Aula 13

Professores:

Carlos Bazílio
Isabel Rosseti

Orientação por Eventos

Conteúdo:

- revisão da aula anterior
- motivação
- orientação por eventos
- exercício

Fundação CECIERJ
Consórcio cederj

Revisão da aula anterior

- hierarquia de composição:
  - componente
  - contêiner
- elementos de interface Swing:
  - janela (JFrame)
  - painel (JPanel)
  - componentes atômicos: JButton, JLabel, ...
- diagramadores

Motivação

- o que deve ocorrer quando o usuário clicar um botão?
- como alterar o conteúdo de um componente quando um outro sofre alguma alteração?
- solução: estabelecer o tratamento de eventos de interface

Orientação por eventos

- modelo de programação que tornou-se bastante difundido com o uso de interfaces gráficas

- o programa deixa de ter o controle do fluxo de execução

Orientação por eventos

- o controle passa a um sistema encarregado de gerenciar a interface
- o programa passa a ser chamado pelo sistema quando algum evento é gerado na interface

Mecanismos de **callback**

- para que o programa possa ser chamado pelo sistema, ele deve registrar uma função para cada evento de interface que ele desejar tratar

- essas funções são chamadas de **callbacks** por serem 'chamadas de volta' pelo sistema

Callbacks em OO

- modelo é ortogonal ao modelo de orientação por objetos
- é possível projetar um sistema OO que use o modelo de orientação por eventos para tratar eventos de interface, comunicações, etc
- problema: não possui o conceito de função. Como resolver então?

Callbacks em OO

- solução: usar um **objeto** que faça o papel de **callback**
- ou seja, onde registraríamos uma função, passamos a registrar um objeto

Objeto **callback**

- quando o sistema precisar executar a **callback**, ele deverá executar um determinado método do objeto
- esse método, então, fará o tratamento do evento

Callbacks em Java

- como Java é uma linguagem OO na qual não existe o conceito de função, **callbacks** devem ser implementadas através de objetos
- um objeto que implementa uma **callback** em Java é chamado de **listener**

Listeners e eventos

- os **listeners** fazem o papel das **callbacks**
- **listeners** são definidos por interfaces e podem estar aptos a tratar mais de um tipo de evento

Listeners e eventos

- quando um **listener** tem um de seus métodos chamados, ele recebe um parâmetro que descreve o evento ocorrido
- esse parâmetro é um objeto
- existem classes para modelar diferentes grupos de eventos

Listeners

- definem interfaces que representam um grupo de **callbacks**
- são registrados junto aos componentes
  - **java.awt.event.MouseListener**

```
public abstract void mouseClicked(MouseEvent e)
public abstract void mousePressed(MouseEvent e)
public abstract void mouseReleased(MouseEvent e)
public abstract void mouseEntered(MouseEvent e)
public abstract void mouseExited(MouseEvent e)
```

Registro de listeners

- mecanismo de **callbacks**
- implementação da interface
- uso de classes aninhadas

```
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      System.out.println("Botão pressionado");
    }
});
```

Eventos

- trazem informações sobre o evento ocorrido
- são passados aos **listeners (callbacks)**
  - **java.awt.event.MouseEvent**

```
public int getX()
public int getY()
public int getClickCount()
```

WindowEvent

- modela os eventos que podem ocorrer em uma janela
- essa classe declara constantes que identificam os diversos eventos

```
public static final int WINDOW_OPENED
public static final int WINDOW_CLOSING
public static final int WINDOW_CLOSED
public static final int WINDOW_ICONIFIED
public static final int WINDOW_DEICONIFIED
public static final int WINDOW_ACTIVATED
public static final int WINDOW_DEACTIVATED

public Window getWindow()
```

WindowListener

- modela a **callback** de um evento do tipo **WindowEvent**
- essa interface declara um método para cada evento do grupo

```
public abstract void windowOpened(WindowEvent e)
public abstract void windowClosing(WindowEvent e)
public abstract void windowClosed(WindowEvent e)
public abstract void windowIconified(WindowEvent e)
public abstract void windowDeiconified(WindowEvent e)
public abstract void windowActivated(WindowEvent e)
public abstract void windowDeactivated(WindowEvent e)
```

Implementando um **listener**

- para criarmos um **listener** para um evento de janela devemos criar uma classe que implemente a interface **WindowListener**
- nessa classe, codificaremos o método correspondente ao evento que desejamos tratar

Implementando um **listener**

- problema: não podemos implementar uma interface e deixar de codificar algum método
- solução: precisaremos implementar todos os sete métodos definidos

Adaptadores

- neste caso, seis implementações são vazias pois só desejamos responder a um único evento
- esta é uma situação comum
- pacote **event**: define adaptadores para todas as interfaces de **listeners** que têm mais de um método

OBS: Na verdade, os adaptadores fornecem implementações vazias para os métodos de uma interface.

Adaptadores

- são classes que implementam o **listener** e fornecem implementações vazias para todos os métodos

Exemplo

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

public class Janela {
  JButton botaoLimpa; JTextField campoTexto; JFrame janela;
  public Janela() {
    botaoLimpa = new JButton("Limpa");
    campoTexto = new JTextField(10);
    janela = new JFrame ("Exemplo de Listener");
    janela.setSize(300,100); JPanel painel = new JPanel();
    janela.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    painel.add (botaoLimpa); painel.add (campoTexto);
    janela.getContentPane().add(painel);
    botaoLimpa.addActionListener (new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        campoTexto.setText("");
      }});
    janela.setVisible(true);
  }
```

Exemplo (cont.)

