Manipulação de Eventos

SCC0604 - Programação Orientada a Objetos

Prof. Fernando V. Paulovich
http://www.icmc.usp.br/~paulovic
paulovic@icmc.usp.br

Instituto de Ciências Matemáticas e de Computação (ICMC)
Universidade de São Paulo (USP)

25 de julho de 2010

Sumário

Sumário

Manipulação de Eventos

- A **monitoração** do que está ocorrendo em uma interface gráfica é feita através de **eventos**
- Na manipulação de eventos, temos dois extremos: o **originador** (interface gráfica) e o **ouvinte** (quem trata) dos eventos
- Qualquer classe podem ser um ouvinte de um evento, para isso é necessário **registrar** essa classe como um **ouvinte** de uma classe originadora

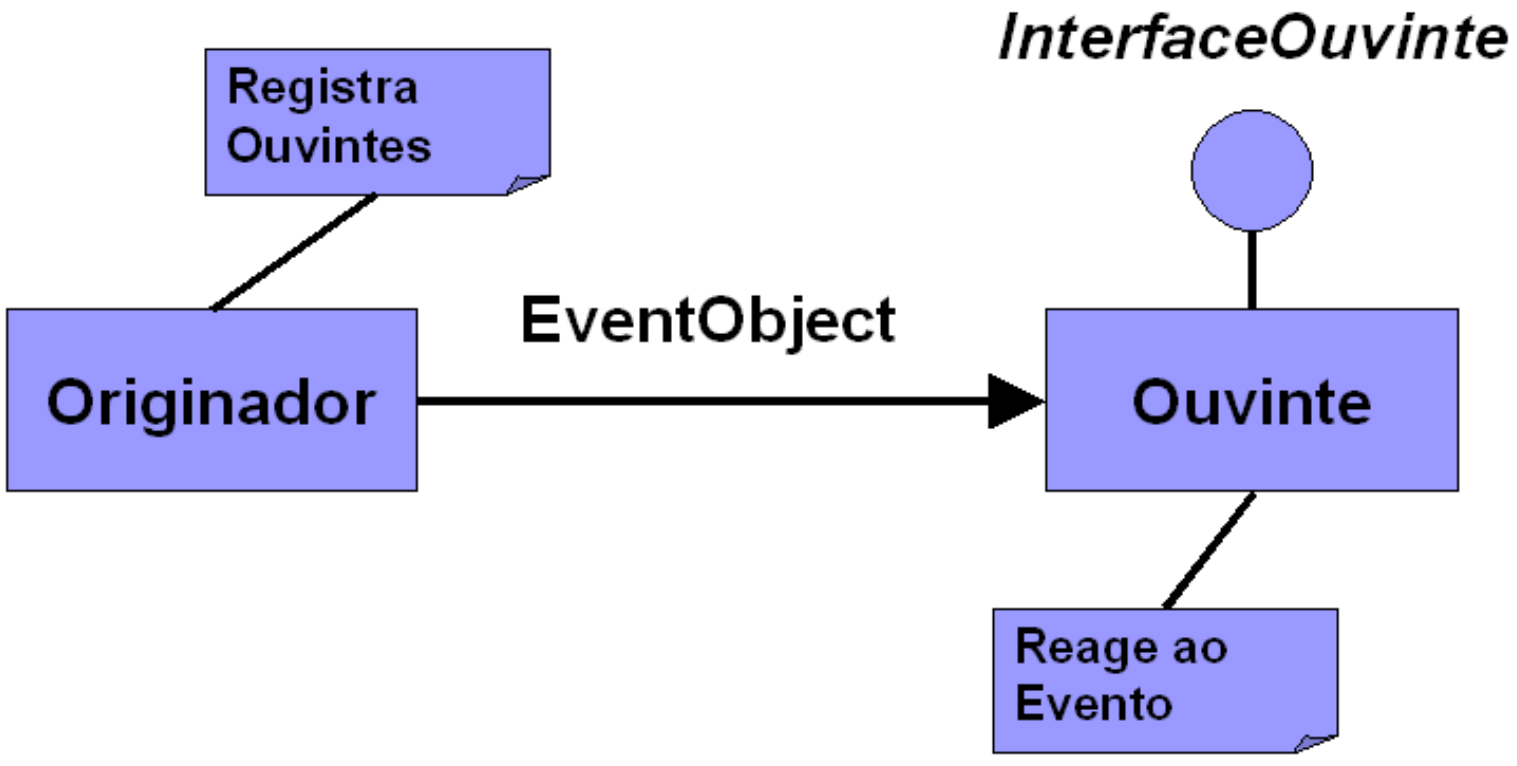
Manipulação de Eventos

- Em Java um evento é um objeto do tipo **java.util.EventObject**, ou um subtipo como **ActionEvent** ou **WindowEvent**
- Origens diferentes de eventos podem produzir eventos diferentes
  - O mouse gera eventos diferentes do teclado

Manipulação de Eventos

- Visão geral da manipulação de eventos
  - Um objeto ouvinte é uma instância de uma classe que **implementa uma interface ouvinte**
  - Uma origem de evento é um objeto que pode **registrar objetos ouvintes** e **envia** a estes os **objetos de eventos**
  - A **origem de evento envia** objetos **eventos** para **todos os ouvintes registrados** quando esse evento ocorre
  - Os **objetos ouvintes** vão então usar a informação do objeto evento recebido para **determinar sua reação ao evento**

