



Edição 38



ISSN 2179625-4



TESTES COM JUNIT

Programando com
qualidade e agilidade

Trabalhando com arquivos em Java
Aprenda a acessar arquivos e outros
recursos de forma simples e eficiente



Primeiros passos com Android
Construa aplicativos para
com a IDE Android Studio

Introdução ao Garbage Collection
Entendendo o gerenciamento
automático de memória da JVM

DESENVOLVA PARA CORPORAÇÕES OU PARA O DIA A DIA COM JAVA

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.**

➤ 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!

CERTIFICAÇÕES:

➤ Oracle Certified Professional, Java SE 6 Programmer

➤ Oracle Certified Professional, Java EE 5 Web Component Developer



Sumário

Artigo no estilo Solução Completa

06 – Java I.O: Trabalhando com arquivos em Java

[John Soldera]

Artigo sobre Boas Práticas

13 – Testes automatizados com JUnit

[Andrei de Oliveira Tognolo]

Artigo no estilo Solução Completa

22 – Android Studio: Primeiros passos com Android

[Alessandro Jatobá]

Artigo sobre Boas Práticas

32 – Introdução ao Java Garbage Collection

[Carlos Araújo]

CURSOS ONLINE

A Revista Java Magazine oferece aos seus assinantes uma série de Cursos Online de alto padrão de qualidade.



CONHEÇA ALGUNS DOS CURSOS:

- Curso de noSQL (Redis) com Java
- Curso Básico de JDBC
- Java Básico: Aplicações Desktop
- JSF com Primefaces
- Conhecendo o Apache Struts

Para mais informações :

www.devmedia.com.br/curso/javamagazine

(21) 3382-5038



Edição 38 • 2014 • ISSN 2173625-4



Assine agora e tenha acesso a todo o conteúdo da DevMedia:
www.devmedia.com.br/mvp

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 à 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

DÊ UM SALTO EM CONHECIMENTO!



Acesse o maior
portal para
desenvolvedores
da América
Latina!





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 código (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

Java I.O: Trabalhando com arquivos em Java

Aprenda a acessar arquivos e outros recursos de forma simples e eficiente

Em muitas aplicações, além do desenvolvimento de regras de negócio, é necessário gerenciar arquivos e/ou diretórios, criando, excluindo ou copiando os mesmos para outras localizações. Para essas finalidades, o sistema operacional oferece, através do shell, um conjunto de comandos capaz de criar, excluir e gerenciar arquivos, além de ser possível executar tais comandos em aplicações Java; porém, essa não é uma solução ideal para ser utilizada, pois leva a uma vinculação muito forte da aplicação ao sistema operacional, desperdiçando assim a portabilidade da plataforma Java e, em muitos casos, degradando a performance do sistema.

Vale ressaltar que é muito comum ocorrerem situações em que projetos de software precisem ser migrados para plataformas diferentes, implicando em adaptações que muitas vezes são difíceis, pois cada plataforma tem a sua forma de manipular arquivos, comandos específicos e detalhes sobre o funcionamento do sistema de arquivos. Para superar essa barreira e ser possível criar aplicações multiplataforma, a linguagem Java oferece comandos genéricos para realizar a manipulação de arquivos sem criar um vínculo forte com a plataforma e, ao mesmo tempo, permitindo o controle e o tratamento de erros.

Para tais finalidades, o pacote **java.io** oferece a classe **File**. Esta classe permite a criação, exclusão e outras operações com arquivos. Adicionalmente, o mesmo pacote também disponibiliza *streams* para arquivos, que são canais de comunicação direcionados para gravar e ler dados de arquivos.

A fim de mostrar o funcionamento de tais recursos, é proposta uma aplicação exemplo que realiza as principais operações de manipulação de arquivos, incluindo a cópia de arquivos por *streams* e o controle de erros em cada operação. Por fim, é proposta também uma interface gráfica para a aplicação exemplo, visando melhorar a sua usabilidade.

A classe **java.io.File**

A classe **java.io.File** está presente desde o JDK 1.0 e oferece uma abstração de arquivo como sendo um

Fique por dentro

Este artigo apresenta comandos para criar, excluir e copiar arquivos utilizando classes básicas da linguagem Java pertencentes ao pacote **java.io**. Dentre elas, a principal é a classe **File**. A fim de ilustrar a utilização de tais funcionalidades, é proposta uma aplicação exemplo que realiza várias operações, mostrando que em Java não é necessário atrelar a aplicação a uma plataforma de sistema operacional específica, obtendo boa performance e ainda possibilitando o controle de erros.

recurso, escondendo detalhes de funcionamento do sistema operacional. Uma instância de **File** tem a função de apontar para um arquivo ou diretório no sistema de arquivos e disponibiliza vários comandos para manipular o recurso referenciado. Seus construtores são listados a seguir:

- **File(File parent, String child)**: Cria um novo objeto **File** com o caminho indicado por **parent** concatenado ao valor de **child**. Não é necessário que o arquivo ou diretório apontado exista;
- **File(String pathname)**: Cria uma nova instância de **File** usando uma **String** com o caminho até o recurso;
- **File(String parent, String child)**: Cria um novo objeto **File** com o caminho indicado por **parent** concatenado ao valor de **child**. Não é necessário que o arquivo ou diretório apontado exista;
- **File(URI uri)**: Recebe como parâmetro o caminho para um recurso (URI), que pode ser um arquivo, diretório ou outro recurso local ou remoto.

É importante observar nessa listagem que para criar URIs para arquivos no Windows, aciona-se o sufixo “**file:///**”, e para arquivos no Linux, adiciona-se o sufixo “**file://**” (sendo a terceira barra no exemplo para Linux a raiz do sistema de arquivos). Documentos em sites da internet também podem ser acessados através de URIs, inclusive documentos disponibilizados por outros protocolos de comunicação, como o FTP (*File Transfer Protocol*).

Os métodos mais relevantes da classe **File** são:

- **createNewFile()**: Cria o arquivo indicado por essa instância de **File**. Retorna **true** se a operação tiver sucesso. Se retornar **false**, o arquivo não foi criado;

Nota

Uma URI é um caminho para um recurso local ou remoto que tem um nome. A sua forma geral pode ser definida pela seguinte expressão:

[scheme]:content[#fragment].

Onde: scheme é o tipo de recurso indicado pela URI, que pode ser um nome de protocolo ou indicador de arquivo em um determinado sistema operacional; conteúdo, dado por content, é um caminho e/ou o nome do objeto; e fragment é opcional e informa uma posição específica no recurso, como por exemplo, o nome de um marcador (landmark) HTML, iniciado sempre por #, como na **Listagem 1**, que, adicionalmente, mostra outros tipos de URIs.

Listagem 1. Exemplos de URI.

```
// Arquivo no Windows:  
file:///c:/calendar.  
// Arquivo no Linux:  
file:///~/calendar.  
// Endereço de e-mail:  
mailto:java-net@java.sun.com.  
// Esquema de Grupos de Notícias e Artigos:  
news:comp.lang.java.  
//Marcador em página HTML:  
http://docs.oracle.com/javase/7/docs/api/java/io/File.html#delete().
```

- **delete()**: Exclui o recurso indicado por esse objeto **File**. Retorna um indicativo de sucesso;
- **deleteOnExit()**: Faz com que o recurso indicado pelo objeto **File** seja excluído quando a JVM for finalizada;
- **exists()**: Verifica se o recurso existe, seja local ou remoto;
- **getAbsolutePath()**: Retorna o caminho absoluto do recurso apontado por esse objeto;
- **getFreeSpace()**: Se o recurso está em uma unidade de armazenamento, retorna a quantidade de bytes livres na partição em que ele se encontra;
- **getName()**: Retorna o nome desse recurso;
- **getPath()**: Retorna o caminho para esse recurso;
- **getTotalSpace()**: Se o recurso está em uma unidade de armazenamento, retorna o tamanho total da partição em que ele se encontra;
- **isDirectory()**: Informa se o recurso é um diretório;
- **isFile()**: Informa se o recurso é um arquivo;
- **isHidden()**: Informa se o recurso é um arquivo oculto pelo sistema operacional;
- **lastModified()**: Informa a data da última modificação nesse recurso;
- **length()**: Retorna o tamanho do arquivo;
- **list()**: Se o recurso apontado for um diretório, retorna um vetor com o nome de todos os arquivos e/ou diretórios internos;
- **list(FilenameFilter filter)**: Se o recurso apontado for um diretório, retorna um vetor com o nome de todos os arquivos e/ou diretórios internos que se enquadram no filtro informado;
- **listFiles()**: Se o recurso apontado for um diretório, retorna um vetor **File[]** com todos os arquivos e/ou diretórios internos;
- **listRoots()**: Lista todas as raízes de sistemas de arquivos disponíveis no sistema operacional;

- **mkdir()**: Se o recurso for um nome de diretório, cria o mesmo. Retorna o indicativo de sucesso;
- **renameTo(File dest)**: Troca o nome desse arquivo para o nome informado. Retorna o indicativo de sucesso;
- **setExecutable(boolean executable, boolean ownerOnly)**: Atualiza as permissões do arquivo no sistema operacional para que o usuário corrente (ou todos) tenha permissão de executar o arquivo apontado. Retorna o indicativo de sucesso;
- **setReadable(boolean readable, boolean ownerOnly)**: Atualiza as permissões do arquivo no sistema operacional para que o usuário corrente (ou todos) tenha permissão de ler o arquivo apontado. Retorna o indicativo de sucesso;
- **setWritable(boolean writable, boolean ownerOnly)**: Atualiza as permissões do arquivo no sistema operacional para que o usuário corrente (ou todos) tenha permissão de alterar o arquivo apontado. Retorna o indicativo de sucesso;
- **setReadOnly()**: Define o arquivo apontado como somente leitura no sistema operacional. Retorna **true** ou **false** indicando o sucesso da operação;
- **toURI()**: Retorna uma representação na forma de URI do recurso apontado por esse objeto **File**;
- **toURL()**: Retorna uma representação na forma de URL do recurso apontado. Esse método está definido como *deprecated*, ou seja, é desaprovado para uso por que ele pode apresentar problemas em algumas situações. Ao invés dele, recomenda-se utilizar o método **toURI()** e em seguida o método **toURL()** da classe **URI**.

Pode-se notar pelos métodos da classe **File** que ela permite realizar operações básicas de manipulação individual de arquivos, como trocar permissões, criar arquivos e diretórios, excluir, listar, etc. Porém, ela não oferece recursos comuns da interface do sistema operacional, como copiar, recortar e colar arquivos e diretórios. Para realizar tais tarefas, uma das melhores alternativas disponibilizadas no JDK é fazer uso de outras classes do pacote **java.io**, como **FileInputStream** e **FileOutputStream**, que representam *streams* que possibilitam copiar blocos de dados de um arquivo para outro, fornecendo dessa forma a base para diversas funcionalidades que são encontradas no shell.

Com base nos conhecimentos que foram apresentados, a seguir será implementada uma aplicação exemplo para mostrar o funcionamento dos métodos básicos da classe **File**, incluindo um exemplo de como copiar arquivos usando *streams*.

Manipulando arquivos na prática

Para mostrar o funcionamento dos principais comandos da classe **File**, uma aplicação exemplo será desenvolvida. Esta aplicação receberá comandos pela linha de comando do shell e realizará operações como criar e remover arquivos e diretórios, listar diretório e copiar arquivos (o que inclui a criação de arquivos e a gravação de dados nestes).

A nossa aplicação exemplo foi codificada na classe **FileApp**, apresentada na **Listagem 2**. Esta contém o método **main()** e diversos métodos estáticos que poderão ser chamados para executar

susas respectivas funcionalidades de manipulação de arquivos. O código de cada um desses métodos será apresentado em listagens separadas (**Listagens 3 a 7**), que serão analisadas de acordo com a operação em estudo da aplicação.

Listagem 2. Aplicação exemplo para manipulação de arquivos.

```
01. package fileAppPkg;
02.
03. import java.io.File;
04. import java.io.FileInputStream;
05. import java.io.FileNotFoundException;
06. import java.io.FileOutputStream;
07. import java.io.IOException;
08. import java.io.InputStream;
09. import java.io.OutputStream;
10.
11. public class FileApp {
12.
13.     public static void main(String[] args) {
14.         if (args.length == 0)
15.             System.out.println("Primeiro parâmetro deve ser operação.");
16.         else if (args.length == 1)
17.             System.out.println("O segundo parâmetro é obrigatório.");
18.         else {
19.             String tipoOperacao = args[0];
20.             String segundoParametro = args[1];
21.             String terceiroParametro = null;
22.             if (args.length == 3)
23.                 terceiroParametro = args[2];
24.
25.             if (tipoOperacao.equalsIgnoreCase("criarDiretorio"))
26.                 criarDiretorio(segundoParametro);
27.             else if (tipoOperacao.equalsIgnoreCase("removerDiretorio"))
28.                 removerArquivo(segundoParametro);
29.             else if (tipoOperacao.equalsIgnoreCase("removerArquivo"))
30.                 removerDiretorio(segundoParametro);
31.             else if (tipoOperacao.equalsIgnoreCase("listarDiretorio"))
32.                 listarDiretorio(segundoParametro);
33.             else if (tipoOperacao.equalsIgnoreCase("copiarArquivo"))
34.                 copiarArquivo(segundoParametro, terceiroParametro);
35.         }
36.     }
37. ...
38. }
```

Listagem 3. Método para criação de diretórios.

```
01. public static void criarDiretorio(String caminhoCriar) {
02.     System.out.println("Criando novo diretório ...");
03.     File dirBase = new File(caminhoCriar);
04.     boolean sucesso = dirBase.mkdir();
05.     if (sucesso)
06.         System.out.println("Diretório criado com sucesso.");
07.     else
08.         System.out.println("Erro ao criar diretório.");
09. }
```

Para executar os métodos apresentados, **FileApp** recebe parâmetros pela linha de comando, sendo o primeiro deles uma **String** que especifica o tipo da operação a ser realizada. As operações suportadas são: **criarDiretorio**, **removerDiretorio**, **removerArquivo**, **listarDiretorio** e **copiarArquivo**. O segundo parâmetro da linha de comando também é uma **String**, e representa o parâmetro da operação. No caso da operação de cópia de arquivos, devemos informar três parâmetros: a operação, o nome do arquivo de origem e o diretório de destino. Caso sejam informados parâmetros excedentes a qualquer operação, eles serão ignorados.

Como pode ser observado na linha 1, a classe **FileApp** foi criada no pacote **fileAppPkg**. Nas linhas 3 a 9 são declarados os **imports** necessários para o nosso exemplo, e no restante dessa listagem, temos o código do método **main()**. A implementação dos demais métodos é apresentada nas **Listagens 3 a 7**.

Voltando ao nosso código, na linha 14 é verificado o caso de nenhum parâmetro ter sido informado na linha de comando. Se essa situação for verdadeira, é mostrada uma mensagem de erro, definida na linha 15, e a aplicação é encerrada. Caso contrário, o primeiro parâmetro, que é considerado o tipo da operação, é identificado. Como para qualquer operação é necessário ao menos mais um parâmetro, verificamos se a aplicação recebeu apenas um parâmetro. Caso positivo, uma mensagem de erro é apresentada (linhas 16 e 17). No entanto, se dois ou mais parâmetros foram especificados, o processamento segue na linha 19 até a linha 36. Nas linhas 19 a 20, o primeiro e o segundo parâmetros são acessados e são criadas variáveis próprias para cada um deles, e nas linhas 21 a 23, é acessado o terceiro parâmetro. Feito isso, na linha 25 é testado se o comando informado foi **criarDiretorio**. Caso positivo, é chamado o método **criarDiretorio()** passando como parâmetro o segundo parâmetro do método **main()** – neste caso, o nome do diretório a ser criado. De forma semelhante, até o final desse método são testadas e executadas as operações para remover diretório ou arquivo, listar diretório ou copiar arquivos. Cada uma dessas operações será detalhada nas listagens a seguir. Na **Listagem 3**, por exemplo, temos o método **criarDiretorio()**.

Criando diretórios

Este método recebe como parâmetro o diretório a ser criado, ou seja, é necessário informar o caminho completo do diretório a ser criado pela linha de comando. Na linha 3, é instanciado um objeto **File** para o caminho indicado e, na linha 4, o novo diretório é criado.

Observe que o retorno do método **mkdir()** é atribuído à variável **sucesso**. Na linha 5 é testado se essa variável é verdadeira, ou seja, se não houve nenhum erro durante a criação do diretório. Se for esse o caso, é escrito para o usuário uma mensagem de sucesso (linha 6). Entretanto, se ocorreu algum erro, como permissões insuficientes ou caminho incorreto, a variável **sucesso** conterá **false** e será exibida uma mensagem de erro (linha 8).

Removendo arquivos

Continuando nossa análise, na **Listagem 4** é apresentado o código do método **removerArquivo()**.

Esse método recebe como parâmetro o nome completo do arquivo a ser removido, ou seja, deve ser informado o caminho completo com o nome do arquivo pela linha de comando. Logo no início, na linha 3, é criado um objeto do tipo **File** para o nome de arquivo informado. Através desse objeto é possível verificar se o recurso existe e se realmente trata-se de um arquivo.

Na linha 4 é testado se o arquivo não existe, e se isso for verdade, é mostrada uma mensagem de erro (linha 5).

Outra situação que impossibilita a remoção do arquivo é quando o objeto **File** não aponta para um arquivo, e sim para outro tipo de recurso. Nesse caso, o processo de remoção também é cancelado. Assim, nas linhas 8 a 10 é verificado se o objeto **File** não é um arquivo. Se esse teste for afirmativo, é cancelada a remoção.

Finalmente, na linha 12 ocorre a exclusão do arquivo, o que é feito pelo método **delete()**. Porém, existem situações que podem fazer a remoção do arquivo falhar, como erros de permissão. Sendo assim, criamos a variável **sucesso** para armazenar o retorno do processo de exclusão. Logo após, um teste é feito na linha 13. Se o arquivo foi excluído com sucesso, é mostrada uma mensagem de sucesso (linha 14); caso contrário, é mostrada uma mensagem de erro (linha 16).

Removendo diretórios

Para analisar o método relacionado à exclusão de diretórios, **removerDiretorio()**, veja a [Listagem 5](#).

Listagem 4. Método para exclusão de arquivos.

```

01. public static void removerArquivo(String nomeArquivo) {
02. System.out.println("Removendo arquivo ...");
03. File arquivo = new File(nomeArquivo);
04. if (!arquivo.exists()) {
05. System.out.println("Não existe arquivo " + nomeArquivo + ".");
06. return;
07. }
08. else if (!arquivo.isFile()) {
09. System.out.println(nomeArquivo + " não é um arquivo.");
10. return;
11. }
12. boolean sucesso = arquivo.delete();
13. if (sucesso)
14. System.out.println("Arquivo removido com sucesso.");
15. else
16. System.out.println("Erro ao remover arquivo.");
17. }

```

Listagem 5. Método para exclusão de diretórios.

```

01. public static void removerDiretorio(String nomeDiretorio) {
02. System.out.println("Removendo diretório ...");
03. File diretorio = new File(nomeDiretorio);
04. if (!diretorio.exists()) {
05. System.out.println("Não existe diretório " + nomeDiretorio + ".");
06. return;
07. }
08. else if (!diretorio.isDirectory()) {
09. System.out.println(nomeDiretorio + " não é um diretório.");
10. return;
11. }
12. boolean sucesso = diretorio.delete();
13. if (sucesso)
14. System.out.println("Diretório removido com sucesso.");
15. else
16. System.out.println("Erro ao remover diretório.");
17. }

```

O método **removerDiretorio()** funciona de forma análoga ao método **removerArquivo()**. Porém, agora se trata da remoção de um diretório, e não de um arquivo. Então, inicialmente é criado um objeto **File** para o diretório na linha 3 e é verificado se ele não existe (linha 4).

