

Unidade II

3 EVENTOS

Os eventos são as possíveis ações de interação do usuário com o sistema o qual ele está utilizando. Eles podem acontecer:

- pelo acionamento de um botão;
- pela passagem do mouse sobre algum componente;
- pelo acionamento das ações de janela como minimização, maximização etc.;

A linguagem Java tem um modo particular de controlar os eventos. Para isso, ela conta com as interfaces chamadas Listeners, criadas para serem implementadas pelas classes de componentes cujos eventos precisam ser controlados.



Lembrete

Interface é um elemento da linguagem Java que se assemelha a uma classe totalmente abstrata, ou seja, com métodos abstratos, e atributos estáticos e finais.

São alguns exemplos de Listeners:

`WindowListener` (eventos de **janela**)

Esta interface de eventos possui os seguintes métodos abstratos:

- **`windowOpened(WindowEvent e)`** : método acionado na primeira vez que a janela é aberta.
- **`windowClosing(WindowEvent e)`** : método acionado quando o ícone que fecha a janela é selecionado (usado para encerrar o programa).
- **`windowClosed(WindowEvent e)`** : método acionado na efetivação do fechamento de uma janela.
- **`windowActivated(WindowEvent e)`** : método acionado quando uma janela é ativada, ou quando é dado o foco a ela (clikando sobre ela, por exemplo).

- **windowDeactivated(WindowEvent e)**: método acionado quando uma janela é desativada, ou quando é retirado o foco dela (clitando sobre outra janela, por exemplo).
- **windowIconified(WindowEvent e)**: método acionado quando uma janela é minimizada.
- **windowDeiconified(WindowEvent e)**: método acionado quando uma janela é restaurada a partir de um ícone.

MouseListener (eventos específicos de mouse)

Esta interface de eventos possui os seguintes métodos abstratos:

- **mouseClicked(MouseEvent e)**: método acionado quando o botão do mouse é clicado (e solto) sobre um componente.
- **mousePressed(MouseEvent e)**: método acionado quando o botão do mouse é clicado sobre um componente.
- **mouseReleased(MouseEvent e)**: método acionado quando o botão do mouse é solto sobre um componente.
- **mouseEntered(MouseEvent e)**: método acionado quando o mouse "entra" na área de um componente.
- **mouseExited(MouseEvent e)**: método acionado quando o mouse deixa a área de um componente.

MouseMotionListener (eventos de movimentação do Mouse)

Esta interface de eventos possui os seguintes métodos abstratos:

- **mouseMoved(MouseEvent e)**: método acionado quando o ponteiro do mouse se movimenta sobre um componente.
- **mouseDragged(MouseEvent e)**: método acionado quando o mouse se movimenta sobre um componente enquanto o seu botão está pressionado.

KeyListener (eventos de teclado)

Esta interface de eventos possui os seguintes métodos abstratos:

- **keyTyped(KeyEvent e)**: método acionado quando uma tecla do teclado é pressionada e solta (com exceção das teclas SHIFT, ALT, CTRL, direcionais, Insert, Delete, teclas de função etc.).

- **keyPressed(KeyEvent e)** : método acionado quando uma tecla do teclado é pressionada.
- **keyReleased(KeyEvent e)** : método acionado quando uma tecla do teclado é solta.

ActionListener (eventos de ação)

Esta interface de eventos possui o seguinte método abstrato:

- **actionPerformed(ActionEvent e)** : método acionado quando uma ação ocorre em algum componente da janela (por exemplo, um botão da janela é pressionado).



Saiba mais

Para aprender mais sobre o funcionamento e o tratamento de eventos, leia o capítulo 8 da obra a seguir:

HORSTMANN, C. S.; CORNELL, G. *Core Java*. Volume I: fundamentos. 8. ed. São Paulo: Pearson Prentice Hall, 2010.

3.1 Utilizando-se dos eventos (controlando-os)

A utilização dos eventos pode ser programada de duas formas:

- fazendo com que a classe implemente as interfaces relacionadas a cada um deles (lembrando que uma classe pode fazê-lo com mais de uma ao mesmo tempo);
- fazendo com que o componente implemente a interface ao adicionar o Listener no componente a ser controlado.

Caso optemos por implementar as interfaces dos Listeners à classe, devemos lembrar de usar todos os métodos que a interface possui.



Lembrete

Implementar uma interface é utilizar, na declaração da classe, a palavra reservada **implements** com o nome das interfaces. Quando implementamos a interface em uma classe, todos os métodos existentes na interface devem ser criados (implementados) na própria classe.

Exemplo de aplicação

Vamos considerar a seguinte interface:

```
public interface interqualquer {  
    public void metodoH(int par01, double par02);  
    public double metodoK(double par02);  
}
```

Ela contém 2 métodos (que são abstratos).

Para fazer com que a classe (como a `ClasseA` a seguir) implemente a interface, ela deverá conter a implementação dos métodos da interface. A implementação dos seguintes métodos não está realizando efetivamente uma ação específica, de forma que os métodos `metodoH(...)` e `metodoK(...)` devem existir na classe, já que ela está implementando a interface `interqualquer`.

```
public class ClasseA implements interqualquer {  
    public void metodoH(int par01, double par02) {  
        //...  
    }  
  
    public double metodoK(double par02) {  
        double d1;  
        //...  
        return d1;  
    }  
}
```

3.1.1 Passos (na programação) para uso dos eventos

Veremos agora que é possível realizar (programar) um controle de eventos de duas maneiras: implementando a interface equivalente ao evento, o que é feito na própria declaração da classe; e adicionando o evento ao componente controlado, sendo ambos programados ao longo de um método.

