o driver
        catch (ClassNotFoundException ex) {
            System.err.println(ex); // Código apenas para debug
            }

        // Por padrão, retorna erro no fim.
        return false;
    }
```

4. USANDO O BEAN

PASSO 1: Modique o **index.jsp** da seguinte forma:

```
<%--
    Document   : index
    Created on : 23/08/2011, 10:38:32
    Author     : djcaetano
--%>

<%@page contentType="text/html" pageEncoding="UTF-8" %>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Teste</title>
    </head>
    <body>
          <h1><a href="NovoCliente">Novo Cliente</a></h1>

    </body>
</html>
```

PASSO 2: Crie um pacote java chamado **sisclientes** e, dentro dele, crie um servlet de nome **NovoCliente**, lembrando de adicionar as informações do descritor XML.

PASSO 3: Nesse servlet, modifique o método processRequest da seguinte forma:

NovoCliente.java (processRequest)

```
    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {

        try {
            Cliente cli = new Cliente();
            cli.setNome("José da Silva");
            cli.setCpf("01234567890");

            request.setAttribute("cliente", cli);
            RequestDispatcher rd;
            rd = request.getRequestDispatcher("CienteView.jsp");
            rd.forward(request.response);
            return;

        } finally {
        }
    }
```

PASSO 4: Crie, agora, na pasta de arquivos Web, o JSP **ClienteView.jsp**:

ClienteView.jsp

```
<%@page contentType="text/html" pageEncoding="UTF-8" import="entidades.*" %>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Mostra Cliente</title>
    </head>
    <body>

            <%
                  Cliente cli = (Cliente)request.getAttribute("cliente");
                  out.println("<p>" + cli + "</p>");
            %>

    </body>
</html>
```

PASSO 5: Experimente!

5. IMPLEMENTANDO A RESTAURAÇÃO

A restauração é o processo inverso do armazenamento, isto é, é relativo à recuperação de dados previamente armazenados. A restauração será feita com a implementação de um método chamado **restore** na classe Cliente, como indicado a seguir.

Cliente.java (restore)

```
    /** Restaura um cliente a partir do banco de dados.
     *  @return true se cliente foi armazenado/atualizado com sucesso.
     */
    public boolean restore(String umCpf) {
        try {
            // *** CONECTA AO BANCO
            // Linka com driver
            Class.forName("org.apache.derby.jdbc.ClientDriver");
            // Conecta ao banco
            Connection con = DriverManager.getConnection(
                    "jdbc:derby://localhost:1527/sisclientes",
                    "sisclientes", "sisclientes");
            if (con == null) throw new SQLException();
            // Cria a transação
            Statement trans = con.createStatement();

            //*** Executa SELECT!
            String query  = "SELECT * FROM app.cliente WHERE ";
            query += "cpf = '" + umCpf + "'";
            ResultSet res = transacao.executeQuery(query);
            if (res.next()) {
                    setCpf( res.getString("cpf") );
                    setNome( res.setString("nome") );
                   transacao.close();
                   con.close();
                   return true;
            }
            else setCpf(umCpf);

            //*** Finaliza transação e conexão
            transacao.close();
            con.close();
           }
```

```
        // se houve algum erro nas transações de SQL...
        catch (SQLException ex) {
            System.err.println(ex); // Código apenas para debug
            }
        // Se não encontrou o driver
        catch (ClassNotFoundException ex) {
            System.err.println(ex); // Código apenas para debug
            }
        // Por padrão, retorna erro no fim.
        return false;
    }
```

Para testar, vamos tentar recuperar o cliente no servlet **NovoCliente**:

NovoCliente.java (processRequest)

```
    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {

        try {
            Cliente cli = new Cliente();
            cli.restore("01234567890");

            request.setAttribute("cliente", cli);
            RequestDispatcher rd;
            rd = request.getRequestDispatcher("CienteView.jsp");
            rd.forward(request.response);
            return;

        } finally {
        }
    }
```

Experimente!

6. RESTAURAÇÃO AUTOMÁTICA

Será que não é possível criar um processo de "restauração automática" quando o objeto é criado?

Na verdade, é. Sempre que quisermos executar um código logo que um objeto é criado, devemos usar o método **construtor**. O método construtor **sempre terá o mesmo nome que a classe**. Por exemplo, na classe cliente, podemos declará-lo assim:

Cliente.java (construtor)

```
    public Cliente() {

    }
```

Observe que o construtor **não retorna nenhum tipo de dado**. De fato, **é proibido** usar **return** no construtor. Mas dentro desse bloco podemos escrever o que quisermos. Por exemplo:

Cliente.java (construtor)

```
    public Cliente() {
        restore();
    }
```

Obviamente isso não vai funcionar... afinal, o método **restore** precisa de um parâmetro: **o cpf do cliente a restaurar**!

Ora, foi dito que o construtor não pode retornar nenhum valor... **mas ele pode receber parâmetros**. Modifique-o assim:

Cliente.java (construtor)

```
public Cliente(String cpf) {
    restore(cpf);
}
```

E pronto!

Agora só falta fazer algumas modificações no Servlet **NovoCliente.** Observe no código a seguir.

NovoCliente.java (processRequest)

```
protected void processRequest(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {

    try {
        Cliente cli = new Cliente("01234567890");

        request.setAttribute("cliente", cli);
        RequestDispatcher rd;
        rd = request.getRequestDispatcher("CienteView.jsp");
        rd.forward(request.response);
        return;

    } finally {
    }
}
```

Experimente!

7. MÚLTIPLOS CONSTRUTORES

O método que usamos anteriormente é perfeitamente legítimo; entretanto, ele sempre nos obriga a uma busca no banco de dados **mesmo que saibamos** que o cliente não existe lá. Como poderíamos evitar aquela busca ao banco?

SIMPLES... criando **outro construtor diferente**.

Nada me obriga a ter um único construtor. **A única exigência** quando se usa múltiplos construtores **é que os parâmetros sejam diferentes**.

Assim, é perfeitamente **correto** o código a seguir:

Cliente.java (construtores)

```
public Cliente(String cpf) {
    restore(cpf);
}

public Cliente() {
}
```

Com estes dois construtores, quando o objeto for criado assim:

Cliente cli = new Cliente();   // sem parâmetros

O objeto será criado com o construtor que não faz nada. Por outro lado, se o objeto for criado assim:

Cliente cli = new Cliente("01234567890");   // com um parâmetro String

O objeto será criado com o construtor que executa o **restore**.

8. BIBLIOGRAFIA

QIAN, K; ALLEN, R; GAN, M; BROWN, R. **Desenvolvimento Web Java**. Rio de Janeiro: LTC, 2007.

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

Unidade 09: Sessão e Login

Prof. Daniel Caetano

Objetivo: Permitir compartilhamento de dados temporário entre diferentes Servlets que compõem uma aplicação.

INTRODUÇÃO

Até agora, sempre que precisamos transferir um dado entre um servlet para outro nos bastava inseri-lo em uma requisição e transferi-la. Entretanto, isso funciona apenas em aplicações mais simples. Quando se deseja construir uma aplicação profissional, é frequente precisar armazenar algumas informações de maneira que vários servlets possam compartilhá-la.

Na aula de hoje aprenderemos como fazer isso por meio das variáveis de sessão, que será usada, nessa aula, para implementar o controle de login.

1. ESTADOS E O COMPORTAMENTO DOS COMPONENTES

Antes de mais nada, precisamos compreender o que significa "login".

Vamos começar entendendo os dois tipos de componente de um sistema, quando classificados de acordo com seu comportamento: **existem os componentes que executam sempre a mesma tarefa**... e **existem os componentes que, dependendo da situação, executam tarefas diferentes**.

No primeiro caso, isto é, em componentes que fazem sempre a mesma tarefa, para um dado conjunto de parâmetros, o resultado sempre o mesmo. Por exemplo: se criarmos um componente que realiza a soma de dois números, sempre que passarmos os números 2 e 3 como parâmetros, o componente nos responderá o resultado 5.

No segundo caso, o componente não age sempre da mesma forma. Digamos que criemos um componente que, dependendo da situação, ele calcula a soma ou a multiplicação. Neste caso, se passarmos os valores 2 e 3 para este componente poderemos obter como resposta os valores 5 (resultado da soma) ou 6 (resultado da multiplicação).

Neste segundo caso, dizemos que a resposta do componente depende do **estado** deste componente. Para entender o conceito, vamos a um exemplo mais concreto. Quando somos jovens, eventualmente precisamos pedir dinheiro para nossos pais para, por exemplo, pagar o ônibus até a escola. Neste caso, nosso pai pode responder fornecendo o dinheiro ou pode responder que não tem dinheiro. A resposta, depende, portanto, do **estado** financeiro de nosso pai, isto é, se ele tem ou não o dinheiro.

Por essa razão, os componentes que sempre se comportam da mesma maneira, considerando apenas os parâmetros, chamamos de "**componentes sem estado**" ou, usando o

jargão da área, são componentes *stateless*. Já os componentes que podem se comportar de maneira diferente, isto é, fornecer respostas diferentes para uma mesma requisição, são chamados de "**componentes com estado**" ou, usando o jargão da área, são componentes *statefull*.

Quando desenvolvemos um sistema que envolve componentes com estado é importante identificar 3 elementos:

a) Quais são esses estados?
b) Como ocorre a mudança de um estado para outro?
c) Qual o comportamento do componente em cada um destes estados?

As perguntas (a) e (b) são respondidas por um diagrama chamado "Diagrama de Estados", como o representado abaixo:

A bolinha preta indica o estado inicial do sistema (no caso, "Usuário não Autenticado") e as setas indicam as ações que levam à transição entre tais estados - A autenticação no Banco de Dados muda o estado para "Usuário Autenticado", por exemplo.

A pergunta (c) é, normalmente, respondida pela descrição dos Casos de Uso: uma das seções desta descrição é chamada de "Pré-Condições". Nessa seção deve ser descrito, por exemplo: "Para esse caso de uso ocrrer, o sistema deve estar no estado "Usuário Autenticado".

2. O LOGIN E OS ESTADOS RELACIONADOS

Sempre que um sistema possui login, praticamente todos os seus componentes passam a ter **dois** estados distintos:

1. Logado: atua normalmente
2. Não Logado: redireciona a execução para a tela de login

A transição entre estes estados se dá nos seguintes casos:

<u>**Não Logado para Logado**</u>:
1. Quando o usuário realiza o Login

<u>**Logado para Não Logado**</u>:
1. Quando o usuário seleciona a opção "sair"
2. Quando passou tempo demais sem que o usuário fizesse nada.

Para que os componentes de um sistema (Servlets ou JSPs) possam reagir adequadamente em cada caso, eles precisam ter acesso a uma variável que lhes indique se um determinado usuário **já fez o login** ou se **não fez o login**.

Para armazenar essa informação usaremos um recurso do Java EE chamado Sessão.

3. A SESSÃO

De maneira direta, a Sessão é um objeto do Java EE que permite o compartilhamento de dados/objetos entre os vários Servlets/JSPs que compõem uma Aplicação Web.

A sessão pode ser encarada como uma grande caixa de "achados e perdidos" aos quais todos os Servlets/JSPs têm acesso, pois existe uma referência para ela dentro da **request**. Para "descobrir" quem é a sessão do Servlet/JSP atual, usa-se o seguinte comando:

