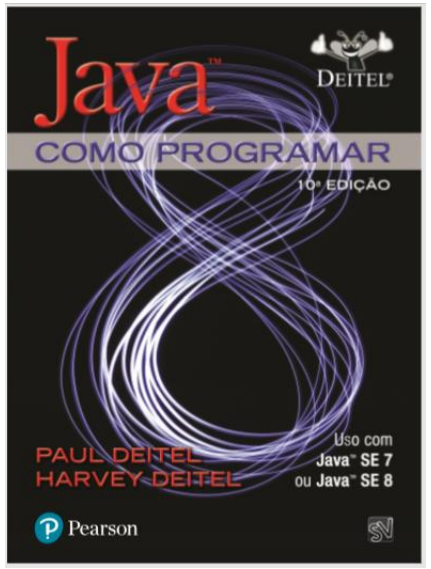
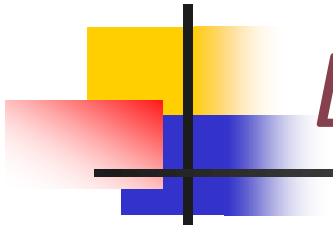


Java – Aula 10
Arquivos e fluxos

Notas de Aula

Prof. André Bernardi



Expressões lambda

- Uma expressão lambda representa um método anônimo — uma notação abreviada para implementar uma interface funcional.
- O tipo de uma lambda é o tipo da interface funcional que a lambda implementa.
- Expressões lambda podem ser usadas em qualquer lugar em que interfaces funcionais são esperadas.

Expressões lambda

- Uma lambda consiste em uma lista de parâmetros seguida pelo símbolo seta (->) e um corpo, como em:

```
(listaDeParâmetros) -> {instruções}
```

Exemplo: a seguinte lambda recebe dois `ints` e retorna sua soma:

```
(int x, int y) -> {return x + y;}
```

O corpo dessa lambda é um bloco de instruções que pode conter uma ou mais instruções entre chaves.

Expressões lambda

- Os tipos de parâmetro de uma lambda podem ser omitidos, como em:

```
(x, y) -> {return x + y;}
```

caso em que o parâmetro e os tipos de retorno são determinados pelo contexto da lambda.

- Uma lambda com um corpo de uma expressão pode ser escrita como:

```
(x, y) -> x + y
```

Nesse caso, o valor da expressão é retornado implicitamente.

Expressões lambda

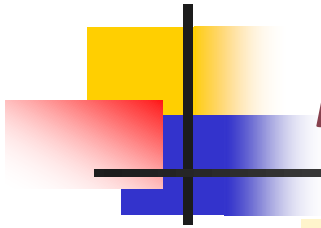
- Quando a lista de parâmetro contém um único parâmetro, os parênteses podem ser omitidos, como em:

```
value -> System.out.printf("%d ", value)
```

- Uma lambda com uma lista de parâmetros vazia é definida com () à esquerda do símbolo seta (->), como em:

```
() -> System.out.println("Welcome to lambdas!")
```

- Também há formas de lambda abreviadas e especializadas que são conhecidas como referências de método.



Rotinas de tratamento de eventos Lambda

- Algumas interfaces ouvintes de evento como ***`ActionListener`*** e ***`ItemListener`*** são interfaces funcionais.
- Para essas interfaces, você pode implementar rotinas de tratamento de evento com lambdas.

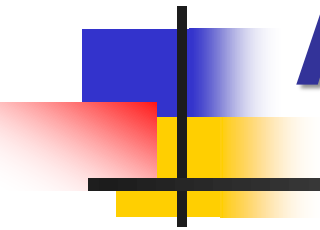
Rotinas de tratamento de eventos Lambda

```
imagesJComboBox.addItemListener(
      new ItemListener() // classe interna anônima
      {
         // trata evento JComboBox
         @Override
         public void itemStateChanged(ItemEvent event)
         {
            // determina se o item está selecionado
            if (event.getStateChange() == ItemEvent.SELECTED)
               label.setIcon(icons[
               imagesJComboBox.getSelectedIndex()]);
         }
      } // fim da classe interna anônima
); // fim da chamada para addItemListener
```

Rotinas de tratamento de eventos Lambda

```
imagesJComboBox.addItemListener(event -> {
        if (event.getStateChange() == ItemEvent.SELECTED)
            label.setIcon(icons[imagesJComboBox.getSelectedIndex()]);
    });
```