Com o objetivo de fazer com que a classe implemente as interfaces necessárias para controle dos eventos, siga os seguintes passos:

- Implementar as interfaces à classe (com o termo `implements`).
- Gerar todos os métodos das interfaces na classe.
- Adicionar os eventos diretamente ao componente controlado ("ligar" o componente), já com a programação de controle.

- Implementar a lógica nos métodos (ou evento específico) a serem controlados (observação: quando a interface possui vários métodos, apesar de ser necessário implementar os métodos na classe, não é preciso codificar o controle de todos eles, mas apenas para aqueles a que se quer efetivamente controlar).

Exemplo de aplicação

```
// Implementando o controle de eventos de janela
public class TelaPrincipal extends JFrame implements WindowListener {

    // Declaração dos Atributos / Componentes
    // Método de montagem da tela
    public void montaTela() {
        // ...
        // Adicionando o Evento ao Componente (ligando-o)
        // Obs.: o parâmetro indica o componente controlador
        // ...que no caso, é a própria janela
        this.addWindowListener(this);
        // ...
    }
    // Métodos da interface (obrigatório)
    // ...
    public void windowClosing(WindowEvent e) {
        // Lógica quando o X for apertado (p.ex.: encerra o programa)
        System.exit(0);
    }
    // ...
    // E mais TODOS os outros Métodos da interface (obrigatório)
}
```

Para o caso de controlar o evento utilizando a adição do evento ao componente, sem implementar a interface (Listener), mas o controle no parâmetro do método de adição de evento, devemos usar os seguintes passos:

- Adicionar o evento ao componente controlado.
- No parâmetro do método de adição da interface, instancia-se o Listener do evento a ser controlado (observação: neste caso, a implementação também exige a geração do método abstrato. Para alguns Listeners que possuem diversos métodos abstratos, existe uma classe Adapter a qual pode-se sobrescrever o método necessário, que se deseja controlar – como o `WindowAdapter`, ou o `MouseAdapter` etc.).
- Implementar o código de controle no método (ou evento específico) a ser controlado com a lógica necessária.



Observação

As classes [...]Adapter são abstratas que implementam a interface correspondente, e que possibilitam a sua utilização sem a necessidade de implementação de todos os métodos, como aqueles que não precisam ser controlados.

Exemplo de aplicação

```
public class TelaPrincipal extends JFrame {  
    // Declaração dos Atributos / Componentes  
    // Método de montagem da Tela  
    public void montaTela() {  
        // ...  
        // Adicionando o Evento ao Componente, já com a lógica  
        // ...e neste caso utilizando o Adapter dos eventos de janela  
        // ...evitando ter que adicionar todos os outros métodos  
        this.addWindowListener (new WindowAdapter() {  
            public void windowClosing(WindowEvent e) {  
                // Implementando a lógica  
                System.exit(0); // (p.ex.: encerrando o programa)  
            }  
        });  
        // ...  
    }  
}
```



Observação

Se utilizarmos o WindowListener ao invés do WindowAdapter, seremos obrigados a implementar todos os sete métodos existentes na interface WindowListener, mesmo que não coloquemos códigos em seu corpo.

3.2 Os eventos de ação

Os eventos de ação são os eventos (ou ações) ligados ao clicarmos em algum "botão" da tela, ou selecionarmos algum item de menu.

Ao implementar o ActionListener, que é a interface de eventos de ação, deve-se criar o método actionPerformed(...):

```
public void actionPerformed(ActionEvent e) {  
    // lógica de ação  
}
```

(este é o método abstrato daquela interface)

Desta forma, sempre que um componente for "ligado" (ao qual for adicionado o `ActionListener`), por exemplo, ao clicarmos sobre um botão, este método será acionado (iniciado e rodado) automaticamente.



Observação

O objeto "e" (representando o `ActionEvent` do parâmetro do método) recebe todas as informações necessárias para programar a performance dos botões.

Sabendo que ao acionar um botão (ou um componente controlado por esse evento), seja ele qual for, este método será acionado, como poderíamos então diferenciar os diversos botões existentes na tela, e com isso diferenciar as ações que cada um deles deverá realizar?

Para diferenciar um botão do outro, podemos utilizar um objeto de comparação, que recebe o objeto do componente exato que foi acionado (obtido a partir do `ActionListener`, do parâmetro do método), e que pode ser comparado aos componentes existentes na tela (na classe da janela), desde que eles sejam atributos da classe que representa a janela. Assim, fazemos:

```
Object obj = e.getSource();
```



Observação

Ao estudarmos o conceito de polimorfismo de classes, vimos que uma classe "mãe" (ou uma superclasse) pode se comportar como uma de suas classes filhas (as subclasses).

Neste caso, estamos instanciando um objeto do tipo `Object`, que é a classe mãe de todas as classes do Java, o qual receberá o objeto representante em memória do componente acionado, lembrando que o objeto "e" (do exemplo) representa o evento de ação (e é recebido via parâmetro do método).

