SCC0504 – Programação Orientada a Objetos

Eventos

Luiz Eduardo Virgilio da Silva
ICMC, USP

Material baseado nos slides do professor:
Fernando Paulovich (ICMC/USP)

Sumário

- Eventos
- Manipulação de Eventos
  - Objeto onde eventos são gerados
  - Objeto ouvinte de eventos (*listeners*)
- Tipos de Eventos
- Multicast
- SwingWorker

Introdução

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

Introdução

- 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
    - Nele, os eventos são tratados
  - Objetos capazes de gerar eventos (objeto origem) devem registrar os ouvintes que irão receber os eventos
  - Objeto onde os eventos foram gerados envia objetos eventos para todos os ouvintes registrados quando esse evento ocorre
  - Os objetos ouvintes podem então usar a informação do objeto evento recebido para determinar sua reação ao evento

Manipulação de Eventos

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

```
objetoOrigem.addEventListener(objetoOuvinte);
```

- Exemplo

```
MyPanel panel = new MyPanel();  // implemented as listener
JButton button = new JButton("Clean");
button.addActionListener(panel);
```

Manipulação de Eventos

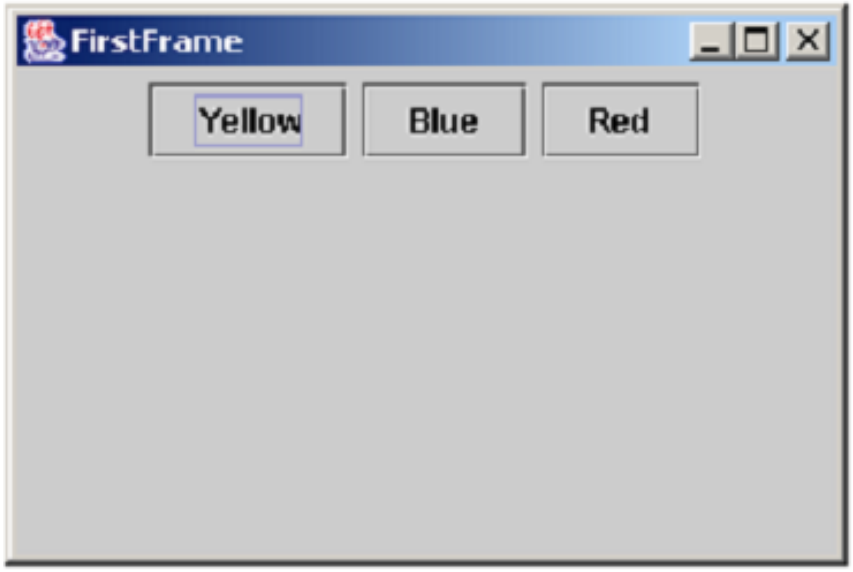
- O código anterior exige que a classe **MyPanel** implemente a interface ouvinte apropriada (ActionListener)
- Ao implementar a interface ActionListener, a classe ouvinte precisa definir um método (**actionPerformed**) que recebe como parâmetro um evento (**ActionEvent**)

Manipulação de Eventos

```
class MyPanel extends JPanel implements ActionListener {

    public void actionPerformed(ActionEvent evt) {
        // event reaction
    }
}
```

Manipulação de Eventos



Manipulação de Eventos

- No caso anterior, é preciso registrar pelo menos um ouvinte para cada botão, para que o evento gerado em cada um possa ser “ouvido”
- A classe que trata os eventos (*listener*) precisa implementar a interface ActionListener
  - actionPerformed(ActionEvent)
- Porém, podemos registrar o mesmo objeto *listener* para todos os botões
- Neste caso, para identificar a fonte do evento usaremos o método **getSource()** do objeto evento

Manipulação de Eventos

```
class MyPanel extends JPanel implements ActionListener {

    ...

    public void actionPerformed(ActionEvent evt) {
        Object source = evt.getSource();

        if(source == yellowButton) {
            // event handling
        } else if(source == blueButton) {
            // event handling
        } else if(source == redButton) {
            // event handling
        }
    }
}
```

Manipulação de Eventos

- Após determinarmos que o painel (**MyPanel**) contendo os botões irá tratar os eventos, devemos registrá-lo como ouvinte de cada botão
  - Dessa forma, eventos do botão serão enviados à instância de **MyPanel** registrada

