Interfaces Gráficas em Java

Atualmente, o Java suporta, oficialmente, dois tipos de bibliotecas gráficas: **AWT** e **Swing**. A AWT foi a primeira API para interfaces gráficas a surgir no Java e foi, mais tarde, superada pelo Swing (a partir do Java 1.2), que possui diversos benefícios em relação a seu antecessor.

AWT e Swing são bibliotecas gráficas oficiais inclusas em qualquer JRE ou JDK. Além destas, existem algumas outras bibliotecas de terceiros, sendo a mais famosa, o SWT - desenvolvida pela IBM e utilizada no Eclipse e em vários outros produtos.

Portabilidade

As APIs de interface gráfica do Java favorecem, ao máximo, o lema de portabilidade da plataforma Java. O **look-and-feel** do Swing é único em todas as plataformas onde roda, seja ela Windows, Linux, ou qualquer outra. Isso implica que a aplicação terá exatamente a mesma interface (cores, tamanhos etc) em qualquer sistema operacional.

Com Swing, não importa qual sistema operacional, qual resolução de tela, ou qual profundidade de cores: sua aplicação se comportará da mesma forma em todos os ambientes.

Look And Feel

Look-and-Feel (ou LaF) é o nome que se dá à "cara" da aplicação (suas cores, formatos e etc). Por padrão, o Java vem com um look-and-feel próprio, que se comporta exatamente da mesma forma em todas as plataformas suportadas.

Componentes JFrame, JPanel e JButton

Os componentes mais comuns já estão frequentemente prontos e presentes na API do Swing. Contudo, para montar as telas que são específicas do seu projeto, será necessário compor alguns componentes mais básicos, como JFrames, JPanels, JButtons, JLabels, etc.

Classe JFrame

É na interface gráfica que se definem o desenho do programa e seu funcionamento, é no formulário que são inseridos os componentes de controle e operação do programa em execução. Na linguagem Java isso é conseguido com a classe JFrame.

Classe JLabel

A classe JLabel permite definir um texto que pode ser adicionado a um outro componente (frame, painel, etc.). Podem ser definidads várias propriedades para esse texto, tais como alinhamento, tipo de letra, tamanho, cor etc. Uma possível sintaxe para a declaração e criação de um objeto JLabel é:

JLabel <nome-do-objeto> = new JLabel ("<texto do label>", JLabel.<alinhamento>);

Exemplo:

JLabel lblNome = new JLabel("Nome", JLabel.RIGHT);

Esse exemplo cria um objeto chamado lbNome contendo o texto "Nome", alinhado à direita.

Resumo dos métodos da classe JLabel:

Método	Função
JLabel()	Cria um Label vazio (sem texto)
JLabel(String)	Cria um Label com o texto dado
JLabel(String,int)	Cria um Label com o texto e o alinhamento dados
JLabel(String, Image)	Cria um Label com o texto e a imagem dados
JLabel(String, Image, int)	Cria um Label com o texto, a imagem e o alinhamento dados
getText()	Obtém o texto do Label
setText()	Especifica o texto do Label

Classe JTextField

Permite criar uma caixa de texto gráfica em que o usuário pode digitar dados. Assim como JLabel, existem diversas propriedades cujos conteúdos podem ser modificados. Uma possível sintaxe para a declaração e criação de um objeto JTextField é:

JTextField <nome do objeto> = new JTextField();

Exemplo:

JTextField tfNome = new JTextField();

Esse exemplo cria um objeto chamado tfNome com conteúdo em branco.