Se o teste for afirmativo, é mostrada uma mensagem de erro (linha 5) e a exclusão é cancelada. Se o caminho informado não for um diretório (teste da linha 8), é mostrada uma mensagem de erro (linha 9) e a exclusão também é cancelada.

Se nenhuma dessas situações ocorreu, é chamado o método **delete()** na linha 13, de forma que seu retorno é atribuído à variável **sucesso**. Se a exclusão foi realizada com sucesso, é mostrada uma mensagem de sucesso (linha 15); caso contrário, é mostrada uma mensagem de erro (linha 17).

Listando diretórios

Outra operação demonstrada na aplicação exemplo é a listagem de diretórios, que é semelhante ao comando **dir** do shell do sistema operacional. Veja a [Listagem 6](#).

Listagem 6. Método para listagem de diretórios.

```

01. public static void listarDiretorio(String nomeDiretorio) {
02. System.out.println("Listando arquivos ...");
03. System.out.println("A/D Caminho Completo:");
04. File fileObj = new File(nomeDiretorio);
05. if (!fileObj.exists()) {
06. System.out.println("Não existe diretório " + nomeDiretorio + ".");
07. return;
08. }
09. File [] arquivos = fileObj.listFiles();
10. if (arquivos == null) {
11. System.out.println("Não foi possível listar o diretório " + nomeDiretorio + ".");
12. return;
13. }
14. for (File arquivo : arquivos) {
15. String flagDir;
16. if (arquivo.isDirectory())
17. flagDir = "D";
18. else
19. flagDir = "A";
20. System.out.println("[" + flagDir + "] " + arquivo);
21. }
22. System.out.println("Listagem completa.");
23. }

```

Nessa listagem é implementado o método **listarDiretorio()**. Este recebe um diretório como parâmetro e lista todos os arquivos e subdiretórios internos, mostrando a listagem no console. Arquivos e diretórios são objetos que têm diversos atributos gerenciados pelo sistema operacional, como caminho completo, permissões de escrita, leitura, exclusão e execução, se é oculto, etc., porém estamos interessados somente em uma listagem simples para demonstrar o funcionamento do método **listFiles()** da classe **File**, de forma que serão mostrados apenas o caminho completo dos objetos encontrados e um indicador (A/D) informando se o objeto é um arquivo ou diretório. Na linha 3 encontra-se o comando de escrita do cabeçalho da listagem, que tem a função de mostrar a organização dos dados.

Inicialmente, é criado um objeto **File** para o caminho informado (linha 4). Nas linhas 5 a 7 é verificado se o caminho existe. Caso não exista, o processo de listagem é encerrado. Na linha 9 é chamado o método **listFiles()**. Este retorna um vetor de **File** com cada arquivo

ou diretório encontrado na listagem do diretório informado como parâmetro. Caso tenha ocorrido algum erro durante a execução do comando `listFiles()`, o vetor retornado terá conteúdo nulo e o processo de listagem é cancelado (linhas 10 a 12).

Com a listagem em mãos, é necessário percorrê-la e escrever cada elemento encontrado na tela. Para isso, na linha 14 é usado o comando `for`. Na linha 16 é testado se o elemento corrente é um diretório. Se sim, a variável `flagDir` é definida como `D`; caso contrário, como `A`. Logo em seguida, na linha 20 é escrito o nome completo do recurso atual, incluindo a `flagDir` no início. Na linha 22 temos o código que escreve a mensagem de finalização do processo.

Copiando arquivos

Por fim, a última funcionalidade oferecida pela aplicação exemplo é a cópia de arquivos. Como a classe `File` não tem suporte à cópia de arquivos, uma alternativa é ler o arquivo de origem com uma *stream* de entrada (`FileInputStream`) e gravar o arquivo destino com uma *stream* de saída (`FileOutputStream`), como apresenta a [Listagem 7](#).

Listagem 7. Método para cópia de arquivos.

```
01. public static boolean copiarArquivo(String nomeArquivoOrigem, String
    nomeArquivoDestino) {
02.     System.out.println("Copiando arquivo ...");
03.     File arquivoOrigem = new File(nomeArquivoOrigem);
04.     File arquivoDestino = new File(nomeArquivoDestino);
05.     InputStream is = null;
06.     OutputStream os = null;
07.     try {
08.         is = new FileInputStream(arquivoOrigem);
09.         os = new FileOutputStream(arquivoDestino);
10.        byte[] buffer = new byte[1024];
11.        int length;
12.        while ((length = is.read(buffer)) > 0)
13.            os.write(buffer, 0, length);
14.    }
15.    catch (Exception e) {
16.        System.out.println("Erro no processamento:" + e.getMessage());
17.        return false;
18.    }
19.    finally {
20.        try {
21.            is.close();
22.            os.close();
23.        } catch (Exception e) {
24.            System.out.println("Erro ao fechar arquivos:" + e.getMessage());
25.            return false;
26.        }
27.    }
28.    System.out.println("Arquivo copiado.");
29.    return true;
30.}
```

Nessa listagem é proposta uma forma de copiar arquivos em Java. Para isso, inicialmente o método `copiarArquivos()` cria um objeto `File` para o arquivo de origem e um para o arquivo de destino. Já nas linhas 8 e 9 são criadas *streams* para esses arquivos. De forma genérica, uma *stream* é um canal virtual pelo qual dados podem ser transferidos entre objetos, programas ou computadores.

Para o arquivo a ser lido, é criada uma `FileInputStream` para realizar a leitura dos dados, e para o arquivo a ser criado, é criada uma `FileOutputStream` para realizar a gravação dos mesmos no novo arquivo. Tal procedimento é realizado nas linhas 10 a 14, onde são copiados dados do arquivo de origem para o arquivo de destino sempre em blocos de 1024 bytes, obtendo assim uma boa taxa de transferência, já que o sistema operacional lê e grava arquivos em blocos com tamanho menor. Se ocorrer algum erro no processamento, como um erro de escrita, este é mostrado no console. Como último passo do processamento, nas linhas 21 e 22 as streams são fechadas.

Executando FileApp na linha de comando

Para executar a classe `FileApp` e ver o resultado obtido pelas suas operações, é necessário utilizar a linha de comando do prompt (Windows) ou do shell (Unix/Linux). Porém, primeiro deve-se adicionar o caminho dos aplicativos `javac.exe` e `java.exe` na variável de ambiente `path` do sistema operacional. Esses aplicativos ficam armazenados no diretório `bin` da instalação do JDK. Feita essa configuração, na [Listagem 8](#) é mostrado um exemplo de execução de `FileApp` informando a opção para listar diretórios.

Como resultado do comando `listarDiretorio`, se o diretório `c:\teste` existe, será escrito no console um conteúdo semelhante ao apresentado na [Listagem 9](#), que expõe todos os arquivos e subdiretórios existentes.

Listagem 8. Execução do programa `FileApp` pela linha de comando.

```
cd <diretório logo acima do diretório fileAppPkg>
javac fileAppPkg/FileApp.java
java fileAppPkg.FileApp listarDiretorio c:\teste\
```

Listagem 9. Exemplo de resultado da execução dos comandos da [Listagem 8](#).

```
Listando arquivos ...
A/D Caminho Completo.
[A] E:\JohnTextos\ArtigosJava\Artigo2-File\Testes\a.txt
[A] E:\JohnTextos\ArtigosJava\Artigo2-File\Testes\b.txt
[A] E:\JohnTextos\ArtigosJava\Artigo2-File\Testes\c.txt
[A] E:\JohnTextos\ArtigosJava\Artigo2-File\Testes\d.txt
[D] E:\JohnTextos\ArtigosJava\Artigo2-File\Testes\diretorio
Listagem completa.
```

Criando uma interface gráfica para a aplicação exemplo

Conforme apresentado, a classe `FileApp` contém o método `main()` e pode ser executada pelo usuário no shell do sistema operacional. Esse mesmo resultado pode ser obtido usando os mesmos parâmetros em um ambiente de desenvolvimento como o Eclipse ou o NetBeans. Pensando em uma alternativa à linha de comando, podemos criar uma interface gráfica para a nossa aplicação, disponibilizando assim uma maneira mais intuitiva para o usuário interagir com as operações. Deste modo, ele pode simplesmente preencher alguns campos de texto e usar botões de ação para executar a funcionalidade desejada.

Colocando as mãos na massa, na [Listagem 10](#) é implementada uma nova aplicação, `FileAppUI`, com uma janela AWT e

componentes como campos de texto, combos e botões. O objetivo dessa aplicação é receber de forma mais simples os comandos que seriam escritos no shell e repassar à classe **FileApp** para execução das respectivas operações.

A classe **FileAppUI** representa uma aplicação gráfica desenvolvida em Java que usa a API básica AWT (*Abstract Window Toolkit*). Esta API fornece classes para exibição de janelas, caixas de diálogo, gerenciadores de *layout* (formas de organizar os componentes), *listeners* (código-fonte que é executado como resposta a um evento específico, como o de um clique), dentre outros.

Como demonstra **FileAppUI**, a disposição visual dos componentes na tela foi definida diretamente no código fonte, porém é possível realizar o mesmo trabalho de forma mais simples fazendo uso de recursos presentes em IDEs. Tais recursos permitem criar as janelas de uma aplicação de forma visual, restando ao desenvolvedor apenas arrastar os componentes, posicioná-los e redimensioná-los da maneira que desejarem usando o mouse.

Com esta classe construída, ao executá-la será exibida a janela exposta na **Figura 1**. A partir dela o usuário deve selecionar no campo *Operação* uma das operações disponíveis e, em seguida, preencher os campos de texto com os parâmetros requisitados. Feito isso, basta clicar no botão *Realizar Operação* para executar a operação.

Como a classe **FileAppUI** localiza-se no mesmo pacote da classe **FileApp** (veja a linha 1 da **Listagem 10**), não é necessário importar

esta última. Na linha 13, observa-se que **FileAppUI** estende **Frame**, herdando assim as características de janela do AWT. Nas linhas 16 a 22 são declarados os componentes que serão exibidos; componentes como **Label** (rótulo), **TextField** (campo de entrada de texto), **Choice** (combo de seleção) e **Button** (botão de ação).

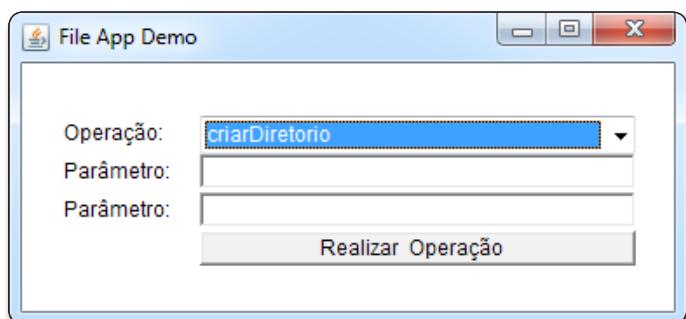


Figura 1. Interface gráfica da aplicação exemplo

No construtor da classe, iniciado na linha 24, é definido o tamanho da janela e o tamanho de seus componentes. Na linha 25 é informado o título da janela ao construtor da classe **Frame**. Já na linha 26, com **setLayout(null)**, o objetivo é não definir nenhum *layout*, a fim de que os componentes aceitem posições fixas. Depois disso, na linha 27, com **setSize()**, é definido o tamanho da janela.

Listagem 10. Aplicação gráfica para manipulação de arquivos.

```

01. package fileAppPkg;
02.
03. import java.awt.Button;
04. import java.awt.Choice;
05. import java.awt.Frame;
06. import java.awt.Label;
07. import java.awt.TextField;
08. import java.awt.event.ActionEvent;
09. import java.awt.event.ActionListener;
10. import java.awt.event.WindowAdapter;
11. import java.awt.event.WindowEvent;
12.
13. public class FileAppUI extends Frame {
14.
15.     private static final long serialVersionUID = 1L;
16.     private Label label1 = new Label("Operação:");
17.     private Label label2 = new Label("Parâmetro:");
18.     private Label label3 = new Label("Parâmetro:");
19.     private Choice param1 = new Choice();
20.     private TextField param2 = new TextField("");
21.     private TextField param3 = new TextField("");
22.     private Button operacao = new Button("Realizar Operação");
23.
24.     public FileAppUI(){
25.         super("File App Demo");
26.         setLayout(null);
27.         setSize(390, 180);
28.         add(label1);
29.         add(label2);
30.         add(label3);
31.         add(param1);
32.         add(param2);
33.         add(param3);
34.         add(operacao);
35.         label1.setBounds(30, 60, 70, 20);
36.         label2.setBounds(30, 82, 70, 20);
37.         label3.setBounds(30, 104, 70, 20);
38.         param1.setBounds(110, 60, 250, 20);
39.         param2.setBounds(110, 82, 250, 20);
40.         param3.setBounds(110, 104, 250, 20);
41.         operacao.setBounds(110, 126, 250, 20);
42.         param1.addItem("criarDiretorio");
43.         param1.addItem("removerDiretorio");
44.         param1.addItem("removerArquivo");
45.         param1.addItem("listarDiretorio");
46.         param1.addItem("copiarArquivo");
47.         addWindowListener( new WindowAdapter() {
48.             public void windowClosing(WindowEvent we) {
49.                 System.exit(0);
50.             }
51.         });
52.         operacao.addActionListener(new ActionListener() {
53.             public void actionPerformed(ActionEvent arg0) {
54.                 FileApp.main(new String[]{param1.getSelectedItem(), param2.getText(),
55.                     param3.getText()});
56.             }
57.         });
58.         public static void main(String[] args) {
59.             FileAppUI ui = new FileAppUI();
60.             ui.setVisible(true);
61.         }
62.     }

```

Java I.O: Trabalhando com arquivos em Java

Nas linhas 28 a 34, por sua vez, usando o comando `add()`, todos os componentes são adicionados à janela, e o tamanho e posicionamento dos mesmos são definidos nas linhas 35 a 41, com o comando `setBounds()`. O último elemento a ser criado é o combo, que contém os tipos de operações (linhas 42 a 46).

Ainda no construtor, nas linhas 47 a 51 é criado o *listener* com a ação de encerrar a aplicação para o botão fechar (localizado no canto superior direito da janela), e nas linhas 52 a 56 é criado o *listener* para o evento de clique do botão *Realizar Operação*. Observe que na linha 54, na ação do botão, o método `main()` da classe `FileApp` é chamado, recebendo como primeiro parâmetro a operação selecionada no combo, e como demais parâmetros, os campos de texto, executando em seguida a operação selecionada. Por fim, o método `main()` – descrito nas linhas 58 a 61 – instancia a janela `FileAppUI` e a exibe ao invocar o método `setVisible()`.

A partir disso, ao executar a aplicação o usuário poderá realizar as operações implementadas de forma simples e sem a necessidade de conhecer comandos. Como desafio, fica para o leitor a possibilidade de implementar novas funcionalidades, como a de mover diretórios e arquivos.

Como destacado neste artigo, o Java oferece diversas opções para a manipulação de arquivos e outros recursos, possibilitando a escrita de código fonte não acoplado a uma plataforma específica, fator que facilita a migração de projetos para outras plataformas, o que pode ocorrer por diversos motivos.

Devido a este e outros motivos, é muito importante conhecer todos os recursos oferecidos pelo JDK, a fim de facilitar o trabalho

de programação e ser possível realizar tarefas com qualidade, sem precisar de aplicações de terceiros e sem vincular fortemente aplicações a plataformas específicas. O desenvolvimento Java com conhecimento apropriado implica em código de alta qualidade, ótima legibilidade e independente de plataforma, características estas de grande importância e que facilitam a manutenção, testes e implantação dos sistemas construídos.

Autor



John Soldera

johnsoldera@gmail.com

É bacharel em Ciências da Computação pela UCS (Universidade de Caxias do Sul), mestre em Computação Aplicada pela Unisinos e cursa atualmente doutorado em Ciências da Computação pela UFRGS (Universidade Federal do Rio Grande do Sul). Trabalha com Java há 12 anos e possui a certificação SCJP.



Links:

Javadoc no site da Oracle.

<http://docs.oracle.com/javase/7/docs/api/>

Site da IDE Eclipse.

<http://www.eclipse.org/>

Site da IDE NetBeans.

<https://netbeans.org/>

Conhecimento faz diferença!

The advertisement features four issues of the magazine "engenharia de software" stacked vertically. The top issue is "Edição 29 :: Ano 3". The middle issue is "Edição 28 :: Ano 2". The bottom issue is "Edição 29 :: Ano 2". The fourth issue is partially visible on the left. A large red starburst graphic in the foreground contains the text "+ de 290 vídeos para assinantes". At the bottom, there is a call to action: "Faça já sua assinatura digital! | www.devmedia.com.br/es".

Faça um upgrade em sua carreira

Em um mercado cada vez mais focado em qualidade, ter conhecimentos aprofundados sobre requisitos, metodologia, análises, testes, entre outros, pode ser a diferença entre conquistar ou não uma boa posição profissional. Sabendo disso a DevMedia lançou uma publicação totalmente especializada em Engenharia de Software. Todos os meses você pode encontrar artigos sobre Metodologias Ágeis; Metodologias tradicionais (document driven); ALM (application lifecycle); SOA (aplicações orientadas a serviços); Análise de sistemas; Modelagem; Métricas; Orientação à Objetos; UML; testes e muito mais. **Assine Já!**



DEVMEDIA

Testes automatizados com JUnit

Crie seus primeiros testes com o JUnit

Uma das diretrizes do manifesto ágil diz que “*passamos a valorizar responder a mudanças mais que seguir um plano*”. Para que seja possível responder às mudanças de forma a agregar valor a nossos clientes, necessitamos de mecanismos que nos tragam segurança ao fazer alterações em nossa aplicação, evitando o surgimento de efeitos colaterais. O mecanismo essencial para termos essa segurança no código é o teste de software. Esse teste, quando feito por uma pessoa, ou equipe, é sujeito a falhas e pode demorar um tempo considerável caso seja necessário analisar o software em sua totalidade. É neste cenário que os testes automatizados vêm se tornando cada vez mais relevantes.

Um teste automatizado é um trecho de código que testa uma parte da aplicação que estamos desenvolvendo. Sendo o teste também um software, podemos executá-lo repetidas vezes, de forma automatizada. Como, normalmente, cada teste se refere a uma parte restrita da aplicação, para abrangê-la completamente, necessitamos de um conjunto deles, ao qual damos o nome de suíte de testes.

Uma vez que esses testes são criados utilizando uma linguagem de programação, a responsabilidade de garantir a qualidade do software deixa de ser tarefa exclusiva de um grupo específico de pessoas, os testadores, para ser compartilhada com todos os desenvolvedores. Uma analogia é pensarmos no processo de desenvolvimento de software como uma escalada, onde os testes automatizados são nossos equipamentos de segurança. Podemos fazer uma escalada sem os equipamentos, mas um bom profissional reconhece o perigo que isso representa.

A codificação de um teste automatizado se baseia em três etapas, que serão estudadas ao longo deste artigo: criação do cenário, invocação do método e a verificação do resultado esperado. Essa codificação, em geral, é feita na linguagem de programação em que a aplicação está sendo desenvolvida, muitas

Fique por dentro

Neste artigo vamos apresentar os principais conceitos sobre testes automatizados, um assunto que cada vez mais tem atraído profissionais que buscam qualidade e agilidade. Através de fundamentos e exemplos práticos, forneceremos uma base sólida para leitores que desejam iniciar os estudos nessa área.

vezes utilizando frameworks para auxiliar neste processo. Os exemplos desse artigo utilizarão o JUnit, um dos frameworks de testes mais consolidados na comunidade Java.

