

Java 应用程序设计

GUI 程序设计

王晓东

wxd2870@163.com

中国海洋大学

April 10, 2018



参考书目

1. 张利国、刘伟 [编著], Java SE 应用程序设计, 北京理工大学出版社, 2007.10.



本章学习目标

1. Java 图形用户界面设计
2. GUI 事件处理机制
3. GUI 常用组件和视觉控制
4. AWT 绘图
5. Applet
6. Swing 概述
7. Swing 典型组件
 - ▶ JFrame
 - ▶ Swing 按钮、菜单和工具条
 - ▶ Swing 标准对话框
 - ▶ 表格和树
 - ▶ 定时器



大纲

Java GUI 设计

GUI 事件处理

AWT 常用组件和视觉控制

Applet

Swing 概述

Swing 典型组件



接下来...

Java GUI 设计

GUI 事件处理

AWT 常用组件和视觉控制

Applet

Swing 概述

Swing 典型组件



概念和术语

❖ 图形用户界面

GUI (Graphical User Interface), Java 主要分为 AWT 和 Swing 两大系列 GUI API。

❖ 抽象窗口工具集

AWT (Abstract Window Toolkit)

❖ 相关软件包

- ▶ java.awt 包：提供基本 GUI 组件、视觉控制和绘图工具 API。
- ▶ java.awt.event 包：提供 Java GUI 事件处理 API。



组件和容器

组件

组件（Component）是图形用户界面的基本组成元素，凡是能够以图形化方式显示在屏幕上并能够与用户进行交互的对象均为组件，如菜单、按钮、标签、文本框、滚动条等。

- ▶ 组件不能独立地显示出来，必须将组件放在一定的容器中才可以显示出来。
- ▶ JDK 的 `java.awt` 包中定义了多种 GUI 组件类，如 `Menu`、`Button`、`Label`、`TextField` 等。
- ▶ 抽象类 `java.awt.Component` 是除菜单相关组件之外所有 Java AWT 组件类的根父类，该类规定了 GUI 组件的基本特性，如尺寸、位置和颜色效果等，并实现了作为一个 GUI 部件所应具备的基本功能。
- ▶ `java.awt.MenuComponent` 是所有与菜单相关的组件的父类。



组件和容器

容器

容器（Container）实际上是 Component 的子类，容器类对象本身也是一个组件，具有组件的所有性质，另外还具有容纳其它组件和容器的功能。容器类对象可使用方法 add() 添加组件。

两种主要的容器类型

`java.awt.Window` 可自由停泊的顶级窗口。

`java.awt.Panel` 可作为容器容纳其他组件，但不能独立存在，必须被添加到其他容器（如 Frame）中。



常用的组件和容器 ①

组件类型	父类	说明
Button	Component	可接收点击操作的矩形 GUI 组件
Canvas	Component	用于绘图的面板
Checkbox	Component	复选框组件
CheckboxMenuItem	MenuItem	复选框菜单项组件
Choice	Component	下拉式列表框，内容不可改变
Component	Object	抽象的组件类
Container	Component	抽象的容器类
Dialog	Window	对话框组件，顶级窗口、带标题栏
FileDialog	Dialog	用于选择文件的平台相关对话框
Frame	Window	基本的 Java GUI 窗口组件
Label	Component	标签类



常用的组件和容器 ②

组件类型	父 类	说 明
List	Component	包含内容可变的条目的列表框组件
MenuBar	MenuComponent	菜单条组件
Menu	MenuItem	菜单组件
MenuItem	MenuComponent	菜单项组件
Panel	Container	基本容器类，不能单独停泊
PopupMenu	Menu	弹出式菜单组件
Scrollbar	Component	滚动条组件
ScrollPane	Container	带水平及垂直滚动条的容器组件
TextComponent	Component	TextField 和 TextArea 的基本功能
TextField	TextComponent	单行文本框
TextArea	TextComponent	多行文本域
Window	Container	抽象的 GUI 窗口类，无布局管理器



Frame 类

Frame 类的显示效果是一个标准的图形窗口，它封装了 GUI 组件的各种属性信息，如尺寸、可见性等。

Frame 类继承层次

```
java.lang.Object
|
+----java.awt.Component
      |
      +----java.awt.Container
            |
            +----java.awt.Window
                  |
                  +----java.awt.Frame
```

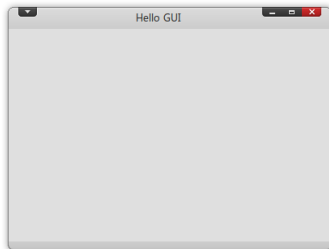
1. Frame 对象的显示效果是一个可自由停泊的顶级“窗口”，带有标题和尺寸重置角标。
2. Frame 默认初始化为不可见的，可以调用 Frame 对象的 `setVisible(true)` 方法使之变为可见。
3. 作为容器 Frame 还可使用 `add()` 方法包含其他组件。



First GUI Program

CODE ♦ TestFrame.java

```
1 import java.awt.Frame;  
3 public class TestFrame {  
4     public static void main(String args[]) {  
5         Frame f = new Frame("Hello GUI");  
6         f.setSize(200, 100);  
7         f.setVisible(true);  
8     }  
9 }
```

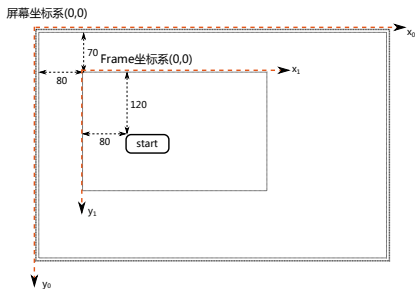


组件定位

Java 组件在容器中的定位由**布局管理器**决定。如要人工控制组件在容器中的定位，可取消布局管理器，然后使用 Component 类的下述成员方法设置：

- ▶ setLocation()
- ▶ setSize()
- ▶ setBounds()