- Para isso usamos o método **addActionListener**

Manipulação de Eventos

```
public class MyPanel extends JPanel implements ActionListener {

    private JButton yellowButton = new JButton("Yellow");
    private JButton blueButton = new JButton("Blue");
    private JButton redButton = new JButton("Red");

    public MyPanel() {
        this.add(yellowButton);
        this.add(blueButton);
        this.add(redButton);

        yellowButton.addActionListener(this);
        blueButton.addActionListener(this);
        redButton.addActionListener(this);
    }

...
}
```

Manipulação de Eventos

```
public class MyPanel extends JPanel implements ActionListener {

    private JButton yellowButton = new JButton("Yellow");
    private JButton blueButton = new JButton("Blue");
    private JButton redButton = new JButton("Red");

    public MyPanel() {
        this.add(yellowButton);
        this.add(blueButton);
        this.add(redButton);

        yellowButton.addActionListener(this);
        blueButton.addActionListener(this);
        redButton.addActionListener(this);
    }

    ...
}
```

Manipulação de Eventos

- Exemplo

```
class ButtonPanel extends JPanel implements ActionListener {
    ...

    public void actionPerformed(ActionEvent evt) {
        Object source = evt.getSource();

        if(source == yellowButton)
            setBackground(Color.YELLOW);
        else if(source == blueButton)
            setBackground(Color.BLUE);
        else if(source == redButton)
            setBackground(Color.RED);
    }

}
```

Manipulação de Eventos

- 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

Manipulação de Eventos

```
class ButtonPanel extends JPanel implements ActionListener {

    public void actionPerformed(ActionEvent evt) {
        String command = evt.getActionCommand();

        if(command.equals("Yellow"))
            setBackground(Color.YELLOW);
        else if(command.equals("Blue"))
            setBackground(Color.BLUE);
        else if(command.equals("Red"))
            setBackground(Color.RED);
}

...
}
```

Manipulação de Eventos

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

```
yellowButton.setActionCommand(“Yellow”);
```

Eventos de Janelas

- Os eventos causando por alterações em uma janela (JFrame) geram um objeto do tipo WindowEvent
  - Fechar, maximizar, minimizar, ativar, ...
- Se quisermos, por exemplo, executar alguma ação quando um usuário tentar fechar uma janela, precisamos ter um objeto ouvinte apropriado
  - Criamos uma classe ouvinte de eventos de janela ou utilizamos uma já existente
    - Nela está implementada a ação a ser executada quando o evento for gerado
  - Adicionamos o ouvinte de eventos de janela ao JFrame

Eventos de Janelas

```
public class Terminator implements WindowListener {

   // Methods to be implemented
}
```

```
public class FirstFrame extends JFrame {

    public FirstFrame(){
        addWindowListener(new Terminator());
    }

    ...
}
```

Eventos de Janelas

- A classe **Terminator** precisa ser um objeto do tipo WindowListener (interface)
- Quando uma classe implementa a interface WindowListener, a classe 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)
  - ...

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 implementado, mas todos os outros
  - Outros deverão ter corpos vazios

Eventos de Janelas

```
class Terminator implements WindowListener {
    public void windowActivated(WindowEvent e) {}
    public void windowClosed(WindowEvent e) {}
    public void windowClosing(WindowEvent e) {
        System.exit(0);
    }
    public void windowOpened(WindowEvent e) {}
    public void windowIconified(WindowEvent e) {}
    ...
}
```

Classes Adaptadoras

- Implementar métodos que não fazem nada deixa o código poluído
- Java oferece as classes adaptadoras, que implementam as interfaces *listeners* com muitos métodos
  - Todos os métodos com corpo vazio
- Dessa forma, ao invés de se implementar uma interface ouvinte pode-se estender uma classe adaptadora
- Para o controle de janelas, a classe adaptadora é chamada WindowAdapter

Classes Adaptadoras

```
class Terminator extends WindowAdapter {
    public void windowClosing(WindowEvent e) {
        System.exit(0);
    }
}
```

```
class Terminator implements WindowListener {
    public void windowActivated(WindowEvent e) {}
    public void windowClosed(WindowEvent e) {}
    public void windowClosing(WindowEvent e) {
        System.exit(0);
    }
    public void windowOpened(WindowEvent e) {}
    public void windowIconified(WindowEvent e) {}
    ...
}
```