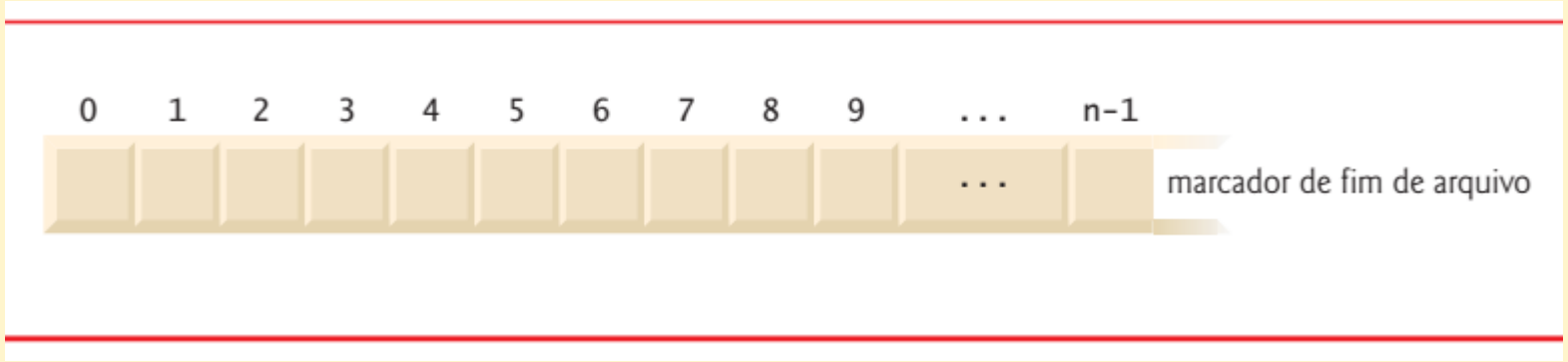
- Para uma rotina de tratamento de evento simples como essa, uma lambda reduz significativamente a quantidade de código que você precisa escrever.



Arquivos e fluxos

Arquivos e fluxos

- O Java vê cada arquivo como um ***fluxo de bytes*** sequencial



Fluxos baseados em caracteres e em bytes

Fluxos de arquivos podem ser utilizados para entrada e saída de dados como bytes ou caracteres.

- **Fluxos baseados em bytes** geram e inserem dados em um formato *binário* — um `char` tem dois bytes, um `int` tem quatro bytes, um `double` tem oito bytes etc.

Fluxos baseados em caracteres e em bytes

- **Fluxos baseados em caracteres** geram e inserem dados como uma *sequência de caracteres* na qual cada caractere tem dois bytes — o número de bytes para determinado valor depende do número de caracteres nesse valor. Por exemplo, o valor `2000000000` requer 20 bytes (10 caracteres a dois bytes por caractere), mas o valor `7` só demanda dois bytes (um caractere a dois bytes por caractere).

Classes e interfaces NIO

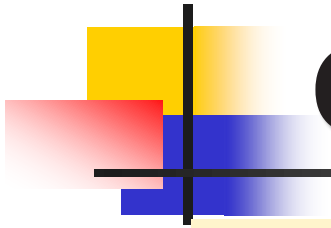
Obtendo informações de arquivo e diretório

- As interfaces **Path** e **DirectoryStream** e as classes **Paths** e **Files** (todas do pacote java.nio.file) são úteis para recuperar informações sobre arquivos e diretórios no disco:

Interface Path

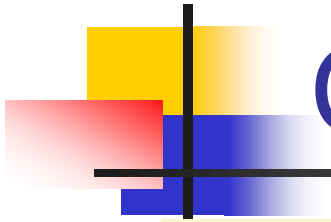
- Os objetos das classes que implementam essa interface representam o local de um arquivo ou diretório.

- Objetos ***Path*** não abrem arquivos nem fornecem capacidades de processamento deles.



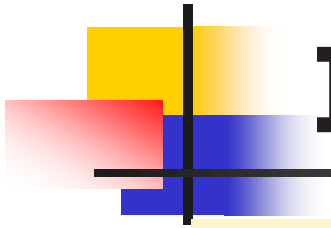
Classe Paths

- Fornece métodos *static* utilizados para obter um objeto ***Path*** representando um local de arquivo ou diretório.



Classe **Files**

- Oferece os métodos **static** para manipulações de arquivos e diretórios comuns, como:
  - copiar arquivos;
  - criar e excluir arquivos e diretórios;
  - obter informações sobre arquivos e diretórios;
  - ler o conteúdo dos arquivos;
  - obter objetos que permitem manipular o conteúdo de arquivos e diretórios.



Interface DirectoryStream

- Os objetos das classes que implementam essa interface possibilitam que um programa itere pelo conteúdo de um diretório.

```java
// Figura 15.2: FileAndDirectoryInfo.java
// A classe File utilizada para obter informações de arquivo e de diretório.
import java.io.IOException;
import java.nio.file.DirectoryStream;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.Scanner;

public class FileAndDirectoryInfo
{
   public static void main(String[] args) throws IOException
   {
      Scanner input = new Scanner(System.in);

      System.out.println("Enter file or directory name:");

      // cria o objeto Path com base na entrada de usuário
      Path path = Paths.get(input.nextLine());
      // se o caminho existe, gera uma saída das informações sobre ele
      if (Files.exists(path))
      {
         // exibe informações sobre o arquivo (ou diretório)
         System.out.printf("%n%s exists%n", path.getFileName());
```

```
25    System.out.printf("%s a directory%n", Files.isDirectory(path) ? "Is" : "Is not");
26    System.out.printf("%s an absolute path%n", path.isAbsolute() ? "Is" : "Is not");
27    System.out.printf("Last modified: %s%n", Files.getLastModifiedTime(path));
28    System.out.printf("Size: %s%n", Files.size(path));
29    System.out.printf("Path: %s%n", path);
30    System.out.printf("Absolute path: %s%n", path.toAbsolutePath() );
31
32       if (Files.isDirectory(path)) // listagem de diretório de saída
33       {
34          System.out.printf("%nDirectory contents:%n");
35
36          // objeto para iteração pelo conteúdo de um diretório
37          DirectoryStream<Path> directoryStream =  Files.newDirectoryStream(path);
38
39             for (Path p : directoryStream)
40                System.out.println(p);
41       }
42    }
43    else // se não for arquivo ou diretório, gera saída da mensagem de erro
44    {
45       System.out.printf("%s does not exist%n", path);
46    }
47 } // fim de main
48 } // fim da classe FileAndDirectoryInfo
```

