easy java Magazine

Edição 39

Entendendo a classe Object
A base de todas as classes do Java

Primeiros passos com a JPA
Conheça a principal API de persistência do Java

Desenvolva aplicações com JSch
Aprenda a transferir arquivos de forma rápida e segura

DEVMEDIA

ISSN 2179625-4

DEBUG NO ECLIPSE

Elimine os erros do seu código

DESENVOLVA PARA CORPORAÇÕES OU PARA O DIA A DIA COM JAVA

A linguagem Java está presente em complexos sistemas como em aplicativos mobile e desktop, servindo a milhares de pessoas. Inicie sua carreira e certifique-se no instituto que valoriza seu currículo. Faça Infnet!

FORMAÇÃO DESENVOLVEDOR JAVA

DOMINE O DESENVOLVIMENTO DE APLICAÇÕES ORIENTADAS A OBJETO COM **UML** E APRENDA TÓPICOS AVANÇADOS DE PROGRAMAÇÃO COMO O USO DE **SERVLETS, JSP, FRAMEWORKS WEB DENTRO DO PADRÃO MVC.**

CERTIFICAÇÕES:

- Oracle Certified Professional, Java SE 6 Programmer
- Oracle Certified Professional, Java EE 5 Web Component Developer

infnet TRAINING

Central de Matrículas: 21 2122-8800 | www.infnet.edu.br

Sumário

Edição 39 • 2014 • ISSN 2173625-4

EXPEDIENTE

Editor
Eduardo Spínola (*eduspinola@gmail.com*)

Consultor Técnico Davi Costa (davigc_08@hotmail.com)

Produção

Jornalista Responsável Kaline Dolabella - JP24185

Capa e Diagramação Romulo Araujo

Atendimento ao leitor

A DevMedia possui uma Central de Atendimento on-line, onde você pode tirar suas dúvidas sobre serviços, enviar críticas e sugestões e falar com um de nossos atendentes. Através da nossa central também é possível alterar dados cadastrais, consultar o status de assinaturas e conferir a data de envio de suas revistas. Acesse **www.devmedia.com.br/central**, ou se preferir entre em contato conosco através do telefone 21 3382-5038.

Publicidade

publicidade@devmedia.com.br – 21 3382-5038

Anúncios – Anunciando nas publicações e nos sites do Grupo DevMedia, você divulga sua marca ou produto para mais de 100 mil desenvolvedores de todo o Brasil, em mais de 200 cidades. Solicite nossos Media Kits, com detalhes sobre preços e formatos de anúncios.

Fale com o Editor!

É muito importante para a equipe saber o que você está achando da revista: que tipo de artigo você gostaria de ler, que artigo você mais gostou e qual artigo você menos gostou. Fique a vontade para entrar em contato com os editores e dar a sua sugestão!
Se você estiver interessado em publicar um artigo na revista ou no site Easy Java Magazine, entre em contato com o editor, informando o título e mini-resumo do tema que você gostaria de publicar:

EDUARDO OLIVEIRA SPÍNOLA
eduspinola.wordpress.com
@eduspinola / @Java_Magazine

summa
Technology + Business
Premium Partner

CRIANDO APLICAÇÕES INOVADORAS QUE VÃO MUDAR O MUNDO?

DESENVOLVA COM AS FERRAMENTAS QUE OS MELHORES DESENVOLVEDORES USAM: GIT, REDMINE, JENKINS, SONAR, MAVEN E MUITO MAIS.

A suite ToolsCloud é um conjunto integrado das melhores ferramentas para desenvolvimento de software do mercado.

Um ambiente pronto para usar e com suporte, que os desenvolvedores curtem e os gerentes se surpreendem.

- Aumenta a produtividade;
- Facilita a colaboração e comunicação;
- Maior visibilidade e acompanhamento;
- Melhor planejamento e mais agilidade.

Na suite ToolsCloud você tem a sua disposição:

- As ferramentas mais usadas, integradas e de uso imediato;
- Repositório de codigo (git e svn), integração contínua (Jenkins/Hudson), Gerência de pendências (Redmine), e mais Maven, Nexus, Sonar;
- Infraestrutura robusta, em ambiente de nuvem, suporte e segurança;
- Facilita práticas de agilidade e lean startups;
- Ambiente isolado, customizável e flexível;
- Colaboração garantida: wiki, fóruns de discussão, chat (IRC).

Acompanhe a Toolscloud:

toolscloud@toolscloud.com

twitter.com/toolscloud

ToolsCloud

toolscloud.com

Eclipse Debug: Conhecendo seus recursos – Parte 1

Ampliando sua caixa de ferramentas para melhorar a qualidade do seu código

Este artigo faz parte de um curso

Fique por dentro

Esse artigo é útil para quem quer aprender como encontrar erros de código em qualquer tipo de programa desenvolvido no Eclipse e como utilizar as ferramentas que estão disponíveis neste IDE para conseguir atingir este objetivo. Deste modo, através de exemplos, ele demonstra diferentes maneiras de resolver vários tipos de erros no código de um programa, quer sejam erros de exceções, quer sejam erros lógicos, mais difíceis de se detectar por não gerarem mensagens possíveis de se rastrear.

Antes de começarmos a falar sobre *debugging* de código, é importante falarmos sobre o responsável por sua criação, o *bug*. Mas afinal, o que é um *bug*? Em linguagem computacional, um *bug* é um erro num programa, ou sistema, que produz um resultado incorreto ou inesperado. Mas *bug*, em inglês, significa inseto. Então, qual a relação entre insetos com erros computacionais?

O termo *bug*, usado para descrever erros técnicos, remonta, pelo menos, do ano de 1878, pelas palavras do grande inventor Thomas Edison, mas é mais comumente lembrado e associado à Grace Hopper (uma analista de sistemas da Marinha dos Estados Unidos) que, juntamente com sua equipe, descobriu um erro computacional causado por um inseto (uma traça) que havia ficado preso em um dispositivo de controle de intensidade de corrente elétrica, num circuito do computador Mark II. Os operadores que o encontraram eram familiares com o termo (*bug*) de engenharia e guardaram o inseto com a nota: "Primeiro caso real de um *bug* a ser encontrado.". Daí em diante, quando havia erros, ou falhas computacionais, Grace Hopper e sua equipe atribuíam os mesmos a um possível inseto, ou *bug*. Este termo pegou e ficou até os nossos dias.

Agora que sabemos como surgiu o termo *bug*, fica mais fácil de entendermos de onde vem o termo *debugging*. No inglês, o "de" inserido no prefixo de uma palavra, pode indicar o seu antônimo, por exemplo: attached/detached (anexar/separar), compile/decompile (compilar/descompilar), etc. Dito isso, temos que a definição do termo *debugging*, ou depuração, em português, é o processo de localizar e remover *bugs* (ou erros) em código de uma linguagem computacional ou *hardware*. Neste artigo vamos focar na parte do *software*.

Existem diversas formas ou técnicas de se fazer o *debugging* de um programa, algumas mais simples e outras mais complexas. Sendo assim, vamos começar por um procedimento bastante simples para entender o conceito base e depois vamos avançando.

Para realizar o *debugging* de um programa, podemos utilizar o seguinte método:

1. Identificar: Primeiro, temos que identificar um problema que faz com que o programa não funcione da maneira esperada, causando erros;

2. Isolar: O segundo passo é identificar e isolar a fonte deste problema, ou seja, sabemos que existe um problema, mas em que parte(s) do programa ele ocorre?

3. Resolver: Por fim, consertar o código causador do problema e voltar para o passo um, até que não existam mais problemas.

Este é um processo necessário em praticamente todo o desenvolvimento de *software*. Quanto mais complexo é um programa,

maior a chance de existirem erros e de se ter a necessidade de fazer o *debugging* dos mesmos. Quando identificados vários problemas num mesmo programa, é aconselhável realizar a priorização destes, ou seja, ordenar os *bugs*, do mais prioritário para o menos prioritário, para que os mais problemáticos sejam resolvidos primeiro. Uma correta priorização pode diminuir bastante o tempo de correção dos erros, já que alguns destes erros podem causar outros em várias partes do código.

Estatisticamente, devemos saber que a grande maioria dos *bugs* é causada por pessoas, tanto no desenvolvimento do código quanto no *design* do programa, e muito raramente irá acontecer de existirem erros causados, por exemplo, por compiladores que produzem código incorreto, ou pelo *framework* com o qual se desenvolve (o que não deixaria de ser uma falha humana, na construção do compilador, ou do *framework*). Conforme a experiência demonstra, na maioria das vezes em que é encontrado um erro e este é dado como sendo da plataforma, do *framework*, ou mesmo da IDE, raras serão as vezes onde este erro não será, na verdade, do programador que desenvolveu o programa. Então, antes de atribuir um *bug* para um terceiro, devemos procurar da melhor maneira possível pela verdadeira causa do problema.

Não é incomum o fato de um programador passar mais tempo procurando e resolvendo *bugs* de seus programas do que desenvolvendo o código. Daí a importância de se compreender, o melhor possível, as ferramentas que ajudem a diminuir este custo. Tais ferramentas são conhecidas como *debuggers* e é sobre elas que vamos falar neste artigo.

Primeiro contato

Antes de começarmos a procurar por erros com a ajuda de um *debugger*, temos uma opção para fazer o *debugging* de maneira rápida e simples, sendo por vezes a única alternativa disponível, como acontece no caso das aplicações distribuídas, onde pode não ser possível a ligação de uma ferramenta de *debugging* com o programa em execução. Inserir instruções de log no código é um método relativamente simples de torná-lo depurável e, por isso, um bom planejamento do log a ser inserido no código pode evitar muitas dores de cabeça e ajudar em muito na procura das causas e resolução dos problemas.

Existem muitas ferramentas e *frameworks* para ajudá-lo a fazer o *logging* de um programa, sendo que, para a linguagem Java, duas das mais conhecidas e utilizadas são o **java.util.logging** e o bom e velho Log4j. Apesar de a primeira opção seguir melhorando consideravelmente a cada nova versão, o Log4j oferece uma série de funcionalidades ainda não existentes no **java.util.logging**. Uma das funcionalidades mais interessantes do Log4j é a flexibilidade do seu **PatternLayout**, que nos permite descrever praticamente o que quisermos ver no log em termos de valores de variáveis, datas e formatação dos dados (quer o log seja em arquivo, console, e-mail, etc.) através da definição de padrões descritos nos arquivos de configuração da ferramenta. Outra funcionalidade interessante e muito útil do Log4j é a facilidade de podermos enviar e-mails com a descrição de erros utilizando o **SMTPHandler**, onde podemos definir quais os erros que queremos que nos sejam enviados por e-mail e quais devem ser logados em outros meios, de forma a separarmos a informação essencial para entendermos o problema, de toda a informação necessária para executar sua correção no código.

Através da definição de um bom padrão de log na configuração do Log4j, podemos ter informações muito relevantes que nos ajudarão no primeiro passo do *debug*, ou seja, a identificação dos problemas. A **Listagem 1** apresenta um exemplo de um padrão de configuração.

Listagem 1. Exemplo de um padrão de configuração do Log4j.

```
log4j.appender.logFile.layout.ConversionPattern=[%d{yyyy-MM-dd HH:mm:ss}]
[%-5p] %c{1}|%M:%L - %m%n
```

Este exemplo mostra uma instrução de um padrão de logging para o Log4j, que descreve a forma como cada linha de log será escrita em um arquivo. Neste caso, este padrão nos diz que cada linha deverá mostrar: a data de criação da linha do log, a prioridade do evento (que pode ser uma entre as opções: OFF, FATAL, ERROR, WARN, INFO, DEBUG, TRACE, por ordem do mais prioritário para o menos prioritário), o nome da classe, seguido do nome do método e o número da linha onde foi gerada a informação e, por fim, a mensagem informativa que escrevemos no código, no momento da criação do *log*, com aquilo que queremos ver, quer sejam valores estáticos, ou valores de variáveis, seguida de uma nova linha (*ENTER*). Como as configurações do sistema de *logging* estão fora do escopo deste artigo, para mais detalhes sobre o **PatternLayout** do Log4j, veja o endereço da página da Apache que se encontra na seção **Links**. A **Listagem 2** apresenta o exemplo de um resultado do output do *log* que utiliza este padrão.

Listagem 2. Exemplo de saída de um log.

```
[2014-01-01 00:18:25] [INFO ] Integrator|main:36 - START
[2014-01-01 00:18:25] [DEBUG] Integrator|main:62 - Main frame executing on its
own thread.
[2014-01-01 00:18:25] [INFO ] Integrator|main:63 - END
[2014-01-01 00:18:29] [INFO ] Resources|setLocale:64 - Got new resource bundle
for language-country: en-US
[2014-01-01 00:18:29] [INFO ] MainFrame|init:53 - Initializing....
[2014-01-01 00:18:29] [INFO ] MainFrameMenuBar|init:74 - Creating Frame Menu
Bar....
[2014-01-01 00:18:29] [DEBUG] MainFrameMenuBar|init:89  - File menu created.
[2014-01-01 00:18:29] [DEBUG] MainFrameMenuBar|init:112 - View menu created.
[2014-01-01 00:18:29] [DEBUG] MainFrameMenuBar|init:125 - Help menu created.
[2014-01-01 00:18:29] [INFO ] MainFrameMenuBar|init:133 - Frame Menu Bar
created.
[2014-01-01 00:18:29] [DEBUG] MainFrame|init:65 - Menu bar created.
[2014-01-01 00:18:29] [DEBUG] MainFrame|init:69 - Main panel created.
[2014-01-01 00:18:29] [INFO ] TabPanelSettings|init:44 - Initializing....
[2014-01-01 00:18:29] [DEBUG] TabPanelSettings|loadEnvironment:69 - Loading
environment settings....
[2014-01-01 00:18:29] [DEBUG] TabPanelSettings|loadEnvironment:75 - Environ-
ments loaded!
[2014-01-01 00:18:29] [INFO ] TabPanelSettings|init:62 - Finished panel
initialization.
```

Apesar de ser um padrão que nos dá muita informação relevante, este não seria aconselhado para programas com muito *log* por ser demasiadamente pesado, ou seja, é algo que requer demasiado processamento por linha, mas para casos simples, como no deste exemplo, que é o de uma aplicação criada com Java Swing, que necessita de pouca informação de log para sabermos sobre todos os acontecimentos importantes e relevantes que acontecem, é ótimo para se conseguir fazer um bom rastreamento de qualquer tipo de problema.

Com um log detalhado como este, não deve ser difícil ultrapassar os dois primeiros pontos de *debugging* descritos anteriormente: a identificação e o isolamento do problema. Por isso, elabore uma boa estratégia para o *logging* de sua aplicação e grande parte do problema será resolvido de forma mais rápida e simples.

Quanto à decisão sobre que biblioteca usar, particularmente, uso um processo de escolha bastante simples. Se o programa que for desenvolver for bastante simples e precisar de pouca informação de *logging*, use o **java.util.logging**, que já está incluído na biblioteca do Java e você não precisará fazer qualquer configuração extra ou adicionar qualquer JAR. Por outro lado, se for desenvolver uma aplicação grande e complexa, se pretende flexibilidade no formato do *logging*, enviar alertas por e-mail, fazer posteriores alterações de configuração de *logging* de forma simples e sem alterações no código, etc., então o Log4j é o que você procura. É claro que esta é uma questão de preferência. Se encontrar uma ferramenta ou *framework* com que se sinta mais à vontade, ou que ache mais útil, ou intuitiva, use-a.

Não vamos nos preocupar aqui com os detalhes da configuração do Log4j, porque este não é o nosso objetivo e porque este assunto daria matéria suficiente para um novo artigo. No entanto, se quiser obter mais informações sobre o assunto, a documentação da Apache é excelente, e pode ler mais sobre o mesmo em sua página web, que se encontra na seção **Links**.

Quando o logging não é suficiente

Olhamos para o log do programa, procuramos pelo que poderia ser o motivo do erro, perdemos bastante tempo analisando cada linha, mas, por algum motivo, não conseguimos ainda entender o porquê de certo valor no *output*, ou mesmo o porquê de certos valores intermediários, mostrados no log do programa. Para podermos entender o que está acontecendo, precisamos de mais dados, e para conseguir estes dados é possível alterar o log para nos passar cada um dos valores que achemos necessário saber. No entanto, isso seria um trabalho moroso e chato! Teríamos que alterar o código, compilar e o executar novamente para procurar por situações bem específicas.

E se ainda assim continuarmos precisando de mais dados e quisermos ver o valor de outras variáveis, teríamos que seguir o mesmo processo novamente. Necessitamos então de algo mais rápido e poderoso do que isso, pois queremos diminuir o tempo necessário para conseguir mais detalhes e dados que transitam na execução do programa, e o Eclipse tem aquilo que precisamos.

O depurador do Eclipse

O Eclipse, através do projeto JDT (*Java Development Tools*, em português: Ferramentas de Desenvolvimento Java), nos proporciona um depurador que fornece todas as funcionalidades padrão de depuração:

- Inicializar uma JVM em modo normal ou depuração;
- Ligar o código de um programa a uma JVM em execução;
- Avaliar código Java interativamente (passo a passo);
- Definir pontos de interrupção (*breakpoints*) para inspecionar valores de variáveis;
- Suspender e retomar *threads*;
- Carregar classes de forma dinâmica (quando suportado pela JVM).

Para as demonstrações deste artigo, usaremos o Eclipse Kepler Java EE (4.3.1).

O debugger do Eclipse nada mais é que uma série de *plug-ins*. E para termos acesso a essas ferramentas, o Eclipse tem uma perspectiva especial para depuração, que nos dá um conjunto pré-configurado de visões. Para mudar para a perspectiva de debugging, existem duas opções. Uma delas é iniciar a aplicação em modo *debugging*. Isso faz com que o conjunto de visões mude automaticamente, sendo que na primeira vez será perguntado ao programador se ele deseja fazer esta mudança toda vez que a aplicação entrar neste modo de execução. A **Figura 1** demonstra como alterar a perspectiva dessa forma.

A segunda opção é selecionar, no canto superior direito do IDE, onde existe uma série de perspectivas disponíveis, a opção *Debug*, como demonstra a **Figura 2**.

Se você não encontrar a perspectiva de debug, é possível adicioná-la selecionando o primeiro ícone da figura. Isso também pode ser feito através do menu: *Window* > *Open Perspective*.

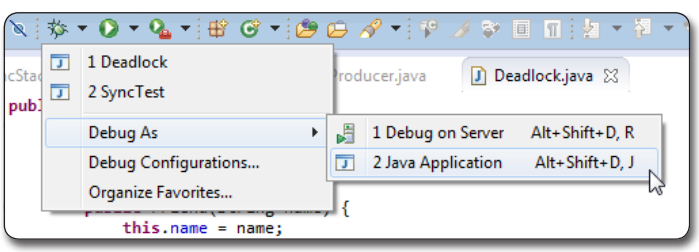


Figura 1. Iniciando a aplicação em modo Debug

Figura 2. Perspectivas do Eclipse

A perspectiva de debug

Como já foi dito, uma perspectiva é um conjunto agrupado de visões. Dependendo da tarefa que estivermos realizando em determinado momento, iremos nos preocupar com a visualização de diferentes dados do nosso programa, ou seja, iremos querer ter um outro ponto de vista (ou perspectiva) sobre aquilo que estamos fazendo.

Quando estivermos desenvolvendo uma aplicação, as principais visões serão as do código sendo escrito, a visão dos erros de compilação do código, de tarefas por terminar, etc. A perspectiva de depuração, por sua vez, mostra um conjunto de visões diferentes do existente na perspectiva de desenvolvimento, facilitando o processo de depuração. A **Figura 3** apresenta essa perspectiva no seu modo mais simples, padrão do Eclipse. Essa figura foi enumerada para que fosse mais fácil explicar as visões, analisadas a seguir:

• O canto superior esquerdo da imagem, indicado com o número 1, mostra a visão de debug, que nos indica qual a JVM onde estamos executando nosso programa, as classes que estão em execução, cada uma de suas *threads* e a *stack frame*, que basicamente diz onde estamos no programa (qual a classe e a linha de execução);

• O canto superior direito, indicado com o número 2, mostra a visão das variáveis, que irão aparecer conforme a linha que paramos no código no modo *debug*, ou seja, poderemos ver os valores das variáveis utilizadas pelo método onde se encontra parado o *debugger*. Além desta, há a visão dos *breakpoints*, com cada um dos *breakpoints* que marcamos em nosso código, e a visão de expressões, que são instruções de código que podemos escrever em tempo de *debug* para determinar os seus valores correntes;

• O centro da imagem, com a indicação do número 3, mostra o editor de código Java. É a mesma visão disponível na perspectiva Java. Através desta visão podemos ver os nossos *breakpoints* marcados na barra lateral esquerda e a linha corrente de execução, que fica realçada com uma cor diferente no código;

• E por fim, a parte inferior da imagem, indicada com o número 4, mostra a visão do console, com o *output* do programa, quer sejam *logs* definidos com a utilização de uma *framework* (como o caso do Log4J), quer seja o *output* gerado pela chamada do método **print()** do **System.out**.

Essa é a perspectiva padrão, a partir da qual você pode adicionar outras visões, caso avalie como necessário. Como um complemento, alguns desenvolvedores gostam de adicionar a visão de *outline* e a das *tasks*. Já que hoje em dia temos monitores maiores, podemos aproveitar esse espaço extra, mas convém não exagerar para que possamos focar no que é mais importante.