```
request.getSession()
```

Frequentemente ele aparece no código associado a uma variável de objetos do tipo HttpSession, com o seguinte formato:

```
HttpSession session = request.getSession();
```

NOTA: O passo de pegar uma referência para a sessão é desnecessário nos JSPs, já que o objeto **session** já é previamente criado. Basta usá-lo.

O armazenamento de objetos na sessão pode ser feito **exatamente** da mesma forma que na requisição:

```
session.setAttribute("etiqueta", objeto);
```

Recuperar um objeto da sessão também se parece com o caso da requisição:

```
Object objeto = (Object)session.getAttribute("etiqueta");
```

A diferença é que tudo que for colocado na sessão permanecerá lá até que seja retirado ou até que a sessão seja encerrada.

Para tirar um objeto da sessão, basta removê-lo com o comando apropriado:

```
/* Coloca o Objeto */
session.setAttribute("etiqueta", objeto);

/* Remove o Objeto */
session.removeAttribute("etiqueta");
```

3.1. Ciclo de Vida da Sessão

A sessão é criada pelo Java automaticamente, assim que é solicitada à **request** pela primeira vez . Depois disso, há duas formas de ela ser destruída:

a) Por tempo
De acordo com o tempo especificado no arquivo **web.xml**, chamdo ***timeout***

b) Por solicitação do usuário
Pelo método **invalidate**: **session.invalidate();**

Um Servlet/JSP pode alterar o ***timeout*** de uma sessão através do seguinte comando:

session.setMaxInactiveInterval(tempo);

Por exemplo, para mudar o tempo máximo de inatividade para 5 minutos (300 segundos), basta dar o comando:

session.setMaxInactiveInterval(300);

4. O COMPONENTE DE LOGON

Uma vez que nenhum sistema comercial profissional estará completo sem um sistema de logon apropriado, é interessante criá-lo da maneira mais reutilizável possível... e é exatamente isso que veremos nessa aula.

Vamos começar criando uma nova Aplicação Web, **WProjeto8**. Vamos fazer algo realmente simples para testar:

PASSO A. Crie um Pacote Java chamado **teste**
PASSO B. Crie um Servlet chamado **Teste**
PASSO C. Faça este servlet inserir uma String na requisição:

Etiqueta	String
msg	"Olá, você obteve acesso!"

Teste.java (método processRequest)

```
protected void processRequest(HttpServletRequest request,
  HttpServletResponse response)
  throws ServletException, IOException {

    RequestDispatcher rd;

    try {
        request.setAttribute("msg","Olá, você obteve acesso!");
        rd = request.getRequestDispatcher("/Teste.jsp");
        rd.forward(request, resposnse);
        return;
    } finally {
    }
}
```

PASSO D. Crie, agora, um JSP chamado **TesteView**
PASSO E. Modifique o JSP para que imprima a mensagem **msg** da requisição.
PASSO F. Teste o seu servlet, executando:

http://localhost:8080/WProjeto8/Teste

Observe que você obteve acesso.

Vamos, agora, proteger este "sistema" com um logon.

4.1. Criando a Base do Componente de Logon

Como pretendemos criar um módulo de login reaproveitável, vamos criá-lo como um pacote separado. Com o projeto **WProjeto8** aberto...

PASSO 1: Clique com o botão direito na pasta "Pacotes de Código Fonte" e selecione **Novo > Pacote Java**. Indique o nome **Logon** para este pacote e clique, depois em Finalizar.

PASSO 2: Clique com o botão direito no pacote "Logon" e seleciona **Novo > Classe Java**. Indique o nome **Logon** para essa classe.

PASSO 3: Agora precisamos criar o servlet **WLogon**, que irá receber as informações de um formulário e verificá-la. Clique com o botão direito no pacote Logon e selecione **Novo > Servlet**. Dê o nome de **WProcessaLogon** ao servlet e não esqueça de marcar a caixa que adiciona as informações do servlet no descritor. Esse é um servlet de processamento, que deve ter o seguinte código de processRequest:

WLogon (método processRequest)

```
protected void processRequest(HttpServletRequest request,
  HttpServletResponse response)
  throws ServletException, IOException {

    try {
        // Recupera parâmetros
        String nome = request.getParameter("user");
        String pass = request.getParameter("pass");
        String sucesso = request.getParameter("sucesso");
        String fracasso = request.getParameter("fracasso");
        // Pega referência para a Sessão
        HttpSession sessao = request.getSession();
        // Define referência para o request dispatcher
        RequestDispatcher rd;


        // Verifica logon...
        boolean logon = false;
        if (nome.compareTo("admin")==0 && pass.compareTo("12345")==0) {
            logon = true;
        }

        // Se logon passou...
        if (logon == true) {
            // Coloca nome do usuário na sessão, com etiqueta
            // Igual a "user"
            sessao.setAttribute("user",nome);
            // Redireciona para destino em caso de sucesso.
            rd = request.getRequestDispatcher(sucesso);
```

```
                rd.forward(request,response);
                return;
            }
            // SE logon NÃO passou...
            else {
                // Remove usuário da sessão
                sessao.removeAttribute("user");
                // Redireciona para destino em caso de fracasso
                rd = request.getRequestDispatcher(fracasso);
                rd.forward(request,response);
                return;
            }
        }
    }
```

PASSO 4: Em caso de fracasso, podemos adicionar um código de erro também:

WLogon (método processRequest)

```
    protected void processRequest(HttpServletRequest request,
      HttpServletResponse response)
      throws ServletException, IOException {

        try {
            // Recupera parâmetros
            String nome = request.getParameter("user");
            String pass = request.getParameter("pass");
            String sucesso = request.getParameter("sucesso");
            String fracasso = request.getParameter("fracasso");
            // Pega referência para a Sessão
            HttpSession sessao = request.getSession();
            // Define referência para o request dispatcher
            RequestDispatcher rd;

