

Pós-Graduação em Distância



Java SE - Programação Avançada

Prof. Emilio Celso de Souza



Apresentação	5
Módulo 1 - Tratamento de exceção	6
Aula 1 – Utilização do tratamento de exceções em Java	6
Visão geral do tratamento de exceções	6
Lançando uma exceção: throw new	10
Classes Java para tratamento de exceções	13
Aula 2 - Hierarquia de exceções em Java	14
Hierarquia de exceções	14
Sumário	
Aula 3 - Tipos de exceções	17
Exceções verificadas e não verificadas	17
O bloco finally	20
Exercícios do Módulo 1	21
Módulo 2 - Coleções	22
Aula 4 - Visão geral das coleções	22
Características das coleções	22
Aula 5 - Interface Collection e classe Collections	25
A interface List	25
A interface Set	33
A interface Map	34
Aula 6 - Classe Arrays	36
Definição da classe Arrays	36
Exercícios do Módulo 2	37
Módulo 3 - Documentação API do Java	39
Aula 7 - Conceitos da API do Java	39
Utilização da API do Java	39
Exercícios do Módulo 3	41
Módulo 4 - Multithreading	42
Aula 8 – Criação e execução de threads	42
Definição de threads	42
Criação de threads	42
Definindo um nome para a thread	45
Executando múltiplas threads	46
Aula 9 - Estados de threads	48
A classe Thread	48
Prioridades e agendamento de threads	49
Aula 10 - Relacionamentos entre threads - sincronismo	50
Aula 11 - Multithreading com GUI	54
Exercícios do Módulo 4	57
Módulo 5 - Componentes GUI	58
Aula 12 - Introdução aos componentes GUI	58
Definindo a classe	58
Exibindo a janela	58



Aula 13 - Eventos GUI	61
Tratamento de eventos GUI	63
Aula 14 - Entrada e saída baseadas em GUI	63
Visão geral de componentes <i>swing</i>	64
Aula 15 - <i>Applets</i>	66
Ciclo de vida de um <i>applet</i>	66
Exemplo de aplicação usando Applet	67
A classe Applet	67
Chamando um <i>applet</i>	67
Aula 16 - Informando parâmetros para o <i>applet</i>	68
Exibindo imagens	69
Reproduzindo áudio e vídeo	71
Aula 17 - Java Media Framework	72
Adicionando recursos de áudio e vídeo a uma aplicação Java	72
Codec	72
Arquitetura do JMF	73
Exercícios do Módulo 5	77
Módulo 6 - Arquivos	78
Aula 18 - Entrada e saída em Java usando arquivos	78
Entrada e saída em Java: <i>streams</i>	78
InputStream e OutputStream	78
Manipulação de arquivos: FileInputStream e FileOutputStream	80
Aula 19 - A classe File	83
Utilizando a classe File	83
FileReader e FileWriter	85
Exercícios do Módulo 6	89
Módulo 7 - Acesso a Banco de Dados JDBC	91
Aula 20 - Conectividade com BD	91
Java DataBase Connectivity API	91
Tipos de <i>drivers</i> JDBC	92
Utilizando o <i>driver</i> JDBC	92
Inserindo o <i>connector</i> Java para acesso ao MySQL	92
Aula 21 - Objetos de acesso a dados	95
Declarando objetos de acesso a dados usando interfaces	95
Definindo a <i>string</i> de conexão	95
A classe DriverManager	96
Criando métodos para realizar as operações CRUD - <i>Create, Recover, Update, Delete</i>	97
Hierarquia de classes JDBC	101
Métodos das interfaces PreparedStatement e ResultSet	101
Aula 22 - <i>Stored procedures</i> no MySQL	103
Criando e invocando <i>stored procedures</i> no MySQL	103
Chamando uma <i>procedure</i> MySQL em Java	104
Exercícios do Módulo 7	105

Considerações Finais	107
Respostas Comentadas dos Exercícios	108
Exercício 1 – Módulo 1	108
Exercício 2 - Módulo 1	109
Exercício 1 – Módulo 2	110
Exercício 2 - Módulo 2	112
Exercício 3 – Módulo 2	113
Exercício 1 – Módulo 3	114
Exercício 1 – Módulo 4	116
Exercício 2 – Módulo 4	116
Exercício 3 – Módulo 4	118
Exercício 1 – Módulo 5	119
Exercício 2 – Módulo 5	121
Exercício 3 – Módulo 5	122
Exercício 1 – Módulo 6	123
Exercício 2 – Módulo 6	126
Exercício 1 – Módulo 7	127
Exercício 2 – Módulo 7	128
Referências Bibliográficas	130

Apresentação

Caro aluno, nesta disciplina, vamos estudar os conceitos avançados da linguagem Java.

É importante ter um bom conhecimento dos conceitos em Java SE na parte referente à programação básica, pois são essenciais para uma completa compreensão das aulas de Java SE na parte de programação avançada, que é objeto de nosso estudo.

Nesta disciplina, trataremos de temas como tratamento de erros, coleções, *threads*, manipulação de arquivos e acesso a banco de dados. Colocando de outra forma, trabalharemos com aplicações mais próximas àquelas usadas pelas empresas. Portanto, seus horizontes profissionais certamente serão bastante ampliados a partir desse novo conhecimento.

Todo bom sistema possui acesso a banco de dados, e esse mecanismo será abordado aqui. Suponha que você esteja trabalhando em um aplicativo qualquer (um cadastro de clientes, por exemplo). Enquanto seus clientes são cadastrados, seu aplicativo atualiza o valor do dólar. Essa atualização ocorre de forma assíncrona, e, para entendê-la, estudaremos *threads*.

A quantidade de código nesta disciplina é substancialmente ampla porque nosso foco é na compreensão dos recursos mais avançados da linguagem.

Espero que todos tenham um excelente aprendizado e aproveitem ao máximo nossas aulas. Você poderá contar conosco para ajudá-lo ao longo dessa importante jornada rumo à sua profissionalização como desenvolvedor Java!

Prof. Emilio

Módulo 1 - Tratamento de exceção

Aula 1 – Utilização do tratamento de exceções em Java

Visão geral do tratamento de exceções

Como o próprio nome sugere, uma exceção é algo que acontece de forma inesperada frente a um evento considerado como regra. Trazendo esse conceito para os nossos programas, uma exceção ocorre quando a execução de um código produz um resultado diferente daquele considerado “normal”.

Vamos considerar o exemplo apresentado no quadro 1, em que exceções ocorrem e não são tratadas.

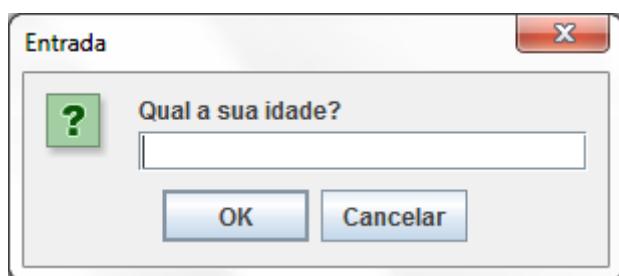
Quadro 1: classe TesteExcecoes sem tratamento de exceções

```

1 package com.ead.programas;
2
3 import javax.swing.JOptionPane;
4
5 public class TesteExcecoes {
6     public static void main(String[] args) {
7         String idade = JOptionPane.showInputDialog("Qual a sua idade?");
8         int valorIdade = Integer.parseInt(idade);
9         System.out.println("Sua idade é " + valorIdade);
10    }
11 }
```

No quadro 1, as linhas que merecem destaque são:

Linha 7: o usuário fornece um valor para a idade.



Linha 8: o valor fornecido é convertido para inteiro.

Quando esse programa é executado, pode ocorrer uma das três situações a seguir:

- O usuário fornece um valor que é possível converter para inteiro. Por exemplo: 10. Nesse caso, a conversão ocorre normalmente, e a resposta é apresentada na tela, de acordo com a execução do código na Linha 9:

```
<terminated> TesteExcecoes [Java Application]
Sua idade é 10
```

2. O usuário fornece um valor que não pode ser convertido para inteiro. Por exemplo: "vinte", ou mesmo 020. Esses valores não podem ser devidamente convertidos para o inteiro que o programa espera receber, pois não possuem um formato adequado. Se um deles (digamos, "vinte") for informado pelo usuário, teremos a saída a seguir:

```
Exception in thread "main" java.lang.NumberFormatException: For input string: "vinte"
at java.lang.NumberFormatException.forInputString(Unknown Source)
at java.lang.Integer.parseInt(Unknown Source)
at java.lang.Integer.parseInt(Unknown Source)
at com.ead.programas.TesteExcecoes.main(TesteExcecoes.java:8)
```

Esse erro indica justamente a impossibilidade de conversão ocorrida na linha 8. Veja que a indicação do erro, lendo de baixo para cima, apresenta o nome da classe seguido do número da linha. Observe, também, que, seguindo esse erro no mesmo sentido, na segunda linha, temos:

at java.lang.Integer.parseInt(Unknown Source)

o que mostra que a causa do erro gerado na linha 8 é a execução do método *parseInt()*.

3. Ao invés de fornecer um valor, o usuário clica no botão "Cancelar". Nesse caso, o erro é um pouco diferente:

```
Exception in thread "main" java.lang.NumberFormatException: null
at java.lang.Integer.parseInt(Unknown Source)
at java.lang.Integer.parseInt(Unknown Source)
at com.ead.programas.TesteExcecoes.main(TesteExcecoes.java:8)
```

Esse erro, ocorrido na mesma linha que o anterior, é produzido pela falta de um valor. Se o usuário clicar em "Cancelar", como indicado no exemplo, nenhum valor é atribuído à *string* na linha 7, e ela recebe o valor *null*. Como *null* não pode ser convertido para inteiro, ocorre a exceção.

Esse e muitos outros tipos de exceção podem ocorrer sempre que for detectada uma inconsistência na execução do programa.

Mas fique tranquilo, pois exceções são muito comuns. Por mais perfeito que possa parecer um programa, elas são inevitáveis, especialmente quando se depende de agentes externos. Alguns exemplos de exceções são:

- Um programa tenta gravar alguma informação em um disco externo, mas esse disco não está disponível.
- Um programa tenta enviar um *e-mail*, mas a internet está indisponível.
- Um programa tenta enviar um arquivo para uma pasta compartilhada na rede, mas a rede está fora do ar.

Mecanismo de tratamento e captura de exceções: **try...catch**

Como vimos, não podemos evitar as exceções, mas podemos tratá-las. No nosso exemplo, podemos reescrever o programa da forma mostrada no quadro 2.

Quadro 2: Classe TesteExcecoes com tratamento de exceções

```

1 package com.ead.programas;
2
3 import javax.swing.JOptionPane;
4
5 public class TesteExcecoes {
6     public static void main(String[] args) {
7         try {
8             String idade = JOptionPane.showInputDialog("Qual a sua idade?");
9             int valorIdade = Integer.parseInt(idade);
10            System.out.println("Sua idade é " + valorIdade);
11        } catch (NumberFormatException e) {
12            System.out.println(e.getMessage());
13        }
14    }
15 }
```

Nessa versão do programa, usamos um bloco *try...catch*. Esse bloco, de forma geral, funciona como mostra o quadro 3.

Quadro 3: Sintaxe do *try...catch*

```

try{
    código desejado
} catch(ClasseDaExceçãoException e){
    código alternativo, se uma exceção ocorrer no bloco try.
}
```

No quadro 3, temos as seguintes partes a destacar:

- **Código desejado:** é o código considerado normal para o nosso programa. No nosso exemplo do quadro 2, o código desejado é:

```

8     String idade = JOptionPane.showInputDialog("Qual a sua idade?");
9     int valorIdade = Integer.parseInt(idade);
10    System.out.println("Sua idade é " + valorIdade);
```

- **ClasseDaExceçãoException:** é a classe que define uma referência responsável por receber o erro gerado pelo código. No nosso exemplo, é NumberFormatException:

```

11 } catch (NumberFormatException e) {
12     System.out.println(e.getMessage());
13 }
```

- **Código alternativo:** é o código escrito como alternativa ao erro. No nosso exemplo, apenas mostramos uma mensagem na tela informando a mensagem de erro:

`System.out.println(e.getMessage());`

Quando executamos o programa do quadro 2 informando um valor inválido (por exemplo, a palavra "vinte", como sugerido anteriormente), temos o resultado:

```
<terminated> TesteExcecoes [Java Application]
For input string: "vinte"
```

Dessa forma, dizemos que o código está protegido pelo bloco *try*, e o bloco *catch* tem por finalidade capturar a exceção gerada. A captura da exceção é realizada por meio da atribuição dessa exceção à referência cujo tipo é definido no parâmetro do bloco *catch()*.

Para entender o mecanismo de captura de exceções, vamos considerar o mesmo exemplo do quadro 2, de forma mais simplificada, no quadro 4.

Quadro 4: Exemplo de tratamento de exceções simplificado

```
public static void main(String[] args) {
    try {
        String idade = "20";
        int valorIdade = Integer.parseInt(idade);
    } catch (NumberFormatException e) {
        System.out.println(e.getMessage());
    }
}
```

Aqui, o erro é causado no momento em que ocorre a conversão de uma *string* para *int*. Falando mais especificamente, o erro é gerado pelo método *parseInt()*. Esse método, pela sua definição, verifica se é possível converter a *string* mencionada. Se não for, o método cria a exceção do tipo *NumberFormatException*, ou seja, cria um objeto dessa classe que é passado para a referência definida no bloco *catch()*, como mostra o quadro 5.

Quadro 5: Explicação do tratamento de exceções

```

public static void main(String[] args) {
try {
    String idade = "20";
    int valorIdade = Integer.parseInt(idade);

} catch (NumberFormatException e) {
    System.out.println(e.getMessage());
}
}

```

O método `parseInt()` lança a exceção, que é recebida pelo bloco `catch()`

Lançando uma exceção: `throw new`

Como a exceção é produzida por um método? No caso do `parseInt()` dos nossos exemplos anteriores, não temos acesso à sua implementação, por ser um método pertencente à API do Java. Por isso, vamos criar o nosso método capaz de lançar uma exceção.

Vamos considerar um exemplo com base no seguinte cenário: um método recebe como parâmetro a idade de uma pessoa e retorna o texto “Maior de Idade” se o valor for maior ou igual a 18 e o texto “Menor de Idade” caso contrário.

A definição do método é apresentada no quadro 6.

Quadro 6: Classe Pessoa com método que não trata exceções

```

1 package com.ead.classes;
2
3 public class Pessoa {
4
5     public static String exibirMaioridade(int idade){
6         if(idade < 18){
7             return "Menor de Idade";
8         }
9         else {
10            return "MAior de Idade";
11        }
12    }
13 }

```

O problema com o método `exibirMaioridade()` é que ele pode receber valores negativos. Como regra geral, não existe idade negativa, e isso deve ser evitado. Como o nosso método permite receber qualquer valor inteiro e está definido em uma classe capaz de ser aproveitada para qualquer tipo de aplicação, não

convém criarmos uma mensagem de erro apenas. Já que idade negativa é inválida, não podemos aceitá-la no método.

Sendo assim, nós podemos fazer o método lançar uma exceção toda vez que ele receber um valor negativo para a idade. A nova versão é mostrada no quadro 7.

Quadro 7: Classe Pessoa com método que usa *throw new*

```

1 package com.ead.classes;
2
3 public class Pessoa {
4
5     public static String exibirMaioridade(int idade){
6         if(idade < 0){
7             throw new IllegalArgumentException("A idade não pode ser negativa!");
8         }
9         if(idade < 18){
10            return "Menor de Idade";
11        }
12        else {
13            return "MAior de Idade";
14        }
15    }
16 }
```

Como pode ser visto no quadro 7, uma exceção é criada (e, portanto, lançada) por um método por meio da instrução:

```
throw new IllegalArgumentException("A idade não pode ser negativa!");
```

O comando *throw* tem por objetivo criar uma nova exceção (se for seguido do operador *new*) ou propagar uma exceção existente.

A utilização do método *exibirMaioridade()* da classe Pessoa fica como mostra o quadro 8.

Quadro 8: classe TesteExcecoes02

```

1 package com.ead.programas;
2
3 import com.ead.classes.Pessoa;
4
5 public class TesteExcecoes02 {
6     public static void main(String[] args) {
7         String resultado = Pessoa.exibirMaioridade(-10);
8         System.out.println(resultado);
9     }
10 }
```

Observe o valor negativo informado como parâmetro. Sua execução produz o resultado mostrado no quadro 9.

Quadro 9: Explicação do tratamento de exceções

Veja a mensagem, a mesma informada como parâmetro no construtor da classe `IllegalArgumentException`.

```
<terminated> TesteExcecoes02 [Java Application] C:\Program Files (x86)\Java\jre7\bin\javaw.exe (15/05/2014 16:24:24)
Exception in thread "main" java.lang.IllegalArgumentException: A idade não pode ser negativa!
    at com.ead.classes.Pessoa.exibirMaioridade(Pessoa.java:7)
    at com.ead.programas.TesteExcecoes02.main(TesteExcecoes02.java:7)
```

No exemplo do quadro 8, escolhemos uma classe da API chamada `IllegalArgumentException` para capturar nossa exceção. Para tratarmos esse erro, vamos usar o bloco `try...catch()` de forma adequada, como mostra o quadro 10.

Quadro 10: Classe TesteExcecoes02 com `try...catch`

```
1 package com.ead.programas;
2
3 import com.ead.classes.Pessoa;
4
5 public class TesteExcecoes02 {
6     public static void main(String[] args) {
7         try {
8             String resultado = Pessoa.exibirMaioridade(-10);
9             System.out.println(resultado);
10        } catch (IllegalArgumentException e) {
11            System.out.println(e.getMessage());
12        }
13    }
14 }
```

O quadro 11 apresenta uma ilustração da exceção gerada pelo método `exibirMaioridade()`.

Quadro 11: Explicação do tratamento de exceções

```
public static void main(String[] args) {
    try { String resultado = Pessoa.exibirMaioridade(-10);

        O método exibirMaioridade() lança a
        exceção do tipo IllegalArgumentException,
        que é capturada pelo bloco
        catch().

    } catch (IllegalArgumentException e) {
        System.out.println(e.getMessage());
    }
}
```

O método exibirMaioridade() lança a exceção do tipo **IllegalArgumentException**, que é capturada pelo bloco **catch()**.



Classes Java para tratamento de exceções

Existem diversas classes disponíveis para tratamento de exceções. Algumas do pacote *java.lang* são apresentadas a seguir:

- **ArithmaticException.**
- **ArrayIndexOutOfBoundsException.**
- **ArrayStoreException.**
- **ClassCastException.**
- **ClassNotFoundException.**
- **CloneNotSupportedException.**
- **EnumConstantNotPresentException.**
- **Exception.**
- **IllegalAccessException.**
- **IllegalArgumentException.**
- **IllegalMonitorStateException.**
- **IllegalStateException.**
- **IllegalThreadStateException.**
- **IndexOutOfBoundsException.**
- **InstantiationException.**
- **InterruptedException.**
- **NegativeArraySizeException.**
- **NoSuchFieldException.**
- **NoSuchMethodException.**
- **NullPointerException.**
- **NumberFormatException.**
- **ReflectiveOperationException.**
- **RuntimeException.**
- **SecurityException.**

- `StringIndexOutOfBoundsException`.
- `TypeNotPresentException`.
- `UnsupportedOperationException`.

Cada uma dessas classes foi criada com um propósito definido, mas elas seguem uma hierarquia.

Aula 2 - Hierarquia de exceções em Java

Hierarquia de exceções

Basicamente, a hierarquia de classes em Java segue o esquema mostrado na figura 1.

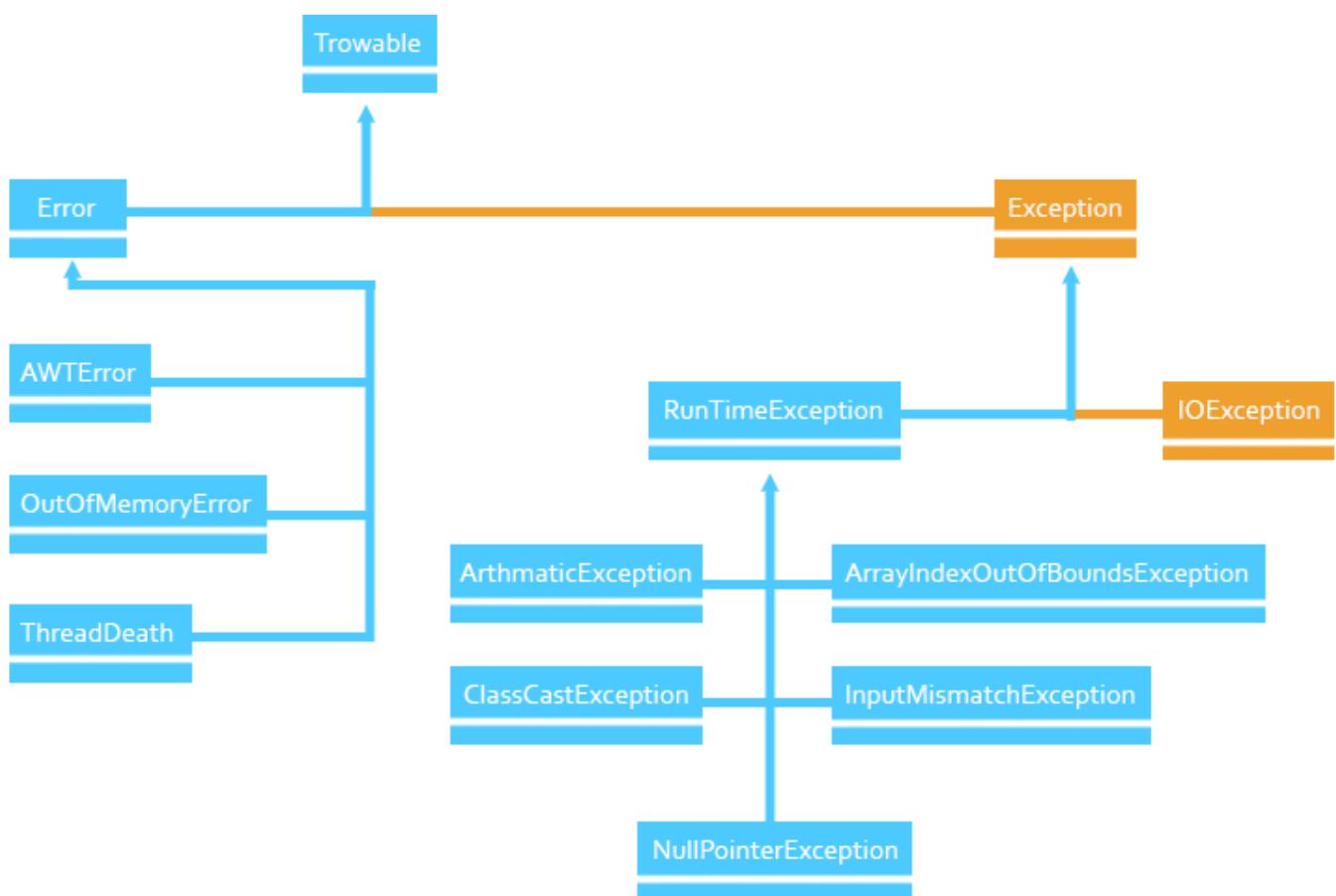


Figura 1 – Hierarquia das exceções em Java.

Na hierarquia apresentada, podemos verificar que:

- A superclasse de todas as classes representativas de exceções é `Throwable`.
- `Throwable` possui duas subclasses: `Exception` e `Error`.
- A classe `Error` representa os erros que estão fora do controle do programa, por exemplo, memória insuficiente, erro do JVM, erro no disco, entre outros. Eles não são capturados por não representarem dados específicos do seu programa.

A classe `Exception` possui como subclasses praticamente todas as exceções da API.

Com base nessa hierarquia, vamos considerar mais um exemplo, apresentado no quadro 12.

Quadro 12: Classe TesteExcecoes3

```

1 package com.ead.programas;
2
3 import javax.swing.JOptionPane;
4
5 import com.ead.classes.Pessoa;
6
7 public class TesteExcecoes03 {
8     public static void main(String[] args) {
9         try {
10             String idade = JOptionPane.showInputDialog("Informe sua idade:");
11             int valorIdade = Integer.parseInt(idade);
12             String resultado = Pessoa.exibirMaioridade(valorIdade);
13             System.out.println(resultado);
14         } catch (NumberFormatException e) {
15             System.out.println(e.getMessage());
16         } catch (IllegalArgumentException e) {
17             System.out.println(e.getMessage());
18         }
19     }
20 }
```

Veja, no quadro 12, que temos os dois métodos anteriores sendo executados:

- `parseInt()` => lança `NumberFormatException`.
- `exibirMaioridade()` => lança `IllegalArgumentException`.

É possível definir um bloco `catch()` para cada exceção separadamente. Mas por que faríamos isso? Essa é uma boa prática quando queremos tratar cada exceção de uma forma diferente. Por exemplo, se o erro for `NumberFormatException`, podemos enviar um e-mail para o usuário, e, se for `IllegalArgumentException`, podemos gerar um arquivo de *log*.

No exemplo do quadro 12, as duas exceções, quando capturadas, apresentam uma mensagem na tela. Sendo assim, podemos definir uma única exceção capaz de capturar ambas. A condição é que essa exceção represente uma superclasse comum a todas as classes de exceção que queremos simplificar.

Por exemplo, o programa do quadro 12 pode ser escrito da forma mostrada no quadro 13.

Quadro 13: Classe TesteExcecoes3 reescrita

```

1 package com.ead.programas;
2
3 import javax.swing.JOptionPane;
4
5 import com.ead.classes.Pessoa;
6
7 public class TesteExcecoes03 {
8     public static void main(String[] args) {
9         try {
10             String idade = JOptionPane.showInputDialog("Informe sua idade:");
11             int valorIdade = Integer.parseInt(idade);
12             String resultado = Pessoa.exibirMaioridade(valorIdade);
13             System.out.println(resultado);
14         } catch (Exception e) {
15             System.out.println(e.getMessage());
16         }
17     }
18 }
```

Veja que, nesse exemplo, a exceção declarada no bloco `catch()` é do tipo `Exception`, que é superclasse de `NumberFormatException` e de `IllegalArgumentException`. Aliás, `Exception` é superclasse de todas as exceções, como vimos na hierarquia.

Nesse caso, porém, um cuidado deve ser tomado. Ainda que queiramos tratar as exceções individualmente, podemos ainda incluir a captura para `Exception` para as demais. Mas é necessário que essa seja a última exceção a ser capturada, como mostra o quadro 14.

Quadro 14: Classe TesteExcecoes3 reescrita com mais de um bloco `catch`

```

1 package com.ead.programas;
2
3 import javax.swing.JOptionPane;
4
5 import com.ead.classes.Pessoa;
6
7 public class TesteExcecoes03 {
8     public static void main(String[] args) {
9         try {
10             String idade = JOptionPane.showInputDialog("Informe sua idade:");
11             int valorIdade = Integer.parseInt(idade);
12             String resultado = Pessoa.exibirMaioridade(valorIdade);
13             System.out.println(resultado);
14         } catch(NumberFormatException e){
15             //tratamento para NumberFormatException
16         } catch(IllegalArgumentException e){
17             //tratamento para IllegalArgumentException
18         } catch (Exception e) {
19             //tratamento para Exception
20         }
21     }
22 }
```

Se fizermos o contrário, ou seja, se colocarmos `Exception` no primeiro bloco `catch()`, os demais não serão alcançados, e o compilador entende dessa forma. Veja o erro de compilação indicado no Eclipse no quadro 15.

Quadro 15: classe TesteExcecoes3 com exceções aninhadas incorretamente

```

1 package com.ead.programas;
2
3 import javax.swing.JOptionPane;
4
5 import com.ead.classes.Pessoa;
6
7 public class TesteExcecoes03 {
8     public static void main(String[] args) {
9         try {
10             String idade = JOptionPane.showInputDialog("Informe sua idade:");
11             int valorIdade = Integer.parseInt(idade);
12             String resultado = Pessoa.exibirMaioridade(valorIdade);
13             System.out.println(resultado);
14         }catch (Exception e) {
15             //tratamento para Exception
16         } catch(NumberFormatException e){
17             //tratamento para NumberFormatException
18         } catch(InvalidArgumentException e){
19             //tratamento para InvalidArgumentException
20         }
21     }
22 }

```

A regra geral para essa situação é: "Se usar mais de um bloco catch(), verificar a hierarquia de classes envolvida. As classes mais específicas devem ser indicadas primeiro, e as mais generalizadas, por último".

Aula 3 - Tipos de exceções

Exceções verificadas e não verificadas

Como mostrado na hierarquia de exceções em Java, sabemos que todas elas, direta ou indiretamente, são subclasses de `Exception`. Mesmo assim, temos dois tipos de exceção:

Exceções verificadas	Exceções não verificadas
São exceções cujas classes são subclasses de <code>Exception</code> , em qualquer nível, não passando pela classe <code>RuntimeException</code> . Elas são verificadas por serem tratadas de forma diferente pelo JVM, já que representam exceções mais difíceis de ser contornadas pelo programa.	São exceções cujas classes são subclasses de <code>RuntimeException</code> , apesar de esta ser subclasse de <code>Exception</code> . Esse grupo representa as exceções cujas ações podem ser contornadas pelo programa.

No caso das exceções verificadas, os métodos que as lançam, obrigatoriamente:

- Ou devem ser colocados em um bloco `try...catch`.
- Ou devem declarar que lançam exceção verificada.

Nos exemplos usados nas aulas anteriores, tratamos das exceções não verificadas. Veja o exemplo mostrado no quadro 16.

Quadro 16: Classe TesteExcecoes02 reescrita, que lança exceção não verificada

```

1 package com.ead.programas;
2
3 import com.ead.classes.Pessoa;
4
5 public class TesteExcecoes02 {
6     public static void main(String[] args) {
7         String resultado = Pessoa.exibirMaioridade(-10);
8         System.out.println(resultado);
9     }
10 }
```

Apesar de o método `exibirMaioridade()` lançar uma exceção do tipo `IllegalArgumentException`, nós protegemos o código por questões de melhor interação com o usuário, mas observe que é permitido que ele seja executado fora do bloco protegido.

Consideremos, agora, um método da classe `FileReader`, responsável por abrir um arquivo texto, mostrado no quadro 17 (trataremos de arquivos texto mais à frente nesta disciplina).

Quadro 17: Classe Arquivos contendo métodos que lançam exceções verificadas

```

1 package com.ead.classes;
2
3 import java.io.FileReader;
4
5 public class Arquivos {
6     public static void lerArquivo(String nomeArquivo){
7         FileReader reader = new FileReader(nomeArquivo);
8         //procedimento para leitura do arquivo
9         reader.close();
10    }
11 }
```

Verifique, no quadro 17, que estão ocorrendo erros tanto no construtor da classe como no método `close()`. Isso porque o construtor lança uma exceção do tipo `FileNotFoundException`, e o método lança outra do tipo `IOException`, que são exceções verificadas.

Como visto, podemos resolver esse problema de duas formas, como mostrado nos quadros 18 e 19.

Quadro 18: Protegendo o código como exceções verificadas

```

1 package com.ead.classes;
2
3 import java.io.FileNotFoundException;
4 import java.io.FileReader;
5 import java.io.IOException;
6
7 public class Arquivos {
8     public static void lerArquivo(String nomeArquivo){
9         try {
10             FileReader reader = new FileReader(nomeArquivo);
11             //procedimento para leitura do arquivo
12             reader.close();
13         } catch (FileNotFoundException e) {
14             // TODO Auto-generated catch block
15             e.printStackTrace();
16         } catch (IOException e) {
17             // TODO Auto-generated catch block
18             e.printStackTrace();
19         }
20     }
21 }
```

Quadro 19: Declarando o método *main* lançando uma exceção verificada

```

1 package com.ead.classes;
2 import java.io.FileNotFoundException;
3 import java.io.FileReader;
4 import java.io.IOException;
5
6 public class Arquivos {
7     public static void lerArquivo(String nomeArquivo) throws FileNotFoundException,
8         IOException{
9         FileReader reader = new FileReader(nomeArquivo);
10        //procedimento para leitura do arquivo
11        reader.close();
12    }
13 }
14 }
```

Declaramos que o método lança exceção verificada usando a palavra *throws* após a lista de parâmetros do método *lerArquivo()*, seguido das classes envolvidas na exceção.