GUI 坐标系



Panel 类

- ▶ Panel 提供容纳组件的空间。
- ▶ Panel 不能独立存在，必须被添加到其他容器中。
- ▶ 可以采用和所在容器不同的布局管理器。

Panel 类继承层次

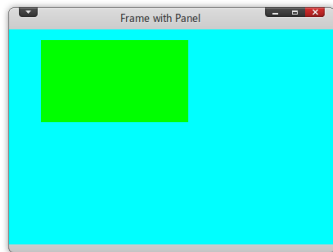
```
java.lang.Object
|
+----java.awt.Component
      |
      +----java.awt.Container
            |
            +----java.awt.Panel
```



Frame with Panel

CODE ♦ TestFrameWithPanel.java

```
1 import java.awt.Frame;  
2 import java.awt.Panel;  
3 import java.awt.Color;  
4 public class TestFrameWithPanel {  
5     public static void main(String args[]) {  
6         Frame f = new Frame("Frame_with_Panel");  
7         Panel pan = new Panel();  
8         f.setSize(200,170);  
9         f.setBackground(Color.cyan);  
10        f.setLayout(null); // 取消默认布局管理器  
11        pan.setSize(80,80);  
12        pan.setBackground(Color.green);  
13        f.add(pan);  
14        pan.setLocation(40,40);  
15        f.setLocation(300,300);  
16        f.setVisible(true);  
17    }  
18 }
```



布局管理器

容器对其中所包含组件的排列方式，包括组件的位置和大小设定，被称为容器的布局（Layout）。

为了使图形用户界面具有良好的平台无关性，Java 语言提供了布局管理器来管理容器的布局，而不建议直接设置组件在容器中的位置和尺寸。

布局管理器类层次

LayoutManager----FlowLayout

|

+----GridLayout

LayoutManager2----BorderLayout

|

+----CardLayout

|

+----GridBagLayout

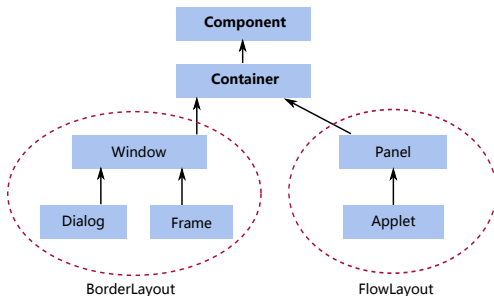
注：LayoutManager2 是 LayoutManager 的子接口。



布局管理器

- ▶ 每个容器都有一个布局管理器，当容器需要对某个组件进行定位或判断其大小尺寸时，就会调用其对应的布局管理器。
- ▶ 可以在容器创建后调用其 `setLayout()` 方法设置其布局管理器类型。
- ▶ `Container` 类型容器没有默认的布局管理器，即其 `layoutMgr` 属性为 `null`，在其子类中才进行分化。

默认的布局管理器



FlowLayout

❖ 流式布局

是 Panel（及其子类）类型容器的默认布局管理器类型。

❖ 布局效果

- ▶ 组件在容器中按照加入次序逐行定位，行内从左到右，一行排满后换行。
- ▶ 不改变组件尺寸，即按照组件原始大小进行显示。
- ▶ 组件间的对齐方式默认为居中对齐，也可在构造方法中设置不同的组件间距、行距及对齐方式。



FlowLayout

❖ 构造方法

▶ **public FlowLayout()**

组件对齐方式默认为居中对齐，组件的水平和垂直间距默认为 5 个像素。

▶ **public FlowLayout(int align)**

显式设定组件的对齐方式，组件的水平和垂直间距默认为 5 个像素。

FlowLayout.LEFT

FlowLayout.RIGHT

FlowLayout.CENTER

▶ **public FlowLayout(int align, int hgap, int vgap)**

显式设定组件的对齐方式、组件的水平和垂直间距。



FlowLayout 示例

CODE ▶ TestFlowLayout.java

```
1  import java.awt.Frame;  
2  import java.awt.Button;  
3  import java.awt.FlowLayout;  
4  public class TestFlowLayout {  
5      public static void main(String args[]) {  
6          Frame f = new Frame("FlowLayout");  
7          Button button1 = new Button("Ok");  
8          Button button2 = new Button("Open");  
9          Button button3 = new Button("Close");  
10         f.setLayout(new FlowLayout());  
11         f.add(button1);  
12         f.add(button2);  
13         f.add(button3);  
14         f.setSize(100, 100);  
15         f.setVisible(true);  
16     }  
17 }
```



BorderLayout

❖ 边界布局

是 Window 及其子类（包括 Frame、Dialog）容器的默认布局管理器。

❖ 布局效果

- ▶ BorderLayout 将整个容器的布局划分成东、西、南、北、中五个区域，组件只能被添加到指定的区域。如不指定组件的加入部位，则默认加入到 Center 区域。
- ▶ 每个区域只能加入一个组件，如加入多个，则先前加入的组件会被遗弃。
- ▶ 组件尺寸被强行控制，即与其所在区域的尺寸相同。



BorderLayout

❖ 构造方法

▶ **public BorderLayout()**

构造一个 BorderLayout 布局管理器，其所包含的组件/区域间距为 0。

▶ **public BorderLayout(int hgap, int vgap)**

构造一个 BorderLayout 布局管理器，根据参数的组件/区域间距。

❖ BorderLayout 型布局容器尺寸缩放原则

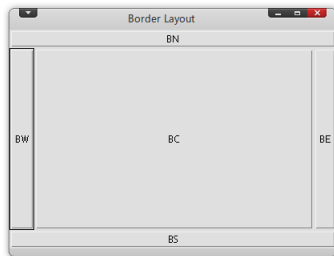
- ▶ 北、南两个区域只能在水平方向缩放（宽度可调整）。
- ▶ 东、西两个区域只能在垂直方向缩放（高度可调整）。
- ▶ 中部可在两个方向上缩放。