Resumo dos métodos da classe JTextField:

Método	Função
JTextField()	Cria uma caixa de texto vazia
JTextField(String)	Cria uma caixa de texto com a string dada
JTextField(String,int)	Cria uma caixa de texto com a string e a quantidade de colunas especificada
JTextField(int)	Cria uma caixa de texto com a quantidade de colunas especificada
getText()	Obtém o texto do objeto
getSelectedText()	Obtém o texto selecionado no objeto
isEditable()	Verifica se o componente é editável ou não
selectAll()	Seleciona todo o texto
setEditable(boolean)	Especifica se o componente é editável ou não
setText()	Especifica o texto contido no componente

Classe JPasswordField

De forma semelhante à JTextField, a classe JPasswordField permite criar um componente visual em que o usuário digita os caracteres, porém eles são substituídos (visualmente) por outro caractere. O funcionamento da classe JPasswordField é praticamente o mesmo da classe JTextField, a diferença é que o caractere digitado é substituído por outro para ocultar a senha digitada. O caractere default que aparece no momento da digitação é o asterisco (*), entretanto qualquer caractere pode ser definido pelo método setEchoChar. Uma possível sintaxe para a declaração e criação de um objeto JPasswordField é:

JPasswordField <nome do objeto> = new JPasswordField();

Exemplo:

JPasswordField pfSenha = new JPassowordField();

Esse exemplo cria um objeto chamado pfSenha com conteúdo em branco.

Resumo dos métodos da classe JPassowordField

Método	Função
JPasswordField()	Cria uma caixa de texto vazia
JPasswordField (String)	Cria uma caixa de texto com a string dada
JPasswordField (int)	Cria uma caixa de texto com a quantidade de colunas especificada

Método	Função
getPassword()	Obtém o texto do objeto, porém retornando um array do tipo char. Cada caractere é armazenado num elemento do array
getEchoChar()	Obtém o caractere usado na substituição dos caracteres digitados
setEchoChar()	Define o caractere a ser usado em substituição aos caracteres digitados

Classe JButton

A classe JButton permite a criação de botões gráficos a serem adicionados em outros componentes gráficos (como painéis e frames). Um botão pode ser criado com apenas um texto e/ou com ícones para tornar o ambiente mais intuitivo. Neste primeiro exemplo vamos criar botões da maneira mais simples.

Uma possível sintaxe para a declaração e criação de um objeto JButton é:

JButton <nome do objeto> = new JButton(<texto do botão>);

Exemplo:

JButton btCalcular = new JButton (“Calcular”);

Esse exemplo cria um objeto chamado btCalcular contendo o texto “Calcular”.

Resumo dos métodos da classe JButton

Método	Função
Button()	Cria um botão sem texto
Button(String)	Cria um botão com o texto dado
Button(String, Image)	Cria um botão com o texto e a imagem dados
getText()	Obtém o texto do botão
setText(String)	Especifica o texto do botão
setEnabled (boolean)	Define se o botão está habilitado (true) ou desabilitado (false)
setHorizontalTextPosition()	Define o tipo de alinhamento horizontal do texto em relação a uma imagem. Pode assumir LEFT (esquerda) ou RIGHT (direita)
setMnemonic(char)	Define uma letra que será usada como acionadora do evento clique, em conjunto com a tecla ALT
setToolTipText(String)	Possibilita atrelar uma mensagem ao botão. Quando o ponteiro do mouse estaciona sobre o botão, a mensagem é apresentada
setVerticalTextPosition()	Define o tipo de alinhamento vertical do texto em relação a uma imagem. Pode assumir TOP (topo) ou BOTTOM (abaixo)

Quando um botão de ação é pressionado, ele gera um ActionEvent. A classe ActionEvent é definida por AWT e também é usada por Swing. JButton fornece os métodos a seguir, que são usados para adicionar ou remover o ouvinte de uma ação:

void addActionListener (ActionListener al)

void removeActionListener (ActionListener al)

Aqui, al especifica um objeto que receberá notificações de eventos. Esse objeto deve ser instância de uma classe que implemente a interface ActionListener. A interface ActionListener só define um método: actionPerformed( ). Ele é mostrado abaixo:

void actionPerformed(ActionEvent ae)

