



www.devmedia.com.br

[versão para impressão]

Link original: <https://www.devmedia.com.br/java-i-o-trabalhando-com-arquivos-em-java/30323>

Java I.O: Trabalhando com arquivos em Java

Este artigo apresenta comandos para criar, excluir e copiar arquivos utilizando classes básicas da linguagem Java pertencentes ao pacote java.io.

Por que eu devo ler este artigo: 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.

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 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.

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; o 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().
```

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;
- **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 readable, 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 suas 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.

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.

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. }
```

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).

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. }
```

Removendo arquivos

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

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. }
```

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 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.    }
13.    boolean sucesso = diretorio.delete();
14.    if (sucesso)
15.        System.out.println("Diretório removido com sucesso.");
16.    else
17.        System.out.println("Erro ao remover diretório.");
18. }
```

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.

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\
```

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 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.

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(), param3.getText()});
55.         }
56.     });
57. }
58. public static void main(String[] args) {
59.     FileAppUI ui = new FileAppUI();
60.     ui.setVisible(true);
61. }
62. }
```

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.

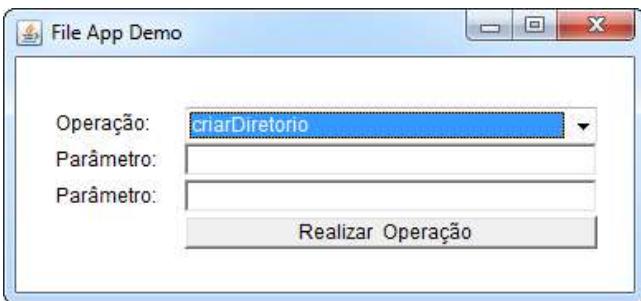


Figura 1. Interface gráfica da aplicação exemplo.

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).

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. 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.

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/>