BorderLayout

CODE TestBorderLayout.java

```
1 import java.awt.Frame;  
2 import java.awt.Button;  
3 public class TestBorderLayout {  
4     public static void main(String args[]) {  
5         Frame f = new Frame("Border_Layout");  
6         Button bn = new Button("BN");  
7         Button bs = new Button("BS");  
8         Button bw = new Button("BW");  
9         Button be = new Button("BE");  
10        Button bc = new Button("BC");  
11        f.add(bn, "North");  
12        f.add(bs, "South");  
13        f.add(bw, "West");  
14        f.add(be, "East");  
15        f.add(bc, "Center");  
16        f.setSize(200, 200);  
17        f.setVisible(true);  
18    }  
19 }
```



下述两条语句作用相同:

```
1 f.add(bs, "South");  
2 f.add(bs, BorderLayout.SOUTH);
```



GridLayout

❖ 网格布局

❖ 布局效果

- ▶ 将容器区域划分成规则的矩形网格，每个单元格区域大小相等，组件被添加到每个单元格中，按组件加入顺序先从左到右填满一行后换行，行间从上到下。
- ▶ GridLayout 型布局的组件大小也被布局管理器强行控制，与单元格同等大小，当容器尺寸发生改变时，组件的相对位置保持不变，但大小自动调整。

❖ 构造方法

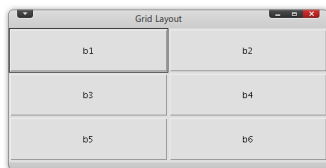
- ▶ **public GridLayout()** 所有组件于一行中，各占一列。
- ▶ **public GridLayout(int rows, int cols)**
通过参数指定布局的行数和列数。
- ▶ **public public GridLayout(int rows, int cols, int hgap, int vgap)**
通过参数指定布局的行数、列数，以及组件间水平间距和垂



GridLayout

CODE TestGridLayout.java

```
1  import java.awt.Frame;  
2  import java.awt.Button;  
3  import java.awt.GridLayout;  
  
5  public class TestGridLayout {  
6      public static void main(String args[]) {  
7          Frame f = new Frame("Grid Layout");  
8          Button b1 = new Button("b1");  
9          Button b2 = new Button("b2");  
10         Button b3 = new Button("b3");  
11         Button b4 = new Button("b4");  
12         Button b5 = new Button("b5");  
13         Button b6 = new Button("b6");  
14         f.setLayout(new GridLayout(3, 2));  
15         f.add(b1);  
16         f.add(b2);  
17         f.add(b3);  
18         f.add(b4);  
19         f.add(b5);  
20         f.add(b6);  
21         f.pack(); // 注意  
22         f.setVisible(true);  
23     }  
24 }
```



注意：pack() 方法是 Window 类中定义的，其功能是调整此窗口的大小，使之紧凑化以适合其中所包含的组件的原始尺寸和布局。



CardLayout

❖ 布局效果

- ▶ 将多个组件在同一容器区域内交替显示，相当于多张卡片摞在一起，只有最上面的卡片是可见的。
- ▶ 可以按名称显示某一张卡片，或按先后顺序依次显示，也可以直接定位到第一张或最后一张卡片。

❖ 主要方法

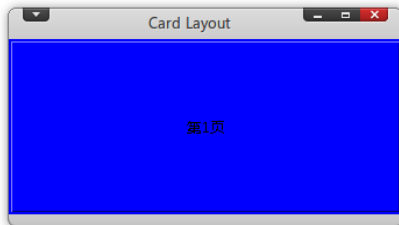
- ▶ `public void first(Container parent)`
- ▶ `public void last(Container parent)`
- ▶ `public void previous(Container parent)`
- ▶ `public void next(Container parent)`
- ▶ `public void show(Container parent, String name)`



CardLayout

CODE ▶ TestCardLayout.java

```
1 import java.awt.CardLayout;
2 ... ..
3 public class TestCardLayout {
4     public static void main(String args[]) {
5         Frame f = new Frame("CardLayout");
6         CardLayout cl = new CardLayout();
7         f.setLayout(cl);
8         Button[] b = new Button[4];
9         for (int i = 0; i < 4; i++) {
10             b[i] = new Button("第" + i + "页");
11             f.add(b[i], "page" + i);
12         }
13         b[0].setBackground(Color.green);
14         b[1].setBackground(Color.blue);
15         b[2].setBackground(Color.red);
16         f.pack();
17         f.setVisible(true);
18         while (true) {
19             try {
20                 Thread.sleep(1000);
21             } catch (InterruptedException e) {
22                 e.printStackTrace();
23             }
24             cl.next(f);
25         }
26     }
27 }
```

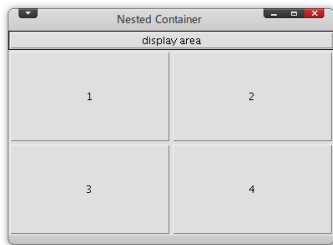


容器的嵌套使用

利用容器嵌套可以在某个原本只能包含一个组件的区域中显示多个组件。

CODE NestedContainer.java

```
1  import java.awt.Frame;
2  ... ...
3  public class NestedContainer {
4      public static void main(String args[]) {
5          Frame f = new Frame("Nested_Container");
6          Button b0 = new Button("display_area");
7          Panel p = new Panel();
8          p.setLayout(new GridLayout(2, 2));
9          Button b1 = new Button("1");
10         Button b2 = new Button("2");
11         Button b3 = new Button("3");
12         Button b4 = new Button("4");
13         p.add(b1);
14         p.add(b2);
15         p.add(b3);
16         p.add(b4);
17         f.add(b0, "North");
18         f.add(p, "Center");
19         f.pack();
20         f.setVisible(true);
21     }
22 }
```



接下来...