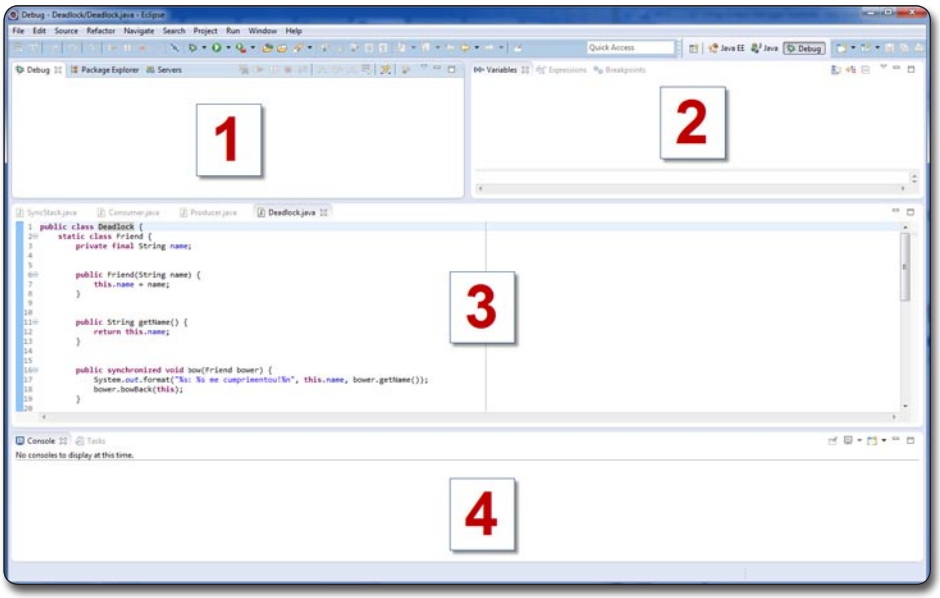


Figura 3. Perspectiva de depuração

```
16        public synchronized void bow(Friend bower) {
17            System.out.format("%s: %s me cumprimentou!%n", this.name, bower.getName());
18            bower.bowBack(this);
19        }
20
```

Figura 4. Adicionando breakpoints

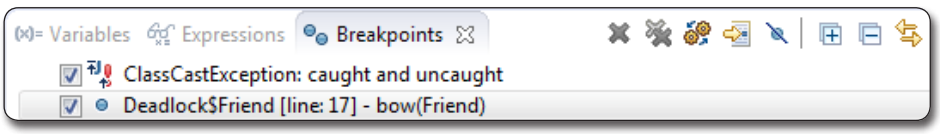


Figura 5. Visão de breakpoints com o breakpoint adicionado

Adicionando breakpoints

Já vimos como executar nossos programas em modo *debug* e como mudar para a perspectiva de *debug*, então só nos resta agora adicionar *breakpoints* para podermos controlar a execução passo a passo de um programa e aprender a usar as ferramentas de depuração. Para adicionar *breakpoints*, basta fazer um duplo clique na barra esquerda da visão do editor Java, na linha de código onde queremos parar. Isso fará com que apareça uma pequena bola azul na mesma barra, que confirma que o *breakpoint* foi adicionado com sucesso. A **Figura 4** mostra um *breakpoint* adicionado na linha 17 de um do código de exemplo.

Temos que ter em atenção que o *breakpoint* só será adicionado se for inserido numa linha de código com instruções, ou seja, não é possível inserir *breakpoints* em linhas em branco ou comentários, sendo que se a linha em branco (ou comentário) estiver dentro de um método, o *breakpoint* será adicionado na próxima instrução válida dentro deste método e se estiver fora de um método, o *breakpoint* não será adicionado. Da mesma forma, se um *breakpoint* for inserido na declaração de um método, o *debugger* só irá parar imediatamente na primeira linha executável dentro deste método.

Depois de executado o passo anterior com sucesso, se voltar à visão de *Breakpoints*, poderá notar lá o novo *breakpoint* adicionado. Esta visão será muito útil quando tiver vários *breakpoints* no código e pretender ir a um específico, ou então quando quiser remover *breakpoints* adicionados. A **Figura 5** demonstra esta visão.

Experimente as opções proporcionadas por esta visão, clicando em cada um dos ícones no canto superior direito para se habituar às funcionalidades oferecidas. O Eclipse mostra um *tool tip* quando colocamos o mouse por cima de um ícone que explica exatamente qual a função de cada opção. Nessa janela, as funcionalidades mais utilizadas serão:

- Para remover o *breakpoint* selecionado;
- Para remover todos os *breakpoints*;
- Para mostrar o arquivo que contém o *breakpoint* selecionado;
- Para desabilitar temporariamente todos os *breakpoints*.

Controlando a execução passo a passo do programa

Agora que já temos *breakpoints* adicionados em nosso código para que o *debugger* pare na(s) linha(s) desejada(s), podemos controlar a execução do programa passo a passo. Para isso, existem quatro teclas de atalho, detalhadas na **Tabela 1**, que queiramos ou não, decoraremos com o tempo, pois serão as mais utilizadas no processo de depuração.

Estes são os atalhos mais importantes para fazermos a depuração no código de um programa. Existem outras opções, que veremos mais à frente, mas que serão usadas mais esporadicamente. Na visão de Debug você poderá ver ícones com estas mesmas funcionalidades, como demonstra a **Figura 6**, que podem ser utilizados em vez das teclas de atalhos.

Atalho	Descrição
F5	Executa a linha de código selecionada e vai para a próxima linha de execução no programa. Se a linha selecionada for uma chamada a um método, o debugger entra no método e continua a depuração, adicionando uma nova stack frame em cima da stack frame corrente (vamos falar sobre as stack frames mais adiante).
F6	Excuta a linha de código corrente e para na linha seguinte. Basicamente, faz o mesmo que o F5, mas não executa as chamadas aos métodos.
F7	Termina a execução do método corrente e retorna ao método que fez a chamada (caller) limpando a stack frame criada pelo método chamado.
F8	Continua a execução normal do programa até o seu término, ou até que encontre o próximo breakpoint.

Tabela 1. Atalhos para funções de depuração

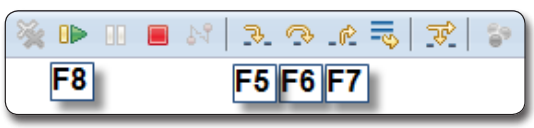


Figura 6. Controles do debugger

Debugging na prática

Neste momento, já vimos tudo o que é preciso para fazer uma depuração simples no código de um programa. Então vamos colocar o conhecimento em prática. Para um primeiro exemplo, foi construído um método que conta a maior sequência de um mesmo valor que ocorre em um array e nos diz qual é esse valor. Para que possamos acompanhar bem o exemplo, a **Listagem 3** contém o código completo do programa.

Listagem 3. Exemplo de contador de sequências.

```
01 public class Sequences {
02
03  static int longestSequence(int[] array) {
04   int length = array.length;
05
06   if (length == 0) {
07    return -1;
08   }
09
10   if (length == 1) {
11    return array[0];
12   }
13
14   int curSize = 1;
15   int maxSize = 0;
16   int longest = -1;
17
18   for (int i = 0; i < length; i++) {
19    if (array[i] == array[i - 1]) {
20     curSize++;
21
22     if (curSize > maxSize) {
23      maxSize = curSize;
24      longest = array[i];
25     }
26    }
27    else {
28     curSize = 1;
29    }
30   }
31
32   return longest;
33  }
34
35
36  static void testLongestSequence() {
37   int[] intArray = { 6, 9, 1, 2, 2, 2, 1, 14, 8, 10, 8, 12, 3, 2, 1, 3, 6, 2, 2, 6, 9, 6, 6,
38    6, 9, 1, 2, 2, 1, 14, 1, 1, 1, 1, 4, 4, 4, 4, 8, 8, 8, 8, 8, 8, 2, 3, 4, 2, 1, 453,
39    45, 55, 5, 1, 1, 1, 1, 1, 3, 4, 5, 6, 6, 6, 6, 6, 4, 6, 4, 21, 3, 4, 3, 6, 7, 8, 7,
40    78, 78, 87, 87, 87, 4, 5, 4, 2, 3, 1, 22, 2, 2, 2, 34, 4, 5, 5, 5, 12, 6, 8, 87, 9,
41    9, 6, 4, 4, 45, 5, 65, 4, 2, 80, 87, 8, 8, 90, 90, 45, 34, 2, 1, 21, 21, 3, 43, 43,
42    6, 78, 650, 453, 54, 453, 231, 7, 8, 98, 0, 9, 9, 9, 0, 9, 0, 0, 0, 0, 87, 7, 7, 8 };
43   System.out.println("Longest sequence: " + longestSequence(intArray) + "\n");
44  }
45
46
47  public static void main(String[] args) {
48   testLongestSequence();
49  }
50 }
```

O programa é simples para que possamos focar no que nos interessa, o *debugger* do Eclipse. Como pode ser verificado, o código tem um pequeno erro, básico, mas que acontece muitas vezes e que nos gera a seguinte exceção:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: -1
        at Sequences.longestSequence(Sequences.java:19)
        at Sequences.testLongestSequence(Sequences.java:43)
        at Sequences.main(Sequences.java:48)
```

Quem já programa há algum tempo na linguagem Java (ou em outra linguagem orientada a objetos) vai saber imediatamente

que esta exceção, como o próprio nome diz (**ArrayIndexOutOfBoundsException**), indica que tentamos acessar uma posição de um *array* com um índice que está fora dos seus limites.

O *output* gerado pela exceção nos devolve uma *stack trace*. Para os que ainda não estão familiarizados com o termo, uma *stack trace* nada mais é que uma lista com os nomes de todos os métodos que foram chamados na execução do programa, a partir do momento que o erro aconteceu.

Quando olhamos para a *stack trace* de uma exceção, a melhor abordagem para começarmos a procurar pelo erro é olhar para os nomes das classes que nos pertencem, ou seja, que nós escrevemos, e não nos interessar (pelo menos por agora) pelos métodos que estão dentro de arquivos JAR de terceiros ou de frameworks, começando pela última linha da pilha (*stack*) que, no caso do exemplo, é "**at Sequences.longestSequence(Sequences.java:19)**". Como este exemplo é bastante simples, todas as linhas da *stack trace* nos pertence, sendo chamadas dentro do mesmo objeto **Sequences**.

Graças ao IDE, podemos clicar no nome da classe seguida pelo número da linha e acompanhar o que aconteceu. Seguindo a ordem de chamadas (que é de baixo para cima), podemos ver que:

- **Sequences.java:48** – O método **testLongestSequence()** é chamado a partir do método **main()**;
- **Sequences.java:43** – O método **testLongestSequence()** chama o método **longestSequence(int[] array)**;
- **Sequences.java:19** – O método **testLongestSequence()** tem a expressão **if (array[i] == array[i - 1]) {**, que gera a exceção **java.lang.ArrayIndexOutOfBoundsException** com o argumento "-1".

Vamos então adicionar um *breakpoint* na linha 19, que é a linha onde o *stack trace* nos diz que foi onde o erro aconteceu e executar o programa em modo de *debug*. A **Figura 7** mostra a vista do editor Java na perspectiva de depuração.

Se olharmos para a visão de *debug* (primeira visão no canto superior esquerdo), poderemos ver qual o JRE que estamos

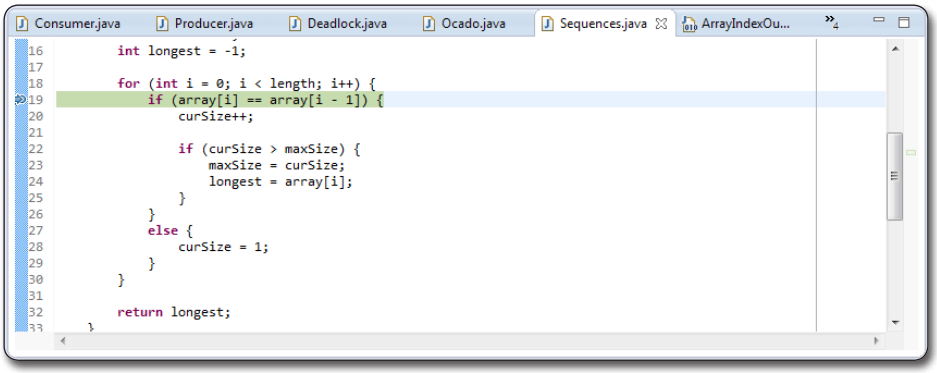


Figura 7. Debugger com breakpoint na linha 19 da classe Sequences

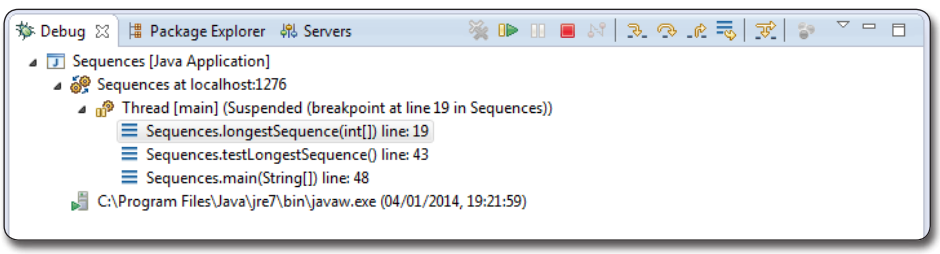


Figura 8. Debug view

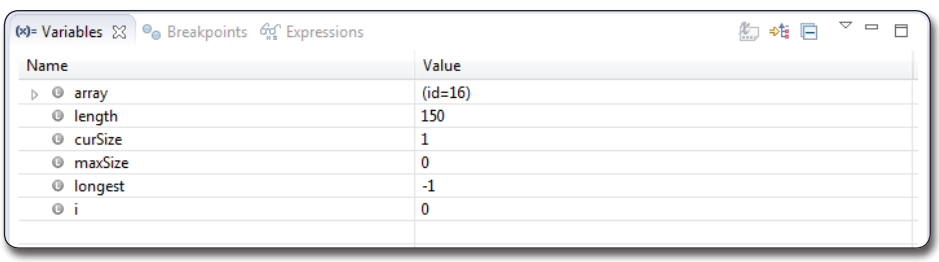


Figura 9. Visão das variáveis

usando para executar nosso programa (neste caso, o JRE 7), as *threads* do programa (no nosso caso é apenas uma, a *thread* "main") e a *stack frame*, com os nomes das classes (no exemplo, existe apenas a classe **Sequence**), métodos (no exemplo, os métodos **main()**, **testLongestSequence()** e **longestSequence()**) e as linhas de execução do código do programa (no exemplo, linhas 48, 43 e 19). Como só temos uma *thread*, essa é a que se encontra suspensa para análise, parada na linha 19, no *breakpoint* que inserimos, conforme ilustra a **Figura 8**.

Olhando para a *stack frame*, ainda na visão de *Debug*, podemos perceber que temos exatamente as mesmas linhas que nos foram mostradas no *stack trace* da exceção que tivemos no resultado da execução do programa, ou seja, por ordem, as linhas 48, 43 e 19 da classe **Sequences**. Daí ter sido dito logo no início do artigo que um bom log feito no programa pode resolver o problema de maneira mais rápida, pois os dados do mesmo, junto com os dados do *stack trace*, podem ser o suficiente para se conseguir fazer a identificação do problema e de sua causa, facilitando na solução do mesmo.

Já sabemos onde o programa está parado, e agora precisamos saber qual o seu estado atual. Se olharmos para a visão de variáveis, podemos ver todas as variáveis que estão sendo usadas no método **longestSequence()** da classe **Sequence**, selecionado no *stack frame* da visão de *Debug*. Veja a **Figura 9**.

Experimente selecionar as outras *frames* da *stack frame* na visão de *Debug*, e assim poderá ver as variáveis usadas por cada um dos outros métodos que selecionar. Por exemplo, se selecionar o método **main(String[] args)** na *stack frame*, pode-

rá ver o valor da variável **args** (um array vazio de **String**) e, do mesmo modo, se selecionar o método **testLongestSequence()** na *stack frame*, poderá ver o valor da variável **intArray**. Isso é bastante útil para saber o estado de cada uma das variáveis no momento das chamadas de cada método.

Neste ponto já podemos concluir qual foi a causa do erro. Olhando para os valores das variáveis usadas no método **longestSequence()**, na linha 19, conforme podemos ver na **Figura 9** (selecionando a respectiva *frame* na visão de *Debug*), temos que o valor da variável **i** é **i = 0**. E nesta mesma linha de código, tentamos acessar ao *array* da seguinte forma: **array[i - 1]**. Logo, é neste ponto que estamos tentando acessar a posição -1 de um *array*, o que, por ser uma posição inválida, nos gera a exceção **ArrayIndexOutOfBoundsException**.

Mas antes de corrigirmos o código, vamos avançar um pouco mais para usarmos o conhecimento adquirido e entender um pouco melhor o funcionamento do *debugger*. Se clicarmos na tecla *F6*, para darmos um passo na execução do programa, vamos gerar a exceção e não vamos ver muita coisa. Então, vamos fazer um *step into*, clicando no *F5* e olhar mais uma vez para o *stack frame* na visão de *Debug*. Teremos o que podemos ver na **Figura 10**.

Selecionando cada uma das linhas da *stack frame* e olhando para a visão de variáveis para ver o estado do programa em cada passo, podemos concluir que:

- O método **main()** foi chamado na linha 48 com um array de argumentos vazio;
- O método **longestSequence()** foi chamado na linha 43, com um array de números inteiros como argumento;
- O método **longestSequence()** gerou um **ArrayIndexOutOfBoundsException()** na linha 19;
- Foi inicializado um objeto do tipo **ArrayIndexOutOfBoundsException** com o argumento -1;
- O objeto **ArrayIndexOutOfBoundsException** chamou o super construtor **IndexOutOfBoundsException()** com o mesmo argumento;
- O objeto **IndexOutOfBoundsException** chamou o super construtor **RuntimeException()** com o mesmo argumento;
- O objeto **RuntimeException** chamou o super construtor **Exception()** com o mesmo argumento;
- Por fim, o objeto **Exception**" chamou o super construtor **Throwable()**, na linha 264, com o mesmo argumento ("-1"), e o super construtor preencheu o *stack trace* e a mensagem da exceção, conforme demonstrado na **Listagem 4**.

Listagem 4. Construtor da classe Throwable.

```
public Throwable(String message) {
  fillInStackTrace();
  detailMessage = message;
}
```

Entendeu agora como foi gerada a exceção e como foi preenchido o *stack trace*? Se continuar clicando agora no *F6*, vai ver a pilha da *stack frame* ir removendo cada uma das frames até que seja lançada a exceção. Em seguida o *debugger* se desconecta da JVM e por fim é terminada a execução da mesma.

Para quem ainda estiver confuso com a descrição dos acontecimentos na *stack frame*, faça o seguinte: abra o código da classe **ArrayIndexOutOfBoundsException** (*Ctrl+Shift+T* e escreva o nome da classe) e em seguida pressione a tecla *F4* para abrir a janela de hierarquia do tipo. Feito isso, poderá ver a mesma imagem que está representada pela **Figura 11**.

Agora compare essa figura com a **Figura 10**. Como você pode observar, temos a mesma hierarquia que é demonstrada na sequência das chamadas dos métodos (super construtores) que podemos ver na *stack frame* na visão de *debug*. E não poderia ser de outra forma, pois estamos falando dos mesmos objetos.

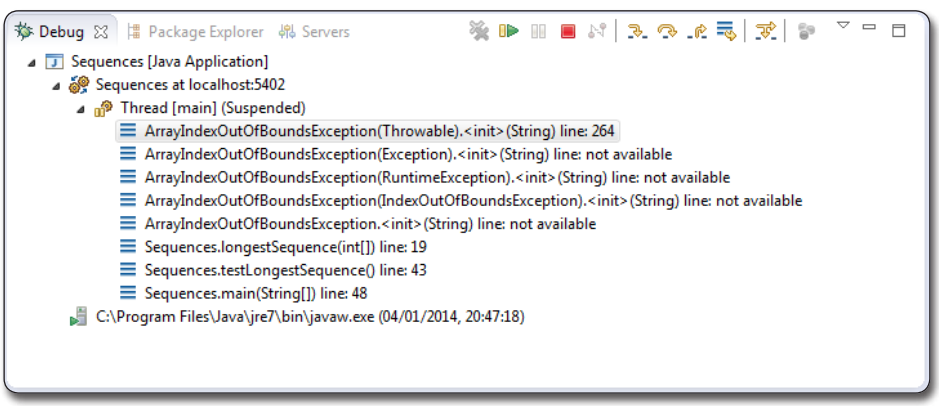


Figura 10. Stack frame na vista de Debug

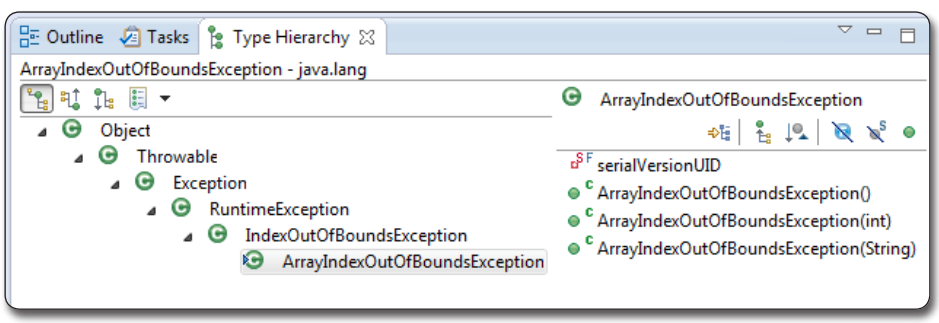


Figura 11. Hierarquia do tipo ArrayIndexOutOfBoundsException

Uma configuração muito útil

No caso de você tentar abrir uma classe que pertence a um JAR da API do Java, como foi no caso da classe **ArrayIndexOutOfBoundsException**, pode acontecer de ser exibida a mensagem de erro "*Source not found*" no editor Java. Isso acontece porque o IDE não sabe onde está o código fonte relativo à classe que tentou abrir. Por vezes pode ser muito útil visualizarmos

o código fonte destas classes, quer seja para aprofundar nosso conhecimento, quer seja por mera curiosidade.

Para corrigir este problema, execute os seguintes passos:

1. Clique no botão que aparece no editor Java que diz *Edit Source Lookup Path...*;
2. Na janela seguinte, selecione a opção *Add...* para adicionar um novo arquivo;
3. Na próxima janela que aparecer, selecione *External Archive* e em seguida, clique em *Ok*;
4. Selecione o arquivo *src.zip* no diretório de instalação do JDK, por exemplo: *C:\Program Files\Java\jdk1.7.0_10* e clique em *Ok*.

As **Figuras 12** e **13** demonstram os passos que deve seguir para solucionar este problema.

Pronto, agora já pode inspecionar o código fonte de todas as classes da API do Java que está usando e tirar suas dúvidas.

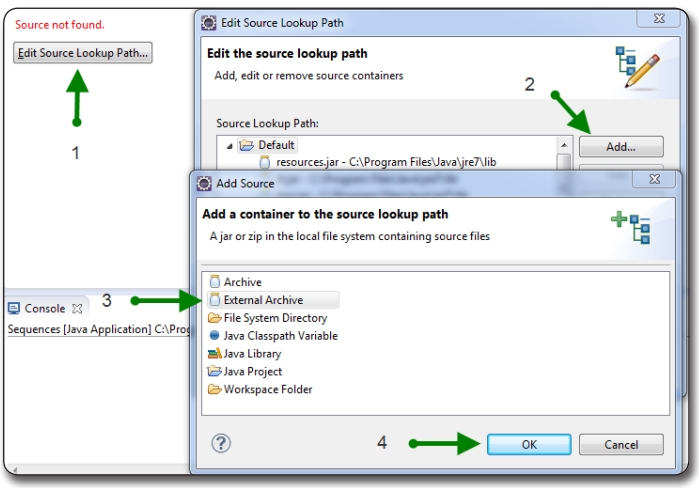


Figura 12. Anexando o código fonte do framework para visualização no Eclipse

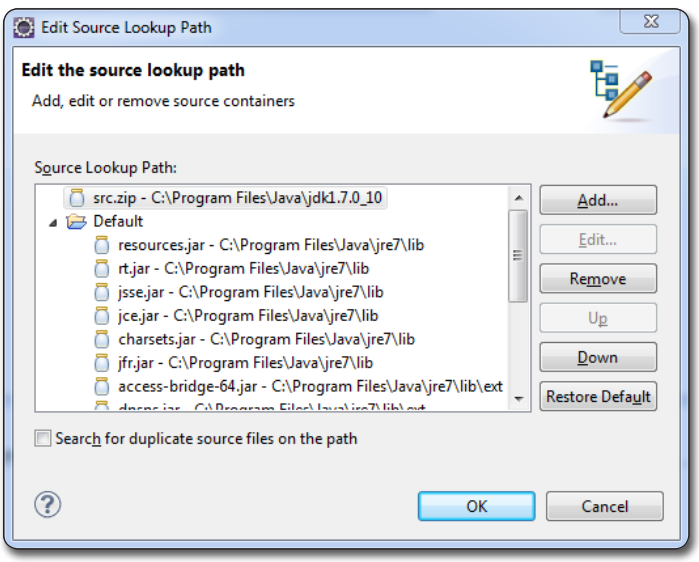


Figura 13. Quadro final na anexação do código fonte

Um novo exemplo

Para apresentarmos um novo exemplo, continuaremos usando o mesmo código, mas corrigindo o erro identificado anteriormente. Então, depois de encontrado e corrigido o erro, alterando o ciclo **for** da linha 18 do código para que inicialize a variável **i** com o valor de **1** em vez de **0** (ver **Listagem 5**), podemos avançar com mais um exemplo útil do *debugger* do Eclipse.

Listagem 5. Correção do código contador de sequências.