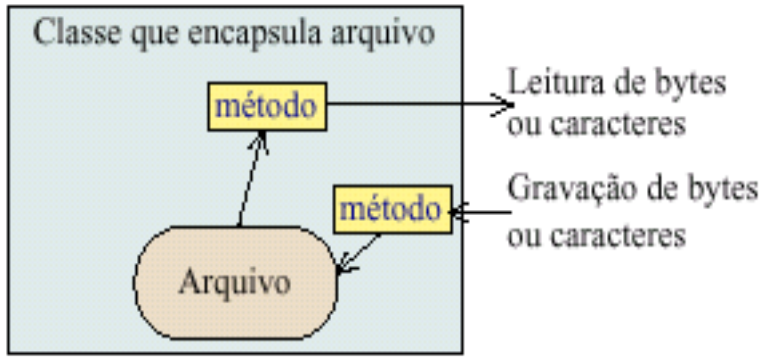
```
Enter file or directory name:
c:\examples\ch15

ch15 exists
Is a directory
Is an absolute path
Last modified: 2013-11-08T19:50:00.838256Z
Size: 4096
Path: c:\examples\ch15
Absolute path: c:\examples\ch15

Directory contents:
C:\examples\ch15\fig15_02
C:\examples\ch15\fig15_12_13
C:\examples\ch15\SerializationApps
C:\examples\ch15\TextFileApps
```

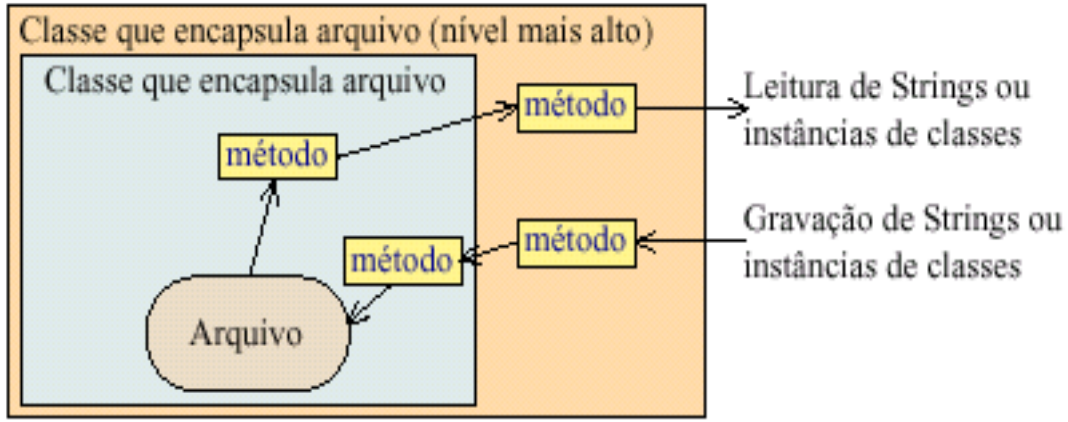
```
Enter file or directory name:
C:\examples\ch15\fig15_02\FileAndDirectoryInfo.java

FileAndDirectoryInfo.java exists
Is not a directory
Is an absolute path
Last modified: 2013-11-08T19:59:01.848255Z
Size: 2952
Path: C:\examples\ch15\fig15_02\FileAndDirectoryInfo.java
Absolute path: C:\examples\ch15\fig15_02\FileAndDirectoryInfo.java
```

Arquivos

Arquivos criados com base nos fluxos de bytes são chamados **arquivos binários**, e aqueles criados com base nos fluxos de caracteres são **arquivos de texto**.

- Tipos:
  - **Arquivos de *texto*;**
  - **Arquivos *binários*;**
  - **Arquivos de *instâncias*.**





Arquivos Texto Seqüenciais (Escrita)

- Classe BufferedWriter
  - **BufferedWriter(FileWriter)**
  - **write(String)**
  - **newLine()**
  - **flush()**
  - **close()**

- Classe Formatter
  - **Formatter("arq.txt")**
  - **format(String)**
  - **close()**