A criação de testes automatizados não tem como objetivo garantir que a aplicação funcione conforme solicitado pelo cliente, mas sim fornecer um feedback rápido que o software continua funcionando conforme o especificado pelos desenvolvedores do teste. Uma vez que as especificações dos testes nos dizem o que deve acontecer em determinados cenários, testes bem escritos podem ser uma boa forma de documentação da aplicação, diminuindo a necessidade de uma documentação extensa. Desta maneira, os testes automatizados corroboram também com a diretriz do manifesto ágil: “*passamos a valorizar software em funcionamento mais que documentação abrangente*”.

Escrever testes com boa qualidade é importante não só para criar uma fonte de documentação, mas também para facilitar a manutenção e evolução da aplicação e dos próprios testes. Eles são parte integrante do nosso software, necessitando da mesma atenção e zelo que as funcionalidades. Embora medir a qualidade de um teste seja uma tarefa difícil e passível de diferentes interpretações, esse artigo tem como objetivo não somente guiar o leitor na criação de seus primeiros testes, mas também fornecer embasamento para uma análise crítica da qualidade dos mesmos.

O estudo da qualidade dos testes, assim como outros conceitos apresentados nesse artigo, será guiado através de exemplos extraídos do nosso dia a dia. A criação desses exemplos não requer a utilização de uma IDE específica, mas com o intuito de facilitar o entendimento, iremos adotar o Eclipse, cujo link para download pode ser encontrado na seção **Links**.

Nível de abrangência dos testes automatizados

Teste automatizado é a denominação dada a todo tipo de teste que é programado com o objetivo de garantir o correto funcionamento de uma parte do software, de modo que possa ser executado de forma automática. A parte testada pode variar desde um único método, como por exemplo, calcular o saldo de uma venda, até um fluxo inteiro, contendo todas as etapas de uma venda. Quanto mais complexa a parte testada, maior o nível de abrangência do teste, que serve como base para classificação dos testes automatizados, como veremos a seguir.

Denominamos micro-testes os testes com o menor nível de abrangência. Seu objetivo é testar um comportamento de uma classe de forma isolada. Assim, necessitam de poucos recursos, sendo extremamente rápidos para completar sua execução. Como exemplo, podemos imaginar o teste de um método que valida CPF.

Nota

Os micro-testes também podem receber o nome de testes unitários. Entretanto, existem fontes que utilizam esse termo como referência para qualquer tipo de teste automatizado.

Quando o teste necessita de mais classes e/ou recursos para execução, dizemos que é um teste integrado. O objetivo desse tipo de teste é validar a integração entre as classes e camadas do sistema. Como exemplo de teste integrado, podemos imaginar um teste que invoca a funcionalidade de inserir uma venda no banco de dados e depois verifica se esta foi inserida de forma correta. É importante notar que os testes integrados tendem a ser mais lentos que os micro-testes, pois utilizam mais recursos e/ou camadas.

Elementos básicos de um micro-teste

Um micro-teste, conforme vimos na seção anterior, visa testar um comportamento de uma classe. Esse comportamento, em geral, é acionado através da invocação de um método. Considerando que o resultado da chamada de um método pode depender do valor presente nos atributos do objeto, antes de realizarmos a chamada, precisamos definir os atributos do objeto de modo a representar o cenário apropriado.

Dessa forma, podemos dividir nosso micro-teste em três partes, detalhadas a seguir:

- **Criação do cenário:** Na construção do cenário do nosso teste, iremos instanciar objetos e definir valores para seus atributos de modo a representar um estado possível da aplicação. A construção desse cenário varia em complexidade, sendo necessário, em alguns casos, apenas instanciar um objeto e, em outros, instanciar um conjunto de objetos com relações entre si;
- **Invocação do método:** Com o cenário criado, iremos invocar o método que estamos querendo testar e armazenar seu retorno para que possa ser utilizado na etapa de verificação do resultado esperado. Note que essa invocação é a ligação que existe entre nosso teste e o código da aplicação;
- **Verificação do resultado esperado:** Quando estamos desenvolvendo nosso teste, precisamos saber qual é o resultado esperado

quando a ação for executada no cenário criado. Assim, no caso de métodos que retornam valor, por exemplo, podemos comparar o valor esperado com o valor de retorno.

Configurando o JUnit

Como dissemos anteriormente, os testes automatizados são trechos de código que testam nossa aplicação. Dessa forma, é necessário utilizar uma linguagem de programação para a codificação do teste. Em geral, utilizamos a mesma linguagem na qual a aplicação foi construída; em nosso caso, o Java. Apesar de ser possível escrevermos nossos testes utilizando apenas as bibliotecas que são padrões do Java, existem frameworks que podem nos ajudar nesta tarefa, como por exemplo, o JUnit.

O JUnit é um framework bastante simples, porém poderoso, para auxiliar na escrita de testes automatizados. Por ser uma ferramenta consolidada na comunidade Java, a maioria das IDEs têm plugins para integrá-lo, sendo necessária somente a inclusão de dois JARs ao classpath da sua aplicação. Veremos a seguir como realizar esse procedimento utilizando o Eclipse.

Criando um novo projeto no Eclipse

No momento da inicialização do Eclipse precisamos, primeiramente, selecionar o workspace em que iremos trabalhar. O workspace é um diretório onde serão armazenadas informações dos nossos projetos e preferências do Eclipse. Você pode utilizar o diretório sugerido por ele, se preferir.

Agora que selecionamos o workspace, podemos criar um novo projeto no Eclipse para nossa aplicação de exemplo. Para isso, selecione o menu *File > New > Java Project* e defina um nome para o projeto. Note que existe a possibilidade de alterar o diretório onde seu projeto será armazenado, caso necessário. Em seguida, clique em *Finish*.

Dentro do novo projeto, temos um diretório com o nome *src*. É neste diretório que iremos criar nossas classes, tanto da aplicação de exemplo quanto dos nossos testes.

Adicionando o JUnit ao classpath

Para adicionar o JUnit à nossa aplicação, devemos fazer o download dos dois JARs necessários: *junit.jar* e *hamcrest-core.jar*. Eles podem ser encontrados no site do JUnit, clicando no link *Download and Install guide*, na subseção *Plain-old JAR*. As últimas versões disponíveis até o momento da escrita deste artigo são: *junit-4.11.jar* e *hamcrest-core-1.3.jar*.

Para adicionar esses JARs ao seu projeto no Eclipse, clique com o botão direito no projeto e selecione *Build Path > Configure Build Path*. Na aba *Libraries*, clique em *Add External JARs*. Depois de selecionar os dois arquivos obtidos, esta aba deve mostrar um conteúdo semelhante ao da **Figura 1**.

Nossa aplicação exemplo

Antes de escrevermos nossos testes automatizados, é importante conhecermos bem os requisitos do sistema que estamos querendo testar, de modo a empregar um foco maior nos pontos cruciais

da aplicação. Deste modo, nesta seção iremos estudar a aplicação de exemplo para a qual vamos construir os micro-testes. Essa aplicação se baseará em um sistema de vendas bastante simples, contendo apenas duas classes: **Venda** e **ItemVenda**.

Essas classes podem ser criadas através do Eclipse clicando com o botão direito sobre o diretório *src* e então selecionando *New > Class*. Vamos criar, primeiramente, a classe **ItemVenda**, descrita na **Listagem 1**. Esta classe representa cada item que compõe a venda realizada.

Uma lista de **ItemVenda** compõe, juntamente com os dados do comprador, uma **Venda**, como mostra a **Listagem 2**. Na classe **Venda** temos ainda o método **totalVenda()**, que é composto pela soma do valor de todos os itens mais o valor do frete, que é determinado como 10% do valor total da venda.

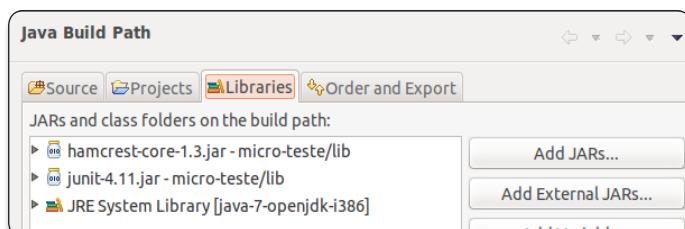


Figura 1. Adicionando os JARs do JUnit ao projeto

Listagem 1. Código da classe ItemVenda.

```
public class ItemVenda {  
  
    private String nomeProduto;  
  
    private Double valor;  
  
    public ItemVenda(String nomeProduto, Double valor) {  
        this.nomeProduto = nomeProduto;  
        this.valor = valor;  
    }  
    // gets e sets  
}
```

Listagem 2. Código da classe Venda.

```
public class Venda {  
  
    private String nomeComprador;  
  
    private String cpfComprador;  
  
    private List<ItemVenda> itens = new ArrayList<ItemVenda>();  
  
    public Double totalVenda() {  
        Double total = 0.0;  
        for (ItemVenda item : itens) {  
            total += item.getValor();  
        }  
  
        Double frete = total * 0.1;  
        total += frete;  
  
        return total;  
    }  
    // gets e sets  
}
```

Criando nosso primeiro micro-teste

A criação da interface para o usuário utilizar nossa aplicação de exemplo está além do escopo desse artigo. Porém, podemos imaginar que o método **totalVenda()** será invocado no momento em que formos apresentar o valor da venda para o usuário. Quando não estamos trabalhando com testes automatizados, é comum primeiramente criarmos essa interface com o usuário para, daí então, testar nosso método. Assumindo que estamos desenvolvendo uma aplicação web e que já temos a interface que exibe o resultado do método **totalVenda()**, o processo para testar nossa aplicação seria algo parecido com:

1. Subir o servidor web;
2. Logar na aplicação;
3. Navegar até a funcionalidade de venda;
4. Criar o cenário;
5. Verificar se o valor total está correto.

Note que caso seja notado um erro no método **totalVenda()**, teremos de corrigir o método e realizar esses passos novamente, o que torna nosso ciclo de desenvolvimento bastante lento. Além disso, toda alteração feita em nosso método requer uma nova execução desses passos.

Em cenários como este, podemos criar um micro-teste para termos um rápido feedback sobre o funcionamento do método.

Não perca tempo reinventando a roda!

COBREBEMX

**Componente completo para sua
Cobrança por Boleto Bancário
e Débito em Conta Corrente**

Mais de 40 exemplos
em diversas linguagens
de programação

Geração e leitura de arquivos
(remessa e retorno) nos padrões
FEBRABAN e CNAB

Testes e Downloads
gratuitos em nosso site



ACESSE E CONHEÇA O COMPONENTE EM:
WWW.COBREBEM.COM

Testes automatizados com JUnit

Uma vez codificado, podemos executar nosso micro-teste de forma automatizada, não necessitando de intervenção humana. Note que, se o teste falhar, teremos de corrigir o método e executar o teste novamente, que em milésimos de segundos nos trará o feedback se nossa alteração teve o efeito desejado.

Cada micro-teste é um método que cria o cenário, executa o método a ser testado e verifica o resultado esperado. Este método que descreve o micro-teste estará contido em uma classe. No nosso exemplo, criaremos a classe **VendaTest**, que contém o método **testValorTotal()**. O conteúdo dessa classe, assim como nosso micro-teste, pode ser visto na **Listagem 3**.

A criação do cenário e a chamada do método que estamos testando são feitas utilizando apenas as classes já existentes em nossa aplicação. Já a verificação do retorno do método é feita utilizando o método estático **org.junit.Assert.assertEquals()**, que recebe dois valores e faz a comparação entre eles. Caso os valores recebidos não sejam iguais, este método lança uma exceção e o teste falha. No nosso exemplo, o teste irá falhar se o total da venda for diferente de 165.

Nota

Este método, apesar de servir bem ao propósito de testar o funcionamento do método **totalVenda()**, poderia ser melhor escrito. Na subseção Legibilidade iremos refatorar esse código visando facilitar futuras manutenções.

Listagem 3. Código da classe VendaTest.

```
import org.junit.Assert;
import org.junit.Test;

public class VendaTest {

    @Test
    public void testTotalVenda() {
        // Cenário
        Venda varda = new Venda();
        varda.getItens().add(new ItemVenda("Camiseta", 50.0));
        varda.getItens().add(new ItemVenda("Calça", 100.0));

        // Chamada ao método que estamos testando
        Double totalVenda = varda.totalVenda();

        // Verificação do resultado esperado
        Assert.assertEquals(new Double(165), totalVenda);
    }
}
```

A annotation **@Test** presente no método **testTotalVenda()** indica que esse método é um teste do JUnit. Para executarmos esse método, podemos clicar com o botão direito sobre a classe **VendaTest** e selecionarmos **Run As > JUnit Test**, ou simplesmente pressionar **Ctrl + F11** enquanto o cursor estiver na classe **VendaTest**. O resultado desse teste está ilustrado na **Figura 2**.

Além do **assertEquals()**, existem outros métodos na classe **Assert** que servem para compararmos o retorno do método com o valor esperado. Em geral, esses métodos têm nomes

autoexplicativos, tais como: **assertNotEquals()**, **assertTrue()**, **assertFalse()**, **assertNull()** e **assertNotNull()**.

Embora nosso método de teste invoque apenas uma vez o método **assertEquals()**, temos a liberdade de utilizar esse método, assim como os outros supracitados, quantas vezes acharmos necessário durante um teste. Quando todas as comparações, feitas através desses métodos, são verdadeiras, o Eclipse desenha o resultado em verde e dizemos que nosso teste “está passando”. Neste momento é importante lembrar que, apesar do nosso teste estar passando, isso não garante que nosso método **totalVenda()** está de acordo com o que foi requisitado pelo cliente.

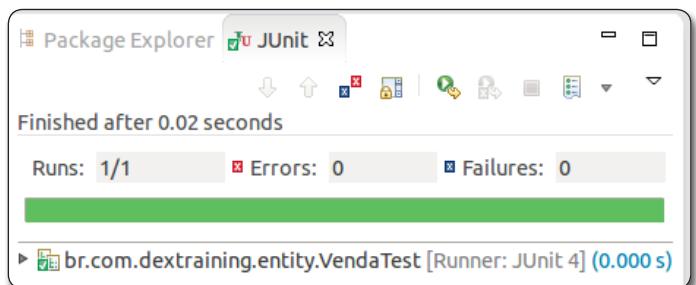


Figura 2. Executando nosso teste automatizado

Desenvolvendo testes com qualidade

Os testes automatizados que desenvolvemos são parte integrante da nossa aplicação, merecendo a mesma atenção que damos a outras partes do nosso código. Apesar de não existir uma receita pronta para produzir código de boa qualidade, temos diretrizes que nos guiam nessa direção. O mesmo acontece com os micro-testes. Sendo assim, nesta sessão abordaremos alguns pontos importantes na criação dos nossos testes.

Legibilidade

Como consequência da popularização das metodologias ágeis, documentação extensa vem perdendo prioridade nos projetos. É importante notar que as metodologias ágeis não dizem que documentação é algo ruim, mas sim que a relação custo-benefício deve ser analisada, levando em consideração o tempo de criá-la e mantê-la atualizada.

Visando a fácil continuidade e manutenção das aplicações, precisamos de uma “nova” forma de documentá-las. Nos últimos tempos, duas formas de documentação ganharam destaque: código limpo e testes automatizados.

Os testes automatizados se tornam uma fonte de documentação quando escritos de forma que, ao ler seu código, conseguimos facilmente entender o requisito que está sendo testado. Para ilustrar esse conceito, vamos analisar o teste apresentado na **Listagem 3**. Uma maneira de ler esse teste seria: “*Dado que temos uma Venda com dois ItemVenda, um custando 50 e outro 100 reais, quando invocarmos o método totalVenda, então seu retorno deve ser 165 reais.*”

Essa leitura não ajuda muito a entender o requisito do nosso sistema, pois não está explícito como a conta é feita. Uma maneira clara de observarmos isso é o fato de que o total da venda depen-

de do valor do frete, mas tal relação não foi descrita em nosso teste. Visando melhorar a legibilidade deste teste, poderíamos reescrevê-lo conforme a **Listagem 4**.

Listagem 4. Código da classe VendaTest reescrita para ter melhor legibilidade.

```
import org.junit.Assert;
import org.junit.Test;

public class VendaTest {

    @Test
    public void testTotalVenda() {
        // Cenário
        Venda venda = new Venda();
        venda.getItens().add(new ItemVenda("Camiseta", 50.0));
        venda.getItens().add(new ItemVenda("Calça", 100.0));

        // Chamada ao método que estamos testando
        Double totalVenda = venda.totalVenda();

        // Verificação do resultado esperado
        Double frete = (50.0 + 100.0) * 0.1;
        Double totalEsperado = 50 + 100 + frete;
        Assert.assertEquals(totalEsperado, totalVenda);
    }
}
```

Uma possível maneira, mas não única, de termos nosso teste depois de reescrito é: “*Dado que temos uma Venda com dois ItemVenda, um custando 50 e outro 100 reais, quando invocarmos o método totalVenda, então o frete deve ser 10% da soma dos itens e o total da venda deve ser a soma dos itens mais o frete*”.

Melhorar a legibilidade de um teste facilita não só a documentação do seu projeto, mas também o processo de manutenção do código (tanto do método que está sendo testado, quanto do teste desse método).

Independência entre testes

É comum executarmos mais de um teste em sequência. Como resultado, pode acontecer de um teste influenciar no estado inicial do próximo a ser executado. Testes que durante sua execução alterem o valor de atributos estáticos, por exemplo, podem influenciar no próximo, pois o valor do atributo estático se mantém durante a execução de todos os testes. Outro exemplo de cenário em que essa interferência pode ocorrer é quando a execução do teste altera dados do banco de dados. Deste modo, é uma boa prática escrevermos testes autossuficientes, ou seja, que são executados de forma independente do teste anterior. Para atingirmos esse objetivo, é comum realizarmos operações antes e depois da execução de cada teste.

No framework JUnit, utilizamos as annotations **@Before** e **@After** para essa finalidade. No exemplo da **Listagem 5**, indicamos ao JUnit para invocar o método **preparacao()** antes da execução de cada teste e o método **finalizacao()** após a execução de cada um deles. Esse comportamento se restringe aos métodos de teste presentes na classe em que foram declarados. Em nosso caso, na classe **VendaTest**.

Desacoplamento

Nem sempre é fácil criar micro-testes para as funcionalidades. Em geral, essa tarefa torna-se difícil quando o código que estamos querendo testar está desnecessariamente fortemente acoplado. O método **inserir()** da **Listagem 6** é um exemplo de código fortemente acoplado. Esse método insere uma **Venda** no banco de dados caso ela esteja válida. Consideramos uma venda válida quando ela tem pelo menos um **ItemVenda** e o **cpf** do comprador tem 14 caracteres.

Nota

A real validação de um CPF é bem mais complexa do que a apresentada nesse exemplo. Utilizamos uma validação simples por motivos didáticos.