Exemplo de aplicação

Consta a seguir um exemplo de programa que controla o evento de ação ao acionarmos um botão de Ok:

```
public class TelaPrincipal extends JFrame implements ActionListener {  
    // Declaração dos Componentes (atributos)  
    // Método de montagem da Tela  
    public void montaTela() { // ...por exemplo  
        // ...  
        // Adicionando o Evento ao Componente (ligando-o)
```

```

        botOk.addActionListener(this);
        // ...
    }
    public void actionPerformed(ActionEvent e) {
        // Para saber qual botão foi acionado...
        Object obj = e.getSource();
        if (obj.equals(botOk)) {
            // ...
            // Lógica de ação para o Botão Ok
            // ...
        }
    }
}

```

Pode-se colocar diretamente a lógica de funcionamento ao adicionar o evento de ação ao componente:


```

public class TelaPrincipal extends JFrame {
    // Declaração dos Componentes (atributos)
    // ...
    // Método de montagem da Tela
    public void montaTela() {
        // ...
        // Adicionando o Evento ao Componente, já com a lógica
        botOk.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                // ...
                // Lógica de ação para o Botão Ok
                // ...
            }
        });
        // ...
    }
}

```

3.3 Os eventos de janela

Os eventos de janela são aqueles que controlam o funcionamento geral da janela (e não especificamente de seus componentes), tais como: minimização, maximização, abertura inicial e fechamento da janela (ao sair do sistema), entre outros eventos.

Neste item, vamos mostrar o exemplo de como criar uma janela que é fechada (encerrando o programa) ao clicar sobre o ícone ( – ícone de fechamento) do canto superior direito de qualquer janela do Windows.

Ao criarmos uma janela com a linguagem Java e com a biblioteca Swing, o ícone de fechamento não vem preparado para fechar e encerrar automaticamente o programa que estamos rodando, de forma que a fim de fazer que isso funcione corretamente, precisamos inserir elementos de código que possibilitem tal execução.

Para este exemplo, criaremos uma janela simples (sem componentes internos), mas que estará preparada para fechar (e encerrar o programa) ao clicar no ícone de fechamento ().

Criaremos duas classes (Janela1 e Janela2), de forma que ambas estejam preparadas para funcionar com o ícone de fechamento, porém a Janela1 utilizará um processo mais geral, que funciona tanto com componentes do AWT quanto com aqueles do Swing. Já a Janela2 usará um processo que somente funciona com componentes do Swing, uma vez que o método utilizado pertence apenas à biblioteca Swing. Essa última é mais prática, porém como já dito, apenas funcionará com a biblioteca Swing.

- **Classe Janela1:** criando uma janela com a biblioteca AWT e utilizando uma forma de controle de fechamento e encerramento do programa que funciona em ambas as bibliotecas (AWT e Swing), mas que ainda é mais prática que a implementação da interface na declaração da classe.

```
import java.awt.Frame;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
public class Janela1 extends Frame {

    public Janela1() {
        //Método Construtor - que "constrói" a tela:

        this.setBounds(200, 200, 300, 300);
        //O método setBounds posiciona e dá tamanho ao componente
        //...componente.setBounds(posH, posV, tamH, tamV);

        // ##Adicionando o evento à Janela##
        //...vamos controlar o evento de
        //...fechamento e encerramento do Programa/Janela
        this.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
        //...nesta classe, não fizemos a implementação geral (total)
        //...da interface windowlistener,
        //...pois se o tivéssemos feito, teríamos que inserir
        //...todos os métodos desta interface.
        //... Assim, para esta classe (Janela1)
        //... utilizamos uma classe chamada WindowAdapter
        //... que nos facilitou a utilização dos métodos
        //... daquela interface.
        //... Como, por exemplo, o método "windowClosing(...)"

        this.setVisible(true);
    }

    public static void main(String[] args) {
        //... instanciando esta classe e
        //... consequentemente acionando seu Método Construtor
        //... e todas as suas funcionalidades
    }
}
```

```
        Janela1 j1 = new Janela1();  
    }  
}
```

- **Classe Janela2:** criando uma janela com a biblioteca Swing e utilizando uma forma de controle de fechamento e encerramento do programa que funciona apenas para a biblioteca Swing.

```
import javax.swing.JFrame;  
  
public class Janela2 extends JFrame {  
  
    public Janela2() {  
        this.setBounds(200, 200, 300, 300);  
  
        // Acionando um método desta biblioteca  
        // que faz funcionar o "ícone de fechamento"  
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
  
        this.setVisible(true);  
    }  
  
    public static void main(String[] args) {  
        //... instanciando esta classe e  
        //... consequentemente acionando seu Método Construtor  
        Janela2 j2 = new Janela2();  
    }  
}
```

Nesta segunda janela, como utilizamos a biblioteca Swing, podemos usar o método que faz funcionar o ícone de fechamento ().

Observação: no segundo caso, para acionarmos ou controlarmos os outros eventos de janela (como o de minimização etc.), devemos utilizar a classe WindowAdapter, como fizemos no exemplo anterior, da Janela1.

4 JDBC

JDBC (Java DataBase Connectivity) é a biblioteca de persistência em BD relacionais do Java. O processo de armazenamento de dados é também chamado de persistência da informação.

Para que o Java se conecte com um determinado BD, é necessário que haja um driver que permita o acesso a ele. Assim como no processo do ODBC, o JDBC funciona através de drivers responsáveis pela conexão com o banco, e pela execução das instruções SQL.

Os drivers Java são implementações das interfaces do pacote java.sql. Eles geralmente são disponibilizados gratuitamente na forma de um arquivo JAR (ou seja, Java ARchive) pelo fabricante do BD (em seus sites) ou por terceiros.

Neste material sugere-se em nossos exercícios a utilização do MySQL como servidor de BD. Essa sugestão ocorreu por se tratar de um produto que pode ser baixado e instalado gratuitamente, e pela facilidade de manipulação (a partir de seu workbench).

Para a instalação e configuração do MySQL, caso ainda não esteja instalado, pode-se encontrar dicas e sugestões no **Apêndice A** deste livro-texto. É importante saber que os exercícios estarão voltados a este SGBD (servidor de banco de dados). No entanto, sua teoria e o conhecimento gerado sobre a parte Java, podem ser ampliados a qualquer outro banco de dados.