Toda a manipulação das instâncias destas classes, inclusive a sua criação, deve estar dentro de um bloco try, e deve existir ao menos um bloco catch para a exceção IOException.

```
1 // Figura 15.3: CreateTextFile.java
2 // Gravando dados em um arquivo de texto sequencial com a classe Formatter.
3 import java.io.FileNotFoundException;
4 import java.lang.SecurityException;
5 import java.util.Formatter;
6 import java.util.FormatterClosedException;
7 import java.util.NoSuchElementException;
8 import java.util.Scanner;
9
10 public class CreateTextFile
11 {    // envia uma saída de texto para um arquivo
12    private static Formatter output;
13
14    public static void main(String[] args)
15    {
16       openFile();
17       addRecords();
18       closeFile();
19    }
20
```

```
Enter account number, first name, last name and balance.
Enter end-of-file indicator to end input.
? 100 Bob Blue 24.98
? 200 Steve Green -345.67
? 300 Pam White 0.00
? 400 Sam Red -42.16
? 500 Sue Yellow 224.62
? ^Z
```

```
21    // abre o arquivo clients.txt
22    public static void openFile()
23    {
24       try
25       {
26          output = new Formatter("clients.txt"); // abre o arquivo
27       }
28       catch (SecurityException securityException)
29       {
30          System.err.println("Write permission denied. Terminating.");
31          System.exit(1); // termina o programa
32       }
33       catch (FileNotFoundException fileNotFoundException)
34       {
35          System.err.println("Error opening file. Terminating.");
36          System.exit(1); // termina o programa
37       }
38 }
39
```

```
40    // adiciona registros ao arquivo
41    public static void addRecords()
42    {
43       Scanner input = new Scanner(System.in);
44       System.out.printf("%s%n%s%n? ",
45             "Enter account number, first name, last name and balance.",
46             "Enter end-of-file indicator to end input.");
47       // faz um loop até o indicador de fim de arquivo
48       while (input.hasNext())
49       {
50          try
51          {
52          // gera saída do novo registro para o arquivo; supõe entrada válida
53             output.format("%d %s %s %.2f%n", input.nextInt(),
54                   input.next(), input.next(), input.nextDouble());
55          }
56          catch (FormatterClosedException formatterClosedException)
57          {
58             System.err.println("Error writing to file. Terminating.");
59             break;
60          }
```

```
61          catch (NoSuchElementException elementException)
62          {
63             System.err.println("Invalid input. Please try again.");
64             input.nextLine(); // descarta entrada do usuário
65          }
66
67          System.out.print("? ");
68       } // fim do while
69    } // fim do método addRecords
70
71    // fecha o arquivo
72    public static void closeFile()
73    {
74          if (output != null)
75             output.close();
76    }
77 } // fim da classe CreateTextFile
```

```
Enter account number, first name, last name and balance.
Enter end-of-file indicator to end input.
? 100 Bob Blue 24.98
? 200 Steve Green -345.67
? 300 Pam White 0.00
? 400 Sam Red -42.16
? 500 Sue Yellow 224.62
? ^Z
```

Sistema operacional	Combinação de teclas
UNIX/linux/Mac OS X	*<Enter> <Ctrl> d*
Windows	*<Ctrl> z*

Dados de exemplo			
100	Bob	Blue	24.98
200	Steve	Green	-345.67
300	Pam	White	0.00
400	Sam	Red	-42.16
500	Sue	Yellow	224.62

Arquivos Texto Seqüenciais (Leitura)

- Classe BufferedReader.
  - **BufferedReader(FileReader)**
  - **readLine()**
  - **close()**

- Classe Scanner.
  - **Scanner(Paths.get("arq.txt"));**
  - **next()**
  - **close()**

Toda a manipulação das instâncias destas classes, inclusive a sua criação, deve estar dentro de um bloco try, e deve existir ao menos um bloco catch para a exceção IOException.

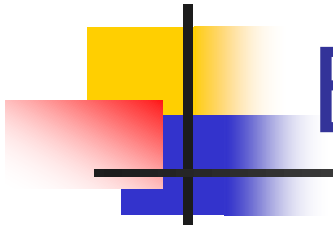
```
1 // Figura 15.6: ReadTextFile.java
2 // Esse programa lê um arquivo de texto e exibe cada registro.
3 import java.io.IOException;
4 import java.lang.IllegalStateException;
5 import java.nio.file.Files;
6 import java.nio.file.Path;
7 import java.nio.file.Paths;
8 import java.util.NoSuchElementException;
9 import java.util.Scanner;
10
11 public class ReadTextFile
12 {
13    private static Scanner input;
14
15    public static void main(String[] args)
16    {
17       openFile();
18       readRecords();
19       closeFile();
20    }
```

```
21
22    // abre o arquivo clients.txt
23    public static void openFile()
24    {
25       try
26       {
27          input = new Scanner(Paths.get("clients.txt"));
28       }
29       catch (IOException ioException)
30       {
31          System.err.println("Error opening file. Terminating.");
32          System.exit(1);
33       }
34    }
35
36    // lê o registro no arquivo
37    public static void readRecords()
38    {
39       System.out.printf("%-10s%-12s%-12s%10s%n", "Account",
40             "First Name", "Last Name", "Balance");
41
```

```
42      try
43      {
44         while (input.hasNext()) // enquanto houver mais para ler
45         {
46            // exibe o conteúdo de registro
47            System.out.printf("%-10d%-12s%-12s%10.2f%n", input.nextInt(),
48                  input.next(), input.next(), input.nextDouble());
49         }
50      }
51      catch (NoSuchElementException elementException)
52      {
53         System.err.println("File improperly formed. Terminating.");
54      }
55      catch (IllegalStateException stateException)
56      {
57         System.err.println("Error reading from file. Terminating.");
58      }
59   } // fim do método readRecords
60
```

```
61    // fecha o arquivo e termina o aplicativo
62    public static void closeFile()
63    {
64       if (input != null)
65          input.close();
66    }
67 } // fim da classe ReadTextFile
```

```
Account   First Name  Last Name     Balance
100       Bob         Blue            24.98
200       Steve       Green         -345.67
300       Pam         White           0.00
400       Sam         Red           -42.16
500       Sue         Yellow         224.62
```



Exercícios Arquivos

1) Escreva um programa em Java que leia um arquivo de palavras e crie três arquivos de saída. As palavras lidas que tiverem menos do que seis caracteres devem ser gravadas no primeiro arquivo de saída, as que tiverem entre seis e dez caracteres devem ser gravadas no segundo arquivo de saída e as que tiverem mais de dez caracteres devem ser gravadas no terceiro arquivo de saída. Ao final da execução do programa mostre no terminal quantas palavras foram gravadas em cada um destes arquivos.