Java GUI 设计

GUI 事件处理

AWT 常用组件和视觉控制

Applet

Swing 概述

Swing 典型组件



Java 事件和事件处理机制

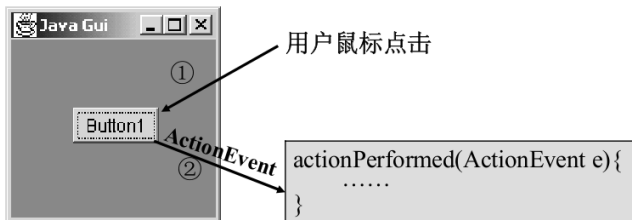
从 JDK 1.1 开始，Java 采用了一种名为“事件代理模型”（Event Delegation Model）的事件处理机制。基本原理如下：

1. 事先定义多种事件类型
2. 约定各种 GUI 组件在与用户交互时，遇到特定操作则会触发相应的事件，即自动创建事件类对象并提交给 Java 运行时系统。
3. 系统接收到事件类对象后，立即将其发送给专门的事件处理对象，该对象调用其事件处理方法，处理先前的事件类型对象，实现预期的处理逻辑。



Java 事件和事件处理机制

若需要关注某个组件产生的事件，则可以在该组件上注册适当的事件处理方法，实际上注册的事件处理器方法所属类型的一个对象——事件监听器。



事件处理相关概念

事件 (Event) 一个事件类型的对象，用于描述了发生什么事情，当用户在组件上进行操作时会触发相应的事件。

事件源 (Event Source) 能够产生事件 GUI 组件对象，如按钮、文本框等。

事件处理方法 (Event Handler) 能够接收、解析和处理事件类对象，实现与用户交互功能的方法。

事件监听器 (Event Listener) 调用事件处理方法的对象。



GUI 事件处理示例

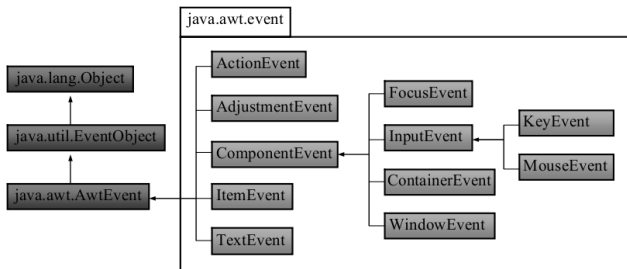
CODE TestActionEvent.java

```
1  import java.awt.*;
2  import java.awt.event.*;
3  public class TestActionEvent {
4      public static void main(String args[]) {
5          Frame f = new Frame("Test");
6          Button b = new Button("Press_Me!");
7          MyMonitor mm = new MyMonitor(); // A
8          b.addActionListener(mm);
9          f.add(b, BorderLayout.CENTER);
10         f.pack();
11         f.setVisible(true);
12     }
13 }
14 class MyMonitor implements ActionListener { // B
15     public void actionPerformed(ActionEvent e) { // C
16         System.out.println("A_button_has_been_pressed!");
17     }
18 }
```

- ▶ 通过注册监听器的方式对所关注的事件源进行监控。
- ▶ 注册监听器时应指明该监听器监控（感兴趣）的事件种类。
- ▶ 当事件源发生了某种类型的事件时，只触发事先已就该种事件类型注册过的监听器。



GUI 事件类型层次



GUI 事件及相应监听器接口

事件类型	相应监听器接口	监听器接口中的方法
Action	ActionListener	actionPerformed(ActionEvent)
Item	ItemListener	itemStateChanged(ItemEvent)
Mouse	MouseListener	mousePressed(MouseEvent) ...
MouseMotion	MouseMotionListener	mouseDragged(MouseEvent) ...
Key	KeyListener	keyPressed(KeyEvent)
Focus	FocusListener	focusGained(FocusEvent)
Adjustment	AdjustmentListener	adjustmentValueChanged(AdjustmentEvent)
Component	ComponentListener	componentMoved(ComponentEvent) ...
Window	WindowListener	windowClosing(WindowEvent) ...
Container	ContainerListener	componentAdded(ContainerEvent) ...
Text	TextListener	textValueChanged(TextEvent) ...



多重监听器

- ▶ 一般情况下，事件源可以产生多种不同类型的事件，因而可以注册（触发）多种不同类型的监听器。
- ▶ 一个事件源组件上可以注册多个监听器，针对同一个事件源的同一种事件也可以注册多个监听器，一个监听器可以被注册到多个不同的事件源上。



多重监听器示例

CODE ▶ TestActionEvent2.java

```
1  import java.awt.*;
2  import java.awt.event.*;
3  public class TestActionEvent2 {
4      public static void main(String args[]) {
5          Frame f = new Frame("Test");
6          Button b1 = new Button("Start");
7          Button b2 = new Button("Stop");
8          Monitor2 bh = new Monitor2();
9          b1.addActionListener(bh);
10         b2.addActionListener(bh);
11         //b2.setActionCommand("Game Over!");
12         f.add(b1, "North");
13         f.add(b2, "Center");
14         f.pack();
15         f.setVisible(true);
16     }
17 }
18 class Monitor2 implements ActionListener {
19     public void actionPerformed(ActionEvent e) {
20         System.out.println(e.getActionCommand());
21     }
22 }
```