            // Verifica logon...
            boolean logon = false;
            if (nome.compareTo("admin")==0 && pass.compareTo("12345")==0) {
                logon = true;
            }
            // Se logon passou...
            if (logon == true) {
                // Coloca nome do usuário na sessão, com etiqueta
                // Igual a "user"
                sessao.setAttribute("user",nome);
                // Redireciona para destino em caso de sucesso.
                rd = request.getRequestDispatcher(sucesso);
                rd.forward(request,response);
                return;
            }
            // SE logon NÃO passou...
            else {
                // Remove usuário da sessão
                sessao.removeAttribute("user");
                 // Adiciona código de erro na requisição
                 request.setAttribute("erro","Usuário ou senha incorretos!");
                // Redireciona para destino em caso de fracasso
                rd = request.getRequestDispatcher(fracasso);
                rd.forward(request,response);
                return;
            }
        }
    }
```

4.2. Criando o Formulário de Logon na Aplicação Web

PASSO 5: Vamos, agora, criar um formulário de logon. Esse formulário será criado no index.jsp, como a tela de entrada do sistema. Modifique o HTML do **index.jsp** assim:

index.jsp (apenas html)

```
<%   String erro = (String)request.getAttribute("erro");
     if (erro == null) erro = "";
%>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Acesso ao Sistema</title>
    </head>
    <body>
        <h1>Logon no Sistema</h1>
        <p><%= erro %>
        <form action="WLogon" method="post">
            <p>Usuário: <input type="text" name="user"></p>
            <p>Password: <input type="password" name="pass"></p>
            <input type="hidden" name="sucesso" value="/Teste">
            <input type="hidden" name="fracasso" value="/index.jsp">
            <input type="submit" value="Entrar!">
        </form>
    </body>
</html>
```

Este formulário enviará os dados para o servlet WProcessaLogon, que precisaremos criar. **Observe os campos "input hidden".** Estes campos servem para indicar os **servlets destino**, caso o logon ocorra com sucesso ou com fracasso.

Teste! O logon já deve funcionar... mas o acesso direto ainda é permitido. Veja:

http://localhost:8080/WProjeto8/Teste

4.3. Verificando o Logon

PASSO 6: Iremos agora modificar os Servlets/JSPs para que verifiquem se o logon está realizado, através da ***bandeira*** "user" (flag). Por exemplo, vamos começar a modificação pelo servlet Teste.

Teste.java (método processRequest)

```
    protected void processRequest(HttpServletRequest request,
      HttpServletResponse response)
      throws ServletException, IOException {

        RequestDispatcher rd;

        // Verifica Logon
        HttpSession s = request.getSession();
        String user = (String)s.getAttribute("user");
        if (user == null) {
            rd = request.getRequestDispatcher("/index.jsp");
            rd.forward(request,response);
            return;
        }

        try {

            request.setAttribute("msg","Olá, você obteve acesso!");
            rd = request.getRequestDispatcher("/Teste.jsp");
            rd.forward(request, resposnse);
            return;
        } finally {
        }
    }
```

PASSO 7: Experimente! Observe que isso funciona, mas é um tanto tosco. Será que teremos que adicionar esse código a todas as páginas? Muito código para pouco serviço! Para dar um jeito nisso, vamos transferir essa função, muito comum, para o servlet WLogon.

A idéia é criar criar um método estático chamado **verifica** no Servlet **WLogon**. E podemos fazer isso? Wlogon não é um Servlet?

Claro que podemos. Servlet ou não, Wlogon continua sendo uma **classe** Java e, sendo assim, podemos fazer com ela o que bem entendermos. Observe o código que deve ser acrescentado na classe WLogon:

WLogon.java (método verifica)

```
    /**
     * Verifica se logon já foi realizado.
     * Verifica se existe marca de logon na sessão. Se houver,
     * retorna true. Se não houver, encaminha execução para
     * servlet de logon e retorna falso.
     * @param request Objeto de requisição.
     * @param response Objeto de resposta.
     * @param logUrl URL, iniciada por /, indicando o endereço de logon
     * @return true caso o estado seja de logon ativo
     * @throws ServletException
     * @throws IOException
     */
    public static boolean verifica( HttpServletRequest request,
            HttpServletResponse response, String logUrl)
            throws ServletException, IOException {
        // Recupera "user" da sessão
        HttpSession s = request.getSession();
        String user = (String)s.getAttribute("user");
        // Se não existir, vai pra tela de logon
        if (user == null) {
            RequestDispatcher rd;
            rd = request.getRequestDispatcher(logUrl);
            rd.forward(request,response);
            return false;
        }
        return true;
    }
```

Observe a necssidade de mandar um monte de informações para que esse método possa fazer seu serviço: o objeto de requisição, o objeto de resposta e a URL de logon. Infelizmente, tudo isso é necessário.

PASSO 8: O método **verifica** que acabamos de criar fará o "serviço sujo"; ele responde verdadeiro caso usuário esteja logado e, se não, responde falso e encaminha a execução para a tela de login fornecida. Mas para que ele faça seu serviço, precisamo usá-lo! O jeito de usá-lo é esse:

WLogon.verifica(request, response, "/index.jsp");

Todos os parâmetros, como já ressaltado, são necessários para o check realizar o processamento dele; o último é o único que eventualmente mudaremos, pois ele indica o endereço da tela de logon.

PASSO 9: Vamos agora voltar ao código do WProjeto8 e vamos substituir aquele monte de código por esta nova linha de código, lembrando de corrigir as importações... e que o texto "cortado" abaixo deve ser **apagado**!

Teste.java (método processRequest)

```
protected void processRequest(HttpServletRequest request,
  HttpServletResponse response)
  throws ServletException, IOException {

    RequestDispatcher rd;

    // Verifica Logon
     if (WLogon.verifica(request,response,"/index.jsp") == false) return;

    // Verifica Logon
    HttpSession s = request.getSession();
    String user = (String)s.getAttribute("user");
    if (user == null) {
        rd = request.getRequestDispatcher("/index.jsp");
        rd.forward(request,response);
        return;
    }

    try {

        request.setAttribute("msg","Olá, você obteve acesso!");
        rd = request.getRequestDispatcher("/Teste.jsp");
        rd.forward(request, resposnse);
        return;
    } finally {
    }
}
```

PASSO 10: Experimente! Sem realizar o Logon, tente entrar direto no servlet WSisCli, usando: http://localhost:8080/WProjeto8/Teste

Agora, faça o logon e veja como funciona!

Sempre que quisermos usar o sistema de Logon, temos que acrescentar essa linha no topo da página:

if (WLogon.verifica(request, response, "/index.jsp") == false) return;

Incluindo o JSP... obviamente dentro de uma seção **scriplet**:

<% if (WLogon.verifica(request, response, "/index.jsp") == false) return; %>

Essas verificações devem ser **sempre** as primeiras coisas em cada arquivo.

4.4. Saindo do Sistema

PASSO 11: Um último toque, relevante, é acrescentar a opção para fazer o logout... ou seja, precisamos de um jeito de realizar o logout de maneira simples. Na pasta Logon, crie um servlet chamado "Logoff", com o seguinte código:

WLogoff.java (método processRequest)

```
protected void processRequest(HttpServletRequest request,
  HttpServletResponse response)
  throws ServletException, IOException {

    String destino = (String)request.getAttribute("logoff");

    HttpSession session = request.getSession();
    session.invalidate();

    RequestDispatcher rd = request.getRequestDispatcher(destino);
    rd.forward(request, response);
    return;

    } finally {
    }
}
```

Para fazer o logoff, basta, no HTML, usar um link:

<p>Sair</p>

Experimente acrescentar isso ao **TesteView.jsp**!

Com isso o nosso sistema de logon básico está implementado.

5. INCREMENTANDO O LOGON DO SISTEMA (OPCIONAL)

Como poderíamos modificar o componente Logon de maneira a verificar, em um banco de dados, se o usuário existe? Bem, basicamente precisaríamos criar uma tabela no banco de dados, como indicado a seguir, usando o administrador de banco de dados do NetBeans, como visto em aulas anteriores.

PASSO 1: Crie um bando de dados chamado **logon**, usuário **logon** e senha **logon**.

PASSO 2: Crie uma tabela chamada **usuario**, que irá armazenar os dados de nossos usuários. Como os usuários possuem apenas **nome** e **pass** teremos duas colunas nesta tabela, conforme indicado a seguir.

nome : PK, VARCHAR(20)	password : VARCHAR(50)
...	...

PASSO 3: Vamos, agora, insira manualmente os dados nessa tabela, por exemplo:

nome: **admin**
password: **12345**.

```
INSERT INTO usuarios VALUES ('admin', '12345');
```

PASSO 4: Execute o select a seguir e observe que o password foi gravado de forma codificada!