Outra forma é simplificar as exceções declaradas no método por uma superclasse, como no quadro 20.

Quadro 20: Método *main()* lançando a exceção generalizada *Exception*

```

1 package com.ead.classes;
2 import java.io.FileReader;
3
4 public class Arquivos {
5     public static void lerArquivo(String nomeArquivo) throws Exception {
6         FileReader reader = new FileReader(nomeArquivo);
7         //procedimento para leitura do arquivo
8         reader.close();
9     }
10 }
11 }
```

Vale ressaltar que, em todos os lugares em que esse método for chamado, ou ele deve estar em um bloco protegido, ou o método que o executar também deve declarar que lança uma exceção verificada compatível.

O bloco *finally*

Além do tradicional bloco *try...catch*, podemos ter também o bloco *finally*. Ele é executado sempre, independentemente do que ocorrer no bloco *try* ou no bloco *catch()*. Geralmente, é usado para liberar recursos consumidos pelo programa, por exemplo, fechar a conexão com um banco de dados que foi aberta inicialmente, fechar um arquivo aberto para edição etc.

O exemplo do quadro 21 ilustra a utilização do bloco *finally*.

Quadro 21: Utilização do bloco *catch* com o bloco *finally*

```

1 package com.ead.classes;
2 import java.io.FileReader;
3
4 public class Arquivos {
5     public static void lerArquivo(String nomeArquivo) throws Exception {
6         FileReader reader = null;
7         try {
8             reader = new FileReader(nomeArquivo);
9         } catch (Exception e) {
10             // tratamento do erro
11         } finally {
12             reader.close();
13         }
14     }
15 }
```

Para encerrar nossa abordagem sobre tratamento de exceções, podemos usar o bloco *try* juntamente com o bloco *finally*, dispensando, assim, o uso do bloco *catch()*. No exemplo do quadro 21, se não desejamos tomar nenhuma providência no bloco *catch()*, podemos omiti-lo, como é mostrado no quadro 22.

Quadro 22: Utilização do bloco *finally* sem o bloco *catch*

```

1 package com.ead.classes;
2 import java.io.FileReader;
3
4 public class Arquivos {
5     public static void lerArquivo(String nomeArquivo) throws Exception {
6         FileReader reader = null;
7         try {
8             reader = new FileReader(nomeArquivo);
9         } finally {
10             reader.close();
11         }
12     }
13 }
```

Resumindo esse assunto, podemos ter uma das situações apresentadas no quadro 23.

Quadro 23: Modelos de utilização de blocos protegidos

<code>try { //código } catch(Exception e) { // exceção } finally { // finalização }</code>	<code>try { //código } catch(Exception e) { // exceção }</code>	<code>try { //código } catch(Exception e) { } finally { // finalização }</code>
--	---	---

Exercícios do Módulo 1

Exercício 1: Calculadora

Escrever um programa contendo um método que receba três parâmetros:

- O primeiro operando (*string*).
- O segundo operando (*string*).
- A operação (*int*).

Com essas informações, o método deve retornar o resultado.

Esse código deve ser escrito dentro de um bloco *try...catch*, prevendo as seguintes exceções:

- *NumberFormatException* – para as conversões de texto para o tipo *int*.
- *ArithmaticException* – quando ocorrer divisão por zero.

O método deverá lançar essas exceções, quando surgirem.

Escrever um programa contendo o método *main()* que teste o método que você criou. Esse método também deve ter um bloco *try...catch* para capturar a exceção e apresentar uma mensagem adequada para o usuário.

Exercício 2: Cálculo da média das notas de um aluno

Um aluno possui quatro notas na disciplina Java Avançado. Escrever um programa que solicite essas notas e as armazene em um *array* de variáveis do tipo *double*. O programa deverá possuir uma estrutura de repetição com um índice iniciando em 0 e terminado em 4. Como não existe o índice 4, o programa lançará uma exceção do tipo *ArrayIndexOutOfBoundsException*. Capturar essa exceção em um bloco *try...catch...finally*. No bloco *finally*, calcular a média e, em seguida, apresentá-la na tela.

Módulo 2 - Coleções

Aula 4 - Visão geral das coleções

Características das coleções

É muito comum encontrarmos elementos dispostos em coleções, que são estruturas de dados pré-empacotadas. Por exemplo, uma sala de aula possui uma coleção de alunos, um estacionamento possui uma coleção de automóveis, uma empresa possui uma coleção de funcionários, e assim por diante.

As coleções possuem características peculiares, que definem certas diferenças entre elas:

- Em uma biblioteca, podem existir livros idênticos, ou seja, exemplares repetidos.
- Em uma sala de aula, não podem existir alunos idênticos, ou seja, alunos não podem se repetir na mesma sala.

Essas peculiaridades também se aplicam às coleções em Java. Analise a figura 2.

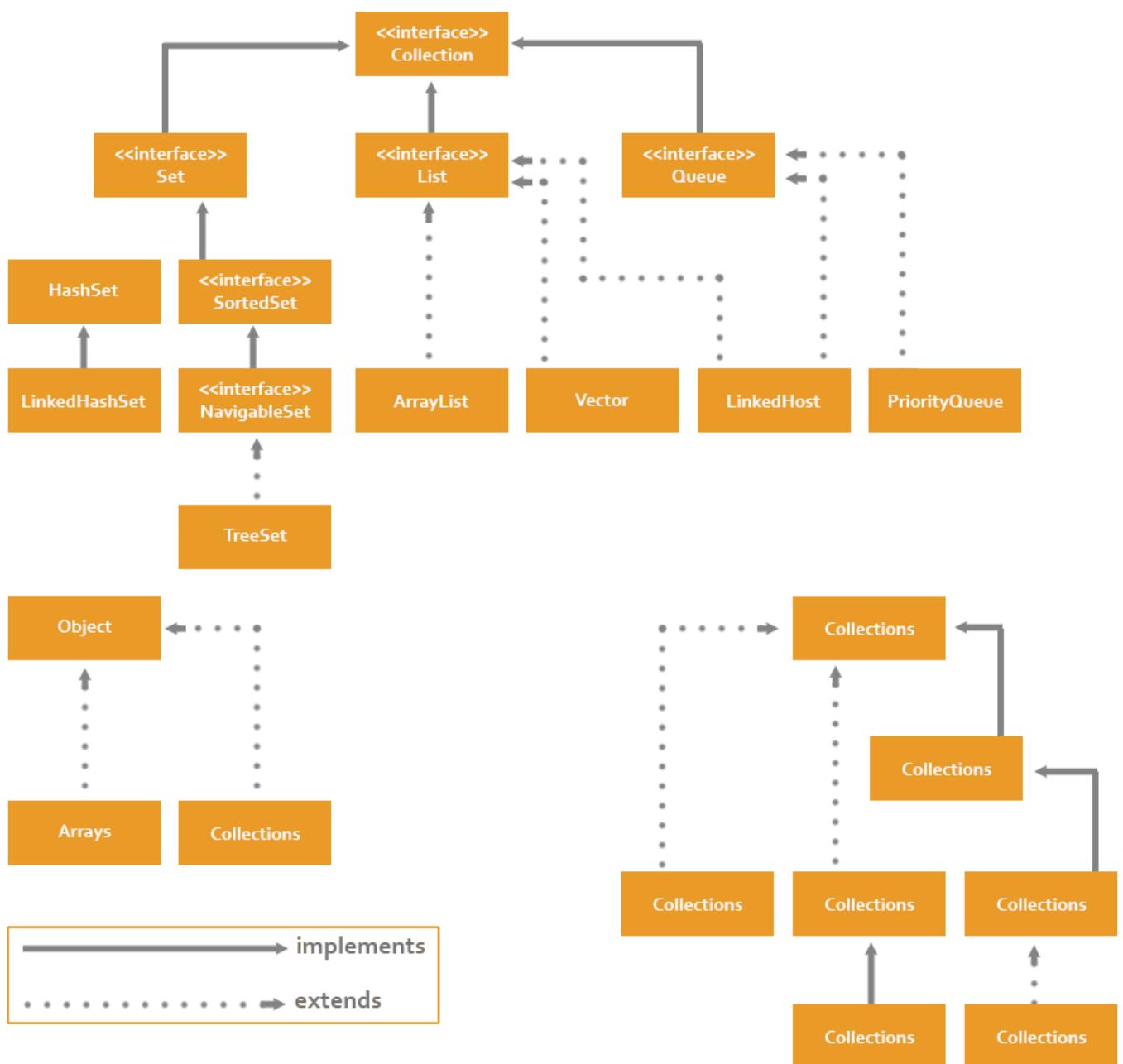


Figura 2 – Hierarquia das coleções em Java. Fonte: Sierra e Bates (2008).

Na figura, podemos ver que *Collection* é a interface estendida pelas interfaces *List*, *Set* e *Queue*. Além disso, temos outra interface em outra hierarquia, a interface *Map*. Cada uma delas apresenta as peculiaridades que comentamos anteriormente.

Neste curso, abordaremos as interfaces *List*, *Set* e *Map*, com as principais classes que as implementam. Para entender suas características, vamos aprender alguns conceitos preliminares:

- **Índice**: número que indica a posição de um elemento na coleção. Por exemplo, se meu nome for o primeiro em uma lista, seu índice é 0 (zero). Isso mesmo! O primeiro índice de uma coleção é zero.

- **Chave:** outra forma de indicar um elemento na coleção, mas não tem a ver com sua posição. Por exemplo, em uma lista de pessoas candidatas a um emprego, a localização de um candidato pode ser realizada por seu CPF.
- **Ordenação:** significa a ordem baseada na posição do elemento na coleção, tal como está. Se um funcionário chamado Manoel for o primeiro a ser inserido em uma coleção, pela ordenação, ele será o primeiro elemento da lista. Ordenação está relacionada à posição dos elementos na sua forma natural.
- **Classificação:** muitas vezes, a classificação confunde-se com a ordenação, mas não é a mesma coisa. Ela tem por finalidade mudar a ordem (ordenação) tomando como base alguma regra predefinida. Por exemplo, se uma empresa inclui seus funcionários em uma coleção, a ordem de inserção será a ordem natural. Posteriormente, a empresa decide listar esses funcionários em ordem alfabética. Essa é a regra estabelecida para alterar a ordem original.

Agora podemos definir as classes que implementam as três interfaces que estudaremos: List, Set e Map. Elas estão no pacote `java.util.*`, conforme consta na tabela 1.

Tabela 1 – Definição das interfaces List, Set e Map

Classe	Interface	Ordenada	Classificada
HashMap	Map	Não.	Não.
Hashtable		Não.	Não.
TreeMap		Sim.	Pela ordem natural ou customizada.
LinkedHashMap		Pela ordem de inserção ou último acesso.	Não.
HashSet	Set	Não.	Não.
TreeSet		Sim.	Pela ordem natural ou customizada.
LinkedHashSet		Pela ordem de inserção.	Não.
ArrayList	List	Pelo índice.	Não.
Vector		Pelo índice.	Não.
LinkedList		Pelo índice.	Não.
PriorityQueue		Sim.	Pela ordem de prioridade.

Sobre essas três interfaces, temos as seguintes informações, de forma sumarizada:

- **List:** permite inserção e remoção em qualquer posição, além de ordenação. Admite elementos repetidos.
- **Set:** permite inserção e remoção, mas não em qualquer posição. Não admite elementos repetidos.
- **Map:** utiliza conceito de chave-valor. Cada elemento dessa coleção é formado pelo par chave-valor e não apenas pelo valor.

A seguir, estudaremos cada uma delas, por meio de exemplos.

Aula 5 - Interface Collection e classe Collections

A interface List

A interface List é implementada pelas classes ArrayList, Vector e LinkedList.

- Os métodos da classe ArrayList não são sincronizados e, por isso, são mais rápidos. Atualmente, ela é preferida em relação à Vector.
- A classe LinkedList utiliza o conceito de lista ligada para tratar seus elementos.

Em relação às classes ArrayList/Vector e LinkedList, podemos apresentar as seguintes conclusões:

	Inserção / remoção	Busca
LinkedList	Rápida	Lenta
ArrayList / Vector	Lenta	Rápida

Essas classes, na verdade, fazem a mesma coisa. Com objetos delas, podemos manipular o que chamamos de listas, já que implementam a mesma interface. A classe Vector era predominante até a versão 1.3 do Java; na versão 1.4, foi incluída a ArrayList. A diferença é que os métodos da classe Vector são *thread-safe*, ou seja, são sincronizados (veremos o conceito de sincronismo e *thread-safe* no módulo sobre *multithreading*).

Agora vamos estudar alguns exemplos. Comecemos com o apresentado no quadro 24.

Quadro 24: Classe TesteListas, contendo a declaração de um objeto ArrayList

```

1 package com.ead.programas;
2
3 import java.util.ArrayList;
4
5 public class TesteListas {
6     public static void main(String[] args) {
7         ArrayList lista = new ArrayList();
8         lista.add("Manoel");
9         lista.add("James");
10        lista.add("Claudia");
11
12        for (int i = 0; i < lista.size(); i++) {
13            System.out.println(lista.get(i));
14        }
15    }
16}
```

No exemplo do quadro 24, apresentamos os métodos *add()*, *size()* e *get()*. O método *add()* recebe como parâmetro Object, significando que ele pode receber literalmente qualquer tipo de elemento. O método *size()* não recebe nenhum parâmetro e retorna o número de elementos da lista. O método *get()* recebe uma posição como parâmetro e retorna o elemento nela. As definições desses métodos são:

```
public boolean add(Object e);
public int size();
public Object get(int index);
```

Nessa versão do nosso exemplo, consideramos que a lista pode receber qualquer tipo de elemento. Analisemos, agora, o exemplo do quadro 25.

Quadro 25: Classe TesteListas reescrita, especificando o tipo *String* como elemento

```
1 package com.ead.programas;
2
3 import java.util.ArrayList;
4
5 public class TesteListas {
6     public static void main(String[] args) {
7         ArrayList lista = new ArrayList();
8         lista.add("Manoel");
9         lista.add("James");
10        lista.add("Claudia");
11
12        for (int i = 0; i < lista.size(); i++) {
13            String nome = (String)lista.get(i);
14            System.out.println(nome);
15        }
16    }
17 }
```

Observe, no quadro 25, que foi necessário realizar um *typecast* na linha 13. Isso foi necessário porque o método *get()* retorna *Object*, e seu retorno foi atribuído a uma *string*, que é subclasse de *Object*.

No exemplo anterior, do quadro 24, esse procedimento não foi necessário porque o método *println()* possui uma versão sobrecarregada que recebe *Object* como parâmetro.

A execução do programa do quadro 25 produz a saída já esperada:

```
<terminated> TesteListas [Java Application]
Manoel
James
Claudia
```

Agora, e se inserirmos um elemento na lista diferente de *string*? Veja o exemplo do quadro 26.

Quadro 26: Classe TesteListas reescrita, especificando elementos como *string* na saída, mas incluindo um elemento inteiro ao final da lista

```

1 package com.ead.programas;
2
3 import java.util.ArrayList;
4
5 public class TesteListas {
6     public static void main(String[] args) {
7         ArrayList lista = new ArrayList();
8         lista.add("Manoel");
9         lista.add("James");
10        lista.add("Claudia");
11        lista.add(1200);
12
13        for (int i = 0; i < lista.size(); i++) {
14            String nome = (String)lista.get(i);
15            System.out.println(nome);
16        }
17    }
18 }
```

A execução desse programa produz a saída:

```
<terminated> TesteListas [Java Application] C:\Program Files (x86)\Java\jre7\bin\javaw.exe (16/05/2014 19:32:20)
Manoel
James
Claudia
Exception in thread "main" java.lang.ClassCastException: java.lang.Integer cannot be cast to java.lang.String
at com.ead.programas.TesteListas.main(TesteListas.java:14)
```

Por que ocorreu esse erro? Foi justamente na linha 14, que faz um *typecast* para o tipo *String*, sendo que o valor armazenado na posição indicada é um inteiro.

A solução para esse problema é fazer uma verificação de tipos, se estivermos, por exemplo, interessados apenas nos elementos do tipo *String*. Veja a solução no quadro 27.

Quadro 27: Classe TesteListas reescrita

```

1 package com.ead.programas;
2
3 import java.util.ArrayList;
4
5 public class TesteListas {
6     public static void main(String[] args) {
7         ArrayList lista = new ArrayList();
8         lista.add("Manoel");
9         lista.add("James");
10        lista.add("Claudia");
11        lista.add(1200);
12
13        for (int i = 0; i < lista.size(); i++) {
14            if(lista.get(i) instanceof String){
15                String nome = (String)lista.get(i);
16                System.out.println(nome);
17            }
18        }
19    }
20 }
```

Antes da versão 5 do Java, o que o quadro 27 apresenta era o único recurso disponível. Essa abordagem exigia que a verificação fosse feita sempre que o elemento da lista tivesse de ser atribuído a um tipo específico, justamente para evitar a exceção do tipo ClassCastException, como vimos. Na versão 5, foi incluído o conceito de genéricos (tradicionalmente chamado de *generics*). As interfaces e classes que implementam a interface Collection passaram a ser genéricas. Com isso, especificamos o tipo do elemento no momento da criação do objeto. Analisemos o exemplo do quadro 28.

Quadro 28: Classe TesteListasGeneric, declarando um objeto *ArrayList* parametrizado

```

1 package com.ead.programas;
2
3 import java.util.ArrayList;
4
5 public class TesteListasGeneric {
6     public static void main(String[] args) {
7         ArrayList<String> lista = new ArrayList<String>();
8         lista.add("Manoel");
9         lista.add("James");
10        lista.add("Claudia");
11
12        for (int i = 0; i < lista.size(); i++) {
13            String nome = lista.get(i);
14            System.out.println(nome);
15        }
16    }
17 }
```

Qualquer tentativa de inserirmos algum elemento com tipo diferente de *string* causará um erro de compilação, não mais uma exceção. Veja a utilização dos caracteres “<>” na declaração e na instanciação da classe (linha 7 do quadro 28).

Daqui para a frente, todas as coleções que utilizarmos serão genéricas.

- **Removendo elementos**

Podemos remover elementos de duas formas, ambas indicadas no exemplo do quadro 29.

Quadro 29: Classe TesteListasGeneric reescrita, com um elemento sendo removido

```

1 package com.ead.programas;
2
3 import java.util.ArrayList;
4
5 public class TesteListasGeneric {
6     public static void main(String[] args) {
7         ArrayList<String> lista = new ArrayList<String>();
8         lista.add("Manoel");
9         lista.add("James");
10        lista.add("Claudia");
11
12        for (int i = 0; i < lista.size(); i++) {
13            String nome = lista.get(i);
14            System.out.println(nome);
15        }
16        System.out.println("-----");
17        lista.remove(0); //remove Manoel da lista
18        lista.remove("Claudia"); //remove a String "Claudia" da lista
19    }
20 }
```

Na linha 17, removemos o elemento informando o índice, e, na linha 18, informamos o elemento que desejamos remover.

- **Ordenando elementos**

É possível, também, classificar os elementos de uma lista por meio de sua ordenação. A classificação é realizada pela ordem natural dos elementos. A seguir, apresentamos a ordem natural de alguns deles:

- String: a ordem natural é alfabética.
- **Números** (int, double etc.): a ordem natural é numérica.

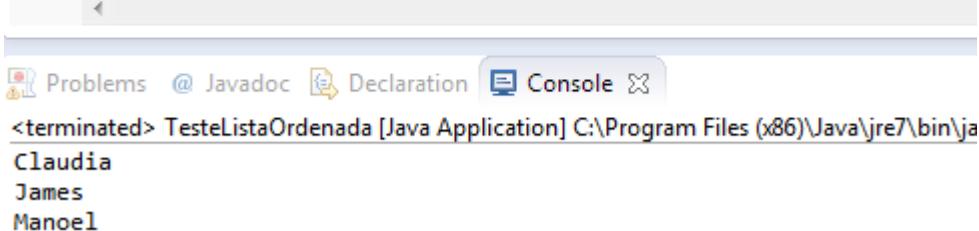
A ordenação é realizada por intermédio do método `sort()`, da classe Collections. Atenção: a classe Collections é diferente da interface Collection. Observe o nome:

Collections.sort(lista)

Vamos a um exemplo, mostrado no quadro 30.

Quadro 30: Classe TesteListasOrdenada, com uma instrução para ordenar a lista e sua saída

```
1 package com.ead.programas;
2
3 import java.util.ArrayList;
4 import java.util.Collections;
5
6 public class TesteListaOrdenada {
7     public static void main(String[] args) {
8         ArrayList<String> lista = new ArrayList<String>();
9         lista.add("Manoel");
10        lista.add("James");
11        lista.add("Claudia");
12
13        Collections.sort(lista);
14
15        for (int i = 0; i < lista.size(); i++) {
16            String nome = lista.get(i);
17            System.out.println(nome);
18        }
19    }
20}
21
```



The screenshot shows an IDE interface with several tabs at the top: Problems, Javadoc, Declaration, and Console. The Console tab is active, displaying the output of a Java application named 'TesteListaOrdenada'. The output shows three names printed one per line: 'Claudia', 'James', and 'Manoel'.

```
Problems @ Javadoc Declaration Console
<terminated> TesteListaOrdenada [Java Application] C:\Program Files (x86)\Java\jre7\bin\ja
Claudia
James
Manoel
```

Agora, vamos a um exemplo diferente. Teremos uma classe chamada Pessoa, e o objetivo é incluir elementos do tipo Pessoa em uma lista. A solução é mostrada no quadro 31.

Quadro 31: Classe Pessoa

```

1 package com.ead.classes;
2
3 public class Pessoa {
4     private int codigo;
5     private String nome;
6
7     public Pessoa(int codigo, String nome){
8         setCodigo(codigo);
9         setNome(nome);
10    }
11
12    public int getCodigo() {
13        return codigo;
14    }
15    public void setCodigo(int codigo) {
16        this.codigo = codigo;
17    }
18    public String getNome() {
19        return nome;
20    }
21    public void setNome(String nome) {
22        this.nome = nome;
23    }
24}
--
```

Vamos criar um programa que permita inserir objetos da classe Pessoa em uma lista, como mostrado no quadro 32.

Quadro 32: Classe TesteListaPessoas declarando um objeto *ArrayList* parametrizado para Pessoa

```

1 package com.ead.programas;
2
3 import java.util.ArrayList;
4
5 import com.ead.classes.Pessoa;
6
7 public class TesteListaPessoas {
8     public static void main(String[] args) {
9         ArrayList<Pessoa> pessoas = new ArrayList<Pessoa>();
10        pessoas.add(new Pessoa(212, "Eusebio"));
11        pessoas.add(new Pessoa(154, "Matheus"));
12        pessoas.add(new Pessoa(421, "Ana"));
13
14        for (Pessoa pessoa : pessoas) {
15            System.out.println(pessoa);
16        }
17    }
18}
```

A execução do programa do quadro 32 produz o seguinte resultado:

```
<terminated> TesteListaPessoas [Java Application]!
com.ead.classes.Pessoa@1186fab
com.ead.classes.Pessoa@14b7453
com.ead.classes.Pessoa@c21495
```

Parece estranho, não? O problema aqui é que estamos usando a instrução

```
System.out.println(pessoa);
```

que imprime o objeto *Pessoa*, e não um de seus atributos!

Como podemos resolver esse problema? É simples. Toda classe possui um método chamado *toString()*, que é uma representação *string* do objeto. Esse método é definido na classe Object e deve ser sobreescrito nas subclasses se desejarmos criar uma representação personalizada em forma de *string* de um objeto. A implementação padrão desse método é a exibição do nome da classe seguido do endereço de memória do objeto. Sendo assim, precisamos implementar o método *toString()* na classe *Pessoa*. A solução está no quadro 33.

Quadro 33: Classe *Pessoa* reescrita, sobrepondo o método *toString()*

```

1 package com.ead.classes;
2
3 public class Pessoa {
4     private int codigo;
5     private String nome;
6
7     public Pessoa(int codigo, String nome){
8         setCodigo(codigo);
9         setNome(nome);
10    }
11
12    public int getCodigo() {
13        return codigo;
14    }
15    public void setCodigo(int codigo) {
16        this.codigo = codigo;
17    }
18    public String getNome() {
19        return nome;
20    }
21    public void setNome(String nome) {
22        this.nome = nome;
23    }
24
25    public String toString(){
26        return "[" +codigo + ", " + nome + "]";
27    }
28 }
```

Observe, no quadro 33, que escolhemos a representação de uma pessoa da seguinte forma: “[*codigo, nome*]”.

A execução do programa, agora, produz a seguinte saída:

```
<terminated> TesteListaPessoas [Java Application]
[212, Eusebio]
[154, Matheus]
[421, Ana]
```

A chamada ao método `toString()` é implícita, não sendo necessário chamar explicitamente o método, a menos que o objetivo seja atribuí-lo a uma *string*.

A interface Set

Diferentemente das listas, a *interface Set* define o que chamamos de conjuntos. Estes, como principal característica, não permitem elementos duplicados.

Veja o exemplo apresentado no quadro 34.

Quadro 34: Classe TesteConjuntos. Inclusão de elementos no *HashSet* verificando se incluiu

```

1 package com.ead.programas;
2
3 import java.util.HashSet;
4
5 public class TesteConjuntos {
6     public static void main(String[] args) {
7         HashSet<String> nomes = new HashSet<String>();
8         System.out.println(nomes.add("Emanoel"));
9         System.out.println(nomes.add("Bernardo"));
10        System.out.println(nomes.add("Carlos"));
11        System.out.println(nomes.add("Benedito"));
12        System.out.println(nomes.add("Emanoel"));
13    }
14 }
```

Observe que o nome "Emanoel" foi inserido duas vezes. Vamos ver o resultado?

```

<terminated> TesteConjuntos [Java Application] !
true
true
true
true
false
```

A tentativa de inclusão de um elemento repetido causou falha na execução do método, razão pela qual ele retornou *false*.

Se desejarmos colocar os elementos dessa lista em ordem, não fazemos da mesma forma que em *List*, uma vez que os elementos de um conjunto (*Set*) não permitem tal ordenação. É necessário usar um mecanismo que permita inserir os elementos já ordenados. Isso é feito pela classe *TreeSet*, apresentada no quadro 35.

Quadro 35: Classe TesteConjuntos02, definindo um objeto *TreeSet*

```

1 package com.ead.programas;
2
3 import java.util.TreeSet;
4
5 public class TesteConjuntos02 {
6     public static void main(String[] args) {
7         TreeSet<String> nomes = new TreeSet<String>();
8         System.out.println(nomes.add("Emanoel"));
9         System.out.println(nomes.add("Bernardo"));
10        System.out.println(nomes.add("Carlos"));
11        System.out.println(nomes.add("Benedito"));
12        System.out.println(nomes.add("Emanoel"));
13        System.out.println("-----");
14
15        for (String nome : nomes) {
16            System.out.println(nome);
17        }
18    }
19 }
```

O resultado é exibido a seguir. Observe que os nomes foram exibidos em ordem alfabética.

```
<terminated> TesteConjuntos02 [Java Application]
true
true
true
true
false
-----
Benedito
Bernardo
Carlos
Emanoel
```

Para remover um elemento de um conjunto, temos disponível apenas o método *remove()*, que recebe como parâmetro o objeto a ser removido.

A interface Map

As coleções do tipo Map consideram, como já foi dito, elementos constituídos por uma chave e um valor. O valor pode ser repetido, mas a chave deve ser exclusiva. No exemplo do quadro 36, a seguir, incluímos uma lista de alunos contendo o RA (registro acadêmico) como chave e o nome como valor.

Quadro 36: Classe TesteMapas, incluindo elementos em uma coleção *HashMap*

```

1 package com.ead.programas;
2
3 import java.util.HashMap;
4 import java.util.Map;
5
6 public class TesteMapas {
7     public static void main(String[] args) {
8         HashMap<Integer, String> alunos = new HashMap<Integer, String>();
9         alunos.put(12547, "José Carlos");
10        alunos.put(36279, "Antonio");
11        alunos.put(92478, "Daniela Mendes");
12        alunos.put(38547, "Denis Nunes");
13
14        for(Map.Entry<Integer, String> aluno: alunos.entrySet()){
15            System.out.println(aluno.getKey() + " - " + aluno.getValue());
16        }
17    }
18 }
```

Uma atenção especial deve ser dada ao código das linhas 14 a 16. Na estrutura de repetição desse código, percorremos a coleção levando em conta que cada elemento é composto de duas partes. Podemos fazer uma comparação com um dicionário, em que temos palavras e seus significados. A declaração *Map.Entry<Integer, String> aluno* declara um elemento desse dicionário, considerando as parametrizações da classe genérica. A chave de uma coleção do tipo Map é, na verdade, um conjunto baseado em Set.

- **Buscando elementos em um Map**

Podemos realizar a busca de elementos em um Map de forma semelhante às listas ou aos conjuntos. Analise o código do quadro 37.

Quadro 37: Classe TesteMapas reescrita, com uma busca de elemento pela chave

```

1 package com.ead.programas;
2
3 import java.util.HashMap;
4 import java.util.Map;
5
6 public class TesteMapas {
7     public static void main(String[] args) {
8         HashMap<Integer, String> alunos = new HashMap<Integer, String>();
9         alunos.put(12547, "José Carlos");
10        alunos.put(36279, "Antonio");
11        alunos.put(92478, "Daniela Mendes");
12        alunos.put(38547, "Denis Nunes");
13
14        for(Map.Entry<Integer, String> aluno: alunos.entrySet()){
15            System.out.println(aluno.getKey() + " - " + aluno.getValue());
16        }
17
18        if(alunos.containsKey(12547)){
19            String aluno = alunos.get(12547);
20            //restante do código
21        }
22    }
23 }
```

O método `containsKey()` recebe como parâmetro um `Object` indicando a chave desejada e retorna `true` se for encontrado um elemento com ela. O método `get()` recebe como parâmetro a chave e retorna o elemento com esta.

Aula 6 - Classe Arrays

Definição da classe Arrays

A classe `Arrays` é muito útil quando desejamos manipular `arrays`. Ela é uma das classes disponíveis na API do Java.

Como exemplo, vamos ordenar e buscar elementos em um `array`. Neste exemplo, iremos usar o método `sort()` e o método `binarySearch()`, que faz uma busca por um determinado elemento e retorna sua localização no `array`.

Inicialmente, definimos um `array` de inteiros quaisquer. Para exibir os elementos em forma de `string`, usamos o método `Arrays.toString(array);`, que recebe um `array` como parâmetro.

Depois, vamos usar o método `sort()`, que ordena o `array` em ordem crescente, desta forma: `Arrays.sort(array);`

Depois, mostramos o `array` novamente. Em seguida, vamos armazenar em um inteiro, `posicao`, a posição do elemento “2112” no `array`, desta forma:

Arrays.binarySearch(array, numero_que_estamos_procurando);

IMPORTANTE: o método `Arrays.binarySearch` só funciona se o `array` estiver na ordem crescente! Ou seja, só use `binarySearch()` após usar `sort()`, pois ele usa uma busca inteligente.



Vamos ao exemplo apresentado no quadro 38.

Quadro 38: Classe TesteArrays com o resultado da execução

```

1 package com.ead.programas;
2
3 import java.util.Arrays;
4
5 public class TesteArrays {
6     public static void main(String[] args) {
7         int[] numeros = {1, 4, 0, -13, 2112, 14, 17};
8         int posicao;
9
10        System.out.println("Os elementos do array são: " + Arrays.toString(numeros));
11        System.out.println("Ordenando...");
12
13        Arrays.sort(numeros);
14
15        System.out.println("Array ordenado: " + Arrays.toString(numeros));
16        posicao = Arrays.binarySearch(numeros, 2112);
17
18        System.out.println("Posição do elemento '2112': " + posicao);
19    }
20
21

```

Problems @ Javadoc Declaration Console

<terminated> TesteArrays [Java Application] C:\Program Files (x86)\Java\jre7\bin\javaw.exe (16/05/2014 21:30:28)

Os elementos do array são: [1, 4, 0, -13, 2112, 14, 17]

Ordenando...