Exercícios Arquivos

2) Escreva um programa em Java que use o algoritmo de César, ou qualquer outro de criptografia para codificar um arquivo de texto, linha por linha. Faça com que o usuário possa entrar o nome do arquivo de entrada, o nome do arquivo de saída e a chave a ser usada pela linha de comando.

3) Escreva um programa em Java que leia dois arquivos de texto simultaneamente e compare os conteúdos destes arquivos (linha por linha). Se os arquivos forem iguais, uma mensagem apropriada deve ser impressa. Se os arquivos forem diferentes, assim que a diferença for detectada, as linhas diferentes devem ser impressas e o programa deve terminar. *Dica:* use a classe BufferedReader para ler os arquivos, e compare-os com o método compare da classe String, que retorna zero **somente** se as Strings sendo comparadas são iguais.

Arquivos para tipos Nativos (Leitura)

DataInputStream e FileInputStream

- Alguns métodos úteis da classe *DataInputStream* são:
  - **DataInputStream():** o construtor da classe. Este construtor espera um argumento que é uma instância da classe *FileInputStream*, que por sua vez será criada usando um nome de arquivo como argumento.
  - **available**(): este método retorna o número de bytes que ainda pode lido do arquivo. Este método pode ser usado para verificar o fim do arquivo, quando não houver mais bytes disponíveis para leitura.

toda a manipulação das instâncias destas classes, inclusive a sua criação, deve estar dentro de um bloco try, e deve existir ao menos um bloco catch para a exceção IOException.

Arquivos para tipos Nativos (Leitura)

- **readBoolean**(): este método lê e retorna um valor do tipo boolean do arquivo.
- **readChar**(): este método lê e retorna um valor do tipo char do arquivo.
- **readByte**(): este método lê e retorna um valor do tipo byte do arquivo.
- **readUnsignedByte**(): este método lê e retorna um byte do arquivo, mas considera que este valor não tem sinal (estando na faixa de valores 0 _ 255), e retorna o valor lido como um inteiro.

Arquivos para tipos Nativos (Leitura)

- **readShort**(): este método lê e retorna um valor do tipo short do arquivo.
- **readUnsignedShort**(): este método lê e retorna um short do arquivo, mas considera que este valor não tem sinal (estando na faixa de valores 0 _ 65535), e retorna o valor lido como um inteiro.
- **readInt**(): este método lê e retorna um valor do tipo int do arquivo.
- **readLong**(): este método lê e retorna um valor do tipo long do arquivo.

Arquivos para tipos Nativos (Leitura)

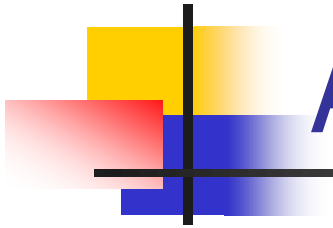
- **readFloat**(): este método lê e retorna um valor do tipo float do arquivo.
- **readDouble**(): este método lê e retorna um valor do tipo double do arquivo.
- **readUTF**(): este método lê e retorna uma String codificada no formato UTF *(Unicode Transfer Forma*t), que representa o tamanho da String junto com a própria String. Uma String representada no formato UTF deve ter no máximo 65535 caracteres. Este método só deve ser usado para ler Strings se estas forem escritas no formato UTF - veja o método writeUTF da classe DataOutputStream.
- **close**(): este método fecha o arquivo associado com a instância da classe DataInputStream, fe-chando também o arquivo associado com a instância da classe FileInputStream que foi usada para construir a instância de DataInputStream.

Arquivos para tipos Nativos (Escrita)

DataOutputStream e FileOutputStream

- Alguns métodos úteis da classe DataOutputStream são:
  - **DataOutputStream**(): o construtor da classe. Este construtor espera um argumento que é uma instância da classe FileOutputStream, que por sua vez será criada usando um nome de arquivo como argumento.
  - **writeBoolean**(): este método recebe como argumento e grava um valor do tipo boolean no arquivo.
  - **writeChar**(): este método recebe como argumento e grava um valor do tipo char no arquivo.

toda a manipulação das instâncias destas classes, inclusive a sua criação, deve estar dentro de um bloco try, e deve existir ao menos um bloco catch para a exceção IOException.