Diagrama Esquemático



Manipulação de Eventos

- O objeto ouvinte é registrado no objeto origem com a seguinte código

```
1 objetoEventoOrigem.addEventListener(objetoOuvinte)
```

- Por exemplo:

```
1 Mypanel painel = new MyPanel();
2 JButton botao = new JButton("Limpar");
3 botao.addActionListener(painel);
```

Manipulação de Eventos

- Um código como anterior exige que a classe ao qual o objeto ouvinte pertença seja implementada pela interface apropriada (**ActionListener**)
- Para implementar a interface **ActionListener**, a classe ouvinte precisa ter um método (**actionPerformed**) que recebe o evento **ActionEvent**

Manipulação de Eventos

```
1  class MyPanel extends JPanel implements ActionListener {
2
3    public void actionPerformed(ActionEvent evt) {
4      //a reação ao clique do botão
5    }
6  }
```

Criando Botões

- Um botão é um objeto do tipo **JButton**

```
1 JButton yellowButton = new JButton("Yellow");
```

- Para se adicionar um botão a um painel, usamos o método **add()**

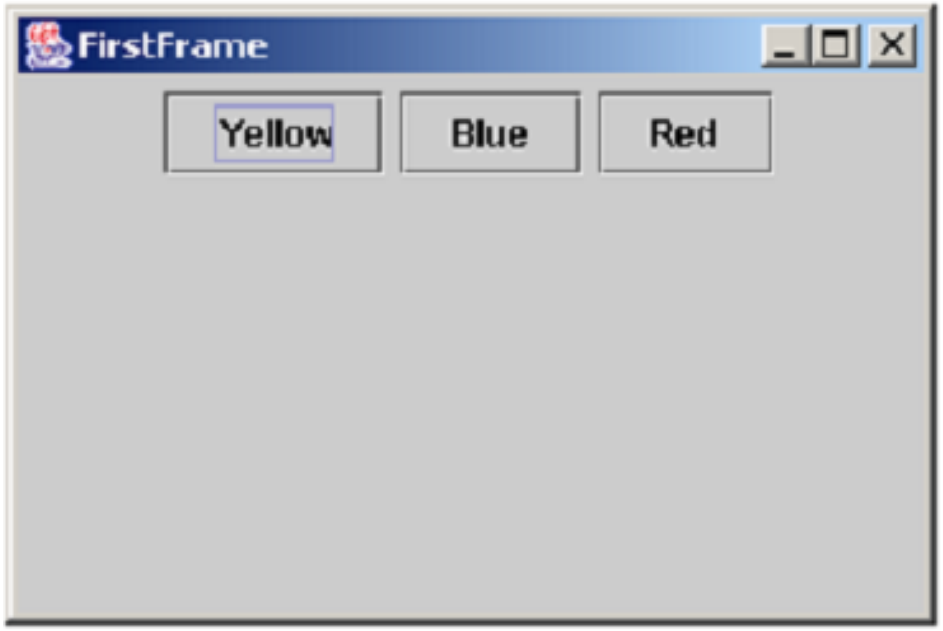
Criando a Adicionando Botões

```
1  public class MyPanel extends JPanel implements ActionListener {
2
3    private JButton yellowButton = new JButton("Yellow");
4    private JButton blueButton = new JButton("Blue");
5    private JButton redButton = new JButton("Red");
6
7    public MyPanel() {
8      this.add(yellowButton);
9      this.add(blueButton);
10     this.add(redButton);
11   }
12
13   ...
14 }
```