Array ordenado: [-13, 0, 1, 4, 14, 17, 2112]

Posição do elemento '2112': 6

É importante consultar a API do Java para conhecer e utilizar os métodos disponíveis para as coleções.

Exercícios do Módulo 2

Exercício 1: Lista de alunos

- Criar uma classe chamada Aluno, contendo os atributos:
 - RM.
 - Nome.
 - Curso.
- Nessa classe, incluir os métodos *getters* e *setters* (usar o recurso do Eclipse para essa operação).
- Incluir, na classe Aluno, um construtor que receba como parâmetros os três atributos da classe.
- Na classe, sobrescrever o método *toString*, de forma a representar um objeto da classe Aluno no seguinte formato: "[rm – nome – curso]".
- Agora, criar uma classe contendo o método *main()* que crie uma lista de cinco alunos, ou seja, um *ArrayList* contendo cinco objetos da classe Aluno.
- Em uma estrutura de repetição, exibir os dados dos alunos na tela, aproveitando o método *toString()* que se sobrescreveu.

Exercício 2: Classificação da lista de alunos

- a. Alterar a classe Aluno do exercício 1, de forma que ela implemente a interface Comparable<> e, consequentemente, sobrescreva o método *compareTo()* dessa classe. Esse método deve retornar um inteiro, e os possíveis valores são:
 - 1: indica que o conjunto de elementos está em ordem.
 - 0: indica que os elementos são iguais.
 - -1: indica que os elementos não estão em ordem.
- b. Levar em conta que os alunos são classificados pelo nome.
- c. Alterar também o programa do exercício 1 de forma que o método *main()* execute a instrução:
Collections.sort(alunos);
- d. Exibir os alunos, agora classificados pelo nome.

Exercício 3: Criando um Map a partir da lista de alunos

- a. Escrever um método que receba como parâmetro uma lista de alunos e retorne uma coleção Map contendo como chave o *RM* do aluno e como valor o *nome* do aluno.
- b. Alterar, novamente, o método *main()* do programa do exercício 2, de forma a executar o método criado no item a.
- c. Exibir na tela os alunos a partir da coleção Map.

Módulo 3 - Documentação API do Java

Aula 7 - Conceitos da API do Java

Utilização da API do Java

A API do Java é a documentação oficial da linguagem. No passado, era comum a API de uma determinada linguagem ser disponibilizada de forma impressa, mas esse conceito mudou bastante. No caso do Java, a cada nova versão, um grupo de classes e/ou métodos é inserido ou alterado. Nesse contexto, é importante consultar a API para obtermos as definições atualizadas.

A API pode ser obtida no *link*: <http://docs.oracle.com/javase/7/docs/api/>.

Para a confecção deste material, a tela obtida quando acessamos esse *link* produz a interface apresentada na figura 3. Você deve estar atento às novas atualizações.

Package	Description
java.applet	Provides the classes necessary to create an applet and the classes an applet uses to communicate with its applet context.
java.awt	Contains all of the classes for creating user interfaces and for painting graphics and images.
java.awt.color	Provides classes for color spaces.
java.awt.datatransfer	Provides interfaces and classes for transferring data between and within applications.
java.awt.dnd	Drag and Drop is a direct manipulation gesture found in many Graphical User Interface systems that provides a mechanism to transfer information between two entities logically associated with presentation elements in the GUI.
java.awt.event	Provides interfaces and classes for dealing with different types of events fired by AWT components.
java.awt.font	Provides classes and interface relating to fonts.
java.awt.geom	Provides the Java 2D classes for defining and performing operations on objects related to two-dimensional geometry.
java.awt.im	Provides classes and interfaces for the input method framework.
java.awt.im.spi	Provides interfaces that enable the development of input methods that can be used with any Java runtime environment.
java.awt.image	Provides classes for creating and modifying images.
java.awt.image.renderable	Provides classes and interfaces for producing rendering-independent images.
java.awt.print	Provides classes and interfaces for a general printing API.
java.beans	Contains classes related to developing beans -- components based on the JavaBeans™ architecture.
java.beans.beancontext	Provides classes and interfaces relating to bean context.

Figura 3 - Tela de especificações de API. Fonte: <http://docs.oracle.com/javase/7/docs/api/>.

Utilizamos a API do Java sempre que desenvolvemos uma aplicação e não estamos interessados em reinventar a roda. Seria isso necessário?

Vamos considerar o seguinte cenário para ilustrar a utilização da API: desejamos usar a classe JOptionPane, mas não sabemos qual o seu pacote. O que devemos fazer?

Na aba correspondente aos pacotes, existe um *link* escrito: "All Classes". Selecionado-o, temos a lista de todas as classes no painel esquerdo inferior. Nele, selecionamos a classe desejada, como mostra a figura 4.

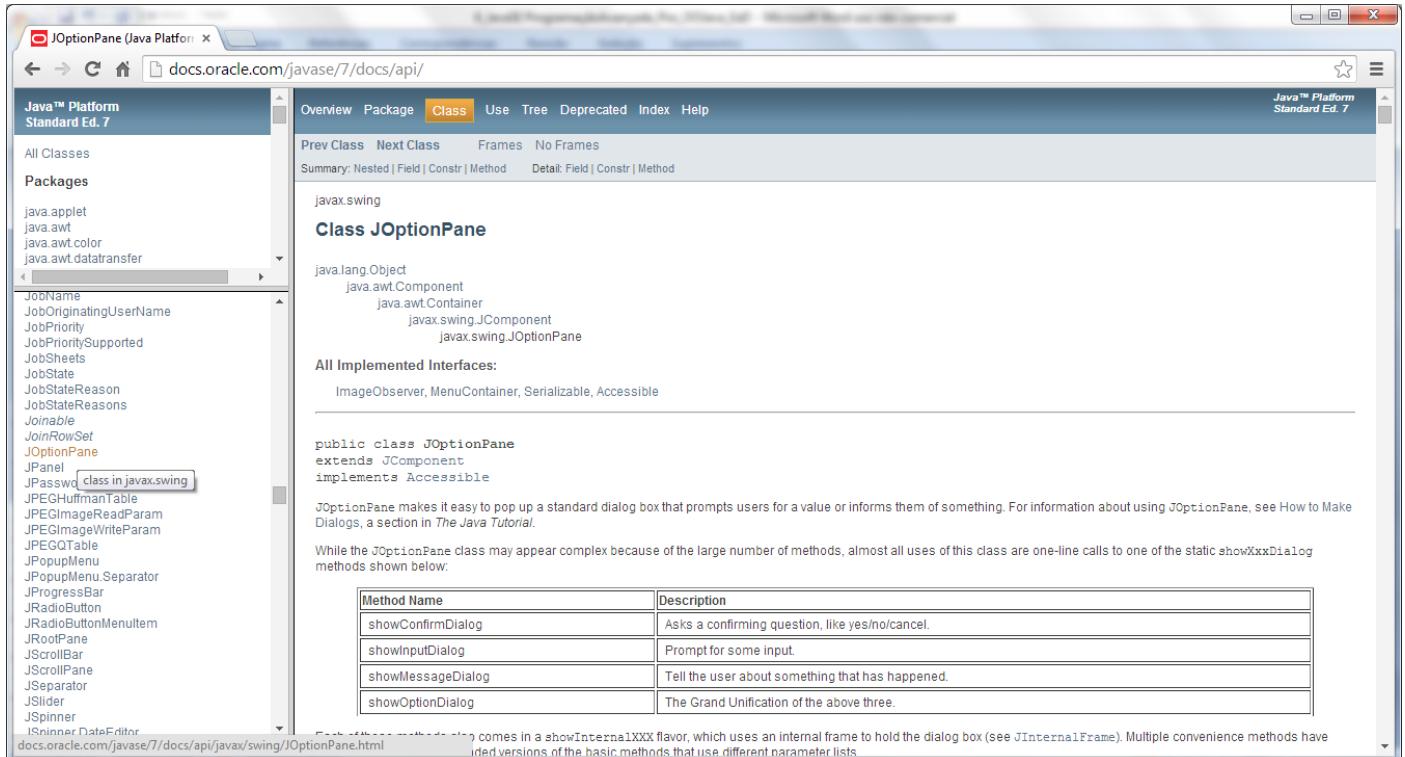


Figura 4 – A classe JOptionPane mostrada na API. Fonte: <http://docs.oracle.com/javase/7/docs/api/>.

Na definição da classe, temos todos os métodos documentados. Basta selecionar o desejado e implementá-lo. Por exemplo, queremos implementar o método *showMessageDialog()*. Vamos ver como fazer isso? Localizamos o método desejado, como mostra a figura 5.



Figura 5 – Demonstração dos métodos da classe JOptionPane. Fonte: <http://docs.oracle.com/javase/7/docs/api/>.

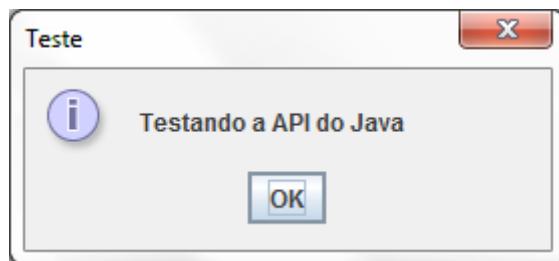
Na figura, vemos que existem três versões sobrecarregadas. Escolhemos a que melhor nos atende. Por exemplo, da forma mostrada no quadro 39.

Quadro 39: Classe TesteAPI

```

1 package com.ead.programas;
2
3 import javax.swing.JOptionPane;
4
5 public class TesteAPI {
6     public static void main(String[] args) {
7         JOptionPane.showMessageDialog(null, "Testando a API do Java", "Teste",
8             JOptionPane.INFORMATION_MESSAGE);
9     }
10 }
11 }
```

A execução produz o resultado:



Exercícios do Módulo 3

Exercício 1: Pesquisa por classes e métodos

- Entrar na API do Java, na internet, pelo link: <http://docs.oracle.com/javase/7/docs/api/>.
- Pesquisar pelas seguintes classes e métodos:
 - Classe Random: métodos *nextDouble()*, *nextBytes()*.
 - Classe JOptionPane: método *showInputDialog()* (verificar o método, que possui sete parâmetros).
 - Classe JOptionPane: constantes.
- Com base na sua pesquisa, criar uma aplicação de sua escolha que utilize os métodos pesquisados.

Módulo 4 - Multithreading

Aula 8 – Criação e execução de threads

Definição de threads

Threads permitem que um programa simples possa executar várias tarefas diferentes ao mesmo tempo, independentemente umas das outras.

A linguagem Java suporta a execução paralela de código por meio do *multithreading*, em que uma *thread* é um caminho de execução independente que está apto para rodar simultaneamente com outras *threads*.

Um programa Java, quando iniciado, roda em uma *thread* criada pela JVM (a *thread* principal - *main*) e torna-se *multithread* pela criação de *threads* adicionais.

Thread *safety* é um conceito existente no contexto de programas *multithread*, em que um trecho de código é *thread-safe* se ele funcionar corretamente durante execução simultânea por várias *threads*.

Se o código não for *thread-safe* e estiver rodando em um ambiente *multithread*, os resultados obtidos podem ser inesperados ou incorretos, pois uma *thread* pode desfazer o que a outra já realizou.

Esse é o problema clássico da atualização do contador em uma aplicação *multithread*. Para resolver esse problema, devemos usar o sincronismo, que vai bloquear os recursos até que a *thread* que os está usando acabe de processá-los.

Vamos ver nos próximos tópicos como isso funciona.

Criação de threads

A criação de uma *thread* pode ser realizada de duas formas:

- Criando uma classe que estende a classe *Thread*.
- Criando uma classe que implementa a interface *Runnable*.

A forma recomendada é implementar a interface *Runnable*, pois, quando estendemos uma classe, não podemos fazê-lo com nenhuma outra, já que não existe herança múltipla em Java.

Mas vamos apresentar exemplos dos dois casos. O primeiro exemplo estenderá a classe *Thread*, e os demais implementarão a interface *Runnable*.

O quadro 40 traz o primeiro exemplo.

Quadro 40: Classe ExemploThread estendendo a classe Thread

```

1 package com.ead.threads;
2
3 public class ExemploThread extends Thread{
4     public void run(){
5         try {
6             for (int i = 0; i < 20; i++) {
7                 System.out.print("y");
8                 Thread.sleep(200);
9             }
10        } catch (Exception e) {
11            // tratamento da exceção
12        }
13    }
14}
```

Observe que essa classe possui um método chamado *run()*. Esse método é executado pela *thread* no momento em que é iniciada. Nós não o executamos por nossa conta!

O código que desejamos executar pela *thread* deve ser escrito dentro desse método. Mas como fazemos para executá-lo? Vamos ver o programa que o utiliza no quadro 41.

Quadro 41: Classe TesteThread definindo uma nova Thread

```

1 package com.ead.threads;
2
3 public class TesteThread {
4     public static void main(String[] args) {
5         ExemploThread thread = new ExemploThread();
6         thread.start();
7         try {
8             for (int i = 0; i < 20; i++) {
9                 System.out.print("x");
10                Thread.sleep(200);
11            }
12        } catch (Exception e) {
13            // tratamento da exceção
14        }
15    }
16}
```

Instanciamos a classe *ExemploThread* e, a partir dessa instância, executamos o método *start()*. É ele quem inicia a *thread* e, quando estiver pronto, seu método *run()* é executado.

Observe que tanto no método *run()* da classe *ExemploThread* como no método *main()* existe uma estrutura de repetição que imprime um caractere na tela. No momento em que o *thread* é iniciado, ele passa a ser executado de forma independente, e as duas estruturas de repetição são executadas assim. Para visualizar o resultado, incluímos um tempo de 200 milissegundos entre cada caractere a ser mostrado.

A execução do programa do quadro 41 traz o resultado:

```
<terminated> TesteThread [Java Application] C:\Program Fi
xyyxyxxyxxyxxyxxyxxyxxyxxyxxyxxyxxyxxyxxyxxyx
```

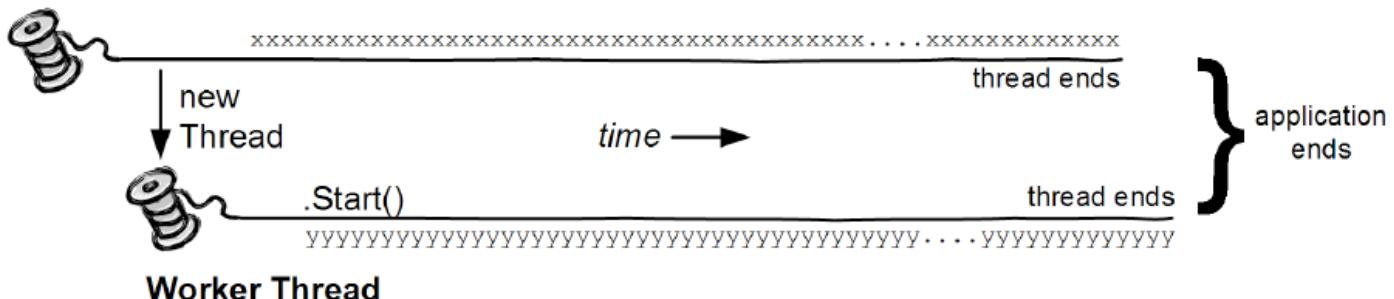
A impressão dos caracteres "x" e "y" ocorre de forma desordenada, pois não temos controle sobre eles, já que se executam de forma independente.

Esse mecanismo é ilustrado na figura 6.

```
xxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxyy
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
yyyyyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
...

```

Main Thread



Worker Thread

Figura 6 – Execução de uma *thread* ao longo do tempo. Fonte: <http://www.albahari.com/threading/>.

O próximo exemplo realiza a mesma tarefa, porém vamos implementar a interface Runnable.

Quadro 42: Classe ExemploThreadRunnable implementando Runnable

```
1 package com.ead.threads;
2
3 public class ExemploThreadRunnable implements Runnable {
4
5     public void run() {
6         try {
7             for (int i = 0; i < 20; i++) {
8                 System.out.print("y");
9                 Thread.sleep(200);
10            }
11        } catch (Exception e) {
12            // tratamento da exceção
13        }
14    }
15 }
```

A interface Runnable define o método *run()*, de forma que as classes que a implementarem devem implementar esse método.

A classe Thread não nos obriga a implementar o método *run()* porque, pela própria definição, ela já implementa Runnable. A criação da *thread*, nesse caso, passa a ser a apresentada no quadro 43.

Quadro 43: Classe TesteThread02 definindo uma nova *thread*

```

1 package com.ead.threads;
2
3 public class TesteThread02 {
4     public static void main(String[] args) {
5         ExemploThreadRunnable tr = new ExemploThreadRunnable();
6         Thread t = new Thread(tr);
7         t.start();
8         try {
9             for (int i = 0; i < 20; i++) {
10                 System.out.print("x");
11                 Thread.sleep(200);
12             }
13         } catch (Exception e) {
14             // tratamento da exceção
15         }
16     }
17 }
```

Instanciamos a classe ExemploThreadRunnable e passamos essa instância para o construtor da classe Thread. Observe que criamos um objeto da classe Thread e não mais de uma subclasse. O resultado é o mesmo que no caso anterior.

Definindo um nome para a *thread*

Podemos definir um nome para a *thread* de duas formas:

- Pelo método *setName()*.
- Pelo construtor da classe Thread.

Essas formas estão ilustradas no exemplo dos quadros 44 e 45.

Quadro 44: Classe ExemploThreadRunnable02: obtendo o nome da *thread*

```

1 package com.ead.threads;
2
3 public class ExemploThreadRunnable02 implements Runnable {
4
5     public void run() {
6         String nome = Thread.currentThread().getName();
7         System.out.println("Nome inicial do novo thread: " + nome );
8     }
9 }
```

Quadro 45: Classe TesteThread03: atribuindo um nome para a *thread*

```

1 package com.ead.threads;
2
3 public class TesteThread03 {
4     public static void main(String[] args) {
5         ExemploThreadRunnable02 tr = new ExemploThreadRunnable02();
6         //Definindo o nome do thread como parâmetro do construtor
7         Thread t = new Thread(tr, "thread_pessoal");
8         t.start();
9         String nome = Thread.currentThread().getName();
10        System.out.println("Nome do thread principal : " + nome );
11    }
12 }
```

O resultado após a execução do método *main()* no quadro 45 é:

```
<terminated> TesteThread03 [Java Application] C:\Program
Nome do thread principal : main
Nome inicial do novo thread: thread_pessoal
```

Como cada *thread* executa em um processo separado, a instrução

```
String nome = Thread.currentThread().getName()
```

fornecerá o nome da *thread* que está em execução, e não de outra. Se ela estiver no método *main()*, mostrará o nome da *thread* que o executa. No caso, a *thread* principal possui, coincidentemente, esse nome: "main".

Quando atribuímos um novo nome para a *thread* que criamos, como em

```
Thread t = new Thread(tr, "thread_pessoal");
```

o método *run()* chamado por ela mostrará o seu nome, como no exemplo.

Podemos refatorar nosso programa de forma que o nome da *thread* seja informado, ou alterado, posteriormente, da forma mostrada no quadro 46.

Quadro 46: Classe TesteThread03 reescrita: atribuindo o nome para a *thread* posteriormente

```

1 package com.ead.threads;
2
3 public class TesteThread03 {
4     public static void main(String[] args) {
5         ExemploThreadRunnable02 tr = new ExemploThreadRunnable02();
6         Thread t = new Thread(tr);
7         //Definindo o nome do thread através do método setName()
8         t.setName("novo_nome");
9         t.start();
10        String nome = Thread.currentThread().getName();
11        System.out.println("Nome do thread principal : " + nome );
12    }
13 }
```

Executando múltiplas *threads*

É muito comum existirem várias *threads* sendo executadas simultaneamente, e, na maioria dos casos, elas atuam sobre um mesmo objeto. Vamos elucidar a execução de múltiplas *threads* pelo próximo exemplo, mostrado no quadro 47.

Quadro 47: Classe ThreadMultiplo

```

1 package com.ead.threads;
2
3 public class ThreadMultiplo implements Runnable {
4     public void run() {
5         try {
6             String nome = Thread.currentThread().getName();
7             for (int i = 0; i < 5; i++) {
8                 System.out.println("Thread: " + nome + ", posição " + i);
9                 Thread.sleep(200);
10            }
11        } catch (Exception e) {
12            // Tratamento da exceção
13        }
14    }
15 }
```

A chamada é mostrada no quadro 48.

Quadro 48: Classe TesteThread04: definindo mais de uma *thread* sobre o mesmo objeto

```

1 package com.ead.threads;
2
3 public class TesteThread04 {
4     public static void main(String[] args) {
5         ThreadMultiplo tm = new ThreadMultiplo();
6
7         Thread t1 = new Thread(tm, "João");
8         Thread t2 = new Thread(tm, "Maria");
9
10        t1.start();
11        t2.start();
12    }
13 }
```

A execução produz o resultado:

```

<terminated> TesteThread04 [Java Application]
Thread: Maria, posição 0
Thread: João, posição 0
Thread: João, posição 1
Thread: Maria, posição 1
Thread: João, posição 2
Thread: Maria, posição 2
Thread: João, posição 3
Thread: Maria, posição 3
Thread: João, posição 4
Thread: Maria, posição 4
```

Observe mais uma vez que não há concordância entre as duas *threads*, porque estão em processos separados.

Aula 9 - Estados de threads

A classe Thread

Dependendo do momento no ciclo de vida da *thread*, existem estados específicos. O diagrama da figura 7 mostra os estados nos quais uma *thread* em Java pode estar e alguns métodos que podem ser usados para mudar de um para outro.

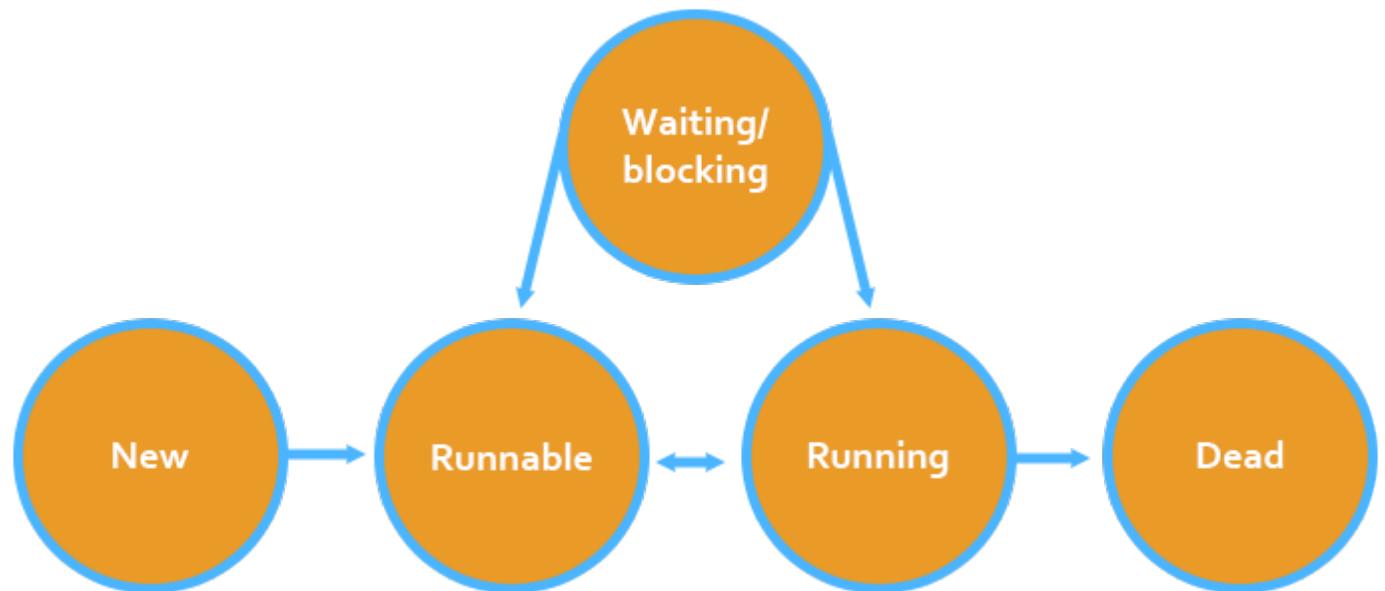


Figura 7 - Estados da *thread*. Fonte: Sierra e Bates (2008).

A descrição dos estados, juntamente com exemplos, é mostrada a seguir:

Estado	Descrição
NewThread	<p>Inicialização da <i>thread</i>: realizada por meio do construtor <code>Thread()</code>.</p> <pre>Thread t1 = new Thread(tm, "João"); Thread t2 = new Thread(tm, "Maria");</pre> <p>Nesse estado, nenhum recurso do sistema foi alocado para a <i>thread</i> ainda.</p>
Runnable / Running	<p>Este é o estado em que a <i>thread</i> está <i>pronta para rodar</i>. O método <code>start()</code> requisita os recursos do sistema necessários para rodar a <i>thread</i> e chama o seu método <code>run()</code>.</p> <pre>Thread t1 = new Thread(tm, "João"); Thread t2 = new Thread(tm, "Maria"); t1.start(); t2.start();</pre> <p>O estado <i>Running</i> ocorre quando o agendador de tarefas coloca a <i>thread</i> para funcionar. Quando uma <i>thread</i> está “<i>Running</i>”, está também “<i>Runnable</i>”, e as instruções do seu método <code>run()</code> é que estão sendo executadas pela CPU.</p>

Waiting / Blocking

O estado *Waiting* significa que a *thread* está impedida de executar por alguma razão, que pode ser:

Alguém executa o método *suspend()*.

Alguém executa o método *sleep()*.

A *thread* bloqueia, esperando por I/O.

A *thread* usa seu método *wait()* para esperar por uma variável de condição.

O exemplo abaixo coloca a *thread* atual “para dormir” por um segundo.

```
try {
    Thread.sleep(1000);
} catch (InterruptedException e) {
}
```

Cada uma dessas maneiras tem a sua forma específica de sair do estado *Waiting / Blocking*.

Se a *thread* foi suspensa, alguém precisa mandar-lhe a mensagem *resume()*.

Se a *thread* foi posta para dormir, voltará a ser *Runnable* quando o número de milissegundos determinado passar.

Se a *thread* está bloqueada, esperando por I/O, esta precisa ser completada.

Se a *thread* está esperando por uma variável de condição, o objeto que a “segura” precisa liberá-la, por meio de um *notify()* ou de um *notifyAll()*.

Dead

Uma *thread* pode morrer de “causas naturais” (quando o seu método *run()* acaba normalmente) ou pode ser finalizada pelo método *stop()*.

O método *stop()* não é considerado seguro, mas ainda assim é executado, finalizando a *thread* esteja ela onde estiver executando alguma tarefa.

```
public void run() {
    try {
        String nome = Thread.currentThread().getName();
        for (int i = 0; i < 5; i++) {
            System.out.println("Thread: " + nome + ", posição " + i);
            Thread.sleep(200);
        }
    } catch (Exception e) {
        // Tratamento da exceção
    }
}
```

Prioridades e agendamento de *threads*

Quando trabalhamos com *threads* em Java, devemos saber que cada uma possui uma prioridade de execução e que o gerenciador de *threads* (*scheduler*) decidirá qual executar primeiro.

Por padrão, todas as *threads* possuem prioridade *NORM_PRIORITY*, cujo valor é 5 e está declarada na classe *Thread*. Além disso, cada uma herda automaticamente a prioridade da *thread* que a criou.

As constantes *MAX_PRIORITY* (prioridade máxima), *MIN_PRIORITY* (prioridade mínima) e *NORM_PRIORITY* (prioridade normal) são usadas para definir as prioridades das *threads* Java.

Veja um exemplo no qual temos duas *threads*. A primeira possui a prioridade máxima, enquanto a segunda possui a prioridade mínima, conforme pode ser visto nos quadros 49 e 50.

Quadro 49: Classe ExemploPrioridades

```
1 package com.ead.threads;
2
3 public class ExemploPrioridades implements Runnable {
4
5     public void run() {
6         for (int i = 0; i < 20; i++) {
7             System.out.print(Thread.currentThread().getName());
8         }
9     }
10 }
```

Quadro 50: Classe TesteThread05: definindo prioridades para as threads

```
1 package com.ead.threads;
2
3 public class TesteThread05 {
4     public static void main(String[] args) {
5         ExemploPrioridades ep = new ExemploPrioridades();
6         Thread t1 = new Thread(ep, "t1");
7         Thread t2 = new Thread(ep, "t2");
8
9         t1.setPriority(Thread.MIN_PRIORITY);
10        t1.start();
11        t2.setPriority(Thread.MAX_PRIORITY);
12        t2.start();
13    }
14 }
```

<terminated> TesteThread05 [Java Application] C:\Program Files (x86)\Java\jre7\bin\javaw.exe (17/05/2014)

Observe que atribuímos prioridade máxima a t2 e, quando ela entrou em execução, tomou a frente da thread que havia sido iniciada antes, t1.

Aula 10 - Relacionamentos entre *threads* - sincronismo

As *threads* são executadas em processos separados por definição. Por padrão, não existem formas de relacioná-las. Mas podemos mudar esse cenário.

Como exemplo, suponhamos que uma conta corrente seja acessada por duas *threads* diferentes. Cada uma consulta o saldo e, em seguida, efetua o saque, se houver saldo. As classes envolvidas nesse exemplo são apresentadas nos quadros 51, 52 e 53.

Quadro 51: Classe ContaCorrente.java: indica a classe cujo objeto será manipulado pelas *threads*

```

1 package com.ead.threads.sincronismo;
2
3 public class ContaCorrente {
4     private double saldo;
5
6     public ContaCorrente(){
7         saldo = 200;
8     }
9
10    public boolean verificarSaldo(double valor){
11        return (valor <= saldo);
12    }
13
14    public double efetuarSaque(double valor){
15        saldo -= valor;
16        return saldo;
17    }
18 }
```

Quadro 52: Classe ProcessaContaCorrente.java: é a classe que representará cada *thread*, pois implementa a interface Runnable

```

1 package com.ead.threads.sincronismo;
2
3 public class ProcessaContaCorrente implements Runnable{
4
5     ContaCorrente conta = new ContaCorrente();
6
7     private void efetuarOperacao(double valor){
8         String nome = Thread.currentThread().getName();
9         try {
10             if(conta.verificarSaldo(valor)){
11                 System.out.println("Valor sacado por " + nome + " : " + valor);
12                 System.out.println("Saldo para " + nome + ": " +
13                                     conta.efetuarSaque(valor));
14             }
15             else {
16                 System.out.println("Saldo Insuficiente para " + nome);
17             }
18         } catch (Exception e) {
19             e.printStackTrace();
20         }
21     }
22
23     public void run() {
24
25         try {
26             for (int i = 0; i < 10; i++) {
27                 Thread.sleep(180);
28                 efetuarOperacao(20);
29             }
30         } catch (Exception e) {
31             e.printStackTrace();
32         }
33     }
34 }
```

Quadro 53: Classe TesteContaCorrente.java: Programa que criará duas *threads* e as executará

```

1 package com.ead.threads.sincronismo;
2
3 public class TesteContaCorrente {
4     public static void main(String[] args) {
5         ProcessaContaCorrente cc = new ProcessaContaCorrente();
6         Thread t = new Thread(cc, "João");
7         Thread t2 = new Thread(cc, "Maria");
8         try {
9             t.start();
10            t2.start();
11        } catch (Exception e) {
12            e.printStackTrace();
13        }
14    }
15 }
```

Quando executamos esse programa mais de uma vez, obtemos diferentes resultados em cada execução. Uma delas é mostrada a seguir:

```

<terminated> TesteContaCorrente [Java Appli
Valor sacado por Maria : 20.0
Valor sacado por João : 20.0
Saldo para Maria: 180.0
Saldo para João: 160.0
Valor sacado por Maria : 20.0
Saldo para Maria: 140.0
Valor sacado por João : 20.0
Saldo para João: 120.0
Valor sacado por Maria : 20.0
Saldo para Maria: 100.0
Valor sacado por João : 20.0
Saldo para João: 80.0
Valor sacado por Maria : 20.0
Valor sacado por João : 20.0
Saldo para João: 60.0
Saldo para Maria: 40.0
Valor sacado por Maria : 20.0
Saldo para Maria: 20.0
Valor sacado por João : 20.0
Saldo para João: 0.0
Saldo Insuficiente para João
Saldo Insuficiente para Maria
Saldo Insuficiente para João
Saldo Insuficiente para Maria
Saldo Insuficiente para Maria
Saldo Insuficiente para João
Saldo Insuficiente para Maria
Saldo Insuficiente para João
Saldo Insuficiente para Maria
Saldo Insuficiente para João
```

Se houvesse uma harmonia entre as *threads*, em todos os momentos apareceriam as mensagens, nesta ordem:

Valor sacado por Maria: 20.0

Saldo para Maria: 100.0

Ou seja, primeiro seria indicado o saque e, em seguida, o saldo. Mas não é o que vemos. Além disso, existe o risco de ocorrer um saque sem saldo suficiente, tornando o saldo final negativo, porque uma *thread* pode verificá-lo ao mesmo tempo que a outra e, havendo saldo disponível, as duas realizariam o saque.