```
SELECT * FROM usuarios;
```

PASSO 5: Agora, ao código. Crie, dentro do pacote Logon, uma classe chamada **LogonDAO** como indicada a seguir:

LogonDAO.java

```
package Logon;
import java.sql.*;
/**
 * Responsável por Verificar Logon
 * Essa classe torna o uso do BD "transparente" para o logon.
 * @author djcaetano
 */
public class LogonDAO {
    /**
     * Verifica Logon.
     * @param nome Nome do usuário.
     * @param pass Password do usuário.
     * @return True se usuário existe com esse password.
     */
    public static boolean verifica(String nome, String password) {
        try {
            // *** CONECTA AO BANCO
            // Seleciona driver
            Class.forName("org.apache.derby.jdbc.ClientDriver");
            // Indica endereco do banco
            Connection con = DriverManager.getConnection(
                "jdbc:derby://localhost:1527/logon",
                "logon", "logon");
            if (con == null) throw new SQLException();
            // Cria a transação
            Statement transacao = con.createStatement();

            //*** Executa Busca (Query)!
            // Cria a query de busca...
            String query  = "SELECT nome FROM usuarios WHERE ";
            query += "nome LIKE '" + nome + "'";
            query += " && ";
            query += "password LIKE '" + password + "'";
            ResultSet res = transacao.executeQuery(query);

            if (res.next()) {
                transacao.close();
                con.close();
                return true;
            }

            //*** Finaliza transação e conexão
            transacao.close();
            con.close();
            }

        // se houve algum erro nas transações de SQL...
        catch (SQLException ex) {
            System.err.println(ex); // Código apenas para debug
            }
        // Se não encontrou o driver
        catch (ClassNotFoundException ex) {
            System.err.println(ex); // Código apenas para debug
            }

        // Por padrão, retorna não encontrado.
        return false;
    }
```

PASSO 6: Agora modifique a classe **WLogon**, deste mesmo pacote **Logon**, como indicado a seguir, lembrando que as linhas riscadas devem ser removidas:

WLogon (método processRequest)

```
    protected void processRequest(HttpServletRequest request,
      HttpServletResponse response)
      throws ServletException, IOException {
        try {
            // Recupera parâmetros
            String nome = request.getParameter("user");
            String pass = request.getParameter("pass");
            String sucesso = request.getParameter("sucesso");
            String fracasso = request.getParameter("fracasso");
            // Pega referência para a Sessão
            HttpSession sessao = request.getSession();
             // Define referência para o request dispatcher
            RequestDispatcher rd;

            // Verifica logon...
            boolean logon = LogonDAO.verifica(nome,pass);
             boolean logon = false;
            if (nome.compareTo("admin")==0 && pass.compareTo("12345")==0) {
                 logon = true;
             }

            // Se logon passou...
            if (logon == true) {
                // Coloca nome do usuário na sessão, com etiqueta
                // Igual a "user"
                sessao.setAttribute("user",nome);
                // Redireciona para destino em caso de sucesso.
                rd = request.getRequestDispatcher(sucesso);
                rd.forward(request,response);
                 return;
            }
            // SE logon NÃO passou...
            else {
                // Remove usuário da sessão
                sessao.removeAttribute("user");
                 // Adiciona código de erro na requisição
                 request.setAttribute("erro","Usuário ou senha incorretos!");
                // Redireciona para destino em caso de fracasso
                rd = request.getRequestDispatcher(fracasso);
                rd.forward(request,response);
                 return;
            }
        }
    }
```

Pronto! Seu sistema já é capaz de verificar logon a partir de informações no banco de dados!

6. COMENTÁRIOS SOBRE BANCO DE DADOS DE LOGON

Você deve ter observado que é muito simples a criação de um banco de dados de logon. Entretanto, alguns cuidados são essenciais:

a) **Evite colocar o password do usuário em um objeto**. Dependendo da forma com que o objeto for usado, poderá significar que o seu password viajará pela rede sem nenhuma codificação, criando uma falha de segurança.

b) **NUNCA armazene os passowrds sem codificação**. Fizemos uma implementação simplificada que armazena o password "em aberto". Não faça isso. Use comandos de

codificação da linguagem Java para codificar o password **antes** de armazená-lo e posteriormente codifique para verificar também. Se você armazenar os dados de password de forma não codificada e algum hacker roubar seu banco de dados, automaticamente ele saberá os passwords de todos os usuários!

O item (b) pode ser observado para qualquer informação "sensível". Algumas informações devemos evitar armazenar (como, por exemplo, números de cartão de crédito). Mas, caso os guardemos, devemos sempre guardá-los de maneira **codificada**.

Uma informação importante: **não deve ser possível** recuperar o valor original da informação de password armazenada. Sugere-se usar algo como SHA-1 ou algo do gênero, e uma vez codificada, a informação não poderá mais ser decodificada. Por isso que, na verificação, ao invés de decodificar o password do banco, nós devemos codificar o password fornecido pelo usuário.

7. BIBLIOGRAFIA

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

QIAN, K; ALLEN, R; GAN, M; BROWN, R. **Desenvolvimento Web Java.** Rio de Janeiro: LTC, 2010.

Unidade 10: Uso de Sessão e DAO e Servlets

Servlets de Comportamento Variável
Prof. Daniel Caetano

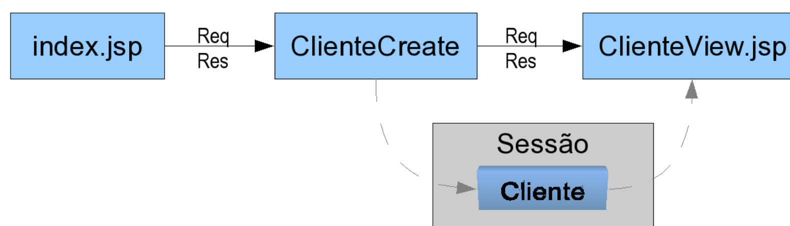
Objetivo: Implementar servlets de comportamento variável de acordo com o estado de atributos de requisição.

Bibliografia: QIAN, 2007; DEITEL, 2005.

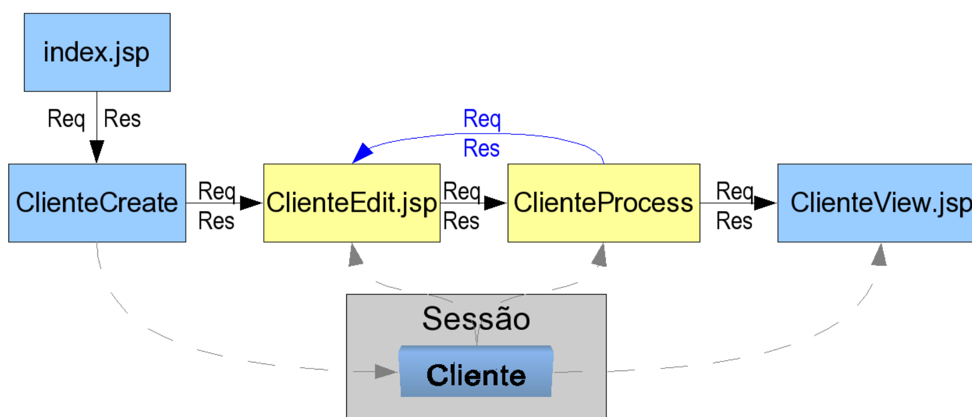
INTRODUÇÃO

Nas aulas anteriores estudamos todos os elementos para construir uma Aplicação Web elaborada de forma desconexa, isto é, vimos cada uma das partes de maneira isolada.

Nesta aula iremos construir um sistema básico muito simples, como o indicado abaixo, usando alguns dos elementos que já construímos anteriormente (em especial, o Cliente e o ClienteDAO):



E, mais ao fim da aula, modificaremos o mesom para um novo estágio, representado a seguir, permitindo que o cliente criado pelo **ClienteCreate** possa ser editado pela dupla de edição (a JSP de apresentação **ClienteEdit.jsp** e o servlet de processamento **ClienteProcess**) antes de ser exibido na tela.



1. CRIANDO O APLICATIVO WEB BÁSICO

O Servlet base é o servlet inicial de nossa aplicação, normalmente fornecendo um menu para acesso às principais funções do aplicativo. Para criá-lo, façamos o seguinte:

PASSO 1. Clique em **Criar Projeto** e selecione "**Java Web**" e "**Aplicação Web**" e clique em **Próximo.** Dê o nome ao projeto de **WProjeto9** e clique em **Próximo**, verifique se o servidor de aplicações selecionado é o **GlassFish** e clique em **Finalizar**.

PASSO 2. Primeiramente, vamos "migrar" as informações úteis de projetos anteriores. Abra o projeto **WProjeto6**, da aula 10, clique com o botão direito no pacote "**entidades**" e selecione a opção "**copiar**". Clique no "**Pacotes de Código Fonte**" do WProjeto9 que acabamos de criar e selecione "**colar**". Feche o WProjeto6. Consideramos aqui que o banco de dados necessário já está criado, por conta do antigo projeto!

PASSO 3. Vamos agora ajustar o index.jsp para apontar para nosso servlet inicial que, pelo projeto, irá se chamar ClienteCreate:

index.jsp

```
<%--
    Document   : index
    Created on : 04/03/2011, 10:13:15
    Author     : djcaetano
--%>