Classes Anônimas

- Uma instância de **Terminator** foi registrada como ouvinte de eventos de janela de **FirstFrame**

```
public class FirstFrame extends JFrame {

    public FirstFrame(){
        addWindowListener(new Terminator());
    }

    ...
}
```

Classes Anônimas

- Se não quisermos definir uma classe separada para tratar o evento, podemos criar uma classe anônima

```
public class FirstFrame extends JFrame {

    public FirstFrame(){
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
    }

...
}
```

Hierarquia de Eventos

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

Hierarquia de Eventos

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

Hierarquia de Eventos

- Dessas interfaces (especicamente aquelas que têm mais de um método), sete possuem classes adaptadoras
  - ComponentAdapter
  - ContainerAdapter
  - FocusAdapter
  - KeyAdapter
  - MouseAdapter
  - MouseMotionAdapter
  - WindowAdapter

Hierarquia de Eventos

- O AWT faz distinção entre eventos **de baixo nível** e eventos **semânticos**
- Os eventos de baixo nível representam entradas de baixo nível
  - Por exemplo, as entradas diretas do usuário (mouse, teclado)
- Um evento semântico expressa o que o usuário está fazendo
  - Acessar um menu, acionar um botão, alterar um texto

Hierarquia de Eventos

- 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

Hierarquia de Eventos

- 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

Hierarquia de Eventos

- Em geral, é preferível utilizar eventos semânticos sempre que possível
  - De quantas maneira podemos acionar um botão, além do clique do mouse?
  - O mesmo vale para outros compenentes
    - Inserir dados em uma caixa de texto
    - Alternar opção em um ComboBox

- Há eventos que não dependem do usuário
  - Uma tabela pode disparar eventos sempre que receber dados de uma banco de dados

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

- 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

```
public class MyPanel extends JPanel {

    private JButton yellowButton = new JButton("Yellow");
    private JButton blueButton = new JButton("Blue");
    private JButton redButton = new JButton("Red");

    public MyPanel() {
        this.add(yellowButton);
        this.add(blueButton);
        this.add(redButton);

        yellowButton.addFocusListener(new FocusListener());
        blueButton.addFocusListener(new FocusListener());
        redButton.addFocusListener(new FocusListener());
    }

    ...
}
```

Eventos de Foco

```
public class MyPanel extends JPanel {
    ...

    class FocusListener extends FocusAdapter {
        public void focusGained(FocusEvent e) {
            Object source = e.getComponent();
            if(source == yellowButton)
                setBackground(Color.YELLOW);
            else if(source == blueButton)
                setBackground(Color.BLUE);
            else if(source == redButton)
                setBackground(Color.RED);
        }
    }
}
```

Eventos de Foco

```
public class FirstFrame extends JFrame {

    public FirstFrame() {
        setTitle("FirstFrame");
        setSize(300, 200);
        this.setDefaultCloseOperation(EXIT_ON_CLOSE);
        Container contentPane = getContentPane();
        contentPane.add(new MyPanel());
    }

    public static void main(String[] args) {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                new FirstFrame().setVisible(true);
            }
        });
    }
}
```

Eventos de Foco

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

- Vamos usar uma classe adaptadora, KeyAdapter, para não ser necessário implementar todos os métodos de tratamento de teclado
- Para se tratar os eventos do teclado, primeiro podemos identificar o código da tecla
  - Método **getKeyCode** do objeto evento
- Para identicar teclas, Java usa a seguinte nomenclatura (constantes da classe KeyEvent)
  - VK_A ... VK_Z
  - VK_0 ... VK_9
  - VK_COMMA, VK_ENTER, etc.