Essa é a situação ideal para o que chamamos de *sincronismo*, ou seja, devemos sincronizar essas operações para garantir que, enquanto uma *thread* estiver consultando o saldo, ela permita o acesso a outra somente após finalizar a operação de saque. Em outras palavras, uma *thread* consulta o saldo e realiza o saque, enquanto a outra aguarda sua vez para entrar em processamento.

Na classe ProcessaContaCorrente, temos o método que realiza esse trabalho para nós, que é *efetuarOperacao()*. Esse método deve ser sincronizado. Veja o quadro 54.

Quadro 54: Definição do método *efetuarOperacao()* sincronizado

```
private synchronized void efetuarOperacao(double valor){
    String nome = Thread.currentThread().getName();
    try {
        if(conta.verificarSaldo(valor)){
            System.out.println("Valor sacado por " + nome + " : " + valor);
            System.out.println("Saldo para " + nome + ": " +
                               conta.efetuarSaque(valor));
        }
        else {
            System.out.println("Saldo Insuficiente para " + nome);
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

O resultado agora parece mais organizado. Veja:

```
<terminated> TesteContaCorrente [Java Application]
Valor sacado por João : 20.0
Saldo para João: 180.0
Valor sacado por Maria : 20.0
Saldo para Maria: 160.0
Valor sacado por João : 20.0
Saldo para João: 140.0
Valor sacado por Maria : 20.0
Saldo para Maria: 120.0
Valor sacado por João : 20.0
Saldo para João: 100.0
Valor sacado por Maria : 20.0
Saldo para Maria: 80.0
Valor sacado por Maria : 20.0
Saldo para Maria: 60.0
Valor sacado por João : 20.0
Saldo para João: 40.0
Valor sacado por Maria : 20.0
Saldo para Maria: 20.0
Valor sacado por João : 20.0
Saldo para João: 0.0
Saldo Insuficiente para João
Saldo Insuficiente para Maria
Saldo Insuficiente para João
Saldo Insuficiente para Maria
Saldo Insuficiente para Maria
Saldo Insuficiente para João
Saldo Insuficiente para Maria
Saldo Insuficiente para João
Saldo Insuficiente para Maria
Saldo Insuficiente para João
Saldo Insuficiente para João
```

Aula 11 - Multithreading com GUI

É possível executar *threads* em interface gráfica (GUI – *graphical user interface*), da mesma forma que em outras aplicações. No exemplo que apresentaremos, quando o usuário clicar em um botão, a hora é atualizada em uma caixa de textos a cada segundo.

É importante entendermos que esse exemplo trata de interface gráfica, mas que o tema associado a ela será abordado no próximo módulo. O objetivo aqui é ilustrar sua funcionalidade. Vamos ao exemplo, então. Veja o quadro 55.

Quadro 55: Classe Cronometro: definição de uma interface gráfica para execução de *threads*

```

19
20  public static void main(String[] args) {
21    EventQueue.invokeLater(new Runnable() {
22      public void run() {
23        try {
24          Cronometro frame = new Cronometro();
25          frame.setVisible(true);
26        } catch (Exception e) {
27          e.printStackTrace();
28        }
29      }
30    });
31  }
32
33 Thread t = null;
34  public Cronometro() {
35    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
36    setBounds(100, 100, 443, 160);
37    contentPane = new JPanel();
38    contentPane.setBorder(new EmptyBorder(5, 5, 5, 5));
39    setContentPane(contentPane);
40    contentPane.setLayout(null);
41
42
43 JButton btnIniciar = new JButton("Iniciar");
44  btnIniciar.addActionListener(new ActionListener() {
45    public void actionPerformed(ActionEvent e) {
46
47      t = new Thread(new Runnable(){
48        public void run() {
49          try {
50            for (int i = 0; i < 50; i++) {
51              textField.setText(new Date().toString());
52              Thread.sleep(1000);
53            }
54          } catch (InterruptedException e) {
55            e.printStackTrace();
56          }
57        }
58      });
59      t.start();
60    }
61  });
62  btnIniciar.setBounds(10, 43, 91, 23);
63  contentPane.add(btnIniciar);
64
65 JTextField textField = new JTextField();
66 textField.setBounds(111, 44, 259, 20);
67 contentPane.add(textField);
68 textField.setColumns(10);
69
70 JButton btnTerminar = new JButton("Terminar");
71  btnTerminar.addActionListener(new ActionListener() {
72    public void actionPerformed(ActionEvent e) {
73      if(t != null){
74        t.stop();
75      }
76    }
77  });
78  btnTerminar.setBounds(10, 77, 91, 23);
79  contentPane.add(btnTerminar);
80}
81 }
```

A execução desse programa produz o resultado:



Quando clicamos no botão “Iniciar”, a *thread* é criada e iniciada, mostrando a hora atualizada a cada segundo por certo tempo. Para terminarmos antecipadamente a execução, clicamos no botão “Terminar”, o que encerra a *thread*.

O ponto importante a ser destacado aqui é que a classe possui a interface gráfica e a *thread* manipula um elemento dela, que é a caixa de textos. A classe *Cronometro*, que gerou essa janela, não pode ser a implementadora de *Runnable* porque teríamos que atualizar a janela a cada segundo, já que esse é o objetivo da *thread*. Não podemos, também, criar uma classe separada, por esta não ter acesso à caixa de textos. Para mantermos a orientação a objetos coerente, sem as famosas “gambiarras”, usaremos um modelo sofisticado conhecido como “classe interna anônima”. O código que utiliza esse conceito está destacado no quadro 56.

Quadro 56: Código que implementa uma classe interna anônima como evento de ação

```

JButton btnIniciar = new JButton("Iniciar");
btnIniciar.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {

        t = new Thread(new Runnable(){
            public void run() {
                try {
                    for (int i = 0; i < 50; i++) {
                        textField.setText(new Date().toString());
                        Thread.sleep(1000);
                    }
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });
        t.start();
    }
});
```

Observe que foi passado como parâmetro para o construtor da classe *Thread* uma instância de *Runnable*, mas não podemos criar instâncias de *Runnable* por ser uma interface. Então, o que aconteceu?

Verifique que temos a instrução *new Runnable() {}*, que não é uma instância isolada; ela é seguida de um bloco de código (código entre chaves). O código entre chaves é uma implementação anônima da interface, que, por ser anônima, possui apenas conteúdo, mas nenhum nome.

De toda forma, foi passado um objeto *Runnable* para a *thread*, porém ele não foi atribuído a uma referência.

Os demais detalhes a respeito de interfaces gráficas serão apresentados no próximo módulo.

Exercícios do Módulo 4

Exercício 1: Revisão

Implementar todos os programas sugeridos neste módulo, já comentados ao longo do texto.

Exercício 2: Acesso simultâneo a objetos

a. Desenvolver uma classe chamada *Produtos* contendo os métodos:

- *verificarPreco()* – retorna o preço de um produto.
- *reajustarPreco()* – recebe como parâmetro uma taxa de reajuste e aplica-a ao preço do produto.
- *efetuarCompra()* – realiza a compra do produto, retornando o preço que foi aplicado nela.
- Um atributo *preco*, do tipo *double*.

b. Uma *thread* deve realizar os três passos ao realizar uma compra: verificar o preço, efetuar a compra e, em seguida, reajustar o preço. Definir a classe *ProcessaProdutos* para essa *thread*.

c. Em um programa, criar três *threads* que sejam capazes de acessar o mesmo objeto da classe *Produtos*. Executar a aplicação e verificar o resultado.

Exercício 3: Sincronismo de métodos

Alterar a classe *ProcessaProdutos* de modo a ter seus métodos sincronizados. O que é possível verificar após a execução da aplicação?

Módulo 5 - Componentes GUI

Aula 12 - Introdução aos componentes GUI

Definir uma interface gráfica em Java não é uma tarefa difícil. A definição da interface consiste em declararmos e instanciarmos elementos com base nas classes que os definem.

Existem diversos editores de interface gráfica disponíveis para integrarmos com o Eclipse. *WindowBuilder* é um editor de que eu gosto bastante de usar e que recomendo.

Neste módulo, conhceremos os mecanismos de definição de classes que nos permitirão definir uma interface gráfica. O universo é vasto, por isso trabalharemos com as que nos permitem criar uma aplicação para Windows. Para ilustrar o desenvolvimento de uma interface gráfica, vamos elaborar uma passo a passo.

Definindo a classe

O primeiro passo é representado pelo código no quadro 57.

Quadro 57: Classe Programa estendendo a classe JFrame

```

1 package com.ead.gui;
2
3 import javax.swing.JFrame;
4
5 public class Programa extends JFrame {
6
7     public Programa(){
8         this.setBounds(100, 100, 400, 300);
9     }
10 }
```

Quando desejarmos criar uma interface gráfica, de modo geral, criamos uma classe que estende *JFrame*. No construtor dessa classe, definimos o estado inicial do objeto. No caso, será a nossa janela. Cha-

mamos o método *setBounds()* informando os quatro parâmetros, nessa ordem:

- *int x*: distância horizontal do canto superior esquerdo da janela.
- *int y*: distância vertical do canto superior esquerdo da janela.
- *int width*: largura da janela.
- *int height*: altura da janela.

Exibindo a janela

A exibição da janela consiste na instanciação da classe que criamos, ou seja, do nosso *JFrame*. O programa no quadro 58 será usado em todas as etapas. Sempre que mencionarmos a execução da janela, na verdade estamos nos referindo à execução do método *main()* na classe *TestePrograma* desse quadro.

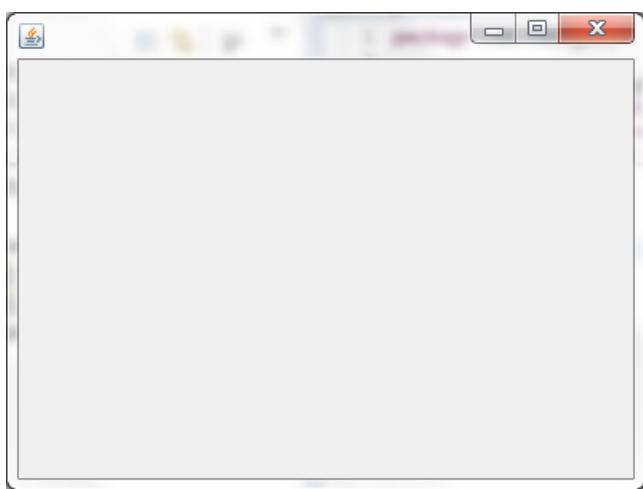
Quadro 58: Classe *TesteProgramma* instanciando e exibindo a janela definida na classe *Programma*

```

1 package com.ead.gui;
2
3 public class TesteProgramma {
4     public static void main(String[] args) {
5         Programa p = new Programa();
6         p.setVisible(true);
7     }
8 }
```

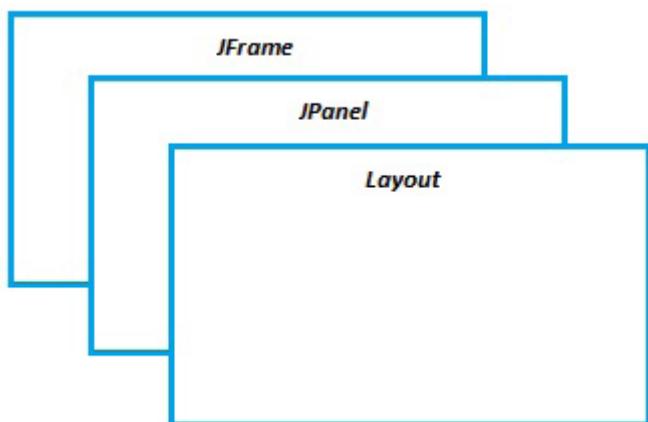
O método *setVisible()* na linha 6 é usado para tornar a janela visível quando informamos *true* como parâmetro. Esse método poderia ter sido chamado no construtor da classe.

A execução desse programa mostra-nos, até o momento, a interface:



Incluindo um painel e um *layout*

Uma interface gráfica possui três camadas que devemos considerar. A figura 8 mostra essas camadas e suas características.



JFrame: nossa janela.

JPanel: uma área para inclusão e agrupamento de componentes.

Layout: Um formato para disposição dos componentes.

Figura 8 - Camadas da interface gráfica. Fonte: produção do autor.

Essa divisão existe porque a JVM é capaz de executar a janela em qualquer sistema operacional, e as regras que são comuns no Windows podem não valer para outros ambientes operacionais. Por isso não são considerados valores padrão.

A classe Programa atualizada é mostrada no quadro 59.

Quadro 59: Classe Programa reescrita: definindo o painel

```

1 package com.ead.gui;
2
3 import javax.swing.JFrame;
4 import javax.swing.JPanel;
5
6 public class Programa extends JFrame {
7
8     private JPanel painel;
9
10    public Programa(){
11        painel = new JPanel();
12        this.setContentPane(painel);
13        painel.setLayout(null);
14        this.setBounds(100, 100, 400, 300);
15    }
16 }
```

Criamos um objeto da classe JPanel (linhas 8 e 11). Em seguida, adicionamos esse painel à nossa janela (linha 12) e definimos o *layout* (linha 13). O valor *null* para o *layout* indica que os componentes poderão ser inseridos livremente na interface, baseados nas suas coordenadas, também configuradas pelo método *setBounds()*.

Incluindo elementos na interface

Para ilustrar a inclusão dos elementos na interface, incluiremos:

- Uma caixa de textos (Classe JTextField).
- Um rótulo (classe JLabel).
- Uma lista suspensa (classe JComboBox).
- Um botão de comandos (classe JButton).

Os elementos são declarados na classe. No construtor, eles são instanciados e posicionados no painel.

O código completo é mostrado nos quadros 60 e 61.

Quadro 60: Classe Programa reescrita: declarando os componentes de interface gráfica

```

1 package com.ead.gui;
2
3 import javax.swing.JButton;
4 import javax.swing.JComboBox;
5 import javax.swing.JFrame;
6 import javax.swing.JLabel;
7 import javax.swing.JPanel;
8 import javax.swing.JTextField;
9
10 public class Programa extends JFrame {
11
12     private JPanel painel;
13     private JTextField txtNome;
14     private JLabel lblNome;
15     private JLabel lblSexo;
16     private JComboBox<String> cmbSexo;
17     private JButton btnEnviar;
18
19

```

Declaração dos Componentes

Quadro 61: Construtor da classe Programa instanciado e dimensionando os componentes

```

20 public Programa(){
21     painel = new JPanel();
22     this.setContentPane(painel);
23     painel.setLayout(null);
24     this.setBounds(100, 100, 400, 300);
25
26     txtNome = new JTextField();
27     lblNome = new JLabel("Seu Nome:");
28     lblSexo = new JLabel("Sexo: ");
29     cmbSexo = new JComboBox<String>();
30     btnEnviar = new JButton("Enviar");
31
32     lblNome.setBounds(10, 20, 100, 20);
33     txtNome.setBounds(120, 20, 200, 20);
34     lblSexo.setBounds(10, 50, 100, 20);
35     cmbSexo.setBounds(120, 50, 200, 20);
36     btnEnviar.setBounds(120, 80, 150, 25);
37
38     painel.add(lblNome);
39     painel.add(txtNome);
40     painel.add(lblSexo);
41     painel.add(cmbSexo);
42     painel.add(btnEnviar);
43 }
44

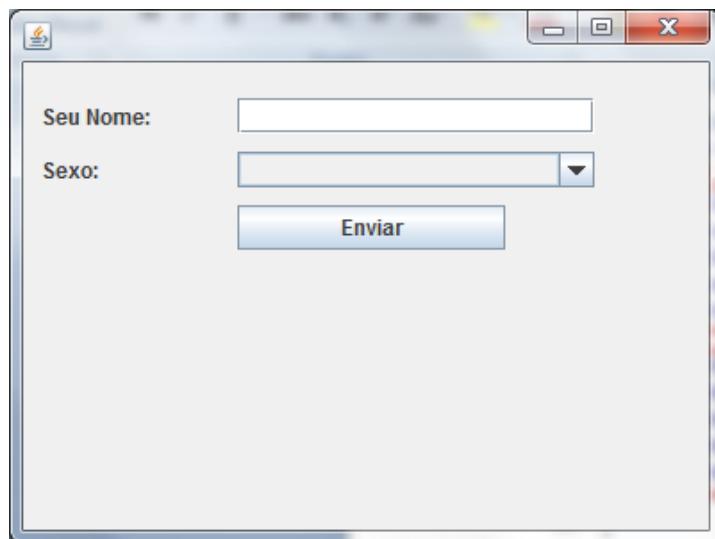
```

Instanciação dos Componentes

Dimensionamento dos Componentes

Adição dos elementos no painel

A execução desse programa gera como resultado:



Aula 13 - Eventos GUI

A interface gráfica em Java é dita orientada a eventos. Eventos são ações produzidas *sobre* os componentes, diferentemente de métodos, que são ações executadas *a partir* dos componentes. Exemplos de eventos:

- Clicar em um botão.
- Selecionar um item em uma lista.
- Pressionar uma tecla.
- Redimensionar uma janela.
- Iniciar uma aplicação.

Se uma aplicação está interessada em um evento específico (por exemplo, “clique em um botão”), deve solicitar ao sistema para “escutar” o evento. Os eventos são executados pelo sistema operacional; quando falamos em “escutar”, dizemos que o sistema aguarda por uma ação, e, quando esta é produzida, o código escrito para o evento em questão é executado.

Eventos são definidos por meio de interfaces, conhecidas como *listeners*. Existem *listeners* específicos para cada tipo de evento.

Apresentamos na tabela 2 os principais eventos ocorridos em uma interface gráfica Java.

Tabela 2 – Interfaces usadas nos eventos de interfaces gráficas

Interface	Definição	Métodos
ActionListener	Evento de ação, como clique do botão.	<code>void actionPerformed(ActionEvent e)</code>
AdjustmentListener	Eventos de ajustes.	<code>adjustmentValueChanged(AdjustmentEvent e)</code>

AWTEventListener	Evento disparado para componentes AWT.	<code>void eventDispatched(AWTEvent event)</code>
ComponentListener	Evento disparado para alterações em dimensões ou outros estados dos componentes.	<code>void componentResized(ComponentEvent e)</code> <code>void componentMoved(ComponentEvent e)</code> <code>void componentShown(ComponentEvent e)</code> <code>void componentHidden(ComponentEvent e)</code>
ContainerListener	Evento para fins de notificação apenas.	<code>void componentAdded(ContainerEvent e)</code> <code>void componentRemoved(ContainerEvent e)</code>
FocusListener	Ocorre quando um componente recebe ou pede o foco.	<code>void focusGained(FocusEvent e)</code> <code>void focusLost(FocusEvent e)</code>
HierarchyBoundsListener	Ocorre para notificar eventos de mudanças de estado em componentes ancestrais.	<code>void ancestorMoved(HierarchyEvent e)</code> <code>void ancestorResized(HierarchyEvent e)</code>
HierarchyListener	Ocorre quando ocorre alguma mudança na hierarquia dos elementos.	<code>void hierarchyChanged(HierarchyEvent e)</code>
InputMethodListener	Eventos de entrada de métodos.	<code>void inputMethodTextChanged (InputMethodEvent event)</code> <code>void caretPositionChanged (InputMethodEvent event)</code>
ItemListener	Ocorre quando itens são manipulados nos componentes.	<code>void itemStateChanged(ItemEvent e)</code>
KeyListener	Ocorre quando pressionamos teclas.	<code>void keyTyped(KeyEvent e)</code> <code>void keyPressed(KeyEvent e)</code> <code>void keyReleased(KeyEvent e)</code>
MouseListener	Ocorre nas mais diversas ações produzidas pelo mouse.	<code>void mouseClicked(MouseEvent e)</code> <code>void mousePressed(MouseEvent e)</code> <code>void mouseReleased(MouseEvent e)</code> <code>void mouseEntered(MouseEvent e)</code> <code>void mouseExited(MouseEvent e)</code>
MouseMotionListener	Ocorre quando movimentamos o mouse.	<code>void mouseDragged(MouseEvent e)</code> <code>void mouseMoved(MouseEvent e)</code>
MouseWheelListener	Ocorre quando rolamos o botão circular do mouse.	<code>void mouseWheelMoved(MouseWheelEvent e)</code>
TextListener	Ocorre quando o componente produz alterações em seu texto.	<code>void textValueChanged(TextEvent e)</code>
WindowFocusListener	Ocorre quando a janela recebe ou perde o foco.	<code>void windowGainedFocus(WindowEvent e)</code> <code>void windowLostFocus(WindowEvent e)</code>
WindowListener	Ocorre nas mais diversas manipulações produzidas na janela.	<code>void windowActivated(WindowEvent e)</code> <code>void windowClosed(WindowEvent e)</code> <code>void windowClosing(WindowEvent e)</code> <code>void windowDeactivated(WindowEvent e)</code> <code>void windowDeiconified(WindowEvent e)</code> <code>void windowIconified(WindowEvent e)</code> <code>void windowOpened(WindowEvent e)</code>

WindowStateListener	Ocorre para responder ao registro de mudança de estado de uma janela.	<code>void windowStateChanged(WindowEvent e)</code>
---------------------	---	---

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

Tratamento de eventos GUI

Vamos ver como os eventos são manipulados em uma interface. Existem alguns que são usados com mais frequência que outros.

Na interface que estamos desenvolvendo para este módulo, vamos usar o botão de comando para ilustrar a utilização do evento de ação. Usaremos o conceito de classes internas anônimas, abordado no módulo sobre *multithreading*.

Vamos mostrar aqui apenas o trecho do código para o botão que recebe o evento de ação (*ActionListener*). O código no quadro 62 permite executar esse evento.

Quadro 62: Definição do evento de ação para o botão btnEnviar no construtor da classe Programa

```

34     btnEnviar = new JButton("Enviar");
35     btnEnviar.addActionListener(new ActionListener() {
36
37         public void actionPerformed(ActionEvent arg0) {
38             JOptionPane.showMessageDialog(null,
39                 "Nome informado: " + txtNome.getText());
40         }
41     });

```

De modo geral, tomamos o componente a partir do qual queremos executar o evento e chamamos o método *addXXX()*, em que *XXX* é o nome da interface. No nosso exemplo, para a interface *ActionListener*, tomamos o método *addActionListener()*.

Como parâmetro para esse método, informamos uma implementação anônima da interface.

Aula 14 - Entrada e saída baseadas em GUI

Normalmente, os eventos são usados para colher informações de outros componentes ou atribuir valores a eles. Por exemplo, uma interface representando um *login* (tela para entrada de usuário e senha) possui um botão que, quando acionado, realiza a validação do usuário. Esta é realizada por meio da leitura das caixas de texto nas quais o usuário fornece suas credenciais.

Outro exemplo que podemos mencionar é quando fazemos uma busca de um cliente em um banco de dados usando seu CPF. Fornecemos o CPF em uma caixa de texto, clicamos em um botão, e o evento de ação deste executa um conjunto de instruções capazes de buscar as informações em registros de um banco de dados. Cada campo do registro é exibido em caixas de texto.

As entradas ou saídas de dados em componentes de interface gráfica estão relacionadas aos métodos que executamos para cada um deles.

No exemplo que estamos explorando neste módulo, consideremos que o sexo seja preenchido no componente `JComboBox` quando a janela for construída, ou seja, no construtor da classe. O código correspondente a essa ação é mostrado no quadro 63.

Quadro 63: Adicionando elementos ao componente `cmbSexo`, do tipo `JComboBox`

```
34     cmbSexo = new JComboBox<String>();
35     cmbSexo.addItem("MASCULINO");
36     cmbSexo.addItem("FEMININO");
```

A tabela 3 apresenta alguns exemplos de métodos que podemos utilizar.

Tabela 3 – Métodos usados nos eventos de interfaces gráficas

Método	Descrição
<code>getName()</code>	Retorna o nome do componente.
<code>setName(String n)</code>	Atribui um nome ao componente.
<code>getParent()</code>	Retorna o <i>container</i> onde o componente <code>getParent()</code> foi adicionado.
<code>setVisible(boolean b)</code>	Torna o componente visível ou não, de acordo com o parâmetro.
<code>isShowing()</code>	Retorna <code>true</code> se o componente for visível e estiver dentro de um outro componente que está sendo mostrado.
<code>isEnabled()</code>	Determina se um componente está habilitado para receber a entrada do usuário e gerar eventos.
<code>setEnabled(boolean b)</code>	Habilita ou desabilita o componente.
<code>setToolTipText(String text)</code>	Registra o texto para ser mostrado em <i>tooltip</i> .
<code>setPreferredSize(Dimension d)</code>	Atribui o tamanho desejado.
<code>getPreferredSize()</code>	Retorna o tamanho do componente.

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

Visão geral de componentes swing

Agora que tivemos uma visão geral dos componentes, bem como de seu funcionamento, seus métodos e seus eventos, vamos apresentar uma visão deles tomando como base sua hierarquia e suas classes. Essa hierarquia é apresentada na figura 9.

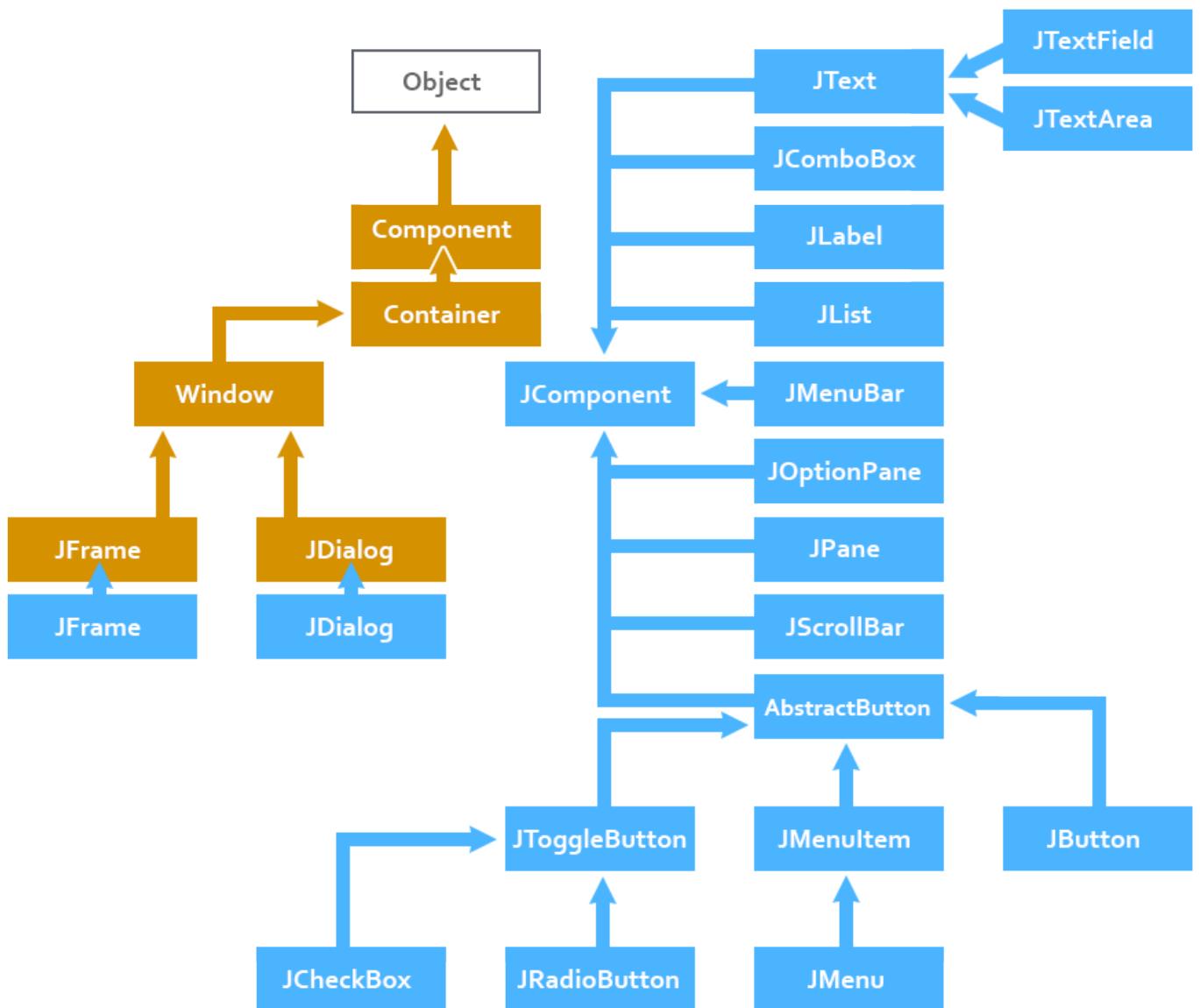


Figura 9 - Hierarquia dos componentes GUI.

A tabela 4 apresenta a definição de alguns componentes.

Tabela 4 – Componentes de interface gráfica

Componente	Descrição
<i>JLabel</i>	Componente onde podemos exibir textos não editáveis ou ícones.
<i>JTextField</i>	Componente onde inserimos texto, normalmente pelo teclado.
<i>JButton</i>	Componente usado para executar uma ação, normalmente por um clique.
<i>JCheckBox</i>	Componente que apresenta dois estados: selecionado ou não selecionado.
<i>JComboBox</i>	Lista de itens a partir da qual o usuário pode fazer uma seleção clicando em um item na lista ou digitando na caixa, se for permitido.
<i>JList</i>	Área em que uma lista de itens é exibida. O usuário pode fazer uma seleção clicando uma vez em qualquer elemento na lista.

JPanel*Container* onde os componentes podem ser inseridos.Fonte: <http://docs.oracle.com/javase/7/docs/api/>.

Aula 15 - Applets

Applet é um programa Java executado no *browser*. Ele pode ser uma aplicação Java totalmente funcional, pois tem toda a API à sua disposição.

Existem diferenças entre um *applet* e um aplicativo *swing*:

- Um *applet* é uma classe Java que estende a classe `java.applet.Applet`.
- O método `main()` não é chamado em um *applet*.
- *Applets* são projetados para serem incorporados em uma página HTML.
- Quando um usuário exibe uma página HTML que contém um *applet*, o código deste é baixado para a máquina do usuário.
- A JVM é necessária para visualizar um *applet*. Ela pode ser um *plug-in* do navegador da web ou um ambiente de tempo de execução separado.
- A JVM na máquina do usuário cria uma instância da classe *Applet* e invoca vários métodos durante seu ciclo de vida.
- *Applets* têm regras rígidas de segurança que são aplicadas pelo *browser*.

Ciclo de vida de um *applet*

Cinco métodos na classe *Applet* fornecem as funcionalidades de que você precisa. Eles estão definidos na tabela 5.

Tabela 5 – Métodos do ciclo de vida dos *applets*.

Método	Descrição
<code>init()</code>	Esse método destina-se a tudo o que é necessário para a inicialização do <i>applet</i> .
<code>start()</code>	Esse método é chamado automaticamente depois que o navegador chama o método <code>init()</code> . Ele também é chamado sempre que o usuário retorna para a página que contém o <i>applet</i> depois de ter ido para outras.
<code>stop()</code>	Esse método é chamado automaticamente quando o usuário sai da página que contém o <i>applet</i> . Pode, portanto, ser chamado repetidamente na mesma aplicação.
<code>destroy()</code>	Esse método só é chamado quando o navegador fecha normalmente. Como <i>applets</i> são desenvolvidos para ser executados em uma página HTML, é importante liberar recursos alocados por eles, e essa limpeza é feita nesse método.
<code>paint(Graphics)</code>	Invocado imediatamente após o método <code>start()</code> e a qualquer momento em que o <i>applet</i> precisar redesenhar seu conteúdo no navegador..

Exemplo de aplicação usando Applet

A classe cujo código está apresentado no quadro 64 representa um *applet*.

Quadro 64: Classe Introdução estendendo a classe Applet