<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>

<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Acessa o Sistema</title>
    </head>
    <body>
        <p><a href='ClienteCreate'>Criar Cliente!</a></p>
   </body>
</html>
```

PASSO 4. Agora que a aplicação está confgiruada, vamos criar o nosso servlet base. Clique com o botão direito em "**Pacotes de Código Fonte**" e selecione **Novo > Pacote Java**. Dê o nome de **cadcli** ao pacote, clicando depois em **Finalizar**.

PASSO 5. Clique com botão direito no pacote **cadcli** e selecione **Novo > Servlet**. Dê o nome **ClienteCreate** para o servlet e clique em **Próximo**. Marque a opção "**Adicionar informação ao descritor de implementação (web.xml)**" e clique em **Ok/Finalizar**.

PASSO 6. Já com o arquvo ClienteCreate.java aberto, remova tudo que é desnecessário, considerando que este é um servlet de processamento. A função deste Servlet é criar um objeto Cliente e inseri-lo na sessão. Observe como isso pode ser feito:

ClienteCreate.java (método processRequest)

```
protected void processRequest(HttpServletRequest request, HttpServletResponse response)
 throws ServletException, IOException {

     try {

         // Cria um Cliente de Teste
         Cliente c = new Cliente();
         c.setNome("Fulano da Silva");
         c.setCpf("12345678901");

         // Requisita Objeto de Sessão
         HttpSession sessao = request.getSession();
         // Insere cliente na sessão
         sessao.setAttribute("cliente", c);

         // Solicita o despachante de requisições
         RequestDispatcher rd;
         rd = request.getRequestDispatcher("/ClienteView.jsp");
         // Encaminha requisição
         rd.forward(request,response);
         return;

     } finally {
     }
 }
```

PASSO 7. Vamos, finalmente, criar o ClienteView.jsp. Clique com o botão direito em "**Páginas Web**" e selecione **Novo > JSP...**. Dê o nome de **ClienteView** ao JSP e modifique o código dele como se segue:

ClienteView.jsp

```
<%@page contentType="text/html" pageEncoding="UTF-8" import="entidades.*" %>

<%
      String texto = "Nenhum cliente para visualizar.";
      Cliente c = (Cliente)session.getAttribute("cliente");
      if (c != null) texto = c.toString();
%>

<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Cliente View</title>
    </head>
    <body>

        <%= texto %>

    </body>
</html>
```

Experimente a aplicação! Veja se funciona!

2. ADICIONANDO A EDIÇÃO À APLICAÇÃO

PASSO 8. Primeiramente, vamos criar a apresentação da edição, ou seja, o ClienteEdit.jsp. Para tanto, clique com o botão direito em "**Páginas Web**" e selecione **Novo > JSP...**. Dê o nome de **ClienteEdit** ao JSP e modifique o código dele como se segue:

ClienteEdit.jsp

```
<%@page contentType="text/html" pageEncoding="UTF-8" %>

<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Cliente Edit</title>
    </head>
    <body>

      <form action="ClienteProcess" method="post">
             <p>CPF: <input type="text" name="cpf"></p>
             <p>Nome: <input type="text" name="nome"></p>
             <input type="submit" value="Gravar">
      </form>

    </body>
</html>
```

PASSO 9: Precisamos, agora, criar o servlet de processamento ClienteProcess, que irá receber estes dados e armazená-lo no objeto Cliente. Clique com botão direito no pacote **cadcli** e selecione **Novo > Servlet**. Dê o nome **ClienteProcess** para o servlet e clique em **Próximo**. Marque a opção "**Adicionar informação ao descritor de implementação (web.xml)**" e clique em **Ok/Finalizar**.

PASSO 10. Já com o arquvo ClienteProcess.java aberto, remova tudo que é desnecessário, considerando que este é um servlet de processamento. A função deste Servlet é coletar os dados do formulário, modificar o objeto cliente que está na sessão e, se tudo correr bem, persistí-lo. Observe como isso pode ser feito no código a seguir.

ClienteProcess.java (método processRequest)

```
   protected void processRequest(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
        try {
            // Recupera cliente da sessão
            HttpSession sessao = request.getSession();
            Cliente c = (Cliente)sessao.getAttribute("cliente");
            // Se não há um cliente... erro fatal. Redireciona.
            if (c == null) response.sendRedirect("index.jsp");

            // Recupera dados do formulário
            String nome = request.getParameter("nome");
            String cpf = request.getParameter("cpf");

            // Insere dados no cliente
            boolean res = true
            if (c.setNome(nome) == false) {
                res = false;
            }
            else if (c.setCpf(cpf) == false) {
                res = false;
            }
            // Se tudo correu bem...
            if (res == true) {
                // Persiste objeto cliente
                ClienteDAO.adiciona(cliente);
                // Envia processamento para Mostra Cliente
```

```
                RequestDispatcher rd;
                rd = request.getRequestDispatcher("/ClienteView.jsp");
                rd.forward(request,response);
                return;
            }
            // Caso contrário...
            else {
                // Envia processamento de volta para tela de edição
               RequestDispatcher rd;
                rd = request.getRequestDispatcher("/ClienteEdit.jsp");
                rd.forward(request,response);
                return;
            }

        } finally {
        }
    }
```

PASSO 11: O toque final é modificar o fluxo da aplicação: o nosso ClienteCreate redirecionava a execução diretamente para o ClienteView.jsp. Vamos modificá-lo para redirecionar par o ClienteEdit.jsp:

ClienteCreate.java (método processRequest)

```
    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
     throws ServletException, IOException {

        try {

            // Cria um Cliente de Teste
            Cliente c = new Cliente();
            c.setNome("Fulano da Silva");
            c.setCpf("12345678901");

            // Requisita Objeto de Sessão
            HttpSession sessao = request.getSession();
            // Insere cliente na sessão
            sessao.setAttribute("cliente", c);

            // Solicita o despachante de requisições
            RequestDispatcher rd;
            rd = request.getRequestDispatcher("/ClienteEdit.jsp");
            // Encaminha requisição
            rd.forward(request,response);
            return;

        } finally {
        }
    }
```

PASSO 12: Experimente! Veja o que acontece quando você tenta editar um cliente e usar um CPF inválido!

3. COMPLEMENTANDO COM RECURSOS ADICIONAIS

Nosso sistema de edição tem dois problemas graves ainda:

1) Quando vou editar um cliente que já está na sessão, seus dados não são mostrados no formulário;

2) Quando erramos algo no formulário de edição, ele reaparece sem nenhuma indicação do erro.

Vaoms corrigir essas duas pendências.

PASSO 13: Primeiramente vamos modificar o ClienteEdit.jsp para que mostre os dados do cliente em seus campos - se o cliente existir, é claro...

ClienteEdit.jsp

```
<%@page contentType="text/html" pageEncoding="UTF-8" import="entidades.*" %>

<%
      String nome = "";
      String cpf = "";
      Cliente c = (Cliente)session.getCliente();
      if (c != null) {
            nome = c.getNome();
            cpf = c.getCpf();
      }
%>

<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Cliente Edit</title>
    </head>
    <body>

      <form action="ClienteProcess" method="post">
            <p>CPF: <input type="text" name="cpf" value="<%= cpf %>"></p>
            <p>Nome: <input type="text" name="nome" value="<%= nome %>"></p>
            <input type="submit" value="Gravar">
      </form>

    </body>
</html>
```

PASSO 14: O segundo problema exige um pouco mais de modificações. Primeiramente, vamos fazer com que o servlet ClienteProcess armazene mensagens de erro na requisição, com a etiqueta **erros**. Como os erros podem ser muitos, vamos usar uma **ArrayList** para isso.

ClienteProcess.java (método processRequest)

```
    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
        try {
            // Recupera cliente da sessão
            HttpSession sessao = request.getSession();
            Cliente c = (Cliente)sessao.getAttribute("cliente");
            // Se não há um cliente... erro fatal. Redireciona.
            if (c == null) response.sendRedirect("index.jsp");

            // Recupera dados do formulário
            String nome = request.getParameter("nome");
            String cpf = request.getParameter("cpf");

            // Cria lista de erros
            ArrayList<String> erros = new ArrayList<String>();

            // Insere dados no cliente
            boolean res = true
            if (c.setNome(nome) == false) {
                res = false;
                erros.add("Nome inválido");
            }
            else if (c.setCpf(cpf) == false) {
                res = false;
                erros.add("CPF inválido");
            }
```

```
            // Se tudo correu bem...
            if (res == true) {
                // Persiste objeto cliente
                ClienteDAO.adiciona(cliente);
                // Envia processamento para Mostra Cliente
                RequestDispatcher rd;
                rd = request.getRequestDispatcher("/ClienteView.jsp");
                rd.forward(request,response);
                return;
            }
            // Caso contrário...
            else {
                // Adiciona mensagens de erro à requisição
                request.setAttribute("erros", erros);
                // Envia processamento de volta para tela de edição
                RequestDispatcher rd;
                rd = request.getRequestDispatcher("/ClienteEdit.jsp");
                rd.forward(request,response);
                return;
            }

