Desenvolvimento Desktop

Aula 14 - Arquivos – Lendo, Escrevendo e Criando - Parte 2

Apresentação

Nesta aula, apresentaremos em detalhes os conceitos de leitura, escrita, criação e abertura de arquivos, assim como listagem e conteúdo de diretórios. Na API do Java existe uma grande variedade de métodos de entrada e saída para serem utilizados com arquivos à nossa disposição. Para ajudar no entendimento de tais métodos, vejamos alguns exemplos: readAllBytes; readAllLines; newBufferedReader; newBufferedWriter; newInputStream; newOutputStream; newByteChannel; fileChannel; entre outros. Os métodos readAllBytes, readAllLines e write são projetados para usos simples e comuns. Os métodos como newBufferedReader, newBufferedWriter são usados para iterar sobre um stream ou linhas de texto.

Vídeo 01 - Apresentação

Objetivos

Ao final desta aula, você será capaz de:

- Criar, ler e escrever arquivos utilizando as classes e métodos do Java.
- Criar e ler diretórios.
- Listar conteúdos de diretórios.

Entendendo os Arquivos

Uma das principais facilidades em Java, comparando-o com outras linguagens de programação, é a leitura e gravação de arquivos no sistema operacional sem ter de se preocupar com o sistema operacional no qual sua aplicação está rodando. Sendo Java uma linguagem orientada a objetos, nada mais claro que utilizar classes e instâncias delas (objetos) para lidar com a saída e entrada de dados. O java trata a entrada e saída como fluxos de dados (conhecidos como streams), que você tem pleno controle sobre eles. Além disso, a abstração criada pela linguagem sobre os streams é tão grande, que muitas vezes você está puxando/escrevendo dados em algum stream e não sabe se eles estão vindo da internet, de um arquivo texto, ou do usuário que está digitando no console. E o mais importante: essa informação não faz a menor diferença. As classes ligadas a entrada e saída (input/output) de arquivos, como já falamos, se encontram nos pacotes **java.io** e **java.nio (NEW io)**.

Métodos Geralmente Usados para Arquivos Pequenos

Se você estiver trabalhando com arquivos pequenos e gostaria de ler seu conteúdo de uma vez só, você pode usar os métodos **readAllBytes(Path)** ou **readAllLines(Path, Charset)**. Esses métodos fazem tudo que precisa ser feito, como abrir e fechar o stream. Mas não foram feitos para manipular arquivos grandes. O código a seguir mostra como usar o método **readAllBytes**.

```
1 Path file = Paths.get("arquivo.txt");
2 byte [ ] fileArray;
3 fileArray = Files.readAllBytes(file);
```

Você também pode usar um dos métodos de escrita para escrever bytes, ou linhas em um arquivo:

```
1 Path file = Paths.get("arquivo.txt");
2 byte [ ] buf;
3 Files.write(file, buf);
```

Métodos de Entrada/Saída Bufferizados para Arquivos de Texto

O pacote **java.nio.file** suporta canais de entrada/saída, que fazem uso de **buffers** (posições temporárias de memória), melhorando limitações inerentes ao **stream** (fluxo) de dados.

Lendo um Arquivo Usando Buffered Stream I/O

O método **newBufferedReader(Path, Charset)** abre um arquivo para leitura, retornando um **BufferedReader** o qual pode ser usado para ler texto de um arquivo eficientemente.

O trecho de código a seguir mostra como utilizar o método em questão. O arquivo é codificado no padrão “US-ASCII”.

```
1 Charset charset = Charset.forName("US-ASCII");
2 try (BufferedReader reader = Files.newBufferedReader(file, charset)) {
3     String line;
4     while ((line = reader.readLine()) != null) {
5         System.out.println(line);
6     }
7 } catch (IOException x) {
8     System.err.format("IOException: %s%n", x);
9 }
```

Vídeo 02 - Arquivos: Ler Conteúdo

Escrevendo em um Arquivo Através de um Buffered Stream I/O

Pode-se usar o método **newBufferedWriter(Path, Charset, OpenOption...)** para escrever em um arquivo usando um **BufferedWriter**. O trecho de código a seguir mostra como criar um arquivo codificado no padrão “UTF-8” usando esse método. Note que o padrão de codificação “UTF-8” permite a utilização de acentuação ortográfica e por isso esse padrão foi escolhido. O mesmo também é necessário para a leitura de arquivo que contenham acentos.

```
1  Charset charset = Charset.forName("UTF-8");
2  String s = "Esta linha será escrita no arquivo!";
3  Path file2 = Paths.get("arquivo-escrever.txt");
4  try (BufferedWriter writer = Files.newBufferedWriter(file2, charset)) {
5     writer.write(s, 0, s.length());
6  } catch (IOException x) {
7     System.err.format("IOException: %s%n", x);
8  }
```