```

1 package com.ead.applets;
2
3 import java.applet.Applet;
4 import java.awt.Graphics;
5
6 public class Introducao extends Applet {
7     public void paint(Graphics g){
8         g.drawString("Exemplo Applet", 25,50);
9     }
10 }
```

As importações a seguir são usadas nesse exemplo:

java.applet.Applet.
java.awt.Graphics.

A classe Applet

Cada *applet* é uma extensão da classe `java.applet.Applet`. A classe base `Applet` fornece métodos que seu *applet* pode usar para obter informações e serviços a partir do contexto do navegador. Esses métodos podem realizar as seguintes ações:

- Obter parâmetros do *applet*.
- Obter a localização de rede do arquivo HTML que contém o *applet*.
- Obter o local de rede do diretório de classe do *applet*.
- Imprimir uma mensagem de *status* no navegador.
- Buscar uma imagem.
- Buscar um *audioclip*.
- Tocar um *audioclip*.
- Redimensionar o *applet*.

Chamando um *applet*

Um *applet* pode ser executado pela sua incorporação em um arquivo HTML. A tag `<applet>` é a base para a incorporação de um *applet* em um arquivo HTML.

O exemplo do quadro 65 mostra como utilizá-la.

Quadro 65: Execução do *applet* Introducao em um código HTML

```

<html>
    <title>Exemplo Applets</title>
    <hr>
    <applet code="com.ead.applets.Introducao.class" width="320" height="120">
        Se o browser estiver habilitado para Java,
        uma mensagem aparecerá.
    </applet>
    <hr>
</html>

```

O atributo *code* da tag *<applet>* é necessário. Ele especifica a classe Applet a ser executada. A largura (*width*) e a altura (*height*) também são obrigatórias para especificar o tamanho inicial do painel em que o *applet* será executado.

Navegadores não habilitados para Java não processam o elemento *<applet>*. Portanto, qualquer informação dentro dessa tag não relacionada com o *applet* torna-se visível para eles.

Aula 16 - Informando parâmetros para o *applet*

O exemplo que apresentamos aqui demonstra o uso de parâmetros a ser passados para um *applet*. O método *Applet.getParameter()* obtém o nome do parâmetro a ser passado, em forma de *string*.

O exemplo no quadro 66 mostra o *applet* recebendo parâmetros, e o programa HTML do quadro 67 mostra como passá-los.

Quadro 66: Classe ParametrosApplet: definindo parâmetros para o *applet*

```

1 package com.ead.applets;
2
3 import java.applet.Applet;
4 import java.awt.Graphics;
5
6 public class ParametrosApplet extends Applet{
7
8     public void paint(Graphics g){
9         String nome = getParameter("nome");
10        String email = getParameter("email");
11        g.drawString("Nome: " + nome + " - Email: " + email, 25, 50);
12    }
13 }

```

Quadro 67: Demonstração da passagem de parâmetros para o *applet* pelo código HTML

```

<html>
<title>Checkerboard Applet</title>
<hr>
<applet code="com.ead.applets.ParametrosApplet.class" width="480"
height="320">
<param name="nome" value="José">
<param name="email" value="jose@ead.com">
</applet>
<hr>
</html>

```

Manipulação de eventos

O procedimento de manipulação de eventos no *applet* é semelhante ao apresentado para as interfaces *swing*. O quadro 68 mostra o código para um *applet* contendo um botão de comandos e seu evento de ação.

Quadro 68: Classe EventosApplet: definindo um evento de ação para o botão “botao”

```

1 package com.ead.applets;
2
3 import java.applet.Applet;
4 import java.awt.Button;
5 import java.awt.TextField;
6 import java.awt.event.ActionEvent;
7 import java.awt.event.ActionListener;
8
9 public class EventosApplet extends Applet{
10
11     private static final long serialVersionUID = 1L;
12     private Button botao;
13     private TextField texto;
14
15     public void init(){
16
17         botao = new Button("Mostrar");
18         texto = new TextField(20);
19
20         botao.addActionListener(new ActionListener() {
21             public void actionPerformed(ActionEvent e) {
22                 texto.setText("Exemplo Evento Ação Applet");
23             }
24         });
25     }
26 }

```

Exibindo imagens

Um *applet* pode exibir imagens em formatos GIF, JPEG, BMP, dentre outros. No quadro 69, temos um exemplo de como exibir uma imagem no *applet*.

Quadro 69: Classe ImagemApplet: exibindo uma imagem

```

1 package com.ead.applets;
2
3 import java.applet.Applet;
4 import java.applet.AppletContext;
5 import java.awt.Graphics;
6 import java.awt.Image;
7 import java.net.URL;
8
9 public class ImagemApplet extends Applet {
10
11     private static final long serialVersionUID = 1L;
12     private Image imagem;
13     private AppletContext contexto;
14
15     public void init(){
16         contexto = this.getAppletContext();
17         String urlImagen = this.getParameter("imagem");
18         if(urlImagen == null){
19             urlImagen= "padrao.jpg";
20         }
21         try {
22             URL url = new URL(this.getDocumentBase(), urlImagen);
23             imagem = contexto.getImage(url);
24         } catch (Exception e) {
25             e.printStackTrace();
26         }
27     }
28
29     public void paint(Graphics g){
30         contexto.showStatus("Exibindo imagem");
31         g.drawImage(imagem, 0,0,200,84,null);
32     }
33 }
```

Agora vamos chamar esse *applet*. O código HTML no quadro 70 mostra essa chamada.

Quadro 70: Executando o *applet* que exibe uma imagem

```

<html>
    <title>Exemplo Imagem</title>
    <hr>
    <applet code="com.ead.applets.ImagemApplet.class" width="300"
            height="200">
        <param name="imagem" value="java.jpg">
    </applet>
    <hr>
</html>
```

Reproduzindo áudio e vídeo

Além de imagens, é possível reproduzir arquivos de áudio e vídeo. Os métodos para essa finalidade estão listados na tabela 6.

Tabela 6 – Métodos para reprodução de áudio e vídeo

Método	Descrição
<i>play()</i>	Reproduz o clipe de áudio ou vídeo de uma vez, desde o início.
<i>loop()</i>	Faz com que o clipe de áudio ou vídeo se reproduza continuamente.
<i>stop()</i>	Para a reprodução do clipe de áudio ou vídeo.

Para obter um objeto AudioClip, é necessário chamar o método *getAudioClip()* da classe Applet. Esse método retorna imediatamente se a URL resolve um arquivo de áudio ou vídeo real. O arquivo não é baixado até que seja feita uma tentativa de reproduzir o clipe.

O applet no quadro 71 ilustra um exemplo.

Quadro 71: Classe AudioApplet: reprodução de áudio

```

8 public class AudioApplet extends Applet {
9     private AudioClip clip;
10    private AppletContext contexto;
11
12    public void init()
13    {
14        contexto = this.getAppletContext();
15        String audioURL = this.getParameter("audio");
16        if(audioURL == null) {
17            audioURL = "default.au";
18        }
19        try {
20            URL url = new URL(this.getDocumentBase(), audioURL);
21            clip = contexto.getAudioClip(url);
22        }catch(Exception e) {
23            e.printStackTrace();
24            contexto.showStatus("Não foi possível executar audio");
25        }
26    }
27    public void start() {
28        if(clip != null) {
29            clip.loop();
30        }
31    }
32    public void stop() {
33        if(clip != null) {
34            clip.stop();
35        }
36    }
37 }
```

Podemos executar esse *applet* por intermédio do programa HTML do quadro 72.

Quadro 72: Execução do *applet* que reproduz o áudio

```
<html>
<title>The ImageDemo applet</title>
<hr>
<applet code="com.ead.applets.AudioApplet.class" width="0" height="0">
<param name="audio" value="teste.wav">
</applet>
<hr>
</html>
```

Você pode usar o teste.wav no seu PC para testar o exemplo.

Aula 17 - Java Media Framework

Nesta aula, apresentaremos o uso do JMF - Java Media Framework. Mostraremos uma visão geral do *framework*, incluindo as principais funcionalidades e pontos de extensão.

Adicionando recursos de áudio e vídeo a uma aplicação Java

O JMF é um *framework* da Oracle (desenvolvido em parceria com a Intel e a IBM) que permite manipular recursos multimídia em aplicações Java.

Para utilizar os serviços de áudio e vídeo disponibilizados pelo *framework*, são necessários dispositivos de entrada (microfones, câmeras, arquivos de áudio/vídeo, *streaming* via rede) e/ou dispositivos de saída (monitor e caixa de som), dependendo da necessidade, que poderá ser, por exemplo:

- Tocar arquivos de mídia em um *applet* ou aplicação *swing*.
- Capturar áudio e vídeo de um microfone e uma câmera de vídeo USB.
- Transmitir áudio e vídeo em tempo real pela internet.

Para o desenvolvimento de aplicações multimídia, é importante o entendimento do conceito de *codec*.

Codec

Um *codec* é utilizado para compressão e descompressão de dados. A compressão é utilizada quando esses dados vão ser transmitidos ou armazenados. No componente receptor, eles são descompactados para um formato de apresentação. A tabela 7 mostra os formatos geralmente utilizados pelos *codecs* e os protocolos utilizados.

Tabela 7 – Formatos utilizados pelos codecs.

Protocolos	HTTP, FTP, RTP.
Áudio	AAC, MP3, outros.
Vídeo	MPEG-2, MPEG-4, Cinepak, H.261, H.263, AVI, outros.

Fonte: <http://www.devmedia.com.br/artigo-java-magazine-57-aplicacoes-multimidia-em-java/9423>.

Arquitetura do JMF

Na documentação desenvolvida originalmente pela Sun, a arquitetura do JMF é comparada ao processo decorrido desde a filmagem até a visualização do vídeo em um dispositivo de saída.

De acordo com Jorge, Vinicius e Martins (2012),

um **DataSource** encapsula o stream de áudio/vídeo, como um Vídeo Tape e um **Player** disponibiliza processamento e mecanismos de controle (avançar, retroceder), similar a um vídeo-cassete, que reproduzirá a mídia em um dispositivo de saída, tal como microfones e monitores.

A figura 10 mostra esse processo.

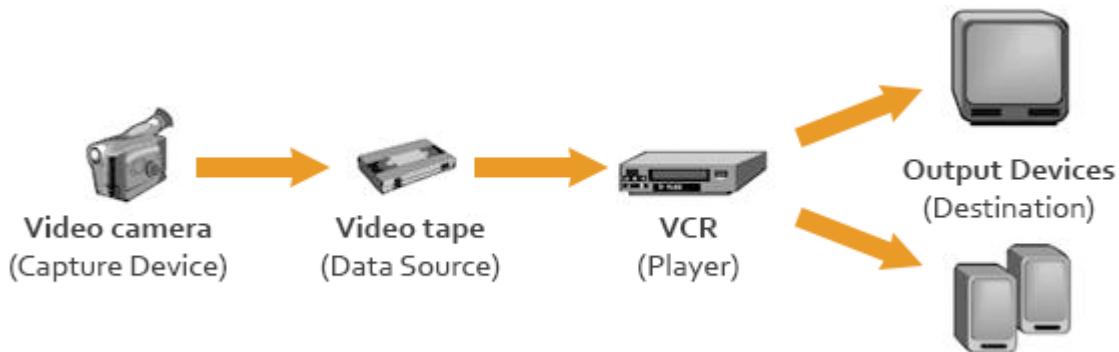


Figura 10 - Analogia das soluções JMF. Fonte: <http://www.devmedia.com.br/artigo-java-magazine-57-aplicacoes-multimidia-em-java/9423>.

Para a criação dos componentes providos pelo JMF para capturar, processar e apresentar os recursos de multimídia, são utilizados os seguintes componentes:

Manager: Responsável pela criação de **DataSources** e **Players**, entre outros, além de possibilitar a customização de componentes do framework;

PackageManager: Mantém um registro dos pacotes que contém classes do JMF, tal como classes customizadas: **Players**, **Processors** e **DataSources**;

CaptureDeviceManager: Contém uma base com os dispositivos de captura detectados na máquina;

PlugInManager: Responsável por manter a lista de plug-ins disponíveis para uso pelo JMF. É possível instalar novos plug-ins (JORGE; VINICIUS; MARTINS, 2012).

Para trabalharmos com JMF, primeiro precisamos obter o *framework*. Isso pode ser feito de: <http://www.oracle.com/technetwork/java/javasebusiness/downloads/java-archive-downloads-java-client-419417.html#7372-jmf-2.1.1e-oth-JPR>. Verifique possíveis atualizações e mudanças de *link*.

Após selecionarmos a opção Download, chegamos à tela mostrada na figura 11.

Java Media Framework (JMF) 2.1.1e

You must accept the Oracle Binary Code License Agreement for Java SE to download this software.

Accept License Agreement Decline License Agreement

Product / File Description	File Size	Download
Cross-platform Java	1.87 MB	jmf-2_1_1e-alljava.zip
Linux Performance Pack	2.31 MB	jmf-2_1_1e-linux-i586.bin
Solaris SPARC Performance Pack	4.27 MB	jmf-2_1_1e-solaris-sparc.bin
Windows Performance Pack	4.98 MB	jmf-2_1_1e-windows-i586.exe

[Back to top](#)

Figura 11. Download do Java Media Framework. Fonte: <http://www.oracle.com/technetwork/java/javasebusiness/downloads/java-archive-downloads-java-client-419417.html#7372-jmf-2.1.1e-oth-JPR>.

É necessário aceitar os termos de licença. O arquivo a ser baixado é o “Cross-platform Java”

Para adicionar esse *framework* à sua aplicação, é necessário incluí-lo no projeto. Para tanto, siga os passos:

- a. Clicar com o botão direito do mouse no projeto e selecionar *Properties*.
- b. Na janela que aparecer (figura 11), clique em *Add External Jars...*, selecionando os arquivos indicados na figura 12.

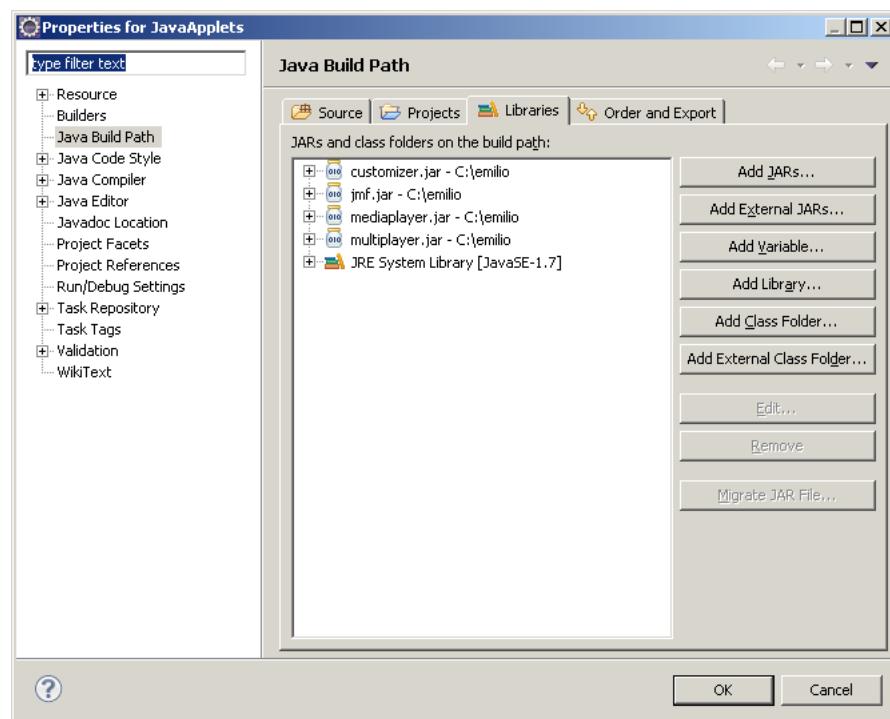


Figura 12. Inclusão do JMF ao projeto, no Eclipse. Fonte: Produção do autor.

Agora estamos prontos para criar uma aplicação. O código no quadro 73 apresenta um *applet* utilizando o JMF.

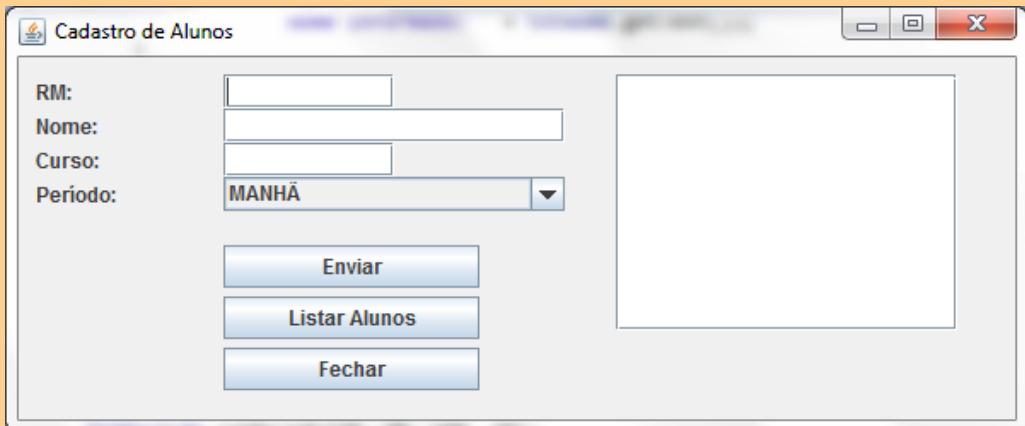
Quadro 73: Classe JMFApplet: reproduzindo um áudio

```
2① import java.applet.Applet;
3 import java.awt.Component;
4 import java.io.IOException;
5 import java.net.URL;
6
7 import javax.media.CannotRealizeException;
8 import javax.media.Manager;
9 import javax.media.NoPlayerException;
10 import javax.media.Player;
11
12 public class JMFApplet extends Applet {
13②     public void init(){
14
15         Manager.setHint(Manager.LIGHTWEIGHT_RENDERER, true);
16
17         try {
18             URL url = new URL(this.getDocumentBase(), "movie.wmv");
19             Player player = Manager.createRealizedPlayer(url);
20             Component c = player.getVisualComponent();
21
22             if(c != null)
23                 add(c);
24
25             player.start();
26         } catch (IOException ex) {
27             ex.printStackTrace();
28         } catch (NoPlayerException ex) {
29             ex.printStackTrace();
30         } catch (CannotRealizeException ex) {
31             ex.printStackTrace();
32         }
33     }
34 }
```

Exercícios do Módulo 5

Exercício 1: Elaboração de uma interface gráfica.

- a. Criar uma interface gráfica a partir de uma classe chamada *AlunosGUI* para receber informações de alunos. A interface deverá se parecer com esta:



- b. Escrever uma classe chamada *TesteAlunos* contendo o método *main()*, que instancie a classe *AlunosGUI* e a apresente na tela.

Exercício 2: Definição da classe Alunos e separação de camadas.

- Criar uma classe chamada *Aluno*, com os atributos: *rm*, *nome*, *curso*, *periodo*.
- Nessa classe, inserir os *getters* e *setters*. Para isso, usar o *template* do Eclipse.
- Sobrescrever o método *toString()* na classe, de modo a retornar a representação *string* de um aluno no formato: "[rm – nome – curso – período]".
- Na classe *Aluno*, definir um objeto estático do tipo *List<Aluno>*, instanciado como *new ArrayList<Aluno>()*.

Exercício 3: Definição das funcionalidades da aplicação.

- No evento de ação do botão "*Enviar*":
 - Criar um objeto da classe *Aluno*.
 - Atribuir o conteúdo de cada caixa de textos para cada atributo da classe, por intermédio de seus *setters*.
 - Incluir o objeto na lista que você definiu no item d do exercício 2.
- No evento de ação do botão "*Listar Alunos*":
 - Escrever uma estrutura de repetição para percorrer a lista de alunos.
 - Incluir cada elemento na lista no componente *JList*, do lado direito da interface gráfica.
- No evento de ação do botão "*Fechar*":
 - Escrever uma instrução para finalizar a aplicação.

Módulo 6 - Arquivos

Aula 18 - Entrada e saída em Java usando arquivos

Entrada e saída em Java: *streams*

Em Java, o processo de entrada/saída (*input/output*) é realizado por meio de *streams*. *Streams* consistem em uma abstração criada para representar locais reais (teclado, disco, monitor, rede de comunicação etc.) nos quais dados devem ser obtidos ou escritos.

Streams representam, portanto, a fonte de um fluxo de dados em dispositivos de entrada ou de saída. Observe que esses fluxos são unidirecionais: um *stream* de entrada pode ser lido, e um *stream* de saída pode ser escrito. O interesse maior dessa abstração é oferecer o mesmo conjunto de serviços para manipular diferentes tipos de dispositivos e arquivos.

Observe que canais de comunicação e arquivos em disco podem ser abertos em modo entrada ou modo saída.

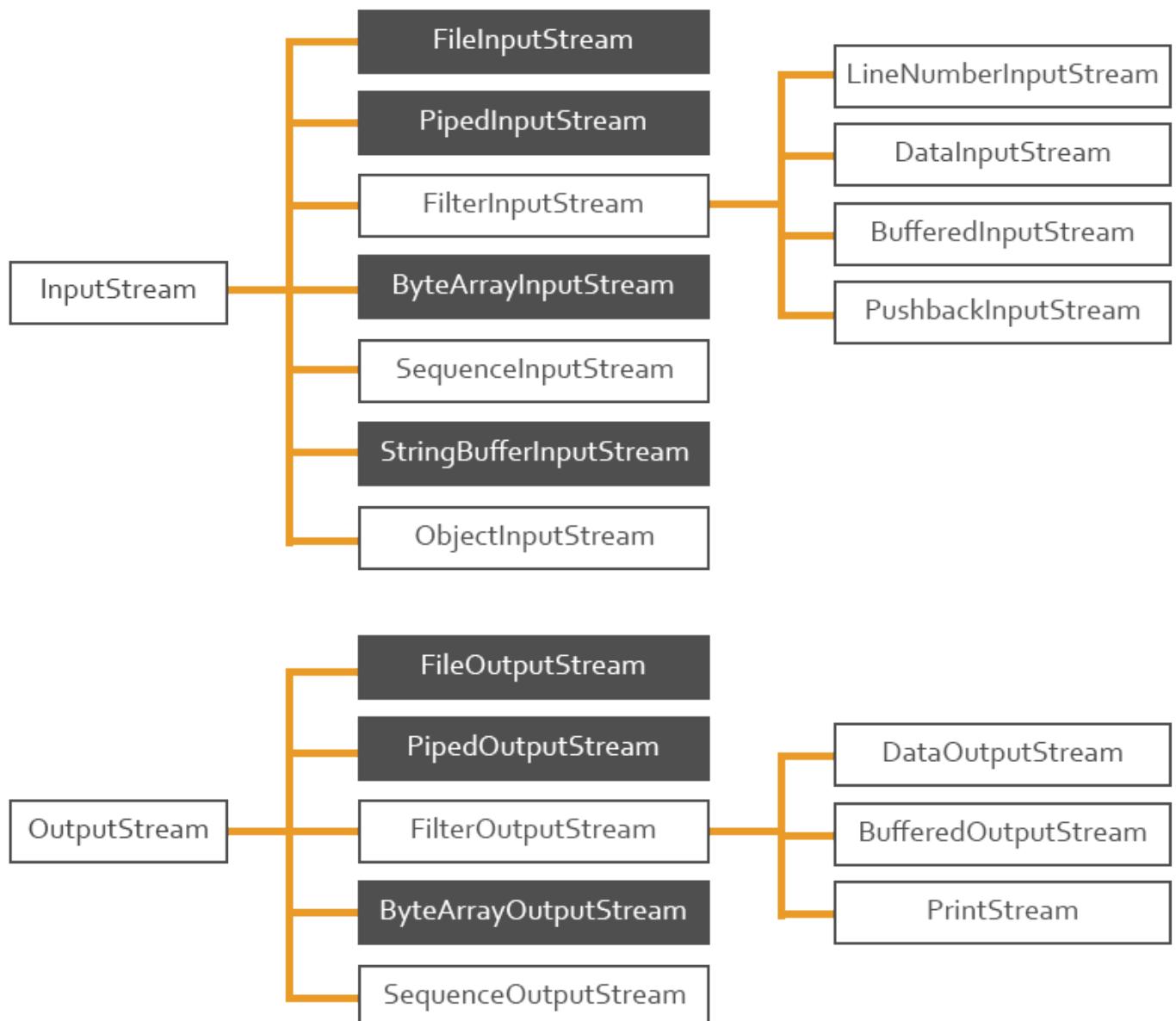
Existem dois mecanismos básicos para manipular *streams*: bufferizados e não bufferizados. Estes últimos acessam diretamente os arquivos ou dispositivos indicados como de entrada ou saída de dados. Mecanismos bufferizados empregam uma zona de memória própria ao mecanismo de entrada e saída para armazenar temporariamente os dados manipulados.

Em Java, são definidos dois tipos de *streams*: de caracteres e de bytes. O primeiro é dito modo texto, cujo conteúdo é comprehensível na linguagem humana. O segundo consiste na representação da informação em bytes, a mesma representação utilizada pelos computadores.

O pacote `java.io` disponibiliza um grande conjunto de classes para manipular *streams*. Nele, existem classes para manipular tanto *streams* bufferizados quanto não bufferizados, em modo texto e em modo byte. Algumas dessas classes e seus serviços serão objetos de estudo neste material.

InputStream e OutputStream

Essas classes são as mais genéricas para entrada e saída de *stream* de bytes. Elas definem a estrutura para entrada e saída que é especializada por outras classes, conforme mostra a figura 13.

Figura 13. Hierarquia de classes `InputStream` / `OutputStream`.

A tabela 8 descreve alguns dos serviços disponíveis na classe `InputStream`.

Tabela 8 – Métodos da classe `InputStream`

Serviço	Descrição
<code>int read()</code>	Lê um byte e retorna o seu valor em um inteiro. Retorna <code>-1</code> se chegou ao fim do arquivo.
<code>int read(byte b[])</code>	Escreve os bytes lidos na <i>array</i> de bytes <i>b</i> . Serão lidos, no máximo, <i>b.length bytes</i> . Retorna o número de bytes lidos.
<code>int read(byte b[], int off, int length)</code>	Escreve <i>length</i> bytes lidos no <i>array</i> de bytes passado como parâmetro. O primeiro byte lido é armazenado na posição <i>off</i> do <i>array</i> . Retorna o número de bytes lidos.

<code>void close ()</code>	Fecha a <i>stream</i> . Se existir uma pilha de <i>streams</i> , fechará a do topo da pilha e depois todas as outras.
<code>int available ()</code>	Retorna o número de bytes disponíveis para leitura.
<code>long skip (long nroBytes)</code>	Este método é usado para movimentar o ponteiro do arquivo. Ele descarta <i>nroBytes</i> bytes da <i>stream</i> . Retorna o número de bytes descartados.
<code>boolean void markSupported ()</code>	Retorna <i>true</i> se os métodos <i>mark()</i> e <i>reset()</i> são suportados pela <i>stream</i> .
<code>void mark (int posicao)</code>	Marca um determinado byte no arquivo.
<code>void reset()</code>	Volta ao ponto marcado.

Fonte: Sierra e Bates (2008).

A tabela 9 apresenta a descrição dos serviços da classe OutputStream.

Tabela 9 – Métodos da classe OutputStream

Serviço	Descrição
<code>void write (int)</code>	Grava um byte na <i>stream</i> .
<code>void write (byte b [])</code>	Grava os bytes contidos no array <i>b</i> na <i>stream</i> .
<code>void write (byte b[], int off, int length)</code>	Grava <i>length</i> bytes do array para a <i>stream</i> . O byte <i>b[off]</i> é o primeiro a ser gravado.
<code>void flush()</code>	Algumas vezes, a <i>stream</i> de saída pode acumular os bytes antes de gravá-los. O método <i>flush ()</i> força a gravação dos bytes.
<code>void close ()</code>	Fecha a <i>stream</i> .

Fonte: Sierra e Bates (2008).

Manipulação de arquivos: FileInputStream e FileOutputStream

A manipulação de arquivos em Java pode ser realizada com as classes *FileInputStream* e *FileOutputStream*. Ao contrário do teclado e do monitor, arquivos não se encontram abertos para leitura ou escrita. Para realizar essas operações, devem ser indicados de forma explícita no código.

Assim como o arquivo é aberto, ele deve ser fechado após seu uso. Na tabela 10, são apresentados os construtores e alguns métodos dessas classes.

Tabela 10 – Construtores e métodos das classes FileInputStream e FileOutputStream

FileInputStream	FileOutputStream
Construtores	
<i>FileInputStream(String n)</i>	Abre um arquivo para leitura que possui como nome uma <i>string</i> n. <i>FileOutputStream(String n, boolean a)</i>
	<i>FileOutputStream(String n)</i> Abre um arquivo para escrita que possui como nome um <i>string</i> n, se o booleano a for <i>true</i> , e adiciona o que for escrito no final do arquivo já existente. Caso seja <i>false</i> , deleta o arquivo existente.
<i>int available()</i>	Retorna o número de bytes disponíveis para serem lidos no <i>stream</i> .
<i>void close()</i>	Fechá o <i>stream</i> de entrada.
<i>int read()</i>	Lê um byte de dados.
<i>int read(byte[] b)</i>	Lê <i>b.length</i> bytes de dados e coloca no array <i>b</i> .
<i>long skip(long n)</i>	Descarta n bytes do <i>stream</i> .
	<i>void write(int b)</i> Grava o byte <i>b</i> no <i>stream</i> de saída.
	<i>void write(byte[] b)</i> Grava <i>b.length</i> bytes de <i>b</i> no <i>stream</i> de saída.

Fonte: Sierra e Bates (2008).

O exemplo apresentado no código dos quadros 74, 75 e 76 ilustra a gravação e a leitura de um objeto no arquivo. O objeto a ser gravado deve ser serializado, ou seja, a classe deve implementar *Serializable*.

Quadro 74: Classe Imovel

```

1 package com.ead.classes;
2
3 import java.io.Serializable;
4
5 public class Imovel implements Serializable {
6     private double area;
7     private String endereco;
8
9     public Imovel(double area, String endereco) {
10         this.setArea(area);
11         this.setEndereco(endereco);
12     }
13
14     public Double getArea() {
15         return area;
16     }
17     public void setArea(double area) {
18         this.area = area;
19     }
20     public String getEndereco() {
21         return endereco;
22     }
23     public void setEndereco(String endereco){
24         this.endereco = endereco;
25     }
26 }
```

Quadro 75: Classe GravarImovel

```

1 package com.ead.arquivos;
2
3 import java.io.FileOutputStream;
4 import java.io.ObjectOutputStream;
5
6 import javax.swing.JOptionPane;
7
8 import com.ead.classes.Imovel;
9
10 public class GravarImovel {
11     public static void main(String[] args) {
12         try {
13             Imovel imovel = new Imovel(125, "Av. paulista, 3545");
14             //Armazenar o objeto imovel em um arquivo
15             FileOutputStream fos =
16                 new FileOutputStream("C:/Curso_Java/imovel.dat");
17             ObjectOutputStream oos = new ObjectOutputStream(fos);
18             oos.writeObject(imovel);
19             oos.close();
20             JOptionPane.showMessageDialog(null, "Arquivo criado!");
21
22         } catch (Exception e) {
23             JOptionPane.showMessageDialog(null, e.getMessage());
24         }
25     }
26 }
```

O exemplo no quadro 76 ilustra a leitura do objeto.

Quadro 76: Classe LerImovel