        } finally {
        }
    }
```

PASSO 15: Agora vamos modificar o JSP ClienteEdit.jsp para que ele **mostre** os erros, caso eles existam!

ClienteEdit.jsp

```
<%@page contentType="text/html" pageEncoding="UTF-8"
      import="entidades.*, java.util.*" %>

<%
      String nome = "";
      String cpf = "";
      Cliente c = (Cliente)session.getCliente();
      if (c != null) {
             nome = c.getNome();
             cpf = c.getCpf();
      }

      ArrayList<String> lista = (ArrayList<String>)request.getAttribute("erros");
%>

<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Cliente Edit</title>
    </head>
    <body>

<%
      if (lista != null && lista.size() > 0) {
             int i;
             out.println("<p>Por favor, corrija os erros indicados abaixo:</p>");
             for (i = 0; i < lista.size(); i++) {
                    out.println("<p>" + lista.get(i) + "</p>");
             }
      }
%>

      <form action="ClienteProcess" method="post">
             <p>CPF: <input type="text" name="cpf" value="<%= cpf %>"></p>
             <p>Nome: <input type="text" name="nome" value="<%= nome %>"></p>
             <input type="submit" value="Gravar">
      </form>

    </body>
</html>
```

NOTA: OBSERVE que as mensagens de erro não estão sendo colcoadas na sessão! Isso ocorre porque **não** queremos que a mensagem de erro fique registrada permanentemente.... ela é **temporária**. A sessão é usada apenas para informações **que devem ser mantidas ao longo da execução, permanentes**, o que não é o caso de uma mensagem de erro.

PASSO 16: Para arrematar, vamos voltar à classe ClienteCreate para remover as informações iniciais que estão sendo preenchdias automaticamente no cliente!

ClienteCreate.java (método processRequest)

```
protected void processRequest(HttpServletRequest request, HttpServletResponse response)
 throws ServletException, IOException {

     try {

          // Cria um Cliente de Teste
          Cliente c = new Cliente();
          c.setNome("Fulano da Silva");
          c.setCpf("12345678901");

          // Requisita Objeto de Sessão
          HttpSession sessao = request.getSession();
          // Insere cliente na sessão
          sessao.setAttribute("cliente", c);

          // Solicita o despachante de requisições
          RequestDispatcher rd;
          rd = request.getRequestDispatcher("/ClienteView.jsp");
          // Encaminha requisição
          rd.forward(request,response);
          return;

     } finally {
     }
 }
```

<u>4. ATIVIDADE</u>

Agora que criamos um elemento mais completo de um sistema, você saberia protegê-lo com o sistema de logon criado na aula passada?

Experimente copiar o pacote "logon" do WProjeto8, feito na aula 11 para os pacotes de código fonte do projeto atual (WProjeto9). <u>Não se esqueça de criar as entradas necessárias no **web.xml**</u> pois, caso contrário, você não será capaz de usar os servlets /WLogon e /WLogoff.

Modifique o index.jsp para conter um formulário de login que redirecione para o servlet ClienteCreate em caso de sucesso e acrescente a linha de verificação de logon em todos os Servlets e JSPs do sistema!

Tente!

5. BIBLIOGRAFIA

QIAN, K; ALLEN, R; GAN, M; BROWN, R. **Desenvolvimento Web Java**. Rio de Janeiro: LTC, 2007.

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

Unidade 12: Web Services

Prof. Daniel Caetano

INTRODUÇÃO

Na maior parte deste curso foram estudados serviços desenvolvidos com o uso de tecnologias Servlet. Entretato, os Servlets nem sempre serão suficientes, isto é, nem sempre serão capazes de atender às nossas necessidades. Sendo assim, uma nova tecnologia torna-se necessária.

Nesta aula veremos uma tecnologia que supera algumas das limitações dos Servlets.

1. O QUE É UM WEB SERVICE?

Antes de entendermos o que é um WebService, é preciso entender... por que um WebService? Já não aprendemos a fazer serviços Java com Servlets?

Sim, já aprendemos a fazer serviços com Servlets. E os servlets possuem uma infinidade de vantagens sobre os WebServices, em especial quando executados em um container como o GlassFish, que permite acesso a todos os recursos do Java. Qual é o problema, então?

O primeiro problema com os Servlets é que, de forma “direta” eles só conseguem transferir requisições entre servlets que estejam em um mesmo equipamento. É possível contornar esse problema, mas não é tão simples.

Um outro problema é que a tecnologia Servlet é exclusiva da plataforma Java... e nem sempre os serviços de uma empresa serão todos em Java. Como assim? Alguém faz algo que não seja em Java? Sim, e muito! E a tendência não é que isso mude! Na verdade, a tendência é que daqui algum tempo (oxalá muito tempo) o Java é que deixe de ser usado, substituído por outras coisas. E aí? Como fica a interoperabilidade entre os Servlets e outros sistemas?

É possível garantir algum tipo de interoperabilidade, mas aquelas que envolvem a transferência de objetos dentro das requisições, por exemplo, certamente trarão enormes problemas. Em geral, toda a troca de dados binários pode levar a problemas potenciais por diferentes razões:

a) Computadores diferentes podem representar dados diferentemente na memória.
b) Linguagens diferentes vão certamente representar os dados de maneira diferente na memória.
c) Os tipos de dados de uma linguagem podem não estar disponíveis em outra linguagem.

Alguns podem se perguntar: "Mas os servlets não trocam requisições e respostas HTTP padrão? Como essa babel pode ocorrer?"

É aí que mora o desastre: o HTTP é um protocolo para **texto**. Tudo que for transmitido como texto tem boa chance de se manter legível entre diferentes plataformas, hoje ou no futuro. É

por isso que foi feita uma referência explícita a **troca de dados binários**, como imagens e objetos de programação. O protocolo HTTP não define uma forma clara e detalhada para especificar esse tipo e informação.

A solução foi criar uma padronização para especificar dados binários e estruturas de dados mais complexas, como um objeto. Essa padronização, criada pela Microsoft e IBM, por ser usada justamente para o desenvolvimento de **serviços** em arquiteturas de software orientadas a serviços, recebeu o nome de **SOAP: Service Oriented Architeture Protocol**.

Como a IBM e a Microsoft já se cansaram de reinventar a roda, o protocolo SOAP foi definido com base na especificação XML, permitindo a definição de tipos complexos. O protocolo SOAP define o formato de mensagens de requisições e de mensagens de resposta, que tem aparência como as indicadas a seguir:

SOAP Request

```
 <?xml version="1.0" encoding="UTF-8"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
    <S:Header/>
    <S:Body>
        <ns2:print xmlns:ns2="http://hello.me.org/">
            <nome>Fulano</nome>
        </ns2:print>
    </S:Body>
</S:Envelope>
```

SOAP Response

```
<?xml version="1.0" encoding="UTF-8"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
    <S:Body>
        <ns2:printResponse xmlns:ns2="http://hello.me.org/">
            <return>OlÃ¡, Fulano</return>
        </ns2:printResponse>
    </S:Body>
</S:Envelope>
```

Foge ao escopo deste curso detalhar o protocolo SOAP, mas é importante saber que a sintaxe dele obedece ao padrão XML e que ele é o caminho encontrado para **padronizar a comunicação** entre os Web Services (Serviços Web).

Ok, então um Web Service é como se fosse um servlet, só que a comunicação entre eles não ocorre por texto/binário puros pelo "HTTP", mas sim pelo envio de arquivos XML formatado segundo as regras do protocolo SOAP, através do HTTP?

ISSO...! Ou quase!

2. WEB SERVICES SÃO DINÂMICOS

O que significa que os Web Services são dinâmicos? E... que os Servlets são estáticos?

Vamos imaginar um exemplo. Imaginemos que estamos construindo um aplicativo que faz reserva em hoteis. Você especifica as características, ele procura o hotel mais barato que as atenda e faz a reserva. Ora, esse programa precisa se conectar com os sistemas dos diversos hoteis para poder fazer as verificações e reservas, correto?

É possível fazer isso com servlets até um determinado ponto: a lista de servidores de hoteis precisa ser definida manualmente, e você precisará entrar em contato com todos os desenvolvedores dos hoteis para obter o protocolo de cada sistema, isto é, como é que você deve solicitar as informações para cada sistema.

Isso muda nos WebServices? **SIM!**

Muda porque, primeiramente, existe um protocolo chamado **UDDI - Universal, Description, Discovery and Integration**. Esse protocolo é usado por um sistema que é uma espécie de "google" dos aplicativos Web, e ele existe para que um programa possa buscar por serviços, de diferentes origens, que estejam online naquele instante e possam responder a requisições. Se um hotel fecha, por exemplo, o UDDI deixará de informar aquele serviço como disponível. Se um novo hotel surge, suas informações são publicadas no UDDI.

Ok, já achei um novo serviço, mas como usá-lo? Também para isso existe um padrão de divulgação, chamado **WSDL: Web Services Description Language**. O WSDL descreve quais operações estão disponíveis, quais parâmetros enviar e o quê será recebido de volta.

Tanto o UDDI quanto o WSDL foram definidos também com base na sintaxe do XML. Um WSDL tem uma cara mais ou menos assim:

```
<!--
 Published by JAX-WS RI at http://jax-ws.dev.java.net. RI's version is JAX-WS RI 2.2.1-hudson-28-.
-->
—
<!--
 Generated by JAX-WS RI at http://jax-ws.dev.java.net. RI's version is JAX-WS RI 2.2.1-hudson-28-.
-->
—
<definitions targetNamespace="http://imcws/" name="IMCWSService">
—
<types>
—
<xsd:schema>
<xsd:import namespace="http://imcws/" schemaLocation="http://localhost:8080/IMCWSApp/IMCWSService?xsd=1"/>
</xsd:schema>
</types>
—
<message name="calcula">
<part name="parameters" element="tns:calcula"/>
</message>
—
<message name="calculaResponse">
<part name="parameters" element="tns:calculaResponse"/>
</message>
—
<portType name="IMCWS">
```

```
–
<operation name="calcula">
<input wsam:Action="http://imcws/IMCWS/calculaRequest" message="tns:calcula"/>
<output wsam:Action="http://imcws/IMCWS/calculaResponse" message="tns:calculaResponse"/>
</operation>
</portType>
–
<binding name="IMCWSPortBinding" type="tns:IMCWS">
<soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
–
<operation name="calcula">
<soap:operation soapAction=""/>
–
<input>
<soap:body use="literal"/>
</input>
–
<output>
<soap:body use="literal"/>
</output>
</operation>
</binding>
–
<service name="IMCWSService">
–
<port name="IMCWSPort" binding="tns:IMCWSPortBinding">
<soap:address location="http://localhost:8080/IMCWSApp/IMCWSService"/>
</port>
</service>
</definitions>
```

Feio, não? Muito feio mas... funciona!

3. QUAL O PROCEDIMENTO PARA USAR ISSO?

Tudo ocorre entre 3 elementos: o "**Solicitante do Serviço**", o "**Distribuidor de Serviços**" e o "**Provedor de Serviço**".

A. Primeiramente, uma aplicação que usará os serviços (o Solicitante do Serviço) consulta o chamado "Distribuidor de Serviços", que usa o protocolo UDDI para receber informações sobre o serviço desejado e devolver uma lista de serviços.

B. Com a lista de serviços, o "Solicitante de Serviço" solicita informações sobre como usar um serviço específico, o que ele recebe do "Distribuidor de Serviços" no formado "WSDL".

C. De posse do WSDL, o "Solicitante do Serviço" consegue contatar o "Provedor de Serviço" e, através do protocolo SOAP, envia uma requisição e, posteriormente, recebe uma resposta.

Tudo isso parece muito complicado!

E É complicado. Por sorte, o pessoal da Oracle/Sun e do NetBeans já pensou em tudo para nós, tornando a tarefa de criar um WebService bastante simples...

4. GERANDO UM WEBSERVICE NO NETBEANS

PASSO 1. Crie um novo projeto para uma Aplicação Web: "**Arquivo > Novo Projeto...**" e selecione "**Java Web > Aplicação Web**". Dê o nome de "**IMCWSApp**" a esta aplicação e finalize.

PASSO 2. Nos pacotes de código fonte, crie um pacote chamado "**imcws**".

PASSO 3. No pacote "**imcws**", clique com o botão direito e selecione "**Novo > Serviço Web**". Dê o nome de "**IMCWS**" para o serviço.

PASSO 4. O NetBeans irá criar uma nova pasta chamada "**Serviços Web**". Abra-a.

PASSO 5. Na pasta "**Serviços Web**", clique com o botão direito no **IMCWS** e selecione "**Adicionar Operação...**".

PASSO 6. No nome da operação indique "**calcula**". Como tipo de retorno, selecione "**double**".

PASSO 7. Acrescente dois parâmetros: "**peso**" e "**altura**", ambos do tipo "**double**". Finalmente, clique em "Ok".

PASSO 8. Edite o código do **IMCWS.java** para que o código do método "calcula" fique com o seguinte conteúdo:

```
double imc = peso / (altura * altura);
  return imc;
```

PASSO 9. Grave e, depois, clique com o botão direito no ícone do projeto, de nome **IMCWSApp** e selecione a opção "**Implantar**".

PASSO 10. Agora, vamos testar se está tudo ok com nosso serviço Web: na pasta "**Serviços Web**", clique com o botão direito em **IMCWS** e selecione a opção "**Testar Serviço Web**".

PASSO 11. O link "**WSDL File**" mostra o arquivo WSDL gerado pelo NetBeans. Se você indicar os valores para peso e altura nos campos e clicar no botão calcula, verá como resultado o valor calculado e também os arquivos SOAP enviados do cliente para o servidor (SOAP Request) e do servidor para o cliente (SOAP Response).

5. CONSUMINDO UM WEBSERVICE NO NETBEANS

Um WebService pode ser consumido a partir de qualquer aplicação Java. Neste tutorial, vamos construir um servlet que usa o WebService criado anteriormente para calcular o IMC. **NÃO** feche o projeto IMCWSApp!

PASSO 12. Crie um novo projeto para uma Aplicação Web: "**Arquivo > Novo Projeto...**" e selecione "**Java Web > Aplicação Web**". Dê o nome de "**IMCWSClienteApp**" a esta aplicação e finalize.

PASSO 13. Clique com o botão direito no ícone do novo projeto, de nome "**IMCWSClienteApp**" e selecione: "**Novo > Cliente para serviço web**".

NOTA: Se a opção não estiver disponível, vá em "**Novo > Outro...**" e depois selecione "**Serviços Web > Cliente para serviço web**".

PASSO 14. Na janela que vai aparecer, quando for solicitado que seja especificado o "WSDL" do serviço web, você pode fazer duas coisas: ou indicar como "**Projeto**" e apontar o nome de nosso outro projeto... ou, caso você não tenha acesso ao código do projeto, você pode indicar na terceira linha direto do endereço do WSDL do serviço, como, neste exemplo http://localhost:8080/IMCWSApp/IMCWSService?WSDL

PASSO 15. Agora, nosso sistema está pronto para usar nosso WebService, mas precisamos criar o sisteminha, que será composto de um formulário "index.jsp", que apresentará o formulário perguntando peso e altura, e um servlet chamado "**imc.java**". Vamos começar pelo **index.jsp**. Modifique o "index.jsp" para que tenha essa aparência:

index.jsp

```
-------------------------------------------------
<%--
   Document   : index
   Created on : 28/05/2011, 16:01:16
   Author     : djcaetano
--%>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>

<html>
   <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
      <title>Cálculo de Índice de Massa Corporal com Web Services</title>
   </head>
   <body>
      <h1>Digite suas informações!</h1>
      <form action ="imc">
         <p>Peso: <input type="text" name="peso"></p>
         <p>Altura: <input type="text" name="altura"></p>
         <input type="submit" value="Calcular!">
      </form>
   </body>
<html/>
```

PASSO 16. Agora é hora de criar o servlet. Criemos o pacote de nossa aplicação: clique com o botão direito em "**Pacotes de Código Fonte**" e selecione "**Novo > Pacote Java**". Dê o nome de "**imcapp**" para o pacote.

PASSO 17. Neste pacote, clique com o botão direito e crie o servlet "**imc**": "**Novo > Servlet**" e dê o nome de "**imc**", não se esquecendo de **adicionar a informação ao descritor**.

PASSO 18. Agora, **atenção**! Na área de projeto do NetBeans, abra a pasta "**Referências de Serviços Web**". Dentro dela haverá várias subpastas. Abra o caminho todo: "**Referências de Serviços Web > IMCWSService > IMCWSService > IMCWSPort > calcula**". Este último terá

uma bolinha vermelha, indicando tratar-se de uma funcionalidade.

PASSO 19. **Mais atenção** ainda! Clique e segure o botão esquerdo sobre esta bolinha e **arraste** o mouse até a área de código do servlet "imc.java", na região **após** o "processRequest". Isso fará aparecer o seguinte "método", que é um acesso ao WebService.