Constituem as classes principais utilizadas no acesso a um BD:

- **Classe DriverManager:** gerencia o driver do BD (que foi carregado pelo ClassLoader).
- **Classe Connection:** responsável pela conexão com o BD (conexão com a base de dados).
- **Classe Statement (ou PreparedStatement):** representa o canal de comunicação com o BD, onde enviamos as queries para serem executadas pelo BD e de onde recebemos os dados vindos do BD.
- **Classe ResultSet:** recebe os dados em forma de listas de informações e utiliza ponteiros para controlar a posição dos registros.



Saiba mais

Para aprender mais sobre JDBC, leia o capítulo 24 da obra a seguir:

DEITEL, P.; DEITEL, H. *Java: como programar*. São Paulo: Pearson Education do Brasil, 2017.

4.1 Os conectores/drivers

Geralmente cada fabricante de BD fornece o driver Java (a biblioteca ".jar") para possibilitar a conexão com seu BD. É possível encontrar versões antigas do MySQL (como a 5.1, por exemplo), do driver no seguinte site:

<https://dev.mysql.com/downloads/connector/j/5.1.html>

Nas versões atuais do MySQL (8.0 ou posterior), o driver já está presente na instalação do próprio MySQL e geralmente se localiza no seguinte diretório:

C:\Program Files (x86)\MySQL\Connector J X.X (onde X.X é o número da versão instalada – por exemplo: C:\Program Files (x86)\MySQL\Connector J 8.0)

De todo modo, em qualquer um dos casos, o arquivo `mysql-connector-java-<versao>.jar` deverá ser importado (como Jar Externo) no Projeto do Eclipse, ou do NetBeans.

Driver do MySQL

Para gerar uma conexão e com isso acessar o BD propriamente dito, deve-se definir qual será o driver de acesso no programa (no código) utilizando-se o seguinte método da classe `Class`:

```
Class.forName(String name);
```

O que este método faz é solicitar ao `ClassLoader` (o responsável por carregar o driver fornecido pelo fabricante do BD) que ele instancie a classe especificada pela string definida em seu parâmetro (indicando o caminho para encontrar a classe do driver: `pacote.Classe`).

Por exemplo: `Class.forName("com.mysql.jdbc.Driver");`

Além disso, a classe `DriverManager` possui alguns métodos com o nome `getConnection()` em sobrecarga, eles são responsáveis por procurar entre os drivers carregados um que seja compatível com a URL fornecida, e com ele solicitar a abertura de uma conexão. Este método utiliza o driver carregado pelo `Class.forName(...)` para fornecer a conexão. A URL é basicamente:

```
protocolo:subprotocolo://IP_servidor:numero_porta/nome_base_dados
```

Por exemplo: `"jdbc:mysql://localhost:3306/aula_alpoo"`

Na string do exemplo anterior, o local do BD `mysql` é a própria máquina do usuário (por isso o `localhost`), e a base de dados se chama `aula_alpoo`.

Importando um arquivo .jar fornecido (como o driver do BD)

- **No Eclipse:** clique com o botão direito do mouse sobre o projeto, vá em `Properties` (última opção), selecione `Java Build Path` no navegador à esquerda, clique na aba `Libraries`, clique no botão `Add External JARs`, selecione o arquivo `JAR` do driver do fabricante do BD, e clique em `OK`.

Consta a seguir a figura demonstrando o processo:

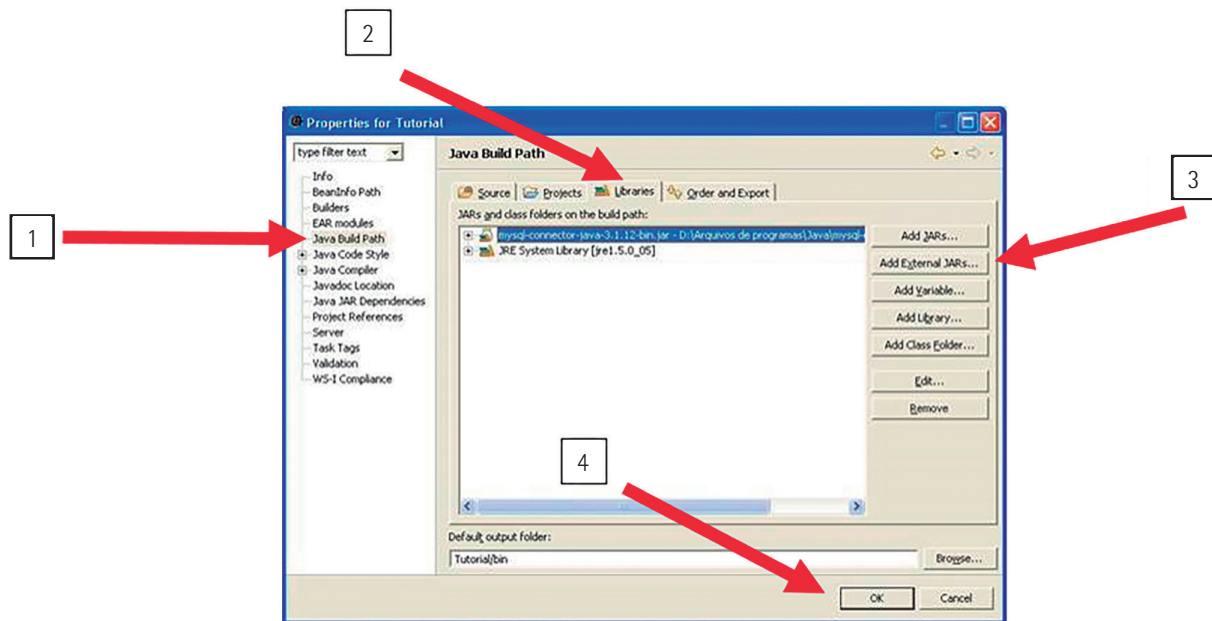


Figura 17 – Tela de propriedades do projeto: esquema de importação de "jar externo"

4.2 Conectando a um banco de dados do MySQL

Vimos que, para conectarmos um sistema em Java a um BD e manipularmos seus dados, utilizamos basicamente quatro classes: DriverManager, Connection, Statement (ou PreparedStatement), e ResultSet.