```

1 package com.ead.arquivos;
2
3 import java.io.FileInputStream;
4 import java.io.ObjectInputStream;
5 import javax.swing.JOptionPane;
6 import com.ead.classes.Imovel;
7
8 public class LerImovel {
9     public static void main(String[] args) {
10         try {
11             FileInputStream fis =
12                 new FileInputStream("C:/Curso_Java/imovel.dat");
13             ObjectInputStream ois = new ObjectInputStream(fis);
14             Imovel imovel = (Imovel)ois.readObject();
15             JOptionPane.showMessageDialog(null, imovel.getArea() + ", " +
16                                         imovel.getEndereco());
17             ois.close();
18             fis.close();
19
20         } catch (Exception e) {
21             JOptionPane.showMessageDialog(null, e.getMessage());
22         }
23     }
24 }
25
26

```

Aula 19 - A classe File

Utilizando a classe File

A classe *File* é uma abstração para manipulação de arquivos em Java. A tabela 11 apresenta os principais métodos dela.

Tabela 11 – Métodos da classe File

Métodos	Descrição
<i>File(String n)</i>	Cria um novo objeto para referenciar o arquivo especificado.
<i>boolean canRead()</i>	Retorna <i>true</i> ou <i>false</i> , indicando se a aplicação pode ou não ler o arquivo.
<i>boolean canWrite()</i>	Retorna <i>true</i> ou <i>false</i> , indicando se a aplicação pode ou não escrever no arquivo.
<i>boolean compareTo(Object ob)</i>	Compara o nome do arquivo corrente com o arquivo passado por parâmetro.
<i>String getName()</i>	Retorna o nome do arquivo.
<i>String getPath()</i>	Retorna o nome do caminho do arquivo.
<i>String isFile()</i>	Retorna <i>true</i> se o objeto se refere a um arquivo.
<i>String isDirectory()</i>	Retorna <i>true</i> se o objeto se refere a um diretório.
<i>String[] list()</i>	Retorna um array de strings contendo todos os arquivos que constam no diretório.

Fonte: Sierra e Bates (2008).

O exemplo do quadro 77 apresenta um código que lista os arquivos e os diretórios a partir da pasta Windows, na unidade C.

Quadro 77: Classe ListaDeArquivos

```
1 package com.ead.arquivos;
2
3 import java.io.File;
4
5 public class ListaDeArquivos {
6     public static void main(String[] args) {
7         //Explorar a classe File
8         String pasta = "C:/Windows";
9         File file = new File(pasta);
10
11         if(file.isDirectory()){
12             System.out.println("\\" + pasta + "\" é um diretório:");
13         }
14
15         String[] arquivos = file.list();
16         for(String arquivo: arquivos){
17             File f = new File(pasta + "/" + arquivo);
18             if(f.isDirectory()){
19                 System.out.println("pasta    -> " + arquivo);
20             }
21             else {
22                 System.out.println("arquivo -> " + arquivo);
23             }
24         }
25     }
26 }
```

FileReader e FileWriter

A manipulação de arquivos de acesso sequencial, ou arquivos texto, dá-se por meio das classes *FileReader* e *FileWriter*, cuja hierarquia é apresentada na figura 14.

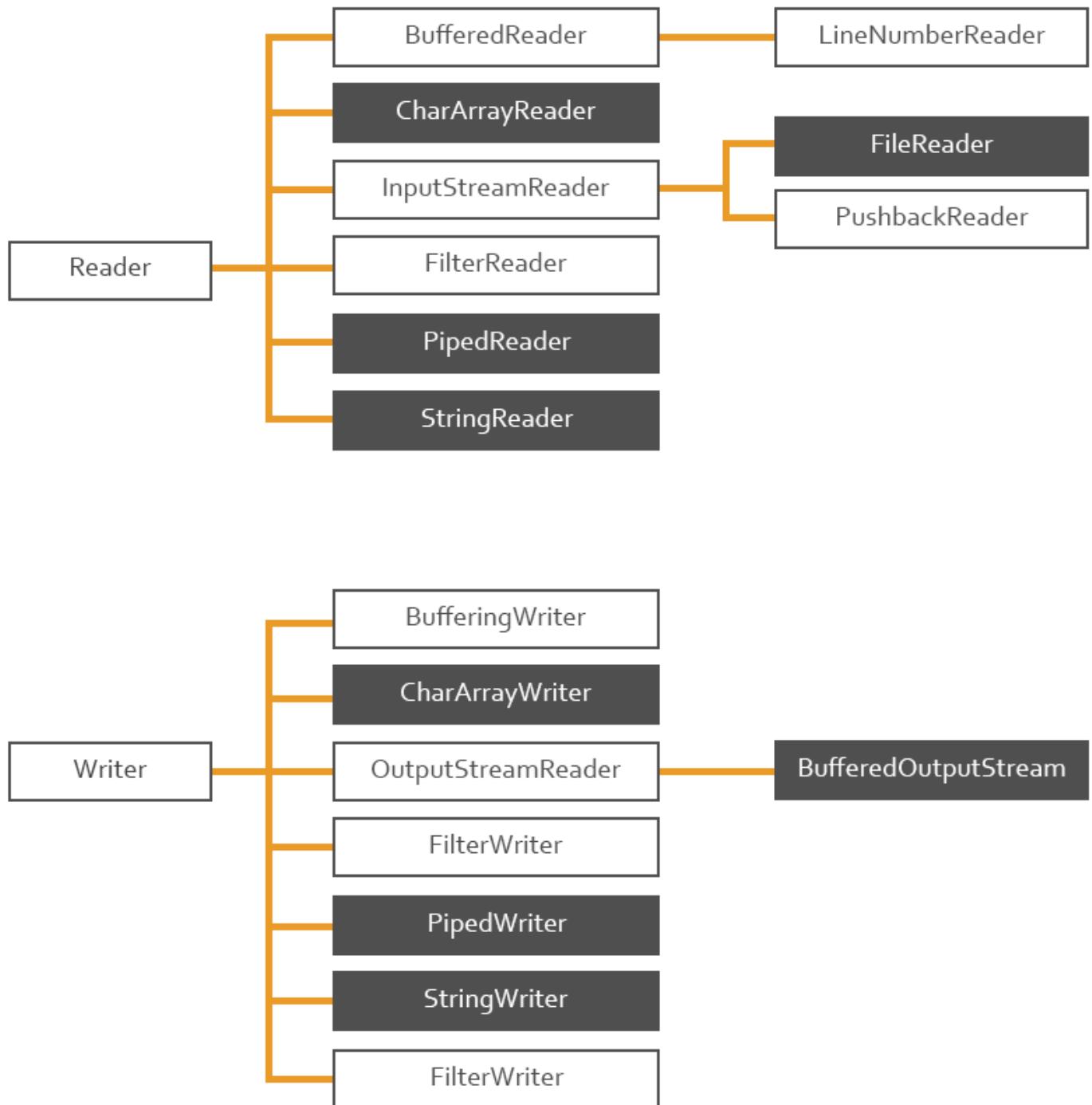


Figura 14 - Hierarquia de classes Reader / Writer. Fonte: <http://flylib.com/books/en/2.33.1.73/1/>.

A tabela 12 apresenta alguns dos serviços disponíveis nessas classes. Essas operações acessam diretamente os arquivos especificados a cada operação

Tabela 19.2 – Métodos das classes FileReader e FileWriter

FileReader	FileWriter		
<i>FileReader(File f)</i>	Cria um objeto para manipular a leitura de um arquivo especificado pelo objeto <i>File f</i> .	<i>FileWriter(File f)</i>	Cria um objeto para manipular a escrita em um arquivo especificado pelo objeto <i>File f</i> .
<i>FileReader(String n)</i>	Cria um objeto para manipular a leitura um arquivo especificado pelo nome n.	<i>FileWriter(String n)</i>	Cria um objeto para manipular a escrita de um arquivo especificado pelo nome n.
<i>close, getEncoding, read, ready</i>	Métodos herdados de InputStreamReader.	<i>FileWriter(String n, append a)</i>	Cria um objeto para manipular a escrita de um arquivo especificado pelo nome n; se o parâmetro a for <i>true</i> , escreve no final do arquivo.
		<i>close, flush, getEncoding, write, write, write</i>	Métodos herdados de OutputStreamWriter.

Fonte: Sierra e Bates (2008).

O exemplo do quadro 78 realiza uma leitura de um arquivo texto caráter por caráter e exibe-os na tela.

Quadro 78: Classe LerArquivoTexto: leitura de caracteres

```

1 package com.ead.arquivos;
2
3 import java.io.FileReader;
4
5 import javax.swing.JOptionPane;
6
7 public class LerArquivoTexto {
8     public static void main(String[] args) {
9
10         try {
11             FileReader arquivo =
12                 new FileReader("C:/Curso_Java/TesteParametros.java");
13             while(true){
14                 int c = arquivo.read();
15                 if(c == -1){ //EOF
16                     break;
17                 }
18                 System.out.print((char)c);
19             }
20
21         } catch (Exception e) {
22             JOptionPane.showMessageDialog(null, e.getMessage());
23         }
24     }
25 }
```

O exemplo do quadro 79 realiza a leitura considerando uma linha de cada vez, ao invés de caracteres.

Quadro 79: Classe LerArquivoTexto02: leitura de linha

```

1 package com.ead.arquivos;
2
3 import java.io.BufferedReader;
4 import java.io.FileReader;
5 import javax.swing.JOptionPane;
6
7 public class LerArquivoTexto02 {
8     public static void main(String[] args) {
9
10         try {
11             FileReader arquivo =
12                 new FileReader("C:/Curso_Java/TesteParametros.java");
13             BufferedReader br = new BufferedReader(arquivo);
14             String texto = "";
15             while(true){
16                 String linha = br.readLine();
17                 if(linha == null){
18                     break;
19                 }
20                 texto += linha + "\r\n";
21             }
22
23         } catch (Exception e) {
24             JOptionPane.showMessageDialog(null, e.getMessage());
25         }
26     }
27 }
--
```

O exemplo do quadro 80 realiza gravação (saída) em um arquivo texto.

Quadro 80: Classe GravaArquivoTexto

```

1 package com.ead.arquivos;
2
3 import java.io.FileWriter;
4 import java.io.IOException;
5
6 import javax.swing.JOptionPane;
7
8 public class GravaArquivoTexto {
9     public static void main(String[] args) {
10         FileWriter arquivo;
11         try {
12             arquivo = new FileWriter("C:/Curso_Java/exemplo.txt");
13             arquivo.write("Aula sobre Arquivos\r\n - Hello World");
14             arquivo.close();
15
16             JOptionPane.showMessageDialog(null, "Arquivo criado!");
17         } catch (IOException e) {
18             e.printStackTrace();
19         }
20     }
21 }
```

Finalmente, o exemplo do quadro 81 apresenta um código que lê uma lista de nomes e os grava em um arquivo.

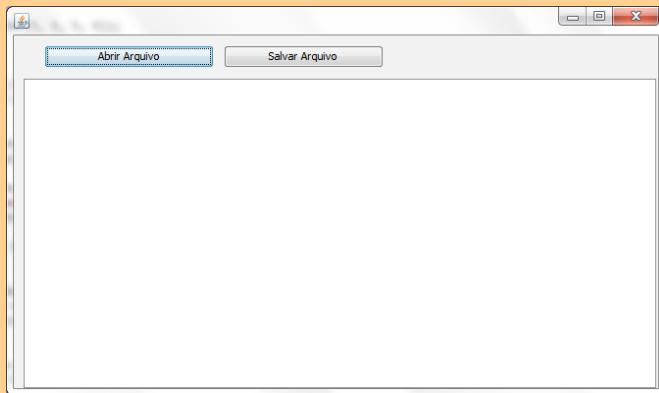
Quadro 81: Classe GravaArquivoTexto03

```
1 package com.ead.arquivos;
2
3 import java.io.FileWriter;
4 import java.io.IOException;
5 import java.util.ArrayList;
6 import java.util.List;
7
8 import javax.swing.JOptionPane;
9
10 public class GravaArquivoTexto03 {
11     public static void main(String[] args) {
12         FileWriter arquivo;
13         //Criar lista de nomes
14         List<String> nomes = new ArrayList<String>();
15         nomes.add("Thiago");
16         nomes.add("Vampeta");
17         nomes.add("Túlio");
18         nomes.add("Jair");
19
20         try {
21             arquivo = new FileWriter("C:/Curso_Java/nomes.txt", true);
22             for(String nome: nomes){
23                 arquivo.write(nome + '\n');
24             }
25             arquivo.close();
26
27             JOptionPane.showMessageDialog(null, "Arquivo criado!");
28         } catch (IOException e) {
29             e.printStackTrace();
30         }
31     }
32 }
```

Exercícios do Módulo 6

Exercício 1: Criação de um editor de textos

- a. Criar uma interface gráfica representando um editor de textos, conforme modelo a seguir:



- b. Para o evento de ação do botão “Abrir Arquivo”, escrever uma instrução para gerar uma caixa de diálogo solicitando o arquivo a ser aberto e exibido na caixa de textos. A classe a ser utilizada nesse evento é JFileChooser, e a instrução é semelhante a:

```
JFileChooser chooser = new JFileChooser();
try {
    if(chooser.showOpenDialog(EditorTextos.this) ==
        JFileChooser.APPROVE_OPTION){
        //seu código aqui

    } catch (Exception e) {
        JOptionPane.showMessageDialog(EditorTextos.this, e.getMessage());
    }
}
```

- c. De forma semelhante, para o evento de ação do botão “Salvar Arquivo”, escrever um código que apresente um diálogo onde o usuário deverá fornecer o nome do arquivo a ser salvo:

```
JFileChooser chooser = new JFileChooser();
try {
    if(chooser.showSaveDialog(EditorTextos.this) ==
        JFileChooser.APPROVE_OPTION){
        //seu código aqui

    } catch (Exception e) {
        JOptionPane.showMessageDialog(EditorTextos.this, e.getMessage());
    }
}
```

- d. Escrever uma classe chamada TesteEditor, contendo o método *main()*, que instancie a classe *EditorTextos* e a apresente na tela.

Exercício 2: Leitura parcial de arquivos.

O objetivo deste exercício é apresentar uma amostra do conteúdo de um arquivo. Para tanto, apenas os 20 primeiros caracteres do arquivo são lidos, e seu conteúdo é seguido por três pontos (...).

Escrever um programa que leia um arquivo texto caráter por caráter e acumule apenas os 20 primeiros caracteres, seguidos por três pontos. Se o conteúdo do arquivo tiver menos de 20 caracteres, mostrar todo seu conteúdo.

Módulo 7 - Acesso a Banco de Dados JDBC

Aula 20 - Conectividade com BD

Uma funcionalidade essencial em qualquer sistema é a habilidade para comunicar-se com um repositório de dados. Podemos definir repositório de dados de várias maneiras, por exemplo, como um conjunto de objetos de negócio em um sistema de arquivos ou um banco de dados.

Java DataBase Connectivity API

Bancos de dados constituem o tipo mais comum de repositório. Java dispõe de uma API para acessar repositórios de dados: Java DataBase Connectivity API, ou JDBC API.

A maioria dos fornecedores de bancos de dados oferece uma implementação particular de *drivers* para acesso a banco de dados. A vantagem de JDBC é a portabilidade da aplicação cliente, inerente à linguagem Java.

A arquitetura da API JDBC é apresentada na figura 15.

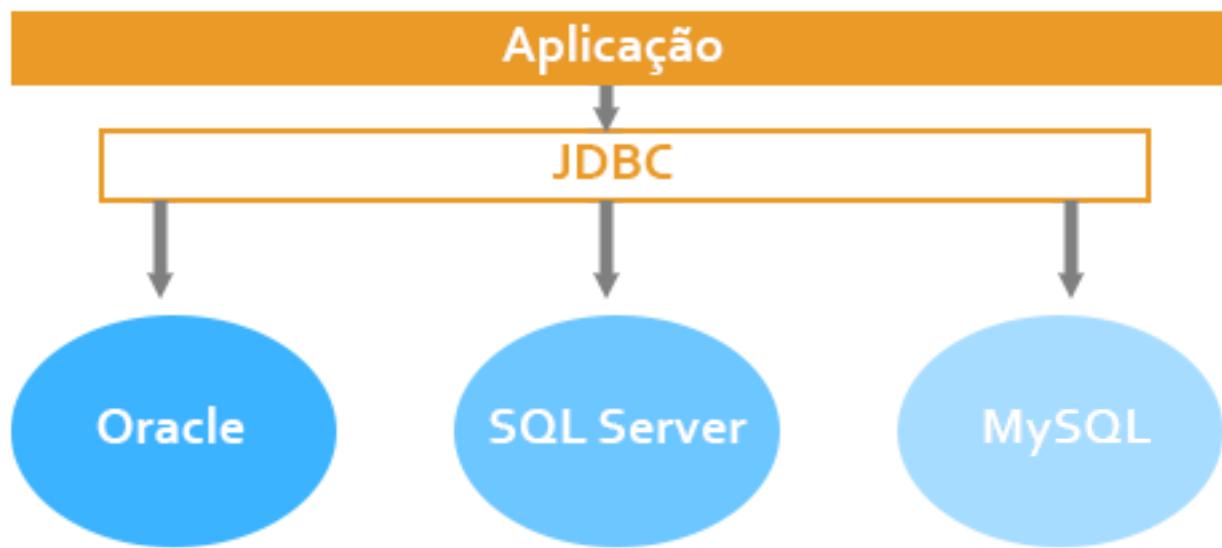


Figura 15. Arquitetura JDBC. Fonte: <http://desenvolvedorinteroperavel.wordpress.com/2011/07/05/conectando-banco-de-dados-mysql-em-java-parte-1/>.

A API JDBC compreende uma especificação para os desenvolvedores de *drivers* JDBC e os desenvolvedores de aplicações clientes que precisem acessar bancos de dados em Java.

Como esse assunto é de extrema importância, estudaremos o acesso a banco de dados em Java por intermédio de aplicações.

Tipos de *drivers* JDBC

Existem quatro tipos de diferentes de *drivers* JDBC:

1. O *driver* mais comum fornecido pela Oracle é o *JDBC-ODBC bridge* (ponte JDBC-ODBC). Esse tipo de *driver* não é portável, pois depende de chamadas a funções de ODBC - *Open DataBase Connectivity* implementadas em linguagem C e compiladas para Wintel, ou outra plataforma ODBC compatível, denominadas funções nativas.
2. O *driver* tipo 2 é implementado parcialmente em Java e parcialmente por meio de funções nativas que implementam alguma API específica do fornecedor de banco de dados. Esse tipo faz o que se chama de *wrap-out*, ou seja, provê uma interface Java para uma API nativa não Java.
3. O tipo 3 é um *driver* totalmente Java que se comunica com algum tipo de *middleware*, que então se comunica com o banco de dados.
4. O tipo 4 é um *driver* totalmente Java, ou seja, é nativo da linguagem.

Utilizando o *driver* JDBC

Para utilizarmos a JDBC em um programa em Java, precisamos declarar o pacote que contém a JDBC API:

```
Import java.sql.*;
```

Esse pacote permite declarar os objetos de acesso a dados.

A seguir, apresentaremos um roteiro para acessarmos o banco de dados. Trata-se de uma tabela chamada *contatos* em um banco de dados MySQL, cujo nome é *agenda*. O campo ID é autoincrementável. A tabela *contatos* é apresentada na figura 16.

contatos	
ID	INT(11)
NOME_CONTATO	VARCHAR(50)
TELEFONE_CONTATO	VARCHAR(50)
EMAIL_CONTATO	VARCHAR(50)
DATA_CADASTRO	DATE
Indexes	

Figura 16 - Tabela *contatos*.

Inserindo o *connector* Java para acesso ao MySQL

A primeira coisa a fazer é estabelecer uma conexão com o banco de dados. O banco de dados MySQL possui seu próprio *driver* de acesso, também chamado de *connector* Java. Esse *connector* pode ser obtido na própria página do banco MySQL (www.mysql.com) ou acessando diretamente o *link*: <http://dev.mysql.com/downloads/connector/>.

Novamente, verifique se o *link* foi alterado ou o acesso atualizado. Assim, a página que você deve acessar é semelhante à mostrada na figura 17, porque a web possui natureza dinâmica e pode mudar a forma de acesso a qualquer momento.

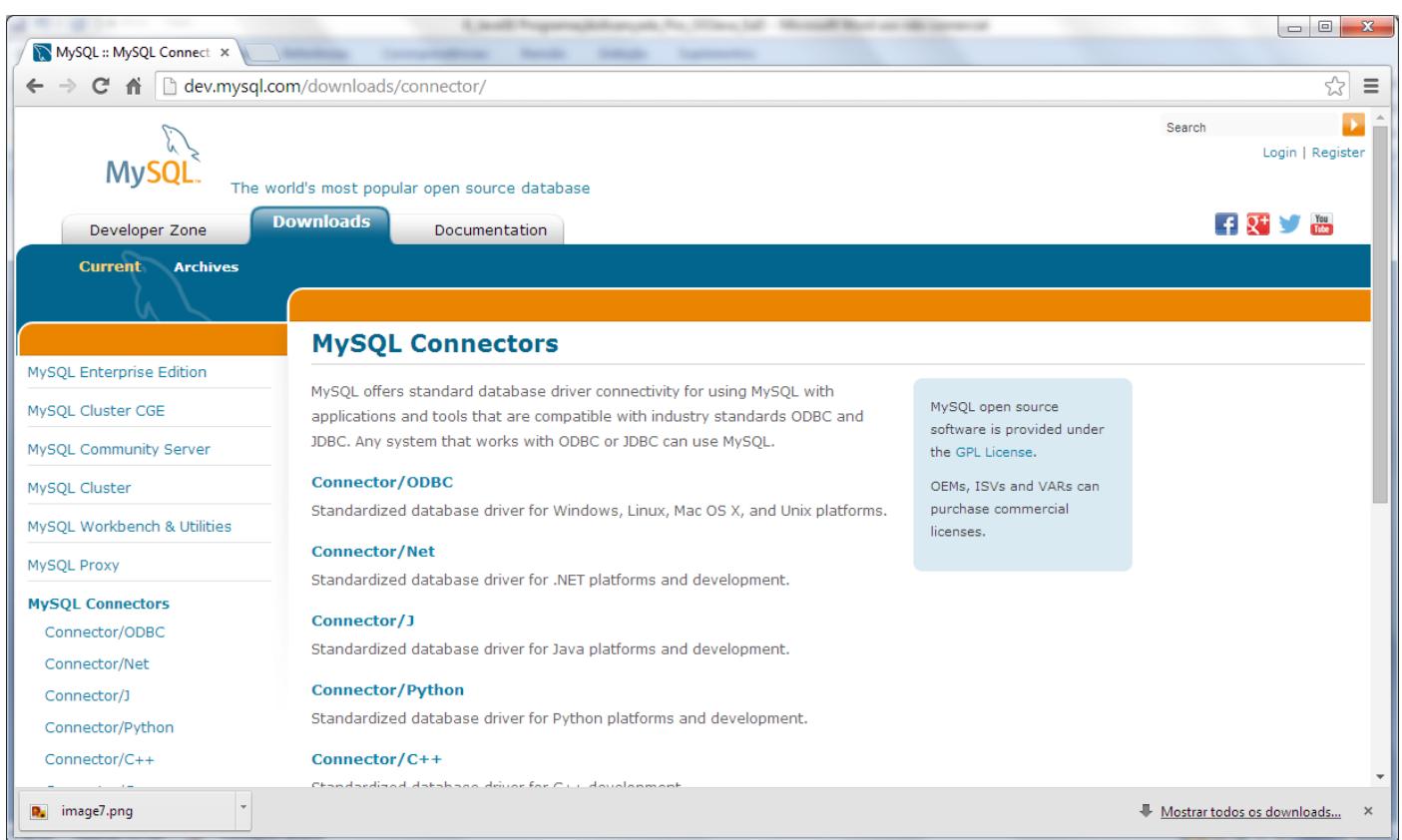


Figura 17 - Página para download do driver MySQL. Fonte: <http://dev.mysql.com/downloads/connector/>.

Uma vez com o *connector*, criamos um novo projeto no Eclipse e inserimo-lo. A maneira mais prática de fazer isso é clicando com o botão direito do mouse sobre o projeto, selecionando a opção *Build Path*, e, em seguida, *Configure Build Path...*, como mostra a figura 18.

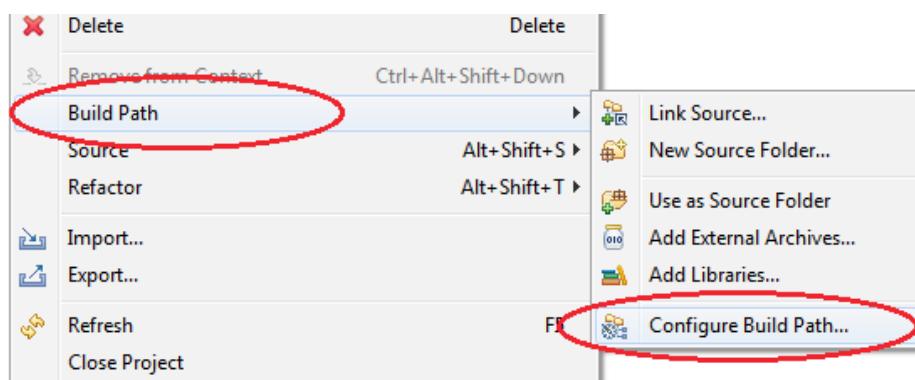


Figura 18 - Inserindo o *connector*.

Em seguida, acessamos a aba *Libraries*, conforme mostrado na figura 19.

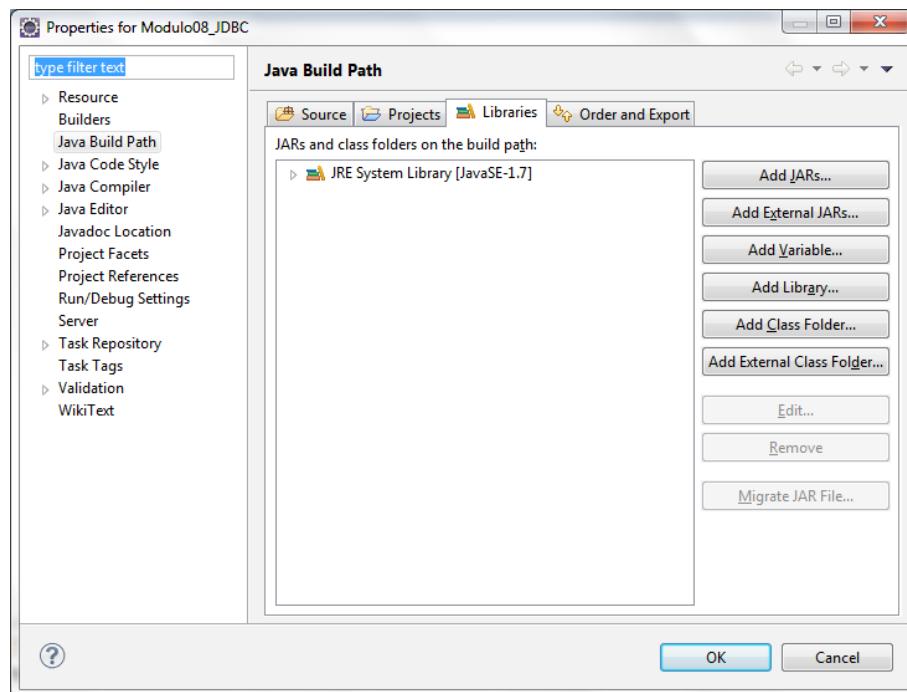


Figura 19 - Tela de inclusão do *driver MySQL*.

Em seguida, clique no botão *Add External JARs...* e selecione o arquivo .jar correspondente ao *driver* do MySQL. Aceite a opção clicando em *OK*. A figura 20 apresenta esse processo.

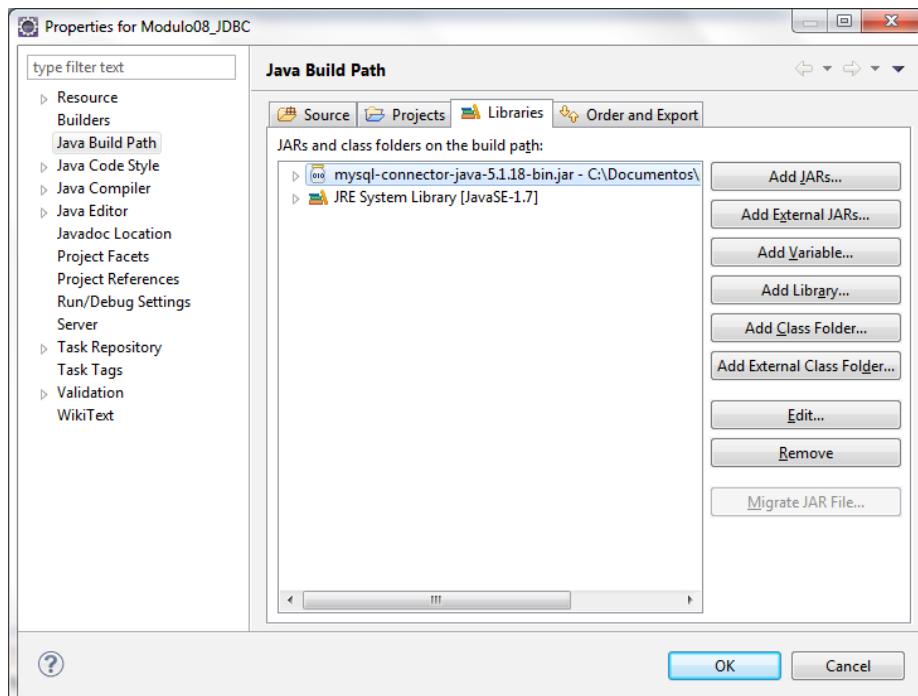


Figura 20 - Tela de inclusão do *driver MySQL*.

Agora estamos prontos para trabalhar.

Aula 21 - Objetos de acesso a dados

JDBC é um padrão no qual está definida uma estrutura geral de acesso ao *driver* de banco de dados por meio de interfaces e o fornecedor se encarrega de implementar as classes que concretamente vão realizar o serviço.

Criaremos uma classe com os métodos que realizarão as operações básicas no banco de dados: INSERIR, REMOVER, BUSCAR e ALTERAR. Esse grupo de operações é tradicionalmente conhecido como *CRUD* (*Create, Recover, Update, Delete*).

Declarando objetos de acesso a dados usando interfaces

A declaração dos objetos de acesso a dados é mostrada no código do quadro 82.

Quadro 82: Classe DaoContatos: declaração de objetos de acesso a dados

```

1 package com.ead.dao;
2 import java.sql.*;
3
4 public class DaoContatos {
5
6     private Connection cn;
7     private PreparedStatement stmt;
8     private ResultSet rs;
9
10 }
```

- A interface *Connection* designa um objeto, no caso *con*, para receber a conexão estabelecida.
- A interface *PreparedStatement* permite-nos usufruir de SQL armazenado ou pré-compilado no banco, quando o banco de dados suportar esse recurso.
- A interface *ResultSet* permite colher os resultados da execução de nossa *query* no banco de dados. Essa interface apresenta uma série de métodos para prover o acesso aos dados.

Definindo a *string* de conexão

Agora devemos definir uma *string* de conexão para o nosso banco de dados *agenda*. Uma *string* de conexão contém as informações necessárias para o acesso, e cada banco de dados possui sua própria estrutura de conexão. A linha 10 do código no quadro 83 mostra a *string* de conexão par ao banco de dados MySQL.

Quadro 83: Classe DaoContatos reescrita: definição da string de conexão

```

1 package com.ead.dao;
2 import java.sql.*;
3
4 public class DaoContatos {
5
6     private Connection cn;
7     private PreparedStatement stmt;
8     private ResultSet rs;
9
10    private String url = "jdbc:mysql://localhost:3306/agenda";
11 }

```

Em seguida, desenvolveremos os métodos para abrir e fechar a conexão com o banco de dados. Fazemos isso em dois passos. Primeiro, carregamos o *driver* para o banco de dados na JVM da aplicação.

A classe DriverManager

Cada fornecedor tem o seu *driver* específico, construído como uma classe JDBC. A chamada à função *forName(classe)* registra a classe nomeada na JVM corrente.

Uma vez carregado, o *driver* registra-se para o *DriverManager* e está disponível para a aplicação. Utilizamos, então, a classe *DriverManager* para abrir uma conexão com o banco de dados. O quadro 84 apresenta o código para essa finalidade.

Quadro 84: Classe DaoContatos reescrita: definição do método abrirConexao()