```
1 public class Sequences {
 ...
18    for (int i = 1; i < length; i++) {
```

Se executar novamente o programa, verá que o *output* do exemplo é: *Longest sequence: 8*.

Agora imagine que precisamos saber quais são todas as maiores sequências, por ordem crescente, que existem neste array, para confirmar se os valores batem certo com os resultados esperados. Contar cada sequência e anotar à mão não parece ser algo rápido de se fazer, até porque o *array* poderia ser infinitamente maior. Poderíamos também tentar alterar o código para nos retornar os resultados pretendidos, o que, neste caso, não seria algo difícil de fazer, devido a ser um exemplo pequeno e simples.

Mas vamos mais uma vez usar o método mais simples e fazer um novo *debug* do nosso pequeno programa.

Em primeiro lugar, precisamos saber onde inserir o *breakpoint*. Caso tenhamos interesse em conhecer todas as maiores sequências, por ordem crescente, que existem no array, quer dizer que queremos os valores da variável **longest**. Então vamos inserir o nosso *breakpoint* na linha 24 do código, onde é feita a atribuição do maior valor de sequência para esta variável.

Como estamos falando de uma linha de código que se encontra dentro de um ciclo **for**, sempre que pressionarmos na tecla *F8* vamos continuar a execução normal do programa até pararmos no mesmo *breakpoint* na linha 24, ou seja, na instrução **longest = array[i];**. Se fizermos isso, ainda teremos que pressionar uma vez no *F6*, para cada ciclo, para que seja atribuído o valor da variável **longest** e assim a possamos ver na visão de variáveis. Lembre-se que quando paramos numa linha específica do código, esta linha ainda não terá sido executada pela JVM enquanto não dermos a ordem para continuar a sua execução.

Para eliminar a necessidade de mais uma instrução para sabermos o valor da variável na linha selecionada, podemos inserir um pequeno *hack*. Como já vimos, um *breakpoint* só para em linhas de código com instruções, não para em linhas em branco, comentários, chavetas, etc. Então, que tal inserirmos uma instrução de código que não tenha qualquer efeito nos valores? Para isso, podemos adicionar a instrução **i = i** na linha 25 do nosso código (demonstrado na **Listagem 6**) e colocar aí o nosso *breakpoint*. Feito isso teremos um *warning* do IDE dizendo que "A atribuição da variável i não tem efeito". Ótimo, é exatamente isso o que queremos.

Listagem 6. Pequeno hack no código para facilitar o processo de debug.

```
22  if (curSize > maxSize) {
23      maxSize = curSize;
24      longest = array[i];
25      i = i;
26  }
```

Em seguida, executamos mais uma vez o programa em modo *debug*.

A **Tabela 2** mostra as alterações dos valores das variáveis do método **longestSequence()**, sendo que cada coluna mostra a alteração destes valores para cada vez que continuamos a execução do programa pressionando a tecla *F8*. Estes são os valores mostrados na visão de variáveis. Como pode reparar, cada vez que uma variável muda de valor, o seu fundo fica realçado com uma cor diferente.

Olhando para a primeira coluna da tabela, na segunda linha, temos a variável **array**, que tem sempre o mesmo **id**, pois nunca fazemos a atribuição para um novo array. Na linha seguinte, temos a variável **length**, que também tem sempre o mesmo valor de 150 (posições), pois não alteramos o tamanho do *array*. Por sua vez, os valores das variáveis **curSize** e **maxSize** serão sempre os mesmos, já que na linha 23 temos a instrução **maxSize = curSize**, que trata de fazer exatamente isso.

Nome	Valor (1)	Valor (2)	Valor (3)	Valor (4)	Valor (5)
array	(id=16)	(id=16)	(id=16)	(id=16)	(id=16)
length	150	150	150	150	150
curSize	2	3	4	5	6
maxSize	2	3	4	5	6
longest	2	2	1	8	8
i	4	5	33	42	43

Tabela 2. Valores das variáveis para breakpoint na linha 25

Na primeira coluna de valores, temos o nosso primeiro resultado. Ele diz que o valor 2 é o primeiro valor que aparece repetido em sequência no *array* (**longest: 2**) e que aparece repetido por duas vezes (**maxSize: 2**), na 4ª iteração do ciclo **for** (**i: 4**), ou, se preferir, na 5ª posição do *array*, respeitando a condição: "curSize > maxSize".

A segunda coluna diz que o valor 2 é mais uma vez o valor que aparece repetido mais vezes. Agora aparecendo repetido por 3 vezes, na 5ª iteração do ciclo **for** (6ª posição do *array*). Por acaso, é na mesma sequência que a anterior, mas poderia ser em outra.

A terceira coluna diz que o valor 1 é o valor que passa a ser o que mais se repete, tendo aparecido 4 vezes em sequência, terminando na posição 34 do *array*, ou seja, aparece nas posições 31, 32, 33 e 34.

E finalmente, a quarta e quinta colunas dizem que o valor 8 foi o valor que apareceu mais vezes, tendo aparecido numa sequência de 5 e, posteriormente, 6 vezes repetidas, no 42º e 43º ciclos **for**, ou seja, terminando sua maior sequência de repetições na 44ª posição do *array*.

Missão cumprida, se tivermos que comparar resultados, ou validar valores intermediários, esta é uma excelente maneira de conseguirmos fazer isso.

Alterando o valor de variáveis

Outra funcionalidade muito útil que o *debugger* nos dá, é a possibilidade de alterar o valor das variáveis em tempo de execução. Por exemplo, sabemos que, de acordo com a **Tabela 2**, o valor 8 é o valor que ocorre o maior número de vezes consecutivas (seis vezes) no *array* de *input* que demos para o método **longestSequence()**.

Como já foi dito antes, este é um *array* pequeno, mas poderia ser muito grande. Digamos que o *array* tem 100 mil posições e que sabemos apenas que o número máximo de vezes que um elemento surge são seis vezes. Como fazemos para validar os valores das variáveis apenas para quando este valor máximo é alcançado?

Vamos fazer isso deixando o *breakpoint* na linha 25 do código (no nosso *hack* que contém a instrução **i=i**) e executar a aplicação em modo *debug*. Quando o *debugger* parar nessa linha, poderemos ver, na visão de variáveis, o valor da variável **longest = 2** e o valor da variável **maxSize = 2**. Neste momento, selecione o valor 2 do **maxSize** com um clique do mouse, altere o valor para 5 e pressione *Enter*, como mostra a **Figura 14**.

Desta forma, se pressionarmos a tecla *F8*, o *debugger* continuará a execução do programam até que a condição **curSize > maxSize**

seja verdadeira, ou seja, até que existam pelo menos seis repetições de um valor, o que nos dará o valor pretendido, conforme nos mostra a **Figura 15**.

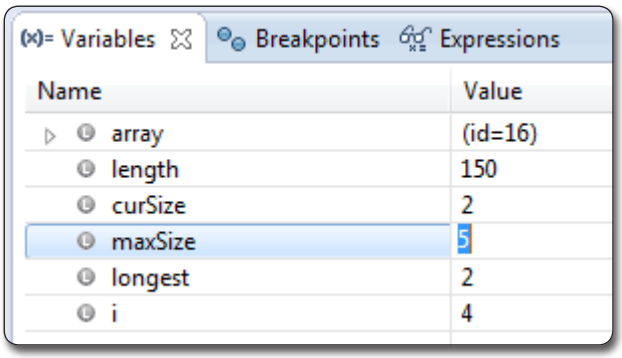


Figura 14. Alterando o valor de variáveis

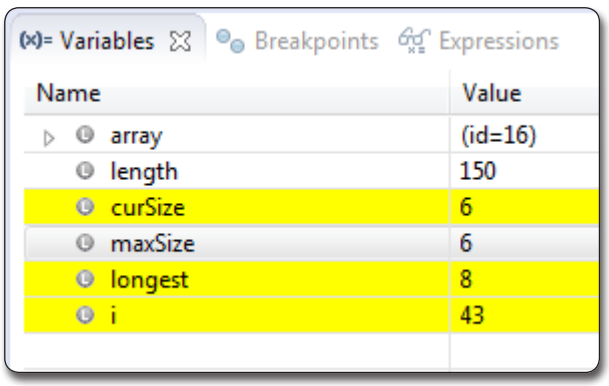


Figura 15. Valor de variável alterada

A partir disso, podemos inspecionar os valores das variáveis que nos interessam, sem nos preocupar com o que aconteceu para trás, e sem ter o trabalho de passar por todas as outras sequências, até porque, como já havíamos dito, o número de maior sequência poderia ser muito maior que o que temos no exemplo, fazendo com que o número de sequências intermediárias fosse igualmente grande, tornando o nosso trabalho bastante moroso. Desta forma, com apenas uma instrução, chegamos no local pretendido.

De igual forma, podemos alterar o valor de qualquer outra variável, até mesmo de valores internos de objetos, como no caso do *array*. Portanto, experimente! Abra o array, troque seus valores e compare os resultados. Ou então, experimente saltar para um índice exato do ciclo **for**.

Por exemplo, se quando estivermos lendo o log de um programa, encontramos uma exceção e vemos, através do log, que esta exceção aconteceu num ciclo **for**, de número 1.476.728, alterar o valor do controlador do ciclo **for** seria uma forma simples de chegar neste ponto exato do ciclo para tentar entender o que acontece neste preciso momento de execução. Outra forma seria fazer um *breakpoint* condicional, que é um dos assuntos que vamos abordar no próximo artigo.

Autor

José Fernandes A. Júnior
jfajunior@gmail.com

É Mestre em Engenharia Informática pela Faculdade de Ciências e Tecnologias da Universidade Nova de Lisboa (FCT-UNL). Trabalha com Java há 10 anos, mas vem trabalhando com diversas linguagens, em diversos ramos e áreas, desde core engines de empresas de telecomunicação, até aplicações móveis para Android e iOS. Trabalhou como formador autorizado da Sun Microsystems nos cursos de Java, Unix, Shell Programming e JCAPS. Possui as certificações de Java Swing, Enterprise Java Beans, Java Composite Application Platform Suite (JCAPS), Certificado de Aptidão Profissional (CAP) e Titanium Certified App Developer (TCAD).

Links:

Site oficial da Apache com documentação da classe PatternLayout.
http://logging.apache.org/log4j/1.2/apidocs/org/apache/log4j/PatternLayout.html

Java Persistence API (JPA): Primeiros passos

A principal API de persistência do Java

Desde o surgimento da linguagem Java foram criadas diversas APIs para fazer a interface do código Java com um SGBD (Sistema Gerenciador de Banco de Dados). Uma das primeiras bibliotecas com esse intuito foi o JDBC (*Java Database Connectivity*), que surgiu no fim dos anos 90 e ainda é muito utilizado. No JDBC, o acesso ao banco de dados é feito através de comandos (*Statements*) escritos na linguagem SQL (*Structured Query Language*), que é a linguagem padrão dos gerenciadores de banco de dados.

Em seguida surgiram as tecnologias de mapeamento objeto-relacional, conhecidas por ORM (*Object-Relational Mapping*). Tais tecnologias estabelecem uma "ponte" entre esses dois modelos de dados: o modelo orientado a objetos, utilizado em linguagens de programação como o Java, e o modelo relacional, utilizado pela maioria dos SGBDs.

Deste modo, foram sendo desenvolvidas APIs que realizam esse mapeamento em Java. O Hibernate foi uma das primeiras bibliotecas criadas com esse intuito. Com essas ferramentas, é possível fazer tanto a geração de código Java a partir das tabelas da base de dados, como a geração de scripts de criação de banco de dados (em SQL) a partir das classes Java.

Com intuito de padronizar as implementações de ORM em Java, foi elaborada a especificação JPA, como parte da plataforma Java EE 5. A versão atual do JPA é a 2.1 e ela é suportada por todos os servidores aderentes ao padrão Java EE existentes no mercado (JBoss, GlassFish, WebSphere, etc.). Além destas, existem também algumas implementações do JPA de código aberto, como por exemplo, a EclipseLink, que usaremos no estudo de caso que será apresentado.

Neste artigo, abordaremos um exemplo simples para ilustrar o funcionamento do JPA. Criaremos um modelo de dados com três objetos (**Aluno**, **Disciplina** e **Professor**), simulando um banco de dados de uma faculdade ou de um colégio. No exemplo, veremos o código em Java das entidades JPA que representam esses três objetos do nosso modelo. Também iremos implementar o código que fará a inserção de alguns dados de testes, para popular o nosso banco de dados. Por fim, implementaremos algumas consultas para extrair relatórios a partir dos dados armazenados.

Fique por dentro

Este artigo vai ajudar o leitor a se familiarizar com a JPA (Java Persistence API) e aprender na prática como usar esta tecnologia. Com a auxílio das ferramentas open source EclipseLink e Derby DB, vamos modelar um banco de dados bem simples e desenvolver um programa que explore os principais recursos da JPA para criar as tabelas, popular os dados e realizar consultas nos dados armazenados.

Criando entidades para representar os dados

No JPA, as classes Java que representam os dados a serem armazenados no SGBD são chamadas de Entidades (*Entity*). Elas são classes comuns que contêm apenas atributos e métodos *getters* (que recuperam o valor dos atributos) e *setters* (que modificam o valor dos atributos). O que torna essa classe uma entidade é a presença da anotação (*Annotation*) **@javax.persistence.Entity**. Tal anotação faz com que o JPA associe a classe Java em questão a uma tabela do banco de dados. A própria implementação do JPA se encarregará de criar a tabela automaticamente no banco de dados relacional.

No código Java, basta criar as instâncias das entidades, preenchendo-as com os dados desejados e depois salvar no banco de dados. Cada entidade, quando persistida, corresponderá a um registro na tabela do banco de dados. Mais adiante veremos como persistir as entidades, quando falarmos do **EntityManager**.

Na **Listagem 1** temos o código da nossa primeira entidade, de nome **Aluno**, com os atributos **matricula**, **nome** e **curso**. Como pode ser observado, não existe nada de complexo neste código.

A declaração da classe, como vimos, deve ter a anotação **@Entity**, indicando para o JPA que essa classe é uma entidade. Por padrão, toda entidade deve ter um construtor sem argumentos. Neste caso, ele foi omitido por ser o construtor *default* em Java. Caso algum construtor com parâmetros seja definido, no entanto, deve-se definir também a versão sem parâmetros, senão a implementação do JPA apresentará uma exceção em tempo de execução.

Listagem 1. Código da classe/entidade Aluno.

```
package br.com.disciplinas.jpa.model;

import javax.persistence.Entity;
import javax.persistence.Id;

@Entity
public class Aluno {

    @Id
    private int matricula;
    private String nome;
    private String curso;

    public int getMatricula() {
        return matricula;
    }

    public void setMatricula(int matricula) {
        this.matricula = matricula;
    }

    public String getNome() {
        return nome;
    }

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

    public String getCurso() {
        return curso;
    }

    public void setCurso(String curso) {
        this.curso = curso;
    }

}
```

Outra característica obrigatória de uma Entidade é que ela deve ter uma chave primária (*Primary Key*), ou seja, um atributo que identifique unicamente cada uma das suas instâncias. No nosso caso, escolhemos o atributo **matricula**, e por isso colocamos na sua declaração a anotação **@Id**.

Você pode definir também uma chave primária que seja gerada automaticamente durante a criação da instância. Para isso, a anotação **@GeneratedValue** deve ser usada, para que o JPA se encarregue de gerar a chave para você, de maneira sequencial. Veremos um exemplo dessa situação no exemplo mais adiante. Além disso, ainda é possível criar chaves compostas por uma combinação de dois ou mais atributos, em alguns casos excepcionais.

Em uma entidade JPA, cada atributo corresponde a uma coluna na tabela da base de dados. Deste modo, a entidade **Aluno** dará origem à **Tabela 1** no banco de dados.

por padrão, o nome da propriedade da classe Java será o nome da coluna da tabela. Mas é possível dar um nome diferente usando a anotação **@Column**. Por exemplo, se quiséssemos que uma propriedade **dataDaMatricula** fosse mapeada em uma coluna **data_da_matricula**, poderíamos fazê-lo da seguinte maneira:

```
@Column(name="data_da_matricula")
private String dataDaMatricula;
```

Do mesmo modo, é possível mudar o nome da tabela gerada a partir da entidade, para que ela tenha um nome diferente do nome da classe. Para isso usa-se a anotação **@Table(name="NOME_DA_TABELA")**, junto com a anotação **@Entity**.

Por definição, todos os atributos da classe Java serão mapeados no banco de dados para serem persistidos. No entanto, se desejar que algum atributo não seja persistido, utilize a anotação **@Transient** na declaração do atributo, conforme o exemplo:

```
@Transient
private String atributoTransiente;
```

Coluna	Tipo
MATRICULA	INTEGER PRIMARY KEY
NOME	TEXT
CURSO	TEXT

Tabela 1. Tabela ALUNO

Relacionamentos entre os dados

No JPA, é possível também mapear os relacionamentos entre as entidades. No nosso modelo, por exemplo, temos que o **Professor** ministra uma ou mais **Disciplinas**, e cada **Aluno** está matriculado em uma ou mais **Disciplinas**. O relacionamento é implementado por simples referências entre os objetos. Vamos então criar um relacionamento para mapear as disciplinas em que um aluno está matriculado. Veja o código a seguir:

```
@ManyToMany
private Set<Disciplina> disciplinas;
```

Aqui estamos dizendo que um **Aluno** pode estar matriculado em várias disciplinas. Por isso ele tem uma coleção (implementada pela classe **java.util.Set**) de objetos da classe **Disciplina**, que por sua vez também é uma entidade JPA.

A anotação **@ManyToMany** indica a cardinalidade (ou multiplicidade) do relacionamento. A seguir analisamos as anotações que podem ser usadas para indicar a multiplicidade dos relacionamentos:

- **@OneToOne:** Usada para mapear um relacionamento "um para um". Ou seja, cada instância de uma entidade A se relaciona com no máximo uma instância de uma entidade B, e *vice-versa*;
- **@OneToMany:** Indica um relacionamento "um para muitos". Assim, uma instância de A se relaciona com várias instâncias de B, no entanto, cada instância de B se relaciona com apenas uma instância de A;
- **@ManyToOne:** Este é o oposto do **@OneToMany**, ou seja, cada instância de A se relaciona com apenas uma instância de B, mas cada instância de B pode se associar a várias instâncias de A;

• **@ManyToMany:** Este é o chamado relacionamento "muitos para muitos", ou seja, cada instância de A pode se associar a várias instâncias B e vice-versa.

Além da cardinalidade, você pode definir relacionamentos unidirecionais, quando apenas uma das entidades envolvidas declara o relacionamento, ou bidirecionais, quando ambas declaram o relacionamento.

Normalmente utiliza-se relacionamentos unidirecionais. Por exemplo, a instância **Disciplina** tem uma referência a uma instância de **Professor**. Se você quiser tornar esse relacionamento bidirecional, você deve acrescentar a referência a uma lista de **Disciplinas** na classe **Professor**. Isto tem a vantagem de fornecer acesso mais rápido às **Disciplinas** de um **Professor** via referência direta no código Java, sem necessidade de buscar no banco de dados. Por outro lado, isso pode carregar demais a memória quando consultas muito grandes forem executadas. Por isso relacionamentos bidirecionais devem ser usados com cautela, principalmente quando envolvem coleções de objetos do tipo "um para muitos" e "muitos para muitos".

Note que o fato de você declarar relacionamentos unidirecionais não impede que você consiga acessar os dados navegando nos dois sentidos. No exemplo supracitado, mesmo declarando o relacionamento apenas na classe **Disciplina**, ainda é possível fazer a busca no sentido inverso, ou seja, buscar as Disciplinas de um Professor. Isso pode ser feito através de uma *query* (consulta) com um "JOIN" entre as tabelas no banco de dados. Veremos isso com mais detalhes mais adiante, na parte de consultas.

Criando as entidades do nosso exemplo

Agora que vimos como declarar as entidades e seus relacionamentos, vamos criar o código das entidades que usaremos no nosso exemplo. Como vimos, nossa proposta é criar um modelo de dados bem simples para representar um cadastro escolar, contendo as entidades **Aluno**, **Professor** e **Disciplina**.

Veja nas **Listagens 2**, **3** e **4** o código das três classes Java que usaremos nos exercícios, seus atributos e os relacionamentos entre elas.

A entidade **Aluno** possui os campos **matricula**, **nome** e **curso**. Conforme indicado anteriormente, o atributo **matricula** representa a chave primária e por isso possui a anotação **@Id**, pois supomos que a matrícula do aluno é um identificador único. Além disso, essa entidade apresenta um relacionamento unidirecional de muitos para muitos com a entidade **Disciplina**. Isto é indicado pelo atributo **disciplinas**, que é declarado como um conjunto de disciplinas (**Set<Disciplina>**) nas quais o **Aluno** está matriculado.

Já a entidade **Disciplina** possui os atributos **id**, **sigla**, **nome**, **diaDaSemana**, **horaInicio** e **horaFim**. O atributo **id** representa a chave primária. Diferentemente da entidade **Aluno**, aqui definimos uma chave primária que é um número do tipo **long**, incrementada automaticamente a cada nova instância. Por isso temos a anotação **@GeneratedValue(strategy = GenerationType.AUTO)**.

Listagem 2. Código da classe Aluno.

