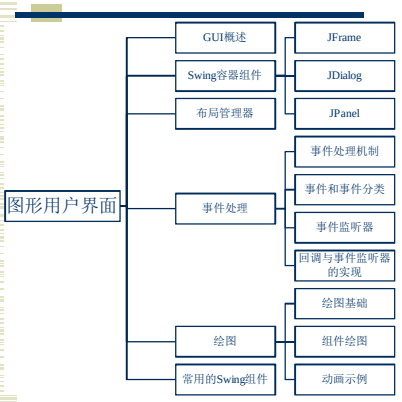
第10章 图形编程



1、主要的GUI类库

1）AWT提供的控件依赖系统的实现。

2）Swing控件由纯Java编写的，使用基本图形元素直接在屏幕上绘制，属于轻量级控件。

在搭建界面时，如果将两者同时使用，当AWT控件与Swing控件重合时，AWT控件的显示优先级高，总是绘制在Swing控件的上面。因此不建议混用。

3）JavaFX和SWT（Standard Widget Toolkit, SWT）：需要以第三方库的形式单独下载。

JavaFX旨在建立大量可在电脑和手机上运行(支持触摸设备)的网络程序。(从JDK11的发布开始，JavaFX已经不被包含在JDK中了)

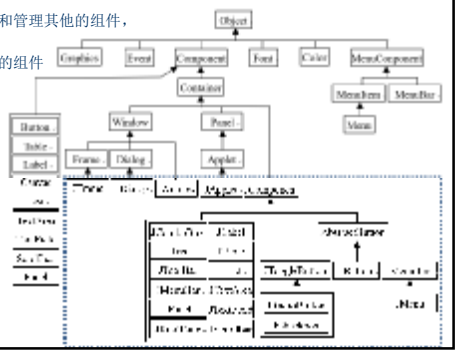
2、GUI编程概述及开发流程

GUI编程：由事件驱动的以图形化进行输入输出数据的编程方式：

- 主要对象：
  - 组件：分为原子组件和容器组件。
    按钮、文本组件、复选框和单选钮 、 下拉式列表 、列表、进度条、菜单 、以及对话框、面板、窗口
  - 布局管理器
  - 事件
- 其他部分：
  - 字体
  - 颜色
  - 绘图

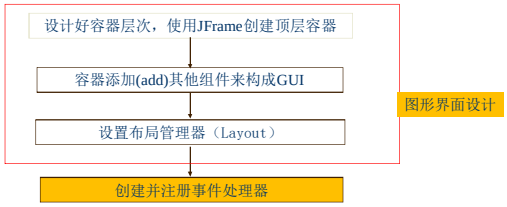
2、GUI编程概述及开发流程

- **容器组件**
  - 可以容纳和管理其他的组件，
- **原子组件**
  - 不可分割的组件

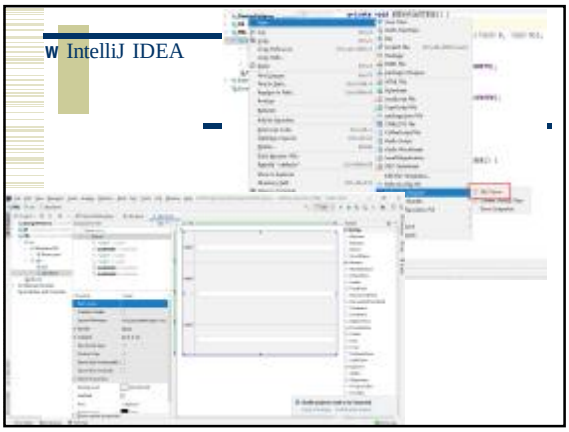


2、GUI编程概述及开发流程

- 图形用户界面设计步骤



w IntelliJ IDEA



10.2 Swing容器组件

根据容器所在的层级，可以将容器分为两类：顶层容器和中间容器。

(1) 顶层容器。可以独立显示，但不能作为组件被包含到其它容器中。

例如，JFrame、JDialog、Jwindow和JInternalFrame，绝大多数 Swing GUI程序使用 JFrame 作为顶层容器。

(2) 中间容器。中间容器不可独立显示,作为基本组件的载体，它必须被放入顶层容器或其它中间容器中。

常见的Swing中间容器:JPanel、JScrollPane、JTabbedPane、JToolBar、JMenuBar等。

顶级容器概述

JFrame, JDialog, JWindow, JApplet以及JInternalFrame这五个Swing容器都实现了RootPaneContainer接口，他们都将操作委托给了JRootPane。

10.2.1JFrame

使用JFrame时，需要注意以下几点：

(1) 每个GUI组件只能被添加到一个容器中。

(2) 创建JFrame对象后，默认情况下布局管理器是BorderLayout，

(3) Swing中事件处理和绘画代码都在一个单独的线程中执行。该线程确保了事件处理器都能串行的执行，并且绘画过程不会被事件打断。因此，在main方法或其他启动界面的线程，尽量使用SwingUtilities.invokeLater(Runnable doRun) 启动线程来操作界面。

10.2.1 JFrame

JFrame类的常用方法:

- Container getContentPane() 返回框架内容格。
- setVisible(boolean b) 使框架可见/不可见(true/false)
- hide() 隐藏框架
- setTitle() 设置框架的标题
- pack() 调整窗口正好容纳各组件
- setSize(int w,int h) 设置框架的尺寸
- setLocation(int x, int y) 设置框架位置
- reSize(int w, int h) 调整框架的尺寸(宽/高为w/h)
- setBounds(int x, int y, int w,int h)
  调整框架的位置及尺寸(左上角为(x,y), 宽、高为w、h)
- add(Component ob) 将其它组件ob加入到框架的中心位置
- add(String p, Component ob) 将组件ob加入到框架的p位置)

```
class JFrameDemo extends JFrame {
   private JButton jButton1, jButton2, jButton3, jButton4 , jButton5 ;
   public JFrameDemo() {
         this.setSize(400, 200); // 设置框架尺寸，但系统在默认屏幕位置上显示框架
         jButton1 = new JButton("北");          // 创建按钮，按钮上的标签文字为“北”
         jButton2 = new JButton("南");          // 创建按钮，按钮上的标签文字为“南”
         jButton3 = new JButton("西");          // 创建按钮，按钮上的标签文字为“西”
         jButton4 = new JButton("东");          // 创建按钮，按钮上的标签文字为“东”
         jButton5 = new JButton("中");          // 创建按钮，按钮上的标签文字为“中”
         add(jButton1, BorderLayout.NORTH); // 将按钮放到窗口的的上部区域
         add(jButton2, BorderLayout.SOUTH); // 将按钮放到窗口的的下部区域
         add(jButton3, BorderLayout.WEST);   // 将按钮放到窗口的的左侧区域
         add(jButton4, BorderLayout.EAST);   // 将按钮放到窗口的的右侧区域
         add(jButton5, BorderLayout.CENTER); // 将按钮放到窗口的的中部区域
         this.setTitle("JFrame");        // 设置窗口标题
         this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); //设置关闭行为
         }
```

```
public static void main(String[] args) {
    JFrameDemo demo=new JFrameDemo();// 创建框架对象
    demo.setVisible(true);// 显示窗口
    }}
```