多重监听器示例

CODE ♦ TestMultiActionEvent.java

```
1 public class TestMultiActionEvent implements MouseMotionListener, MouseListener {
2     ...
3     Frame f = new Frame("多重事件监听");
4     f.addMouseMotionListener(this);
5     f.addMouseListener(this);
6     ...
7
8     public void mouseDragged(MouseEvent e) {
9         String s = "鼠标移动位置_(" + e.getX() + ",_" + e.getY() + ")";
10        tf.setText("s");
11    }
12
13    public void mouseEntered(MouseEvent e) {
14        tf.setText("鼠标进入窗体");
15    }
16    ... // 其他接口方法的实现
17 }
```



事件适配器

- ▶ 当创建事件监听器类时，需要实现相应的监听器接口，而实现类中又必须重写/实现接口中的每一个抽象方法，这在 GUI 事件处理过程中经常会成为一种负担。
- ▶ 事件适配器使用了设计模式中的缺省适配器（Default Adapter）。事件适配器类（Adapter）是针对大多数事件监听器接口定义的相应的实现类，适配器类实现了相应监听器接口中所有的方法，但不做任何事。

具体事件适配器类定义

```
1 package java.awt.event;  
2 public abstract class MouseMotionAdapter implements MouseMotionListener {  
3     public void mouseDragged(MouseEvent e) {}  
4     public void mouseMoved(MouseEvent e) {}  
5 }
```



事件适配器

❖ 常用的 GUI 事件适配器

监听器接口	对应适配器类	说明
MouseListener	MouseAdapter	鼠标事件适配器
MouseMotionListener	MouseMotionAdapter	鼠标运动事件适配器
WindowListener	WindowAdapter	窗口事件适配器
FocusListener	FocusAdapter	焦点事件适配器
KeyListener	KeyAdapter	键盘事件适配器
ComponentListener	ComponentAdapter	组件事件适配器
ContainerListener	ContainerAdapter	容器事件适配器



事件适配器示例

CODE ♦ MyAdapter.java

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

CODE ♦ TestAdapter.java

```
1 import java.awt.Frame;  
2 public class TestAdapter {  
3     Frame f = new Frame("Java GUI");  
4     f.setSize(150, 150);  
5     MyAdapter m = new MyAdapter();  
6     f.addWindowListener(m);  
7     f.setVisible(true);  
8 }
```



内部类和匿名类在 GUI 事件处理中的应用

监听器类中封装的业务逻辑具有非常强的针对性，一般没有重用价值，因此经常采用内部类或匿名类的形式来实现。

在 GUI 事件处理中使用内部类

```
1 public class TestInner {
2     // ... ...
3     private class InnerMonitor implements MouseMotionListener, MouseListener { // 内部类
4         public void mouseDragged(MouseEvent e) {
5             String s = "鼠标移动到位置" + e.getX() + "," + e.getY() + " ";
6             tf.setText(s);
7         }
8         public void mouseEntered(MouseEvent e) {
9             String s = "鼠标已进入窗体";
10            tf.setText(s);
11        }
12        public void mouseExited(MouseEvent e) {
13            String s = "鼠标已移出窗体";
14            tf.setText(s);
15        }
16        public void mouseMoved(MouseEvent e) {}
17        public void mousePressed(MouseEvent e) {}
18        public void mouseClicked(MouseEvent e) {}
19        public void mouseReleased(MouseEvent e) {}
20    }
21    // ... ...
22 }
```



内部类和匿名类在 GUI 事件处理中的应用

如果不使用内部类实现上述代码？需要如何修改事件监听器类？

使用外部类

```
1 public class TestOuter {
2     ...
3     OuterMonitor om = new OutMonitor(tf); // 将组件作为参数传递
4 }

6 class OuterMonitor implements MouseMotionListener, MouseListener { // 外部类
7     public OuterMonitor(TextField tf) { // 接收需要该类修改的外部组件对象
8         this.tf = tf;
9     }
10    public void mouseDragged(MouseEvent e) {
11        String s = "鼠标移动到位置[" + e.getX() + ", " + e.getY() + "]";
12        tf.setText(s);
13    }
14    public void mouseEntered(MouseEvent e) {
15        String s = "鼠标已进入窗体";
16        tf.setText(s);
17    }
18    public void mouseExited(MouseEvent e) {
19        String s = "鼠标已移出窗体";
20        tf.setText(s);
21    }
22    public void mouseMoved(MouseEvent e) {}
23    // ... ...
24 }
25 // ... ...
26 }
```



内部类和匿名类在 GUI 事件处理中的应用

使用匿名类

```
1  // ... ...
2  f.addWindowListener(new WindowAdapter(){ // 匿名类
3      public void windowClosing(WindowEvent e) {
4          System.exit(0);
5      }
6  });
7  // ... ...
```



接下来...

Java GUI 设计

GUI 事件处理

AWT 常用组件和视觉控制

Applet

Swing 概述

Swing 典型组件



AWT 常用组件和视觉控制

本节作为课下实验内容，自行完成。



接下来...

Java GUI 设计

GUI 事件处理

AWT 常用组件和视觉控制

Applet

Swing 概述

Swing 典型组件



Applet

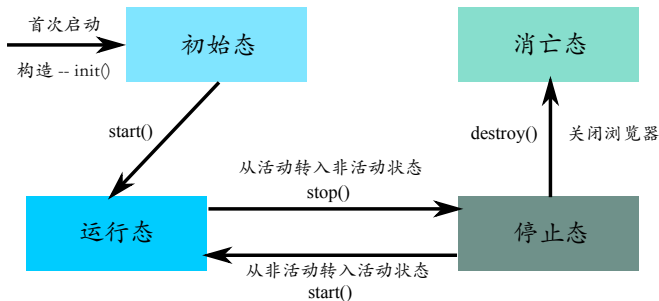
Applet 也称 Java 小程序，在支持 Java 的浏览器环境中运行，通常用于在网页中实现嵌入图片、播放声音等多媒体功能，或添加其他的客户端处理逻辑（如网络计算器）。