```
package br.com.disciplinas.jpa.model;

import java.util.Set;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.ManyToMany;

@Entity
public class Aluno {

  @Id
  private int matricula;
  private String nome;
  private String curso;

  @ManyToMany
  private Set<Disciplina> disciplinas;

  public Aluno() {
  }

  public Aluno(int matricula, String nome, String curso,
      Set<Disciplina> disciplinas) {
    this.matricula = matricula;
    this.nome = nome;
    this.curso = curso;
    this.disciplinas = disciplinas;
  }

  public int getMatricula() {
    return matricula;
  }

  public void setMatricula(int matricula) {
    this.matricula = matricula;
  }

  public String getNome() {
    return nome;
  }

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

  public String getCurso() {
    return curso;
  }

  public void setCurso(String curso) {
    this.curso = curso;
  }

  public Set<Disciplina> getDisciplinas() {
    return disciplinas;
  }

  public void setDisciplinas(Set<Disciplina> disciplinas) {
    this.disciplinas = disciplinas;
  }

  @Override
  public String toString() {
    return "Aluno [matricula=" + matricula + ", nome=" + nome + ", curso="
        + curso + "]";
  }

}
```

Listagem 3. Código da classe Disciplina.

```
package br.com.disciplinas.jpa.model;

import javax.persistence.Entity;
import javax.persistence.EnumType;
import javax.persistence.Enumerated;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.ManyToOne;

@Entity
public class Disciplina {

 public enum DiaDaSemana { SEGUNDA, TERÇA, QUARTA, QUINTA, SEXTA };

   @Id
   @GeneratedValue(strategy = GenerationType.AUTO)
   private Long id;
   private String sigla;
   private String nome;

   @Enumerated(EnumType.STRING)
   private DiaDaSemana diaDaSemana;
   private Integer horaInicio;
   private Integer horaFim;

   @ManyToOne
   private Professor professor;

   public Disciplina() {
   }

   public Disciplina(String sigla, String nome, DiaDaSemana diaDaSemana,
         Integer horaInicio, Integer horaFim, Professor professor) {
      this.sigla = sigla;
      this.nome = nome;
      this.diaDaSemana = diaDaSemana;
      this.horaInicio = horaInicio;
      this.horaFim = horaFim;
      this.professor = professor;
   }

   public Long getId() {
      return id;
   }

   public void setId(Long id) {
      this.id = id;
   }

   public String getSigla() {
      return sigla;
   }

   public void setSigla(String sigla) {
      this.sigla = sigla;
   }

   public String getNome() {
      return nome;
   }

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

   public DiaDaSemana getDiaDaSemana() {
      return diaDaSemana;
   }

   public void setDiaDaSemana(DiaDaSemana diaDaSemana) {
      this.diaDaSemana = diaDaSemana;
   }

   public Integer getHoraInicio() {
      return horaInicio;
   }

   public void setHoraInicio(Integer horaInicio) {
      this.horaInicio = horaInicio;
   }

   public Integer getHoraFim() {
      return horaFim;
   }

   public void setHoraFim(Integer horaFim) {
      this.horaFim = horaFim;
   }

   public Professor getProfessor() {
      return professor;
   }

   public void setProfessor(Professor professor) {
      this.professor = professor;
   }

   @Override
   public String toString() {
      return "Disciplina [id=" + id + ", sigla=" + sigla + ", nome=" + nome
            + ", diaDaSemana=" + diaDaSemana + ", horaInicio=" + horaInicio
            + ", horaFim=" + horaFim + ", professor=" + professor + "]";
   }
}
```

Nesta classe apresentamos também a anotação **@Enumerated**, que permite associar valores de uma enumeração Java (*enum*) a um atributo. Este tipo de anotação é bem útil para mapear atributos que têm um conjunto restrito de valores que podem ser especificados em um *enum*, como é o caso dos dias da semana.

Ademais, destacamos o relacionamento da **Disciplina** com a entidade **Professor**. Trata-se de um relacionamento com cardinalidade "muitos para um", pois uma disciplina pode ser ministrada por apenas um professor, mas um professor pode ministrar várias disciplinas. Notamos ainda que este é um relacionamento unidirecional, visto que a referência ao **Professor** está declarada na classe **Disciplina**, mas o inverso não ocorre, como podemos ver na definição de **Professor**, na **Listagem 4**.

A entidade **Professor**, por sua vez, possui os atributos **registro**, **nome** e **departamento**, sendo que o **registro** é a chave primária. Nesta classe, não declaramos nenhum relacionamento, pois a relação Professor-Disciplina já foi declarada na classe **Disciplina**.

Note que nas três classes, como declaramos um construtor com parâmetros, também foi necessário declarar o construtor padrão (sem parâmetros), pois isso é exigido pelo JPA.

Listagem 4. Código da classe Professor.

```
package br.com.disciplinas.jpa.model;

import javax.persistence.Entity;
import javax.persistence.Id;

@Entity
public class Professor {

  @Id
  private int registro;
  private String nome;
  private String departamento;

  public Professor(int registro, String nome, String departamento) {
    this.registro = registro;
    this.nome = nome;
    this.departamento = departamento;
  }

  public Professor() {

  }

  public int getRegistro() {
    return registro;
  }

  public void setRegistro(int registro) {
    this.registro = registro;
  }

  public String getNome() {
    return nome;
  }

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

  public String getDepartamento() {
    return departamento;
  }

  @Override
  public String toString() {
    return "Professor [registro=" + registro + ", nome=" + nome
        + ", departamento=" + departamento + "]";
  }

  public void setDepartamento(String departamento) {
    this.departamento = departamento;
  }

}
```

Manipulando as entidades com o EntityManager

Agora que temos as nossas entidades implementadas, o próximo passo é popular o banco de dados com alguns objetos como exemplo. Para isso, vamos usar a classe **javax.persistence.EntityManager**, que gerencia a conexão das entidades com o SGBD.

A **EntityManager** é a principal classe do JPA, e faz uma ponte entre o código Java e as respectivas tabelas da base de dados. Ela oferece métodos para salvar, alterar, apagar ou buscar as entidades.

Para instanciar o **EntityManager** no nosso código, utilizaremos a classe **EntityManagerFactory**. Como exemplo disso, na **Listagem 5** apresentamos a classe **CriarBD**, que será usada para popular o banco de dados. Observe que criamos três **Professores**, quatro **Disciplinas** e quatro **Alunos** como exemplo, e salvamos todas essas entidades no banco de dados com chamadas ao método **EntityManager.persist()**.

Listagem 5. Código da classe CriarBD.

```
package br.com.disciplinas.jpa;
import java.util.HashSet;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

import br.com.disciplinas.jpa.model.*;
import br.com.disciplinas.jpa.model.Disciplina.DiaDaSemana;

public class CriarBD {

  public static void main(String[] args) {
    EntityManagerFactory emf = Persistence
        .createEntityManagerFactory("disciplinas");
    EntityManager em = emf.createEntityManager();
    em.getTransaction().begin();

    Professor professor1 = new Professor(1, "Bill Gates", "Computação");
    em.persist(professor1);
    Professor professor2 = new Professor(2, "Oswald de Souza", "Matemática");
    em.persist(professor2);
    Professor professor3 = new Professor(3, "Cesar Lattes", "Física");
    em.persist(professor3);

    Disciplina disciplina1 = new Disciplina("MA-100", "Cálculo 1",
        DiaDaSemana.SEGUNDA, 8, 10, professor2);
    em.persist(disciplina1);
    Disciplina disciplina2 = new Disciplina("MA-100", "Cálculo 1",
        DiaDaSemana.QUARTA, 8, 10, professor2);
    em.persist(disciplina2);
    Disciplina disciplina3 = new Disciplina("CC-100", "Algoritmos",
        DiaDaSemana.QUARTA, 10, 12, professor1);
    em.persist(disciplina3);
    Disciplina disciplina4 = new Disciplina("F-100", "Física 1",
        DiaDaSemana.QUINTA, 14, 16, professor3);
    em.persist(disciplina4);

    HashSet<Disciplina> grade1 = new HashSet<Disciplina>();
    grade1.add(disciplina2);
    grade1.add(disciplina3);
    grade1.add(disciplina4);
    HashSet<Disciplina> grade2 = new HashSet<Disciplina>();
    grade2.add(disciplina1);
    grade2.add(disciplina2);
    grade2.add(disciplina3);
    Aluno aluno1 = new Aluno(20130001, "Adriano", "Computação", grade1);
    em.persist(aluno1);
    Aluno aluno2 = new Aluno(20130002, "Felipe", "Computação", grade1);
    em.persist(aluno2);
    Aluno aluno3 = new Aluno(20130003, "Mariana", "Computação", grade2);
    em.persist(aluno3);
    Aluno aluno4 = new Aluno(20130004, "Rodrigo", "Computação", grade2);
    em.persist(aluno4);
    em.getTransaction().commit();
    em.close();
    emf.close();
  }
}
```

Nesta classe vemos também como usar os métodos **EntityManager.getTransaction().begin()**, para demarcar o início de uma transação, e **EntityManager.getTransaction().commit()**, para confirmar as alterações nos dados. As transações são usadas para delimitar as operações de escrita no banco de dados que devem ocorrer no mesmo "lote", de maneira atômica. Ou seja, ao final da transação (após o "*commit*"), todas as operações são efetivadas de uma vez. Assim, se alguma operação deste lote falhar, nenhuma alteração é feita no banco de dados.

Assim, quando temos uma transação, só temos duas alternativas: ou todas as operações são executadas com sucesso (quando é feito o "*commit*"), ou nenhuma operação é realizada (este evento de se recuperar da falha é chamado de "*rollback*").

Isso evita que a base de dados fique num estado inconsistente. Por exemplo, se um **Professor** não fosse persistido por algum erro, e não usássemos esse recurso de transações, poderíamos ter na nossa base de dados uma **Disciplina** referenciando um **Professor** que não existe.

Além do método **persist()**, que salva os dados da entidade, existem outros no **EntityManager** para manipular as entidades, como é o caso dos métodos **remove(Object entity)**, que apaga a entidade do banco de dados, e **merge(Object entity)**, que atualiza a base de dados com os atributos da entidade em memória.

Consultando os dados armazenados

Veremos agora como usar o **EntityManager** para fazer buscas (*queries*) na base de dados, recuperar as entidades e extrair relatórios simples com os dados obtidos. Para isso, devemos usar o método **createQuery()**, passando como parâmetro a **String** com a consulta desejada, especificada na linguagem JPQL, e depois vamos usar o método **Query.getResultList()**, para obter a lista de entidades resultantes da pesquisa.

No JPA, as consultas são escritas na linguagem JPQL (*Java Persistence Query Language*), que é similar ao tradicional SQL usado nos bancos de dados relacionais. A principal diferença entre elas é que o JPQL realiza as operações usando as entidades na cláusula FROM, no lugar das tabelas, como no SQL padrão. Por exemplo, se quisermos obter todas as entidades **Professor** do nosso banco, por ordem alfabética, podemos usar o seguinte comando JQPL:

```
SELECT p FROM Professor p ORDER BY p.nome
```

Dito isso, vamos então implementar a classe **ConsultarBD**, que fará algumas consultas com os dados que temos na nossa base. Veja o código na **Listagem 6**. Na classe **ConsultarBD** codificamos quatro consultas à base de dados. Na primeira delas, a mais simples, recuperamos a lista todos os professores cadastrados.

Listagem 6. Código da classe ConsultarBD.

```
package br.com.disciplinas.jpa;

import java.util.List;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.Query;

import br.com.disciplinas.jpa.model.*;

public class ConsultarBD {

  @SuppressWarnings("unchecked")
  public static void main(String[] args) {
    EntityManagerFactory emf = Persistence.createEntityManagerFactory("disciplinas");
    EntityManager em = emf.createEntityManager();

    System.out.println("Lista de Professores:");
    Query q = em.createQuery("SELECT p FROM Professor p ORDER BY p.nome");
    List<Professor> list = q.getResultList();
    for (Professor professor : list) {
      System.out.println(professor);
    }
    System.out.println("Total: " + list.size());
    System.out.println("---------------------------------------\n");

    System.out.println("Lista de Alunos e Disciplinas:");
    q = em.createQuery("SELECT a FROM Aluno a");
    List<Aluno> alunos = q.getResultList();
    System.out.println("RA\t\tAluno\t\tDisciplinas");
    for (Aluno aluno : alunos) {
      System.out.print(aluno.getMatricula());
      System.out.print("\t" + aluno.getNome() + "\t\t");
      for (Disciplina materia : aluno.getDisciplinas()) {
        System.out.print(materia.getSigla() + " ");
      }
      System.out.println();
    }
    System.out.println("Total de alunos: " + alunos.size());
    System.out.println("---------------------------------------\n");

    System.out.println("Lista de Alunos do Prof. Cesar Lattes:");
    q = em.createQuery("SELECT DISTINCT a "
        + " FROM Aluno a JOIN a.disciplinas d "
        + " WHERE d.professor.nome = :prof");
    q.setParameter("prof", "Cesar Lattes");
    List<Aluno> lista1 = q.getResultList();
    for (Aluno aluno : lista1) {
      System.out.println(aluno);
    }
    System.out.println("Total de alunos: " + lista1.size());
    System.out.println("---------------------------------------\n");

    System.out.println("Lista de Professores que dão aula quarta-feira pela manhã:");
    q = em.createQuery("SELECT d FROM Disciplina d "
        + " WHERE d.diaDaSemana = ?1 AND d.horaInicio >= ?2 AND d.horaFim <= ?3");
    q.setParameter(1, Disciplina.DiaDaSemana.QUARTA);
    q.setParameter(2, new Integer(8));
    q.setParameter(3, new Integer(12));

    List<Disciplina> lista2 = q.getResultList();
    System.out.println("Professor\t\tDia\tHorario");
    for (Disciplina d : lista2) {
      String linha = String.format("%s\t\t%s\t%02dh-%02dh",
          d.getProfessor().getNome(),
          d.getDiaDaSemana(),
          d.getHoraInicio(),
          d.getHoraFim());
      System.out.println(linha);
    }

    em.close();
    emf.close();
  }

}
```

Na segunda consulta, recuperamos a lista de alunos e as disciplinas em que cada um deles estão matriculados. Para isso, obtemos a lista dos alunos, e depois, para cada **Aluno** obtido, iteramos no atributo **disciplinas** que faz o relacionamento com a entidade **Disciplina**.

A terceira consulta usa todas as três entidades, para obter todos os alunos que têm aula com um determinado professor. Aqui empregamos a palavra **JOIN** dentro da cláusula **FROM** para ligar as entidades **Aluno** e **Disciplina** através do atributo **disciplinas**. Na cláusula **WHERE** fazemos a conexão das entidades **Disciplina** e **Professor** para filtrar o resultado pelo nome do **Professor**. O símbolo **:prof** no comando JPQL indica que esse valor será passado como parâmetro mais adiante. Logo em seguida, usamos o método **Query.setParameter()** para fazer a atribuição deste parâmetro com o nome do professor usado na consulta.

Finalmente, na quarta consulta, obtemos uma lista dos professores que têm aulas às quartas-feiras no período de 8h a 12h. Neste exemplo vemos uma outra maneira de se passar parâmetros para uma consulta, de forma sequencial, através dos símbolos **?1**, **?2** e **?3**.

Ao executarmos este programa com as consultas, o resultado é exibido conforme a **Listagem 7**.

O arquivo persistence.xml – Configurando a conexão com o SGBD

Para finalizar o nosso código, precisamos criar o arquivo *persistence.xml*, que descreve como é feita a conexão com o banco de dados. É importante que esse arquivo seja especificado dentro da pasta *src/META-INF* do seu projeto. Nele estão as definições de persistência do seu projeto, tais como:

- Nome da *Persistence Unit* que deve ser usada pelo **EntityManagerFactory** para se conectar com o banco de dados;
- Nome da classe provedora (**provider**), que é quem implementa o JPA. No nosso caso é a classe **PersistenceProvider**, da biblioteca EclipseLink;
- As classes que implementam as entidades JPA, que foram as que definimos anteriormente;
- Parâmetros para acesso ao banco de dados. Estes valores dependem do SGBD que adotarmos. Aqui usamos os parâmetros para acessar o DerbyDB.

Listagem 7. Resultado da execução do programa.

```
Lista de Professores:
Professor [registro=1, nome=Bill Gates, departamento=Computação]
Professor [registro=3, nome=Cesar Lattes, departamento=Física]
Professor [registro=2, nome=Oswald de Souza, departamento=Matemática]
Total: 3
---------------------------------------------------------------------------------------------------

Lista de Alunos e Disciplinas:
RA        Aluno     Disciplinas
20130001  Adriano   CC-100 F-100 MA-100
20130002  Felipe    CC-100 F-100 MA-100
20130004  Rodrigo   CC-100 MA-100 MA-100
20130003  Mariana   CC-100 MA-100 MA-100
Total de alunos: 4
---------------------------------------------------------------------------------------------------

Lista de Alunos do Prof. Cesar Lattes:
Aluno [matricula=20130001, nome=Adriano, curso=Computação]
Aluno [matricula=20130002, nome=Felipe, curso=Computação]
Total de alunos: 2
---------------------------------------------------------------------------------------------------

Lista de Professores que dão aula quarta-feira pela manhã:
Professor         Dia       Horario
Bill Gates        QUARTA    10h-12h
Oswald de Souza   QUARTA    08h-10h
```

Nota

Persistence Unit é o nome da unidade de persistência que contém todas as informações para se conectar com o banco de dados.

Neste exemplo usamos um gerenciador de banco de dados embarcado chamado Derby. Trata-se de um banco de dados que executa dentro da aplicação cliente, sem a necessidade de se configurar um servidor, como nos bancos de dados tradicionais (Oracle, MySQL, PostgreSQL, etc.).

O conteúdo do arquivo *src/META-INF/persistence.xml* que usaremos neste exemplo é indicado na **Listagem 8**.

Na propriedade **javax.persistence.jdbc.driver** você deve substituir o caminho */Users/JoseRicardo/Documents/* por uma pasta válida no seu computador. É dentro desta pasta que o Derby irá criar o diretório *disciplinasBD*, que vai conter os arquivos onde serão salvos seus dados. As demais propriedades do arquivo você pode deixar conforme a listagem.

Como configurar o ambiente e executar o exemplo

Para executar o exemplo desse artigo, crie um projeto Java na sua IDE favorita (Eclipse, NetBeans, ou outra de sua preferência), e implemente as classes descritas. Não esqueça de definir também o arquivo *persistence.xml*, na pasta src/META-INF.

Listagem 8. Conteúdo do arquivo de configuração persistence.xml.