Criando a Adicionando Botões

```
1  public class FirstFrame extends JFrame{
2
3    public FirstFrame() {
4      setTitle("FirstFrame");
5      setSize(300, 200);
6      this.setDefaultCloseOperation(EXIT_ON_CLOSE);
7      Container contentPane = getContentPane();
8      contentPane.add(new MyPanel());
9    }
10
11   public static void main(String[] args) {
12     JFrame frame = new FirstFrame();
13     frame.setVisible(true);
14   }
15 }
```

Resultado



Tratando os Eventos

- Após adicionar os botões é necessário tratar os eventos dos mesmos
- Isto requer a implementação da interface **ActionListener** - implementação do método **actionPerformed()**
- Para verificar qual evento ocorreu, podemos usar o método **getSource()**

Capturando os Eventos

```
1  public void actionPerformed(ActionEvent evt) {
2    Object source = evt.getSource();
3
4    if(source == yellowButton) {
5      //trata evento
6    } else if(source == blueButton) {
7      //trata evento
8    } else if(source == redButton) {
9      //trata evento
10   }
11 }
```

Capturando os Eventos

- Após determinarmos que o painel contendo os botões irá tratar os eventos, devemos informar aos botões para eles enviarem esses eventos ao painel
- Para isso usamos o método **addActionListener()** junto a classe que cria os botões

Capturando os Eventos

```
1  public class MyPanel extends JPanel implements ActionListener {
2
3    private JButton yellowButton = new JButton("Yellow");
4    private JButton blueButton = new JButton("Blue");
5    private JButton redButton = new JButton("Red");
6
7    public MyPanel() {
8      this.add(yellowButton);
9      this.add(blueButton);
10     this.add(redButton);
11
12     yellowButton.addActionListener(this);
13     blueButton.addActionListener(this);
14     redButton.addActionListener(this);
15   }
16
17   ...
18 }
```

Código

```
1  class ButtonPanel extends JPanel implements ActionListener {
2
3    public void actionPerformed(ActionEvent evt) {
4      Object source = evt.getSource();
5      if(source == yellowButton) setBackground(Color.YELLOW);
6      else if(source == blueButton) setBackground(Color.BLUE);
7      else if(source == redButton)setBackground(Color.RED);
8    }
9
10   ...
11 }
```

Descobrindo quem Gerou o Evento

- Uma outra forma de saber quem gerou o evento é utilizar o método, específico da classe **ActionEvent**, chamado **getActionCommand()**
- Esse método retorna uma string associada a ação - no caso dos botões, essa string é, a priori, o rótulo dos mesmos

Tratando os Eventos

```
1  class ButtonPanel extends JPanel implements ActionListener {
2
3    public void actionPerformed(ActionEvent evt) {
4      String command = evt.getActionCommand();
5      if(command == "Yellow") setBackground(Color.yellow);
6      else if(command == "Blue") setBackground(Color.blue);
7      else if(command == "Red")setBackground(Color.red);
8    }
9
10   ...
11 }
```

Problemas com Rótulos

- Evidentemente, essa abordagem pode trazer problemas, principalmente se for necessário mudar o rótulo de um menu
- Para evitar esse problema, é possível se especificar uma string de rótulo do comando por meio do método **setActionCommand()** no momento da criação do botão:

```
1 yellowButton.setActionCommand("Amarelo");
```

Sumário

Captura de Eventos de Janelas

- Como já foi visto, quando um usuário tenta fechar uma janela, um evento **WindowEvent** é gerado pela classe **JFrame** que representa a janela
- Assim, nós precisamos ter um objeto ouvinte apropriado para tratar tal evento e adicioná-lo à lista de ouvintes