- 其他的容器组件，可自己阅读教材学习。

10.3 布局管理器

- 容器中的组件在容器中的大小和位置是由容器的布局管理器（layout manager）来布置的，这样能根据不同的屏幕自动进行排版。
- 每个容器中都有默认的布局管理器，缺省的布局管理器为：
  所有窗口——BorderLayout(文件对话框除外)
  所有面板(包括Applet)——FlowLayout
- 但容器组件可以通过setLayout方法设置或取消容器的布局管理器，其语法格式为：
  void setLayout(LayoutManager mgr)
  例如：setLayout(new FlowLayout()); //为容器设置FlowLayout布局
  setLayout(null); // 取消容器的布局管理器

10.3 布局管理器

Java预定义了不同的布局管理器类中，主要有：

- FlowLayout（ 流式布局/顺序布局）
- GridLayout（网格布局）
- GridBagLayout（网格包布局）
- BoxLayout（箱式布局）
- GroupLayout（分组布局）
- CardLayout（卡片布局）
- BorderLayout（边界布局）
- SpringLayout（弹性布局）等。

1、FlowLayout布局管理器

FlowLayout布局管理将组件顺序依次摆放。当一行放不下时才往下一行放。每一行组件可以根据FlowLayout创建时alignment参数的指定要求放在屏幕的中心位置(缺省)、左侧或右侧。

FlowLayout类有三个构造方法：

- public FlowLayout()
- public FlowLayout(int alignment)
- public FlowLayout(int alignment,int horizontalGap,int verticalGap)

alignment用于指定放置格式，必须是下面三值之一：

FlowLayout.LEFT　　放左侧
FlowLayout.CENTER　　放中心(缺省)
FlowLayout.RIGHT　　放右侧

horizontalGap、verticalGap指定组件间隔距离(以像素为单位)。如果用户没有指定间隔值，FlowLayout将自动指定其值为5。

```
class FlowLayoutDemo  {
        // 初始化框架
   public void initJFrame() {
        JFrame frame = new JFrame("FlowLayout示例");
        frame.setLayout(new FlowLayout());          // 设置流式布局
        JButton[] btn = new JButton[8];
        for(int i = 0; i < btn.length; i++) {
                btn[i] = new JButton("按钮" + i);
                frame.add(btn[i]);
            }
        frame.setSize(300,200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
        }
   public static void main(String[] args) {
        FlowLayoutDemo demo = new FlowLayoutDemo();
        demo.initJFrame();
        }}
```

(a)改变窗口宽度前　　(b)改变窗口宽度后

图 10-6 例 10-4 的运行结果界面

2. 边界布局：java.awt.BorderLayout

BorderLayout布局管理器将容器分为有五个区域：中心和东、西、南、北。如下图所示：

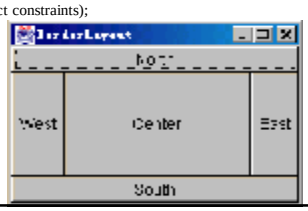
- 还可以限定区域间距 public **BorderLayout**(int hgap, int vgap)

参数： hgap - 水平间距。 vgap - 垂直间距。

- 在BorderLayout布局的容器中加入组件的add方法通常为：

void add(String position,Component c);

void add(Component c, Object constraints);



```
public class BDemo extends JFrame {

   public BDemo() {
      setSize(250,200);
      setLayout(new BorderLayout(5, 3));
      add(new JButton("North"), "North");
      add(new JButton("South"), "South");
      add(new JButton("East"), "East");
      add(new JButton("West"), BorderLayout.WEST);
      add(new JButton("Center"), BorderLayout.CENTER);
   }
}
```

22

3、GridLayout布局管理器

GridLayout布局是将容器空间划分为m行n列的大小相等的网格区域，每个格子允许放置一个组件，组件将自动占满格子。非常适合数量庞大的组件

比顺序布局多了行和列的设置

```
public GridLayout()
public GridLayout(int rows,int cols)
public GridLayout(int rows,int cols,int hgap,int vgap)
```

组件间的水平间隔　　组件间的垂直间隔

```
public class GridLayoutDemo extends JFrame{
   public GridLayoutDemo(String title) {
      super(title);
      setLayout(new GridLayout(3, 2));
      for (int i = 1; i <= 6; i++) {
          add(new JButton(i + ""));
      }
      pack();
   }
}
```

```
class GridLayoutDemo extends JFrame {
        String[] btnLabel = {"0","1","2","3","4","5","6","7","8","9","+","-","*","/","="};
        JButton[] btn;                                  // 计算器按钮
        JPanel resultPanel, btnPanel;  // 计算结果面板和计算器按钮面板
        JTextField resultText;                  // 计算结果显示文本域
        public GridLayoutDemo() {
                btn = new JButton[btnLabel.length];
                resultPanel = new JPanel();    // 构造计算结果面板
                btnPanel = new JPanel();       // 构造计算器按钮面板
                resultText = new JTextField(20);
                resultPanel.add(resultText);   // 结果显示文本域放入结果面板中
                btnPanel.setLayout(new GridLayout(5,3)); // 按钮面板为5行3列的GridLayout
                for(int i = 0; i < btnLabel.length; i++) {   // 将按钮依序加入计算器按钮面板
                        btn[i] = new JButton(btnLabel[i]);
                        btnPanel.add(btn[i]);}
                add(resultPanel, BorderLayout.NORTH); // 结果面板放在上部
                add(btnPanel, BorderLayout.CENTER);            // 按钮面板放在中央
                setSize(200,300);
                setVisible(true);
                setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);}
        public static void main(String[] args) {GridLayoutDemo cal = new GridLayoutDemo();}
```

4、CardLayout布局管理器

CardLayout类可使两个或更多个组件共享同一显示空间。CardLayout所管理的组件就像放在纸盒里的纸牌，在某一时刻只有最上面的一张可见。

- 当向采用CardLayout布局管理的容器中加入组件时，需要使用带两个参数的add方法，其中一个参数指定一个名称。
- 名称类似于图书检索卡片中的索引号，用户可通过指定组件的名字或指定第一或最后的组件(组件的顺序就是它们被加入到容器中的顺序)来选择要显示的组件，这需要使用带两个参数的show方法编程实现。