```

1 package com.ead.dao;
2 import java.sql.*;
3
4 public class DaoContatos {
5
6     private Connection cn;
7     private PreparedStatement stmt;
8     private ResultSet rs;
9
10    private String url = "jdbc:mysql://localhost:3306/agenda";
11
12    private void abrirConexao() throws Exception {
13        try {
14            Class.forName("com.mysql.jdbc.Driver");
15            cn = DriverManager.getConnection(url,"root","root");
16        } catch (Exception e) {
17            throw e;
18        }
19    }
20
21    private void fecharConexao() throws Exception{
22        cn.close();
23    }
24 }

```

Estabelecida a conexão, podemos executar comandos SQL para manipular o banco de dados. Neste exemplo, trabalharemos com o paradigma orientado a objetos. Para tanto, criaremos uma classe que representará as entidades a ser mapeadas para o banco de dados.

Como temos uma tabela chamada *CONTATOS*, criamos uma classe chamada *Contatos* (a coincidência de nomes é opcional) para definir objetos a ser usados como elementos de entidade. Em outras palavras, cada objeto da nossa classe *Contatos* representará um registro na tabela *CONTATOS*. A classe *Contatos* está definida no quadro 85.

Quadro 85: Classe Contatos

```

1 package com.ead.entidade;
2
3 import java.util.Date;
4
5 public class Contatos {
6     private int id;
7     private String nome, telefone, email;
8     private Date data;
9
10+    public int getId() {..}
13+    public void setId(int id) {..}
16+    public String getNome() {..}
19+    public void setNome(String nome) {..}
22+    public String getTelefone() {..}
25+    public void setTelefone(String telefone) {..}
28+    public String getEmail() {..}
31+    public void setEmail(String email) {..}
34+    public Date getData() {..}
37+    public void setData(Date data) {..}
40 }
```

Criando métodos para realizar as operações CRUD - *Create, Recover, Update, Delete*

Agora, criaremos métodos para:

- **Inserir um registro:** o método receberá como parâmetro um objeto da classe *Contatos* para persisti-lo no banco de dados. O código para esse método está ilustrado no quadro 86.

Quadro 86: Método incluir: usa um objeto da classe *Contatos* como parâmetro para a inclusão no banco de dados

```

28+    public void incluir(Contatos contato) throws Exception{
29        try {
30            abrirConexao();
31            stmt = cn.prepareStatement("INSERT INTO CONTATOS (NOME_CONTATO,TELEFONE_CONTATO," +
32                                    "EMAIL_CONTATO,DATA_CADASTRO) VALUES (?,?,?,?,?)");
33            stmt.setString(1, contato.getNome());
34            stmt.setString(2, contato.getTelefone());
35            stmt.setString(3, contato.getEmail());
36            stmt.setDate(4, new java.sql.Date(contato.getData().getTime()));
37
38            stmt.executeUpdate();
39        } catch (Exception e) {
40            throw e;
41        } finally {
42            fecharConexao();
43        }
44 }
```

- **Remover um registro:** recebe como parâmetro o código (ID) do contato a ser removido. O código para remover o registro está apresentado no quadro 87.

Quadro 87: Método *remover()*: recebe o valor do código do contato (parâmetro *id*) para remover o contato

```

46 public void remover(int id) throws Exception{
47     try {
48         abrirConexao();
49         stmt = cn.prepareStatement("DELETE FROM CONTATOS WHERE ID=?");
50         stmt.setInt(1, id);
51
52         stmt.executeUpdate();
53     } catch (Exception e) {
54         throw e;
55     } finally {
56         fecharConexao();
57     }
58 }
```

- **Buscar um registro:** recebe como parâmetro o código do contato e retorna o objeto em questão. Se não houver registro com o código informado, o método retornará *null*. Esse código está apresentado no quadro 88.

Quadro 88: Método *buscar()*: recebe como parâmetro o código do contato (parâmetro *id*) e retorna o objeto com esse código, ou *null* se não houver objeto com o código especificado

```

60 public Contatos buscar(int id) throws Exception{
61     Contatos contato = null;
62     try {
63         abrirConexao();
64         stmt = cn.prepareStatement("SELECT * FROM CONTATOS WHERE ID=?");
65         stmt.setInt(1, id);
66         rs = stmt.executeQuery();
67
68         if(rs.next()){
69             contato = new Contatos();
70             contato.setId(id);
71             contato.setNome(rs.getString("NOME_CONTATO"));
72             contato.setTelefone(rs.getString("TELEFONE_CONTATO"));
73             contato.setEmail(rs.getString("EMAIL_CONTATO"));
74             contato.setData(rs.getDate("DATA_CADASTRO"));
75         }
76     } catch (Exception e) {
77         throw e;
78     } finally {
79         fecharConexao();
80     }
81     return contato;
82 }
```

- **Alterar um registro:** recebe como parâmetro o objeto *Contatos* a ser removido, tomando como base seu ID. O código para alterar um registro está apresentado no quadro 89.

Quadro 89: Método *alterar()*: recebe como parâmetro um objeto da classe Contatos com os valores a ser atualizados

```

110 public void alterar(Contatos contato) throws Exception{
111     try {
112         abrirConexao();
113         stmt = cn.prepareStatement("UPDATE CONTATOS SET NOME_CONTATO=? ,TELEFONE_CONTATO=? , "
114                                     + "EMAIL_CONTATO=? ,DATA_CADASTRO=? WHERE ID=?");
115         stmt.setString(1, contato.getNome());
116         stmt.setString(2, contato.getTelefone());
117         stmt.setString(3, contato.getEmail());
118         stmt.setDate(4, new java.sql.Date(contato.getData().getTime()));
119         stmt.setInt(5, contato.getId());
120
121         stmt.executeUpdate();
122     } catch (Exception e) {
123         throw e;
124     } finally {
125         fecharConexao();
126     }
127 }
```

- **Listar todos os contatos:** o método retornará uma lista contendo todos os contatos. Esse código está ilustrado no quadro 90.

Quadro 90: Método *listar()*: retorna uma lista parametrizada para Contatos, que armazena todos os contatos contidos no banco de dados

```

86 public List<Contatos> listar() throws Exception{
87     List<Contatos> contatos = new ArrayList<Contatos>();
88     try {
89         abrirConexao();
90         stmt = cn.prepareStatement("SELECT * FROM CONTATOS");
91         rs = stmt.executeQuery();
92
93         while(rs.next()){
94             Contatos contato = new Contatos();
95             contato.setId(rs.getInt("ID"));
96             contato.setNome(rs.getString("NOME_CONTATO"));
97             contato.setTelefone(rs.getString("TELEFONE_CONTATO"));
98             contato.setEmail(rs.getString("EMAIL_CONTATO"));
99             contato.setData(rs.getDate("DATA_CADASTRO"));
100
101             contatos.add(contato);
102         }
103     } catch (Exception e) {
104         throw e;
105     } finally {
106         fecharConexao();
107     } return contatos;
108 }
```

O código do quadro 91 apresenta um exemplo de execução de alguns desses métodos.

Quadro 91: Classe TesteBancoDados: instancia a classe Contatos, inclui esta instância no banco de dados e lista todos os contatos

```
1 package com.ead.programa;
2
3 import java.util.Date;
4 import java.util.List;
5
6 import javax.swing.JOptionPane;
7
8 import com.ead.dao.DaoContatos;
9 import com.ead.entidade.Contatos;
10
11 public class TesteBancoDados {
12     public static void main(String[] args) {
13
14         //inserindo um registro
15         Contatos contato = new Contatos();
16         contato.setNome("Jeremias Santos");
17         contato.setTelefone("2254-9800");
18         contato.setEmail("jeremias@email.com");
19         contato.setData(new Date());
20
21         DaoContatos dao = new DaoContatos();
22         try {
23             dao.incluir(contato);
24             JOptionPane.showMessageDialog(null, "Contato inserido com sucesso!");
25
26             //listando os contatos
27             List<Contatos> contatos = dao.listar();
28             for(Contatos c: contatos){
29                 System.out.println("ID: " + c.getId());
30                 System.out.println("NOME: " + c.getNome());
31             }
32         } catch (Exception e) {
33             JOptionPane.showMessageDialog(null, e.getMessage());
34         }
35     }
36 }
```

Hierarquia de classes JDBC

A dependência entre as classes e interfaces JDBC é ilustrada na figura 21.

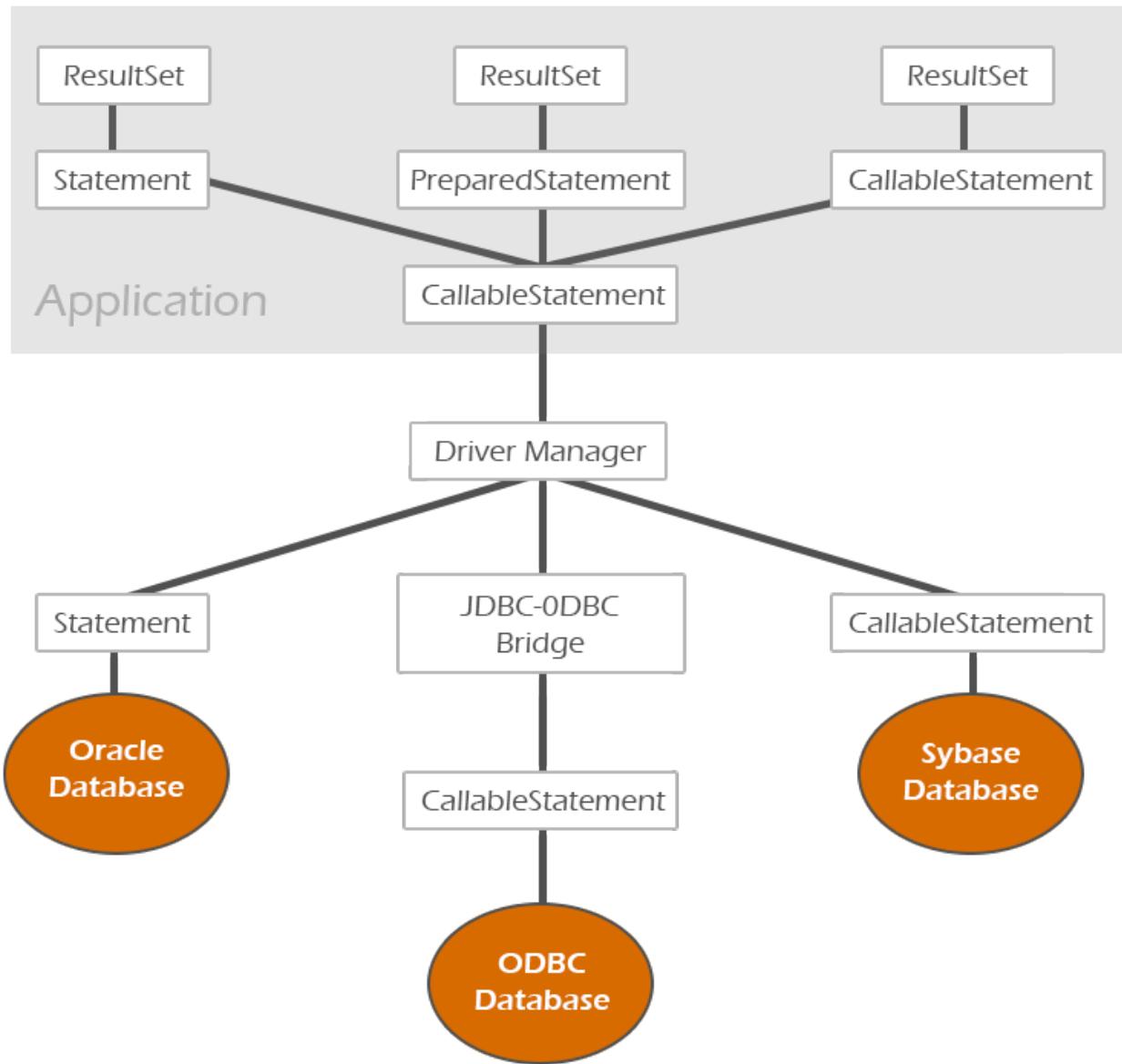


Figura 21. Hierarquia de classes JDBC.

Métodos das interfaces PreparedStatement e ResultSet

Desenvolvemos no nosso exemplo de aplicação diversos métodos, tanto de `PreparedStatement` como de `ResultSet`.

As instruções SQL nos nossos exemplos utilizam o conceito de “curinga”, ou seja, ela espera receber os dados por intermédio de parâmetros:

```
stmt = cn.prepareStatement("INSERT INTO CONTATOS (NOME_CONTATO,TELEFONE_CONTATO," +  
"EMAIL_CONTATO,DATA_CADASTRO) VALUES (?,?,?,?,?)");
```

Temos aqui quatro pontos de interrogação, representando uma incógnita a ser preenchida na execução da instrução. Dependendo do tipo de dado relacionado ao banco de dados, temos um tipo compatível em Java para fornecer. Esses parâmetros são preenchidos por meio do método `setXXX()`, em que XXX representa o tipo a ser usado.

Esse método recebe dois parâmetros. O primeiro representa a posição do parâmetro na instrução SQL. Se for o primeiro, seu valor é 1; se for o segundo, seu valor é 2, e assim por diante. O segundo parâmetro representa o dado a ser inserido, de acordo com o tipo representado por XXX.

As instruções no quadro 92 mostram como informar parâmetros.

Quadro 92: Códigos para informar parâmetros para os campos no banco de dados

```
stmt.setString(1, contato.getNome());  
stmt.setString(2, contato.getTelefone());  
stmt.setString(3, contato.getEmail());  
stmt.setDate(4, new java.sql.Date(contato.getData().getTime()));
```

O método `setString()` recebe como segundo parâmetro uma *string*; o método `setDate()` espera receber um objeto do tipo *java.sql.Date*.

A classe Date do pacote *java.sql* é, na verdade, uma subclasse da classe Date do pacote *java.util*.

De forma semelhante, a interface *ResultSet* possui métodos para recuperar os valores dos campos no banco de dados. Esses métodos possuem o formato `getXXX()`, sendo XXX o tipo correspondente no banco de dados.

Na tabela 17, temos uma relação de tipos no banco de dados que correspondem a tipos no Java.

Tabela 17 – Relação entre os campos no banco de dados e os tipos Java

Tipo de Dados SQL	Tipo de Dados Java
CHAR	<i>String</i>
VARCHAR	<i>String</i>
LONGVARCHAR	<i>String</i>
NUMERIC	<i>java.math.BigDecimal</i>
DECIMAL	<i>java.math.BigDecimal</i>
BIT	<i>java.math.BigDecimal</i>
TINYINT	<i>boolean</i>

SMALLINT	byte
INTEGER	short
BIGINT	long
REAL	float
FLOAT	double
DOUBLE	double
BINARY	byte[]
VARBINARY	byte[]
LONGVARBINARY	byte[]
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp

Fonte: Produção do autor.

Aula 22 - *Stored procedures* no MySQL

Quando desenvolvemos aplicações que acessam banco de dados, é comum executarmos rotinas complexas de manipulação desses dados.

Dependendo da rotina a ser executada, isso pode requerer várias consultas e atualizações na base, o que acarreta um maior consumo de recursos pela aplicação. No caso de aplicações *web*, isso se torna ainda mais crítico devido à maior quantidade de informações que precisam trafegar pela rede e de requisições ao servidor.

Uma boa forma de contornar esse consumo de recursos diretamente pela aplicação é transferir parte do processamento direto para o banco de dados. Assim, considerando que as máquinas servidoras geralmente têm configurações de *hardware* mais robustas, enquanto nada se pode garantir com relação às máquinas clientes, essa pode ser uma alternativa a se considerar.

Para isso, usamos os *stored procedures*. Eles são rotinas definidas no banco de dados e identificadas por um nome pelo qual podem ser invocadas. Um procedimento desses pode executar uma série de instruções, receber parâmetros e retornar valores.

Criando e invocando *stored procedures* no MySQL

Explicaremos a seguir como trabalhar com *procedures* no banco de dados MySQL. Para iniciar, vamos entender como é a sintaxe utilizada para criação desse tipo de objeto:

```
DELIMITER $$
CREATE PROCEDURE nome_procedimento (parâmetros)
BEGIN
/*CORPO DO PROCEDIMENTO*/
END $$
DELIMITER ;
```

Na sintaxe mostrada, nome_procedimento refere-se ao nome que identificará a *procedure*.

Os parâmetros são opcionais, mas, se fornecidos, devem seguir a sintaxe (dentro dos parênteses):

(MODO nome TIPO, MODO nome TIPO, MODO nome TIPO)

O TIPO é o tipo de dado do parâmetro (int, varchar, decimal etc.).

O MODO indica a forma como o parâmetro será tratado no procedimento, se será apenas um dado de entrada, apenas de saída ou se terá ambas as funções. Os valores possíveis para o modo são:

- IN: indica que o parâmetro é apenas para entrada/recebimento de dados, não podendo ser usado para retorno.
- OUT: usado para parâmetros de saída. Para esse tipo, não pode ser informado um valor direto (como "teste", 1 ou 2.3), mas deve ser passada uma variável por referência.
- INOUT: como é possível imaginar, esse tipo de parâmetro pode ser usado para os dois fins (entrada e saída de dados). Nesse caso, também deve ser informada uma variável e não um valor direto.

Chamando uma *procedure* MySQL em Java

A interface *CallableStatement* permite executar procedimentos e funções armazenados no banco de dados. O quadro 93 apresenta um exemplo de como essa chamada é realizada.

Quadro 93: Método *buscarProcedure()*: define um objeto para acessar uma *stored procedure*

```

131     public Contatos buscarProcedure(int id) throws Exception{
132         Contatos contato = null;
133         try {
134             abrirConexao();
135
136             CallableStatement cs = cn.prepareCall("{call Buscar_Contato(?)}");
137
138             cs.execute();
139             stmt.setInt(1, id);
140             rs = stmt.executeQuery();
141
142             if(rs.next()){
143                 contato = new Contatos();
144                 contato.setId(id);
145                 contato.setNome(rs.getString("NOME_CONTATO"));
146                 contato.setTelefone(rs.getString("TELEFONE_CONTATO"));
147                 contato.setEmail(rs.getString("EMAIL_CONTATO"));
148                 contato.setData(rs.getDate("DATA_CADASTRO"));
149             }
150         } catch (Exception e) {
151             throw e;
152         } finally {
153             fecharConexao();
154         }
155         return contato;
156     }

```

Exercícios do Módulo 7

Nestes exercícios, aproveitaremos a interface gráfica definida no Módulo 5.

Exercício 1: Aproveitamento da interface gráfica.

- Em um novo projeto, copiar as classes *AlunosGUI* e *Aluno* do projeto que você definiu no Módulo 5.
- Criar uma classe chamada *AlunosDAO*.
- Nessa classe, definir um método para abrir uma conexão com o banco de dados e outro para fechar a conexão. As assinaturas deles deverão ser análogas às sugeridas abaixo:

private void abrirConexao() throws Exception

private void fecharConexao() throws Exception

- Incluir um método para incluir um aluno no banco de dados. Esse método deverá ter a assinatura:

public void incluir(Aluno aluno) throws Exception

- Incluir um método para retornar a uma lista com todos os alunos. Esse método deverá ter a seguinte assinatura:

public List<Aluno> listarAlunos()

Exercício 2: Definição das funcionalidades.

- No evento de ação do botão “Incluir”:

- Criar um objeto da classe Aluno.
 - Atribuir os valores dos atributos dessa classe a partir dos componentes da interface gráfica.
 - Instanciar a classe AlunosDAO. A instância deve ser chamar *dao*.
 - A partir dessa instância, executar o método *incluir()*, informando como parâmetro o objeto da classe Aluno definido anteriormente.
- b. No evento de ação do botão “Listar Alunos”:
- Definir uma referência do tipo *List<Aluno>*.
 - Definir uma nova instância à classe AlunosDAO.
 - Executar o método *listarAlunos()* e atribuí-lo à referência que você criou.
 - Executar uma estrutura de repetição sobre essa lista e exibir seus dados no componente JList da interface.

Considerações Finais

Caro aluno, estamos encerrando mais uma etapa de nosso curso.

Pudemos aplicar os conceitos de herança, polimorfismo e interfaces em partes mais avançadas da linguagem, como a definição de *threads* a partir da interface Runnable, elementos de interface gráfica por intermédio de classes internas anônimas e muito mais.

Verificamos também que tudo o que o Java pode oferecer possui grande consistência na sua aplicabilidade, afinal, essa linguagem surgiu para ser robusta e consistente.

O mundo de desenvolvimento Java é bastante amplo. Podemos criar desde aplicações simples, baseadas em interface gráfica, até aplicações para a web, contemplando frameworks importantes de mercado.

Para acesso a banco de dados, temos à disposição um framework excelente, o Hibernate, e sua implementação por meio de *annotations*, o JPA (*Java Persistence API*). Essas aplicações utilizam tudo o que estudamos.

Convido você a mergulhar mais ainda nesse maravilhoso mundo do desenvolvimento Java, que certamente trará muitos benefícios profissionais a todos.

Prof. Emilio

Respostas Comentadas dos Exercícios

Caro aluno, a seguir as respostas comentadas das listas de exercícios dos módulos desta disciplina.

Recomendo que você tente primeiro fazer sozinho as tarefas e use os *chats* e fóruns para tirar suas dúvidas. Depois você pode conferir as respostas, ok?

Exercício 1 – Módulo 1

Solução

```

1 package com.ead.modulo1.exercicios;
2 import javax.swing.JOptionPane;
3 public class TesteCalculadora {
4     public static void main(String[] args) {
5
6         try {
7             String valor1 = JOptionPane.showInputDialog("Informar o 1º valor!");
8             String valor2 = JOptionPane.showInputDialog("Informar o 2º valor!");
9             String op = JOptionPane.showInputDialog("Informar a operação (1 a 4)");
10            int operacao = Integer.parseInt(op);
11            JOptionPane.showMessageDialog(null, "Resultado: " +
12                                         efetuarOperacao(valor1, valor2, operacao));
13        } catch (Exception e) {
14            JOptionPane.showMessageDialog(null, e.getMessage());
15        }
16    }
17    private static int efetuarOperacao(String valor1, String valor2, int operacao){
18        int v1 = 0, v2 = 0, resultado = 0;
19        try {
20            v1 = Integer.parseInt(valor1);
21            v2 = Integer.parseInt(valor2);
22            if(operacao < 1 || operacao > 4){
23                throw new ArithmeticException("Operação inválida!");
24            }
25            switch(operacao){
26                case 1: resultado = v1 + v2; break;
27                case 2: resultado = v1 - v2; break;
28                case 3: resultado = v1 * v2; break;
29                case 4: resultado = v1 / v2; break;
30            }
31        } catch (NumberFormatException e) {
32            throw e;
33        } catch(ArithmeticException e) {
34            throw e;
35        }
36        return resultado;
37    }
38}

```

Comentários

- Neste exercício optamos por realizar uma leitura via teclado (linhas 7 a 9).
- Os valores foram armazenados em variáveis do tipo String para justificar a conversão e consequentemente o tratamento de erros.
- Nas linhas 20 e 21 estão os códigos que fazem a conversão, e são vulneráveis a erros porque os valores a serem convertidos são fornecidos pelo usuário, e podem estar em um formato incorreto.
- Nas linhas 32 e 34 as exceções são propagadas e capturadas no bloco try...catch, na linhas 6 a 15.

Exercício 2 - Módulo 1

Solução

```

1 package com.ead.modulo1.exercicios;
2
3 import javax.swing.JOptionPane;
4
5 public class CalculoMedias {
6     public static void main(String[] args) {
7         double[] notas = new double[4];
8         double soma = 0, media = 0;
9
10        try {
11            for (int i = 0; i <= 4; i++) {
12                notas[i] = Double.parseDouble(
13                    JOptionPane.showInputDialog("Nota " + (i+1)));
14                soma += notas[i];
15            }
16        } catch (ArrayIndexOutOfBoundsException e) {
17            JOptionPane.showMessageDialog(null, e.getMessage());
18        } finally {
19            media = soma / 4;
20        }
21        JOptionPane.showMessageDialog(null, "Média: " + media);
22    }
23 }
```

Comentários

- Neste programa declaramos e criamos um array de 4 posições na Linha 7.
- Declaramos também duas variáveis para acumular a soma dos valores das notas, na estrutura de repetição (variável soma), e uma variável para armazenar a média das notas (variável media).
- Observe na estrutura de repetição, na Linha 12, que nossa iteração se estende até o índice 4, que está além do último índice do array, que é 3.
- Quando o loop for chegar até este valor, uma exceção do tipo ArrayIndexOutOfBoundsException é lançada, mas até a posição 3, a soma dos elementos do array já foi acumulada na variável soma.
- No bloco finally, Linha 18 a 20, a média é calculada de forma correta, pois o quinto elemento do array, por não existir, entrou na exceção e, portanto, não foi considerado.

Exercício 1 – Módulo 2

Solução

```

1 package com.ead.modulo2.exercicios;
2
3 public class Aluno {
4     private int rm;
5     private String nome, curso;
6
7     //construtor
8     public Aluno(int rm, String nome, String curso) {
9         super();
10        this.rm = rm;
11        this.nome = nome;
12        this.curso = curso;
13    }
14
15    //getters e setters
16    public int getRm() {
17        return rm;
18    }
19    public void setRm(int rm) {
20        this.rm = rm;
21    }
22    public String getNome() {
23        return nome;
24    }
25    public void setNome(String nome) {
26        this.nome = nome;
27    }
28    public String getCurso() {
29        return curso;
30    }
31    public void setCurso(String curso) {
32        this.curso = curso;
33    }
34
35    public String toString(){
36        return "[" + rm + " - " + nome + " - " + curso + "]";
37    }
38 }
```

Comentários

- O método `toString()`, definido na Linha 35, é uma sobrescrita do método definido na classe `Object`. Este método foi sobrescrito nesta classe para que os objetos da classe `Aluno` tenham uma representação em forma de `String` adequada, especialmente para saídas na tela ou em componentes de interface gráfica.

```

1 package com.ead.modulo2.exercicios;
2
3 import java.util.ArrayList;
4 import java.util.List;
5
6 public class TesteAluno {
7     public static void main(String[] args) {
8         List<Aluno> alunos = new ArrayList<Aluno>();
9         alunos.add(new Aluno(120, "Bento", "ADministração"));
10        alunos.add(new Aluno(351, "Josias", "Direito"));
11        alunos.add(new Aluno(119, "Manoel", "Psicologia"));
12        alunos.add(new Aluno(268, "Bernardo", "TI"));
13        alunos.add(new Aluno(295, "Antonio", "Economia"));
14
15        for(Aluno aluno: alunos){
16            System.out.println(aluno);
17        }
18    }
19 }
```

Comentários

- Na linha 8 desta classe criamos uma lista de alunos, ou seja, um objeto ArrayList parametrizado para a classe Aluno, significando que somente objetos Aluno podem ser inseridos na lista.
- Nas linhas 9 a 13 inserimos objetos na lista, levando em conta o construtor sobreescrito desta classe.
- A Linha 15 define um loop for aprimorado, e na Linha 16 o programa exibe o conteúdo de cada elemento da lista na tela. Sabemos que cada elemento é um objeto da classe Aluno, e nesta ocasião, o método `toString()` é chamado pela JVM. A saída deste programa é:

```

<terminated> TesteAluno [Java Application] C:\Program Fil
[120 - Bento - ADministração]
[351 - Josias - Direito]
[119 - Manoel - Psicologia]
[268 - Bernardo - TI]
[295 - Antonio - Economia]
```

Exercício 2 - Módulo 2

Solução

```

1 package com.ead.modulo2.exercicios;
2
3 public class Aluno implements Comparable<Aluno>{
4     private int rm;
5     private String nome, curso;
6
7     //construtor
8     public Aluno(int rm, String nome, String curso) {
9         super();
10        this.rm = rm;
11        this.nome = nome;
12        this.curso = curso;
13    }
14
15    //getters e setters
16    public int getRm() {}
17    public void setRm(int rm) {}
18    public String getNome() {}
19    public void setNome(String nome) {}
20    public String getCurso() {}
21    public void setCurso(String curso) {}

22    public String toString(){
23        return "[" + rm + " - " + nome + " - " + curso + "]";
24    }
25
26    @Override
27    public int compareTo(Aluno arg0) {
28        return this.nome.compareTo(arg0.nome);
29    }
30}
31
32
33
34
35
36
37
38
39
40
41
42
43 }
```

```

1 package com.ead.modulo2.exercicios;
2
3 import java.util.ArrayList;
4 import java.util.Collections;
5 import java.util.List;
6
7 public class TesteAluno {
8     public static void main(String[] args) {
9         List<Aluno> alunos = new ArrayList<Aluno>();
10        alunos.add(new Aluno(120, "Bento", "Administração"));
11        alunos.add(new Aluno(351, "Josias", "Direito"));
12        alunos.add(new Aluno(119, "Manoel", "Psicologia"));
13        alunos.add(new Aluno(268, "Bernardo", "TI"));
14        alunos.add(new Aluno(295, "Antonio", "Economia"));

15        Collections.sort(alunos);

16        for(Aluno aluno: alunos){
17            System.out.println(aluno);
18        }
19    }
20}
```

Comentários

- A implementação da interface Comparable, conforme codificado na Linha 3 da nova versão da classe Aluno, serve para solicitar a implementação do método compareTo(), definido na Linha 40. Este método é usado pelos mecanismos de ordenação. Nossa responsabilidade é atribuir uma funcionalidade a ele, no sentido de definir o que ocorre entre dois elementos. Em outras palavras, nós devemos levar em conta uma comparação entre o objeto passado como parâmetro e o objeto que executa o método (this). No nosso exemplo, decidimos que a ordenação entre objetos da classe Pessoa ocorre através do nome; então é o nome que mencionamos como critério de ordenação na definição do método. Felizmente, o nome é uma String, e a classe String também define o método compareTo(), por implementar também a interface Comparable.
- Com esta implementação, o método sort() da classe Collections tem condições de classificar os elementos de uma lista de objetos Aluno.

Exercício 3 – Módulo 2

Solução

```

1 package com.ead.modulo2.exercicios;
2
3 import java.util.ArrayList;
4 import java.util.Collections;
5 import java.util.HashMap;
6 import java.util.List;
7 import java.util.Map;
8
9 public class TesteAluno {
10    public static void main(String[] args) {
11        List<Aluno> alunos = new ArrayList<Aluno>();
12        alunos.add(new Aluno(120, "Bento", "ADministração"));
13        alunos.add(new Aluno(351, "Josias", "Direito"));
14        alunos.add(new Aluno(119, "Manoel", "Psicologia"));
15        alunos.add(new Aluno(268, "Bernardo", "TI"));
16        alunos.add(new Aluno(295, "Antonio", "Economia"));
17
18        Collections.sort(alunos);
19        Map<Integer, String> mapAlunos = gerarMap(alunos);
20
21        for(Map.Entry<Integer, String> aluno: mapAlunos.entrySet()){
22            System.out.println(aluno.getKey() + " - " + aluno.getValue());
23        }
24    }
25
26    private static Map<Integer, String> gerarMap(List<Aluno> alunos){
27        Map<Integer, String> mapa = new HashMap<Integer, String>();
28        for(Aluno aluno: alunos){
29            mapa.put(aluno.getRm(), aluno.getNome());
30        }
31        return mapa;
32    }
33}

```

Comentários

- Neste Exercício nós usamos uma estrutura de repetição percorrendo os elementos da lista de alunos (Linha 21 da classe TesteAluno). Como cada elemento desta lista é um objeto da classe Aluno,

tomamos os valores do rm (método getRm()) e o valor do nome (método getNome()) e com eles, adicionamos um novo elemento ao Map (método gerarMap(), linhas 26 a 32).

Exercício 1 – Módulo 3

Solução