Listagem 5. Utilização das annotations **@Before** e **@After**.

```
import org.junit.After;
import org.junit.Before;

public class VendaTest {

    @Before
    public void preparacao() {
        // Exemplo: Configuração de conexão com banco de dados
    }

    @After
    public void finalizacao() {
        // Exemplo: Finalizar conexão com banco de dados
    }

    // métodos de teste
}
```

Listagem 6. Método utilizado para inserir uma Venda no banco de dados.

```
public class VendaService {

    public void inserir(Venda venda) throws CPFInvalidoException, VendaInvalidaException {
        String cpf = venda.getCpfComprador();

        if (cpf.length() != 14) {
            throw new CPFInvalidoException(cpf);
        }

        if (venda.getItens().isEmpty()) {
            throw new VendaInvalidaException();
        }

        BancoDeDados.insert(venda);
    }
}
```

Neste cenário, para criarmos o teste da validação do CPF, necessitamos criar uma **Venda**, adicionar um **ItemVenda** a ela, informar um CPF e invocar o método **inserirVenda()**, como mostra a **Listagem 7**. Nesse caso, não precisamos invocar nenhum método de assert, como por exemplo, **assertEquals()**, pois se o CPF for inválido, o método **inserir()** lançará uma exceção, o JUnit entenderá que o teste falhou por conta disso, e apresentará a barra ilustrada na **Figura 2**, mas agora com a cor vermelha.

Note que foi necessário criar uma **Venda** e adicionar um **ItemVenda** para conseguir testar a validação do CPF. Isto acontece porque o trecho de código que valida o CPF está fortemente acoplado com o método **inserir()**.

Uma maneira de desacoplarmos a validação do CPF é criar uma classe específica para essa finalidade, conforme mostra a **Listagem 8**, e alterar o método **inserir()**, que agora deverá invocar o novo método criado.

Temos, assim, uma classe desacoplada do método **inserir()**, facilitando a criação do nosso teste, como podemos observar na **Listagem 9**.

Listagem 7. Teste para validar o CPF.

```
import org.junit.Test;  
  
public class VendaServiceTest {  
  
    @Test  
    public void testValidacaoCPF() throws CPFInvalidoException,  
        VendaInvalidaException {  
        Venda venda = new Venda();  
        venda.setCpfComprador("000.000.000-00");  
        venda.getItens().add(new ItemVenda("Calça", 50.0));  
  
        // Sem assert, pois se o cpf for invalido, teremos uma exception,  
        // sinalizando a quebra do teste  
        new VendaService().inserir(venda);  
    }  
}
```

Listagem 8. Código da classe para validar CPF.

```
public class ValidadorCPF {  
  
    public boolean valido(String cpf) {  
        return cpf.length() == 14;  
    }  
}
```

Listagem 9. Teste para a classe ValidadorCPF.

```
import org.junit.Assert;  
import org.junit.Test;  
  
public class ValidadorCPFTest {  
  
    @Test  
    public void testCPF() {  
        ValidadorCPF validador = new ValidadorCPF();  
  
        Assert.assertTrue(validador.valido("000.000.000-00"));  
        Assert.assertFalse(validador.valido("000.000.000"));  
    }  
}
```

A conclusão que chegamos através desse exemplo é que existe uma relação direta entre o grau de dificuldade na criação de um teste e a qualidade do código que está sendo testado. Se o código está difícil de testar, provavelmente existem melhorias a serem feitas no código que o tornarão mais fácil de testar e de dar manutenção.

O que devemos testar?

Identificar o que devemos testar em uma funcionalidade nem sempre é uma tarefa fácil, pois depende do cenário em questão e também da importância da mesma. Deste modo, uma funcionalidade com alta relevância, como realizar venda, deveria ter testes mais bem elaborados que outras mais periféricas, como cadastrar fornecedor.

Testes bem elaborados são aqueles capazes de detectar um possível efeito colateral causado por uma alteração. Para tal, precisamos testar as diversas partes que compõem a funcionalidade em questão. Mais do que isso, queremos não somente testar as diversas partes, mas também exercitar diferentes cenários que possam ocorrer. Assim, quando analisamos o que devemos testar, é importante termos em mente duas questões, analisadas a seguir: quais métodos devemos testar e quais cenários devemos testar para cada método.

Quais métodos devemos testar?

Devemos testar os métodos que fazem algum processamento cujo retorno não seja trivial. Em outras palavras, para tudo que gostaríamos de fazer um teste manual, seria interessante termos um teste automatizado. Tomando como exemplo o cálculo do total da venda, provavelmente gostaríamos de testar manualmente essa funcionalidade. Assim, parece prudente criar um teste automatizado para esse cenário. Por outro lado, não existe a necessidade de criarmos testes para métodos que têm retorno óbvio, como é o caso dos métodos set e get de nossas classes.

Analizar a necessidade de um teste pode não ser tão simples como nos exemplos descritos anteriormente. Sendo assim, em geral, quando em dúvida, é interessante optar pela criação do teste, pois podemos removê-lo sem dificuldades posteriormente.

Quais cenários devemos testar para cada método?

Testar todos os cenários que podem ocorrer é simplesmente impossível. Assim, devemos buscar os cenários que julgamos mais frequentes e importantes. No método que calcula o total da venda, por exemplo, optamos por criar somente um cenário onde temos dois itens na venda. Criar um cenário com três itens pode ser desnecessário se considerarmos improvável que o método **totalVenda()** funcione corretamente com dois itens e não funcione com três.

Por outro lado, para o método que valida CPF, criamos dois cenários de teste: um com um valor válido e outro com um valor inválido. Essa decisão foi tomada baseando-se no fato de que conhecemos a implementação do método e sabemos que o retorno depende unicamente do tamanho da **String** informada. Caso esse método verificasse outras características, como por exemplo, a existência de apenas pontos, traços e números, seria interessante adicionar um cenário onde temos uma letra no CPF e verificar que o retorno é **false**.

Além de testarmos os principais cenários, é comum realizarmos testes em cenários “especiais”, como por exemplo, passando uma **String** vazia ou **null**. Outro exemplo de cenário que podemos con-

siderar como especial, mas que não se aplica em nossos exemplos, é passar o número zero ou um número negativo.

Defeitos

A utilização de testes automatizados não garante que o usuário final não irá encontrar defeitos ao utilizar o sistema. Entretanto, defeitos que forem resolvidos não deveriam tornar a se repetir em futuras versões. Isso pode ser garantido criando um teste para cada defeito encontrado.

A resolução de um defeito pode ser feita seguindo duas abordagens diferentes:

1. **Primeiro** corrigir o código fonte e **depois** criar o teste;
2. **Primeiro** criar o teste e **depois** corrigir o código fonte.

Na primeira abordagem, ao corrigir o código fonte antes de criar o teste, temos a desvantagem de não saber se o teste criado detectaria o erro encontrado. Ou seja, não temos a garantia de que o teste criado após a correção do defeito seja capaz de indicar caso esse defeito volte a ocorrer.

Já na segunda abordagem não temos essa dúvida, pois quando criamos o teste ele falha, evidenciando a finalidade do teste. Outra vantagem que podemos observar é que dessa maneira não existe

a possibilidade de nos esquecermos de criar o teste, já que este é feito antes da correção do defeito. Essa abordagem é parte de uma técnica conhecida como TDD (*Test Driven Development*). Para mais detalhes sobre este assunto, consulte o artigo “TDD: Evidenciando a importância do Teste”, referenciado na seção [Links](#).

Testes de métodos que utilizam banco de dados

Quando queremos testar métodos que utilizam banco de dados, precisamos configurar o acesso ao mesmo antes de executarmos os testes. O cenário mais simples para realizarmos esse tipo de teste é assumir que temos um banco de dados instalado e acessível. Porém, neste cenário, perdemos a possibilidade de executar o teste em qualquer máquina.

Uma alternativa para esse problema é a utilização de um banco de dados que não exija instalação, como o HSQLDB, cujo site oficial e link para download são apresentados na seção [Links](#). O HSQLDB é um banco escrito em Java que não exige instalação, pois é executado através de um JAR. Outra importante característica desse banco é a capacidade de ser executado em memória, o que pode ser bastante útil em tempo de desenvolvimento, pois os dados serão automaticamente apagados quando finalizarmos sua execução.

CURSOS ONLINE

A Revista SQL Magazine oferece aos seus assinantes uma série de Cursos Online de alto padrão de qualidade.



CONHEÇA OS CURSOS MAIS RECENTES:

- **Cursos: Curso de noSQL (Redis) com Java**
- **Desenvolvimento para SQL Server com .NET**
- **Curso PostgreSQL - Treinamento de banco de dados (Curso Básico)**

Para mais informações :

<http://www.devmedia.com.br/cursos/banco-de-dados>

(21) 3382-5038

É importante lembrarmos que testes que fazem acesso ao banco de dados são mais lentos que os micro-testes, estudados anteriormente. Essa diferença se dá pelo fato de que as operações realizadas no banco de dados, em geral, são consideravelmente mais complexas que as realizadas em um micro-teste. Como exemplo, podemos imaginar as operações necessárias para a realização de um comando `insert`. Para completar o comando, o banco de dados necessita de uma série de operações, como verificar se os tipos de dados informados estão corretos, verificar chaves estrangeiras, criação de índices, entre outras. Mesmo considerando um banco de dados rodando em memória, essas pequenas operações, quando somadas, fazem com que esse tipo de teste demore centenas, ou até mesmo milhares de vezes mais que nossos micro-testes.

Essa diferença de velocidade pode parecer pequena quando estamos rodando apenas um teste, mas é consideravelmente grande quando rodamos centenas ou milhares.

Integração contínua

Quando temos uma equipe de desenvolvedores contribuindo na construção de uma mesma aplicação, surge a necessidade de gerenciarmos as alterações feitas por cada desenvolvedor. Em Java, essas alterações podem ser gerenciadas através de ferramentas de controle de versão como Subversion e Git. Tais ferramentas permitem que cada desenvolvedor envie alterações para um servidor central em um processo chamado `commit` e outros desenvolvedores possam atualizar sua cópia local com essas mudanças.

Além dessas opções, existem as ferramentas de integração contínua. Estas permitem que a cada commit feito seja executada a suíte de testes, possibilitando detectar, de forma quase que imediata, um eventual efeito colateral causado pelas alterações do commit. Quando a alteração feita quebra um teste, a ferramenta de integração contínua informa, geralmente por e-mail, qual teste falhou e qual alteração gerou essa falha.

A ferramenta de integração contínua inicia a execução dos testes quando acontece um commit. Neste cenário, diante do tamanho da equipe, pode acontecer de um novo commit ser realizado enquanto os testes de outro commit estiverem sendo executados. Quando isso acontece, o último commit será adicionado a uma fila de espera, rodando seus testes apenas após o término do primeiro. Esse cenário se torna particularmente ruim quando o primeiro commit quebra algum teste, pois como o segundo também contém as alterações do primeiro, fatalmente irá falhar, independente de ter quebrado ou não algum teste. Com isso, não teremos um feedback preciso sobre o segundo commit.

Esta situação também tende a acontecer com mais frequência à medida que o tempo de execução dos testes aumenta. A principal arma que temos para evitar que nossa suíte de testes se torne lenta é construir um número significativamente maior de micro-testes do que testes integrados, uma vez que os micro-testes são muito mais rápidos.

Embora fora do escopo deste artigo, citaremos duas ferramentas que podem ajudar no processo de criação de um ambiente de integração contínua:

- **Maven:** Automatizador de builds, permitindo rodar os testes por linha de comando;
- **Travis:** ferramenta (serviço) de integração contínua gratuita para projetos open source.

Os sites de ambos podem ser encontrados na seção **Links**.

Este artigo abordou os principais conceitos dos testes automatizados, com ênfase nos micro-testes. Vimos também que o código de teste é parte integrante da aplicação, necessitando de constante atenção com sua qualidade. Por ser um assunto bastante abrangente, nosso estudo não teve como objetivo se aprofundar em um tópico específico, mas sim gerar uma base sólida para futuros estudos.

Os testes automatizados são, provavelmente, a melhor ferramenta que temos para conseguir responder a mudanças com rapidez e segurança. Eles são a base conceitual para práticas que vêm revolucionando a maneira como entregamos software. Dentre elas, podemos destacar TDD, Continuous Integration e Continuous Delivery.

Apesar dos benefícios decorrentes da utilização de testes automatizados, existe um custo relacionado à criação e manutenção dos mesmos. Contudo, esse custo tem se mostrado pequeno quando comparado aos benefícios que eles trazem, principalmente quando adquirimos mais familiaridade com técnicas relacionadas ao assunto, como as apresentadas nesse artigo.

Autor



Andrei de Oliveira Tognolo

[andreatognolo@gmail.com](mailto:andreitognolo@gmail.com)

É Bacharel em Ciência da Computação pela UNICAMP. Trabalha com Java há seis anos, desenvolvendo aplicações sob medida na empresa Dextra. Também é instrutor Scrum e de cursos relacionados à Java na Dextraining (www.dextraining.com.br).



Links:

Site oficial do Eclipse.

www.eclipse.org

Site oficial do JUnit.

junit.org

Site oficial do HSQLDB.

<http://hsqldb.org>

Endereço para download do HSQLDB.

http://sourceforge.net/projects/hsqldb/files/hsqldb/hsqldb_2_3/hsqldb-2.3.2.zip/
download

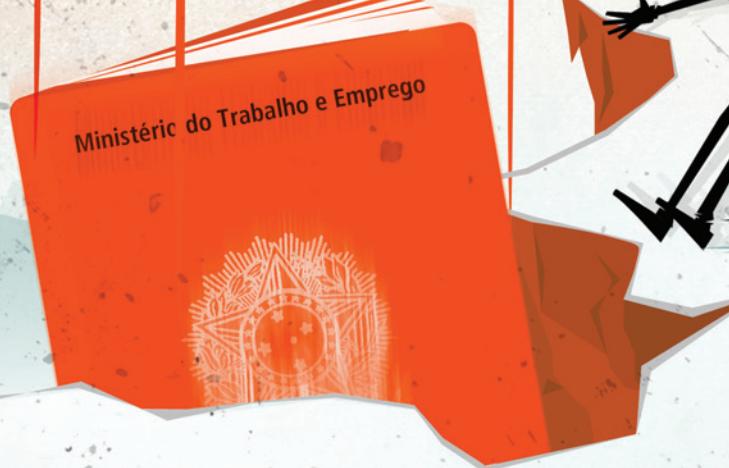
Site oficial do Maven.

<http://maven.apache.org>

Site oficial do Travis.

<https://travis-ci.com/>

SEU CARRO TEM SEGURO, SUA SAÚDE TEM SEGURO, MAS E O SEU EMPREGO... TÁ SEGURO??



NÃO DEIXE JUSTAMENTE A SUA CARREIRA FICAR EM RISCO!

Manter-se atualizado com todas as novidades do mercado de desenvolvimento é obrigação de todo bom programador. Faça agora mesmo um seguro para a sua carreira. Seja um assinante MVP!

Saia do risco!

TENHA ACESSO A:



+ DE 260 CURSOS ONLINE



09 REVISTAS MENSais



7.850 VÍDEO-AULAS

POR APENAS **59,90** MENSais



QUEM TEM ESTÁ TRANQUILO.



DEVMEDIA

Acesse: www.devmedia.com.br/mvp

Android Studio: Primeiros passos com Android

Construa aplicativos para o ecossistema móvel do Google com a IDE Android Studio

Basta olhar para os lados nas ruas, no caminho até o trabalho, na praia, na escola, etc. para perceber o quanto as tecnologias móveis estão presentes nas nossas vidas, fazendo cada vez mais parte do nosso cotidiano.

As pessoas estão conectadas o tempo todo, realizando os mais diversos tipos de tarefas, desde a comunicação por meio de uma rede social até a localização por um serviço de mapas, passando pelo recebimento de e-mails e daí por diante.

A cada dia novos serviços e aparelhos surgem, atendendo e gerando demandas para usuários cada vez mais famintos por aplicações. E o mercado acompanha esse cenário com bastante otimismo.

Em janeiro desse ano, uma reportagem do *IDG Now* trouxe dados de duas consultorias internacionais que apontam 2014 como o ano em que haverá mais smartphones do que PCs sendo utilizados em todo o mundo.

Outra reportagem ainda mais recente informa que em 2013, somente em lojas online, a procura por smartphones no Brasil foi duas vezes maior do que a procura por celulares convencionais, movimentando cerca de R\$1,3 bilhão.

Outra categoria de dispositivos móveis bastante populares são os tablets. Embora suas vendas mundiais no final de 2013 tenham apresentado uma tendência de estabilização, sua presença no mercado mundial já atinge um número aproximado de 76 milhões de unidades. Segundo a consultoria IDC, essa estabilização nas vendas se deve à saturação de mercado, o que é impressionante dado o tempo em que esse tipo de dispositivo está disponível. No Brasil, esse mercado cresceu quase 280% desde 2011.

Pegando carona na popularização do uso de smartphones e tablets, aumenta também a demanda dos usuários por aplicativos e conteúdo e, dessa forma, o movimento para a portabilidade das mídias para plataformas móveis ganha força. Serviços baseados na plataforma tradicional de internet – a exemplo do Facebook e YouTube – intensi-

Fique por dentro

Esse artigo demonstra de maneira prática como iniciar o desenvolvimento de apps para Android. Na medida em que os dispositivos móveis se tornam mais populares e acessíveis e a plataforma Android aumenta sua presença nesse mercado, aprender a desenvolver aplicativos para esse ambiente se torna bastante importante a todo desenvolvedor.

ficam sua presença nos ecossistemas móveis, além do crescimento intenso da navegação na Web sobre a plataforma *mobile*.

No Brasil, que já desponta como o maior mercado desenvolvedor de aplicativos móveis da América Latina, uma pessoa fica cerca de 80 minutos diários usando algum tipo de dispositivo móvel. Portanto, nesse mercado, que movimentou quase US\$ 30 bilhões em 2013, o Brasil deve representar uma considerável fatia.

Dentre os grandes *players*, plataformas como Android, do Google, e iOS, da Apple, devem manter a predominância nesse mercado, embora a Microsoft e o BlackBerry estejam aumentando o seu portfólio de aplicativos. Atualmente, somente o iOS gera mais de dois terços da receita de aplicativos para smartphones e tablets em todo o mundo, uma vez que seus usuários estão mais dispostos a investir na compra de apps. Por outro lado, o sistema operacional Android já ultrapassou o iOS em quantidade de apps disponíveis em sua loja virtual.

Com o lançamento do Android Native Development Kit (Android NDK), que dá suporte a código nativo, os desenvolvedores passaram a poder escrever aplicativos *third-party* (vide **BOX 1**) usando, além de Java, C ou C++. E em meados de 2010, o Google lançou o Android Scripting Environment (ASE), possibilitando também o desenvolvimento de aplicativos por meio das linguagens de script *perl*, *JRuby*, *Python* e *LUA*.

Hoje em dia, algumas outras linguagens são suportadas, mas como a máquina virtual Dalvik (vide **BOX 2**) possui algumas semelhanças com a JVM (vale ressaltar, no entanto, que a Dalvik não é uma JVM!), a linguagem Java conseguiu estabelecer uma base significativa nesse ecossistema e se colocar, de certa forma, como a linguagem “oficial” para a plataforma Android.

BOX 1. Aplicativos third-party

O jargão *third-party* é utilizado para descrever sistemas desenvolvidos por terceiros para serem distribuídos livremente, de forma gratuita, ou vendidos por outros que não sejam os fabricantes originais da plataforma sobre a qual os softwares foram desenvolvidos. No Google Play, loja de apps do Android, estão disponíveis aplicativos desenvolvidos por diversos fabricantes, como Instagram, WhatsApp, entre outros.