例. CardLayout布局管理器的使用

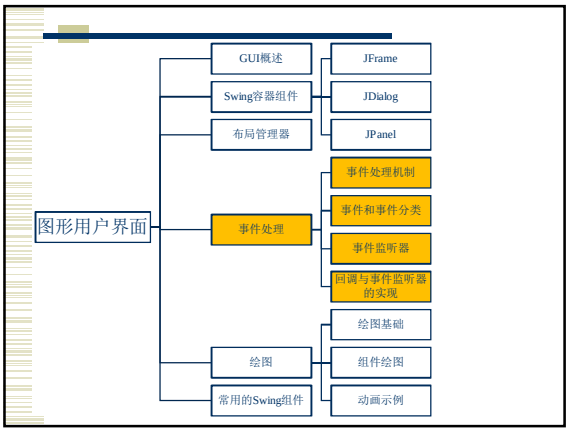
```
import java.awt.*;
import javax.swing.*;

public class  MyFrame
{public static void main (String args[ ])
 {JFrame frm=new JFrame("CardLayout");
   frm.setLayout(new CardLayout(10,15));
   JButton b1=new JButton("按钮1");
   frm.add("1",b1);
   frm.add("2", new JButton("按钮2"));
   frm.add("3", new JButton("按钮3"));
   frm.setVisible(true);
 }
}
```

```
class CardLayoutDemo {
    public static void main(String[] args) {
        JFrame frame = new JFrame("卡片布局示例");
        CardLayout cardLayout = new CardLayout();
        frame.setLayout(cardLayout);        // 设置为卡片布局
        Container container = frame.getContentPane();// 取内容窗格
        container.add(new JLabel("星期一", JLabel.CENTER), "1");
        container.add(new JLabel("星期二", JLabel.CENTER), "2");
        container.add(new JLabel("星期三", JLabel.CENTER), "3");
        container.add(new JLabel("星期四", JLabel.CENTER), "4");
        container.add(new JLabel("星期五", JLabel.CENTER), "5");
        container.add(new JLabel("星期六", JLabel.CENTER), "6");
        frame.setSize(400,200);      frame.setVisible(true);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        cardLayout.show(container,"1");     // 首先显示第一个标签
        for(int i = 0; i < 6; i++){// 间隔2秒显示下一个标签
          try {Thread.sleep(2000);
            cardLayout.next(container);} catch(InterruptedException e){} }}}
```

5、 GridBagLayout布局管理器

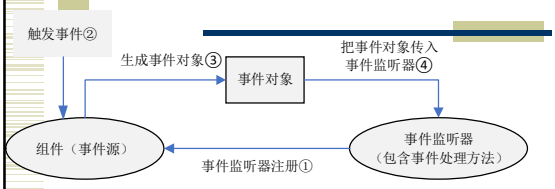
GridBagLayout是AWT提供的最灵活、最复杂的布局管理器，它将组件以多行多列放置，允许每个组件跨多行多列。例如下图就是一个GridBagLayout布局。



10.4 事件处理

1、事件处理机制
2、事件和事件分类
3、事件监听器
4、回调与事件监听器的实现

10.4.1 事件处理机制



应用委托模型编写事件处理程序一般包含以下几个步骤：

(1) 确定事件源应关注的事件。
(2) 设计处理各事件的事件监听器类。
(3) 创建事件监听器对象。
(4) 调用事件源的addXxxListener方法为事件源注册事件监听器

在委托模型中，事件处理主要涉及三个要素：

- 事件源
- 事件对象
- 事件监听器

事件处理机制

1．事件源

能够产生事件的组件都可以成为事件源，例如按钮、菜单、文本框等。

事件源通常提供注册和注销事件监听器（Event Listener）的方法，注册/注销监听器的方法：

事件源对象.addXXXListener(监听器对象)
事件源对象.removeXXXListener(监听器对象)

事件处理机制

2．事件对象（Event Object）

事件对象通常由用户操作触发，由Java虚拟机产生并传播的对象。

例如：点击按钮产生的事件（ActionEvent），按下某个键产生的事件(KeyEvent)，关闭窗口产生的事件(WindowEvent)

事件处理机制

3．事件监视器

- 事件监听器(Event Listener)用于接收和处理事件的对象。
- Java中采用委托模型的方式处理事件。即事件产生以后，不是由事件源处理事件，而是将事件委托给第三方对象——事件监听器来处理。

事件处理机制

3．事件监视器

- 事件监听器能够工作必须满足两个要求：
  第一，事件监听器实现了处理某种事件的接口方法；
  第二，需要将事件监听器注册到事件源中，从而与事件源建立关联。
- 根据事件源产生事件的类型和需要，可以为事件源注册一个或多个监视器，又称为注册监视器。注册监视器的方法：
  事件源对象.addXXXListener(监视器)（XXX为对应的事件类型）。

事件处理机制

4．事件处理方法

当监视器监听到事件源发生了相关的事件后，就要调用自身相应方法来处理事件。

例如：

```
JButton  jbutton=new JButton("Click");
jbutton.addActionListener(  new ActionListener(){
                    public void actionPerformed(ActionEvent  event){
                              处理代码}});
```

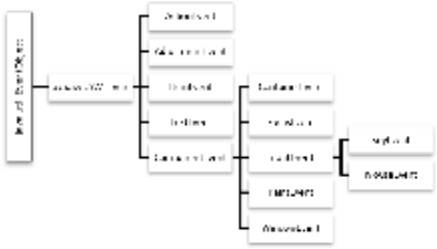
Java将事件进行了分类，并设计了对应特定事件事件处理接口，在这些接口中给出了指定的方法。

监视器根据需要重写其中的方法，从而实现特定事件的处理。

10.4.2 事件和事件分类

事件的分类：

- **低级事件**：低级事件是指基于组件和容器的事件，当一个组件上发生事件，如鼠标的进入、点击、拖放等，或组件的窗口开关等时，触发了组件事件。
- **高级事件（也称语意事件）**：高级事件是基于语义的事件，它可以不和特定的动作相关联。是用来描述用户操作所产生的结果，低级事件是高级事件的基础。



- 低级事件，如：
  - ComponentEvent（组件事件：组件尺寸的变化、移动）；
  - ContainerEvent（容器事件：组件增加、移动）；
  - WindowEvent（窗口事件：关闭窗口、窗口闭合、图标化）；
  - FocusEvent（焦点事件：焦点的获得和丢失）；
  - KeyEvent（键盘事件：键按下、释放）；
  - MouseEvent（鼠标事件：鼠标单击、移动）。
- 高级事件，如：
  - ActionEvent（动作事件：按钮按下，TextField中按Enter键）
  - AdjustmentEvent（调节事件：在滚动条上移动滑块以调节数值）
  - ItemEvent（项目事件：选择项目，不选择“项目改变”）
  - TextEvent（文本事件：文本对象改变）

10.4.3事件监听器

- 事件监听器的实现，有两种方法：

1、实现监听器接口： implements XXXListener

- 在事件源和事件监听器对象中进行约定的接口。
- 事件监听器接口的名称与事件类的名称是相对的，例如：KeyEvent事件类的监听器接口名为KeyListener

2、扩展监听适配类 ： extends XXXAdapter

JDK中也提供了大多数事件监听器接口的最简单的实现类，称之为事件适配器（Adapter）类.