Com o objetivo de adquirirmos uma conexão com um BD do MYSQL, podemos aplicar a seguinte sequência de códigos (criando a classe GeraConexao).

Exemplo de aplicação

```
import java.sql.*;

public class GeraConexao {

    private String driver, url, login, senha;
    private Connection con;

    public GeraConexao() {
        driver = "com.mysql.cj.jdbc.Driver";
        //o driver acima é da versão 8.0 do mysql
        url = "jdbc:mysql://localhost:3306/escola";
        url += "?useTimezone=true&serverTimezone=UTC";
        //em alguns casos a linha acima será necessária
        login = "root"; // nome do usuário (ao instalar o mysql)
        senha = "1234"; // senha do usuário acima no mysql
        con = null;
        //acima: valor inicial da conexão (nulo) - antes do acesso
    }
}
```

```

public Connection getConnection() {
    //retornará a Classe de conexão obtida
    String msg = "";
    try{
        Class.forName(driver);
        this.con = DriverManager.getConnection(url,login,senha);
    }catch(ClassNotFoundException ex){
        msg = ":: ERRO :: Driver JDBC não encontrado na
aplicação!";

        System.out.println(msg);
    }catch(SQLException ex){
        msg = ":: ERRO :: Problemas na conexão com a fonte de
dados";

        System.out.println(msg);
    }catch(Exception ex){
        msg = ":: ERRO :: Outros problemas na conexão...";
        System.out.println(msg);
    }
    return this.con;
}
}

```

Importante: para que o código anterior funcione, o BD precisa conter os seguintes elementos:

Uma base de dados, chamada "escola";

Comentários sobre a classe do exemplo prévio

Pode-se observar que a classe apresentada importa a biblioteca (java.sql). Ela então utilizará esta biblioteca (que já é de seu conjunto de classes instaladas) para executar as linhas de código de acesso ao BD. A base de dados do mysql, a ser acessada pelo código deste exemplo, será a base escola e futuramente deverá conter uma tabela chamada alunos (especificada em exemplos posteriores neste livro-texto).

Além disso, ao instanciar a classe anterior, as principais strings utilizadas no acesso ao BD já são inicializadas (pelo método construtor da classe).

Sendo assim, a url indica qual é o sistema gerenciador do BD (no caso o mysql), além da localização da máquina do servidor do BD, que no caso informa uma instalação local (localhost – já que o BD está no próprio micro), assim como a porta de acesso 3306 (que é a porta padrão do mysql) e o nome do BD a ser acessado no mysql (que no exemplo é escola).

Com essa url, o nome do usuário (login) e a sua senha, que podem ser aqueles criados ao instalar o mysql, o programa solicita uma conexão (gera em memória um objeto do tipo Connection a partir do método getConnection(...), utilizando-se da classe DriverManager.

Veja que a classe do exemplo (classe GeraConexao) possui um método getConnection() que retorna uma conexão (um objeto do tipo Connection). Assim, para utilizá-la (GeraConexao), pode-se acionar as seguintes linhas de código.

Utilizaremos a classe Exemplo com um método main em que há uma simples chamada ao método que gera a conexão do objeto da classe GeraConexao mostrada anteriormente:

```
public class Exemplo() {  
    Connection con;  
    public static void main(String args[]) {  
        GeraConexao gc = new GeraConexao();  
  
        //gerando-se uma instância desta própria classe  
        Exemplo ex = new Exemplo();  
        ex.con = gc.getConnection();  
        if (ex.con != null) {  
            System.out.println("Conexão realizada com sucesso");  
        }  
    }  
}
```



Lembrete

Para que o exemplo anterior funcione com as classes Exemplo e GeraConexao, o driver do mysql já deve ter sido importado (pelo processo de adicionar jar externo) ao projeto em que ambas estão construídas.

4.3 Lendo informações do BD

Para que possamos ler os dados de um BD (as informações das tabelas de suas bases de dados), precisamos antes saber algumas informações, tais como: o nome do BD, o nome da tabela a ser acessada e os campos (as colunas) da tabela com os dados necessários.

Com o objetivo de realizarmos essa leitura, utilizamos a linguagem PLSQL de acesso a BD.

O exemplo a seguir é de um programa que acessa o BD escola e todos os dados (das colunas: id, nome, ra, idade) da tabela alunos.

Para que essa leitura de dados ocorra efetivamente no BD, utilizaremos a classe GeraConexao já exibida (para resgatar uma conexão com o BD). A classe Aluno servirá para representar e guardar os dados de cada um dos alunos cadastrados, ou seja, cada linha da tabela alunos do BD escola.

Para este programa, a tabela alunos, no mysql, deverá possuir os seguintes campos (colunas) criados:

- **id**: representando a primary-key de cada linha de dado (um valor do tipo inteiro, ou "int").
- **nome**: representando o nome do aluno (um valor do tipo "varchar" – que pode ser de tamanho 50 –, ou uma string).
- **ra**: representando o registro do aluno na escola ("varchar" – que pode ser de tamanho 10 –, ou uma string).
- **idade**: representando a idade do aluno (um valor numérico do tipo inteiro, ou "int").