严格的说，Applet 是能够嵌入到 HTML 页面中，且可以通过 Web 浏览器下载并执行的一种 Java 程序。

目前，该项技术在新项目中已经很少使用。



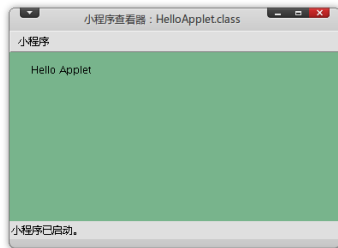
Applet 示例



Applet 示例

CODE HelloApplet.java

```
1 import java.applet.Applet;  
2 import java.awt.Color;  
3 import java.awt.Graphics;  
  
5 public class HelloApplet extends Applet {  
6     String text;  
7     public void init() {  
8         text = "Hello Applet";  
9         this.setBackground(new Color(120, 180, 140));  
10    }  
11    public void paint(Graphics g) {  
12        g.drawString(text, 25, 25);  
13    }  
14 }
```



CODE test.html

```
1 <html>  
2 <applet code="HelloApplet.class" width=200 height=150> </applet>  
3 </html>
```

```
1 >appletviewer test.html
```



接下来...

Java GUI 设计

GUI 事件处理

AWT 常用组件和视觉控制

Applet

Swing 概述

Swing 典型组件



Swing 概述

❖ Swing 与 AWT 的关系

Swing 是建立在 AWT 基础上的一种增强型 Java GUI 组件和工具集

- ▶ 使用轻量组件以替代 AWT 中的绝大部分重量组件。
- ▶ 提供 AWT 所缺少的一些附件组件和观感控制机制。
- ▶ 提供更好的平台无关性。

❖ 相关基本概念

Java 基础类库 (Java Foundation Classes, JFC)

Java 基础类库是用于图形用户界面开发的 Java API 集，具体包括 AWT、2D API、Swing 组件和 Accessibility API。



Swing 概述

重量组件 (Heavy-Weight Components)

- ▶ 重量组件通过委托对等组件（对等组件指底层平台，如 Windows 操作系统的用户界面组件）来完成具体工作，包括组件的绘制和事件响应。AWT 中的组件均为重量组件，或者说，AWT 组件只是对本地对等组件的封装。
- ▶ 开销大、效率低、无法实现组件的“透明”效果。



Swing 概述

轻量组件 (Light-Weight Components)

- ▶ 轻量组件不存在本地对等组件，通过 Java 绘图技术在其所在的容器窗口中绘图得到。
- ▶ 能够实现组件的透明效果，能够做到不同平台上的一致表现。
- ▶ 组件绘制和事件处理机制的开销小。
- ▶ 轻量组件最终需要包含在一个重量容器中。因此，Swing 组件中的几个顶层容器（如 JFrame、JDialog 和 JApplet）采用了重量组件，其余的均为轻量组件。
- ▶ 不建议轻重组件混用。



Swing 概述

Swing 基于 AWT，Swing 组件类均继承了 AWT 中的容器类 `java.awt.Container`。`JComponent` 类是除了几个顶层容器（`JFrame`、`JDialog`）之外所有 Swing 组件的父类，其继承层次如下。

```
java.lang.Object
|
+----java.awt.Component
      |
      +----java.awt.Container
            |
            +----javax.swing.JComponent
```



Swing 概述

可视化组件 (Visual Component)

- ▶ 和 AWT 组件类似，Swing 组件也分为可视化和非可视化组件。
- ▶ 可视化组件为能够显示特定形状、颜色和尺寸的组件；非可视化组件也称支持类，如布局管理等。
- ▶ Swing 可视化组件类名均以 J 开头。



接下来...

Java GUI 设计

GUI 事件处理

AWT 常用组件和视觉控制

Applet

Swing 概述

Swing 典型组件



JFrame

1. JFrame 继承并扩充了 `java.awt.Frame` 类。
JFrame 不再是一个单一容器，而是由相互间存在包含关系的多个不同容器面板（`JRootPane`, `GlassPane`, `LayeredPane`, `ContentPane`）组成，我们实际上只使用其中的内容面板（`ContentPane`）。
2. JFrame 实现了 `javax.swing.WindowConstants` 接口，该接口中定义了用于控制窗口关闭操作的整型常量，包括：
 - ▶ `DO_NOTHING_ON_CLOSE`
 - ▶ `HIDE_ON_CLOSE`
 - ▶ `DISPOSE_ON_CLOSE`
 - ▶ `EXIT_ON_CLOSE`



JFrame 示例

CODE TestJFrame.java

```
1  import java.awt.Color;
2  import java.awt.Container;
3  import java.awt.FlowLayout;
4  import javax.swing.JFrame;
5  import javax.swing.JLabel;

7  public class TestJFrame {
8      public static void main(String[] args){
9          JFrame jf = new JFrame("My_Test");
10         Container c = jf.getContentPane();
11         c.setLayout(new FlowLayout(FlowLayout.LEFT, 20, 20));
12         JLabel greet = new JLabel("Hello, World!");
13         JLabel bye = new JLabel("Bye, World!");
14         bye.setBackground(Color.BLUE);
15         bye.setOpaque(true); // 设置为不透明
16         c.add(greet);
17         c.add(bye);
18         c.setBackground(Color.GREEN);
19         jf.setSize(400, 300);
20         jf.setLocation(400, 200);
21         jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); // 等同于显示注册 WindowListener
22         jf.setVisible(true);
23     }
24 }
```



JFrame

为了方便开发，从 JDK5.0 开始 JFrame 类重写了其 add()、remove() 和 setLayout() 方法，这些重写后的方法将针对 JFrame 的添加组件、移除组件和设置布局管理器等操作自动转发给其内容面板 contentPane，以实现对 contentPane 的直接控制。