举例说明

事件类型	监听接口	接口中的方法	适配器类
ActionEvent	ActionListener	actionPerformed(ActionEvent)	无
AdjustmentEvent	AdjustmentListener	adjustmentValueChanged(AdjustmentEvent)	无
ItemEvent	ItemListener	itemStateChanged(ItemEvent)	无
TextEvent	TextListener	textValueChanged(TextEvent)	无
MouseEvent	MouseListener	mouseClicked(MouseEvent) mouseEntered(MouseEvent) mouseExited(MouseEvent) mousePressed(MouseEvent) mouseReleased(MouseEvent)	MouseAdapter
MouseEvent	MouseMotionListener	mouseDragged(MouseEvent) mouseMoved(MouseEvent)	MouseMotionAdapter
KeyEvent	KeyListener	keyPressed(KeyEvent) keyReleased(KeyEvent) keyTyped(KeyEvent)	KeyAdapter
FocusEvent	FocusListener	focusGained(FocusEvent) focusLost(FocusEvent)	FocusAdapter
WindowEvent	WindowListener	windowActivated(WindowEvent) windowClosed(WindowEvent) windowClosing(WindowEvent) windowDeactivated(WindowEvent) windowDeiconified(WindowEvent) windowIconified(WindowEvent) windowOpened(WindowEvent)	WindowAdapter
DocumentEvent	DocumentListener	changedUpdate(DocumentEvent) removeUpdate(DocumentEvent) insertUpdate(DocumentEvent)	无

如果自定义一个键盘监听类：

```
public class myListener implements KeyListener{
    public void keyPressed(KeyEvent ev){
     ............
      }
    public void keyReleased(KeyEvent ev){
     ............
      }
    public void keyTyped(KeyEvent ev){
     ............
      } }
```

- 问题：如果一个监听器有若干方法,则必须将这些方法全部覆盖

2、扩展监听适配类

用事件适配器来处理事件，对低级事件可以简化事件监听器的编写 ,不用适配器时, 对低级事件的监听器必须重写多个方法。

```
public class myListener extends KeyAdapter {
  public void keyTyped(KeyEvent ev)
  { ............
    }
}
```

10.4.4 回调与事件监听器的实现

- 回调原理
- 实现方法

10.4.4 回调与事件监听器的实现

- 事件监听器接收来自事件源的事件并执行处理方法，而监听器的处理方法一般要回调事件源作用域的方法进行回应。这形成了回调机制。回调是程序模块之间常用的一种相互调用方式。

- 然后B类反过来调用A类中的方法d(),在这里面d()就是回调方法

10.4.4 回调与事件监听器的实现

在委托事件处理模型中，类A相当于事件源，类B相当于事件监视器，实现上述目的的设计方式有下面两类。

- （1）用闭包（closure）类实现事件监视器。可以用事件源的内部类的方式创建事件监视器；也可以把事件源和事件监视器类合而为一；如果监视器只有一个函数，还可以用Lambda表达式简写监视器。
- （2）用外部类实现事件监视器。这时监视器B为了调用事件源A的回调方法，需要给监视器B传入事件源A的引用。

10.4.4 回调与事件监听器的实现

- 1、事件源与事件监听器合而为一
- 2. 用匿名内部类定义事件监听器
- 3. 使用Lambda表达式定义事件监听器
- 4、使用外部类定义事件监听器

1、事件源与事件监听器合而为一

```
import java.awt.event.ActionListener;
import java.awt.EventQueue;
import java.awt.FlowLayout;
import java.awt.event.*;
import javax.swing.*;
public class GuiTest extends JFrame implements ActionListener{

JButton b1,b2;
public GuiTest() {
setLayout(new FlowLayout());
setBounds(500,500,100,100);
b1=new JButton("进入");  b2=new JButton("退出");
add(b1);  add(b2);
b1.addActionListener(this);
b2.addActionListener(this);
setVisible(true);
}
public void actionPerformed(ActionEvent e) {
if(e.getSource()==b1) {
JOptionPane.showMessageDialog(null, "alert", "alert", JOptionPane.ERROR_MESSAGE);
}else if(e.getSource()==b2) {
System.exit(ERROR);
}}
public static void main(String arg[]) {
    EventQueue.invokeLater(new Runnable(){//第一种
    @Override
    public void run() {   GuiTest frame=new GuiTest();} });
    }}
```

2. 用匿名内部类定义事件监听器

```
class MouseFrame2 extends JFrame {
 private JLabel statusbar;
 public MouseFrame2() {
        super("鼠标事件示例");
        statusbar = new JLabel("这是状态栏");
        add(statusbar, BorderLayout.SOUTH);
        // 匿名内部类：通过继承适配器类，实现鼠标事件监听器
 this.addMouseListener(new MouseAdapter() {
     public void mouseClicked(MouseEvent e) {statusbar.setText("您点击了窗口!"); }
     public void mouseExited(MouseEvent e) {statusbar.setText("鼠标离开了窗口!");}
        });
        // 匿名内部类：通过实现接口，实现鼠标移动事件监听器
this.addMouseMotionListener(new MouseMotionListener() {
            public void mouseDragged(MouseEvent e) {
                String s = "鼠标拖拉: x=" + e.getX() + ", y= " + e.getY();
                statusbar.setText(s);
        }
            public void mouseMoved(MouseEvent e) {
                String s = "鼠标移动: x=" + e.getX() + ", y= " + e.getY();
                statusbar.setText(s);            } });
 setSize(300,200);
 setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  }}
```

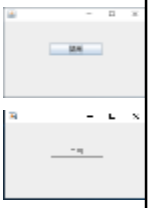
```
public class MouseEventDemo2 {
  public static void main(String[] args) {
        MouseFrame2 frame = new MouseFrame2();
            SwingUtilities.invokeLater(()->{
                frame.setVisible(true);// 创建线程，显示窗口
                });
  }
}
```

3. 使用Lambda表达式定义事件监听器

对函数式接口的监听器接口可以用Lambda表达式代替匿名内部类

```
public class LambdaListenerDemo extends JFrame {
   JButton btn = new JButton("禁用"); // 创建标签为"禁用"的按钮
   public LambdaListenerDemo() {
     try {
        setLayout(null);      // 删除默认布局管理器
        setSize(300,200);
        btn.setBounds(new Rectangle(92, 46, 104, 25));

        btn.addActionListener( event->{ btn.setEnabled(false); } );
        add(btn, null);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            } catch(Exception e) { e.printStackTrace(); }
        }
   public static void main(String[] args) {
        LambdaListenerDemo frame = new LambdaListenerDemo();
        SwingUtilities.invokeLater(()->{
            frame.setVisible(true);// 显示窗口
                });
 } }
```



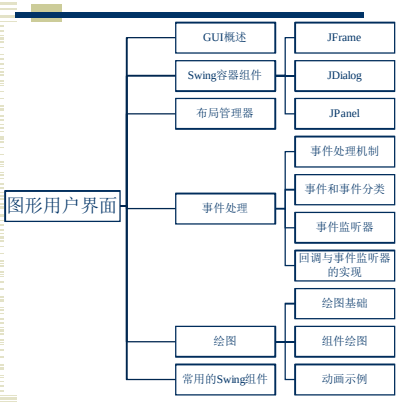
4、使用外部类定义事件监听器

```
class Frame1 extends JFrame {
  JButton jButton1 = new JButton("禁用");
  public Frame1() {
     add(jButton1, BorderLayout.CENTER);
     //注册MyActionListener的对象，并传入当前事件源框体的引用this
     jButton1.addActionListener(new MyActionListener(this));
     this.setBounds(200, 200, 200, 200);
  }
  //实现事件监视器将回调的方法
  public void doAction(ActionEvent e) {
          jButton1.setEnabled(false) ;}
    }

class MyActionListener implements java.awt.event.ActionListener {
     mySubject adaptee;
     //事件监视器初始化时，传入将回调的对象的引用，这类对象用mySubject接口标注
     public MyActionListener(mySubject adaptee) {
                this.adaptee = adaptee;
                }
     public void actionPerformed(ActionEvent e) {
                adaptee.doAction(e);//回调事件源的方法
                }}
```

```
//测试类
public class OuterListenerDemo {
        public static void main(String arg[]) {
                Frame1 frame = new Frame1();
                SwingUtilities.invokeLater(()->{
                        frame.setVisible(true);// 显示窗口
                        });
        }
}
```