Eventos de Teclado

```
public class MyPanel extends JPanel {
    ...

    class KeyboardListener extends KeyAdapter {

        public void keyPressed(KeyEvent evt) {
            if(evt.getKeyCode() == KeyEvent.VK_ENTER) {
                // key event handler
            }
        }
    }
}
```

Eventos de Teclado

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

Eventos de Mouse

- Em geral, não é necessário processar explicitamente os eventos do mouse
  - Se o usuário clicar em um botão, não esperaremos um evento de mouse
- Essas operações são processadas internamente e convertidas em eventos semânticos apropriados
- Por exemplo, pode-se reagir a esses eventos com um método **actionPerformed**
  - Serve para outras fontes de acionamento do botão

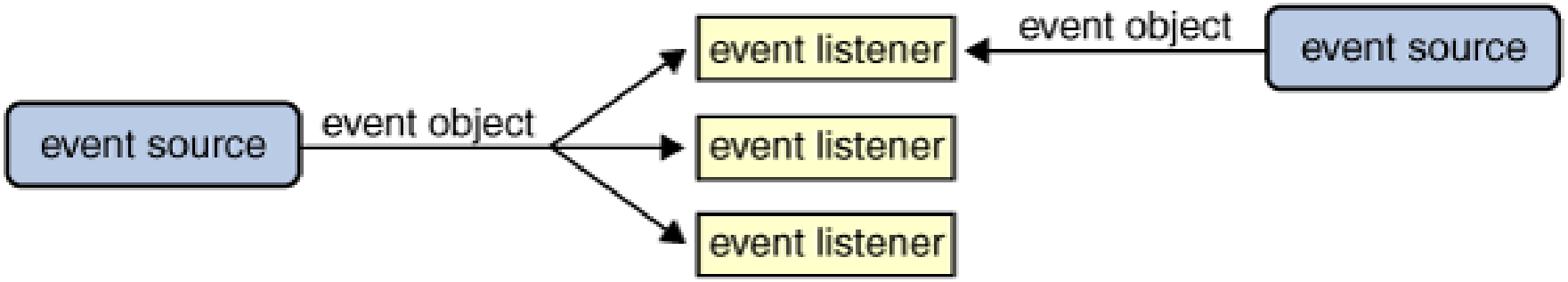
Eventos de Mouse

- Contudo, se necessário, é possível capturar os eventos de movimentação de um mouse
  - Por exemplo, se precisamos saber a posição exata onde o usuário clicou em um componente
- Quando o usuário clica (e solta) um botão do mouse, três métodos ouvintes são chamados
  - mousePressed, mouseReleased e mouseClicked
  - Similar aos eventos do teclado
- O objeto evento (MouseEvent) contem informações importantes
  - Número de cliques, qual botão, posição, etc.