对上述代码的改写：

```
1  JFrame jf = new JFrame("My_Test");  
2  jf.setLayout(new FlowLayout(FlowLayout.LEFT, 20, 20));  
3  JLabel greet = new JLabel("Hello, World!");  
4  jf.add(greet);  
5  jf.getContentPane().setBackground(Color.GREEN); // 注意
```

注意：setBackground() 方法在 JFrame 类中并没有进行重写，因此不会将针对 JFrame 的颜色设置操作自动转发到其内容面板。



Swing 按钮、菜单和工具条

► JButton

能够实现更复杂的显示效果，例如可以使用图片作为按钮标签、设置快捷键和添加工具提示信息。

► 菜单

同样分为菜单条、菜单和菜单项（JMenuBar、JMenu、JMenuItem），用法与 AWT 完全相同。

► 工具条

工具条是用于显示常见组件的条形容器，一般用法是向工具条中添加一系列图标形式的按钮，并将之置于窗口上方边缘。例如，BorderLayout 布局的北部区域，对于大多数的外观，用户可以用鼠标直接将工具栏拖到 BorderLayout 布局的其他未添加组件的边缘区域，如西部或东部，也可以将之拖出到单独的窗口中显示。



Swing 按钮、菜单和工具条

CODE ♦ TestSwing.java

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

public class TestSwing implements ActionListener {
    public static void main(String[] args) {
        new TestSwing().createUI();
    }

    public void createUI() {
        JFrame jf = new JFrame("Test Swing");
        JMenuBar jmb = new JMenuBar();
        JMenu menu_file = new JMenu("File");
        JMenu menu_help = new JMenu("Help");
        JMenuItem mi_new = new JMenuItem("New");
        JMenuItem mi_open = new JMenuItem("Open");
        JMenuItem mi_save = new JMenuItem("Save");
        mi_new.addActionListener(this);
        mi_open.addActionListener(this);
        mi_save.addActionListener(this);
```



Swing 按钮、菜单和工具条

```
mi_new.setMnemonic('N');  
mi_open.setMnemonic('O');  
mi_save.setMnemonic('S');  
menu_file.setMnemonic('F');  
menu_help.setMnemonic('h');
```

```
menu_file.add(mi_new);  
menu_file.add(mi_open);  
menu_file.add(mi_save);
```

```
jmb.add(menu_file);  
jmb.add(menu_help);
```

```
JToolBar jtb = new JToolBar();  
JButton button_new = new JButton("NEW");  
JButton button_open = new JButton("OPEN");  
JButton button_save = new JButton("SAVE");
```

```
button_new.setActionCommand("New");  
button_open.setActionCommand("Open");  
button_save.setActionCommand("Save");
```



Swing 按钮、菜单和工具条

```
button_new.setToolTipText("Create a new file");  
button_open.setToolTipText("Open the file");  
button_save.setToolTipText("Save the file");
```

```
button_new.addActionListener(this);  
button_open.addActionListener(this);  
button_save.addActionListener(this);
```

```
jtb.add(button_new);  
jtb.add(button_open);  
jtb.add(button_save);
```

```
JPanel jp = new JPanel();  
JButton button_start = new JButton("Start");  
JButton button_stop = new JButton("Stop");  
button_start.addActionListener(this);  
button_stop.addActionListener(this);  
jp.add(button_start);  
jp.add(button_stop);
```



Swing 按钮、菜单和工具条

```
jf.setJMenuBar(jmb); // Menu 不需要使用 add() 方法添加  
jf.add(jtb, "North");  
jf.add(jp, "South");
```

```
jf.setSize(600, 400);  
jf.setLocation(400, 200);  
jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
jf.setVisible(true);
```

```
}
```

```
@Override
```

```
public void actionPerformed(ActionEvent e) {  
    System.out.println(e.getActionCommand());
```

```
}
```

```
}
```



标准对话框

使用标准对话框（JOptionPane）可以实现程序与用户的便捷交互，如向用户发送错误通知、警告/确认用户操作、接收用户输入或选择的简单信息等。

CODE ▶ 示例

```
@Override
public void actionPerformed(ActionEvent e) {
    String s = e.getActionCommand();
    if (s.equals("Error")) {
        JOptionPane.showMessageDialog(null, "这是一个错误提示对话框", "错误提示",
            JOptionPane.ERROR_MESSAGE);
    } else if (s.equals("Confirm Quit")) {
        int result = JOptionPane.showConfirmDialog(null, "真的要退出程序么？",
            "请确认退出", JOptionPane.YES_NO_OPTION);
        if (result == JOptionPane.OK_OPTION) {
            System.exit(0);
        }
    }
}
```



标准对话框

```
} else if (s.equals("Warning")) {  
    Object[] options = { "继续", "撤销" };  
    int result = JOptionPane.showOptionDialog(null, "本操作可能导致数据丢失",  
        "警告", JOptionPane.DEFAULT_OPTION,  
        JOptionPane.WARNING_MESSAGE, null, options, options[0]);  
    if (result == 0)  
        System.out.println("继续操作");  
} else if (s.equals("Input")) {  
    String name = JOptionPane.showInputDialog("请输入姓名: ");  
    if (name != null)  
        System.out.println("输入的姓名为" + name);  
} else if (s.equals("Choice")) {  
    Object[] possibleValues = { "体育", "政治", "经济", "文化" };  
    Object selectedValue = JOptionPane.showInputDialog(null,  
        "Choice one", "Input", JOptionPane.INFORMATION_MESSAGE,  
        null, possibleValues, possibleValues[0]);  
    String result = (String) selectedValue;  
    if (result != null)  
        System.out.println("你的选择是: " + result);  
}  
}
```



表格和树

► **javax.swing.JTable**

用于以传统的表格形式来显示数据，通过注册监听器的方式关联响应的处理逻辑。

- 表头：标题行，给出每一列（字段）的名称。
- 表体：由多行多列、规则矩阵形式的单元格组成，真正的数据信息则显示在每个单元格中。


► **javax.swing.JTree**

以树状结构分层次显示数据信息，例如操作系统提供的资源管理器。



表格示例

❖ 表格



编号	姓名	性别	年龄	电话
1	张三	男	18	010.82607080
2	李四	女	24	010.82607080
3	王五	男	30	010.82607080
4	赵六	女	24	010.82607080
5	任七	男	32	010.82607080