BOX 2. Dalvik Virtual Machine

Dalvik é a Máquina Virtual do sistema operacional Android. É responsável por gerenciar os processos e rodar os aplicativos nos aparelhos com essa plataforma, sendo, portanto, parte integral desse sistema operacional. Atualmente, o Google realiza testes com uma nova máquina virtual, a ART, que deve substituir o Dalvik em um futuro próximo.

Vale lembrar também que o Android não é a primeira incursão da plataforma Java no mundo dos dispositivos móveis. Já há bastante tempo que a API Java Micro Edition (Java ME) está disponível, fornecendo uma maneira de lidar com a construção de aplicações para pequenos (e portáteis) dispositivos, com limitações de memória, energia, etc.

A presença do Java nesse mercado também ajudou a colocar a linguagem em uma posição privilegiada no mercado de desenvolvimento de aplicativos. No entanto, as arquiteturas Java ME e Android são bastante diferentes, não havendo sequer compatibilidade dos *Midlets* produzidos para Java ME com o ambiente Android.

No Java ME, o aparelho – um celular ou outro tipo de dispositivo móvel em que o aplicativo vai rodar – é responsável por abstrair o hardware e traduzir suas características para as camadas superiores de software, enquanto as camadas inferiores são embutidas no sistema operacional do dispositivo em questão.

No Android, é responsabilidade do sistema operacional realizar essa abstração. Dessa forma, por exemplo, em qualquer dispositivo, o visual do aplicativo será o mesmo.

Os *Midlets*, por outro lado, possuem um conjunto restrito de componentes, dificultando o desenvolvimento de interfaces gráficas mais sofisticadas.

O gerenciamento de *threads* e processos também apresenta distinções. No Java ME, apenas uma instância da JVM fica em execução, enquanto o Android é capaz de gerenciar várias instâncias do Dalvik. Cada componente pode ser executado por um processo específico, ficando por conta do desenvolvedor o gerenciamento de *threads* que, por exemplo, lidarão com tarefas que consomem mais memória ou processamento.

Neste artigo vamos desenvolver um pequeno aplicativo para o ecossistema Android. Esse aplicativo será composto por um formulário com

alguns campos e será responsável por armazenar os dados informados pelo usuário no banco de dados. Além de ilustrar as características essenciais desse ecossistema, vamos explorar a instalação de todo o ambiente e o uso da IDE Android Studio para apoiar o desenvolvimento.

Instalação do Android SDK

Como vamos desenvolver utilizando Java, é imprescindível a instalação prévia de um Java Software Development Kit, ou seja, antes de iniciar a instalação do Android SDK, assegure que seu ambiente de desenvolvimento possui um Java SDK instalado e configurado apropriadamente. O endereço para *download* do Java SDK pode ser visto na seção **Links**.

De acordo com o site do Android, há duas maneiras de configurar um ambiente de desenvolvimento: baixando somente o SDK para utilizá-lo com uma IDE que você já tenha, como o Eclipse, por exemplo; ou baixando um *Bundle*, que inclui, além do SDK, uma IDE preparada para o desenvolvimento, que pode ser o próprio Eclipse ou então o Android Studio. Neste artigo optamos pela segunda opção, utilizando o Android Studio.

O *bundle* com Android Studio inclui, além da IDE, todas as ferramentas do SDK e o emulador que será utilizado para testar o aplicativo. Para baixá-lo, visite o site oficial de desenvolvedores da plataforma Android (ver seção **Links**) e no menu *Developer Tools*, clique em *Android Studio*, clicando em seguida no *link* para *download*, que indicará o sistema operacional que você estiver utilizando. Na sequência, para que o *link* para o arquivo fique disponível, é necessário concordar com os termos e condições, como mostra a **Figura 1**.

Feito isso, clique no botão *Download* e baixe o pacote, que possui aproximadamente 500 Mb.

No Linux, é necessário descompactar o arquivo para um diretório desejado, enquanto no Windows basta executar o instalador, que

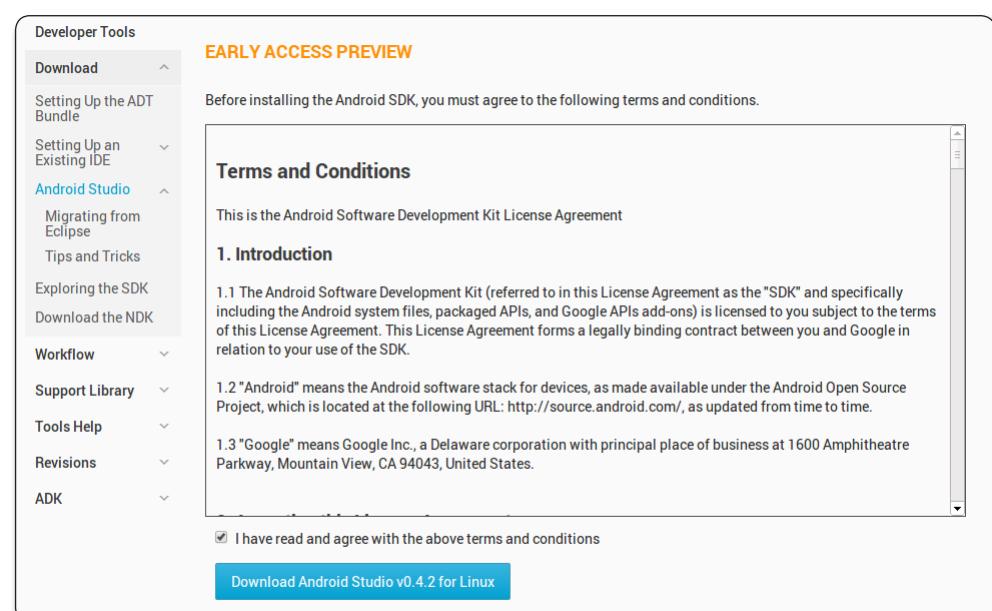


Figura 1. Termos e condições de uso do Android Studio

Android Studio: Primeiros passos com Android

se encarrega inclusive de criar o atalho para a execução da IDE. Para executar o Android Studio no Linux, vá até o diretório em que o arquivo foi descompactado, navegue até o subdiretório `android-studio/bin` e execute o arquivo `studio.sh`.

Assim, o ambiente está configurado e pronto para o desenvolvimento.

Desenvolvendo um primeiro aplicativo para Android

Embora o uso de uma IDE simplifique o desenvolvimento, é importante estabelecer alguns conceitos antes de iniciar a construção do aplicativo.

Nas próximas seções, além de demonstrar os passos necessários para desenvolver seu aplicativo, forneceremos detalhes da estrutura padrão de um aplicativo para Android, começando por conceitos essenciais para iniciar a construção do *app*.

Elementos essenciais do aplicativo

Considerando que sua arquitetura é baseada no padrão *Model-View-Controller* (MVC), todo aplicativo Android possui, entre outros, os seguintes elementos estruturais:

- O arquivo `AndroidManifest.xml` onde são descritas as características essenciais e todos os componentes do aplicativo;
- Uma (ou muitas) *activity*, cujas classes serão os *Controllers* do aplicativo;
- Um documento XML para cada *activity*, que correspondem à camada *View*;
- A classe *R*, responsável por fazer a comunicação entre as camadas.

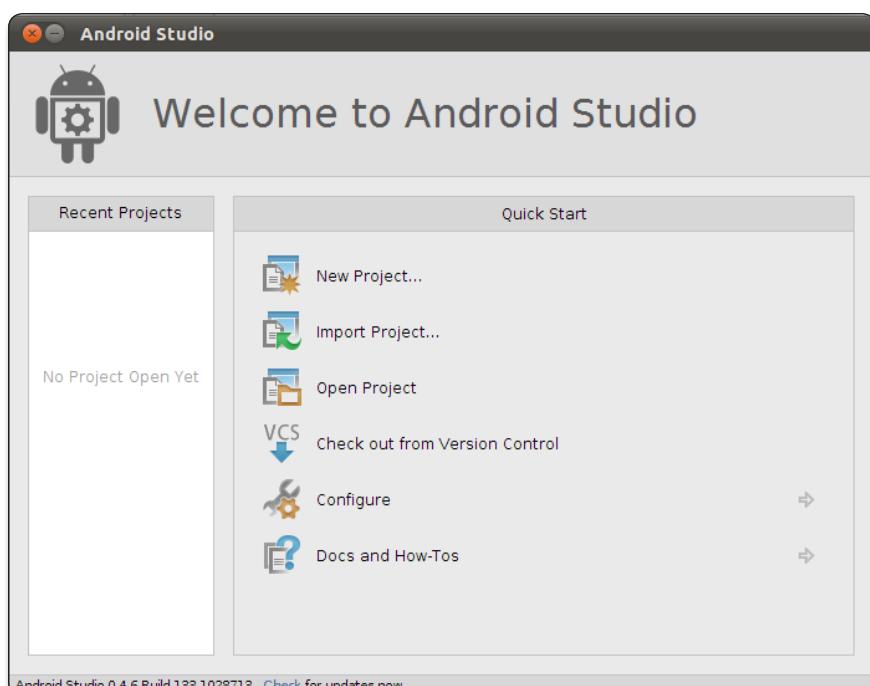


Figura 2. Tela inicial do Android Studio

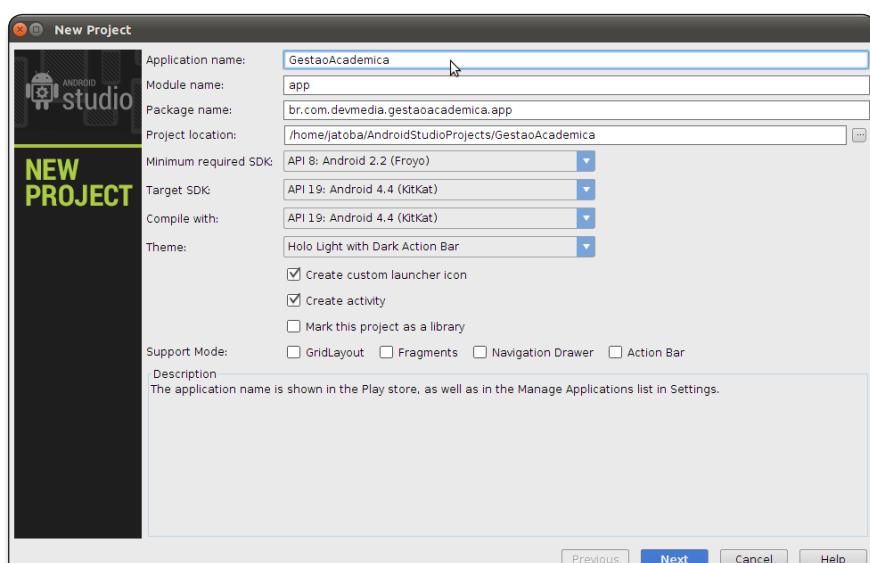


Figura 3. Janela de propriedades do projeto

Uma interface gráfica no Android é construída a partir de *ViewGroups*, utilizados para definir a organização dos componentes na tela (semelhante aos *Layouts* do *Swing*), e uma hierarquia de *Views*, componentes de interface gráfica como botões ou campos de texto. A forma pela qual esses elementos compõem a aparência da interface é definida em documentos XML.

A estrutura de uma *activity* é semelhante a uma página HTML em um site. Em uma página Web, o código HTML define os componentes e o aspecto visual, enquanto seu comportamento e a resposta a eventos podem ser implementados usando JavaScript, por exemplo. No caso de uma *activity*, sua aparência é determinada em um documento XML, enquanto seu comportamento é construído em código Java.

Utilizando o Android Studio para desenvolver sua primeira app

O Android Studio é um ambiente de desenvolvimento bastante semelhante ao Eclipse. Ele fornece ao desenvolvedor ferramentas para implementar e depurar aplicativos para o ecossistema Android. Entre seus recursos estão os assistentes e *templates*, que permitem criar rapidamente *designs* e componentes, e um editor gráfico de *layouts*, que permite “arrastar” componentes e realizar *previews* das telas do *app* de forma bastante simples.

A tela inicial do Android Studio pode ser vista na Figura 2. Para criar um novo projeto, basta clicar na opção *New Project....*

Feito isso, será apresentada a janela com as propriedades do projeto, como pode ser visto na Figura 3. Nessa janela, digite o nome do projeto, que no nosso caso, se chamará “GestaoAcademica”.

Não coloque espaços ou caracteres especiais. Além disso, é recomendável que se utilize a convenção de manter as letras iniciais das palavras em maiúsculas.

Atente também para o campo “*Package name*”. Nesse campo, informe o pacote em que o aplicativo e seus elementos devem ser colocados.

Nessa janela, ainda é preciso especificar para que versão do Android SDK o *app* será compilado. Esse campo é importante porque pode determinar com que dispositivos o aplicativo será compatível.

Não vamos explorar nesse artigo as versões existentes do Android, nem mesmo que versão a maioria dos dispositivos utiliza, pois isso é assunto para um artigo à parte, mas utilizaremos a versão sugerida pela IDE – 4.4 KitKat. É importante ressaltar que quanto menor for a versão utilizada, maior o número de dispositivos disponíveis; no entanto, há uma redução considerável de ferramentas e funcionalidades. Uma vez atribuídos esses valores, clique em *Next*.

A janela seguinte apresentará opções de ícone para seu aplicativo. O campo *Foreground* determina se o aplicativo será acionado ao clicar em uma imagem, um *clipart* ou simplesmente um texto exibido na área de trabalho do sistema operacional Android. Se escolher “ícone”, no campo *Image file* é possível selecionar a figura que será exibida como ícone no aparelho, para dar acesso ao aplicativo. Caso escolha “*clipart*”, também é possível selecionar o arquivo que será utilizado como ícone. De modo semelhante, como mostra a Figura 4, é possível aplicar outras configurações, como o formato do ícone (que pode ser um círculo, ou ter as bordas arredondadas, por exemplo) ou a cor de fundo da interface do aplicativo.

Até o momento, estão sendo configuradas propriedades gerais do aplicativo. A partir do próximo passo, passaremos a tratar da criação da tela principal do *app*. Clicando em *Next*, somos redirecionados para a janela em que escolheremos a estrutura dessa tela.

No Android, cada tela do aplicativo corresponde a uma *activity*, composta de um documento XML e uma respectiva classe Java. Para este exemplo, escolhemos, dentre os *templates* disponíveis, a opção *Blank Activity* (veja a Figura 5).

A estrutura de uma *Blank Activity* é a mais básica de todas. Ela inclui uma barra de título que pode exibir o ícone do aplicativo, um menu de opções localizado no canto superior direito e um espaço abaixo da barra de título em que os componentes visuais podem ser dispostos.

Por sua vez, o template *Fullscreen Activity* fornece uma tela que possibilita alternar entre a aparência de tela cheia (ocupando todo o display do dispositivo) e um layout que apresenta uma barra de título no topo (exibida ao tocar na tela do aparelho). Já *Login Activity* é um template que agiliza a construção de telas de

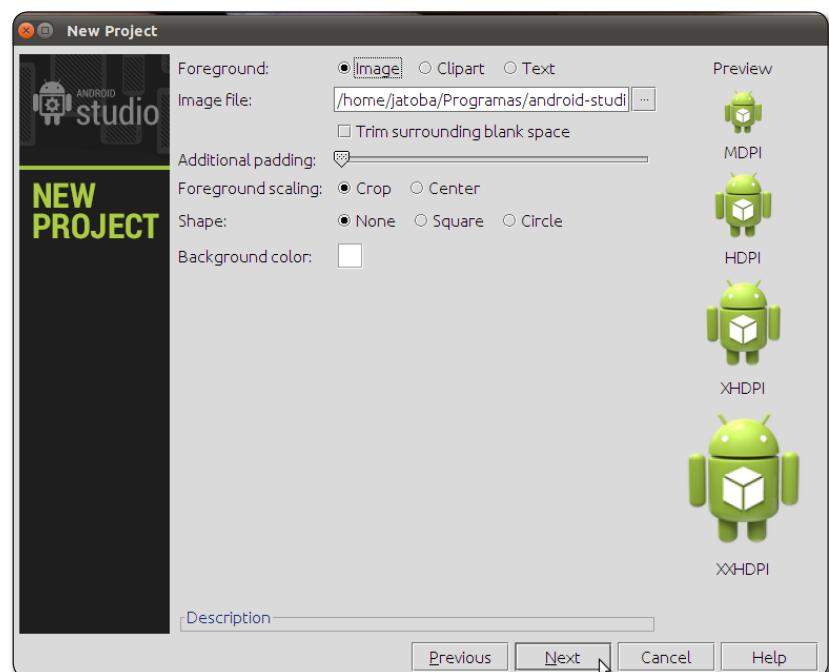


Figura 4. Novo projeto no Android Studio - Janela 2

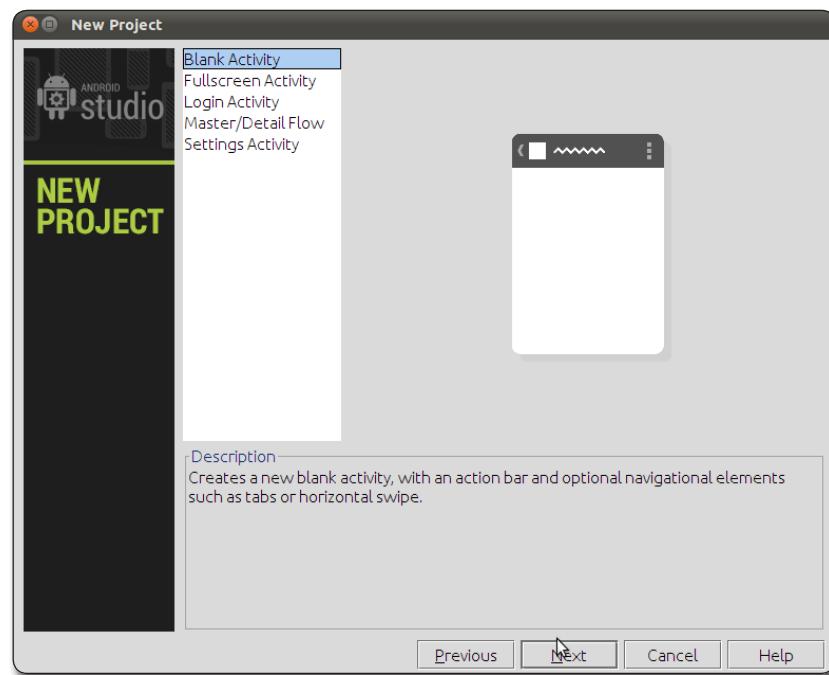


Figura 5. Janela para a escolha do tipo de activity

login padrão, com um campo de texto, um campo para digitação de senhas e um botão.

O recurso mais importante do template *Master/Detail Flow* é a criação de uma tela adaptativa, ou seja, cujos elementos e tamanho variam automaticamente de acordo com as dimensões da tela do aparelho.

O último template, *Settings Activity*, deve ser utilizado apenas para criar telas de configuração do aplicativo, que devem

Android Studio: Primeiros passos com Android

acessar um documento XML que contenha as opções configuráveis do *app*.

No exemplo demonstrado aqui, optamos por manter a nomenclatura padrão sugerida pelo Android Studio, permanecendo, portanto, o nome **MainActivity**, conforme mostra a **Figura 6**.

Nessa janela, devemos clicar em *Finish* para que a IDE inicie a criação da estrutura do projeto. Ao fim desse processo, no “Project Explorer”, você verá a estrutura exibida na **Figura 7**. Dentro do diretório *src* ficará todo o código fonte do aplicativo, tanto Java quanto XML. Além disso, as imagens utilizadas também estarão nesse diretório.