```
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
   http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
   version="2.0" xmlns="http://java.sun.com/xml/ns/persistence">
   <persistence-unit name="disciplinas" transaction-type="RESOURCE_LOCAL">
      <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
      <class>br.com.disciplinas.jpa.model.Aluno</class>
      <class>br.com.disciplinas.jpa.model.Disciplina</class>
      <class>br.com.disciplinas.jpa.model.Professor</class>
      <properties>
       <property name="javax.persistence.jdbc.driver"
           value="org.apache.derby.jdbc.EmbeddedDriver" />
        <property name="javax.persistence.jdbc.url"
           value="jdbc:derby:/Users/JoseRicardo/Documents/disciplinasBD;create=true" />
        <property name="javax.persistence.jdbc.user" value="test" />
        <property name="javax.persistence.jdbc.password" value="test" />

        <!-- EclipseLink should create the database schema automatically -->
        <property name="eclipselink.ddl-generation" value="create-tables" />
          <property name="eclipselink.ddl-generation.output-mode"
        value="database" />
      </properties>
   </persistence-unit>
</persistence>
```

Além disso, é preciso adicionar no *classpath* do projeto as bibliotecas do EclipseLink, que é a implementação do JPA que usamos no exemplo. Para isso, baixe o arquivo zip do site (veja o endereço na seção **Links**) e adicione os seguintes JARs ao projeto:

- *eclipselink.jar*;
- *javax.persistence*.jar*.

Ainda no *classpath*, adicione também o arquivo *derby.jar*. Você pode baixar este arquivo no endereço indicado na seção **Links**.

Uma vez configurado o ambiente, você deve executar primeiro a classe **CriarBD**, que irá criar o banco de dados no arquivo especificado no *persistence.xml* e popular com as entidades. Depois, basta executar a classe **ConsultarBD** para realizar as consultas.

Agora que você conseguiu executar o exemplo, você pode usar seus conhecimentos para criar novas consultas. Por exemplo, que tal criar uma consulta para obter as disciplinas ministradas por professores do departamento de matemática?

Esperamos que este artigo tenha lhe ajudado a dar os primeiros passos com JPA. Agora, é só continuar os estudos e se aprofundar cada vez mais nessa tecnologia. Bons estudos!

Autor

José Ricardo Freitas D'Arce,
Engenheiro de Computação pela UNICAMP, possui a certificação Java Programmer. Trabalha há sete anos no Venturus como arquiteto e desenvolvedor de aplicativos Android e servidores Java EE.

Links:

Introduction to the Java Persistence API.
http://docs.oracle.com/javaee/6/tutorial/doc/bnbpz.html

JPA 2.0 with EclipseLink – Tutorial.
http://www.vogella.com/articles/JavaPersistenceAPI/article.html

Download do EclipseLink.
http://www.eclipse.org/eclipselink/downloads/index.php#2.5

Download do DerbyDB.
http://db.apache.org/derby/releases/release-10.10.1.1.cgi

Java Object Class: Entendendo a classe Object

Conheça a base de todas as classes do Java e os métodos que definem a identidade dos objetos

Ao iniciar no Java, você é apresentado a uma série de frameworks que procuram facilitar o desenvolvimento e aumentar a produtividade. Eles existem aos montes e são utilizados por projetos de todos os tamanhos.

Trabalhando com Java, certamente você irá se deparar com alguns deles em pouco tempo. O primeiro teste que fizer para conectar sua aplicação a um banco de dados já fará uso de um driver que contém a implementação para acessar e manipular um determinado banco de dados. Este drive é um exemplo de framework.

É natural ficar acostumado a estas facilidades e depender delas. Não há nada de errado com isso. Pessoas competentes e com muita experiência trabalharam por muito tempo nesses frameworks, planejando cuidadosamente os recursos disponibilizados e aperfeiçoando-os a cada versão, sempre atentos às necessidade de projetos de outros desenvolvedores.

Obviamente você pode utilizar frameworks sem ter conhecimento de como suas classes foram implementadas, mas você estaria perdendo uma ótima oportunidade de aprender mais sobre Java, programação e lógica em geral explorando o código fonte.

É surpreendente ver como algumas funcionalidades que parecem tão complexas às vezes são implementadas de formas tão simples. Tão impressionante quanto é ver como as classes se integram de forma a colaborarem com o trabalho uma das outras. Por exemplo, você sabia que dentro de um **HashSet** existe uma **HashMap**? E que os objetos que você insere no **Set** são registrados como chave no **Map**? Esse é o tipo de conhecimento que você obtém ao abrir ambas as classes para, por pura curiosidade, entender como elas funcionam.

A experiência que se adquire com este hábito é algo que nenhum professor, instituição ou artigo vai lhe oferecer.

Fique por dentro

Este artigo é útil a todo desenvolvedor que deseja aprender mais sobre a classe Object e seu papel na estrutura de classes do Java. Neste contexto, serão analisados também três de seus métodos: toString(), equals() e hashCode(). Você vai conhecer a implementação padrão de cada um deles, seus contratos e como as outras classes os utilizam. Além disso, apresentaremos algumas técnicas para que estes métodos sejam implementados com o máximo de eficiência e qualidade.

É um conhecimento que tem mais valor porque foi você quem buscou. Você perceberá como seus próximos projetos serão muito beneficiados com todas as novas informações e técnicas que você obteve em sua pesquisa.

Não importa quantos anos você tem ou quantos livros já leu, há sempre mais oportunidades para aprender e tornar-se um profissional melhor do que você foi ontem. E já que a ideia é aprender como as coisas funcionam, existe lugar melhor para começar do que a base?

A classe **Object** é a principal e mais básica classe do Java. Todas as outras a tem como origem, e, portanto, herdam seus métodos, dos quais três você vai conhecer a fundo neste artigo: **toString()**, **equals()** e **hashCode()**. Estes métodos são importantes, pois definem como seu objeto será identificado, logicamente e visualmente (em formato texto).

Você vai aprender sobre a implementação padrão de cada um deles, seus contratos e em quais situações outras classes os utilizam. E vai conhecer também técnicas para implementá-los com o máximo de eficiência, para que seu código tenha ainda mais qualidade.

O pacote java.lang

Ao começar a explorar uma classe, seja ela do próprio Java ou de algum framework, você perceberá que as classes base do Java,

aquelas que integram o pacote **java.lang**, representam boa parte do código.

Você nunca vai ver o **import** de uma classe do **java.lang**, visto que elas são carregadas implicitamente em todas as classes. Mas pode ter certeza que você sempre verá uma **String**, um **Exception** ou algum método da classe **Object** sobrescrito, **toString()**, por exemplo.

O pacote **java.lang** é a base do Java. As classes deste pacote representam a fundação da linguagem, o ponto de partida de qualquer outra estrutura que venha a ser construída sobre ela. Isso não quer dizer que sejam simples ou mesmo que seja fácil dominar suas funcionalidades e contratos. Muito pelo contrário: é necessário bastante conhecimento e atenção para utilizar os elementos da base de forma eficiente e correta.

As classes base do Java foram criadas há muito tempo e vêm sendo aperfeiçoadas e refinadas desde então, sem causar qualquer impacto ao que já foi construído as utilizando.

Ao conhecer estas classes e estudá-las, você terá muitos benefícios e saberá exatamente quais ferramentas tem à disposição e como solucionar problemas, outrora misteriosos, mais facilmente.

É impressionante quantos problemas são causados pela ausência ou má implementação de métodos da classe **Object**. Uma sobrescrita ruim do método **toString()**, por exemplo, pode confundir muito os usuários de uma classe, assim como os objetos de outra classe podem não interagir como esperado com as coleções do Java devido a uma implementação equivocada dos métodos **equals()** e **hashCode()**.

Pensando nisso, o foco deste artigo será a classe **Object** e três de seus métodos: **toString()**, **equals()** e **hashCode()**.

A classe Object

Object é a raiz da hierarquia de classes do Java, a superclasse de todas as classes, direta ou indiretamente.

Sendo a base para todas as classes, **Object** define alguns comportamentos comuns que todos objetos devem ter, como a habilidade de serem comparados uns com os outros, utilizando **equals()**, poderem ser representados como texto, com o método **toString()**, e possuírem um número que identifica suas posições em coleções baseadas em hash, com o **hashCode()**.

O método toString()

O método **toString()** deve retornar um texto conciso e informativo que represente os objetos da sua classe. Este texto deve ser composto pelas informações mais relevantes e únicas dela. Eis alguns exemplos da própria API do Java (veja também a **Listagem 1**):

- A representação visual da classe **String**, é a sequência de caracteres contidos nela;
- Para um **ArrayList**, ou qualquer outra subclasse de **AbstractCollection**, são os elementos presentes na lista separados por vírgula e espaço (", ") envoltos por colchetes ("[]");
- Um objeto da classe **Locale** é representado pela sigla do idioma em minúsculo seguido pela sigla do país em maiúsculo, separados por underline ("_").

Listagem 1. Exemplos de retorno de toString().

```
// String
"John Doe".toString() => John Doe

// ArrayList
vogais.toString() => [a, e, i, o, u]

// Locale
brasilLocale.toString() => pt_BR
```

Implementando toString()

A implementação original do método **toString()** encontra-se na classe **Object**. Ela é bastante simples e faz uso das poucas informações disponíveis em **Object** para retornar um texto que, embora seja pouco informativo, representa objetos de todas as classes que não sobrescreveram **toString()**.

Ao executar **toString()** em uma classe **Usuario** que não sobrescreveu o método, você obterá algo mais ou menos assim: Usuario@0f4b10. A composição original do método **toString()** é, portanto, o nome da classe, um @ e a representação absoluta do hexadecimal do hash code do objeto da classe.

Podemos melhorar bastante essa implementação sobrescrevendo o método e adicionando informações que realmente auxiliem os usuários a visualizar este objeto como uma **String**. Considerando que a classe **Usuario** tenha um atributo **email**, poderíamos utilizá-lo, visto que é uma informação única. E pra deixar mais completo, podemos adicionar também o atributo **nome**, resultando em "John Doe (john-doe@email.com)", por exemplo. Veja a **Listagem 2**.

Listagem 2. Implementando toString().

```
// Implementação nativa de toString()
@Override
public String toString() {
  return getClass().getName() + "@"
    + Integer.toHexString(hashCode());
}

// Implementação mais descritiva
@Override
public String toString() {
  return nome + " (" + email + ")";
}
```

Definir o que é uma boa representação visual depende bastante da classe que você está criando e do que é importante nela. Por outro lado, uma representação visual ruim tem alguns elementos comuns. São eles:

- **null**: elementos que podem ser nulos atrapalham a compreensão do texto e alguns até podem causar **NullPointerException**. No caso da classe **Usuario**, se o e-mail pudesse ser nulo, um dos possíveis retornos seria "John Doe (null)". Qual informação está faltando para preencher o **null**? Você sabe, mas o usuário da sua classe não;
- **Falta de clareza**: numa classe **Retangulo**, você pode ter as propriedades **largura** e **altura**. Se a **String** retornada não evidenciar

o que cada valor significa, o retorno será de pouca utilidade. Portanto, prefira "largura=2, altura=1", ao invés de "2, 1";
• **Excesso de informações:** adicionar muitas informações no retorno polui a visualização e atrapalha a compreensão. Algumas bibliotecas, como a Commons Lang, da Apache, possuem métodos que são capazes de retornar todos os atributos e valores de um objeto (como **ToStringBuilder.reflectionToString(Object)**). Embora isso seja muito bom para debug, é péssimo como representação visual de objetos. Lembre-se: adicione apenas informações importantes;
• **Informações de segurança**: nunca adicione senhas, chaves de acesso ou quaisquer outras informações relacionadas à autenticação ou autorização.

toString() e a interface gráfica da sua aplicação

Muito cuidado com a forma como você utiliza o método **toString()**. É muito comum vê-lo sendo utilizado em interfaces visuais da aplicação para exibir informações. Também é muito comum que ele seja sobrescrito pensando no formato que o usuário da aplicação deseja visualizar os dados. Não faça isso.

A principal função do método **toString()** é representar visualmente o objeto para programadores, não para usuários. Embora ele seja comum e facilmente invocado na camada visual da sua aplicação, recomenda-se que isto não seja feito.

Quando seus objetos interagem com a camada visual de uma aplicação, os atributos dele podem ser exibidos e formatados utilizando os recursos disponibilizados por ela. Com tags comuns do JSP, por exemplo, você pode exibir os atributos de um objeto e formatá-los facilmente, sem a necessidade de fazê-lo nas classes de base da sua aplicação.

Muitas vezes, não é necessário formatar as informações, como datas e valores numéricos, ao sobrescrever **toString()**. Por outro lado, na camada visual, estas informações não só devem ser formatadas como são sensíveis ao **Locale** atual. Fique atento à isso.

Criando um formato e estabelecendo um contrato

Cada classe tem sua forma de representar visualmente os objetos dela. Nos exemplos citados anteriormente, as informações mais relevantes do objeto são exibidas de forma que, ao visualizar, já possamos compreender o objeto e sua composição.

Com as classes nativas do Java isso não é diferente, mas elas vão além. É adicionado ao método **toString()** sobrescrito uma documentação que descreve o formato do texto que será retornado. Portanto, ao utilizar estas classes e invocar o método **toString()** nos objetos dela, você já tem uma ideia de como será o retorno.

Documentar o método em suas classes é uma decisão sua. No momento em que você define o formato em que as informações serão retornadas, você está se comprometendo com os usuários da sua classe, informando que eles podem confiar no valor que será retornado.

Ao firmar um contrato, os usuários da classe podem depender do retorno do método e tornar as características dele parte da lógica do seu próprio código. Eles podem, por exemplo, tentar interpretar a **String** para extrair informações. Outros podem optar por persistir a informação num arquivo ou banco de dados para que possam recuperar esta informação depois.

Portanto, ao estabelecer um contrato, siga-o para que a compatibilidade não seja quebrada. Pense na classe **Locale**. É garantido que o máximo de caracteres do retorno será cinco. Assim, você pode seguramente armazenar esta informação em um banco de dados, numa coluna com tamanho cinco. Se uma atualização mudar isso e houver a possibilidade de retornar mais caracteres, digamos seis, a compatibilidade com seu código será quebrada e você terá vários erros de banco de dados que talvez não possam ser solucionados facilmente.

Esses exemplos sempre citam situações em que o usuário pode querer criar um objeto a partir da **String** retornada pelo método **toString()**. Tornar isso possível é bastante simples pois, ao estabelecer um formato, você pode criar um **factory method** ou construtor na sua classe que receba esta **String** e gere um novo objeto. Veja alguns exemplos em classes do Java:
• **Locale** possui um construtor que faz exatamente isso: recebe uma **String**, no formato retornado pelo método **toString()**, e gera um novo objeto;
• **Integer** possui um método chamado **valueOf()**, que recebe uma **String**. Caso o conteúdo seja um número, um novo **Integer** com aquele valor é retornado;
• Todos os **enums** possuem um método **valueOf()** que retorna um de seus itens baseado na **String** informada.

A **Listagem 3** propõe uma implementação de um factory method **valueOf()** que é capaz de criar um novo objeto da classe **Email** a partir de uma **String**. Para isso, o formato do texto do argumento **String** deve ser igual ao que foi definido e é retornado pelo método **toString()** da classe **Email**. Esta consistência é importante porque reforça o formato da informação e torna a conversão bilateral.

Estabelecer um formato e permitir a criação de um objeto a partir dele é completamente opcional, mas deve ser considerado e implementado sempre que possível.

Pense no usuário: um formato fixo será útil para ele? Haverá benefícios em criar mecanismos para interpretar uma **String** e retornar uma nova instância da classe? Se a resposta para essas respostas for sim, o esforço valerá à pena.

Definindo uma identidade para os objetos

Os métodos **equals()** e **hashCode()** definem a identidade de um objeto durante a execução de uma aplicação. A implementação padrão de cada um deles possui um conceito bastante simples de como identificar um objeto, mas você pode expandir esse conceito nos objetos de suas classes.

A identidade dos seus objetos pode ser definida de várias formas, mas vale lembrar que também existe um contrato a ser respeitado. Neste tópico, você vai aprender como implementar os métodos **equals()** e **hashCode()** com o máximo de qualidade e praticidade, enquanto respeita o contrato estabelecido pela especificação.

Listagem 3. toString() e valueOf(): estabelecendo um contrato.

```
public class Email {

  private final String id;
  private final String dominio;

  public Email(String id, String dominio) {
    this.id = id;
    this.dominio = dominio;
  }

  /** Retorna o e-mail no formato <id>@<dominio> */
  @Override
  public String toString() {
    return id + "@" + dominio;
  }

  /**
   * Cria uma nova instância de e-mail a partir de uma String com o
   * formato definido em #toString()
   */
  public static Email valueOf(String email) {
    // Validações para verificar se a String é válida viriam aqui.
    final String[] partes = email.split("@");
    return new Email(partes[0], partes[1]);
  }

  // Getters e setters...
}
```

A implementação de **equals()** e **hashCode()** de sua classe será colocada à prova quando os objetos dela começarem a interagir com as classes da API Collections. As coleções utilizam as identidades dos objetos para controlar o conteúdo delas e localizar itens previamente inseridos.

Uma implementação ruim ou a violação do contrato podem ter consequências terríveis no comportamento das suas classes e das classes que interagem com elas. Por exemplo, uma coleção pode não conseguir localizar um objeto que você acabou de adicionar.

Para evitar uma implementação incorreta e propensa a erros, a melhor sugestão é não implementar os métodos. Dessa forma, o comportamento padrão será utilizado e, embora seja muito restrito, ele será suficiente para alguns casos.

O método equals()

O método **equals()** indica se um objeto X é "igual" a um objeto Y. No entanto, esta igualdade é um conceito específico para cada classe que implementa este método, como veremos a seguir.

Por padrão, o método **equals()** de **Object** utiliza apenas operador == para comparações. Portanto, a não ser que você sobrescreva o método, dois objetos são considerados iguais apenas se as duas referências apontam para o mesmo objeto.

Por outro lado, algumas classes podem considerar válida a igualdade lógica. A classe **Integer**, por exemplo, considera duas instâncias logicamente iguais quando o valor **int** armazenado nelas é igual. Este é um comportamento exclusivo da classe **Integer**, pois esta é a implementação dela do método **equals()**.

Outro exemplo é a classe **String**. A implementação do método **equals()** dela diz que dois objetos são iguais quando possuem o mesmo conteúdo (caracteres).

Você também pode fazer isso em suas classes. Para a classe **Usuario**, por exemplo, duas instâncias podem ser consideradas logicamente iguais quando os atributos **email** são iguais.

Outro motivo para sobrescrever **equals()** é quando você pretende utilizar o objeto como chave num **Map**. Neste caso, é imprescindível que sobrescreva tanto ele quanto **hashCode()**, como veremos em breve. Se a identidade não for bem definida, os objetos que você inserir em um **Map** podem nunca ser recuperados.

Eis uma lista de razões para você sobrescrever **equals()**:

- Quando sua classe deve ter a noção de igualdade lógica;
- Quando a superclasse da sua classe não implementa **equals()** ou quando a implementação dela não atende às necessidades de sua classe;
- Quando os objetos de sua classe interagem com classes que utilizam o **equals()** para controlar o conteúdo dos objetos delas. **Map** e **Set** são ótimos exemplos.

O contrato de equals

O contrato do método **equals()** é um conjunto de regras que deve ser respeitado tanto pelo fornecedor quanto pelo consumidor do método, de forma a garantir um comportamento constante na interação dos objetos por meio dele.

Este contrato pode ser encontrado na especificação do método. O parágrafo e os marcadores a seguir são uma tradução literal da documentação original, em inglês.

O método **equals()** implementa uma relação de equivalência entre referências não nulas de objetos. Ele é:

- **Reflexivo:** Para qualquer referência não nula de **x**, **x.equals(x)** deve retornar **true**;
- **Simétrico:** Para quaisquer referências não nulas de **x** e **y**, **x.equals(y)** deve retornar **true** somente se **y.equals(x)** retornar **true**;
- **Transitivo:** Para quaisquer referências não nulas de **x**, **y**, **z**, se **x.equals(y)** retornar **true** e **y.equals(z)** retornar **true**, então **x.equals(z)** deve retornar **true**;
- **Consistente:** Para quaisquer referências não nulas de **x** e **y**, múltiplas invocações de **x.equals(y)** consistentemente retornam **true** ou consistentemente retornam **false**, desde que nenhuma informação utilizada no **equals()** tenha sido modificada;
- ***Non-nullity*:** Para qualquer referência não nula de **x**, **x.equals(null)** deve retornar **false**.

Parece complicado? Pode ter certeza que sim. Como já foi dito, implementar **equals()** corretamente é muito mais difícil do que simplesmente criar o método na sua classe.

Nem mesmo as classes do Java estão livres de erros de implementação e violação de regras, mas a maioria obedece ao contrato à risca. Ao desrespeitar qualquer uma dessas regras, você pode fazer seu programa se comportar de forma estranha e imprevisível.

Nota

Lembre-se: se você violar as regras do contrato, você não sabe como outros objetos vão se comportar quando forem confrontados com seus objetos.

Dito isso, vamos entender os itens do contrato e ver como cada um se aplica.

Reflexivo

Um objeto deve ser igual a ele mesmo. A não ser que você realmente queira implementar o método incorretamente (e por que você faria isso?), você dificilmente violará esta regra.

Simétrico

Dois objetos devem estar de acordo sobre sua igualdade. Diferente da primeira regra, esta é bem fácil de quebrar. Um exemplo simples é quando você implementa o método **equals()** de forma que ele aceite objetos de outros tipos.

Suponha que você queira implementar o método **equals()** da sua classe **Usuario** de forma que ele aceite um objeto do tipo **String**. A implementação original diz que se dois objetos do tipo **Usuario** possuem o atributo **email** igual, eles são logicamente iguais. Visto que **email** é do tipo **String**, você decide aceitar um objeto deste tipo e adapta o método.

Embora uma **String** qualquer, cujo conteúdo seja igual ao do atributo **email** de um objeto da classe **Usuario**, seja considerada logicamente igual ao objeto **Usuario**, o contrário nunca será verdadeiro. E embora o método **equals()** da classe **String** aceite o objeto **Usuario**, ele sempre retornará **false**, pois qualquer objeto que não seja do tipo **String** é descartado.

Vamos ver este cenário na **Listagem 4**.

Listagem 4. Violação de simetria de equals().