```
  public static void main(String[] args) {
    new Janela();
  }
}
```

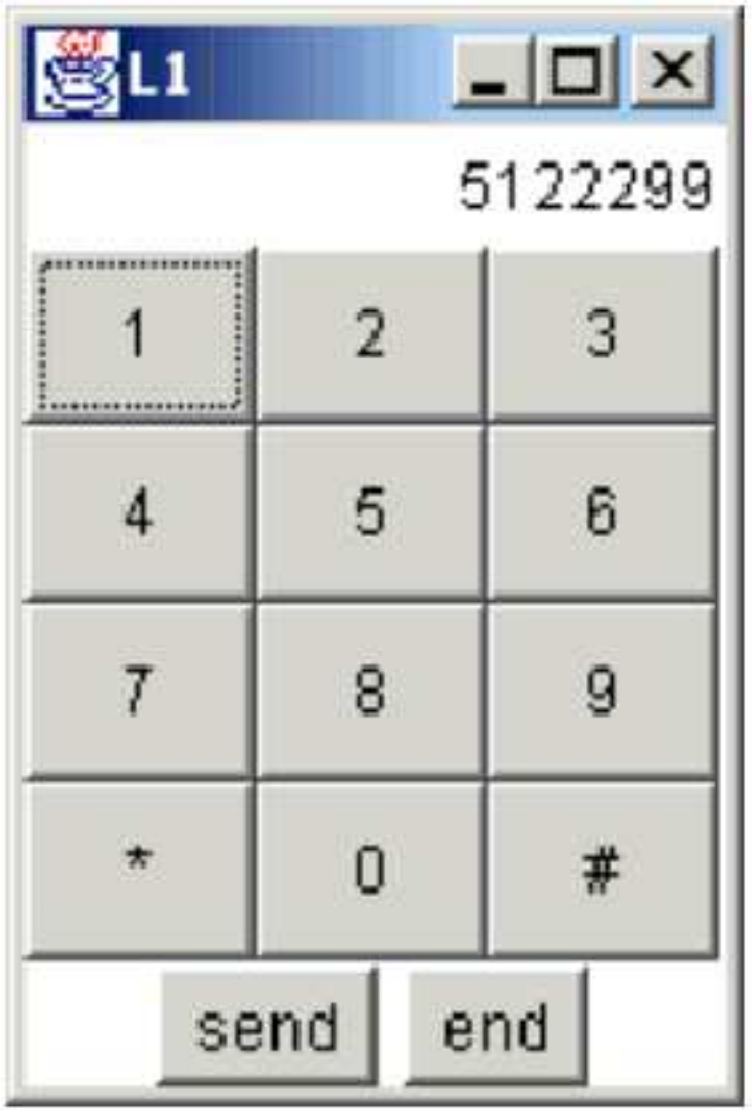
Exercício:enunciado

Usando apenas as classes JFrame, JPanel, JButton, JLabel, BorderLayout, GridLayout e FlowLayout, escreva um programa que, ao ser executado, exiba a tela ao lado.

Faça com que o número que aparece no visor seja o número digitado no teclado numérico da aplicação.

O botão send deve imprimir no console o conteúdo do visor e o botão end deve apagar o visor.

Permita que a aplicação seja terminada fechando-se a janela



Exercício: solução

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class JCelular {
 public static void main(String[] args) {
  JFrame janela = new JFrame("Celular"); // janela
  final JLabel visor = new JLabel("5122299"); // visor
  visor.setHorizontalAlignment(JLabel.RIGHT);
  JPanel numeros = new JPanel(new GridLayout(4,3));//Tecla
  String[] nomes = {"1","2","3","4","5","6","7","8","9","*","0","#"};
  // Cria o listener para as teclas do celular
  ActionListener trataTecla = new ActionListener() {
   public void actionPerformed(ActionEvent e) {
    JButton botaoClicado = (JButton)e.getSource();
    visor.setText(visor.getText()+botaoClicado.getText());
   }};
```

Exercício: solução

```
for(int i=0; i<nomes.length; i++)
((JButton)numeros.add(new
JButton(nomes[i]))).addActionListener(trataTecla);

JPanel botoes = new JPanel(); // Botoes send e end

((JButton)botoes.add(newJButton("send"))).addActionListener
(new ActionListener() {
   public void actionPerformed(ActionEvent e) {
    System.out.println(visor.getText());
   }});

((JButton)botoes.add(new
JButton("end"))).addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) {
    visor.setText(" ");
   }});
```

Exercício: solução

```
  // monta tudo
  janela.getContentPane().add(visor, BorderLayout.NORTH);
  janela.getContentPane().add(numeros, BorderLayout.CENTER);
  janela.getContentPane().add(botoes, BorderLayout.SOUTH);
  // mostra
  janela.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);
  janela.pack();
  janela.show();
 }
}
```

OBS: Na verdade, método show() está descontinuado. Use, em seu lugar, o método setVisible(true).