Tratamento de Eventos

Prof. Marcelo Cohen

1. Visão Geral

- Para implementar a funcionalidade de uma interface gráfica, pode-se fazer uso de uma repetição sem fim:
  - Faz algum processamento
  - Verifica se o mouse foi movido, se algum botão ou alguma tecla foi pressionada.
  - Realiza alguma ação se necessário.
  - Repete o ciclo
- Esse tipo de programação é considerado extremamente ineficiente, pois é necessário explicitamente **"ler"** os periféricos para tratar as ações do usuário
- Dessa forma, o programa perde muito tempo fazendo essas leituras
- Em um sistema multitarefa, esse tipo de implementação seria inviável, pois os demais processos ficariam muito tempo parados

2. Modelo de Eventos em Java

- A interface gráfica em Java é dita orientada a eventos:
  - Cada vez que um usuário clica em um botão, seleciona um item em uma lista, ou pressiona uma tecla, o sistema operacional gera um **evento**
  - Se uma aplicação está interessada em um evento específico (por exemplo, clique em um botão), deve solicitar ao sistema para **"escutar"** o evento
  - Se a aplicação não está interessada, seu processamento continua de forma normal. É importante observar que a aplicação não espera pela ocorrência de eventos: isso é controlado pelo sistema
- Para que um componente ou *container* possa "escutar" eventos, é preciso instalar um *listener*
- ***Listeners*** **são classes criadas especificamente para o tratamento de eventos**

- Há dois tipos de eventos: baixo nível e semânticos.
- Eventos de baixo nível incluem:
  - eventos de *container* (inserção ou remoção de componente)
  - eventos de foco (mouse entrou ou saiu de um componente)
  - eventos de entrada: teclado e mouse
  - eventos de janela: *open*, *close*, *resize*, *minimize*, etc
- Eventos semânticos incluem:
  - eventos de ação: notificam a ação de um componente específico (ex: clique em um botão)
  - eventos de ajuste: movimento de um *scrollbar,* por exemplo
  - eventos de item: seleção de um elemento em uma lista, *checkbox*, etc
  - eventos de texto: alteração do texto em um JTextArea, JTextField, etc

- Como usar os *listeners* ?

- *Listeners* são implementados através de interfaces
- Uma **interface** define um conjunto de métodos que uma classe deve implementar mas não define como esses métodos devem ser implementados
- Ou seja, é um modelo de como escrever esses métodos
- **Não confundir com classes abstratas: as interfaces não servem para criar novas classes, mas sim para implementar funcionalidades diversas!**
- Uma classe **implementa** uma ou mais interfaces
- De certa forma, substitui a herança múltipla em Java
- Exemplo: *MouseListener* - interface para eventos de mouse

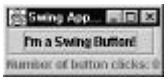
- Exemplo: *MouseListener* - interface para eventos de mouse
  - **mouseClicked(MouseEvent e)** - chamado quando o botão do mouse é clicado (e solto) sobre um componente
  - **mousePressed(MouseEvent e)** - chamado quando o botão do mouse é clicado sobre um componente
  - **mouseReleased(MouseEvent e)** - chamado quando o botão do mouse é solto sobre um componente
  - **mouseEntered(MouseEvent e)** - chamado quando o mouse "entra" na área de um componente
  - **mouseExited(MouseEvent e)** - chamado quando o mouse deixa a área de um componente
- **MouseEvent** é uma classe que representa eventos de mouse
- Cada *listener* utiliza uma classe específica para representar o evento, de acordo com o tipo que será gerado

- Exemplo de utilização: cada clique em um botão deverá incrementar um contador na tela


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

class TestaEventos extends JFrame implements ActionListener
{
  JButton but;
  JLabel texto;
  JPanel painel;
  int cont;

  public TestaEventos()
  {
   ...


  }
}
```

Para usar eventos

Eventos de botão




```
 public TestaEventos() // cria um frame com tudo dentro
 {
   super("Swing Application");
   cont = 0; // zera contador
   painel = new JPanel();
   painel.setLayout(new BoxLayout(painel,BoxLayout.Y_AXIS));
   painel.add(but);
   painel.add(texto);

   // Instala o listener para ações no botão
   but.addActionListener(this);

   getContentPane().add(painel);
 }

 // Método exigido pela interface ActionListener
 public void actionPerformed(ActionEvent e)
 {
    cont++;
    texto.setText("Number of button clicks: "+cont);
 }
...
```

- Classe ActionEvent:
  - método getActionCommand() - retorna o rótulo do botão (String), por default
  - método getSource() - retorna uma referência ao Component que gerou o evento (no caso, um JButton)
- Pode-se usar esses métodos para gerenciar eventos de mais do que um componente, com o mesmo *listener*
- Exemplo: dois botões