```
final Usuario usuario = new Usuario("John Doe", "johndoe@email.com");
final String email = "johndoe@email.com";

// Retorna true. O método equals() de Usuario foi implementado pra
// trabalhar com String.
usuario.equals(email);

// Retorna false. String desconhece Usuario e apenas vai analisar a
// igualdade se o objeto for uma String.
email.equals(usuario);
```

Esta listagem exibe um cenário incomum, mas que demonstra como a simetria pode ser quebrada.

Um exemplo real de violação de simetria na API do Java, evidenciado inclusive na especificação, encontra-se na classe **Timestamp**. Ela é uma subclasse de **Date** e adiciona um atributo próprio (**nanoseconds**) na sobrescrita do **equals()**. Com isso, enquanto um objeto do tipo **Timestamp** pode ser considerado igual a um objeto do tipo **Date**, quando eles possuem o mesmo número de milissegundos (desde 01/01/1970 00:00:00 GMT), o contrário não é verdadeiro. Mesmo que os milissegundos sejam iguais, o objeto **Date** não possui o atributo **nanoseconds** para ser comparado.

Transitivo

A terceira regra diz que se o primeiro objeto é igual ao segundo e o segundo é igual ao terceiro, então o primeiro é igual ao terceiro. Esta regra também pode ser violada facilmente.

Imagine uma classe **Retangulo** que tem apenas duas informações: **altura** e **largura**. A implementação de **equals()** é bastante simples: se dois objetos possuem a mesma **altura** e **largura**, eles são iguais.

Criamos então a classe **RetanguloColorido**, subclasse de **Retangulo**, que adiciona uma informação importante ao objeto que for criado a partir dela, a **cor**. Se a subclasse não sobrescrever o método **equals()**, objetos com cores diferentes serão considerados iguais. Sobrescrevemos então **equals()**, utilizando a implementação da superclasse, que considera **altura** e **largura**, e adicionamos **cor**.

O problema ao sobrescrever **equals()** na subclasse é que podemos obter resultados diferentes ao comparar instâncias de **Retangulo** com instâncias de **RetanguloColorido** e vice-versa. Considere os objetos instanciados na **Listagem 5**.

Listagem 5. Violação da transitividade de equals() – parte 1.

```
Retangulo r = new Retangulo (2, 3);
Retangulo Colorido rc = new Retangulo Colorido(2, 3, Cor.AZUL);
```

Enquanto **r.equals(rc)** retorna **true**, **rc.equals(r)** retorna **false**, violando o contrato da simetria. Isto acontece porque:

- O primeiro **equals()**, da classe **Retangulo**, desconhece o atributo **cor**. Logo, ele não influencia na comparação e **true** é retornado;
- Já o segundo **equals()**, da classe **RetanguloColorido**, compara o atributo **cor** do objeto que ele recebeu. Visto que o objeto da classe **Retangulo** não possui este atributo, ele é nulo e a comparação retorna **false**.

Uma solução para este problema seria alterar a implementação de **RetanguloColorido.equals()** para verificar se o argumento é um **Retangulo**, ignorando a comparação da **cor** em caso positivo. Com esta mudança, no entanto, quebramos a transitividade. Veja a **Listagem 6**.

Listagem 6. Violação da transitividade de equals() – parte 2.

```
RetanguloColorido r1 = new RetanguloColorido(2, 3, Cor.PRETO);
Retangulo r2 = new Retangulo (2, 3);
RetanguloColorido r3 = new RetanguloColorido(2, 3, Cor.BRANCO);
```

Neste novo cenário, **r1.equals(r2)** e **r2.equals(r3)** retornam **true**, mas **r1.equals(r3)** retorna **false**. As duas primeiras comparações ignoram a cor, mas a terceira considera a informação. Novamente a simetria foi quebrada.

Pensando nos exemplos apresentados com as classes **Timestamp** e **Retangulo**, podemos concluir que não há como estender uma classe, adicionar uma informação relevante na identidade do objeto e preservar o contrato de **equals()** ao mesmo tempo.

Algumas soluções para respeitar o contrato apontam que se você utilizar **getClass()** ao implementar **equals()**, este problema não ocorrerá. Veja a **Listagem 7**.

Listagem 7. Utilizando getClass() na implementação de equals().

```
@Override
public boolean equals(Object o) {
   // Se "o" for nulo ou a classe dele for diferente da classe de "this", retorna false.
   if (o == null || this.getClass() != o.getClass()) {
      return false;
   }
   // Continuação da implementação.
}
```

A validação inicial, a cláusula **if** que verifica se o **Class** de ambos os objetos são iguais, estabelece que apenas objetos da mesma classe podem ser comparados e considerados iguais. Subtipos, portanto, serão sempre diferentes, mesmo que suas propriedades sejam iguais. Embora pareça que estamos reforçando uma regra de negócio, as consequências de ignorar os subtipos são inaceitáveis.

Para demonstrar esse problema, vamos criar uma nova classe, **RetanguloContador**. Ela é subclasse de **Retangulo** e não adiciona qualquer informação nova. A única diferença é que ela contabiliza a quantidade de instâncias criadas.

Considerando a implementação de **equals()** proposta pela **Listagem 7**, um **RetanguloContador** nunca será igual a **Retangulo**, mesmo que a **altura** e **largura** deles sejam iguais. Isso acontece porque o método **getClass()** deles retornam objetos diferentes.

Apesar de a identidade deles ser igual, **altura** e **largura**, eles são considerados diferentes, e isso é um grande problema, visto que este comportamento desobedece ao princípio da substituição de Liskov (veja o **BOX 1**).

Mas este não é o único problema, como veremos a seguir. Para essa demonstração, vamos criar um objeto de cada tipo, **Retangulo** e **RetanguloContador**, e ver como eles funcionam em conjunto com um **HashSet**. Acompanhe a **Listagem 8**.

Listagem 8. Violação da transitividade de equals() – parte 3.

```
1 Retangulo r = new Retangulo (1, 2);
2 Retangulo Contador rc = new RetanguloContador(1, 2);
3
4 HashSet<Retangulo> hashSet = new HashSet<Retangulo>();
5 hashSet.add(r);
6
7 System.out.println(hashSet.contains(rc)); // => false
```

Considerando que os objetos **r** e **rc** possuem a mesma identidade e são do mesmo tipo, **Retangulo**, por que o **HashSet** diz que o objeto **rc** não faz parte da coleção? Porque o **getClass()** impede que um **RetanguloContador** seja considerado igual a um **Retangulo** e o método **HashSet.contains()** utiliza o método **equals()** para testar se a coleção contém o elemento informado.

Qual a solução então? Como podemos estender uma classe e adicionar uma informação importante sem violar os contratos de **equals()**? A melhor resposta é: não estenda, prefira a composição. Ao invés de **RetanguloColorido** ser uma subclasse de **Retangulo**, ele pode ser uma classe por si só e conter um atributo **retangulo**, que armazenará uma instância de **Retangulo**. Sendo assim, instâncias das duas classes nunca serão consideradas iguais.

BOX 1. Princípio da substituição de Liskov

O Princípio da Substituição de Liskov é uma definição particular para o conceito de subtipo que visa garantir a interoperabilidade semântica entre os tipos de uma hierarquia. Este princípio diz que qualquer propriedade importante de um tipo também deve ser importante para os subtipos. Logo, um método escrito para o tipo deve funcionar igualmente bem nos subtipos.

Consistente

A quarta regra diz que se dois objetos são iguais, eles devem ser iguais até que um deles seja modificado.

Embora pareça seguro dizer que objetos imutáveis nunca violarão esta regra, visto que seu estado é constante, muito cuidado. A identidade do objeto será corrompida se ela for composta por recursos que podem ou não estar disponíveis no momento que **equals()** for invocado.

Arquivos que são acessados em servidores remotos e serviços on-line, como a busca de CEP dos Correios, são exemplos de recursos que você nunca deve depender, pois não há como prever a disponibilidade deles.

Non-nullity

A última regra também é bastante simples e dificilmente será violada acidentalmente. Ao invocar o método **equals()** passando **null**, **false** deve ser retornado. Obviamente, nenhum objeto é igual à **null**.

Implementando o método equals()

Após muita discussão sobre o contrato, é hora de demonstrar como implementar o método **equals()** com o máximo de qualidade. No entanto, antes de iniciar a codificação, defina o que torna seu objeto logicamente igual a outro. A implementação vem após essa etapa e ela vai depender das regras de negócio que sua classe deve obedecer.

As dicas a seguir servem para a maioria dos casos, mas você pode precisar fazer algo diferente.

Independente de como escolher implementar **equals()**, sempre respeite o contrato e garanta que todas as regras estão sendo seguidas escrevendo testes unitários para suas classes.

Os passos a seguir representam uma forma bastante prática e reutilizável de implementar o método **equals()**:

1. Utilize o operador == para verificar se o argumento é uma referência ao objeto atual, **this**. Em caso positivo, retorne **true**. Este passo visa evitar passos desnecessários, já que ambos objetos apontam para a mesma referência. Ele é especialmente útil quando as comparações são custosas, como quando é necessário calcular um valor complexo ou percorrer vários itens em uma lista;

2. Use **instanceof** para verificar se o argumento informado é do tipo correto e não nulo. Se a verificação retornar **false**, retorne **false**. Eis o porquê:
   a. Geralmente, o tipo correto é o tipo da classe onde você está implementando o método, mas também pode ser uma interface implementada pela classe. Se for utilizar a interface, certifique-se de que a comparação entre classes que a implementam é viável. Coleções como **Set**, **List**, **Map** e **Map.Entry** fazem uso disso;
   b. Quando o argumento à esquerda do **instanceof** é **null**, a validação retorna **false**. Dessa forma, você protege seu método de lançar uma **NullPointerException**.
3. Transforme (cast) seu objeto para o tipo correto. Como a transformação foi precedida pelo **instanceof**, ela é garantida e não lançará **ClassCastException**;
4. Para cada atributo "importante" da sua classe, verifique se o atributo do argumento é equivalente ao atributo do seu objeto. Se todas as verificações forem bem sucedidas, retorne **true**; caso contrário, **false**. Se o tipo no passo 2 for uma interface, você deverá acessar os atributos por métodos; se o tipo for uma classe, você pode acessar os atributos diretamente, dependendo da visibilidade deles. Sobre os tipos dos atributos:
   a. Para primitivos que não forem **float** ou **double**, utilize o operador ==;
   b. Para **float**, use **Float.compare**; para **double**, **Double.compare**;
   c. Para atributos que são referências a outros objetos, utilize o método **equals()** deles. Certifique-se de que eles não são nulos;
   d. Para **arrays**, se todas as posições forem importantes, utilize **Arrays.equals()**.

Implementando equals() com bibliotecas de terceiros

Uma excelente dica para implementar **equals()** corretamente e livrar-se de várias preocupações é utilizar bibliotecas de terceiros. Os passos de 1 a 3 são mais simples e menos propensos a falhas, mas o passo 4 pode ficar bastante complexo, ilegível e propenso a erros, principalmente a **NullPointerException**.

Foi pensando nisso que as bibliotecas Commons Lang, da Apache, e Guava, do Google, possuem métodos utilitários para fazer estas comparações:

• Commons Lang possui uma classe chamada **EqualsBuilder**, onde você pode adicionar vários atributos de seu objeto e do argumento para comparar. Os argumentos são comparados na ordem que você informou e o método falha o mais rápido possível. Isso quer dizer que, se a primeira validação falhar, as outras não são nem executadas, economizando tempo e processamento. Adicione os atributos, retorne **EqualsBuilder.isEqual()** e pronto;

• Já o Guava optou por utilizar uma forma que mais tarde seria implementada nativamente, no Java 7. **Objects.equals()** é um método que aceita dois argumentos e retorna **true** se eles forem iguais. Você compara atributo por atributo, na ordem que achar melhor, e encadeia as chamadas de **Objects.equals()** utilizando o operador **&&**.

As duas bibliotecas facilitam muito a implementação do método **equals()** e são altamente recomendáveis, por esta e outras facilidades. Veja a **Listagem 9**.

Listagem 9. Implementando equals() com Commons Lang e Guava.

```
// Sempre utilize a anotação @Override.
@Override
public boolean equals(Object o) {
  // Passo 1: a referência é a mesma?
  if (this == o) {
    return true;
  }

  // Passo 2: o é do tipo ou subtipo de Usuario?
  if (!(o instanceof Usuario)) {
    return false;
  }

  // Passo 3: neste ponto, é seguro fazer casting.
  final Usuario outro = (Usuario) o;

  // Passo 4 + Apache Commons Lang.
  return new EqualsBuilder().append(email, outro.email).isEqual();

  // Passo 4 + Google Guava.
  return Objects.equals(email, outro.email);
}
```

Embora a maioria das IDEs forneça a opção de sobrescrever **equals()** automaticamente, a implementação deles é, em linhas gerais, muito complexa e desnecessariamente verbosa. Portanto, prefira utilizar as bibliotecas recomendadas aqui.

Sobrescreva, não sobrecarregue

A última dica ao sobrescrever **equals()** é: não o sobrecarregue. É muito fácil errar a assinatura do método e sobrecarregar o método ao invés de sobrescrevê-lo. O impacto disso é terrível: as outras classes sempre utilizam o método de **Object**, nunca a sobrecarga.

A assinatura correta do método **equals()** é **public boolean equals(Object)**. Não são raras as vezes que você verá uma assinatura parecida com esta: **public boolean equals(Usuario)**. Embora funcione e testes simples atestem que o método foi implementado corretamente, classes que implementam **Set** e **Map** invocarão a assinatura original, que recebe um argumento **Object**, e que apenas verifica se as referências apontam para o mesmo objeto (lembre-se, ele não foi sobrescrito).

Para evitar este problema, é simples: utilize a anotação **@Override**. Ela garante, em tempo de compilação, que o método de fato está sendo sobrescrito.

O método hashCode()

Se você sobrescreveu **equals()**, você deve sobrescrever **hashCode()**. Se você não fizer isso, você estará violando o contrato do método **hashCode()**. De acordo com a especificação oficial:

• Sempre que for invocado em um mesmo objeto durante a execução de uma aplicação Java, o método **hashCode()** deve retornar

consistentemente o mesmo inteiro, considerando que nenhuma informação utilizada nas comparações de **equals()** tenha sido modificada. No entanto, o número inteiro não precisa ser o mesmo entre a execução de uma aplicação e outra execução dessa mesma aplicação;

• Se dois objetos são considerados iguais de acordo com a implementação do método **equals(Object)**, então invocar **hashCode()** em cada um dos objetos deve produzir o mesmo número inteiro;

• Não é requerido que dois objetos considerados diferentes de acordo com a implementação do método **equals(Object)** devam retornar números inteiros distintos ao invocar **hashCode()**. No entanto, é importante que se saiba que gerar números inteiros diferentes para objetos diferentes pode aumentar o desempenho de tabelas hash.

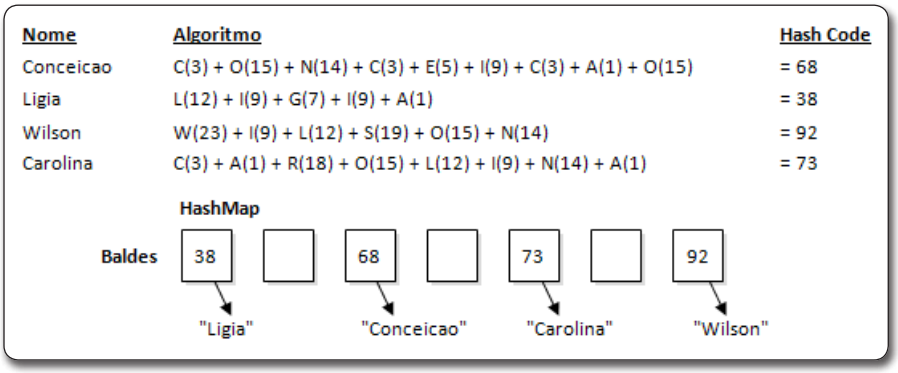


Figura 1. Os baldes recebem os nomes cujo hash code seja igual ao número deles

A segunda regra é a mais importante: objetos iguais devem retornar números inteiros iguais. Não respeitar esta regra pode fazer com que os objetos de sua classe fiquem perdidos dentro de um **HashMap**, por exemplo.

Ainda segundo a especificação, ao invocar **hashCode()** numa classe que não sobrescreveu o método, o valor retornado é, geralmente, o endereço interno do objeto no formato de um número inteiro. Esse comportamento é particular da implementação da JVM que será utilizada para executar o programa. No fim das contas, o que isso significa? Que você nunca deve depender da implementação padrão.

Para que serve o hashCode()?

Hash codes geralmente são utilizados para melhorar o desempenho em grandes coleções de dados. Coleções como **HashMap** e **HashSet** utilizam o hash code de um objeto para determinar onde o objeto será armazenado na coleção. Ele também é utilizado para ajudar a localizar o objeto na mesma coleção.

Para entender como os hash codes são utilizados para armazenar e encontrar objetos em uma coleção, pense em uma série de baldes no chão, cada um com um número inteiro escrito. Alguém entrega para você um papel com um nome escrito. Você pega esse papel e calcula um número inteiro para o nome, considerando que A é 1, B é 2 e assim por diante. Somando todos os números correspondentes às letras, você obtém o número inteiro para aquele nome.

O número inteiro calculado, o hash code, vai ser utilizado para determinar em qual balde o papel deve ser armazenado. Calculado o número, você localiza o balde e coloca o papel dentro dele. Quando alguém chegar até você e pedir um nome, você simplesmente calcula o número inteiro e vai até o balde cujo número é igual ao do resultado procurar o papel.

A **Figura 1** ilustra os baldes numerados e a lógica que calcula os hash codes.

Então, cada balde tem um nome, certo? Na realidade, não. Dois nomes diferentes podem resultar em números inteiros iguais. Isso pode ocorrer com qualquer anagrama, como por exemplo, Nadia e Diana. Mas isso não é um problema. Independente do nome que solicitarem para você, ao calcular o número inteiro, você saberá em que balde está.

Portanto, o hash code te diz em que balde o papel está, mas não como localizar o nome, uma vez que você está procurando dentro do balde. Quem vai te ajudar a localizar o papel dentro do balde é o **equals()**.

Logo, encontrar um papel é um processo em duas etapas:

1. Localizar o balde (**hashCode()**);
2. Localizar o papel dentro do balde (**equals()**).

Para ser mais eficiente, você vai querer distribuir os papéis que receber no maior número de baldes possíveis. Lembre-se que é mais fácil encontrar um papel num balde com dois papéis do que com cem papéis. Se você simplesmente colocasse todos os papéis no balde 17, por exemplo, imagine quantos papéis você teria que olhar para encontrar o correto?

Demorar a encontrar um papel é ruim quando o assunto é desempenho. Mas a situação pode ser pior. Imagine que, ao calcular o número inteiro, uma letra tenha sido ignorada por engano. Deste modo, você procuraria o papel no balde errado e nunca o encontraria.

Considerando este exemplo, é mais fácil entender a segunda regra do contrato para implementação de **hashCode()**, que diz: se dois objetos são iguais segundo **equals()**, os números inteiros retornados ao invocar **hashCode()** neles também devem ser iguais.

Implementando o método hashCode()

Calcular um hash code não é simples e pode ser feito de diversas formas. Uma das opções mais populares é a que Joshua Bloch descreve em seu livro, Effective Java, 2nd Edition. Ainda assim, ela é bastante complexa.

Mas, ao contrário da implementação de **equals()**, esta pode ser simplificada ao extremo quando utilizamos bibliotecas de terceiros. Novamente, Commons Lang e Guava te ajudarão bastante (observe o exemplo na **Listagem 10**). Vejamos como:

• Commons Lang possui a classe **HashCodeBuilder**, que funciona de forma muito simples: você instancia a classe e adiciona os atributos da sua classe que irão compor o hash code. Feito isso, basta retornar **toHashCode()**;

• Guava tem o método **Objects.hashCode()**. Este recebe um *varargs* no qual você pode passar todos os atributos que deseja e pronto, ele já retorna o número inteiro.

Listagem 10. Implementando hashCode() com Commons Lang e Guava.