第10章 图形编程



1、主要的GUI类库

1）AWT提供的控件依赖系统的实现。

2）Swing控件由纯Java编写的，使用基本图形元素直接在屏幕上绘制，属于轻量级控件。

在搭建界面时，如果将两者同时使用，当AWT控件与Swing控件重合时，AWT控件的显示优先级高，总是绘制在Swing控件的上面。因此不建议混用。

3）JavaFX和SWT（Standard Widget Toolkit, SWT）：需要以第三方库的形式单独下载。

JavaFX旨在建立大量可在电脑和手机上运行(支持触摸设备)的网络程序。(从JDK11的发布开始，JavaFX已经不被包含在JDK中了)

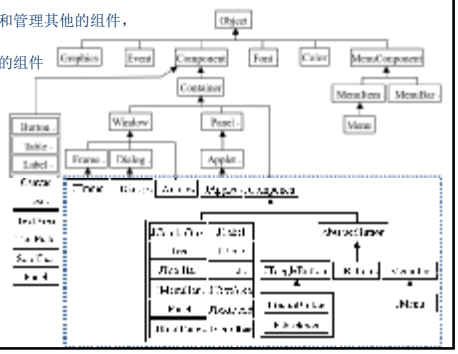
2、GUI编程概述及开发流程

GUI编程：由事件驱动的以图形化进行输入输出数据的编程方式：

- 主要对象：
  - 组件：分为原子组件和容器组件。
    按钮、文本组件、复选框和单选钮 、 下拉式列表 、列表、进度条、菜单 、以及对话框、面板、窗口
  - 布局管理器
  - 事件
- 其他部分：
  - 字体
  - 颜色
  - 绘图

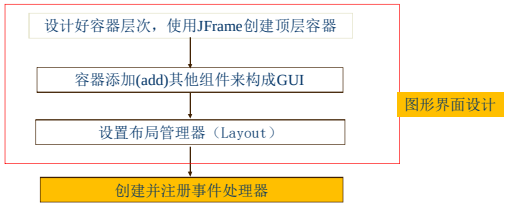
2、GUI编程概述及开发流程

- **容器组件**
  - 可以容纳和管理其他的组件，
- **原子组件**
  - 不可分割的组件

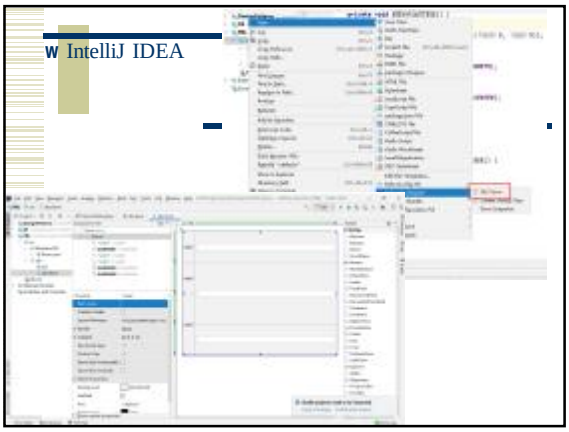


2、GUI编程概述及开发流程

- 图形用户界面设计步骤



w IntelliJ IDEA



10.2 Swing容器组件

根据容器所在的层级，可以将容器分为两类：顶层容器和中间容器。

(1) 顶层容器。可以独立显示，但不能作为组件被包含到其它容器中。

例如，JFrame、JDialog、Jwindow和JInternalFrame，绝大多数 Swing GUI程序使用 JFrame 作为顶层容器。

(2) 中间容器。中间容器不可独立显示,作为基本组件的载体，它必须被放入顶层容器或其它中间容器中。

常见的Swing中间容器:JPanel、JScrollPane、JTabbedPane、JToolBar、JMenuBar等。

顶级容器概述

JFrame, JDialog, JWindow, JApplet以及JInternalFrame这五个Swing容器都实现了RootPaneContainer接口，他们都将操作委托给了JRootPane。

10.2.1JFrame

使用JFrame时，需要注意以下几点：

(1) 每个GUI组件只能被添加到一个容器中。

(2) 创建JFrame对象后，默认情况下布局管理器是BorderLayout，

(3) Swing中事件处理和绘画代码都在一个单独的线程中执行。该线程确保了事件处理器都能串行的执行，并且绘画过程不会被事件打断。因此，在main方法或其他启动界面的线程，尽量使用SwingUtilities.invokeLater(Runnable doRun) 启动线程来操作界面。

10.2.1 JFrame

JFrame类的常用方法:

- Container getContentPane() 返回框架内容格。
- setVisible(boolean b) 使框架可见/不可见(true/false)
- hide() 隐藏框架
- setTitle() 设置框架的标题
- pack() 调整窗口正好容纳各组件
- setSize(int w,int h) 设置框架的尺寸
- setLocation(int x, int y) 设置框架位置
- reSize(int w, int h) 调整框架的尺寸(宽/高为w/h)
- setBounds(int x, int y, int w,int h)
  调整框架的位置及尺寸(左上角为(x,y), 宽、高为w、h)
- add(Component ob) 将其它组件ob加入到框架的中心位置
- add(String p, Component ob) 将组件ob加入到框架的p位置)

```
class JFrameDemo extends JFrame {
   private JButton jButton1, jButton2, jButton3, jButton4 , jButton5 ;
   public JFrameDemo() {
         this.setSize(400, 200); // 设置框架尺寸，但系统在默认屏幕位置上显示框架
         jButton1 = new JButton("北");          // 创建按钮，按钮上的标签文字为“北”
         jButton2 = new JButton("南");          // 创建按钮，按钮上的标签文字为“南”
         jButton3 = new JButton("西");          // 创建按钮，按钮上的标签文字为“西”
         jButton4 = new JButton("东");          // 创建按钮，按钮上的标签文字为“东”
         jButton5 = new JButton("中");          // 创建按钮，按钮上的标签文字为“中”
         add(jButton1, BorderLayout.NORTH); // 将按钮放到窗口的的上部区域
         add(jButton2, BorderLayout.SOUTH); // 将按钮放到窗口的的下部区域
         add(jButton3, BorderLayout.WEST);   // 将按钮放到窗口的的左侧区域
         add(jButton4, BorderLayout.EAST);   // 将按钮放到窗口的的右侧区域
         add(jButton5, BorderLayout.CENTER); // 将按钮放到窗口的的中部区域
         this.setTitle("JFrame");        // 设置窗口标题
         this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); //设置关闭行为
         }
```

```
public static void main(String[] args) {
    JFrameDemo demo=new JFrameDemo();// 创建框架对象
    demo.setVisible(true);// 显示窗口
    }}
```