Antes de tentarmos fazer funcionar o método `getallalunos(...)` da classe a seguir, é importante que, no MySQL, a tabela `alunos` da base de dados `escola` contenha informações (dados de: `id`, `nome`, `ra` e `idade`) dos alunos (com valores fictícios) de uma determinada escola.

Sendo assim, a partir dessas informações, a classe `AlunosBD` utilizará a classe `GeraConexao` para fazer uma leitura no BD, capturará (fará uma leitura) da tabela `alunos` os valores de suas colunas e permitirá a exibição na tela de console das informações cadastradas em cada linha da tabela `alunos` (como já descrito), desde que a conexão ocorra com sucesso. Desta forma, a classe `AlunosBD` deve estar localizada no mesmo projeto que a classe `GeraConexao`.

Vejamos a seguir a classe `Aluno`:

```
public class Aluno {  
  
    /*.....  
    ESTA CLASSE É O QUE PODEMOS CHAMAR DE UM "JAVA BEAN", OU SEJA, UMA CLASSE  
    JAVA QUE REPRESENTARÁ CADA DADO (CADA LINHA) DA TABELA ALUNO QUE LEREMOS (E  
    ALTERAREMOS) NO BANCO DE DADOS. OBSERVE QUE ELA CONTÉM, EM FORMA DE ATRIBUTOS,  
    CADA UM DOS CAMPOS DA TABELA DE DADOS.  
    .....*/  
    private int id;  
    private String nome;  
    private String ra;  
    private int idade;  
  
    /*Uma característica de um JavaBean, é que ele deve ser sempre uma classe  
    Encapsulada, com os atributos privados e com os métodos "set" e "get"*/  
    private int getId() {  
        return id;  
    }  
    public void setId(int id) {  
        this.id = id;  
    }  
    public String getNome() {  
        return nome;  
    }  
    public void setNome(String nome) {  
        this.nome = nome;  
    }  
    public String getRa() {  
        return ra;  
    }  
}
```



```

    }
    public void setRa(String ra) {
        this.ra = ra;
    }
    public int getIdade() {
        return idade;
    }
    public void setIdade(int idade) {
        this.idade = idade;
    }
}

```

Vejamos a seguir a classe AlunosBD (com os métodos getAllAlunos(), mostraAlunos(...), abreCon() e fechaCon()):

```

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;

public class AlunosBD {

    public Connection con;

    public ArrayList<Aluno> getAllAlunos() {
        ArrayList<Aluno> arrAlunos = new ArrayList<Aluno>();
        // ...abrindo a conexão
        this.abreCon();
        String query = "SELECT * FROM alunos";
        try {
            Aluno a1;
            PreparedStatement stmt = con.prepareStatement(query);
            ResultSet rs = stmt.executeQuery();
            // Montando um ArrayList de "Array e Strings"
            while (rs.next()) {
                a1 = new Aluno();
                a1.setId(rs.getLong("id"));
                a1.setNome(rs.getString("nome"));
                a1.setRa(rs.getString("ra"));
                a1.setIdade(rs.getInt("idade"));
                arrAlunos.add(a1);
            }
            stmt.close();
        } catch (Exception e) {
            System.out.println(":: ERRO :: Problemas com a leitura de
dados no BD...");
        }
        // ...fechando a conexão
        this.fechaCon();
        return arrAlunos;
    }

    public void mostraAlunos(ArrayList<Aluno> arrAlunos) {
        String txt = "";
        Aluno a1;
    }
}

```

```

        //Montando a String com todos os alunos cadastrados
        for (int x = 0; x < arrAlunos.size(); x++) {
            al = arrAlunos.get(x);
            txt += "\n" + al.getId();
            txt += "# " + al.getNome();
            txt += " - idade: " + al.getIdade();
            txt += " (ra: " + al.getRa() + ")";
        }
        // Mostrando na console a string já com todos os cadastros
        System.out.println(txt);
    }

    //.....PARA LIDAR COM O BANCO DE DADOS.....
    //.....

    private void abreCon() {
        GeraConexao gc = new GeraConexao();
        this.con = gc.getConnection();
    }

    private void fechaCon() {
        try {
            con.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}

```

Esta classe funcionará com a classe GeraConexao (mostrada e criada no item 4.2).

No exemplo anterior, no método `getAlAlunos()`, utilizamos um objeto do tipo `ResultSet` para receber as informações lidas no BD (a partir da query – um comando – SQL select...).

Consequentemente, para cada linha lida no BD, o programa gera um objeto do tipo `Aluno`, sendo que cada um desses objetos é preenchido com as informações do aluno (lidas no BD), e em seguida o objeto (do tipo `Aluno`) é inserido em um `ArrayList` (uma matriz de objetos).

Ao final da execução do método `getAlAlunos()`, é retornado um objeto do tipo `ArrayList` que conterà vários objetos do tipo `Aluno`, cada um deles com as informações lidas nas linhas da tabela alunos da base de dados escola (do `MySQL`).

Já o método `mostraAlunos` recebe, via parâmetro, o `ArrayList` com todos os objetos do tipo `Aluno` (já preenchidos nos seus atributos com os dados do BD) lê um a um os dados de cada objeto do `ArrayList` montando uma string com todos os dados (de todos os objetos), mostrando o valor da string na tela da console.

A classe `TestaAlunos` (a seguir) utiliza a classe `AlunosBD` para efetivamente rodar os dois métodos da classe `AlunosBD` (um para pegar os dados do BD e o outro para mostrar na console os valores lidos)

e imprimir na tela de console os dados da tabela alunos do BD (esta classe deve ser criada no mesmo projeto das classes dos exemplos anteriores).