```
@Override
public int hashCode() {
 // Apache Commons Lang.
 return new HashCodeBuilder().append(email).toHashCode();

 // Google Guava.
 return Objects.hashCode(email);
}
```

Nota

Se você estiver utilizando o Java 7, a melhor opção é Objects.hash().

Mais uma vez você irá economizar muito tempo se utilizar estes métodos utilitários, além de não precisar se preocupar com valores nulos. A implementação será extremamente objetiva e concisa.

Assim como acontece com **equals()**, as IDEs também fornecem opções para implementar **hashCode()**. No entanto, mais uma vez, prefira as bibliotecas.

Composição de equals() e hashCode()

Os contratos **equals()** e **hashCode()** não estabelecem quais atributos da sua classe devem compor os métodos. Você tem total liberdade para escolher aqueles que são relevantes às necessidades das regras de negócio de sua aplicação.

Mesmo assim, fique atento a algumas recomendações que vão ajudá-lo a selecionar quais atributos são mais adequados para sua implementação. Lembre-se que você tem a opção tanto de evitar um atributo que possa causar problemas no futuro como transformá-lo para que ele possa ser usado com segurança.

Saber quais atributos utilizar para implementar **equals()** e **hashCode()** é o primeiro passo para definir a composição da identidade do objeto.

Atributos importantes

São atributos importantes para implementação do **equals()** todos aqueles que compõem a identidade dos objetos da sua classe. A classe **Usuario** utilizava **email**, uma classe **PessoaFisica** pode utilizar **cpf**, etc. Se a(s) informação(ões) torna(m) seu objeto único, você deve utilizá-las.

E quais atributos da sua classe são importantes para calcular o **hashCode**? Simples: todos aqueles que você utilizou para implementar o **equals()**. Esta é uma das formas mais simples de garantir que todos os contratos sejam respeitados e que as implementações sejam consistentes.

Atributos transient

Nunca utilize atributos **transient** (veja o **BOX 2**) na implementação do **equals()** ou do **hashCode()**, pois não é possível garantir se estes valores estarão disponíveis quando você executar um dos métodos.

Nota

Se sua classe for imutável e o cálculo do hash code for custoso, você pode considerar fazer cache do valor calculado ao invés de executar o cálculo toda vez que for necessário. A classe String faz isso. No entanto, se o ganho de desempenho for insignificante, não use caching.

BOX 2. O modificador transient

Ao marcar um atributo de classe com o modificador transient, você está dizendo para a JVM ignorá-lo ao tentar serializar objetos desta classe. Serialização é um processo que permite que você salve o estado (valor dos atributos) de um objeto em um I/O stream, como num arquivo, por exemplo.

Imagine que a classe **Usuario** tenha utilizado o atributo **transient dataNascimento**, do tipo **Date**, na implementação de **equals()** e **hashCode()**. O objeto se comporta como esperado e todas as regras são respeitadas. Seguro da implementação, você utiliza este objeto como chave de um **Map**.

Digamos que a invocação de **hashCode()** retorne 23. Utilizando o exemplo dos baldes, o objeto seria adicionado no balde de número 23. Neste exato momento, você consegue localizar o objeto e seu valor.

Em seguida, você serializa o objeto da classe **Usuario** e recupera-o (ver **Listagem 11**). Os atributos com o modificador **transient** não são serializados, logo não são recuperados e o objeto recuperado contém o valor padrão deles. Para **boolean**, teríamos **false**; para **int**, 0. No caso de um objeto, como um atributo **Date**, o valor é **null**.

Supondo que sua implementação não tenha sido cuidadosa, neste ponto você pode ter um **NullPointerException**, já que um dos atributos utilizados para implementar **equals()** e **hashCode()** está nulo.

Se sua implementação está segura dessa exceção, você irá invocar **hashCode()** e desta vez o valor 19 será retornado (observe novamente a **Listagem 11**). A ausência de um atributo causou a alteração do valor retornado. Sendo assim, quando você tentar localizar este objeto no **Map**, ele nunca será encontrado, pois ele será procurado no balde de número 19, mas está no balde 23. Ou seja, seu objeto, a chave, e consequentemente o valor associado à ele, ficarão perdidos até que você crie um outro objeto que seja logicamente igual àquele adicionado no **Map**.

Por fim, ainda que o balde certo seja localizado, seu objeto não será encontrado nele. A implementação do método **equals()**, utilizada para localizar o papel dentro do balde, compara as datas de nascimento e o valor do atributo **dataNascimento** do seu objeto é diferente do atributo **dataNascimento** do objeto contido na coleção.

Objetos e atributos mutáveis

Você terá o mesmo problema se os atributos utilizados em **equals()** e **hashCode()** forem mutáveis. Suponha que o usuário da aplicação tenha meios de atualizar o valor de sua **dataNascimento** (uma interface visual, por exemplo). Ele preencheu uma data, ela foi registrada e o objeto **Usuario** armazenado em um **Set**. Em

seguida, o usuário altera a data. Deste modo, novamente o hash code será diferente (veja a **Listagem 12**).

Listagem 11. Atributos transient na implementação do hashCode().

```
final Usuario usuário = new Usuario("John Doe", "johndoe@email.com");

// Para simplificar o exemplo, imagine que ele faça um parse da data.
usuario.setDataNascimento("2006-06-06");

// Retorna 23, pois o valor da data de nascimento é contabilizada.
usuario.hashCode();

// Serializamos e criamos novamente o objeto; a data de nascimento,
// sendo transiente, não é recuperada.
serializaUsuario(usuario);
usuario = deserializaUsuario();

// Retorna 19, pois a data de nascimento é nula e portanto ignorada.
usuário.hashCode();
```

Listagem 12. Atributos mutáveis na implementação do hashCode().

```
usuario.setDataNascimento("2006-06-06");
usuario.hashCode(); // => 23
hashSet.add(usuario); // usuario é adicionado normalmente

usuario.setDataNascimento("2009-09-09");
usuario.hashCode(); // => 27
hashSet.contains(usuario); // retorna false; usuario não foi encontrado
```

Ao tentar localizar o objeto no **Set**, o balde de número 27 será erroneamente vasculhado e, portanto **false** será retornado. O objeto somente será recuperado quando o valor original for atribuído.

Nesse cenário, se **dataNascimento** não fosse considerado na implementação do **hashCode()**, a alteração dele não teria impacto no cálculo e o balde correto seria vasculhado. Mas o que aconteceria se **equals()** utilizasse este atributo em sua implementação? Ao chegar no balde correto, o **Set** não conseguiria encontrar o objeto, pois a data é diferente: ora é 2006, ora 2009.

Em ambas as situações, a presença de um atributo mutável na implementação de **equals()** e **hashCode()** interferiu no comportamento desejado dos métodos de **Set**. Ou seja, os contratos dos métodos podem até estar sendo seguidos, mas a interação dos seus objetos com objetos de outras classes tornou-se imprevisível e propenso a erros.

Identificar este problema quando ele está isolado é fácil. Contudo, dentro de uma aplicação, cercado por outros objetos e códigos de regra de negócio, este é um problema muito difícil de diagnosticar. Portanto, evite atributos mutáveis ao implementar **equals()** e **hashCode()** – ver **BOX 3**.

Lista rápida das recomendações

Como pudemos ver, existem vários detalhes para os quais você deve estar atento ao selecionar os atributos que irão compor os

métodos **equals()** e **hashCode()**. Portanto, lembre-se sempre destas recomendações:

- Nunca utilize atributos **transient**;
- Dê preferência a objetos imutáveis, ou ao menos torne os atributos utilizados na implementação de **equals()** e **hashCode()** imutáveis;
- Crie testes para validar sua implementação;
- Se existe um cenário onde seu objeto pode ser serializado e recuperado, adicione-o ao teste;
- Se for utilizar seu objeto como a chave de um **Map**, ou como elemento em um **Set**, adicione este cenário ao teste.

BOX 3. HashSet e hashCode()

É muito importante deixar claro como a classe HashSet funciona. Esta classe calcula o hash code e determina em que "balde" seu objeto será alocado no momento que o método add(Object) é executado. Se o valor do hash code for atualizado (o valor de um atributo utilizado no cálculo mudou, por exemplo), o objeto permanecerá no balde errado. HashSet não irá rever a alocação do seu objeto.

Todos estes detalhes foram considerados na implementação da classe **String**. Por isso é tão comum ver programadores utilizando os objetos dela como chave para um **Map**. A classe **java.util.Properties** é um excelente exemplo disso. A classe **String**:

- É segura e confiável;
- É imutável;
- Não pode ser estendida;
- Para que uma **String** seja considerada logicamente igual à outra, todos os caracteres que a compõe devem ser iguais.

Assim como **String**, as classes nativas do Java são e devem ser utilizadas como exemplo para as classes que você criar. Portanto, consulte o código delas e explore como foi elas foram implementadas.

Você vai encontrar desde trechos de código bem simples até outros bem complexos. Vai perceber que elas são compostas por muitos outros métodos que não fazem parte da API pública. E, mais do que saber pela documentação, você vai ver e aprender como estas classes implementam **equals()** e **hashCode()**, um conhecimento que será de grande utilidade ao implementar estes métodos em suas classes.

Conhecer o pacote **java.lang** à fundo é um diferencial. Você terá um conhecimento a mais, que alguns descartam por focarem muito nas exigências do mercado, e irá se destacar entre os outros programadores. Nenhuma (ou quase nenhuma) vaga solicita conhecimento profundo na implementação dos métodos **equals()** e **hashCode()**, mas acredite: você vai agregar muito à equipe se tiver este conhecimento. Deste modo, não o ignore e não se deixe levar pelas tendências: é melhor saber muito de pouco que pouco de muito.

A certificação OCPJP6 sustenta essa afirmação. Grande parte dos tópicos diz respeito às classes presentes no pacote **java.lang**. As questões testam se o candidato conhece como as classes funcionam e os detalhes das suas implementações. Apenas saber instanciá-las não é o suficiente.

Sendo assim, saiba o porquê uma implementação é mais adequada para a solução de um problema do que a outra. Leia a especificação das classes que você utiliza e fique atento aos contratos. Dessa forma você garante a qualidade das suas classes, bem como uma interação harmoniosa com as classes do Java.

Autor

William Brombal Chinelato
willchinelato@gmail.com – https://github.com/willchinelato
Programador há 10 anos, sendo seis de Java. Certificações OCPJP, OCPWCD e OCPBCD. Minha paixão por games me levou à programação e desenvolvimento.

JSch: Desenvolvendo aplicações com Java Secure Channel

Aprenda a transferir arquivos de forma rápida e segura

Com a evolução do mercado, a competitividade acirrada e a alta demanda das empresas, surgem necessidades que exercem grande pressão sobre a área de TI, exigindo flexibilidade e agilidade para a solução de problemas. Devido a isto, as empresas estão em busca de tecnologias e ferramentas que tornem o desenvolvimento e gerenciamento de aplicações mais ágeis.

Atualmente, existem inúmeros frameworks disponíveis que auxiliam no desenvolvimento de aplicações web. Assim, é importante saber escolher qual framework poderá auxiliá-lo, levando em conta aspectos como redução de custos, segurança, documentação, detecção de erros, etc. A adoção de um bom framework poderá ajudá-lo a ter maior produtividade e eliminar tarefas repetitivas.

Além disso, com a grande troca de informações entre diversos tipos de sistemas, há a necessidade de oferecer segurança sobre essas informações, requisito fundamental que, por muitas vezes, é sucumbido pela necessidade de entregar o software na data contratada. Entretanto, no desenvolvimento de um projeto, é preciso saber conciliar a segurança e os demais requisitos, independente de qual seja o objetivo da aplicação. Neste cenário, para as soluções que necessitam efetuar a troca de arquivos, conheceremos o JSch.

O Java Secure Channel, ou JSch, foi desenvolvido pela empresa japonesa JCraft com o intuito de permitir que usuários pudessem desfrutar de sessões seguras em suas transferências de arquivos. Desta forma, a JCraft decidiu criar um framework baseando-se nos mecanismos de segurança do protocolo SSH2, tornando as sessões criptografadas e, portanto, seguras.

Neste artigo, utilizaremos o JSch e veremos o funcionamento de alguns dos seus recursos oferecidos para realizar a transferência de arquivos. Para isso, desenvolveremos uma aplicação web trabalhando conjuntamente com o PrimeFaces. Este framework disponibiliza uma grande gama de componentes para se trabalhar com o JavaServer Faces (JSF), inclusive componentes para a realização de download e upload de arquivos, como é o caso da classe **FileUploadEvent**. No entanto, iremos utilizá-lo em conjunto com o JSch para implementar a transferência segura de arquivos, pois apenas com o PrimeFaces não é possível realizar a transferência com a utilização do SSH2.

Fique por dentro

Este artigo apresenta o framework JSch, solução que permite realizar a transferência de arquivos de forma segura utilizando o protocolo SSH2. Sendo assim, este artigo é útil para desenvolvedores que desejam utilizar e aprofundar-se sobre as funcionalidades dos recursos de transferência de arquivos. Demonstraremos na prática os componentes de download e upload fornecidos pelo framework JSch, trabalhando conjuntamente com o PrimeFaces.

O que é SSH?

O Secure Shell (SSH) teve sua primeira versão desenvolvida por Tatu Ylonen em 1995 com o objetivo de obter uma ferramenta que pudesse desfrutar de acessos remotos com segurança. A segunda versão, o SSH2, foi desenvolvida com a inclusão de ferramentas que visaram evitar falhas de segurança identificadas na primeira versão e foi padronizado pelo Internet Engineering Task Force (IETF) em 2006. Este protocolo auxilia na comunicação entre hosts que utilizam a arquitetura cliente-servidor. Ele implementa a criptografia para estabelecer conexões seguras. Sendo assim, as informações transmitidas de um host para outro ficam disponíveis apenas entre eles, inibindo o acesso à informação por desconhecidos. No momento em que os dados são enviados

para outro host, o SSH os criptografa automaticamente e quando chegam ao destino, são descriptografados. A conexão realizada entre os hosts é feita com a utilização de chaves de 512 bits; sendo assim, a possibilidade destes dados serem acessados por hosts não autorizados é extremamente remota.

Entre as características que tornaram o SSH popularmente conhecido, podemos destacar: sua fácil instalação e configuração, estabilidade, segurança, é suportado em diversos sistemas operacionais e possui muitos recursos para manutenção. A **Figura 1** apresenta um exemplo simples de comunicação entre cliente e servidor com o canal SSH encriptado.

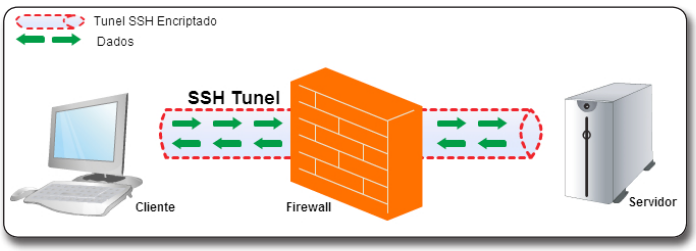


Figura 1. Exemplo de comunicação entre cliente e servidor com túnel encriptado

Criptografia e Autenticação no SSH

Na criptografia, são utilizadas chaves públicas e privadas para criptografar e descriptografar os dados. A chave pública fica disponível a qualquer entidade que deseja utilizá-la. Por outro lado, a chave privada é confidencial ao seu proprietário. Por exemplo, se o cliente deseja enviar algumas informações sigilosas para o servidor, e quer que estas informações sejam lidas apenas por ele, o cliente irá criptografar os dados com a chave pública do servidor que está disponível para uso por qualquer entidade. Após receber os dados, o servidor irá descriptografá-los com sua chave privada, que é confidencial e apenas ele tem acesso. Caso o servidor deseje responder à mensagem recebida, deverá realizar o mesmo processo, porém irá criptografar os dados com a chave pública do cliente, e após o cliente receber a resposta, irá descriptografá-la com a sua chave privada.

A autenticação entre cliente e servidor pode ser efetuada das seguintes maneiras: autenticação de chaves públicas, autenticação por teclado interativo e autenticação de usuário e senha. Na autenticação de chaves públicas é necessário que a chave pública do cliente esteja presente na lista de chaves permitidas, armazenada em um diretório no servidor. Deste modo, após o servidor verificar se a chave pública fornecida pelo cliente existe na lista de chaves permitidas, é realizada a validação desta chave e a comunicação é autenticada.

A autenticação por teclado interativo realiza várias perguntas que somente o usuário tem capacidade para responder. Para ser autenticado, basta que o cliente responda as perguntas corretamente. Outra opção é a autenticação através de usuário e senha. Neste caso, no entanto, para evitar que o sistema fique vulnerável a ataques, também é recomendado que o mecanismo de autenticação de chaves públicas seja utilizado.

O que é SFTP?

O SSH File Transfer Prococol, ou SFTP, foi desenvolvido para ser utilizado em conjunto com o SSH, visando realizar a transferência segura de arquivos e impedir que usuários não autorizados possam ter acesso às informações. Apesar disso, também se destina a funcionalidades como: retomar transferências interrompidas, remoção de diretórios remotos e listagem de diretórios. Assim, a transferência de arquivos para um computador remoto pode ser realizada através de um cliente SFTP, seja ele implementado em modo gráfico ou em linha de comando. O protocolo SFTP é utilizado pelo JSch para realizar a transferência de arquivos.

Nota

Não confunda SFTP e FTPS. Os dois são protocolos completamente distintos. Enquanto o SFTP implementa o SSH para manter a segurança da transferência dos arquivos, o FTPS utiliza o SSL. Além disso, o SFTP utiliza a porta padrão 22, já o FTPS a 990.

Aplicação exemplo

A aplicação a ser desenvolvida irá conter duas páginas. A primeira delas, será a página de download, onde iremos apresentar em uma listagem os arquivos a serem baixados. O conteúdo dessa listagem será proveniente do resultado de uma consulta em um banco de dados (em nosso caso, o MySQL). Por linha, será apresentado um arquivo mais o botão para efetuar o download deste. A segunda página será a página de upload, sendo formada pelos componentes de upload fornecidos pelo PrimeFaces. Nesta tela, o usuário poderá selecionar os arquivos de sua máquina e enviá-los para o diretório remoto.

Para iniciarmos o desenvolvimento da aplicação, crie um *Dynamic Web Project* no Eclipse e importe para o classpath do projeto as APIs do JSF, JSch e PrimeFaces (veja os endereços para download na seção **Links**).

Inicialmente, é preciso que a aplicação web tenha o JSF 2.2 instalado. A utilização de versões anteriores a esta necessitará de configurações adicionais que não serão abordadas em nosso exemplo. Os componentes gráficos de download e upload são disponibilizados pelo PrimeFaces. Neste exemplo, iremos utilizar a versão 4.0 deste framework, sendo esta a última versão disponibilizada até o momento da escrita deste artigo. E para trabalhar com o JSch, utilizaremos a versão 0.1.50, que também é a última disponível para download até o momento da escrita deste artigo.

Considerando que a aplicação consiste em transferir arquivos entre cliente e servidor utilizando o SSH como mecanismo de segurança, precisamos instalar o SSH no servidor para o qual iremos enviar e receber arquivos. Neste exemplo, utilizaremos o Debian como servidor, e para instalar o SSH basta digitar o seguinte comando no terminal: *apt-get install openssh-server*. Caso, após a instalação o SSH não seja inicializado, execute o comando: */etc/init.d/ssh start*.

Por fim, para a execução da aplicação, utilizaremos o Apache Tomcat, versão 7.

Estabelecendo a conexão com o servidor

Para iniciar o desenvolvimento do nosso projeto, crie a classe **ConexaoServidor**. Esta será responsável por realizar as configurações de conexão com o servidor. O código completo dessa classe é apresentado na **Listagem 1**.

Listagem 1. Código da classe ConexaoServidor.

```
01. public class ConexaoServidor {
02.
03.  private String sFtpHost = "10.1.1.5";
04.  private int sFtpPort = 22;
05.  private String sFtpUser = "usuario";
06.  private String sFtpPass = "password";
07.  private Session session = null;
08.  private Channel channel = null;
09.  private ChannelSftp channelSftp = null;
10.
11.  public void iniciarConexao() {
12.    try {
13.      JSch jsch = new JSch();
14.
15.      session = jsch.getSession(sFtpUser, sFtpHost, sFtpPort);
16.      session.setPassword(sFtpPass);
17.      java.util.Properties config = new java.util.Properties();
18.      config.put("StrictHostKeyChecking","no");
19.      session.setConfig(config);
20.      session.connect();
21.      channel = session.openChannel("sftp");
22.      channel.connect();
23.      channelSftp = (ChannelSftp) channel;
24.
25.    } catch (JSchException e) {
26.    e.printStackTrace();
27.
28.    }
29.
30.  }
31. }
```

Neste código, instanciamos a sessão jsch com os parâmetros informados e depois a configuramos indicando como deve ser realizada a autenticação. Nessa etapa, o objetivo do parâmetro **StrictHostKeyChecking** é especificar como a chave do host será analisada durante a etapa de conexão e autenticação. Por padrão, a análise da chave é definida como desativada. Deste modo, o servidor SSH verifica se a chave do host que está tentando se conectar é compatível com alguma chave contida na lista de chaves conhecidas. Caso a chave do host não corresponda com nenhuma chave existente na lista, o JSch simplesmente adiciona a chave do cliente na lista e estabelece a conexão. Quando a opção está definida como *"yes"*, o servidor SSH aceitará apenas a autenticação de hosts que possuírem chaves compatíveis com alguma chave existente na lista de chaves armazenada no servidor, proporcionando maior segurança. Caso você deseje ativar esta verificação, informe **config.put("StrictHostKeyChecking", "yes")** no método **iniciarConexao()**, conforme demonstrado em nosso código.

Implementando os métodos de transferência de arquivos

Depois de criar a classe **ConexaoServidor** e implementar o método responsável por efetuar a conexão com o servidor, o próximo passo é criar a classe **TransferirArquivos** e implementar os métodos que realizarão as tarefas de transferência de arquivos. A **Listagem 2** apresenta o código da classe **TransferirArquivos**.

O método executarDownload()

Ao executar o método **executarDownload()**, o primeiro passo a ser realizado é estabelecer a conexão com o servidor em que estão armazenados os arquivos. Esta pode ser estabelecida através da chamada do método **iniciarConexao()** – linha 17. Com a conexão estabelecida, será repassado ao comando **getChannelSftp().cd()** o diretório onde se encontra o arquivo selecionado para download (linha 25). Em seguida, é informado ao comando **getChannelSftp().get()** o nome do arquivo a ser baixado. Deste arquivo, **getChannelSftp().get()** deverá obter os dados (bytes) e armazená-los na instância de **InputStream** (linhas 28 e 29). Por fim, é criado um novo arquivo. Como pode ser observado nas linhas 32 e 33, é informado no construtor de **DefaultStreamedContent** os dados (bytes) armazenados na instância de **InputStream**, o tipo e o nome do arquivo. Este novo arquivo será o arquivo a ser transferido para o computador do usuário.

Resumindo, o processo de download é feito em cinco etapas: 1) conexão com o servidor; 2) acesso ao diretório do arquivo escolhido para download; 3) obtenção das informações do arquivo (bytes) selecionado para ser transferido; 4) criação de um novo arquivo que irá receber as informações obtidas pelo comando **getChannelSftp().get()**; e 5) após as etapas anteriores, é aberta uma tela que permitirá ao usuário escolher onde deseja salvar o arquivo.

O método executarUpload()

Na página de upload, depois de o usuário selecionar o arquivo para upload e clicar no botão enviar, o método **executarUpload()** será chamado e, consequentemente o evento **FileUploadEvent**, de acordo com nossa implementação, será disparado (linha 40). Logo após, é feita a chamada ao método **iniciarConexao()** para estabelecer a conexão com o servidor que irá receber o arquivo. Feito isso, o nome e os dados (bytes) do arquivo são armazenados nas variáveis **fileName** e **in**, respectivamente. Essas informações são obtidas do arquivo pelo **FileUploadEvent**.

Como próximo passo, devemos criar um arquivo (linha 49). É neste arquivo que será escrito o conteúdo armazenado na variável **in**. Para criá-lo, devemos passar ao construtor de **File** duas informações: o local de armazenamento dele e seu nome. Neste exemplo, o arquivo será armazenado no diretório temporário do Java ("java.io.tmpdir") e após o processo de upload ser finalizado, ele será removido deste diretório, pois não desejamos salvá-lo permanentemente. O mesmo será útil apenas para armazenar os dados a serem enviados para o ambiente remoto.

Logo após, é criada uma instância do tipo **OutputStream** (linha 51), utilizada para escrever os bytes no arquivo criado. Por fim, é informado ao comando **getChannelSftp().cd()** – linhas 61 – o diretório onde deve ser salvo o arquivo e então,

Listagem 2. Código da classe TransferirArquivos.