- 其他的容器组件，可自己阅读教材学习。

10.3 布局管理器

- 容器中的组件在容器中的大小和位置是由容器的布局管理器（layout manager）来布置的，这样能根据不同的屏幕自动进行排版。
- 每个容器中都有默认的布局管理器，缺省的布局管理器为：
  所有窗口——BorderLayout(文件对话框除外)
  所有面板(包括Applet)——FlowLayout
- 但容器组件可以通过setLayout方法设置或取消容器的布局管理器，其语法格式为：
  void setLayout(LayoutManager mgr)
  例如：setLayout(new FlowLayout()); //为容器设置FlowLayout布局
  setLayout(null); // 取消容器的布局管理器

10.3 布局管理器

Java预定义了不同的布局管理器类中，主要有：

- FlowLayout（ 流式布局/顺序布局）
- GridLayout（网格布局）
- GridBagLayout（网格包布局）
- BoxLayout（箱式布局）
- GroupLayout（分组布局）
- CardLayout（卡片布局）
- BorderLayout（边界布局）
- SpringLayout（弹性布局）等。

1、FlowLayout布局管理器

FlowLayout布局管理将组件顺序依次摆放。当一行放不下时才往下一行放。每一行组件可以根据FlowLayout创建时alignment参数的指定要求放在屏幕的中心位置(缺省)、左侧或右侧。

FlowLayout类有三个构造方法：

- public FlowLayout()
- public FlowLayout(int alignment)
- public FlowLayout(int alignment,int horizontalGap,int verticalGap)

alignment用于指定放置格式，必须是下面三值之一：

FlowLayout.LEFT　　放左侧
FlowLayout.CENTER　　放中心(缺省)
FlowLayout.RIGHT　　放右侧

horizontalGap、verticalGap指定组件间隔距离(以像素为单位)。如果用户没有指定间隔值，FlowLayout将自动指定其值为5。

```
class FlowLayoutDemo  {
        // 初始化框架
   public void initJFrame() {
        JFrame frame = new JFrame("FlowLayout示例");
        frame.setLayout(new FlowLayout());          // 设置流式布局
        JButton[] btn = new JButton[8];
        for(int i = 0; i < btn.length; i++) {
                btn[i] = new JButton("按钮" + i);
                frame.add(btn[i]);
            }
        frame.setSize(300,200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
        }
   public static void main(String[] args) {
        FlowLayoutDemo demo = new FlowLayoutDemo();
        demo.initJFrame();
        }}
```

(a)改变窗口宽度前　　(b)改变窗口宽度后

图 10-6 例 10-4 的运行结果界面

2. 边界布局：java.awt.BorderLayout

BorderLayout布局管理器将容器分为有五个区域：中心和东、西、南、北。如下图所示：

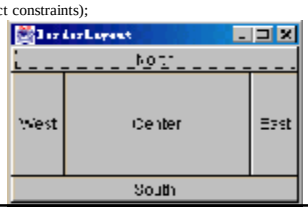
- 还可以限定区域间距 public **BorderLayout**(int hgap, int vgap)

参数： hgap - 水平间距。 vgap - 垂直间距。

- 在BorderLayout布局的容器中加入组件的add方法通常为：

void add(String position,Component c);

void add(Component c, Object constraints);



```
public class BDemo extends JFrame {

   public BDemo() {
      setSize(250,200);
      setLayout(new BorderLayout(5, 3));
      add(new JButton("North"), "North");
      add(new JButton("South"), "South");
      add(new JButton("East"), "East");
      add(new JButton("West"), BorderLayout.WEST);
      add(new JButton("Center"), BorderLayout.CENTER);
   }
}
```

22

3、GridLayout布局管理器

GridLayout布局是将容器空间划分为m行n列的大小相等的网格区域，每个格子允许放置一个组件，组件将自动占满格子。非常适合数量庞大的组件

比顺序布局多了行和列的设置

```
public GridLayout()
public GridLayout(int rows,int cols)
public GridLayout(int rows,int cols,int hgap,int vgap)
```

组件间的水平间隔　　组件间的垂直间隔

```
public class GridLayoutDemo extends JFrame{
   public GridLayoutDemo(String title) {
      super(title);
      setLayout(new GridLayout(3, 2));
      for (int i = 1; i <= 6; i++) {
          add(new JButton(i + ""));
      }
      pack();
   }
}
```

```
class GridLayoutDemo extends JFrame {
        String[] btnLabel = {"0","1","2","3","4","5","6","7","8","9","+","-","*","/","="};
        JButton[] btn;                                  // 计算器按钮
        JPanel resultPanel, btnPanel;  // 计算结果面板和计算器按钮面板
        JTextField resultText;                  // 计算结果显示文本域
        public GridLayoutDemo() {
                btn = new JButton[btnLabel.length];
                resultPanel = new JPanel();    // 构造计算结果面板
                btnPanel = new JPanel();       // 构造计算器按钮面板
                resultText = new JTextField(20);
                resultPanel.add(resultText);   // 结果显示文本域放入结果面板中
                btnPanel.setLayout(new GridLayout(5,3)); // 按钮面板为5行3列的GridLayout
                for(int i = 0; i < btnLabel.length; i++) {   // 将按钮依序加入计算器按钮面板
                        btn[i] = new JButton(btnLabel[i]);
                        btnPanel.add(btn[i]);}
                add(resultPanel, BorderLayout.NORTH); // 结果面板放在上部
                add(btnPanel, BorderLayout.CENTER);            // 按钮面板放在中央
                setSize(200,300);
                setVisible(true);
                setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);}
        public static void main(String[] args) {GridLayoutDemo cal = new GridLayoutDemo();}
```

4、CardLayout布局管理器

CardLayout类可使两个或更多个组件共享同一显示空间。CardLayout所管理的组件就像放在纸盒里的纸牌，在某一时刻只有最上面的一张可见。

- 当向采用CardLayout布局管理的容器中加入组件时，需要使用带两个参数的add方法，其中一个参数指定一个名称。
- 名称类似于图书检索卡片中的索引号，用户可通过指定组件的名字或指定第一或最后的组件(组件的顺序就是它们被加入到容器中的顺序)来选择要显示的组件，这需要使用带两个参数的show方法编程实现。