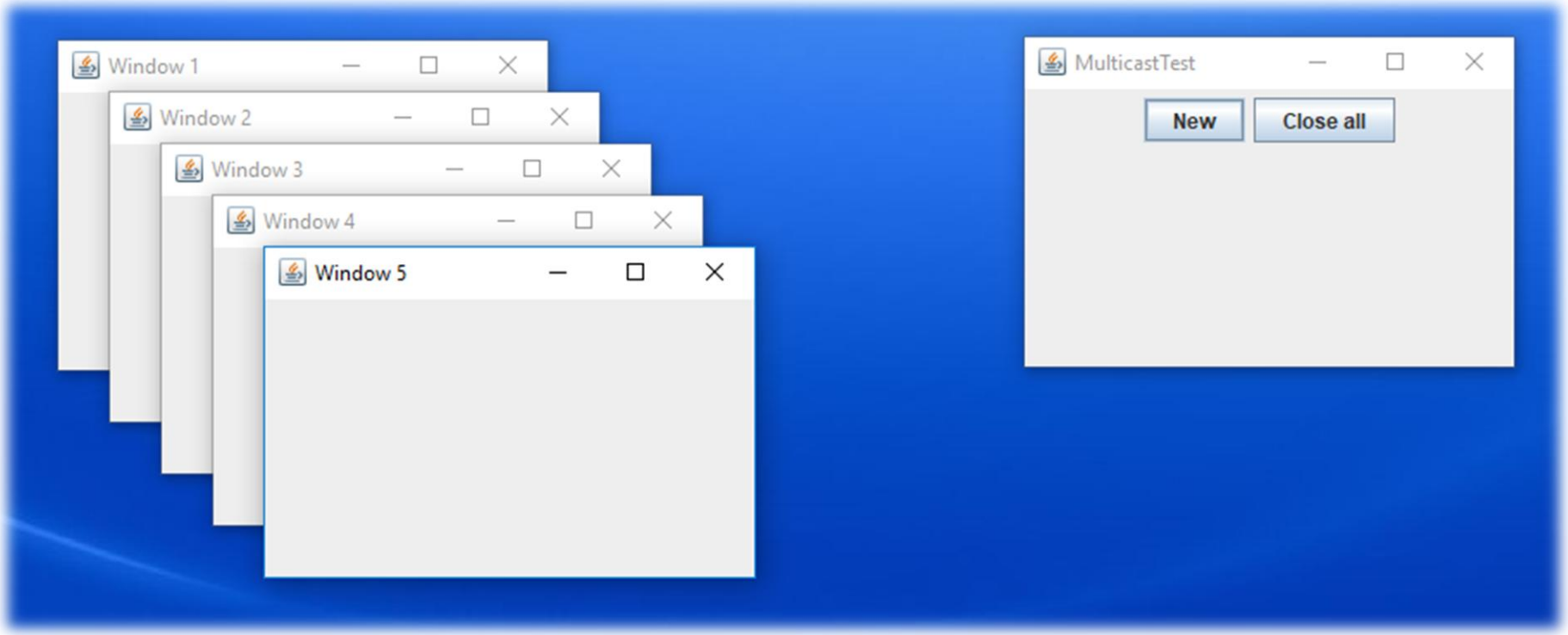
Multicast

- Nos exemplos anteriores, vários eventos (de vários botões) transmitiam seu objeto evento para o mesmo ouvinte de eventos
- Agora vamos fazer o oposto: o mesmo evento será enviado para mais de um ouvinte
  - Isto é chamado de *multicast*
- 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

Multicast


event source
event object
event listener
event listener
event listener
event object
event source

Multicast


Window 1
Window 2
Window 3
Window 4
Window 5
MulticastTest
New
Close all

Multicast

```
class SimpleFrame extends JFrame implements ActionListener {

    public void actionPerformed(ActionEvent e) {
        dispose();
    }
}
```

Multicast

```
class MulticastPanel extends JPanel implements ActionListener {
    private int counter=0;
    private JButton closeAllButton= new JButton("Close all");
    private JButton newButton= new JButton("New");

    public MulticastPanel() {
        this.add(newButton);
        this.add(closeAllButton);
        newButton.addActionListener(this);
    }

    public void actionPerformed(ActionEvent e) {
        SimpleFrame f = new SimpleFrame();
        counter++;
        f.setTitle("Window " + this.counter);
        f.setBounds(30*counter, 30*counter, 200, 150);
        f.setVisible(true);
        closeAllButton.addActionListener(f);
    }
}
```

Multicast

```
public class MulticastFrame extends JFrame {

    public MulticastFrame() {
        this.setTitle("MulticastTest");
        this.setSize(300,200);
        this.setDefaultCloseOperation(EXIT_ON_CLOSE);
        Container content = this.getContentPane();
        content.add(new MulticastPanel());
    }

    public static void main(String[] args) {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                new MulticastFrame().setVisible(true);
            }
        });
    }
}
```