Captura de Eventos de Janelas

```
1  public class FirstFrame extends JFrame{
2
3      public FirstFrame(){
4        addWindowListener(new Terminator());
5      }
6
7      ...
8  }
```

Captura de Eventos de Janelas

- O ouvinte de janela **Terminator** precisa ser um objeto de uma classe que implemente a interface **WindowListener**
- Quando uma classe implementa a interface **WindowListener** a mesma precisa que os seguintes métodos sejam implementados
  - public void windowActivated(WindowEvent e)
  - public void windowClosed(WindowEvent e)
  - public void windowClosing(WindowEvent e)
  - public void windowOpened(WindowEvent e)
  - ...

Captura de Eventos de Janelas

- Assim, como qualquer outra classe Java que implementa uma interface, todos os métodos da interface devem ser providos
- Se estivermos interessados em somente um método (p.ex. **windowClosing()**), não somente esse método deve ser implementados, mas todos os outros, porém com corpos vazios

Captura de Eventos de Janelas

```
1  class Terminator implements WindowListener {
2    public void windowActivated(WindowEvent e) {}
3    public void windowClosed(WindowEvent e) {}
4    public void windowClosing(WindowEvent e) {System.exit(0);}
5    public void windowOpened(WindowEvent e) {}
6    public void windowIconified(WindowEvent e) {}
7    ...
8  }
```

Classes Adaptadoras

- Digitar código para métodos que não fazem nada é um trabalho não muito "atraente", assim Java oferece as classes adaptadoras, que implementam tais métodos com corpo vazio
- Dessa forma, ao invés de se implementar uma interface ouvinte pode-se estender uma classe adaptadora
- Para o controle de Janela, a classe adaptadora é chamada **WindowAdapter**

Classes Adaptadoras

```
1  class Terminator extends WindowAdapter {
2    public void windowClosing(WindowEvent e) {
3      System.exit(0);
4    }
5  }
```

Sumário

A Hierarquia de Eventos AWT

- Os eventos Java são objetos instanciados a partir de classes descendentes da classe **java.util.EventObject**
- Uma lista de tipos de eventos AWT destinados a ouvintes
  - **ActionEvent**
  - **AdjustmentEvent**
  - **ComponentEvent**
  - **ContainerEvent**
  - **FocusEvent**
  - **ItemEvent**
  - **KeyEvent**
  - **MouseEvent**
  - **TextEvent**
  - **WindowEvent**

A Hierarquia de Eventos AWT

- Existem onze interfaces ouvintes para esses eventos
  - **ActionListener**
  - **AdjustmentListener**
  - **ComponentListener**
  - **FocusListener**
  - **ItemListener**
  - **KeyListener**
  - **MouseListener**
  - **MouseMotionListener**
  - **TextListener**
  - **WindowListener**

A Hierarquia de Eventos AWT

- Dessas interfaces (especificamente aquelas que têm mais de um método), sete vêm junto com classes adaptadoras
  - **ComponentAdapter**
  - **ContainerAdapter**
  - **FocusAdapter**
  - **KeyAdapter**
  - **MouseAdapter**
  - **MouseMotionAdapter**
  - **WindowAdapter**

Sumário

Eventos Semânticos e de Baixo Nível no AWT

- O AWT faz distinção entre eventos de baixo nível e eventos semânticos
- Um evento semântico expressa o que o usuários está fazendo, como "clicando aquele botão"
- Já os eventos de baixo nível são os eventos que tornam isso possível

Eventos Semânticos e de Baixo Nível no AWT

- Há 4 classes de eventos semânticos
  - **ActionEvent**: para clique de botão, seleção de menu, etc., clique duplo em um item de uma lista, tecla <ENTER> pressionada em um campo de texto
  - **AdjustmentEvent**: para ajuste de barra de rolagem
  - **ItemEvent**: para seleção de um conjunto de caixas de seleção ou itens de uma lista
  - **TextEvent**: para a modificação de um campo de texto ou área de texto