Arquivos para tipos Nativos (Escrita)

Recebem como argumento e gravam no arquivo os tipos indicados pelos nomes dos métodos:

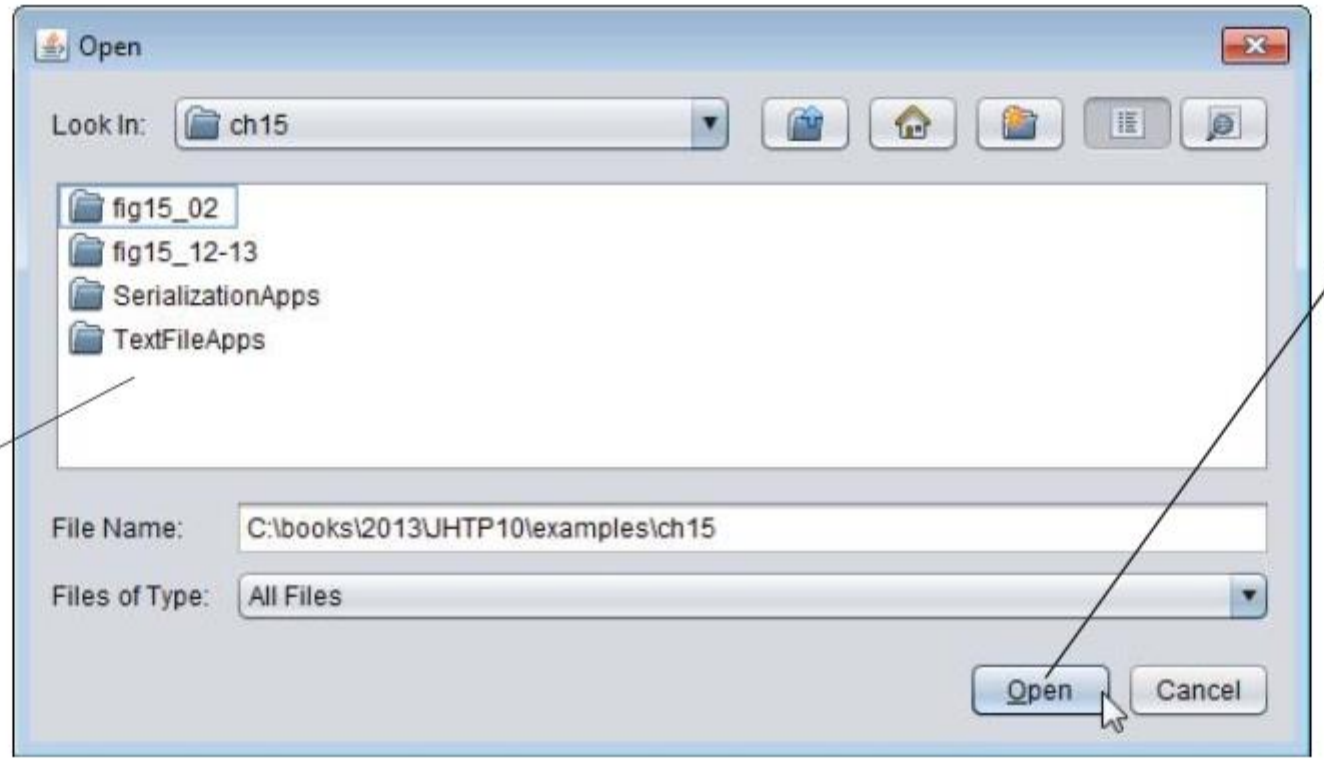
- **writeChar**():
- **writeBoolean**():
- **writeByte**():
- **writeShort**():
- **writeInt**():
- **writeLong**():
- **writeFloat**():
- **writeDouble**():

Arquivos para tipos Nativos (Escrita)

- **writeUTF**(): este método recebe como argumento e grava uma String no arquivo. Esta String estará codificada no formato UTF *(Unicode Transfer Format*), que representa o tamanho da String junto com a própria String. Uma String representada no formato UTF deve ter no máximo 65535 caracteres. Este método só deve ser usado para gravar Strings se estas forem lidas posteriormente no formato UTF
- **flush**(): este método sem argumentos força a gravação de quaisquer valores que ainda não tenham sido gravados no arquivo.
- **close**(): este método fecha o arquivo associado com a instância da classe DataOutputStream, fechando também o arquivo associado com a instância da classe FileOutputStream que foi usada para construir a instância de DataOutputStream. Antes de fechar o arquivo, este método executa o método flush() descrito anteriormente.