Verifique os pré-requisitos de funcionamento da classe AlunosDB descritos logo após o exemplo em código.

```
import java.util.ArrayList;
public class TestaAlunos {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        AlunosBD ad = new AlunosBD();
        ArrayList<Aluno> arrAlunos = ad.getAllAlunos();
        ad.mostraAlunos(arrAlunos);
    }
}
```

4.4 Alterando informações no BD

A alteração de dados de um BD ocorre através das queries (ou seja, dos comandos SQL) do tipo: INSERT, UPDATE e DELETE – já que eles alteram informações no BD. Vejamos suas funcionalidades:

- **INSERT...:** é o comando que permite a inserção de uma nova informação na tabela (acrescenta uma linha de informação no BD).
- **UPDATE...:** é o comando que permite a modificação de informações já existentes na tabela.
- **DELETE...:** é o comando que permite a exclusão de uma linha de informações na tabela.



Observação

Esses comandos SQL podem ser estudados de forma mais profunda em outra disciplina deste mesmo curso que ensina a lidar com BD. Neste livro-texto, utilizaremos alguns exemplos da forma mais simples possível, sobre tais comandos, já que nosso objetivo aqui é utilizar a linguagem Java para acessar informações do BD.

Em um programa Java que realiza os tipos citados de alteração no BD, a diferença poderá ser observada nos exemplos a seguir, de forma que:

- nesses casos não utilizamos a classe ResultSet, já que essa classe é usada para receber informações do BD (como seria para um comando SQL do tipo SELECT).
- para esses casos, o método da classe PreparedStatement a ser rodado será executeUpdate(), e não o método executeQuery, como foi na leitura de informações, já que o executeUpdate() realiza alterações de informações no BD.

Mostraremos no exemplo a seguir um método cujo nome será `insert(...)` e que permite fazer a inserção de uma linha de novas informações no BD. Ele deve ser criado na classe `AlunosBD` apresentada previamente juntamente aos métodos que nela já existam.

Consta na sequência o método `insert()` a ser criado (inserido) na classe `AlunosBD`:

```
public void insert(Aluno al) {  
  
    // montando a query que inserirá uma informação no BD  
    String query = "insert into alunos ";  
    query += "(id, nome, ra, idade) ";  
    query += "values (?, ?, ?, ?)";  
    try {  
        // ...abrindo a conexão  
        this.abreCon();  
        PreparedStatement stmt = con.prepareStatement(query);  
        //Comandos Java que preenchem os  
        //..."pontos de interrogação" da query  
        stmt.setLong(1, al.getId());  
        stmt.setInt(4, al.getIdade());  
        stmt.setString(3, al.getRa());  
        stmt.setString(2, al.getNome());  
        //Executando a query no BD  
        stmt.executeUpdate();  
        stmt.close();  
        this.fechaCon();  
    } catch (Exception e) {  
        e.printStackTrace();  
    }  
}
```

Este método recebe um objeto do tipo `Aluno` como parâmetro, já preenchido com as informações (de apenas um aluno) a serem inseridas no BD (como é somente de um aluno, será inserida apenas uma linha na tabela `aluno`).

Portanto, para testarmos o método mencionado, podemos criar a classe `TestaInsertAlunos` mostrada na sequência, que possui o método `main` e utiliza o método anterior para inserir um dado (uma linha de dados) no BD. Ela deve ser criada no mesmo diretório das classes `Aluno` e `AlunosBD`. Vejamos como executá-la:

```
public class TestaInsertAlunos {  
    public static void main(String[] args) {  
        // Instanciando a classe AlunosBD  
        AlunosBD ad = new AlunosBD();  
        // Criando um "JAVA BEAN" "Aluno"  
        // (um objeto do tipo "Aluno" - com dados)  
        //com as informações de um Aluno  
        Aluno al = new Aluno();  
        al.setId(15);  
        //Obs.: este "id" não deve existir no BD  
        al.setNome("Napoleão Bonaparte");  
        al.setIdade(250);  
    }  
}
```

```
        a1.setRa("FRA1769AGO");  
        //...ESTE DADOS SERÁ INSERIDO NO BD  
        ad.insert(a1);  
    }  
}
```

Para realizarmos outros tipos de alteração de dados em BD (como os realizados pelos comandos SQL update e delete), basta refazermos corretamente o texto da string query do exemplo prévio utilizando o mesmo método empregado para a operação de insert (o método `executeUpdate()` da classe `PreparedStatement`), que permite a execução de alterações no BD.



Resumo

Nesta unidade, aprendemos a dar funcionalidades aos componentes da nossa tela. Quando um usuário interage com um sistema, ele está ativando algum evento, de forma que na linguagem Java existe para cada tipo de evento uma interface específica que determina a sua funcionalidade. Vimos que essas ativações ocorrem ao adicionarmos a interface ao componente que se quer controlar.

Além de adicionarmos o evento ao componente, codificamos todo o controle das ações nos métodos implementados a partir das interfaces. Os eventos de interação (usuário x sistema) são representados na linguagem Java por interfaces que conhecemos como Listeners.

Aprendemos que é possível controlar os seguintes tipos de eventos: de janela, de mouse, de teclado, e de ação (quando o usuário clica ou aciona um componente específico na tela), ou seja, tudo aquilo que utilizamos para interagir com um sistema. Um sistema construído na linguagem Java está preparado para acionar um método específico quando o usuário interage com o sistema. Também mostramos que em quase todos os sistemas existe a necessidade de guardar as informações neles inseridas em bancos de dados.