```
--------------------------------------------------
    private float calcula(float peso, float altura) {
       imcws.IMCWS port = service.getIMCWSPort();
       return port.calcula(peso, altura);
    }
--------------------------------------------------
```

PASSO 20. Vamos usar o método de acesso criado. Altere o método processRequest para receber os dados do formulário e chamar o WebService, da seguinte forma:

imc.java

```
--------------------------------------------------
  protected void processRequest(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
       response.setContentType("text/html;charset=UTF-8");
       PrintWriter out = response.getWriter();

         try {
                // Recebe valores do formulário
            double peso = Double.valueOf(request.getParameter("peso"));
                double altura = Double.valueOf(request.getParameter("altura"));
                if (altura == 0.0) altura = 1.0;

            // Calcula IMC pelo Web Service
                double imc = calcula(peso, altura);

            // Imprime resultado
                out.println("<html>");
                out.println("<head>");
                out.println("<title>Resultado do WebService IMCWS</title>");
                out.println("</head>");
                out.println("<body>");
                out.println("<h1>IMC: " + imc + "</h1>");
                out.println("</body>");
                out.println("</html>");
             } finally {
                out.close();
             }
    }
```

Pronto! Você criou seu primeiro Web Service e um programa que o usa!

Unidade 13: Java Persistence API

Prof. Daniel Caetano

INTRODUÇÃO

Na maior parte deste curso foi usada a persistência implementando o padrão DAO (Data Access Objects), um padrão criado pela Microsoft. O Java, entretanto, tem uma API padronizada para o uso de persistência independente de implementação, chamada Java Persistence API ou JPA.

Para o uso dessa API, é necessário escolhermos uma implementação; neste curso iremos utilizar a implementação TopLink, que já está presente no NetBeans. Com base na JPA e no TopLink, iremos implementar um pequeno cadastro, introduzindo as funcionalidades mais básicas do JPA.

1. CRIANDO UM BANCO DE DADOS DE LOCADORA COM JPA

Primeiramente precisamos criar um banco de dados.

PASSO 1. Antes de mais nada, vamos iniciar o servidor de Banco de Dados. Para isso, vá na aba **Serviços > Banco de Dados > JavaDB** e clique com o botão direito, selecionando a opção "**Inicializar Servidor**".

PASSO 2. Vamos criar agora o banco. Para isso, vá na aba **Serviços > Banco de Dados > JavaDB** e clique com o botão direito, selecionando a opção "**Criar Banco de Dados**". Na janela que vai aparecer, use os seguintes dados para criar o banco:

Nome: **locadora**
User: **nbuser**
Senha: **nbuser**

PASSO 3. A fim de facilitar nossa vida nos próximos passos, vamos dar um nome mais amigável para o banco de dados. Assim, mude o nome do link do banco para "**Locadora DB**" (isso pode ser feito pela opção "propriedades" ou clicando duas vezes, lentamente, no nome do link).

Com o banco de dados criado, agora vamos criar um projeto. Não se preocupe com as tabelas, o NetBeans as criará para nós, usando o JPA.

PASSO 4. Vamos criar o projeto. Para isso, clique no ícone novo projeto. **Selecione Java > Aplicativo Java**. Dê o nome de **Projeto5** para o projeto.

PASSO 5. Vamos criar o pacote para nossos arquivos de dados. Clicando na pastinha de código fonte, selecione a opção **Novo > Pacote...** E dê o nome de "**entidades**" para o pacote.

Vamos agora criar a classe de entidade que representa o filme.

PASSO 6. Vamos criar agora o nosso arquivo de entidade. Clicando com o botão direito no pacote “entidades”, selecione a opção **Novo > Classe da Entidade...** E a configure com o nome **Filme**. Clique no botão **Próximo**. Escolha como Biblioteca o “**Toplink Essentials (JPA 1.0)**” e como conexão o “**Locadora DB**”.

Observe que na pasta META-INF surgiu um arquivo chamado **persistence.xml**. Neste arquivo é configurada a persistência, indicando a “tradução” dos dados.

PASSO 7. Selecione o arquivo **Filme.java** para edição. Vamos adicionar alguns atributos à classe. Abaixo do **private Long id**, insira:

```
private String nome;
private int tempo;
```

PASSO 8. Como inserimos atributos, precisaremos criar os getters e setters. Clique com o botão direito e selecione a opção **Inserir código... > Getter e Setter**. Marque todos os atributos e clique no botão **Gerar**.

PASSO 9. Podemos ajustar os métodos **hashCode**, **equals** e **toString** manualmente, mas é mais fácil deixar que o NetBeans os ajuste por nós. Sendo assim, **apague** os métodos citados acima.

PASSO 10. Vamos recriar o equals e o hashCode. Clique com o botão direito e selecione a opção **Inserir código... > equals() e hashCode()**.Marque todos os atributos e clique no botão **Gerar**.

PASSO 11. Vamos recriar o toString. Clique com o botão direito e selecione a opção **Inserir código... > toString()**.Marque todos os atributos e clique no botão **Gerar**.

A entidade está pronta. Para podermos usar o JPA, entretanto, precisamos criar um elemento chamado JPA Controller, algo similar a um DAO.

PASSO 12. Vamos criar agora o nosso arquivo Controller. Clicando com o botão direito no pacote “entidades”, selecione a opção **Novo > Outro.** Agora selecione **Persistence > Classes do Controlador do JPA de classes de entidade** e clique em **Próximo**. Clique no botão **Adcionar tudo >>** e depois clique em **Próximo**. Clique em **Finalizar**.

Como vamos usar o JavaDB (Derby), precisamos adicionar o driver para o mesmo em nosso projeto. Isso só é necessário porque estamos trabalhando com um aplicativo Java tradicional (desktop). Caos estivéssemos criando uma Web Application com o GlassFish, o Derby estaria incluído automaticamente, já que faz parte do pacote do GlassFish.

PASSO 13. Vamos então inserir o driver do JavaDB. Na área de projeto, selecione a pasta “Bibliotecas” e clique com o botão direito do mouse. No menu, selecione a opção “**Adicionar Jar/Pasta**”. Adicione estes dois arquivos:

```
Program Files>GlassFish>javadb>lib>derby.jar
Program Files>GlassFish>javadb>lib>derbyclient.jar
```

Com tudo isso pronto, vamos agora usar nossa classe de entidade.

PASSO 14. Vá até o arquivo **Projeto5.java**. Dentro do método **main**, insira o seguinte código:

```
int i;
Filme f;
List<Filme> lista;

EntityManagerFactory emf;
emf = Persistence.createEntityManagerFactory("Projeto5PU");
FilmeJpaController jpa = new FilmeJpaController(emf);

f = new Filme();
f.setNome("Harry Potter");
f.setTempo(150);
jpa.create(f);

f = new Filme();
f.setNome("Senhor dos Anéis");
f.setTempo(170);
jpa.create(f);

lista = jpa.findFilmeEntities();
for (i=0; i<lista.size(); i++) {
        f = lista.get(i);
        System.out.println(f);
}

f.setTempo(180);
jpa.edit(f);                  // Acrescentar try/catch com ALT+ENTER

lista = jpa.findFilmeEntities();
for (i=0; i<lista.size(); i++) {
        f = lista.get(i);
        System.out.println(f);
}
```

PASSO 15. Esse código não compilará, visto que a linha **jpa.edit** pode causar erros de processamento. Sendo assim, envolva-a em um try~catch, usando o **ALT+ENTER**.

Tudo pronto? Vejamos se funcionou!

PASSO 16. Vá até a aba Serviços. Selecione "**Desconectar DB**" e depois "**Conectar DB**". No esquema **JavaDB**, clique com o botão direito em **Filme** e escolha **Visualizar Dados**.

2. COMO REALIZAR BUSCAS COM "WHERE"?

O JPAController criado pelo NetBeans contém alguns comandos padrão:

create(filme)	– cria elemento
edit(filme)	– atualiza elemento
destroy(id)	– remove elemento
findFilmeEntities()	– realiza buscas
findFilmeEntities(max,first)	– realiza buscas
findFilmeEntities(all, max,first)	– realiza buscas
findFilme(id)	– realiza buscas

Essas buscas são genéricas, isto é, não "filtram" por atributos específicos, como usualmente fazemos com o SQL. Para buscas com filtros, temos que criar um método no JPAController, incluindo um SELECT diferente. Observe o SELECT do findFilmeEntities(all, max,first)

```
Query q = em.createQuery("select object(o) from Filme as o");
```

Observe como isso se parece com um SQL comum, mas trata com objetos. Para poder filtrar, por exemplo, pelo nome "Harry Potter" podemos usar uma linha como a indicada abaixo:

```
Query q = em.createQuery("select object(o) from Filme as o where
o.nome LIKE 'Harry Potter'");
```

3. COMO EVITAR INSERIR ELEMENTOS REPETIDOS

Para evitar inserir elementos repetidos, deve-se sempre realizar uma busca específica antes, ou, supondo-se que a chave primária ID não é automática, tentar fazer um **edit** antes de um **create**.

Para facilitar esta tarefa, o método edit já retorna a exceção **NonexistentEntityException**, que indica exatamente quando uma entidade não existe e, portanto, precisa ser criada.