Esse método é chamado quando um botão é pressionado. Em outras palavras, ele é o tratador de eventos chamado quando ocorre um evento de pressionamento de botão. A implementação de actionPerformed( ) deve responder rapidamente ao evento e retornar. Como regra geral, os tratadores de eventos não devem se ocupar de operações longas porque isso retarda o aplicativo inteiro. Se um procedimento demorado precisar ser executado, uma thread separada deve ser criada para esse fim. Usando o objeto ActionEvent passado para actionPerformed( ), você pode obter várias informações úteis relacionadas ao evento de pressionamento de botão. A usada por este capítulo é o string do comando de ação associado ao botão. Por padrão, esse é o string exibido dentro do botão. O comando de ação é obtido com uma chamada ao método getActionCommand( ) no objeto de evento. Esta é sua declaração:

String getActionCommand( )

O comando de ação identifica o botão, logo, quando são usados dois ou mais botões dentro do mesmo aplicativo, o comando de ação fornece uma maneira fácil de determinarmos que botão foi pressionado.

Exemplo prático

A tela que vamos criar para o "fatec" terá vários componentes. Para começarmos, faremos dois botões importantes: um dispara a soma entre dois números e outro permite ao usuário sair da aplicação.

Criar um componente do Swing é bastante simples! Por exemplo, para criar um botão:

```
JButton botao = new JButton("SOMA");
```

Contudo, esse botão apenas não faz nossa tela ainda. É preciso adicionar o botão para sair da aplicação. O problema é que, para exibirmos vários componentes organizadamente, é preciso usar um painel para agrupar esses componentes: o JPanel:

```
JButton botaoCarregar = new JButton("SOMA");
JButton botaoSair = new JButton("Sair");

JPanel painel = new JPanel();
painel.add(botaoCarregar);
painel.add(botaoSair);
```

Por fim, temos apenas objetos na memória. Ainda é preciso mostrar esses objetos na tela do usuário. O componente responsável por isso é o JFrame, a moldura da janela aberta no sistema operacional:

```
JFrame janela = new JFrame("fatec");
janela.add(panel);
janela.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
janela.pack();
janela.setVisible(true);
```

O método pack() da classe JFrame, chamado acima, serve para organizar os componentes do *frame* para que eles ocupem o menor espaço possível. E o setVisible recebe um boolean indicando se queremos que a janela esteja visível ou não.

Adicionamos também um comando que indica ao nosso *frame* que a aplicação deve ser terminada quando o usuário fechar a janela (caso contrário a aplicação continuaria rodando).

```
janela.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

Tratando Eventos

Até agora os botões que fizemos não têm efeito algum, já que não estamos tratando os **eventos** que são disparados no componente. E quando falamos de botões, em geral, estamos interessados em saber quando ele foi *disparado* (clicado). No linguajar do Swing, queremos saber quando uma ação (action) aconteceu com o botão.

Mas como chamar um método quando o botão for clicado? Como saber esse momento?

O Swing, como a maioria dos toolkits gráficos, nos traz o conceito de **Listeners (ouvintes)**, que são interfaces que implementamos para sobrescrever métodos que serão disparados pelas ações sobre um botão, por exemplo. Os eventos do usuário são capturados pelo Swing que chama o método que implementamos.

No nosso caso, para fazer um método disparar ao clique do botão, usamos a interface ActionListener. Essa interface nos dá um método actionPerformed:

Classes Internas e Anônimas

Em uma tela grande frequentemente temos muitos componentes que disparam eventos e muitos eventos diferentes. Temos que ter um objeto de listener para cada evento e temos que ter classes diferentes para cada tipo de evento disparado.

Só que é muito comum que nossos listeners sejam bem curtos, em geral chamando algum método da lógica de negócios ou atualizando algum componente. E, nesses casos, seria um grande problema de manutenção criarmos uma classe *top level* para cada evento.

Voltando ao exemplo do botão *Ok*, seria necessário criar uma classe para cada um deles e, aí, como diferenciar os nomes desses botões: nomes de classes diferentes? Pacotes diferentes? Ambas essas soluções causam problemas de manutenibilidade.

O mais comum para tais casos é criarmos **classes internas** que manipulam os componentes que desejamos tratar junto à classe principal.

Classes internas são classes declaradas dentro de outras classes:

```
public class Externa {
  public class Interna {

  }
}
```

Uma classe interna tem nome `Externa.Interna` pois faz parte do objeto da classe externa. A vantagem é não precisar de um arquivo separado e que classes internas podem acessar tudo que a externa possui (métodos, atributos etc). É possível até encapsular essa classe interna marcando-a como `private`. Dessa forma, apenas a externa pode enxergar.

Fora isso, são classes normais, que podem implementar interfaces, ter métodos, ser instanciadas etc.

Uma forma mais específica de classe interna é a chamada **classe anônima** que é muito vista em códigos com Swing. De forma simples, cria-se uma classe sem mesmo declarar seu nome em momento algum, isto é, o código `public class SairListener implements ActionListener{` não existirá em lugar algum.

Em um primeiro momento, a sintaxe das classes anônimas pode assustar. Vamos usar uma classe anônima para tratar o evento de clique no botão que desliga a aplicação:

```
JButton botaoSair = new JButton("Sair");

ActionListener sairListener = new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    System.exit(0);
  }
};

botaoSair.addActionListener(sairListener);
```

Repare que precisamos de um objeto do tipo ActionListener para passar para nosso botão. Normalmente criaríamos uma nova classe para isso e daríamos new nela. Usando classes anônimas damos new e implementamos a classe ao mesmo tempo, usando a sintaxe que vimos acima.

E podemos simplificar mais ainda sem a variável local sairListener:

```
JButton botaoSair = new JButton("Sair");
botaoSair.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    System.exit(0);
  }
});
```

Embora a sintaxe diga `new ActionListener()` sabemos que é impossível criar um objeto a partir de uma interface Java. O que essa sintaxe indica (e as chaves logo após o parênteses indicam isso) é que estamos dando `new` em uma *nova classe* que implementa a interface `ActionListener` e possui o método `actionPerformed` cuja implementação chama o `System.exit(0)`.

Como criamos uma classe no momento que precisamos, é comum nos perguntarmos qual é o nome dessa classe criada. A resposta para essa pergunta é exatamente a esperada: não sabemos! Ninguém definiu o nome para essa implementação particular de um `ActionListener`, então, por não ter nome definido, classes como essas são chamadas de **classes anônimas**.

//importando todas as funções das bibliotecas swing e awt

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;


public class fatec {

 private JFrame janela;     //cria componente JFrame
 private JPanel painelPrincipal;   //cria componente JPainel
 private Label label1,lblr;   // cria componentes label
 private TextField t1,t2,t3;  // cria componentes TextField


   public static void main(String[] args) {
     new fatec().montaTela();    //define método montatela em Classe
          //fatec, responsável pela interface gráfica

}

 private void montaTela() {      //chama os métodos que formarão a tela

     preparaJanela();
     preparaPainelPrincipal();
     preparaLabel();
     preparaText();
     preparaBotaoCarregar();
     preparaBotaoSair();
     mostraJanela();

    }



     private void preparaJanela() {

     janela = new JFrame("Exemplo");

          // como  é necessário utilizar o Jframe no método mostraJanela é necessário
          // que janela seja um atributo ao invés de uma variável local

janela.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

// indica ao nosso frame que a aplicação deve ser terminada quando o usuário
fechar a janela
}
```

```
   private void mostraJanela() {

    janela.pack();
```

// serve para organizar os componentes do *frame* para que eles ocupem o menor //espaço possível.

```
    janela.setSize(600, 300);   //dimensões  do Frame
    janela.setVisible(true);      // recebe um boolean indicando se queremos que
                                  //a janela esteja visível ou não.
  }
```

// prepara o Painel que receberá os botões e adiciona o Painel no Frame

```
  private void preparaPainelPrincipal() {

  painelPrincipal = new JPanel();
  janela.add(painelPrincipal);

}
```