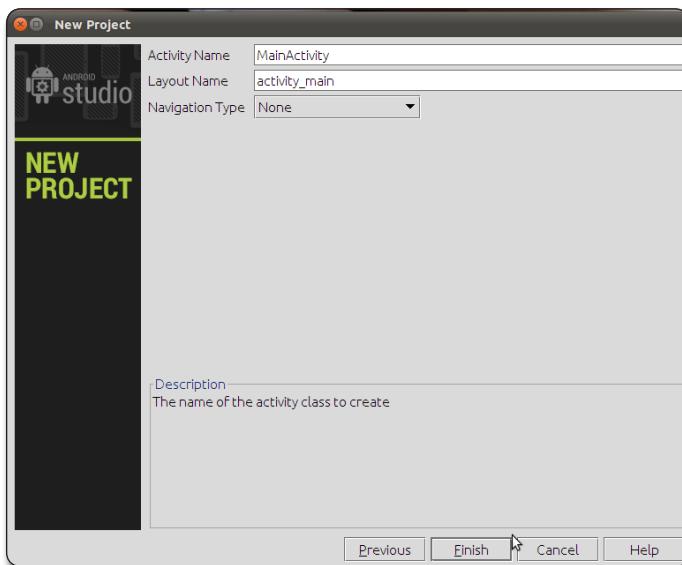


Figura 6. Última janela para a criação do projeto

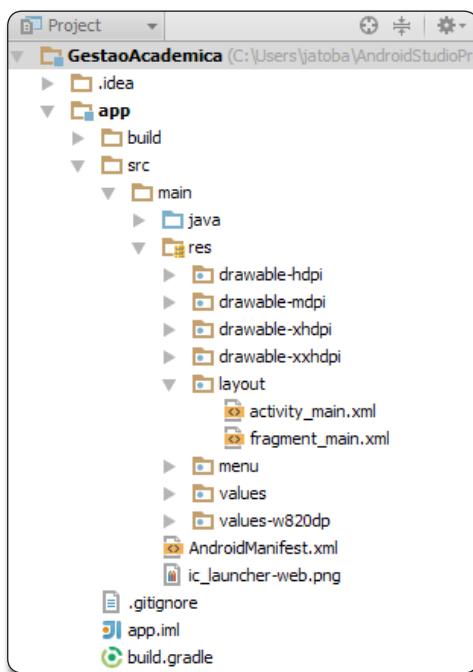


Figura 7. Estrutura completa do projeto

Uma vez com essa estrutura criada, vamos começar a editar a tela do nosso aplicativo. Deste modo, dê um duplo-clique no arquivo *activity-main.xml* para abri-lo. No editor, será exibido o código XML dessa *activity*. Em seguida, nas abas disponíveis na parte inferior do editor, selecione a opção *Design*. O editor passará então a exibir graficamente a *activity*. No lado esquerdo dessa tela, é apresentada a palheta de componentes que podem ser adicionados em nossa *activity* (ver **Figura 8**). Para adicionar componentes, basta arrastá-los para a *activity*. Além disso, na barra de ferramentas do editor, conforme indica a **Figura 8**, há a opção de alterar o aparelho para o qual o *app* será desenvolvido. Para demonstrar essa possibilidade, selecionaremos a opção “Galaxy Nexus”.

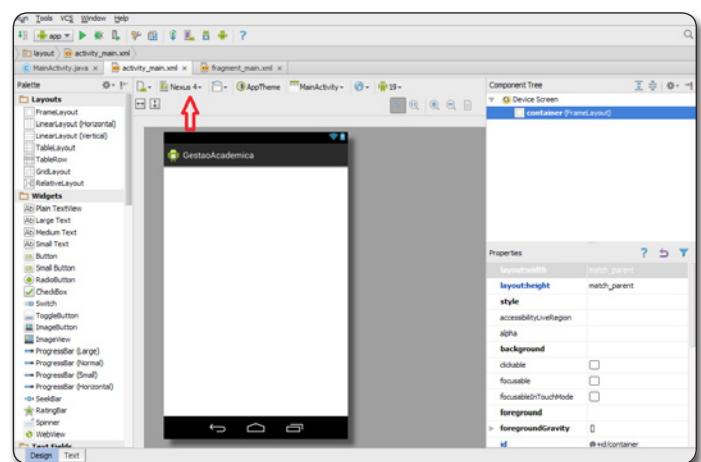


Figura 8. Editor exibindo graficamente a activity

Como o nosso objetivo é desenvolver uma aplicação para cadastro de Docentes, criaremos um formulário onde será possível digitar os dados de nome, matrícula e titulação do professor. Dito isso, podemos começar a adicionar os campos no formulário e seus respectivos títulos na *activity*. Semelhante ao que ocorre no Swing, os componentes da tela são agrupados em um container, que pode ter diversos tipos de layout que definem como esses componentes serão posicionados. No exemplo, optamos por um **LinearLayout**, sobre o qual colocaremos os títulos dos campos (componentes do tipo *TextView*) e os campos de texto (componentes do tipo *EditText*). No **LinearLayout**, os componentes são posicionados um após o outro, horizontal ou verticalmente, de acordo com a orientação do dispositivo, que é definida na propriedade **android:orientation**. Esse layout é bastante conveniente para criar formulários como o do exemplo. O código final do XML da nossa *activity* pode ser visto na **Listagem 1**.

Como informado, cada campo terá um título, ou *label*. Em Android, os valores desses campos são armazenados em um arquivo à parte; por padrão, no *strings.xml*, existente no diretório *res/values*. Para recuperar esses valores, o XML da *activity* faz uma referência a eles conforme o seguinte exemplo: **android:text="@string/nome"**, contido na **Listagem 1**.

Na **Listagem 2** pode ser visto o código do arquivo *strings.xml* e na **Figura 9** podemos ver como está o nosso formulário, com os *labels* dos campos já posicionados e com o título “Cadastro de Docentes”, como determinado no arquivo *strings.xml*.

Listagem 1. Código-fonte de *activity-main.xml*.

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/container"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="br.com.devmedia.gestaooacademica.MainActivity">

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_alignParentTop="true"
        android:orientation="vertical" >

        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textAppearance="?android:attr/textAppearanceLarge"
            android:text="@string/nome"
            android:id="@+id/textView" />

        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textAppearance="?android:attr/textAppearanceLarge"
            android:text="Matrícula:"
            android:id="@+id/textView2" />

        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textAppearance="?android:attr/textAppearanceLarge"
            android:text="Titulação:"
            android:id="@+id/textView3" />

    </LinearLayout>
```

Listagem 2. Código-fonte de *strings.xml*.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">Cadastro de Docentes</string>
    <string name="nome">Nome:</string>
    <string name="matricula">Matrícula:</string>
    <string name="titulacao">Titulação:</string>
    <string name="salvar">Salvar</string>
</resources>
```

Vamos agora colocar os campos de texto no formulário. Para isso, clique e arraste um componente do tipo *Plain Text* da paleta de componentes para o formulário e um *Edit Text* é criado. Repita esse procedimento para inserir os outros dois campos do formulário. Para ver o código XML gerado até aqui, clique na aba *Text* do editor. O conteúdo apresentado será o mesmo que está exposto na **Listagem 3**.

Para que a aparência do campo fique melhor, modificamos a propriedade *android:layout_width*, atribuindo o valor *fill_parent* para que o campo ocupe todo o espaço horizontal do display.

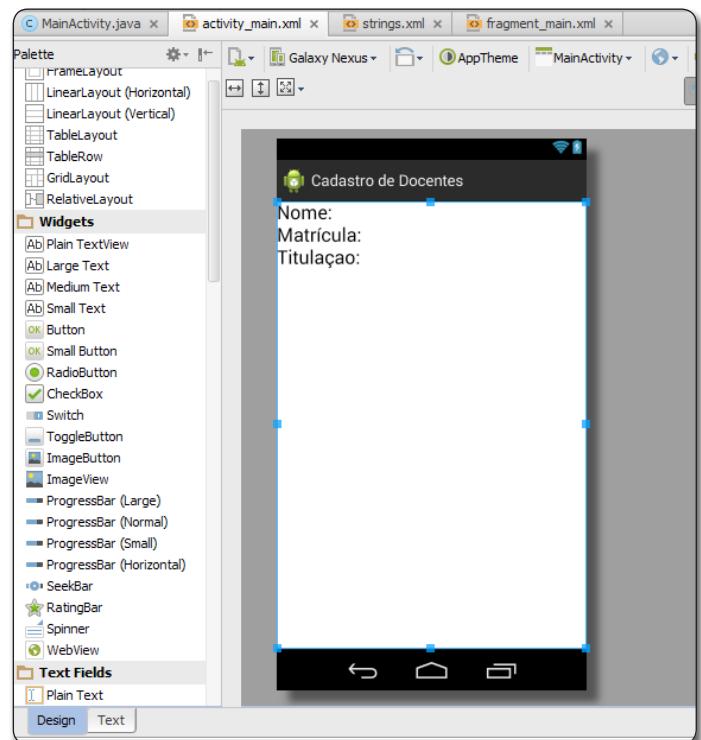


Figura 9. Inserindo os títulos dos campos

Vamos agora adicionar um botão ao formulário, com a ressalva de que queremos que ele apareça na parte inferior da tela. Para isso, ele deve ser criado fora do container *LinearLayout*. O código para inserir esse botão pode ser visto na **Listagem 4**. Repare que o texto deste também é obtido do arquivo *strings.xml*. Feito isso, o formulário está pronto e já pode ser visto na janela de *preview*, localizada no canto direito do Android Studio, como mostra a **Figura 10**.

Armazenando os dados em banco de dados no Android

Enquanto a aparência de uma *activity* é configurada em documentos XML, seu comportamento é codificado em uma classe, especificada na propriedade *tools:context*, como pode ser visto no arquivo *activity-main.xml* (vide **Listagens 1 e 2**).

Essa classe faz parte da camada de controle do nosso aplicativo, que como já informamos, é construído de acordo com o padrão MVC. O principal método da classe **MainActivity** se chama **onCreate()**. Ele é disparado quando a *activity* é instanciada, iniciando os objetos – normalmente os componentes visuais – que a *activity* possui, tornando-os visíveis.

No aplicativo exemplo, faremos com que a classe **MainActivity** seja responsável por capturar os dados do formulário, preencher objetos da classe **Docente** (vide **Listagem 5**) e acionar uma camada de persistência (vide **Listagem 6**) que salvará os dados em banco.

Para criar a classe **Docente** – no pacote **model**, já que ela será nossa camada de modelo –, no *Project Explorer*, navegue até / *app/src/main/java/br.com.devmedia.gestaooacademica* e clique com o botão direito do mouse, selecionando em seguida *New > Package*, conforme a **Figura 11**.

Android Studio: Primeiros passos com Android

Listagem 3. Código-fonte de activity-main.xml, documento XML correspondente à tela principal do app.

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/container"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="br.com.devmedia.gestaoacademica.MainActivity">

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_alignParentTop="true"
        android:orientation="vertical">

        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textAppearance="?android:attr/textAppearanceLarge"
            android:text="@string/nome"
            android:id="@+id/textView"/>

        <EditText
            android:id="@+id/nome"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:inputType="text"/>

        <EditText
            android:id="@+id/matricula"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:inputType="text"/>

        <EditText
            android:id="@+id/titulacao"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:inputType="text"/>
    
```

</LinearLayout>

```
</RelativeLayout>
```



Figura 10. App sendo visualizado na janela de preview do editor

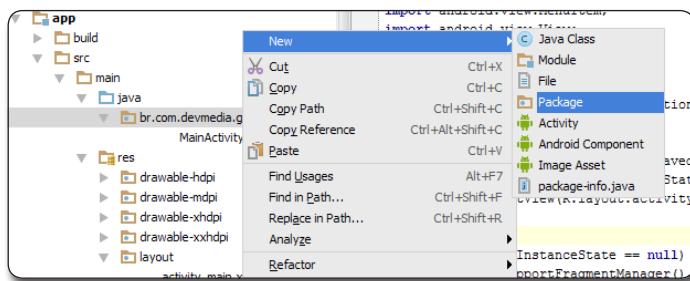


Figura 11. Criando pacote para a classe da camada de modelo

Listagem 4. Código-fonte para inserir um botão, no arquivo activity-main.xml.

```
...
</LinearLayout>
<Button
    android:id="@+id/salvar"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:layout_alignParentBottom="true"
    android:text="@string/salvar"/>
...

```

Na caixa de diálogo que aparecerá, digite como nome do pacote o valor “br.com.devmedia.gestaoacademica.model”. Em seguida, clicando com o botão direito do mouse sobre esse pacote, selecione *New > Java Class*. Como mostra a **Figura 12**, no campo *Name*, digite “Docente” e depois clique em *Finish*. Assim, a classe **Docente** será criada. Vamos então modificar seu código para inserir os devidos atributos, *setters* e *getters*, para que seu código fonte fique semelhante ao apresentado na **Listagem 5**.

Construída a classe da camada de modelo, vamos agora implementar o código da classe responsável pela persistência dos dados. Essa classe, que chamaremos de **Persistencia**, ficará num pacote que chamaremos de **db**, criado da mesma maneira que o pacote **model**.

Após criar o pacote e a classe, abra o código desta e faça com que ela herde **android.database.sqlite.SQLiteOpenHelper**. Essa classe fornece os métodos para as operações essenciais com o banco de dados. Não exploraremos todos eles, mas no exemplo, vamos escrever o método **onCreate()**, que será responsável por

criar a tabela *TB_DOCENTES* e seus campos (vide **Listagem 6**), onde os dados do formulário serão armazenados. O método **onUpgrade()** também deve ser implementado, mas não o utilizaremos no exemplo demonstrado nesse artigo.

Listagem 5. Código fonte da classe Docente.

```
package br.com.devmedia.gestaoacademica.model;

public class Docente {

    private String nome;
    private String matricula;
    private String titulacao;

    public String getNome() {
        return nome;
    }

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

    public String getMatricula() {
        return matricula;
    }

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

    public String getTitulacao() {
        return titulacao;
    }

    public void setTitulacao(String titulacao) {
        this.titulacao = titulacao;
    }
}
```

Listagem 6. Código fonte da classe Persistencia.

```
public class Persistencia extends SQLiteOpenHelper{

    public Persistencia(Context context, String name, int version) {
        super(context, name, null, version);
    }

    @Override
    public void onCreate(SQLiteDatabase sqld) {
        sqld.execSQL("CREATE TABLE TB_DOCENTES (" +
                    "id_docentes INTEGER PRIMARY KEY autoincrement," +
                    "+ nome varchar(50) NOT NULL," +
                    "+ matricula varchar(11) NOT NULL," +
                    "+ titulacao varchar(50) NOT NULL" +");");
    }

    @Override
    public void onUpgrade(SQLiteDatabase sqld, int i, int i1) {
    }
}
```

Para que os dados digitados no formulário sejam salvos quando o usuário clicar no botão *Salvar*, faremos com que a classe **MainActivity** implemente a interface **OnClickListener**, nos forçando a escrever o método **onClick()**. Esse método terá como função

preencher os atributos de um objeto do tipo **Docente** com os valores digitados no formulário e salvá-lo no banco, como mostra a **Listagem 7**. No método **onCreate()** é instanciado um objeto do tipo **Button**, cuja correspondência com o botão *Salvar* da interface do aplicativo é feita pelo código **findViewById(R.id.salvar)**.

Ainda no método **onCreate()**, é atribuída a escuta do evento “clique” – ou *listener* – ao botão *Salvar*, através do código **salvar.setOnClickListener(this)**.

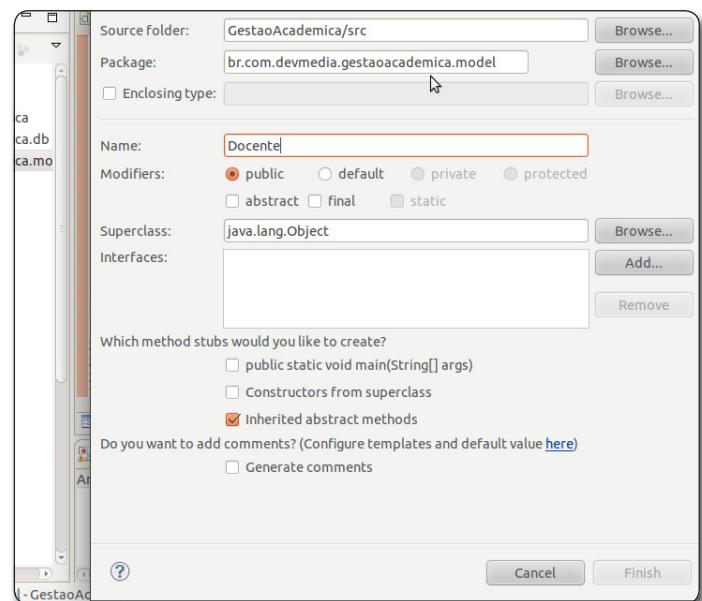


Figura 12. Criando a classe Docente

Como já informado, a classe **R** faz a ligação entre as camadas de modelo e controle. Por meio de seu membro estático **id**, é possível acessar os componentes da camada de visão do aplicativo definidos em *main-activity.xml*, como pode ser visto no método **onClick()**, no trecho de código **findViewById(R.id.nome)**, que dá acesso ao campo **nome** do formulário. É deste modo que recuperamos os campos que receberam as informações do usuário. E com os valores desses campos, preenchemos o objeto **Docente**.

Finalmente, implementamos o código responsável por efetivar a inserção no banco de dados. Observe as últimas cinco linhas do método **onClick()**, mais precisamente, **bd.getWritableDatabase().insert("TB_DOCENTES", null, contentValues)**. Nestas linhas, o objeto **contentValues** recebe os dados do objeto **docente** e depois o método **insert()** é chamado, persistindo os dados no banco.

Com isso o aplicativo está pronto e já pode ser executado no emulador, como veremos na próxima seção.

Executando o aplicativo no emulador

Para testar o aplicativo, poderíamos colocá-lo para rodar em um dispositivo móvel real ou utilizar um emulador. Nesse artigo, optamos por utilizar um emulador para verificar o funcionamento do *app*.

Android Studio: Primeiros passos com Android

Para fazer isso, é necessário configurar no Android Studio um dispositivo virtual. Isso é feito através do *Android Virtual Device Manager* (AVD Manager). Para iniciá-lo, no menu superior, selecione *Tools > AVD Manager* (ver **Figura 13**).

Embora seja possível criar nossas próprias definições de aparelhos (celulares, tablets, etc.), utilizaremos uma dentre as muitas definições já disponíveis. Para isso, clique na aba *Device Definitions* e escolha um dispositivo – no nosso caso, *Galaxy Nexus*. Em seguida, clique em *Create AVD*, que abrirá uma janela para modificar as propriedades do dispositivo virtual.

Listagem 7. Implementando a camada de controle na classe *MainActivity*.