Métodos para Unbuffered Streams e Interoperação com a API java.io

Lendo:

Para abrir um arquivo para leitura, pode-se usar o método **newInputStream(Path, OpenOption...)**. Esse método retorna um **unbuffered input stream** para ler **bytes** de um arquivo.

```
1  Path file3 = Paths.get("arquivo2.txt");
2  try (InputStream in = Files.newInputStream(file3);
3  BufferedReader reader = new BufferedReader(new InputStreamReader(in))) {
4     String line = null;
5     while ((line = reader.readLine()) != null) {
6        System.out.println(line);
7     }
8  } catch (IOException x) {
9     System.err.println(x);
10 }
```

Criando e escrevendo:

É possível criar, adicionar conteúdo ou escrever em um arquivo através do método **newOutputStream(Path, OpenOption...)**. Esse método abre ou cria um arquivo para escrever **bytes** e retorna um **unbuffered output stream**.

Se o parâmetro opcional **OpenOption** não for especificado e o arquivo não existir, um novo arquivo é criado. Se o arquivo existe, ele é truncado. Essa opção é equivalente a invocar o método com as opções **CREATE** e **TRUNCATE_EXISTING**.

Métodos para Criar Arquivos Temporários e Regulares

Criando arquivos:

Pode-se usar o método **createFile(Path, FileAttribute<?>)** para criar arquivos vazios com um conjunto inicial de atributos. Por exemplo, se na hora da criação você quiser que o arquivo tenha um conjunto de permissões em particular, use o método **createFile**. Se você não especificar qualquer atributo, o arquivo é criado com os atributos padrão. Se o arquivo já existir, o método **createFile** dispara um exceção.

Em uma operação simples e atômica, o método **createFile** checa a existência do arquivo e o cria com os atributos especificados, o que torna o processo mais seguro contra código malicioso. O seguinte trecho de código cria um arquivo com os atributos padrões:

```
1 Path file5 = Paths.get("ArquivoTeste.txt");
2 try {
3    Files.createFile(file5);
4 } catch (FileAlreadyExistsException x) {
5    System.err.format("O arquivo chamado %s já existe%n",file5);
6 } catch (IOException x) {
7    System.err.format("Erro na criação do arquivo: %s%n", x);
8 }
```

Para ver um exemplo de permissões de arquivos veja o link de **POSIX File Permissions**, que se encontra nas referências.

Criando arquivos temporários:

Pode-se criar arquivos temporários utilizando um dos métodos **createTempFile** a seguir:

- **createTempFile(Path, String, String, FileAttribute<?>)**
- **createTempFile(String, String, FileAttribute<?>**)

O primeiro método permite especificar um diretório para o arquivo temporário enquanto que o segundo método cria um novo arquivo no diretório padrão de arquivos temporários. Ambos permitem que se expecifique um sufixo para o nome do arquivo. Além disso, o primeiro método permite que se especifique um prefixo. O seguinte trecho de código mostra um exemplo do segundo método:

```
try {
    Path tempFile = Files.createTempFile(null, ".myapp");
    System.out.format("O arquivo temporário foi criado: %s%n", tempFile);
}
catch (IOException x) {
    System.err.format("IOException: %s%n", x);
}
```

O resultado dessa execução deve ser algo parecido com o seguinte:

O arquivo temporário foi criado: C:\Users\Diego\AppData\Local\Temp\2169560914007070224.myapp

O formato específico do nome do arquivo temporário é dependente da plataforma.

Vídeo 03 - Arquivos: Escrever

Atividade 01

1. Que método é utilizado para adicionar conteúdo ou escrever em um arquivo?

2. O java trata a entrada e saída como fluxos de dados (conhecidos como streams). Essa afirmação é verdadeira ou falsa?

3. Que métodos devemos utilizar para lermos o conteúdo de pequenos arquivos?

Criando e Lendo Diretórios

Listando a raiz de diretórios do sistema de arquivos:

É possível listar todos os diretórios raiz do sistema de arquivos através do método **FileSystem.getRootDirectories**. Esse método retorna um **Iterable** que permite o uso do laço **for** reforçado para iterar (repetir, tornar a fazer a mesma coisa até o limite predeterminado) sobre todos os diretórios raiz.