```
01. @ManagedBean(name = "transferirArquivos")
02. @ViewScoped
03. public class TransferirArquivos {
04.
05.  private Arquivo arquivoSelecionado = new Arquivo();
06.  private String destinoServidor;
07.  private String sFtpDiretorio;
08.  private String nomeArquivo;
09.  private StreamedContent file;
10.
11.  //Método para executar Download.
12.  public void executarDownload() {
13.    try {
14.      ConexaoServidor conexaoServidor = new ConexaoServidor();
15.
16.      // Estabelece conexão com o servidor.
17.      conexaoServidor.iniciarConexao();
18.
19.      // Nome do arquivo selecionado para download, obtido pelo
20.      //componente setPropertyActionListener.
21.
22.      nomeArquivo = arquivoSelecionado.getNomeArquivo().toString();
23.
24.      // Acessa o diretório informado.
25.      conexaoServidor.getChannelSftp().cd(sFtpDiretorio);
26.
27.      // Armazena os bytes do arquivo em na instância de InputStream.
28.      InputStream stream = conexaoServidor.getChannelSftp().get(
29.          nomeArquivo);
30.
31.      // Cria um novo arquivo do tipo StreamedContent
32.      file = new DefaultStreamedContent(stream, "application/txt",
33.          nomeArquivo);
34.
35.    } catch (Exception ex) {
36.      ex.printStackTrace();
37.    }
38.  }
39.
40.  // Método para executar Upload.
41.  public void executarUpload(FileUploadEvent event) {
42.    try {
43.      ConexaoServidor conexaoServidor = new ConexaoServidor();
44.      conexaoServidor.iniciarConexao();
45.
46.      String fileName = event.getFile().getFileName();
47.      InputStream in = event.getFile().getInputstream();
48.      String diretorioTemp = System.getProperty("java.io.tmpdir");
49.
50.
51.      File file = new File(diretorioTemp, fileName);
52.
53.      OutputStream out = new FileOutputStream(file);
54.
55.      int read = 0;
56.      byte[] bytes = new byte[1024];
57.      while ((read = in.read(bytes)) != -1) {
58.        out.write(bytes, 0, read);
59.      }
60.
61.      in.close();
62.      file.deleteOnExit();
63.      conexaoServidor.getChannelSftp().cd(destinoServidor);
64.      conexaoServidor.getChannelSftp().put(new FileInputStream(file),
65.          fileName);
66.    } catch (Exception e) {
67.      e.printStackTrace();
68.    }
69.
70.  }
71.
72. }
```

é realizada a gravação no diretório remoto através do comando **getChannelSftp().put()** – linhas 62 e 63. Resumindo, o processo de upload é realizado em quatro etapas:

1) conexão com o servidor;

2) criação do arquivo que irá receber os dados do arquivo selecionado para upload;

3) criação de uma instância de **OutputStream** para escrever os dados no arquivo criado; e

4) envio do arquivo para o diretório remoto.

Criando a página de Upload

Concluídas as etapas anteriores, vamos criar a página de upload. Esta permitirá ao usuário selecionar os arquivos para enviá-los ao servidor. Para essa tarefa, faremos uso do recurso de **fileUpload**, pertencente à API do PrimeFaces, que pode ser utilizado para enviar arquivos para diretórios locais sem o auxílio de outros frameworks. Porém, para realizar o envio para diretórios remotos, é necessário o uso de algum framework que possa desempenhar essa tarefa. É por isso que também estamos trabalhando com o JSch, conforme mostrado nas **Listagens 1** e **2**. O código da página de upload com o uso do recurso **fileUpload** (linhas 15 a 21) é apresentado na **Listagem 3**.

Listagem 3. Código da página upload.xhtml.

```
01. <html xmlns="http://www.w3.org/1999/xhtml"
02.   xmlns:ui="http://java.sun.com/jsf/facelets"
03.   xmlns:f="http://java.sun.com/jsf/core"
04.   xmlns:h="http://java.sun.com/jsf/html"
05.   xmlns:p="http://primefaces.org/ui">
06.
07. <h:head>
08. </h:head>
09. <h:body>
10.  <ui:composition template="/templates/main.xhtml">
11.    <ui:define name="content">
12.      <p:messages id="messages" showDetail="true" autoUpdate="true"
13.        closable="true" />
14.      <h:form enctype="multipart/form-data">
15.        <p:fileUpload
16.          fileUploadListener="#{transferirArquivos.executarUpload}"
17.            multiple="true"
18.            description="Selecionar Arquivos"
19.                    label="Selecionar"
20.              uploadLabel="Enviar"
21.            cancelLabel="Cancelar" />
22.      </h:form>
23.    </ui:define>
24.  </ui:composition>
25. </h:body>
26. </html>
```

Como neste exemplo faremos o upload de vários arquivos simultaneamente, é preciso utilizar o atributo **multiple** definido como **true** (linha 17). Além dos atributos utilizados na **Listagem 3**, o componente **fileUpload** possui outros. Dentre eles, podemos destacar os atributos **fileLimit, sizeLimit**, **auto** e **allowTypes**. O atributo **fileLimit** permite restringir o número máximo de arquivos a serem enviados. O atributo **sizeLimit** restringe o tamanho máximo de cada arquivo. O upload automático após a seleção dos arquivos pode ser ativado através do atributo **auto**. Por fim, o atributo **allowTypes** permite restringir os tipos de arquivo (jpeg, gif, txt, etc.) que podem ser selecionados para o upload.

As **Figuras 2** e **3** exibem a interface da página de upload após a seleção dos arquivos a serem enviados e a listagem dos arquivos no servidor após o envio, respectivamente.

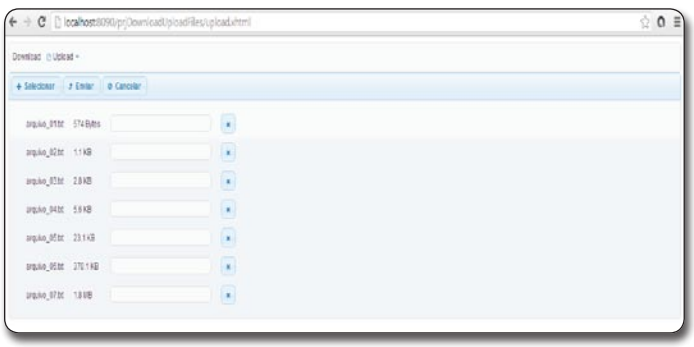


Figura 2. Interface da página de upload com os arquivos selecionados

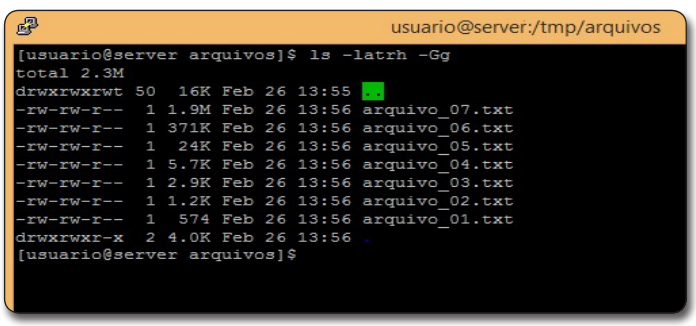


Figura 3. Listagem dos arquivos enviados, armazenados no servidor

Criando a página de download

Nesta etapa iremos criar a página de download. Esta permitirá ao usuário escolher o arquivo do servidor a ser transferido para o seu sistema de arquivos. Para esta tarefa, utilizaremos o recurso **fileDownload**, também fornecido pela API do PrimeFaces. O código XHTML da página de download é apresentado na **Listagem 4**.

Primeiramente, é necessário que o recurso **fileDownload** esteja vinculado a um componente que possa executar uma ação para iniciar o processo de download. No exemplo deste artigo, o **fileDownload** foi vinculado ao componente **commandButton**. Assim, após o botão ser acionado, irá iniciar o processo de download do arquivo selecionado na lista.

Ao acionar o **commandButton**, o primeiro passo é obter o nome do arquivo que foi escolhido para ser transferido.

Listagem 4. Código da página download.xhtml.

```
01. <html xmlns="http://www.w3.org/1999/xhtml"
02.   xmlns:ui="http://java.sun.com/jsf/facelets"
03.   xmlns:f="http://java.sun.com/jsf/core"
04.   xmlns:h="http://java.sun.com/jsf/html"
05   xmlns:p="http://primefaces.org/ui">
06. <h:head>
07. </h:head>
08. <h:body>
09.   <h:outputStylesheet name="css/styles.css" />
10.   <ui:composition template="/templates/main.xhtml">
11.     <ui:define name="content">
12.           <h:form id="formPrincipal" enctype="multipart/form-data">
13.           <p:messages id="messages" showDetail="true" autoUpdate="true"
14.             closable="true" />
15.           <p:dataTable id="tabelaArquivo" var="arq"
16.           value="#{transferirArquivos.listArquivo}" rowKey="#{arq.id_arquivo}"
17.             selectionMode="single" paginatorAlwaysVisible="false">
18.             <p:column headerText="Nome Arquivo" style="text-align: center">
19.               <h:outputText value="#{arq.nomeArquivo}" />
20.             </p:column>
21.           <p:column style="width:9%;text-align: center" headerText="Download">
22.               <p:commandButton id="downloadLink" ajax="false"
23.                 icon="ui-icon-circle-arrow-s" title="Download"
24.               onclick="PrimeFaces.monitorDownload(start, stop)">
25.
26.             <f:setPropertyActionListener value="#{arq}"
27.                     target="#{transferirArquivos.arquivoSelecionado}" />
28.               <f:actionListener
29.                binding="#{transferirArquivos.executarDownload()}" />
30.               <p:fileDownload value="#{transferirArquivos.file}" />
31.               </p:commandButton>
32.             </p:column>
33.           </p:dataTable>
34.       </h:form>
35.     </ui:define>
34.   </ui:composition>
35. </h:body>
36. </html>
```

Este procedimento é necessário porque o comando **channelSftp.get()** necessita receber o nome do arquivo para buscá-lo no diretório remoto. Tal tarefa é executada pelo componente do PrimeFaces **setPropertyActionListener**. Este identifica em qual linha o arquivo selecionado está, e desta forma consegue obter o nome do mesmo. No entanto, para realizar este procedimento, o **setPropertyActionListener** deve saber qual componente exibirá a listagem dos arquivos para download. Neste exemplo empregamos o componente **dataTable**.

O **dataTable** possui um atributo denominado **var**, onde é informada a variável utilizada para identificá-lo. Para estabelecer a "ligação" necessária e que foi comentada no parágrafo anterior, esta variável deve ser informada no atributo **value** do componente **setPropertyActionListener**.

Após obter o nome do arquivo, é necessário armazená-lo em um local, para que o comando **channelSftp.get()** possa recebê-lo e consequentemente buscá-lo no diretório devido. Para isso, informamos ao atributo **target**, pertencente ao componente

setPropertyActionListener, o objeto **arquivoSelecionado**, instanciado na classe **TransferirArquivos**. Desta forma, ao clicar no botão, as informações da linha serão repassadas ao objeto **arquivo-Selecionado**. A segunda tarefa após acionar o **commandButton** é efetuar a chamada ao método **executarDownload()**, que será realizada através do componente **actionListener** (linhas 28 e 29). O terceiro e último passo a ser realizado após o acionamento do botão é executar o componente **fileDownload** (linha 30), para que ele carregue o objeto **file**, referenciado no seu atributo **value** e, em seguida, abra a tela para o usuário escolher o diretório que deseja salvar o arquivo. A **Figura 4** apresenta a página de download.

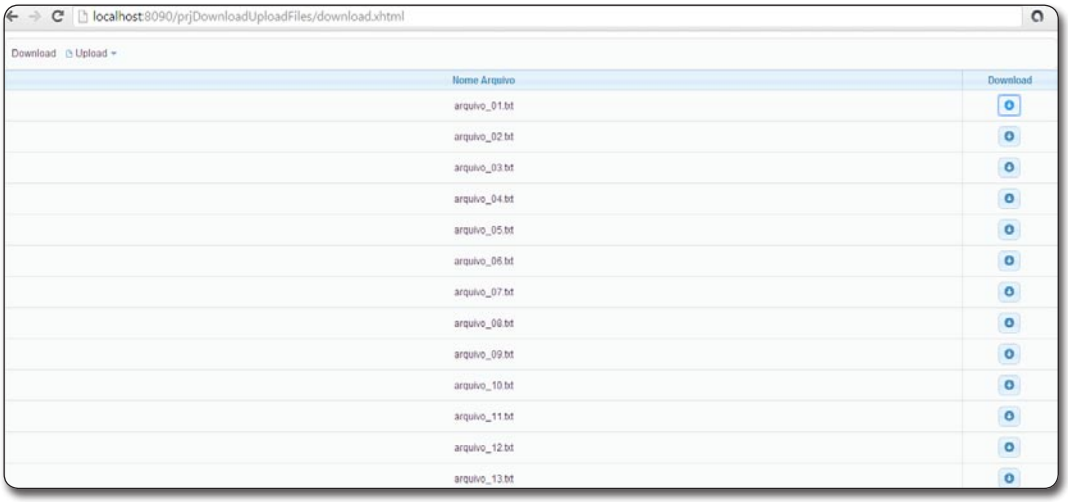


Figura 4. Interface da página de download com os arquivos listados

Nota

No exemplo deste artigo, ao iniciar o download, é solicitado ao usuário que escolha o local onde deseja salvar o arquivo. Caso, ao clicar no botão de download, a transferência seja realizada sem abrir essa tela de solicitação de diretório, verifique as configurações do seu navegador e habilite a opção de escolha de diretório ao efetuar downloads.

Após a leitura deste artigo o leitor terá conhecimento suficiente para utilizar o framework JSch em aplicações web desenvolvidas com o PrimeFaces. Contudo, vale ressaltar que não foram abordadas todas as funcionalidades do framework JSch, pois o objetivo do projeto exemplo foi demonstrar apenas a transferência básica de arquivos. Além deste recurso, ainda existem funções para compactação de arquivos, transferência de arquivos através do protocolo SCP, entre outras.

Autor

Luis Gustavo Souza
os.luisgustavo@gmail.com
Cursa Sistemas de Informação pela Universidade de Franca, desenvolvedor Java, ADF/SOA e PL\SQL no Grupo Amazonas, trabalha há quatro anos na área de informática, sendo dois deles como desenvolvedor. Atualmente tem como objetivo obter a certificação OCA Java.

Links:

Download, exemplos e documentação do framework JSch.
http://www.jcraft.com/jsch/

Download do framework PrimeFaces.
http://www.primefaces.org/downloads

Download do Apache Tomcat 7.
http://tomcat.apache.org/download-70.cgi

Download do JSF 2.2.
https://javaserverfaces.java.net/download.html

PrimeFaces ShowCase.
http://www.primefaces.org/showcase/ui/home.jsf

Somos tão apaixonados por tecnologia que o nome da empresa

diz tudo.

Porta 80 é o melhor que a Internet pode oferecer para sua empresa.

Já completamos 8 anos e estamos a caminho dos 80, junto com nossos clientes.

Adoramos tecnologia.
Somos uma equipe composta de gente que entende e gosta do que faz,

assim como você.

Estrutura

100% NACIONAL. Servidores de primeira linha, links de alta capacidade.

Suporte diferenciado

Treinamos nossa equipe para fazer mais e melhor. Muito além do esperado.

Serviços

Oferecemos a tecnologia mais moderna, serviços diferenciados e antenados com as suas necessidades.

1-to-1

Conhecemos nossos clientes. Atendemos cada necessidade de forma única. Conheça!

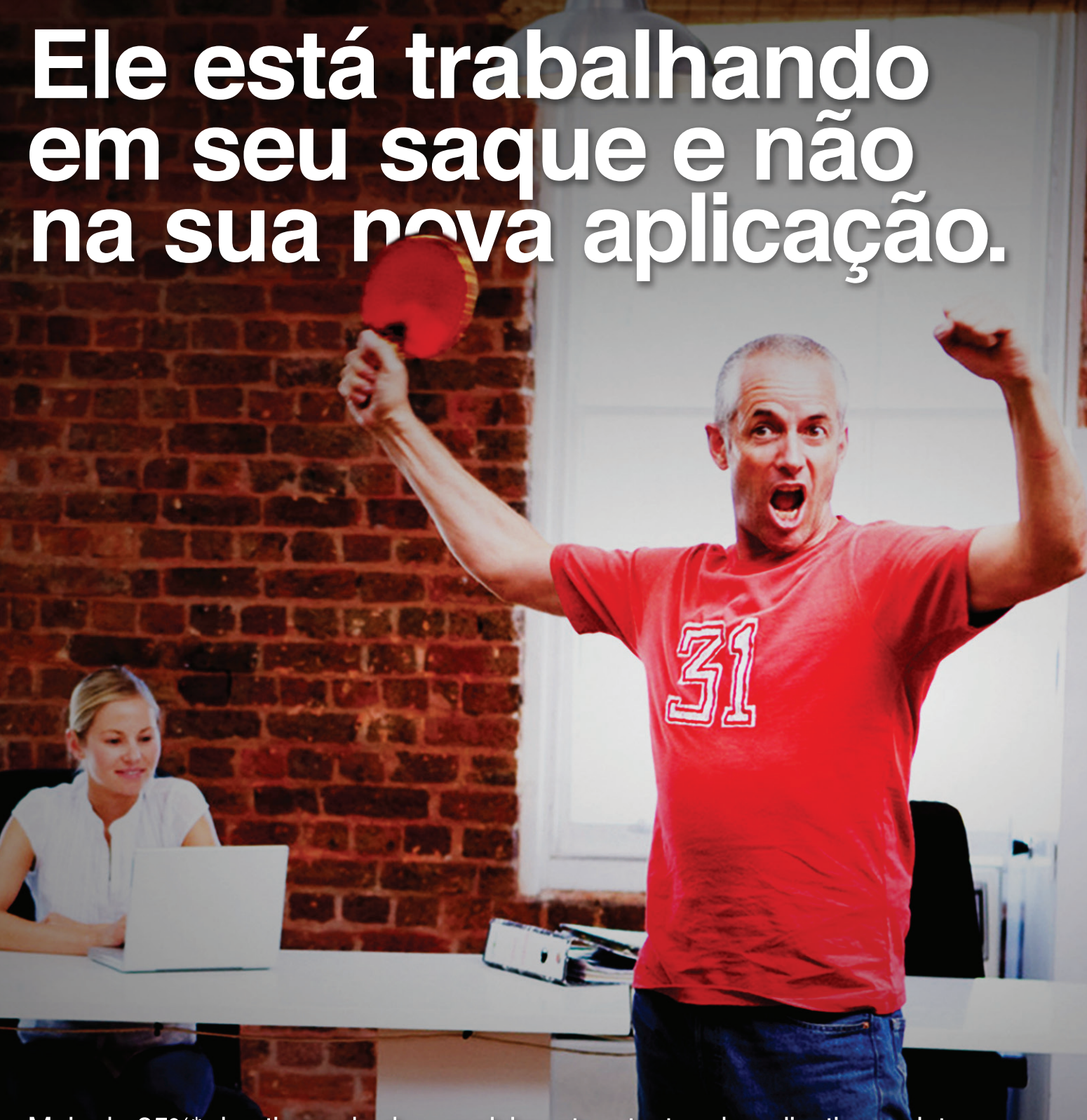
Porta 80
WEB HOSTING

Hospedagem | Cloud Computing | Dedicados | VoIP | Ecommerce | Aplicações | Streaming | Email corporativo

porta80.com.br | comercial@porta80.com.br | twitter.com/porta80

SP 4063-8616 | RJ 4063-5092 | MG 4063-8120 | DF 4063-7486

Ele está trabalhando em seu saque e não na sua nova aplicação.



Mais de 95%* dos times de desenvolvimento e testes de aplicativos relatam tempos de espera e atrasos para acessar os sistemas que necessitam para realizar suas atividades. A Virtualização de Serviços elimina estas dependências gerando ambientes simulados similares à realidade, permitindo o desenvolvimento em paralelo. Isto significa que suas aplicações serão lançadas mais rapidamente, com maior qualidade e menor custo. Game over!

Conheça mais em ca.com/br/GoDevOps

*De acordo com o estudo 2012 North America/Europe Service Virtualization Study, Coleman-Parkes.

ca technologies