O Java possui em sua biblioteca padrão classes (Connection, DriverManager, Statement e ResultSet) que possibilitam a persistência (inserção e leitura) das informações nesses bancos de dados. Para que o sistema se conecte com um BD específico (que nos exemplos deste livro-texto conectamos com o SGBD MySQL), precisamos de uma biblioteca de classes, geralmente fornecida pela empresa proprietária do BD (isto é, a companhia que criou o MySQL), biblioteca essa fornecida na forma de um arquivo com extensão .jar, que adicionamos ao projeto. Consequentemente, elaboramos uma série de exemplos que permitem: conectar com o BD; ler as informações de uma tabela do banco; inserir ou alterar informações naquela tabela, ações essas feitas a partir de queries (no formato SQL), geradas em strings e enviadas pelos statements, para execução direta no BD.



Exercícios

Questão 1. Os eventos são as possíveis ações de interação do usuário com o sistema que ele está utilizando. Eles podem acontecer pelo acionamento de um botão ou pela passagem do mouse sobre algum componente, entre outros. A linguagem Java controla os eventos por meio de interfaces denominadas listeners, criadas para serem implementadas pelas classes de componentes cujos eventos precisam ser controlados.

A respeito desse tema, avalie as afirmativas a seguir.

I – A interface `KeyListener` define métodos para o reconhecimento de quando uma tecla foi pressionada, solta ou digitada.

II – A interface `MouseListener` define métodos para o reconhecimento de quando o mouse teve seu botão clicado, entrou em um componente, saiu de um componente, e teve seu botão pressionado ou seu botão solto.

III – A interface `WindowListener` define métodos para o reconhecimento de quando um componente foi adicionado ou removido de um container.

É correto o que se afirma em:

A) I, apenas.

B) II, apenas.

C) I e III, apenas.

D) II e III, apenas.

E) I, II e III.

Resposta correta: alternativa A.

Análise das afirmativas

I – Afirmativa correta.

Justificativa: a interface `KeyListener` é utilizada para monitorar eventos que são disparados pelo pressionamento, pela liberação ou pela digitação de uma tecla qualquer. Ou seja, ela monitora o uso do teclado.

II – Afirmativa incorreta.

Justificativa: a interface `MouseWheelListener` define um método para o reconhecimento de quando a roda do mouse foi movida. O texto da afirmativa descreve a interface `MouseListener`.

III – Afirmativa incorreta.

Justificativa: a interface `WindowListener` define métodos para o reconhecimento de quando uma janela foi ativada, fechada, desativada, desiconificada, iconificada, aberta ou abandonada. O texto da afirmativa descreve a interface `ContainerListener`.

Questão 2. (Consulplan/2017, adaptada) A maioria das aplicações, independentemente do porte, utiliza a persistência de dados atrelada aos seus sistemas. O banco de dados se faz necessário em uma aplicação não apenas para persistir as informações, mas para que nos comuniquemos com ele para recuperar, modificar e apagar informações. Portanto, podemos afirmar que o gerenciamento dos dados é de fundamental importância ao correto funcionamento da aplicação.

Quando se trata de persistência de dados em Java, geralmente a forma utilizada para guardar dados é um banco de dados relacional. A fim de abrir uma conexão com um banco de dados, precisamos utilizar sempre um driver. A classe `DriverManager` é responsável por realizar essa comunicação, e a API do Java JDBC (Java Database Connectivity) fornece a especificação de como a linguagem Java se comunicará com um banco de dados. O parâmetro passado é do tipo `String`, contendo a URL para localizar o banco de dados que, por sua vez, contém as informações para a conexão com o banco de dados.

Assinale a alternativa correta que contém os parâmetros URL padrão, a senha e o login para a conexão com o banco MySQL, por meio do comando `DriverManager.getConnection()`, considerando que o usuário, o servidor e a porta do banco são padrão, o banco não tem senha e o nome do banco de dados é teste.

A) ("mysql:jdbc://localhost:3306/teste","", "root").

B) ("jdbc:mysql://localhost:3306/teste","root", "").

C) ("mysql:jdbc://localhost:3306/teste","root", "").

D) ("jdbc:mysql://localhost:3306/teste","", "root").

E) ("root","jdbc:mysql://localhost:3306/teste", "").

Resposta correta: alternativa B.

Análise da questão

Com o objetivo de abrir uma conexão com um BD, precisamos utilizar sempre um driver. A classe `DriverManager` é a responsável por se comunicar com todos os drivers disponíveis. Para isso, utilizamos o método `getConnection()`.

Os parâmetros do comando `DriveManager.getConnection()` utilizam a sintaxe a seguir.

```
DriverManager.getConnection("URL","login","senha")
```

Vamos, agora, comentar cada um dos parâmetros.

- **Parâmetro URL:** uma URL básica de conexão entre a linguagem Java e o BD MySQL tem o seguinte formato, considerando servidor e porta do banco padrão:

```
jdbc:mysql://localhost:3306/nome_do_banco_de_dados
```

Como o nome do BD da questão é teste, temos o formato a seguir:

```
jdbc:mysql://localhost:3306/teste
```

- **Parâmetro login:** trata-se do nome do usuário. Ao instalar o MySQL, o nome padrão do usuário é root.
- **Parâmetro senha:** trata-se da senha de acesso ao BD. De acordo com o enunciado, o banco não tem senha.

Logo, a instrução completa a ser inserida nos parênteses da instrução `DriveManager.getConnection()` é dada de acordo com o exposto a seguir.

```
"jdbc:mysql://localhost:3306/teste","root",""
```