例. CardLayout布局管理器的使用

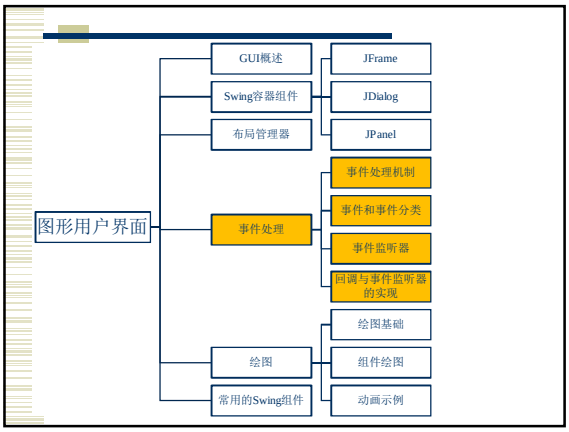
```
import java.awt.*;
import javax.swing.*;

public class  MyFrame
{public static void main (String args[ ])
 {JFrame frm=new JFrame("CardLayout");
   frm.setLayout(new CardLayout(10,15));
   JButton b1=new JButton("按钮1");
   frm.add("1",b1);
   frm.add("2", new JButton("按钮2"));
   frm.add("3", new JButton("按钮3"));
   frm.setVisible(true);
 }
}
```

```
class CardLayoutDemo {
    public static void main(String[] args) {
        JFrame frame = new JFrame("卡片布局示例");
        CardLayout cardLayout = new CardLayout();
        frame.setLayout(cardLayout);        // 设置为卡片布局
        Container container = frame.getContentPane();// 取内容窗格
        container.add(new JLabel("星期一", JLabel.CENTER), "1");
        container.add(new JLabel("星期二", JLabel.CENTER), "2");
        container.add(new JLabel("星期三", JLabel.CENTER), "3");
        container.add(new JLabel("星期四", JLabel.CENTER), "4");
        container.add(new JLabel("星期五", JLabel.CENTER), "5");
        container.add(new JLabel("星期六", JLabel.CENTER), "6");
        frame.setSize(400,200);      frame.setVisible(true);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        cardLayout.show(container,"1");     // 首先显示第一个标签
        for(int i = 0; i < 6; i++){// 间隔2秒显示下一个标签
          try {Thread.sleep(2000);
            cardLayout.next(container);} catch(InterruptedException e){} }}}
```

5、 GridBagLayout布局管理器

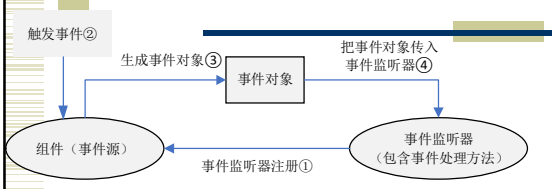
GridBagLayout是AWT提供的最灵活、最复杂的布局管理器，它将组件以多行多列放置，允许每个组件跨多行多列。例如下图就是一个GridBagLayout布局。



10.4 事件处理

1、事件处理机制
2、事件和事件分类
3、事件监听器
4、回调与事件监听器的实现

10.4.1 事件处理机制



应用委托模型编写事件处理程序一般包含以下几个步骤：

(1) 确定事件源应关注的事件。
(2) 设计处理各事件的事件监听器类。
(3) 创建事件监听器对象。
(4) 调用事件源的addXxxListener方法为事件源注册事件监听器

在委托模型中，事件处理主要涉及三个要素：

- 事件源
- 事件对象
- 事件监听器

事件处理机制

1．事件源

能够产生事件的组件都可以成为事件源，例如按钮、菜单、文本框等。

事件源通常提供注册和注销事件监听器（Event Listener）的方法，注册/注销监听器的方法：

事件源对象.addXXXListener(监听器对象)
事件源对象.removeXXXListener(监听器对象)

事件处理机制

2．事件对象（Event Object）

事件对象通常由用户操作触发，由Java虚拟机产生并传播的对象。

例如：点击按钮产生的事件（ActionEvent），按下某个键产生的事件(KeyEvent)，关闭窗口产生的事件(WindowEvent)

事件处理机制

3．事件监视器

- 事件监听器(Event Listener)用于接收和处理事件的对象。
- Java中采用委托模型的方式处理事件。即事件产生以后，不是由事件源处理事件，而是将事件委托给第三方对象——事件监听器来处理。

事件处理机制

3．事件监视器

- 事件监听器能够工作必须满足两个要求：
  第一，事件监听器实现了处理某种事件的接口方法；
  第二，需要将事件监听器注册到事件源中，从而与事件源建立关联。
- 根据事件源产生事件的类型和需要，可以为事件源注册一个或多个监视器，又称为注册监视器。注册监视器的方法：
  事件源对象.addXXXListener(监视器)（XXX为对应的事件类型）。

事件处理机制

4．事件处理方法

当监视器监听到事件源发生了相关的事件后，就要调用自身相应方法来处理事件。

例如：

```
JButton  jbutton=new JButton("Click");
jbutton.addActionListener(  new ActionListener(){
                    public void actionPerformed(ActionEvent  event){
                              处理代码}});
```

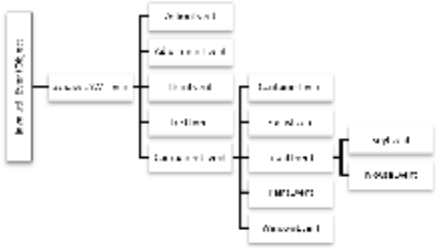
Java将事件进行了分类，并设计了对应特定事件事件处理接口，在这些接口中给出了指定的方法。

监视器根据需要重写其中的方法，从而实现特定事件的处理。

10.4.2 事件和事件分类

事件的分类：

- **低级事件**：低级事件是指基于组件和容器的事件，当一个组件上发生事件，如鼠标的进入、点击、拖放等，或组件的窗口开关等时，触发了组件事件。
- **高级事件（也称语意事件）**：高级事件是基于语义的事件，它可以不和特定的动作相关联。是用来描述用户操作所产生的结果，低级事件是高级事件的基础。



- 低级事件，如：
  - ComponentEvent（组件事件：组件尺寸的变化、移动）；
  - ContainerEvent（容器事件：组件增加、移动）；
  - WindowEvent（窗口事件：关闭窗口、窗口闭合、图标化）；
  - FocusEvent（焦点事件：焦点的获得和丢失）；
  - KeyEvent（键盘事件：键按下、释放）；
  - MouseEvent（鼠标事件：鼠标单击、移动）。
- 高级事件，如：
  - ActionEvent（动作事件：按钮按下，TextField中按Enter键）
  - AdjustmentEvent（调节事件：在滚动条上移动滑块以调节数值）
  - ItemEvent（项目事件：选择项目，不选择“项目改变”）
  - TextEvent（文本事件：文本对象改变）

10.4.3事件监听器

- 事件监听器的实现，有两种方法：

1、实现监听器接口： implements XXXListener

- 在事件源和事件监听器对象中进行约定的接口。
- 事件监听器接口的名称与事件类的名称是相对的，例如：KeyEvent事件类的监听器接口名为KeyListener

2、扩展监听适配类 ： extends XXXAdapter

JDK中也提供了大多数事件监听器接口的最简单的实现类，称之为事件适配器（Adapter）类.