```

1 package com.ead.modulo3.exercicios;
2
3 import java.util.Random;
4
5 public class TesteRandom {
6     public static void main(String[] args) {
7         Random rnd = new Random();
8         double d = rnd.nextDouble();
9
10        System.out.println("Nr. gerado: " + d);
11        byte[] b = new byte[10];
12        rnd.nextBytes(b);
13
14        for (int i = 0; i < b.length; i++) {
15            System.out.println(b[i]);
16        }
17    }
18 }
```

Comentários

- A classe Random, como já podíamos imaginar, define métodos para manipular números aleatórios.
- Na Linha 7 criamos um objeto desta classe.
- Na Linha 8 executamos o método nextDouble() a partir deste objeto. Este método retorna um valor aleatório entre 0 e 1.
- Na Linha 11 criamos um array de bytes.
- O método nextBytes() recebe como parâmetro o array de bytes, e define um valor aleatório para cada elemento do array.
- Ao executar este programa, temos o seguinte resultado:

```
<terminated> TesteRandom [Java Application] C:\F
Nr. gerado: 0.5330781887203199
-93
-54
-118
-36
90
-27
3
50
-36
72
```

- Se você tentar executar este programa, certamente obterá valores diferentes, visto tratar-se de números aleatórios

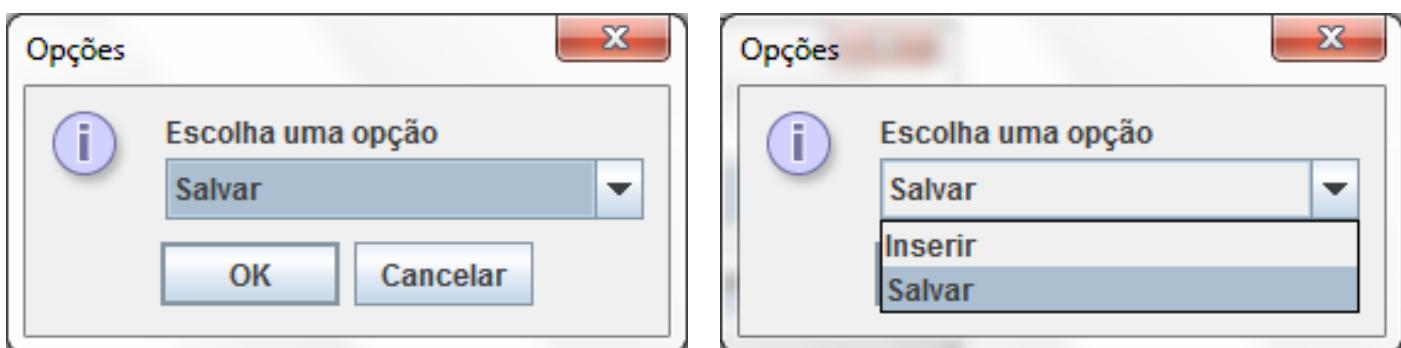
```

1 package com.ead.modulo3.exercicios;
2
3 import javax.swing.JOptionPane;
4
5 public class TesteJOptionPane {
6     public static void main(String[] args) {
7         String opcao = (String) JOptionPane.showInputDialog(null,
8                         "Escolha uma opção",
9                         "Opções",
10                        JOptionPane.INFORMATION_MESSAGE,
11                         null,
12                         new String[]{"Inserir", "Salvar"},
13                         "Salvar");
14
15         JOptionPane.showMessageDialog(null, "Você escolheu: " + opcao);
16     }
17 }

```

Comentários

- O método `showInputDialog()` da classe `JOptionPane` possui mais de uma sobrecarga. A versão contendo 7 parâmetros é a mais completa e, de acordo com a API, temos:
 - o 1º parâmetro: A origem da janela. O valor `null` indica que ela está sendo executada a partir de uma aplicação baseada em Console.
 - o 2º parâmetro: O texto a ser exibido.
 - o 3º parâmetro: O título da janela
 - o 4º parâmetro: Uma constante que representa o ícone da janela. Existem diversas constantes disponíveis na classe `JOptionPane`.
 - o 5º parâmetro: Um objeto da classe `Icon`, representando o ícone ao lado do título da janela. Na sua ausência, informamos `null`, significando que o ícone padrão será usado.
 - o 6º parâmetro: Um array de `Object`. No nosso exemplo, o array é de `String`, mas podemos ficar tranquilos, pois `String` é subclasse de `Object`. Este array significa as opções que teremos em nossa janela. Isso mesmo! Esta versão da janela não mostra uma caixa de textos, e sim um componente `ComboBox`, onde é possível selecionar uma opção ao invés de digitá-la.
 - o 7º parâmetro: Esta é a opção inicial, dentre as opções disponíveis no array correspondente ao 6º parâmetro.
- A execução deste programa produz a seguinte janela:



Exercício 1 – Módulo 4

Solução

A solução está disponível na apostila.

Comentários

Os comentários também estão disponíveis na apostila. É importante implementar e executar estes códigos antes de prosseguir com os exercícios.

Exercício 2 – Módulo 4

Solução

```
1 package com.ead.modulo4.exercicios;
2
3 public class Produtos {
4
5     private double preco;
6
7     public Produtos(double preco){
8         this.preco = preco;
9     }
10
11    public double verificaPreco(){
12        //preço original
13        return preco;
14    }
15
16    public void reajustaPreco(double taxa){
17        preco = preco * (1 + taxa / 100);
18    }
19
20    public double efetuarCompra(){
21        //preço após reajuste
22        return preco;
23    }
24 }
```

```

1 package com.ead.modulo4.exercicios;
2
3 public class ProcessaProdutos implements Runnable {
4
5     Produtos produto = new Produtos(120);
6
7     private void realizarCompra() throws Exception{
8         String nome = Thread.currentThread().getName();
9         System.out.println("Compra para: " + nome);
10        System.out.println("Valor do produto: " + produto.verificaPreco());
11        Thread.sleep(200);
12        System.out.println("Reajuste de 10% para " + nome);
13        produto.reajustaPreco(10);
14        Thread.sleep(200);
15        System.out.println("Novo valor para " + nome + ": " +
16                           produto.efetuarCompra());
17        System.out.println("-----");
18    }
19
20    @Override
21    public void run() {
22        try {
23            realizarCompra();
24        } catch (Exception e) {
25            e.printStackTrace();
26        }
27    }
28}

```

```

1 package com.ead.modulo4.exercicios;
2
3 public class TesteProdutos {
4     public static void main(String[] args) {
5         ProcessaProdutos pp = new ProcessaProdutos();
6         Thread t1 = new Thread(pp, "Joao");
7         Thread t2 = new Thread(pp, "Roger");
8         Thread t3 = new Thread(pp, "Maria");
9
10        t1.start();
11        t2.start();
12        t3.start();
13    }
14}

```

Comentários

- A classe Produtos representa um objeto-alvo para nossos threads.
- A classe ProcessaProdutos é a classe responsável por definir os threads. Observe que, para efeito de testes, incluímos um temporizador nas linhas 11 e 14 desta classe. Isso foi feito para demonstrar o que ocorre quando dois ou mais threads acessam o mesmo objeto. A execução desejada é a definida no método realizarCompra(), ou seja:
 - Verificar o preço (método verificarPreco() – Linha 10)
 - Aplicar o reajuste (método reajustaPreco() – Linha 13)
 - Efetuar a venda (método efetuarCompra() – Linha 16)

- Na classe TesteProdutos foram criados três threads apontando para o mesmo objeto. Quando executamos este programa, obtemos o seguinte resultado:

```
<terminated> TesteProdutos [Java Application] C:\Program Files (x86)\Ja
Compra para: Joao
Compra para: Maria
Compra para: Roger
Valor do produto: 120.0
Valor do produto: 120.0
Valor do produto: 120.0
Reajuste de 10% para Roger
Reajuste de 10% para Maria
Reajuste de 10% para Joao
Novo valor para Maria: 159.7200000000003
Novo valor para Joao: 159.7200000000003
-----
Novo valor para Roger: 159.7200000000003
-----
```

- Os resultados parecem sem sentido, não? Isso corre porque os três threads são executados simultaneamente, e a ordem é caótica. Para que possamos garantir a ordem a execução completa e exclusiva para cada thread, é necessário sincronizar o método realizarCompra() da classe ProcessaProdutos. Este é justamente o objetivo do Exercício 3.

Exercício 3 – Módulo 4

Solução

```
1 package com.ead.modulo4.exercicios;
2
3 public class ProcessaProdutos implements Runnable {
4
5     Produtos produto = new Produtos(120);
6
7     private synchronized void realizarCompra() throws Exception{
8         String nome = Thread.currentThread().getName();
9         System.out.println("Compra para: " + nome);
10        System.out.println("Valor do produto: " + produto.verificaPreco());
11        Thread.sleep(200);
12        System.out.println("Reajuste de 10% para " + nome);
13        produto.reajustaPreco(10);
14        Thread.sleep(200);
15        System.out.println("Novo valor para " + nome + ": " +
16                           produto.efetuarCompra());
17        System.out.println("-----");
18    }
19
20    @Override
21    public void run() {
22        try {
23            realizarCompra();
24        } catch (Exception e) {
25            e.printStackTrace();
26        }
27    }
28 }
```

Comentários

- Neste exercício tomamos o método realizarCompra() e incluímos a palavra reservada synchronized. Desta forma sincronizamos todas as operações contidas neste método, ou seja, quando um thread estiver executando-o, o outro deverá aguardar pelo seu término. O resultado da execução do mesmo programa produz o seguinte resultado:

```
<terminated> TesteProdutos [Java Application] C:\Program Files (x86)\J
Compra para: Joao
Valor do produto: 120.0
Reajuste de 10% para Joao
Novo valor para Joao: 132.0
-----
Compra para: Roger
Valor do produto: 132.0
Reajuste de 10% para Roger
Novo valor para Roger: 145.2000000000002
-----
Compra para: Maria
Valor do produto: 145.2000000000002
Reajuste de 10% para Maria
Novo valor para Maria: 159.7200000000003
```

Exercício 1 – Módulo 5

Solução

```
1 package com.ead.modulo5.exercicios;
2
3 import javax.swing.JButton;
4 import javax.swing.JComboBox;
5 import javax.swing.JFrame;
6 import javax.swing.JLabel;
7 import javax.swing.JList;
8 import javax.swing.JPanel;
9 import javax.swing.JScrollPane;
10 import javax.swing.JTextField;
11
12 @SuppressWarnings("serial")
13 public class AlunosGUI extends JFrame{
14     private JPanel painel;
15     private JTextField txtRM;
16     private JTextField txtNome;
17     private JTextField txtCurso;
18
19     private JLabel lblRM;
20     private JLabel lblNome;
21     private JLabel lblCurso;
22     private JLabel lblPeriodo;
23
24     private JList<String> lstAlunos;
25     private JScrollPane scrool;
26     private JComboBox<String> cmbPeriodo;
27     private JButton btnEnviar;
28     private JButton btnFechar;
29     private JButton btnListar;
30 }
```

```
31@ public AlunosGUI(){
32     super("Cadastro de Alunos");
33     painel = new JPanel();
34     this.setContentPane(painel);
35     painel.setLayout(null);
36     this.setBounds(100, 100, 600, 250);
37
38     lblRM= new JLabel("RM:");
39     txtRM = new JTextField();
40     lblNome = new JLabel("Nome:");
41     txtNome = new JTextField();
42     lblCurso= new JLabel("Curso:");
43     txtCurso = new JTextField();
44     lblPeriodo = new JLabel("Período:");
45
46     cmbPeriodo= new JComboBox<String>();
47     cmbPeriodo.addItem("MANHÃ");
48     cmbPeriodo.addItem("TARDE");
49     cmbPeriodo.addItem("NOITE");
50
51     scrool = new JScrollPane();
52     lstAlunos = new JList<String>();
53
54     btnEnviar = new JButton("Enviar");
55     btnFechar = new JButton("Fechar");
56     btnListar = new JButton("Listar Alunos");
57
58     lblRM.setBounds(10, 10, 70, 20);
59     txtRM.setBounds(120, 10, 100, 20);
60
61     lblNome.setBounds(10, 30, 100, 20);
62     txtNome.setBounds(120, 30, 200, 20);
63
64     lblCurso.setBounds(10, 50, 100, 20);
65     txtCurso.setBounds(120, 50, 100, 20);
66
67     lblPeriodo.setBounds(10, 70, 100, 20);
68     cmbPeriodo.setBounds(120, 70, 200, 20);
69
70     scrool.setBounds(350, 10, 200, 150);
71     lstAlunos.setBounds(350, 10, 200, 150);
72     scrool.setViewportView(lstAlunos);
73     btnEnviar.setBounds(120, 110, 150, 25);
74     btnListar.setBounds(120, 140, 150, 25);
75     btnFechar.setBounds(120, 170, 150, 25);
76
77     painel.add(lblRM);
78     painel.add(txtRM);
79     painel.add(lblNome);
80     painel.add(txtNome);
81     painel.add(lblCurso);
82     painel.add(txtCurso);
83     painel.add(lblPeriodo);
84     painel.add(cmbPeriodo);
85     painel.add(btnEnviar);
86     painel.add(btnFechar);
87     painel.add(btnListar);
88     painel.add(scrool);
89
90 }
91 }
```

```

1 package com.ead.modulo5.exercicios;
2
3 public class TestePrograma {
4     public static void main(String[] args) {
5         AlunosGUI p = new AlunosGUI();
6         p.setVisible(true);
7     }
8 }
```

Comentários

- Nesta classe declaramos os componentes da interface gráfica (Linhas 14 a 29).
- Na sequencia definimos o construtor da classe onde:
 - Instanciamos cada um dos componentes
 - Definimos suas dimensões (execução do método setBounds())
 - Adicionamos os componentes no painel.

Exercício 2 – Módulo 5

Solução

```

1 package com.ead.modulo5.exercicios;
2
3 import java.util.ArrayList;
4 import java.util.List;
5
6 public class Aluno {
7     private int rm;
8     private String nome, curso, periodo;
9
10    public static List<Aluno> alunos = new ArrayList<Aluno>();
11
12    public int getRm() {}
13    public void setRm(int rm) {}
14    public String getNome() {}
15    public void setNome(String nome) {}
16    public String getCurso() {}
17    public void setCurso(String curso) {}
18    public String getPeriodo() {}
19    public void setPeriodo(String periodo) {}
20
21    public String toString(){
22        return "[" + rm + " - " + nome + " - " + curso + " - " + periodo;
23    }
24}
```

Comentários

- Neste exercício criamos uma classe chamada Aluno com os atributos selecionados.
- Sobrescrevemos o método `toString()` (Linha 37)com o objetivo de definirmos uma representação String para os objetos desta classe.
- A lista de alunos (`List<Aluno>`), definida na Linha 10, servirá para receber objetos da classe Aluno.

Exercício 3 – Módulo 5

Solução

```

61      btnEnviar = new JButton("Enviar");
62      btnEnviar.addActionListener(new ActionListener() {
63
64          @Override
65          public void actionPerformed(ActionEvent arg0) {
66              Aluno aluno = new Aluno();
67              aluno.setRm(Integer.parseInt(txtRM.getText()));
68              aluno.setNome(txtNome.getText());
69              aluno.setCurso(txtCurso.getText());
70              aluno.setPeriodo((String)cmbPeriodo.getSelectedItem());
71
72              Aluno.alunos.add(aluno);
73
74              JOptionPane.showMessageDialog(null, "Aluno incluído na lista");
75          }
76      });
77      btnFechar = new JButton("Fechar");
78      btnFechar.addActionListener(new ActionListener() {
79
80          @Override
81          public void actionPerformed(ActionEvent e) {
82              System.exit(0);
83          }
84      });
85      btnListar = new JButton("Listar Alunos");
86      btnListar.addActionListener(new ActionListener() {
87
88          @Override
89          public void actionPerformed(ActionEvent e) {
90              DefaultListModel<String> model = new DefaultListModel<String>();
91              for(Aluno aluno: Aluno.alunos){
92                  model.addElement(aluno.toString());
93              }
94              lstAlunos.setModel(model);
95          }
96      });
97  });
98 });

```

Comentários

- Nas Linhas 62, 79 e 88 foram definidos os métodos `addActionListener()` para os três botões da interface gráfica, seguindo o critério de classe interna anônima. Houve uma implementação da interface `ActionListener` para cada um dos botões.
- Nas Linhas 65 a 74 foram definidos os códigos para criar um objeto da classe `Aluno` e inseri-lo na lista.
- Na linha 92 foi definido um objeto da classe `DefaultListModel<>`. Para maniopular um elemento `JList`, o procedimento é diferente do usado para inserir elementos em um componente `JComboBox`. É necessário criar um modelo, e em seguida, inserir este modelo no componente `JList`. Observe que nas Linhas 93 a 95 foram incluídos objetos no modelo, e após a execução da estrutura de repetição, o modelo foi adicionado ao `JList` através do método `setModel()` (Linha 96).

Exercício 1 – Módulo 6

Solução

```

1 package com.ead.modulo6.exercicios;
2
3 import java.awt.event.ActionEvent;
4 import java.awt.event.ActionListener;
5 import java.io.FileReader;
6 import java.io.FileWriter;
7
8 import javax.swing.JButton;
9 import javax.swing.JFileChooser;
10 import javax.swing.JFrame;
11 import javax.swing.JOptionPane;
12 import javax.swing.JPanel;
13 import javax.swing.JScrollPane;
14 import javax.swing.JTextArea;
15 import javax.swing.border.EmptyBorder;
16
17 @SuppressWarnings("serial")
18 public class EditorTextos extends JFrame{
19     private JPanel contentPane;
20
21     public EditorTextos() {
22         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
23         setBounds(100, 100, 673, 397);
24         contentPane = new JPanel();
25         contentPane.setBorder(new EmptyBorder(5, 5, 5, 5));
26         setContentPane(contentPane);
27         contentPane.setLayout(null);
28
29         JScrollPane scrollPane = new JScrollPane();
30         scrollPane.setBounds(10, 44, 645, 315);
31         contentPane.add(scrollPane);
32
33         final JTextArea textArea = new JTextArea();
34         scrollPane.setViewportView(textArea);
35
36         JButton btnAbrirArquivo = new JButton("Abrir Arquivo");
37         btnAbrirArquivo.addActionListener(new ActionListener() {
38             public void actionPerformed(ActionEvent arg0) {
39
40                 JFileChooser chooser = new JFileChooser();
41                 try {
42
43                     if(chooser.showOpenDialog(EditorTextos.this) ==
44                         JFileChooser.APPROVE_OPTION){
45                         FileReader arquivo =
46                             new FileReader(chooser.getSelectedFile());
47                         String texto = "";
48                         while(true){
49                             int c = arquivo.read();
50                             if(c == -1){ //EOF
51                                 break;
52                             }
53                             texto += (char)c;
54                         }
55                     }
56                 }
57             }
58         });
59         scrollPane.add(btnAbrirArquivo);
60     }
61
62     public static void main(String[] args) {
63         EditorTextos frame = new EditorTextos();
64         frame.setVisible(true);
65     }
66 }

```

```

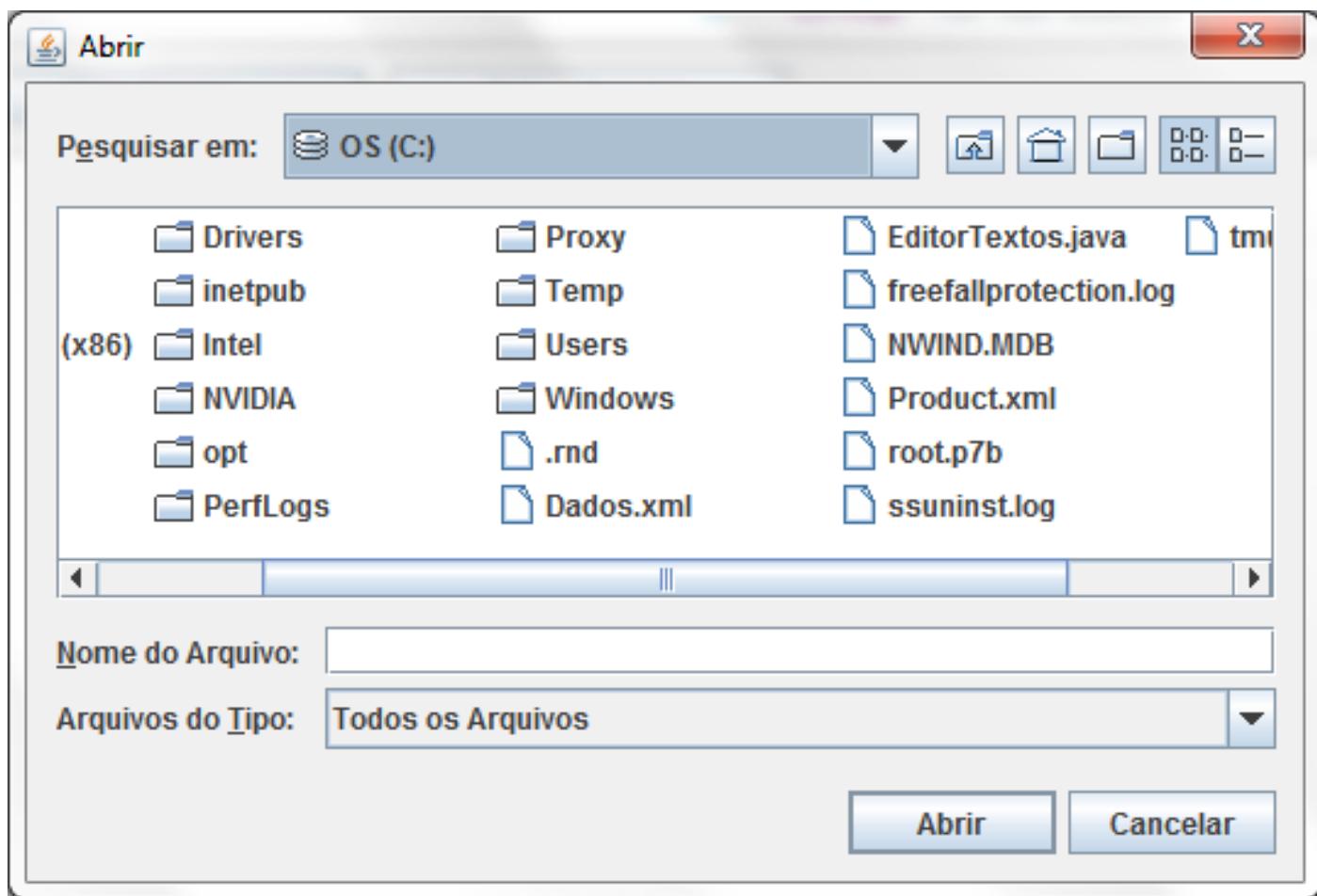
55         textArea.setText(texto);
56     }
57     //throw new Exception("Erro!");
58 } catch (Exception e) {
59     JOptionPane.showMessageDialog(EditorTextos.this, e.getMessage());
60 }
61 });
62 btnAbrirArquivo.setBounds(31, 10, 173, 23);
63 contentPane.add(btnAbrirArquivo);
64
65 JButton btnSalvarArquivo = new JButton("Salvar Arquivo");
66 btnSalvarArquivo.addActionListener(new ActionListener() {
67     public void actionPerformed(ActionEvent arg0) {
68         JFileChooser chooser = new JFileChooser();
69         try {
70
71             if(chooser.showSaveDialog(EditorTextos.this) ==
72                 JFileChooser.APPROVE_OPTION){
73                 FileWriter arquivo =
74                     new FileWriter(chooser.getSelectedFile());
75                 String texto = textArea.getText();
76                 arquivo.write(texto);
77                 arquivo.close();
78                 JOptionPane.showMessageDialog(EditorTextos.this,
79                     "Arquivo criado com sucesso");
80             }
81
82         } catch (Exception e) {
83             JOptionPane.showMessageDialog(EditorTextos.this, e.getMessage());
84         }
85     }
86 }
87 });
88 btnSalvarArquivo.setBounds(214, 10, 163, 23);
89 contentPane.add(btnSalvarArquivo);
90 }
91 }

1 package com.ead.modulo6.exercicios;
2
3 public class TesteEditor {
4     public static void main(String[] args) {
5         EditorTextos editor = new EditorTextos();
6         editor.setVisible(true);
7     }
8 }

```

Comentários

- O evento de ação do botão “Abrir Arquivo” é usado para abrir um arquivo e exibir seu conteúdo na caixa de textos. A classe JFileChooser permite definirmos objetos que apresentam caixas de diálogo para manipulação de arquivos. O método showOpenDialog() apresentado na Linha 43 produz o diálogo abaixo:



- Se o usuário selecionar um arquivo e clicar no botão “Abrir”, o método retorna o valor da constante JFileChooser.APPROVE_OPTION, que é a condição para darmos continuidade à abertura do arquivo. Na linha 46 estamos executando o método getSelectedFile(), que retorna justamente o arquivo que queremos abrir.
- Na linha 55 já temos o conteúdo do arquivo armazenado na variável texto, e o incluímos na caixa de textos.
- Um procedimento semelhante é realizado na Linha 67, onde o método showSaveDialog() é executado. A diferença é que o usuário deve fornecer o nome para o arquivo. Se o nome do arquivo fornecido for existente, o próprio objeto JFileChooser realiza a confirmação de sobreescrita do arquivo.

Exercício 2 – Módulo 6

Solução

```

1 package com.ead.modulo6.exercicios;
2
3 import java.io.FileReader;
4 import java.io.IOException;
5
6 public class LeituraParcial {
7     public static void main(String[] args) {
8         try{
9             FileReader reader = new FileReader("C:/EditorTextos.java");
10            String texto = "";
11            int contador = 0;
12            while(true){
13                int ch = reader.read();
14                if(ch == -1) { //fim do arquivo
15                    break;
16                }
17                if(++contador > 20){
18                    texto += "...";
19                    break;
20                }
21                texto += (char)ch;
22            }
23            System.out.println(texto);
24            reader.close();
25        } catch(IOException e){
26            System.out.println(e.getMessage());
27        }
28    }
29 }
```

Comentários

- Neste exercício realizamos a leitura d o arquivo um caractere de cada vez. Na linha 17 verificamos se o n mero de caracteres atingiu o valor 20. Se atingiu, concatenamos a String “...” ao final da vari vel que armazena os 20 primeiros caracteres do arquivo ´a vari vel texto.
- Na linha 20, o resultado esperado  exibido na tela.

Exercício 1 – Módulo 7

Solução

```

1 package com.ead.modulo7.exercicios;
2 import java.sql.*;
3 import java.util.ArrayList;
4 import java.util.List;
5
6 public class AlunosDAO {
7     Connection cn;
8     PreparedStatement stmt;
9     ResultSet rs;
10
11     String url="jdbc:mysql://localhost:3306/ead_java";
12
13     private void abrirConexao() throws Exception {
14         try {
15             Class.forName("com.mysql.jdbc.Driver");
16             cn = DriverManager.getConnection(url,"root","root");
17
18         } catch (Exception e) {
19             throw e;
20         }
21     }
22
23     private void fecharConexao() throws Exception {
24         if(!cn.isClosed()){
25             cn.close();
26         }
27     }
28
29     public List<Aluno> listarAlunos() throws Exception{
30         List<Aluno> alunos = new ArrayList<Aluno>();
31         try {
32             abrirConexao();
33             stmt = cn.prepareStatement("SELECT * FROM ALUNOS");
34             rs = stmt.executeQuery();
35
36             while(rs.next()){
37                 Aluno a = new Aluno();
38                 a.setRm(rs.getInt("RM"));
39                 a.setNome(rs.getString("NOME"));
40                 a.setCurso(rs.getString("CURSO"));
41                 a.setPeriodo(rs.getString("PERIODO"));
42                 alunos.add(a);
43             }
44
45         } catch (Exception e) {
46             throw e;
47         } finally {
48             fecharConexao();
49         }
50
51         return alunos;
52     }
53
54 }
```

Comentários

- Para este exercício utilizamos o banco de dados que chamamos de ead_java, conforme String de conexão definida na Linha 11. A tabela do banco de dados deste Exercício é a mostrada abaixo:

alunos	
RM	INT(11)
NOME	VARCHAR(45)
CURSO	VARCHAR(45)
PERIODO	VARCHAR(45)
Indexes	

- O connector do MySQL foi inserido no projeto deste Exercício.
- O método abrirConexao() (Linhas 13 a 21) define o driver de acesso ao MySQL, e executa o método getConnection() que, de acordo com a String de conexão, estabelece a conexão com nosso banco de dados.
- O método fecharConexao() (Linhas 23 a 27) verifica se a conexão está fechada e, se não tiver, a fecha.
- O método incluir() (Linhas 29 a 45) recebe um objeto da classe Aluno e, através de uma instrução SQL e dos métodos setters aplicados ao objeto PreparedStatement (stmt), permitem a inclusão de um novo aluno no banco de dados.
- O método listarAlunos() (Linhas 47 a 71) realiza uma consulta no banco de dados e, para cada registro localizado, cria um objeto da classe Aluno e atribui cada registro a cada atributo da classe (linhas 55 a 59).
- Na linha 59 o objeto obtido de cada registro é adicionado na lista, que será retornada por este método.

Exercício 2 – Módulo 7

Solução

```

61      btnEnviar = new JButton("Enviar");
62      btnEnviar.addActionListener(new ActionListener() {
63
64          @Override
65          public void actionPerformed(ActionEvent arg0) {
66
67              try {
68                  Aluno aluno = new Aluno();
69                  aluno.setRm(Integer.parseInt(txtRM.getText()));
70                  aluno.setNome(txtNome.getText());
71                  aluno.setCurso(txtCurso.getText());
72                  aluno.setPeriodo((String)cmbPeriodo.getSelectedItem());
73
74                  AlunosDAO dao = new AlunosDAO();
75                  dao.incluir(aluno);
76
77
78                  JOptionPane.showMessageDialog(null, "Aluno incluído no Banco");
79              } catch (Exception e) {
80                  JOptionPane.showMessageDialog(null, e.getMessage());
81              }
82          }
83      });
84      btnListar = new JButton("Listar Alunos");
85      btnListar.addActionListener(new ActionListener() {
86
87
88
89
90
91
92
93
94
95

```

Comentários

- A alteração neste exercício ocorreu na classe AlunosGUI. Os métodos dos botões “Enviar” e “Listar Alunos” foram adaptados para refletir o acesso ao banco de dados.
- Nas linhas 68 a 72 um objeto da classe Aluno é criado e populado com as informações da interface gráfica.
- Na linha 74, um objeto da classe AlunosDAO é criado, para que seus métodos de acesso a dados possam ser usados.
- Na linha 75, executamos o método incluir() informando como parâmetro o objeto aluno, criado para esta finalidade.
- Para listar os alunos no componente JList, a alteração realizada ocorreu no evento de ação do botão “Listar Alunos”.
- Na Linha 102 executamos o método listarAlunos() a partir de uma instância da classe AlunosDAO(). O retorno deste método foi atribuído a uma lista de alunos, e esta lista foi usada para preencher o componente JList (Linhas 104 a 107).

Referências Bibliográficas

ARNOLD, K.; GOSLING, J.; HOLMES, D. **A linguagem de programação Java**. Porto Alegre: Bookman, 2007.

BARNES, D. J.; KÖLLING, M. **Programação orientada a objetos com Java**. São Paulo: Pearson Prentice-Hall, 2004.

DEITEL, H.; DEITEL, P. **Java: como programar**. São Paulo: Prentice-Hall, 2010.

FLANAGAN, D. **Java: o guia essencial**. Porto Alegre: Bookman, 2006.

JORGE, Eduardo; VINICIUS, Marcus; MARTINS, Silvio. **Aplicações multimídia em Java**. 2012. Disponível em: <http://www.devmedia.com.br/artigo-java-magazine-57-aplicacoes-multimidia-em-java/9423>, acesso em: 27.jun.2014.

ORACLE. **Java™ Platform, Standard Edition 7 API specification**. 2014. Disponível em: <http://docs.oracle.com/javase/7/docs/api/>, acesso em: 6.jun.2014.

SIERRA, K.; BATES, B. **Sun certified programmer for Java 6 study guide**. New York: McGraw-Hill, 2008.

TUTORIALSPPOINT. **Java – applets basics**. 2014. Disponível em: http://www.tutorialspoint.com/java/java_applet_basics.htm, acesso em: 5.mai.2014.