```
...
public class MainActivity extends Activity implements View.OnClickListener{

    private Docente docente = new Docente();

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        Button salvar = (Button) findViewById(R.id.salvar);
        salvar.setOnClickListener(this);
    }

    @Override
    public void onClick(View v) {
        final Context context = this;

        EditText nome = (EditText) findViewById(R.id.nome);
        EditText matricula = (EditText) findViewById(R.id.matricula);
        EditText titulacao = (EditText) findViewById(R.id.titulacao);

        docente.setNome(nome.getText().toString());
        docente.setMatricula(matricula.getText().toString());
        docente.setTitulacao(titulacao.getText().toString());

        Persistencia bd = new Persistencia(this, "ga", 1);

        ContentValues contentValues = new ContentValues();
        contentValues.put("nome", docente.getNome());
        contentValues.put("matricula", docente.getMatricula());
        contentValues.put("titulacao", docente.getTitulacao());

        bd.getWritableDatabase().insert("TB_DOCENTES", null, contentValues);
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        getMenuInflater().inflate(R.menu.main, menu);
        return true;
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        int id = item.getItemId();
        if (id == R.id.action_settings) {
            return true;
        }
        return super.onOptionsItemSelected(item);
    }
}
```

Nesta janela, apenas clique em *OK* e o dispositivo virtual será adicionado à lista de dispositivos exibida na **Figura 13**. Assim, basta selecioná-lo e clicar no botão *Start*. Nesse momento o emulador iniciará a carga do sistema operacional exatamente como se fosse um dispositivo real, como mostra a figura **Figura 14**.

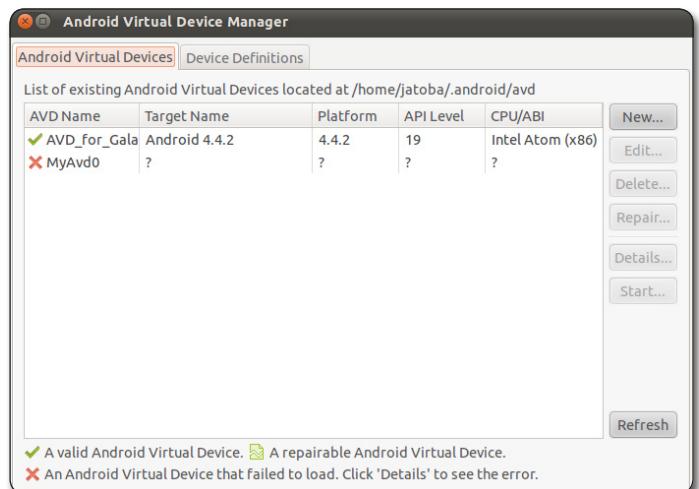


Figura 13. Configurando emulador no AVD Manager

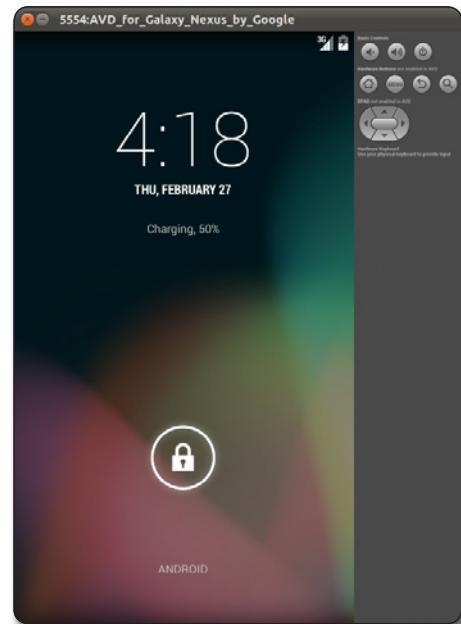


Figura 14. Emulador carregando o Android

Uma vez que o emulador está funcional e o dispositivo virtual configurado, para colocar o aplicativo para rodar basta clicar no botão *Play*, na barra de menu do Android Studio, como mostra a **Figura 15**.

Feito isso, o Android Studio recompilará o projeto e exibirá uma janela para que seja selecionado o dispositivo virtual em que o aplicativo deve ser executado. Caso o emulador não esteja aberto, marque a opção *Launch emulator* e selecione o dispositivo que foi criado, em seguida clicando em *OK*, conforme a **Figura 16**.

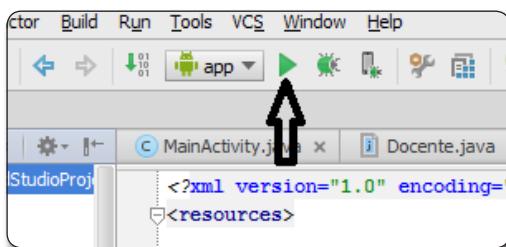


Figura 15. Ícone para iniciar a execução do aplicativo

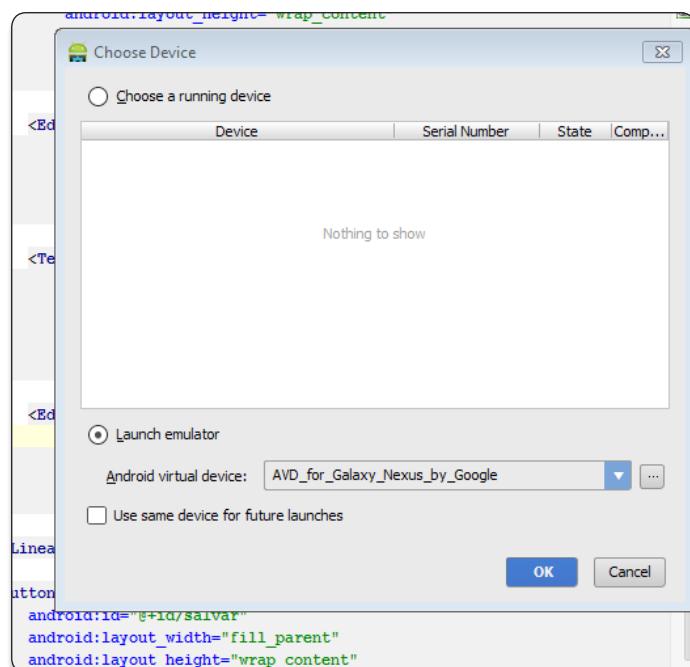


Figura 16. Selecionando o dispositivo virtual para a execução do aplicativo

Com o emulador aberto, exibindo a tela inicial do Android, clique no ícone do aplicativo (*Cadastro de Docentes*) para abrir o app (veja a **Figura 17**).

Com o aplicativo aberto, digite os valores nos campos e clique no botão *Salvar* para que os dados sejam armazenados no banco de dados. A **Figura 18** mostra o aplicativo sendo utilizado no emulador.

Por ter o objetivo de demonstrar os primeiros passos para o desenvolvimento nesse ecossistema, muitos dos recursos disponíveis para a plataforma Android não foram explorados neste artigo, cabendo, portanto, aprofundamentos diversos, como por exemplo, o uso de outros componentes visuais, o estudo de recursos que viabilizem a construção de aplicativos multimídia e na nuvem, entre tantos outros.

Em um mercado bilionário como o de aplicativos para dispositivos móveis, certamente oportunidades surgirão e, da forma como as relações entre as pessoas e as tecnologias vêm se desenvolvendo, apostar nesse caminho pode ser promissor.

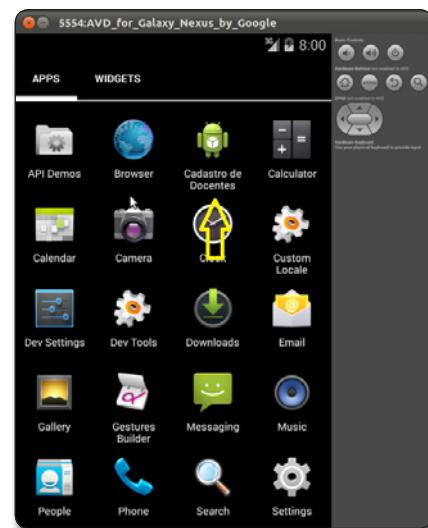


Figura 17. Tela inicial do Android no emulador. Em destaque, o ícone do aplicativo exemplo

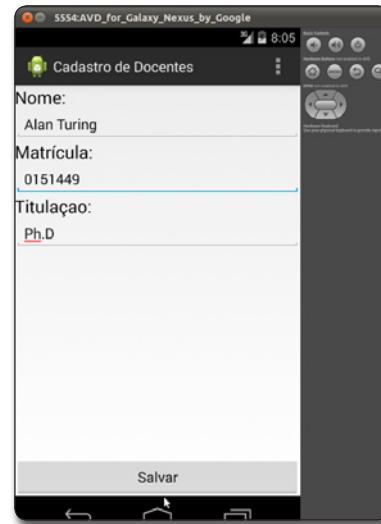


Figura 18. Aplicativo aberto no emulador

Autor



Alessandro Jatobá

jatoba@jatoba.org

É Mestre em Ciência da Computação pela UFRJ, Doutorando em Engenharia também pela UFRJ e, durante 2014 e 2015, fará um Doutorado-Sanduíche pelo Departamento de Engenharia e Projeto de Sistemas da Universidade de Waterloo, no Canadá. Trabalha com Java há mais de 15 anos, a maior parte deles liderando equipes de desenvolvedores em projetos Web. Também é professor universitário lecionando temas ligados ao desenvolvimento orientado a objetos.



Links:

Site oficial do Java.

<http://www.java.com>

Site oficial de desenvolvedores para a plataforma Android.

<http://developer.android.com>

Introdução ao Java Garbage Collection

Entendendo o gerenciamento automático de memória da JVM

Na introdução do documento “The Java Language Specification” – que descreve a especificação completa da sintaxe e semântica do Java –, os autores afirmam que Java é uma linguagem de programação baseada em C/C++, mas que é organizada de maneira bastante diferente. A especificação da linguagem reforça que na construção do Java foram omitidos alguns aspectos de C/C++ e incluídas algumas ideias de outras linguagens. Por exemplo, um dos recursos incluídos em Java é o gerenciamento automático de memória, o qual utiliza um Coletor de Lixo (*Garbage Collector*) que tem a finalidade de evitar os problemas de desalocação explícita dos espaços ocupados por objetos não mais referenciados. Esse gerenciamento explícito é efetivado com a função `free` em C ou com o operador `delete` em C++. A técnica de Coleta de Lixo consiste na recuperação segura do espaço de memória ocupado por um objeto que não é mais referenciado dentro de uma aplicação. Esta técnica é o tema que será estudado nesta matéria. No entanto, antes de iniciarmos essa discussão, julgamos relevante expor quais são as vantagens desse gerenciamento automático.

Denomina-se gerenciamento de memória em linguagens de programação orientadas a objetos, o processo de reconhecimento quando objetos alocados não são mais referenciados e a posterior liberação da memória usada por tais objetos, tornando-a assim disponível para futuras alocações. Na **Listagem 1** mostramos um exemplo de um objeto alocado que deixa de ser referenciado.

Listagem 1. Exemplo de um objeto não mais referenciado.

```
Integer i = new Integer(10);
i = null;
```

Note na **Listagem 1**, que a variável `i` inicialmente refere-se a um `Integer` e em seguida a mesma referência recebe `null`, deixando então de apontar para o objeto. Ou

Fique por dentro

Esta matéria traz uma introdução aos conceitos e à maneira como funciona a coleta de lixo na Máquina Virtual Java (JVM). Nela destacamos quais os problemas que podem ocorrer quando o gerenciamento de memória é feito explicitamente pelo programador, tal como em C++, e como o Garbage Collector - recurso para gerenciamento automático de memória - soluciona essas dificuldades. O texto trata de maneira simplificada como funciona um Coletor de Lixo através de um modelo de algoritmo utilizado apenas para facilitar o entendimento, pois não é claro como realmente ele opera na JVM.

O estudo de Garbage Collection é importante para os desenvolvedores que desejam entender o gerenciamento de memória na JVM e que se preocupam em escrever aplicações que tenham um bom desempenho e sejam escaláveis.

seja, nesse momento o `Integer` criado anteriormente não está mais sendo referenciado, permitindo assim que a memória ocupada por ele seja liberada.

Em algumas linguagens, tais como C e C++, como visto antes, a tarefa de gerenciamento de memória – a qual chamamos de gerenciamento explícito – é de responsabilidade do programador. Nessas linguagens onde o gerenciamento de memória é explícito, podem ocorrer erros, resultados não previstos ou até mesmo o fechamento inesperado (*crash*) da aplicação.

Um dos problemas que ocorrem quando o gerenciamento de memória é feito pelo programador é o de referências pendentes (*dangling references*). Esta falha acontece se o programador desaloca o espaço ocupado por um objeto que ainda está sendo referenciado.

Outro problema com o gerenciamento explícito é o vazamento de memória (*space leaks*). O vazamento ocorre quando um objeto deixa de ser referenciado e sua memória não é liberada. Caso aconteça muito vazamento, o aumento de consumo do espaço de armazenamento pode levar ao esgotamento da memória.

Uma alternativa ao gerenciamento explícito é o gerenciamento automático feito por um programa denominado *Garbage Collector*.

Este mecanismo é empregado em linguagens orientadas a objetos mais modernas, tais como o Java.

Com o uso do *Garbage Collector* o problema de referências pendentes nunca acontece, pois um objeto que ainda está sendo referenciado jamais será candidato – ou elegível – à coleta de lixo e seu espaço de memória não ficará livre. Esta maneira de gerenciar o espaço também resolve o problema do vazamento de memória, visto que toda memória não mais referenciada é liberada automaticamente.

Mas antes de começarmos a estudar o funcionamento do *Garbage Collector*, precisamos aprender onde as variáveis e objetos são alocados em Java. É sobre isso que discutiremos na primeira seção desta matéria.

A Pilha (Stack) e o Heap

Em Java o programador precisa se preocupar com duas áreas de memória – a área onde os objetos são armazenados (*heap*), e a área onde vivem as variáveis locais e as chamadas de métodos (*pilha*).

As variáveis locais podem ser primitivas ou referências a objetos e são conhecidas como variáveis de pilha, o que reforça o que afirmamos anteriormente – tais variáveis são armazenadas na pilha. As variáveis primitivas possuem um tipo e tamanho, os quais determinam o espaço de memória ocupado pelo valor da variável. Por exemplo, se declararmos `int a = 10`, isto indica que o valor 10 será armazenado na pilha como um número inteiro de 32 bits. Por sua vez, as referências não armazenam o objeto propriamente dito, mas uma informação – algo como um endereço ou ponteiro – sobre onde localizar o objeto.

Vejamos então o que acontece quando definimos, por exemplo, `Pessoa p = new Pessoa()` em um programa:

1. `Pessoa p` diz à Máquina Virtual Java (JVM) para alocar espaço com esse nome para uma variável de referência na pilha;
2. `new Pessoa()` diz à JVM para alocar espaço para um novo objeto `Pessoa` no *heap*;
3. Finalmente ocorre a ligação do objeto à variável de referência, através do operador de atribuição (=).

Na **Listagem 2** mostramos o código de uma classe, cuja representação de sua execução na pilha e no *heap* podem ser vistos na **Figura 1**.

Observando a figura notamos que, quando a classe é executada, inicialmente é colocado no topo da pilha o método `main()`, assim como as variáveis locais e/ou referências declaradas dentro dele. Neste caso, `r` é uma referência, a qual irá manter a informação de como localizar o objeto **Retangulo** criado no *heap*. Em seguida, cada método que é invocado – `setBase()` e `setAltura()`, nessa ordem – vai sendo empilhado com as variáveis locais correspondentes.

Além dessas, existem em Java as variáveis de instância. Tais variáveis são declaradas dentro de uma classe, mas não dentro de um método. Elas são, na verdade, os campos ou atributos de um objeto. As variáveis de instância vivem dentro do objeto ao qual elas pertencem. Na **Listagem 3** ilustramos um exemplo.

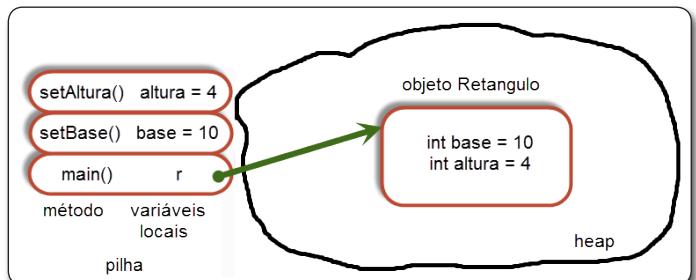


Figura 1. Representação das variáveis na pilha e no heap quando a classe **Retangulo** é executada

Listagem 2. Classe para ilustrar a representação de variáveis na pilha e no heap.

```
public class Retangulo {
    private double base;
    private double altura;

    public double getBase() {
        return base;
    }

    public void setBase(double base) {
        this.base = base;
    }

    public double getAltura() {
        return altura;
    }

    public void setAltura(double altura) {
        this.altura = altura;
    }

    public static void main(String[] args) {
        Retangulo r = new Retangulo();
        r.setBase(10);
        r.setAltura(4);
    }
}
```

Listagem 3. Exemplo de variável de instância.

```
public class Pessoa {
    // variáveis de instância "nome" e "idade"
    String nome;
    int idade;
}
```

Note na **Listagem 3** que foram definidas duas variáveis de instância: uma que é primitiva e outra que é um objeto. As variáveis de instância primitivas vivem dentro do objeto, assim como seus valores. Por sua vez, as variáveis de instância que são referências também são alocadas dentro do objeto, mas os objetos referenciados por elas são alocados no *heap*, conforme podemos ver com o auxílio da **Figura 2**, onde é visualizada uma instância da classe **Pessoa**.

Observe na **Figura 2**, onde representamos um objeto **Pessoa** e o objeto **String** correspondente ao nome da pessoa. Este exemplo ilustra o que foi afirmado anteriormente. A variável de instância `nome` é alocada dentro do objeto **Pessoa** – residente no *heap* – e o objeto **String** que mantém o nome da pessoa também reside no *heap*.

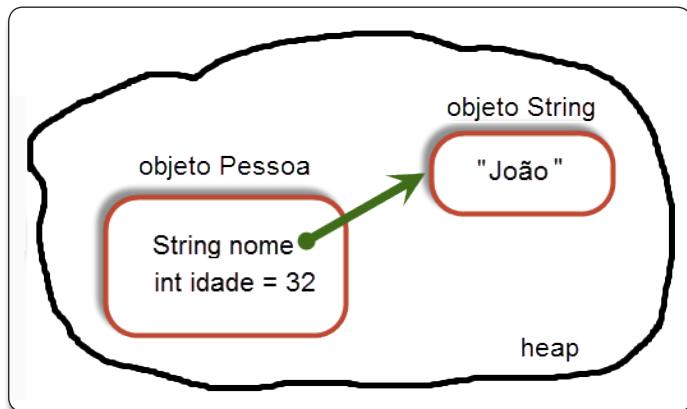


Figura 2. Alocação de variáveis de instância no heap

Visto que estamos falando sobre o uso da memória, julgamos importante destacar um mecanismo utilizado pela JVM para tornar o uso deste recurso mais eficiente, o “pool de constantes String”. Neste pool ficam armazenados todos os literais String. Literais são constantes que podem ser expressas no código da aplicação, tal como “abcd” na Listagem 4. Quando o compilador encontra um literal String – como na primeira linha da Listagem 4 –, ele verifica se existe uma String idêntica no pool. Se existir, a nova referência é direcionada à String existente e não é criado um novo objeto. Mas, quando a String é criada usando new, como na segunda linha da Listagem 4, este novo objeto é criado no heap, e o literal é inserido no pool de constantes String, se não existir um literal idêntico.

Listagem 4. Criação de Strings e o uso do pool de constantes String.