举例说明

事件类型	监听接口	接口中的方法	适配器类
ActionEvent	ActionListener	actionPerformed(ActionEvent)	无
AdjustmentEvent	AdjustmentListener	adjustmentValueChanged(AdjustmentEvent)	无
ItemEvent	ItemListener	itemStateChanged(ItemEvent)	无
TextEvent	TextListener	textValueChanged(TextEvent)	无
MouseEvent	MouseListener	mouseClicked(MouseEvent) mouseEntered(MouseEvent) mouseExited(MouseEvent) mousePressed(MouseEvent) mouseReleased(MouseEvent)	MouseAdapter
MouseEvent	MouseMotionListener	mouseDragged(MouseEvent) mouseMoved(MouseEvent)	MouseMotionAdapter
KeyEvent	KeyListener	keyPressed(KeyEvent) keyReleased(KeyEvent) keyTyped(KeyEvent)	KeyAdapter
FocusEvent	FocusListener	focusGained(FocusEvent) focusLost(FocusEvent)	FocusAdapter
WindowEvent	WindowListener	windowActivated(WindowEvent) windowClosed(WindowEvent) windowClosing(WindowEvent) windowDeactivated(WindowEvent) windowDeiconified(WindowEvent) windowIconified(WindowEvent) windowOpened(WindowEvent)	WindowAdapter
DocumentEvent	DocumentListener	changedUpdate(DocumentEvent) removeUpdate(DocumentEvent) insertUpdate(DocumentEvent)	无

如果自定义一个键盘监听类：

```
public class myListener implements KeyListener{
    public void keyPressed(KeyEvent ev){
     ............
      }
    public void keyReleased(KeyEvent ev){
     ............
      }
    public void keyTyped(KeyEvent ev){
     ............
      } }
```

- 问题：如果一个监听器有若干方法,则必须将这些方法全部覆盖

2、扩展监听适配类

用事件适配器来处理事件，对低级事件可以简化事件监听器的编写 ,不用适配器时, 对低级事件的监听器必须重写多个方法。

```
public class myListener extends KeyAdapter {
  public void keyTyped(KeyEvent ev)
  { ............
    }
}
```

10.4.4 回调与事件监听器的实现

- 回调原理
- 实现方法

10.4.4 回调与事件监听器的实现

- 事件监听器接收来自事件源的事件并执行处理方法，而监听器的处理方法一般要回调事件源作用域的方法进行回应。这形成了回调机制。回调是程序模块之间常用的一种相互调用方式。

- 然后B类反过来调用A类中的方法d(),在这里面d()就是回调方法

10.4.4 回调与事件监听器的实现

在委托事件处理模型中，类A相当于事件源，类B相当于事件监视器，实现上述目的的设计方式有下面两类。

- （1）用闭包（closure）类实现事件监视器。可以用事件源的内部类的方式创建事件监视器；也可以把事件源和事件监视器类合而为一；如果监视器只有一个函数，还可以用Lambda表达式简写监视器。
- （2）用外部类实现事件监视器。这时监视器B为了调用事件源A的回调方法，需要给监视器B传入事件源A的引用。

10.4.4 回调与事件监听器的实现

- 1、事件源与事件监听器合而为一
- 2. 用匿名内部类定义事件监听器
- 3. 使用Lambda表达式定义事件监听器
- 4、使用外部类定义事件监听器

1、事件源与事件监听器合而为一

```
import java.awt.event.ActionListener;
import java.awt.EventQueue;
import java.awt.FlowLayout;
import java.awt.event.*;
import javax.swing.*;
public class GuiTest extends JFrame implements ActionListener{

JButton b1,b2;
public GuiTest() {
setLayout(new FlowLayout());
setBounds(500,500,100,100);
b1=new JButton("进入");  b2=new JButton("退出");
add(b1);  add(b2);
b1.addActionListener(this);
b2.addActionListener(this);
setVisible(true);
}
public void actionPerformed(ActionEvent e) {
if(e.getSource()==b1) {
JOptionPane.showMessageDialog(null, "alert", "alert", JOptionPane.ERROR_MESSAGE);
}else if(e.getSource()==b2) {
System.exit(ERROR);
}}
public static void main(String arg[]) {
    EventQueue.invokeLater(new Runnable(){//第一种
    @Override
    public void run() {   GuiTest frame=new GuiTest();} });
    }}
```

2. 用匿名内部类定义事件监听器

```
class MouseFrame2 extends JFrame {
 private JLabel statusbar;
 public MouseFrame2() {
        super("鼠标事件示例");
        statusbar = new JLabel("这是状态栏");
        add(statusbar, BorderLayout.SOUTH);
        // 匿名内部类：通过继承适配器类，实现鼠标事件监听器
 this.addMouseListener(new MouseAdapter() {
     public void mouseClicked(MouseEvent e) {statusbar.setText("您点击了窗口!"); }
     public void mouseExited(MouseEvent e) {statusbar.setText("鼠标离开了窗口!");}
        });
        // 匿名内部类：通过实现接口，实现鼠标移动事件监听器
this.addMouseMotionListener(new MouseMotionListener() {
            public void mouseDragged(MouseEvent e) {
                String s = "鼠标拖拉: x=" + e.getX() + ", y= " + e.getY();
                statusbar.setText(s);
        }
            public void mouseMoved(MouseEvent e) {
                String s = "鼠标移动: x=" + e.getX() + ", y= " + e.getY();
                statusbar.setText(s);            } });
 setSize(300,200);
 setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  }}
```

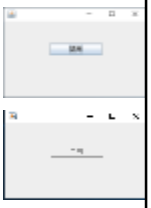
```
public class MouseEventDemo2 {
  public static void main(String[] args) {
        MouseFrame2 frame = new MouseFrame2();
            SwingUtilities.invokeLater(()->{
                frame.setVisible(true);// 创建线程，显示窗口
                });
  }
}
```

3. 使用Lambda表达式定义事件监听器

对函数式接口的监听器接口可以用Lambda表达式代替匿名内部类

```
public class LambdaListenerDemo extends JFrame {
   JButton btn = new JButton("禁用"); // 创建标签为"禁用"的按钮
   public LambdaListenerDemo() {
     try {
        setLayout(null);      // 删除默认布局管理器
        setSize(300,200);
        btn.setBounds(new Rectangle(92, 46, 104, 25));

        btn.addActionListener( event->{ btn.setEnabled(false); } );
        add(btn, null);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            } catch(Exception e) { e.printStackTrace(); }
        }
   public static void main(String[] args) {
        LambdaListenerDemo frame = new LambdaListenerDemo();
        SwingUtilities.invokeLater(()->{
            frame.setVisible(true);// 显示窗口
                });
 } }
```



4、使用外部类定义事件监听器

```
class Frame1 extends JFrame {
  JButton jButton1 = new JButton("禁用");
  public Frame1() {
     add(jButton1, BorderLayout.CENTER);
     //注册MyActionListener的对象，并传入当前事件源框体的引用this
     jButton1.addActionListener(new MyActionListener(this));
     this.setBounds(200, 200, 200, 200);
  }
  //实现事件监视器将回调的方法
  public void doAction(ActionEvent e) {
          jButton1.setEnabled(false) ;}
    }

class MyActionListener implements java.awt.event.ActionListener {
     mySubject adaptee;
     //事件监视器初始化时，传入将回调的对象的引用，这类对象用mySubject接口标注
     public MyActionListener(mySubject adaptee) {
                this.adaptee = adaptee;
                }
     public void actionPerformed(ActionEvent e) {
                adaptee.doAction(e);//回调事件源的方法
                }}
```

```
//测试类
public class OuterListenerDemo {
        public static void main(String arg[]) {
                Frame1 frame = new Frame1();
                SwingUtilities.invokeLater(()->{
                        frame.setVisible(true);// 显示窗口
                        });
        }
}
```