Eventos Semânticos e de Baixo Nível no AWT

- Há 6 classes de eventos de baixo nível
  - **ComponentEvent**: redimensionamento, movimentação, exibição ou ocultação do componente
  - **KeyEvent**: tecla pressionada ou liberada
  - **MouseEvent**: botão do mouse pressionado, liberado,movido ou arrastado
  - **FocusEvent**: componente recebeu ou perdeu foco
  - **WindowEvent**: janela ativada, desativada, minimizada, restaurada ou fechada
  - **ContainerEvent**: componente adicionado ou removido

Eventos de Foco

- Na linguagem Java, um componente tem o foco se puder receber pressionamentos de teclas
- Somente um componente pode ter o foco de cada vez
- Um componente pode ganhar o foco se o usuário clicar o mouse dentro dele, ou quando o usuário usa a tecla <TAB> para trocar de componente
- A priori, os componentes são percorridos da esquerda para a direita e de cima para baixo quando o <TAB> é pressionado

Eventos de Foco

- Por fim, pode-se usar o método **requestFocus()** para mover o foco até qualquer componente visível em tempo de execução
- Um ouvinte de foco precisa implementar dois métodos: **focusGained()** e **focusLost()**. Esses métodos são acionados quando a origem do evento ganhar ou perder o foco

Eventos de Foco

- O método **getComponent()** informa o componente que recebeu ou perdeu o foco, e o método **isTemporary()** retorna true se a mudança de foco for temporária

Eventos de Foco

```
1  public class MyPanel extends JPanel {
2
3    private JButton yellowButton = new JButton("Yellow");
4    private JButton blueButton = new JButton("Blue");
5    private JButton redButton = new JButton("Red");
6
7    public MyPanel() {
8      this.add(yellowButton);
9      this.add(blueButton);
10     this.add(redButton);
11
12     yellowButton.addActionListener(new OuvinteBotao());
13     blueButton.addActionListener(new OuvinteBotao());
14     redButton.addActionListener(new OuvinteBotao());
15     ...
16   }
17 }
```

Eventos de Foco

```
1  public class MyPanel extends JPanel {
2
3    class OuvinteBotao implements ActionListener {
4      public void actionPerformed(ActionEvent evt) {
5        Object source = evt.getSource();
6        if(source == yellowButton) setBackground(Color.YELLOW);
7        else if(source == blueButton) setBackground(Color.BLUE);
8        else if(source == redButton)setBackground(Color.RED);
9      }
10   }
11
12   ...
13 }
```

Eventos de Foco

```
1  public class MyPanel extends JPanel {
2    ...
3    class OuvinteFoco extends FocusAdapter {
4      public void focusGained(FocusEvent e) {
5        Object source = e.getComponent();
6        if(source == yellowButton) setBackground(Color.YELLOW);
7        else if(source == blueButton) setBackground(Color.BLUE);
8        else if(source == redButton)setBackground(Color.RED);
9      }
10   }
11 }
```

Eventos de Foco

```
1  public class FirstFrame extends JFrame {
2
3    public FirstFrame() {
4      setTitle("FirstFrame");
5      setSize(300, 200);
6      this.setDefaultCloseOperation(EXIT_ON_CLOSE);
7      Container contentPane = getContentPane();
8      contentPane.add(new MyPanel());
9    }
10
11   public static void main(String[] args) {
12     JFrame frame = new FirstFrame();
13     frame.setVisible(true);
14   }
15 }
```

Eventos de Janela

- Como já foi visto, é possível ser notificado sempre que uma janela
  - Foi aberta
  - Foi fechada
  - Tornou-se ativa
  - Tornou-se inativa
  - Foi minimizada
  - Foi restaurada, voltando ao tamanho original
  - Está tentando ser fechada pelo usuário

Eventos de Teclado

- Quando uma tecla é pressionada, um evento **KeyEvent.KEY_PRESSED** é gerado; quando a mesma é solta, um evento **KeyEvent.KEY_RELEASE** é gerado
- Esses eventos são capturados pelos métodos **keyPressed()** e **KeyReleased()** de qualquer classe que implemente a interface **KeyListener**
- Esses métodos podem ser usados para capturar pressionamento simples de teclas
- Um terceiro método, **keyTyped()**, combina esses dois, informando os caracteres gerados pelas teclas pressionadas pelo usuário

Eventos de Teclado

- Aqui novamente vamos usar uma classe adaptadora, **KeyAdapter**, para não ser necessário implementar todos os métodos de tratamento de teclado
- Para se trabalhar com esses métodos, primeiro deve-se verificar o código da tecla com o método **getKeyCode()**
- Para identificar teclas, Java usa a seguinte nomenclatura
  - VK_A ... VK_Z
  - VK_0 ... VK_9
  - VK_COMMA, VK_ENTER, etc.