```
String a = "abcd";
String b = new String("abcd");
```

Após essa visão geral sobre o uso da memória, passaremos então a falar sobre o Garbage Collector.

Garbage Collector

Um Garbage Collector tem a função de:

- Alocar memória;
- Assegurar que quaisquer objetos referenciados permaneçam na memória;
- Recuperar a memória alocada pelos objetos que não são mais alcançáveis pelas referências no código em execução.

Objetos referenciados são denominados *vivos*, enquanto que são considerados *mortos* – ou também chamados de *lixo* – os objetos que não são mais referenciados por qualquer *thread* em execução. Entende-se por *thread* como sendo uma linha de execução – um processo – de uma aplicação Java. Toda aplicação Java possui no mínimo uma thread – o método **main()**. Este método, necessário para rodar uma classe na JVM, executa em uma *thread* e é chamado de *thread main*.

Depois dessa breve explicação sobre *threads*, vamos prosseguir com a explicação sobre *lixo*. Diz-se que um objeto X referenciado por x torna-se *lixo* quando x passa a se referir a outro objeto ou a null, ou ainda, se x for uma variável local e o programa retorna do método onde x é declarada. São esses objetos, denominados *lixo*, que precisam ser localizados e ter seu espaço de memória desalocado no processo denominado *Garbage Collection*.

O controle do Coletor de Lixo é feito pela JVM, que é quem decide quando ele será executado. Mas não existe nada que garanta quando o *Garbage Collector* será executado, ainda que seja possível ao programador solicitar via código Java que a JVM o execute. Em geral, quando não há interferência do programador, a máquina virtual executa o Coletor de Lixo quando percebe que a memória está ficando sem espaço. De qualquer maneira, pode ser que uma aplicação encerre sua execução sem que o *Garbage Collector* seja executado uma única vez.

Dessa maneira, pelo que já foi exposto, quando um objeto deixa de ser referenciado, o espaço de memória ocupado por ele não é imediatamente desalocado. O que se diz de um *lixo* é que ele – o objeto – se torna *qualificado* para a coleta de lixo, podendo então ser desalocado na próxima execução do GC (*Garbage Collector*).

A fim de que o GC execute suas funções adequadamente, é necessário que ele possua algumas características. Por exemplo, o GC deve ser seguro e compreensivo. Seguro significa que objetos vivos nunca devem ter seu espaço liberado, e para que o GC seja compreensivo, os objetos não mais referenciados não deveriam permanecer sem ser coletados por mais que um pequeno número de ciclos de coleta. Entende-se por ciclo de coleta o processo que inicia com a marcação dos objetos elegíveis e vai até suas exclusões definitivas.

Também é desejável que um GC opere com eficiência, sem introduzir longas pausas, durante as quais a aplicação irá parar de executar. No entanto, como acontece com a maioria dos sistemas computacionais, muitas vezes existem conflitos entre tempo, espaço e frequência. Por exemplo, se o tamanho do heap é pequeno, a coleta vai ser rápida, mas a pilha vai encher mais rapidamente, exigindo, portanto, coletas mais frequentes. Por outro lado, uma pilha de tamanho maior vai demorar mais tempo a encher e assim, as coletas serão menos frequentes, mas podem demorar mais tempo.

Outra característica importante é a limitação da fragmentação. A fragmentação ocorre quando a memória é liberada e o espaço livre pode aparecer em pequenos pedaços em diversas áreas, tais que pode não haver espaço suficiente em uma área contígua qualquer para ser usada para alocação de um determinado objeto. Uma abordagem para eliminar a fragmentação é chamada de compactação.

Devido ao quesito desempenho, uma característica importante é a escalabilidade em aplicações multithread, pois a alocação de espaço de memória não deve se tornar um gargalo para o aumento do número de tarefas concorrentes sendo executadas. Isso também é válido para aplicações em sistemas com múltiplos processadores. Nas aplicações multithread e em hardware com

múltiplos processadores, pressupõe-se a execução de várias tarefas concorrentemente. Nesses casos, escalabilidade significa aumento de desempenho à medida que se aumenta o número de linhas de execução (*threads*). Posto que cada tarefa ou linha de execução necessita alocar/desalocar espaço para seus objetos, o *Garbage Collector* não deve ter influência na queda de performance da aplicação.

Outra questão a respeito do GC é que não se sabe ao certo como ele funciona, visto que na própria especificação da linguagem isto não é descrito. Mas, para que seja possível entender melhor como se dá a coleta de lixo, é necessário que seja utilizado no estudo um modelo simples de algoritmo. Sendo assim, nos próximos parágrafos abordaremos dois desses algoritmos, os quais podem ser empregados para a compreensão do funcionamento do GC.

O processo de coleta de lixo é basicamente dividido em duas fases. A primeira é a separação entre objetos *vivos* e objetos *mortos* – já conceituados anteriormente nesta matéria. E a segunda fase é a liberação da memória ocupada pelos *mortos*.

Um modelo de coleta de lixo normalmente apresentado na literatura é a contagem de referências. Nesse modelo, quando um objeto X referencia outro objeto Y, o sistema incrementa um contador de referências em Y, e quando X deixa de referenciar Y, o contador é decrementado. Quando o contador chega a zero, Y não está mais *vivo*, e torna-se qualificado para a coleta de lixo. Dessa maneira, os contadores dos objetos aos quais Y eventualmente se refere também serão decrementados.

No entanto, este modelo não funciona quando X e Y referenciam-se mutuamente. Nesta situação nenhum dos contadores de referência se tornará zero, e nenhum dos dois objetos poderá ser coletado. E, portanto, também não serão coletados nenhum dos demais objetos que porventura sejam referenciados por X e Y, direta ou indiretamente. Este é um forte motivo para que muitos coletores de lixo não empreguem esta abordagem.

Um modelo também simples e que não apresenta o problema citado anteriormente é o chamado *marcar-e-varrer*. Este nome é usado devido à maneira que as duas fases de coleta de lixo são implementadas. Para encontrar objetos *vivos*, o Coletor de Lixo determina inicialmente um conjunto de *raízes* que contém aqueles objetos que são referenciados diretamente, tais como aqueles referenciados por variáveis locais.

Depois disso, o *Garbage Collector* irá marcar os objetos que são referenciados – alcançáveis – por essas *raízes*. Após essa etapa, o GC irá examinar as referências em cada um desses objetos. Caso um objeto aqui referenciado já tenha sido marcado no passo anterior, ele é ignorado. Caso contrário, o objeto é marcado como alcançável e suas referências são examinadas. Esse processo continua até que objetos não mais referenciados permaneçam desmarcados. Após concluir essa fase, o Coletor de Lixo pode reclamar os objetos não marcados, varrendo-os da memória.

Durante o processo de *marcar-e-varrer*, qualquer modificação na interconexão dos objetos irá interferir na coleta de lixo. Por exemplo, se um objeto não referenciado passar a ser alcançável enquanto acontece a execução da marcação, tal objeto poderá ser

coletado indevidamente. Por esse motivo, a execução do programa deverá ser pausada durante o processo de marcação.

É válido destacar que o algoritmo *marcar-e-varrer* é apenas um modelo mental para que possamos entender o funcionamento do Coletor de Lixo. As máquinas virtuais utilizam estratégias muito mais complexas para realizar este processo.

Tornando objetos explicitamente elegíveis à coleta de lixo

Com a finalidade de reforçar o entendimento do que foi discutido nas seções anteriores, vejamos alguns exemplos de código que tornam objetos elegíveis à coleta de lixo.

O primeiro exemplo, e o mais simples, consiste em atribuir `null` à variável de referência, conforme podemos ver na [Listagem 5](#).

Outra maneira é fazer uma variável de referência a um objeto A, “apontar” para outro objeto B, como na [Listagem 6](#).

Listagem 5. Anulando uma referência para tornar o objeto elegível ao gc.

```
public class TesteColetor {  
  
    public static void main(String[] args) {  
        StringBuffer sb = new StringBuffer("abcd");  
        // O StringBuffer é alcançável pela referência sb  
        // e não é elegível para coleta de lixo  
        System.out.println(sb);  
        sb = null;  
        // Agora o StringBuffer deixa de ser referenciado  
        // por sb e torna-se candidato à coleta de lixo  
    }  
  
}
```

Listagem 6. Tornando um objeto elegível ao gc por meio da reatribuição de uma referência.

```
public class TesteColetor {  
  
    public static void main(String[] args) {  
        StringBuffer sb1 = new StringBuffer("abcd");  
        StringBuffer sb2 = new StringBuffer("xyz");  
        // Os StringBuffer estão sendo referenciados  
        // e não são elegíveis para coleta de lixo  
        System.out.println(sb1);  
        sb1 = sb2;  
        // Agora o StringBuffer "abcd" deixa de ser referenciado  
        // por sb1 e torna-se candidato à coleta de lixo  
        // O StringBuffer "xyz" por outro lado, está sendo referenciado  
        // por duas variáveis de referência  
    }  
  
}
```

Objetos que são instanciados dentro de métodos também precisam ser avaliados para verificar se são elegíveis, pois como sabemos, as variáveis locais declaradas em um método – sejam primitivas ou de referência – existem apenas durante a sua execução. Dessa maneira, objetos criados e referenciados por variáveis locais, tornam-se elegíveis para a coleta de lixo após o retorno da execução do método. A exceção ocorre quando um objeto é retornado pelo método e atribuído a uma variável de referência no método que chamou, conforme o exemplo da [Listagem 7](#).

Introdução ao Java Garbage Collection

No código da **Listagem 7**, o método `getDate()` cria dois objetos – um **Date** e um **StringBuffer**. Visto que `getDate()` retorna o objeto **Date**, o qual é atribuído a `d` no método `main()`, então tal objeto não se torna candidato ao Coletor de Lixo. Entretanto, o **StringBuffer** torna-se elegível, pois a referência `now` deixa de existir fora de `getDate()`.

No livro Sun Certified Programmer for Java 6 – Study Guide, Kathy Sierra e Bert Bates referem-se ao conceito de *ilha de objetos* ou *ilha de isolação*. Este é um caso onde objetos são elegíveis mesmo que ainda tenham referências válidas. Um exemplo citado pelos autores é quando uma classe tem uma variável de instância que é referência para outra instância da mesma classe, tal como na classe **Ilha**, mostrada na **Listagem 8**.

Listagem 7. Exemplo de objeto retornado por um método que não se torna elegível para o gc.

```
import java.util.Date;
public class TesteColetor {

    public static void main(String[] args) {
        Date d = getDate();
        System.out.println("d = " + d);
    }

    public static Date getDate() {
        Date d1 = new Date();
        StringBuffer now = new StringBuffer(d1.toString());
        System.out.println(now);
        return d1;
    }
}
```

Listagem 8. Exemplo de ilha de objetos.

```
public class Ilha {
    Ilha i;

    public static void main(String[] args) {
        Ilha i1 = new Ilha();
        Ilha i2 = new Ilha();
        Ilha i3 = new Ilha();

        i1.i = i2;
        i2.i = i3;
        i3.i = i1;

        i1 = null;
        i2 = null;
        i3 = null;
    }
}
```

Note no exemplo da **Listagem 8** que os objetos criados passam a referir-se mutuamente e depois suas referências são anuladas. Assim, mesmo que os objetos possuam referências válidas – as variáveis de instância –, eles não podem mais ser alcançados por uma *thread* em execução, e tornam-se portanto elegíveis para a coleta de lixo. Felizmente o *Garbage Collector* é capaz de descobrir essas ilhas de objetos e então removê-las.

Interagindo com o Garbage Collector

Não existe uma maneira explícita de liberar a memória ocupada por objetos não mais referenciados, porém pode-se invocar diretamente o *Garbage Collector* para procurar objetos inalcançáveis. A classe **Runtime**, juntamente com alguns métodos da classe **System**, permite que o Coletor de Lixo seja chamado, que finalizadores pendentes sejam executados ou que se consulte o estado atual da memória. Mais adiante, dedicaremos uma seção ao método `finalize()` – finalizador – que é executado pelo Coletor de Lixo depois que ele determina que um objeto não é mais alcançável, tornando-se assim elegível à coleta de lixo.

Os métodos que Java oferece para interagir com o Coletor de Lixo são membros de **Runtime**, a qual é uma classe especial que tem uma única instância para cada programa. Para obter a instância de **Runtime**, pode-se usar o método `Runtime.getRuntime()`. Após esta instância ter sido definida, pode-se invocar o *Garbage Collector* utilizando o método `gc()`. Ao executar esse método, a coleta de lixo deveria ser efetivada e o programa teria mais memória livre disponível. No entanto, não existe garantia de que o método fará realmente isso. Primeiro, a especificação de Java permite que ele não faça nada. Segundo, outra *thread* – linha de execução – pode ter requisitado muita memória imediatamente após a chamada a `gc()`. Dessa maneira, mesmo que `gc()` tenha executado, o efeito de liberar memória não será obtido, pois a requisição da outra *thread* irá mascarar a liberação de espaço feita pelo *Garbage Collector*. Um terceiro motivo é que pode não haver objetos a serem coletados.

Com o objetivo de tentar ver o efeito do Coletor de Lixo, tente executar o programa da **Listagem 9** – adaptado do livro Sun Certified Programmer for Java 6 – Study Guide. Neste programa podemos visualizar a memória disponível para a JVM, a memória livre antes da criação e depois da criação de 100.000 objetos do tipo **Date**. Em seguida o método `gc()` é chamado e novamente fazemos a consulta à memória livre.

Listagem 9. Programa para interagir com o Garbage Collector.

```
import java.util.Date;

public class VerificaGC {

    public static void main(String[] args) {
        Runtime rt = Runtime.getRuntime();
        System.out.println("Memória total da JVM: " + rt.totalMemory());
        System.out.println("Memória antes da criação dos objetos: " + rt.freeMemory());
        Date d = null;
        for (int i = 0; i < 100000; i++) {
            d = new Date();
            d = null;
        }
        System.out.println("Memória depois da criação dos objetos: " + rt.freeMemory());
        rt.gc();
        System.out.println("Memória depois executar o gc: " + rt.freeMemory());
    }
}
```

Após a execução do programa, ele retornou as seguintes informações no console:

Memória total da JVM: 61210624

Memória antes da criação dos objetos: 60570816

Memória depois da criação dos objetos: 58208432

Memória depois de executar o gc: 60630904

Como podemos constatar pelos dados retornados, o Coletor de Lixo realmente deletou os objetos, visto que a memória livre aumentou após a execução do método `gc()`. Note, porém, que o comportamento de `gc()` pode ser diferente em outras máquinas virtuais, sem garantia de que os objetos elegíveis serão realmente removidos da memória.

Para finalizar esta seção, destacamos que o método `gc()` também pode ser chamado na classe `System`, a qual possui alguns métodos estáticos que criam uma instância de `Runtime`. Assim, pode-se invocar o Coletor de Lixo simplesmente chamando `System.gc()`.

O método `finalize()`

Este método foi idealizado para ser chamado pelo *Garbage Collector* em um objeto que é elegível para a coleta de lixo. Ele é declarado originalmente na classe `Object` e, portanto, herdado por todas as classes Java. Por esse motivo, essas classes devem sobrescrever

este método porque, semelhante aos métodos `toString()`, `equals()` e `hashCode()`, não existe uma implementação de `finalize()` em `Object`. Nessas implementações de `finalize()` – feitas nas subclasses – devem ser liberados quaisquer recursos utilizados pelo objeto que não sejam memória – visto que a memória é liberada pelo próprio Coletor de Lixo.

O contrato – recomendações da linguagem Java – de `finalize()` informa que ele só deve ser executado se e quando a JVM determinar que não existe mais qualquer meio pelo qual o objeto seja referenciado por qualquer *thread*. A exceção ocorre por conta de alguma ação de um `finalize()` de outro objeto que esteja pronto para ser coletado. Isto se explica porque este método pode realizar qualquer ação, inclusive tornar o objeto disponível novamente para outras *threads*. Dessa forma, se um objeto **A** for referenciado por um objeto **B**, o qual está elegível para o GC, então **A** também é elegível. Entretanto, se no método `finalize()` de **B**, este tornar-se disponível novamente, **A** também deixará de ser elegível. No entanto, como já dissemos, seu objetivo principal é liberar recursos antes que o objeto seja irreversivelmente descartado. Por exemplo, o método `finalize()` de um objeto que implementa uma conexão de Entrada/Saída (I/O), pode explicitamente encerrar a conexão antes da coleta definitiva.

FÓRUM DEVMEDIA

**O lugar perfeito para você ficar por dentro
de tudo o que acontece nas tecnologias do
mercado atual**

No fórum da DevMedia você irá encontrar uma equipe disponível e altamente qualificada com consultores e colaboradores prontos para te ajudar a qualquer hora e sobre qualquer assunto. Temos as salas de Java, .NET, Delphi, Banco de Dados, Engenharia de Software, PHP, Java Script, Web Design, Automação comercial, Ruby on Rails e muito mais!



ACESSE AGORA
www.devmedia.com.br/forum

Introdução ao Java Garbage Collection

Um problema que devemos nos preocupar em relação à implementação de `finalize()` é que, como estudamos até agora, não podemos contar que o objeto será removido pelo Coletor de Lixo. Portanto, também não é assegurado que a ação implementada no código de `finalize()` seja efetivada. Por esses motivos concluímos que não se deve escrever código em `finalize()` que seja fundamental para a aplicação, visto que não se pode ter certeza que ele será executado. Na verdade, recomenda-se que, em geral, o método `finalize()` não seja sobreescrito.

Finalmente, destacamos que `finalize()`, para um dado objeto, é executado uma única vez pelo *Garbage Collector*. Suponha, por exemplo, que para um objeto A – elegível para coleta de lixo – esse método seja implementado de maneira que a referência ao objeto A seja passada a outro objeto B, tornando assim A novamente *vivo*. Se algum tempo depois este mesmo objeto A tornar-se novamente elegível, o *Garbage Collector* não executará mais o método `finalize()` e A finalmente será coletado.

A ausência de um gerenciamento adequado de memória pode causar uma exceção do tipo `OutOfMemoryException`. Esse é um dos principais motivos pelos quais o programador precisa conhecer o funcionamento do Coletor de Lixo, pois mesmo que o desenvolvedor não precise se preocupar em liberar explicitamente a memória ocupada pelos objetos candidatos à coleta, ele ainda necessita tratar de tornar elegíveis os objetos que não forem mais úteis à aplicação.

Por fim, fazemos uma ressalva de que este texto é introdutório. Por isso, recomendamos àqueles que pretendem avançar mais neste tema que estude o documento “Memory Management in the Java HotSpot Virtual Machine”, relacionado nos links no final da matéria. Nele são descritos o funcionamento dos quatro tipos de *Garbage Collection* disponíveis na JVM e como o desenvolvedor pode escolher aquele que melhor se enquadra nas necessidades da sua aplicação.

Autor

Carlos Araújo

stonefull.stm@gmail.com

É professor do curso de Sistemas de Informação no Centro Universitário Luterano de Santarém – Pará. Leciona Estruturas de Dados e Linguagem de Programação Orientada a Objetos usando Java, desenvolve sistemas há mais de 20 anos e é certificado SCJP.



Links e Referências:

Especificação da linguagem Java.

docs.oracle.com/javase/specs/jls/se7/jls7.pdf

Descrição do JConsole, ferramenta de monitoramento de desempenho e consumo de recursos de aplicações Java.

docs.oracle.com/javase/7/docs/technotes/guides/management/jconsole.html

Artigo sobre técnicas de garbage collection aplicáveis às linguagens de programação.

maths.lse.ac.uk/Courses/MA407/gcsurvey.pdf

Descrição do gerenciamento de memória na JVM. Como configurar o Garbage Collector.

oracle.com/technetwork/java/javase/memorymanagement-whitepaper-150215.pdf

Livros

The Java Programming Language, Ken Arnold, James Gosling e David Holmes, Addison Wesley, 2005

Excelente referência para os programadores Java, mesmo para os mais experientes.

Sun Certified Programmer for Java 6 – Study Guide, Kathy Sierra e Bert Bates

Referência destinada à preparação para a certificação SCJP.

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

ca
technologies

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