O seguinte trecho de código imprime os diretórios raiz do sistema de arquivos padrão.

```
1 Iterable<Path> irs = FileSystems.getDefault().getRootDirectories();
2 for (Path name: irs) {
3     System.err.println(name);
4 }
```

Criando um Diretório

Pode-se criar um novo diretório através do método **createDirectory(Path, FileAttribute<?>)**. Se não forem especificados nenhum atributo, o novo diretório terá atributos padrão. Por exemplo:

```
1 try {
2     Files.createDirectories(dir);
3 } catch (IOException ex) {
4     ex.printStackTrace();
5 }
```

Listando o Conteúdo de um Diretório

É possível listar todo o conteúdo de um diretório usando o método **newDirectoryStream(Path)**. Esse método retorna um objeto que implementa a interface **DirectoryStream**. A classe que implementa a interface **DirectoryStream** também implementa **Iterable**. Assim, é possível iterar através do **directory stream** (fluxo do diretório) lendo todos os objetos. Essa abordagem funciona bem com diretórios muito grandes.

```
Path dir2 = Paths.get(“c:\\”);
try (DirectoryStream<Path> stream = Files.newDirectoryStream(dir2)) {
    for (Path file3 : stream) {
        System.out.println(file3.getFileName());
    }
} catch (IOException | DirectoryIteratorException x) {
    System.err.println(x);
}
```

Filtrando uma Listagem de um Diretório Usando Globbing

Se a intenção for listar apenas arquivos e subdiretórios em que seus nomes obedecem a um determinado padrão, você pode fazer isso através do método **newDirectoryStream(Path, String)**, que provê um filtro glob embutido. Se você não é familiarizado com a sintaxe glob, consulte a página **What is a Glob**, cujo endereço eletrônico se encontra no tópico **Referências**.

O exemplo a seguir lista os arquivos relacionados a Java: .class, .java, e .jar:

```
Path dir2 = Paths.get("c:\\");
try (DirectoryStream<Path> stream = Files.newDirectoryStream(dir2, "*.{java,class,jar}")) {
    for (Path file3 : stream) {
        System.out.println(file3.getFileName());
    }
} catch (IOException | DirectoryIteratorException x) {
    System.err.println(x);
}
```

Escrevendo seu Próprio Filtro de Diretório

Para mais informações sobre como navegar no sistema de arquivos, consulte **Walking the File Tree**, cujo endereço eletrônico (que também se encontra nas referências) é: http://docs.oracle.com/javase/tutorial/essential/io/walk.html.

Vídeo 04 - Arquivos: Diretórios

Atividade 02

1. Para listarmos todos os diretórios raiz do sistema de arquivos, que métodos devemos utilizar?
2. Para criarmos um novo diretório devemos utilizar o método **createDirectory(Path, FileAttribute<?>)**. Essa afirmação é verdadeira ou falsa?
3. Que método devemos utilizar para criar arquivos vazios com um conjunto inicial de atributos?

Conclusão

Por enquanto é só, pessoal. Com essa aula concluímos o estudo referente a arquivos. Esperamos que vocês tenham aprendido os princípios básicos sobre o assunto e possam criar e manipular seus próprios arquivos. Mas é preciso praticar muito. Consulte a API do Java para conhecer mais opções sobre arquivos, suas classes e seus métodos. Na próxima aula, veremos como empacotar os arquivos de uma aplicação Java. Até lá.

Resumo

Nesta aula, você aprendeu mais alguma coisa sobre arquivos e diretórios. Conheceu vários métodos que podem ser utilizados para criar, ler e escrever informações em arquivos. Aprendeu também como ler diretórios e listar seus conteúdos através das classes do Java. Esperamos que o conteúdo apresentado nessas duas aulas tenha sido suficiente para que você possa manipular arquivos em Java. Até a próxima aula. Boa Sorte.

Autoavaliação

1. Se quiséssemos criar arquivos temporários, que método devemos utilizar?
2. Para se escrever em um arquivo podemos utilizar o método _____ usando um _____.
    a. BufferedWriter - BufferedWriter.
    b. newBufferedWriter - BufferedWriter.
    c. newFileWriter - FileWriter.
    d. newBufferedWriter - newBufferedWriter.

Referências

WHAT IS A GLOB. <http://docs.oracle.com/javase/tutorial/essential/io/fileOps.html#glob>. Acesso em: 13 jul. 2012.

WALKING THE FILE TREE. <http://docs.oracle.com/javase/tutorial/essential/io/walk.html>. Acesso em: 13 jul. 2012.

POSIX FILE PERMISSIONS: <http://docs.oracle.com/javase/tutorial/essential/io/fileAttr.html#posix>. Acesso em: 13 jul. 2012.