```
class TestaEventos2 extends JFrame implements ActionListener
{
  JButton but1, but2;

  public TestaEventos()
  {
    ...
    but1.addActionListener(this);
    but2.addActionListener(this);
    ...
  }

  public void actionPerformed(ActionEvent e)
  {
    if(e.getSource() == but1) { // botão 1 ... }
    else if(e.getSource() == but2) { // botão 2 ... }
  }
```

3. Classes *Adapter*

- Algumas interfaces (listeners) são muito extensas
  - **`MouseListener`** e **`MouseMotionListener`** têm **sete** métodos
    - E se quisermos usar apenas um ???
    - Somos obrigados a definir **todos os métodos** da interface, o que obviamente não é muito prático
- Solução: classes *adapter*
  - Implementam uma interface
    - Definem todos os métodos, com o **corpo vazio** (sem código)
  - Basta então extendermos uma classe *adapter*!
    - Sobrescrevemos os métodos que desejamos utilizar

- Classes *adapter* e listeners equivalentes

`ComponentAdapter`	`ComponentListener`
`ContainerAdapter`	`ContainerListener`
`FocusAdapter`	`FocusListener`
`KeyAdapter`	`KeyListener`
`MouseAdapter`	`MouseListener`
`MouseMotionAdapter`	`MouseMotionListener`
`WindowAdapter`	`WindowListener`

- **Não há "ActionAdapter", pois na interface *ActionListener* só há um método: *actionPerformed(..)***
- Uma vez criada a classe que implementa a interface ou define uma subclasse de *adapter*, esta precisa ser instanciada e o objeto registrado como *listener*
- Estas classes criadas para tratamento de eventos podem ser **anônimas**, ou seja, serem declaradas como ***inner classes*** (classes internas)

- Exemplo 1: usando uma classe interna anônima para tratar o evento gerado quando se pressiona um botão

```
class TestaEventos3 extends JFrame
{
  JButton but1, but2;

  public TestaEventos3()
  {
    ...
    but1.addActionListener(
       new ActionListener() {
          public void actionPerformed(ActionEvent e)
          { System.out.println("Botão 1"); ... }
       });
  ...
  }
}
```

- Observe que se forem utilizadas classes internas anônimas, não será possível tratar múltiplos componentes
- Para cada componente deverá ser escrita uma classe interna anônima
- Útil em alguns casos, especialmente quando o tratamento do evento for simples e rápido
- Não aconselhável para código de tratamento que seja muito longo

- Exemplo 2: usando uma classe interna anônima para tratar eventos de clique do mouse

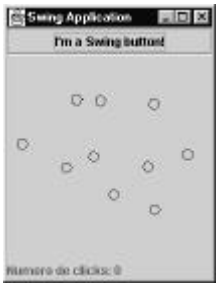
```
class TestaEventos4 extends JFrame
{
  JButton but1, but2;

  public TestaEventos4()
  {
    ...
    but1.addMouseListener(
       new MouseAdapter() {
          public void mouseClicked(MouseEvent e)
          {
            System.out.println("Clicou");
          }
          public void mouseReleased(MouseEvent e)
          {
            System.out.println("Soltou");
          }
       });
   ...
  }
}
```

- Observe que só sobrescrevemos os eventos desejados: mouseClicked(..) e mouseReleased(..)

- Exemplo 3: usando uma classe interna anônima para tratar o evento gerado quando se clica no botão de fechar em uma janela (frame)

```
class TestaEventos5 extends JFrame
{
  JButton but1, but2;

  public TestaEventos5()
  {
    // adiciona um WindowListener no frame
    addWindowListener(
       new WindowAdapter() {
          public void windowClosing(WindowEvent e)
          {
            System.exit(0); // fecha e sai do programa
          }
       });
   ...
  }
}
```

- Este é o exemplo mais utilizado, pois permite controlarmos o encerramento do programa se o usuário tentar fechar a janela



- Exemplo 4: aplicação tem uma área de desenho (um JPanel), um botão e um label. Cada clique no JPanel desenha um pequeno círculo na posição
- Código organizado em duas classes:
  - TestaMouse - classe principal, cria a interface gráfica
  - AreaDesenho - classe para implementar a área de desenho, gerenciar os clicks e manter a lista de círculos já adicionados

Exercícios

1) Altere o programa do exemplo de maneira que o mesmo ligue cada duas bolinhas por uma reta.

2) Altere o exercício 1 de maneira que toda a vez que o usuário clicar em uma bolinha esta mude de cor.

3) Altere o exercício 2 de forma que o usuário possa arrastar as bolinhas sobre a área de desenho e não apenas inserir novas. Consulte a documentação da classe **MouseMotionAdapter** para verificar como detectar os deslocamentos do mouse enquanto “clicado”.