Código

```
1  public class MyPanel extends JPanel {
2    ...
3    class OuvinteTeclado extends KeyAdapter {
4
5      public void keyPressed(KeyEvent evt) {
6        if(evt.getKeyCode() == KeyEvent.VK_ENTER) {
7          ...
8        }
9      }
10   }
11 }
```

Eventos de Teclado

- Para saber se as teclas shift, alt, ctrl ou meta estão pressionadas, é possível empregar os seguintes métodos
  - **isShhiftDown()**
  - **isAltDown()**
  - **isCrtlDown()**
  - **isMetaDown()**

Eventos de Mouse

- Não é necessário processar explicitamente os eventos do mouse caso só seja necessário verificar se um usuário clicou em um botão
- Essas operações são processadas internamente e convertidas em eventos semânticos apropriados
- Por exemplo, pode-se reagir a esses eventos com um método **actionPerformed()**

Eventos de Mouse

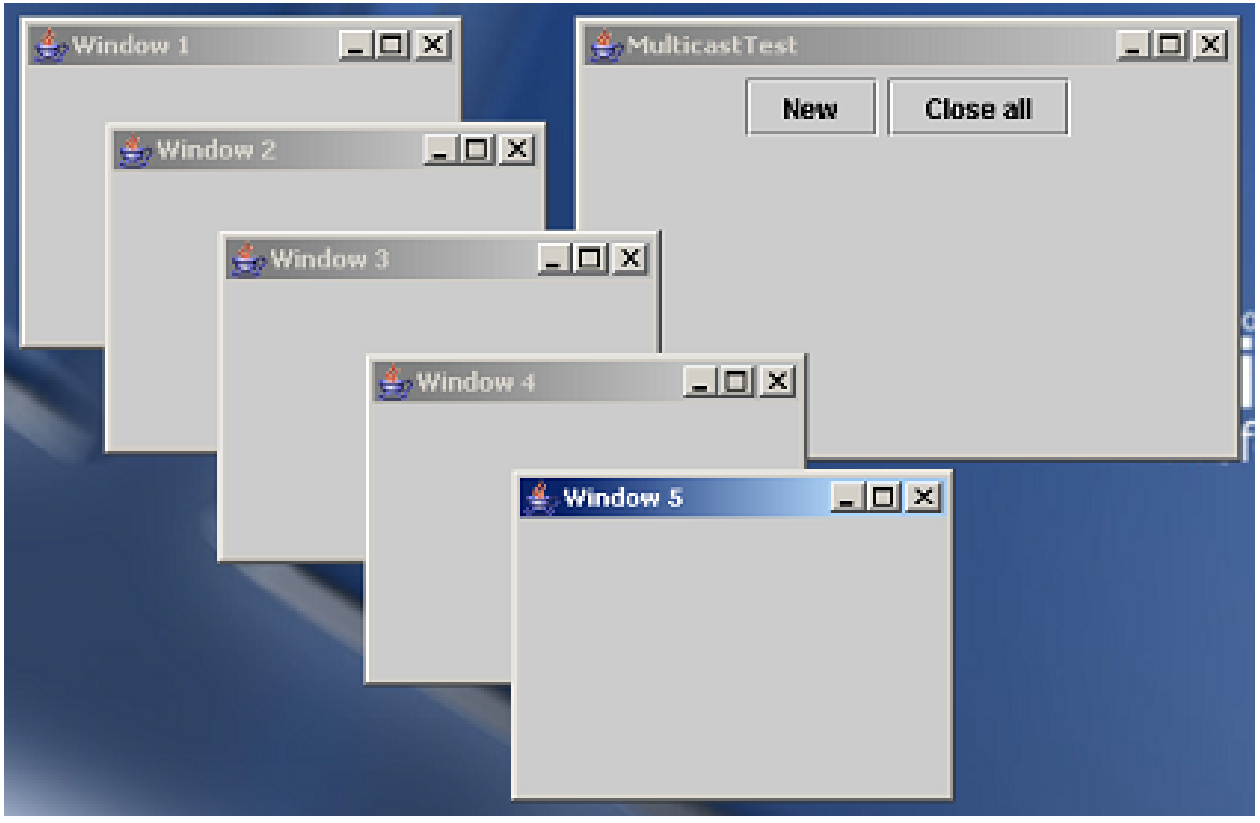
- Contudo é possível capturar os eventos de movimentação de um mouse
- Quando um cliente clica em um botão de um mouse três métodos ouvintes são chamados: **mousePressed()**, **mouseReleased()** e **mouseClicked()**
- Para maiores informações sobre os eventos de mouse: seção Eventos de Mouse, Core Java 2 Volume I (pág. 305-313)