// insere texto no label1 e adiciona o label no painel

```
private void preparaLabel(){

  label1 = new Label("Calculando a Soma de 2 números");
  painelPrincipal.add(label1);

};
```

// insere texto no label, espaços aos TextFields e adiciona os componentes no painel;

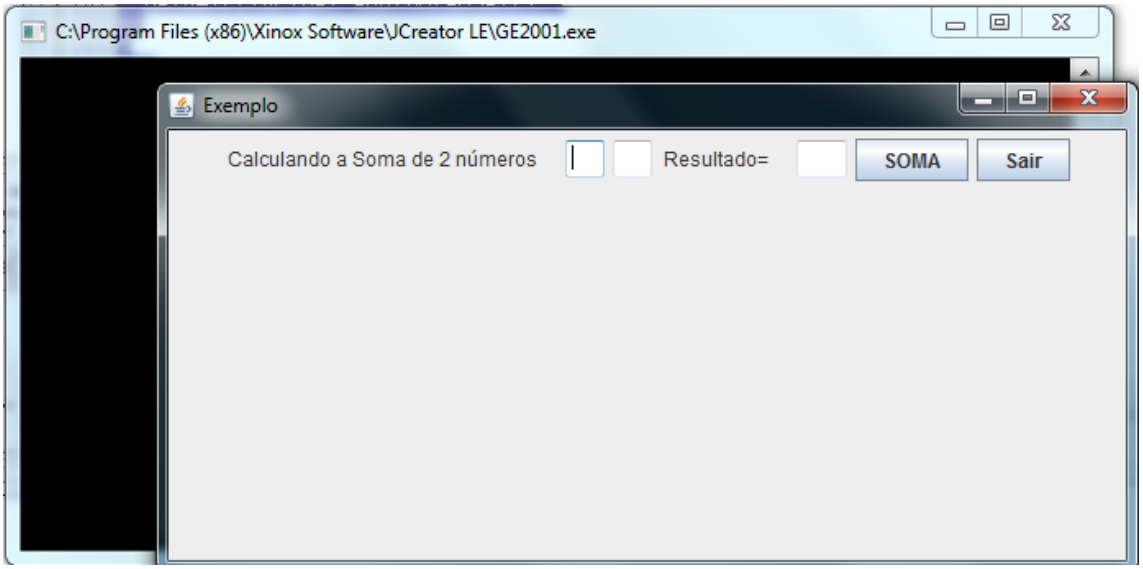
```
private void preparaText(){

  t1 = new TextField("");
  t2= new TextField("");
  lblr = new Label("Resultado=");
  t3= new TextField(" ");
painelPrincipal.add(t1);
painelPrincipal.add(t2);
painelPrincipal.add(lblr);
painelPrincipal.add(t3);

};
```

//criar botão, usar a interface ActionListener para disparar o clique e acionar o método actionPerformed

```
private void preparaBotaoCarregar() {

 JButton botaoCarregar = new JButton("SOMA");
 botaoCarregar.addActionListener(new ActionListener() {

 @Override
  public void actionPerformed(ActionEvent e) {

                int soma=Integer.parseInt(t1.getText())+ Integer.parseInt(t2.getText());
                 String r = Integer.toString(soma);
                t3.setText(r);

        }

});



 painelPrincipal.setLocation(50,100);  //posição do painel no frame
 painelPrincipal.add(botaoCarregar);  //adiciona botão ao painel

}
```

//criar botão, usar a interface ActionListener para disparar o clique e acionar o método actionPerformed

```
private void preparaBotaoSair() {

  JButton botaoSair = new JButton("Sair");
  botaoSair.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
      System.exit(0);

    }

  });

  painelPrincipal.add(botaoSair);  //adiciona botão Sair ao painel

}

}
```


C:\Program Files (x86)\Xinox Software\JCreator LE\GE2001.exe
Exemplo
Calculando a Soma de 2 números
Resultado=
SOMA
Sair