```

1  import java.awt.event.WindowAdapter;
2  import java.awt.event.WindowEvent;
3  import javax.swing.JFrame;
4  import javax.swing.JScrollPane;
5  import javax.swing.JTable;
6  public class TableExample {
7      public static void main(String[] args) {
8          JFrame myFrame = new JFrame("Table Example");
9          Object data[][] = {
10             {1, "张三", "男", "18", "010.82607080"},
11             {2, "李四", "女", "24", "010.82607080"},
12             {3, "王五", "男", "30", "010.82607080"},
13             // ...
14         };
15         String columnNames[] = {
16             "编号", "姓名", "性别", "年龄", "电话";
17         };
18         JTable table = new JTable(data, columnNames);
19         table.setAutoResizeMode(JTable.AUTO_RESIZE_OFF);
20         JScrollPane pane = new JScrollPane(table);
21         myFrame.add("Center", pane);
22         myFrame.setSize(450, 250);
23         myFrame.addWindowListener(new WindowAdapter() {
24             public void windowClosing(WindowEvent e) {
25                 System.exit(0);
26             }
27         });
28         myFrame.setVisible(true);
29     }

```



表格其他参考例程

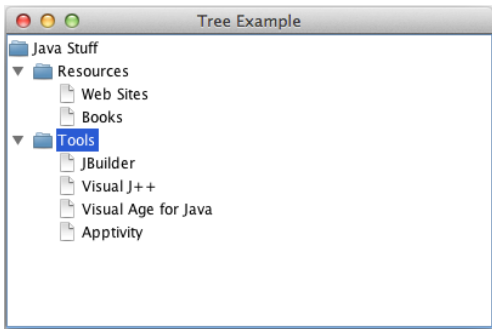
1. A Simple Interactive JTable Tutorial

<http://www.javalobby.org/articles/jtable/>



树示例

❖ 树



树示例

CODE ♦ TreeExample.java

```
1 public class TreeExample {
2     public static DefaultMutableTreeNode createNodes() {
3         DefaultMutableTreeNode rootNode = new DefaultMutableTreeNode("Java_Stuff");
4         DefaultMutableTreeNode resources = new DefaultMutableTreeNode("Resources");
5         DefaultMutableTreeNode tools = new DefaultMutableTreeNode("Tools");
6         rootNode.add(resources);
7         rootNode.add(tools);
8         DefaultMutableTreeNode webSites = new DefaultMutableTreeNode("Web_Sites");
9         DefaultMutableTreeNode books = new DefaultMutableTreeNode("Books");
10        resources.add(webSites);
11        resources.add(books);
12        tools.add(new DefaultMutableTreeNode("JBuilder"));
13        tools.add(new DefaultMutableTreeNode("Visual_J++"));
14        return rootNode;
15    }
16    public static void main(String[] args) {
17        JFrame myFrame = new JFrame("Tree_Example");
18        DefaultMutableTreeNode rootNode = createNodes();
19        JTree tree = new JTree(rootNode);
20        tree.setRootVisible(true);
21        JScrollPane pane = new JScrollPane();
22        pane.setViewPortView(tree);
23        myFrame.add(pane, "Center");
24        myFrame.setSize(400, 250);
25        myFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
26        myFrame.setVisible(true);
27    }
28 }
```



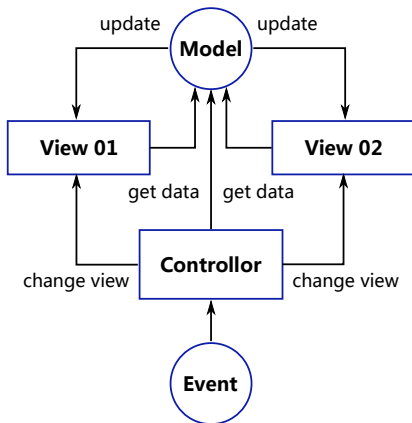
JTable 和 JTree 的 MVC 模式

JTable 和 JTree 采用了相对独立的方式向组件提供要显示的数据，即当显示/处理的数据结构较复杂时，将 GUI 组件结构分为相对独立的**模型**、**视图**、**控制器**三个模块，模块间存在专门的分工和协作关系。

1. **模型**（Model） 维护数据并提供数据访问方法，即数据和数据的处理逻辑。
2. **视图**（View） 绘制模型的视觉表现，即显示数据。视图就是用户能够看到并与之进行交互的用户界面。
3. **控制器**（Controllor） 负责处理事件或者说程序的流程控制，接受用户输入，并调用/操控模型和视图以实现用户需求。



MVC 作用原理



定时器

`javax.swing.Timer` 提供了定时器功能，用于在指定的时间延迟之后触发 `ActionEvent` 事件，以执行所需的处理逻辑。

具体的做法是：首先创建 `Timer` 对象，并在其上注册一个或多个 `ActionListener` 类型的监听器，在监听器事件处理方法 `actionPerformed()` 中应以实现给出要延时执行的任务代码，然后调用 `Timer` 对象的 `start()` 方法启动定时器即可。

❖ 相关方法

- ▶ `setRepeats()` 设置计时器的动作。
- ▶ `setInitialDelay()` 设置首次延迟时间。
- ▶ `start()` 开始计时器。
- ▶ `stop()` 停止计时器。
- ▶ `restart()` 恢复计时器。



THE END

wxd2870@163.com