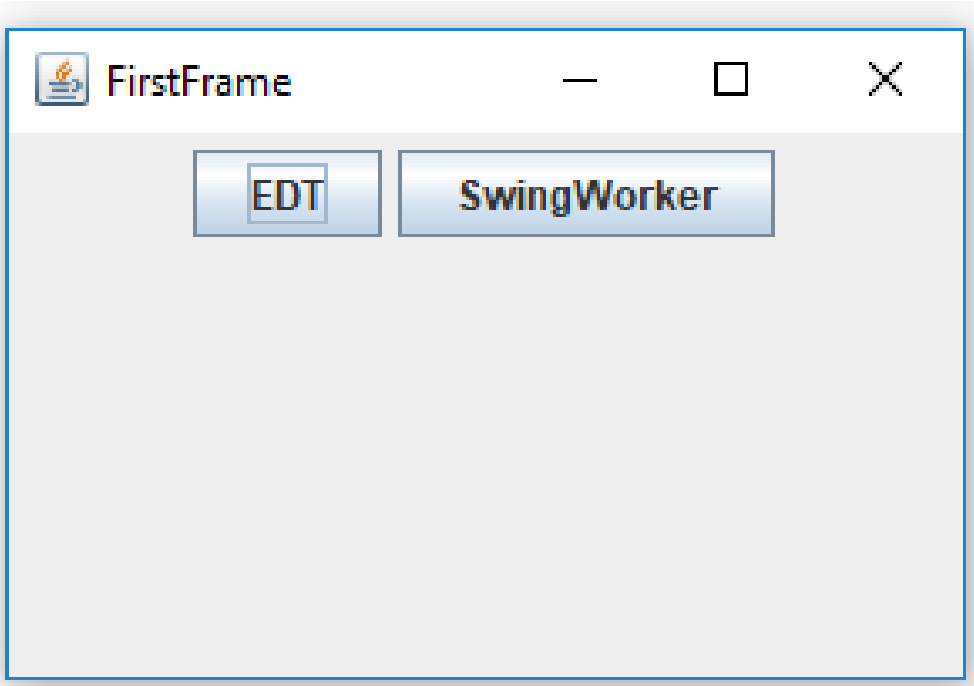
SwingWorker

- Todos os métodos registrados para tratar eventos são executados na mesma Thread que controla os componentes gráficos
  - *Event Dispatch Thread*
- Se o processamento de um evento for muito demorado, o aplicativo ficará não responsivo
  - A GUI congela
- Em caso de eventos que requerem processamento mais demorado, uma nova Thread deve ser utilizada
  - SwingWorker

SwingWorker

- SwingWorker é uma classe abstrata
  - Definir uma nova classe que herda SwingWorker
  - Criar uma classe anônima (casos mais simples)
- Métodos a serem definidos
  - <T> doInBackground()
    - Processamento a ser feito na Thread de trabalho
  - void done()
    - Chamado após o processamento terminar
    - Executado na *Event Dispatch Thread*
    - Opcional
- O método **get()** deve ser chamado para obter o retorno de **doInBackground**

SwingWorker



SwingWorker

```
public class FirstFrame extends JFrame {

    public FirstFrame() {
        setTitle("FirstFrame");
        setSize(300, 200);
        this.setDefaultCloseOperation(EXIT_ON_CLOSE);
        Container contentPane = getContentPane();
        contentPane.add(new MyWorkPanel());
    }

    public static void main(String[] args) {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                new FirstFrame().setVisible(true);
            }
        });
    }
}
```

SwingWorker

```
public class MyWorkPanel extends JPanel {
    private JButton edtButton = new JButton("EDT");
    private JButton swButton = new JButton("SwingWorker");
    private int sleepTime = 5000;

    public MyWorkPanel() {
        this.add(edtButton);
        this.add(swButton);

        edtButton.addActionListener(new ButtonListener());
        swButton.addActionListener(new ButtonListener());
    }

    class ButtonListener implements ActionListener {
        public void actionPerformed(ActionEvent evt) {
            Object source = evt.getSource();


            ...
```

```
            ...

            if(source == edtButton) {
                setBackground(Color.YELLOW);

                try {
                    Thread.sleep(sleepTime);
                    JOptionPane.showMessageDialog(null,
                          “Finished. Sleep time: "+sleepTime,
                          "Event Dispatch Example",
                          JOptionPane.INFORMATION_MESSAGE);
                } catch (InterruptedException ex) {
                    ex.printStackTrace();
                }
            } else if(source == swButton) {
                setBackground(Color.BLUE);

                ButtonWorker bw = new ButtonWorker(sleepTime);
                bw.execute();
            }
        }
    }
}
```

SwingWorker

```
public class ButtonWorker extends SwingWorker<Integer,Void> {
    private int sleepTime;

    public ButtonWorker(int sleepTime) {
        this.sleepTime = sleepTime;
    }

    @Override
    protected Integer doInBackground() throws Exception {
        try {
            Thread.sleep(sleepTime);
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }

        return sleepTime;
    }

    ...
```

SwingWorker

```
    ...

    @Override
    protected void done() {
        int result;
        try {
            result = get();
            JOptionPane.showMessageDialog(null,
                        “Finished. Sleep time: "+result,
                        "Swing Worker Example",
                        JOptionPane.INFORMATION_MESSAGE);
        } catch (InterruptedException | ExecutionException ex) {
            ex.printStackTrace();
        }
    }

}
```

Resumo

- Eventos
- Manipulação de Eventos
  - Objeto onde eventos são gerados
  - Objeto ouvinte de eventos (*listeners*)
- Tipos de Eventos
- Multicast
- SwingWorker

Dúvidas?