Abrindo arquivos com JFileChooser

a) Utilize esse diálogo para localizar e selecionar um arquivo ou diretório

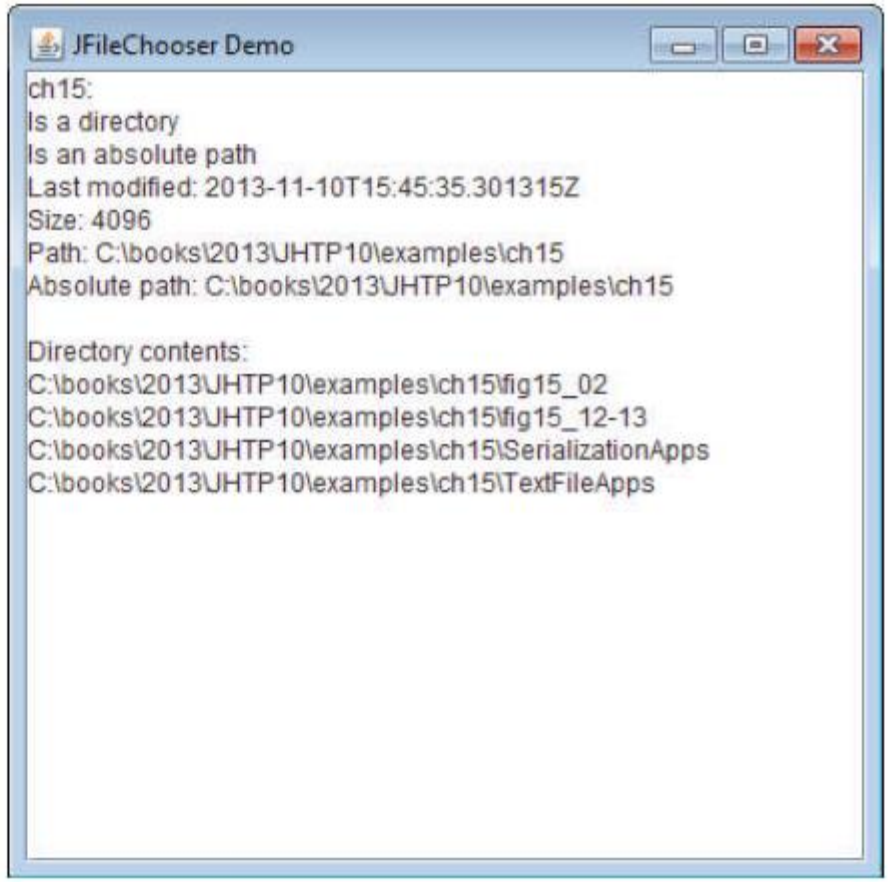


Arquivos e diretórios são exibidos aqui

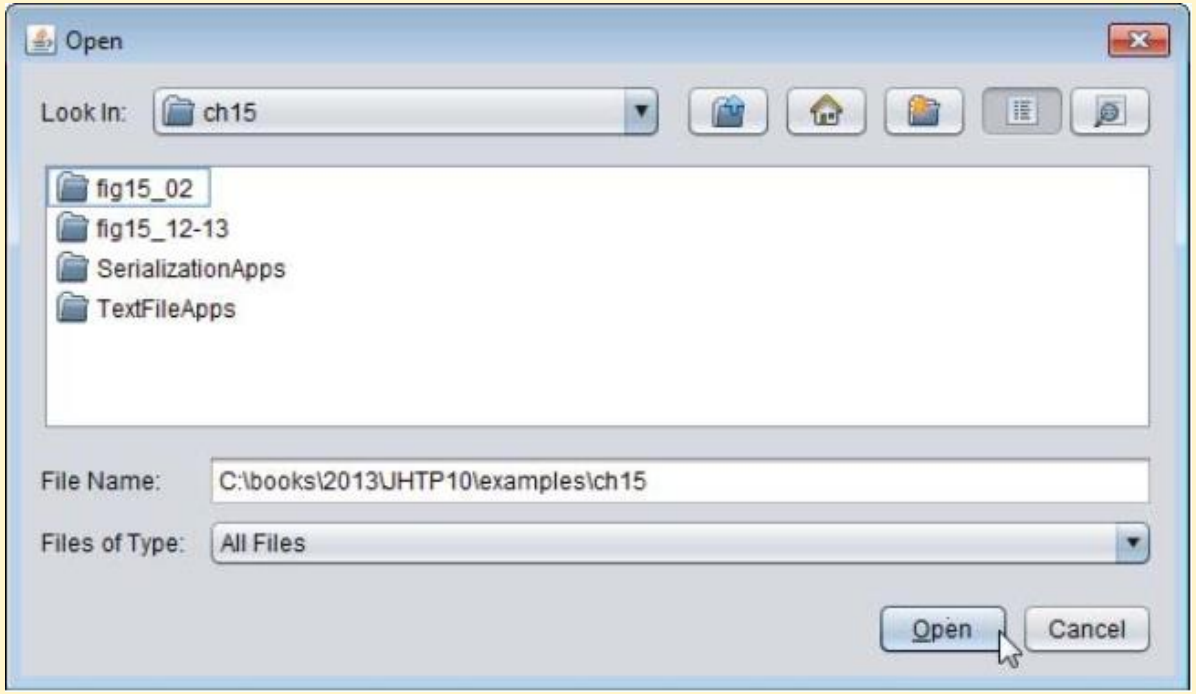
Clique em **Open** para submeter um nome de arquivo ou diretório ao programa

Abrindo arquivos com JFileChooser

b) Informações do arquivo ou diretório selecionado; se for um diretório, o conteúdo dele é exibido