Sumário

Multidifusão (Multicast)

- Anteriormente várias origens de eventos informavam o mesmo ouvinte de eventos
- Agora vamos fazer o oposto: o mesmo evento será enviado para mais de um objeto ouvinte
  - Isto é chamado de multidifusão
- A multidifusão é útil quando um evento desperta o interesse potencial de muitas partes
- Para isso basta adicionar vários ouvintes a uma origem de eventos

Multidifusão (Multicast)



Multidifusão (Multicast)

```
1  public class SimpleFrame extends JFrame implements ActionListener {
2
3    public void actionPerformed(ActionEvent e) {
4      dispose();
5    }
6  }
```

Multidifusão (Multicast)

```
1  public class MulticastPanel extends JPanel implements ActionListener {
2    private int counter=0;
3    private JButton closeAllButton= new JButton("Close all");
4    private JButton newButton= new JButton("New");
5
6    public MulticastPanel() {
7      this.add(newButton);
8      this.add(closeAllButton);
9      newButton.addActionListener(this);
10   }
11
12   public void actionPerformed(ActionEvent e) {
13     SimpleFrame f = new SimpleFrame();
14     counter++;
15     f.setTitle("Window " + this.counter);
16     f.setBounds(30*counter, 30*counter, 200, 150);
17     f.show();
18     closeAllButton.addActionListener(f);
19   }
20 }
```

Multidifusão (Multicast)

```
1  public class MulticastFrame extends JFrame {
2
3    public MulticastFrame() {
4      this.setTitle("MulticastTest");
5      this.setSize(300,200);
6      this.setDefaultCloseOperation(EXIT_ON_CLOSE);
7
8      Container content = this.getContentPane();
9      content.add(new MulticastPanel());
10   }
11 }
```

Multidifusão (Multicast)

```
1  public class MulticastTest {
2
3    public static void main(String[] args) {
4    JFrame frame = new MulticastFrame();
5      frame.show();
6    }
7  }
```

Sumário

Alterando Estilo Visual

- Existem três possíveis estilos visuais de apresentação em Java
  - Motif: **com.sun.java.swing.plaf.motif.MotifLookAndFeel**
  - Windows: **com.sun.java.swing.plaf.windows.WindowsLookAndFeel**
  - Metal: **javax.swing.plaf.metal.MetalLookAndFeel**
- O estilo windows só esta presente para o sistema operacional Microsoft Windows, e por padrão o estilo usado é o metal

Alterando Estilo Visual

```
1  public class FirstFrame extends JFrame{
2
3    public FirstFrame() {
4      setTitle("FirstFrame");
5      setSize(300, 200);
6      this.setDefaultCloseOperation(EXIT_ON_CLOSE);
7      Container contentPane = getContentPane();
8      contentPane.add(new MyPanel());
9
10     try {
11       UIManager.setLookAndFeel("com.sun.java.swing.plaf.motif.↩
             MotifLookAndFeel");
12       SwingUtilities.updateComponentTreeUI(contentPane);
13     } catch(Exception e) { }
14
15   }
16
17   ...
18 }
```