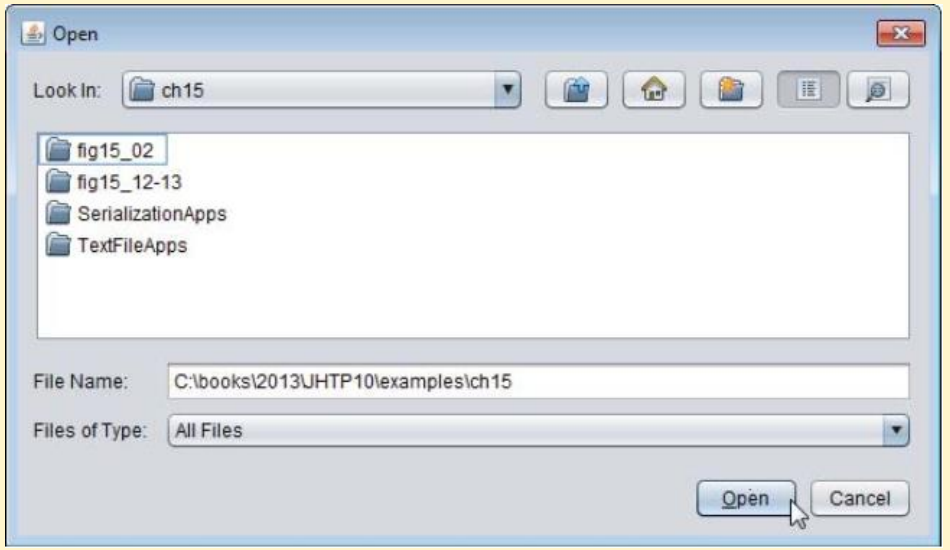
```
1 // Figura 15.12: JFileChooserDemo.java
2 // Demonstrando JFileChooser.
3 import java.io.IOException;
4 import java.nio.file.DirectoryStream;
5 import java.nio.file.Files;
6 import java.nio.file.Path;
7 import java.nio.file.Paths;
8 import javax.swing.JFileChooser;
9 import javax.swing.JFrame;
10 import javax.swing.JOptionPane;
11 import javax.swing.JScrollPane;
12 import javax.swing.JTextArea;
13
14 public class JFileChooserDemo extends JFrame
15 {
16    private final JTextArea outputArea; // exibe o conteúdo do arquivo
17
18    // configura a GUI
19    public JFileChooserDemo() throws IOException
20    {
21       super("JFileChooser Demo");
22       outputArea = new JTextArea();
```



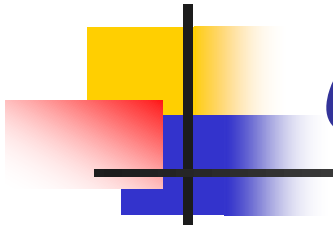
```
23      add(new JScrollPane(outputArea)); // outputArea é rolável
24      analyzePath(); // obtém o Path do usuário e exibe informações
25   }
26
27   // exibe informações sobre o arquivo ou diretório que o usuário especifica
28   public void analyzePath() throws IOException
29   {
30      // obtém o Path para o arquivo ou diretório selecionado pelo usuário
31      Path path = getFileOrDirectoryPath();
32      // se existir, exibe as informações
33      if (path != null && Files.exists(path))
34      {
35      // coleta as informações sobre o arquivo (ou diretório)
36         StringBuilder builder = new StringBuilder();
37         builder.append(String.format("%s:%n", path.getFileName()));
38         builder.append(String.format("%s a directory%n",
39                        Files.isDirectory(path) ? "Is" : "Is not"));
40         builder.append(String.format("%s an absolute path%n",
41                        path.isAbsolute() ? "Is" : "Is not"));
42         builder.append(String.format("Last modified: %s%n",
43                        Files.getLastModifiedTime(path)));
44         builder.append(String.format("Size: %s%n", Files.size(path)));
```

```
45          builder.append(String.format("Path: %s%n", path));
46          builder.append(String.format("Absolute path: %s%n",
47                                       path.toAbsolutePath()));
49          if (Files.isDirectory(path)) // listagem de diretório de saída
50          {
51             builder.append(String.format("%nDirectory contents:%n"));
52
53             // objeto para iteração pelo conteúdo de um diretório
54             DirectoryStream<Path> directoryStream =
55             Files.newDirectoryStream(path);
56
57             for (Path p : directoryStream)
58                builder.append(String.format("%s%n", p));
59          }
61       outputArea.setText(builder.toString()); // exibe o conteúdo de String
62    }
63    else // Path não existe
64    {
65          JOptionPane.showMessageDialog(this, path.getFileName() +
66             " does not exist.", "ERROR", JOptionPane.ERROR_MESSAGE);
67    }
68 } // fim do método analyzePath
```

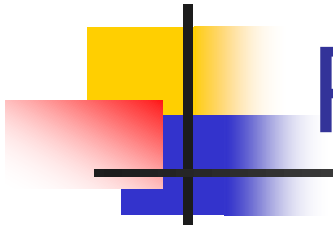
```
69
70    // permite que o usuário especifique o nome de arquivo ou diretório
71    private Path getFileOrDirectoryPath()
72    {
73    // configura o diálogo permitindo a seleção de um arquivo ou diretório
74       JFileChooser fileChooser = new JFileChooser();
75       fileChooser.setFileSelectionMode(JFileChooser.FILES_AND_DIRECTORIES);
77       int result = fileChooser.showOpenDialog(this);
78
79       // se o usuário clicou no botão Cancel no diálogo, retorna
80       if (result == JFileChooser.CANCEL_OPTION)
81          System.exit(1);
82
83       // retorna o Path representando o arquivo selecionado
84       return fileChooser.getSelectedFile().toPath();
85    }
86 } // fim da classe JFileChooserDemo
```




```
1 // Figura 15.13: JFileChooserTest.java
2 // Testa a classe JFileChooserDemo.
3 import java.io.IOException;
4 import javax.swing.JFrame;
5
6 public class JFileChooserTest
7 {
8    public static void main(String[] args) throws IOException
9    {
10      JFileChooserDemo application = new JFileChooserDemo();
11      application.setSize(400, 400);
12      application.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13      application.setVisible(true);
14   }
15 } // fim da classe JFileChooserTest
```



Classes **ObjectInputStream** e **ObjectOutputStream**



Referencias

- Java How to program 3, 4, 5, 6 e 10 ed.

Paul Deitel e Harvey Deitel

- Sun

http://java.sun.com