1、课程名称：GUI 编程

2、知识点

2.1、本节预计讲解的知识点

AWT
组件和容器
布局管理器
事件处理
事件源、事件监听器、事件类型
事件监听器接口和事件适配器
匿名类在 Java 事件处理中的应用
了解 AWT 常用组件
Java 图形
Graphics 类和 paint 方法
了解 Swing 常用组件的使用

3、具体内容

java 的 GUI 编程（Graphic User Interface，图形用户接口），是在它的抽象窗口工具箱（Abstract Window Toolkit，AWT）上实现的，java.awt 是 AWT 的工具类库，其中包括了丰富的图形、用户界面元件和布局管理器的支持。

GUI 主要用在两个地方：
Application；
Applet。

1）GUI 界面：
用户与程序之间交互的一个控制面板，其内包含有菜单，控件（或组件），容器并能响应用户的事件。
现在有各种各样的窗口系统，不同的窗口系统提供给程序设计的程序库是大不一样的，例如，基于 Windows 的 SDK，和基于 Unix 平台的 X Windows 的 Xlib。
为了使程序能在不同的窗口系统下运行，Java 提出了“抽象窗口系统”的概念，提供了 AWT（抽象窗口工具箱），使得 java 能够在不同的窗口系统下运行。

2）Java 中的 GUI 实现方式：
采用 AWT（抽象窗口工具集）从而可使 GUI 适用于不同 OS 的环境。
特点如下：
① 其具体实现由目标平台下的 OS 来解释，从而导致 Java GUI 在不同平台下会出现不同的运行效果（窗口外观、字体等的显示效果会发生变化）。

② 组件在设计时不应采用绝对定位，而应采用布局管理器来实现相对定位，以达到与平台及设备无关。

3）新增的 Swing GUI 组件

AWT 组件以及事件响应不及微软的 SDK 丰富（因为有些 OS 平台无微软的 Windows 组件），Sun 在 Java2 中新增了 Swing GUI 组件。但是，AWT 比较简单，功能也能满足大多数界面需求，特别在 Java Applet 的设计中受到了普遍的应用。同时，这个讨论也为我们进一步研究 Swing GUI 组件打下了比较扎实的基础。

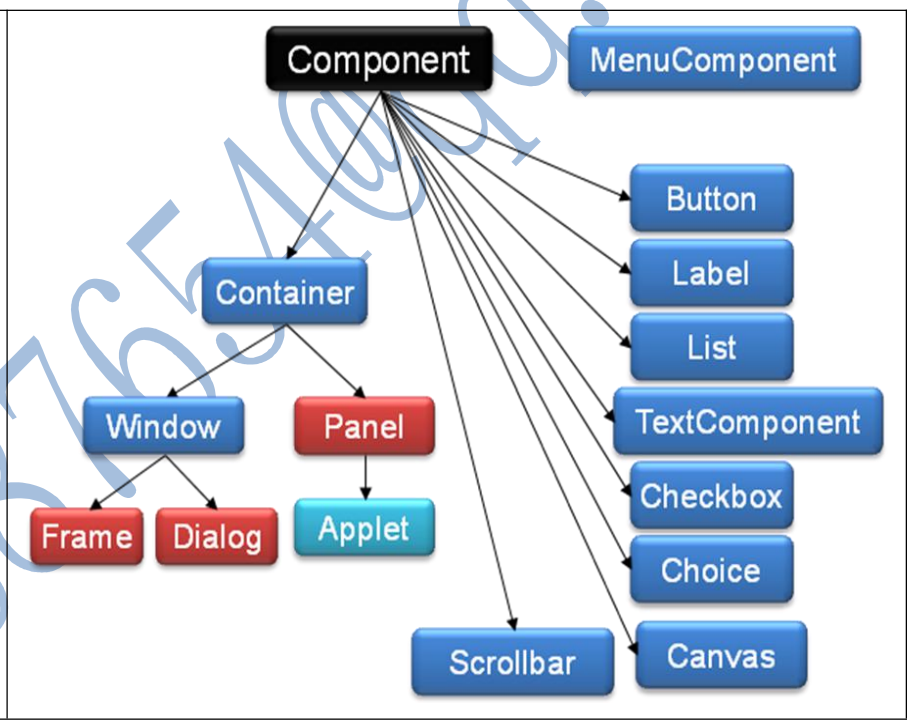
3.1、抽象窗口工具集 AWT

AWT(Abstract Window Toolkit)中包括了多种类和接口，用于在 Java Application 中进行 GUI(Graphical User Interface 图形用户界面)编程 GUI 的 各种元素(如:容器、按钮、文本框等)由 java 类来实现。
使用 AWT 所涉及的类一般在 java.awt 包及其子包中。

AWT 主要类的继承关系

java.awt.Component
- java.awt.Container
  - java.awt.Panel
    - java.applet.Applet
  - java.awt.Window
    - java.awt.Dialog
    - java.awt.Frame
- java.awt.Button
- java.awt.Checkbox
- java.awt.Label
- java.awt.List
- java.awt.TextComponent

java.awt.MenuComponent
- java.awt.MenuBar
- java.awt.MenuItem
  - java.awt.Menu



3.1.1、Component 和 Container

- Java 的图形用户界面的最基本组成部分是组件(Component)，组件是一个可以以图形化的方式显示在屏幕上并能与用户进行交互的对象，例如一个按钮，一个标签等。
  一般的组件不能独立地显示出来，必须将它放在某一容器中才可以显示出来。

- 容器(Container)实际上是 Component 的子类，容器类对象具有组件的所有性质，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

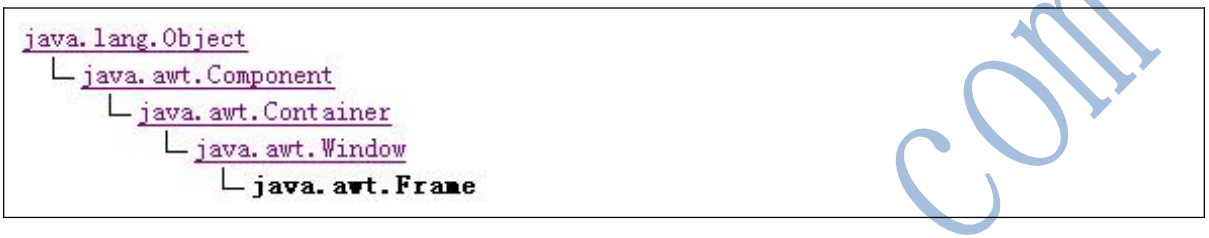
- 两种主要的容器类型

Window: 可自由停泊的顶级窗口

Panel: 可作为容器容纳其它组件，但不能独立存在，必须被添加到其它容器中(如 Window 或 Applet)

3.1.2、Frame 类

- Frame 类是抽象类 Window 的子类，Frame 对象显示效果是一个“窗体”，带有标题和尺寸重置角标。
- Frame 类的继承层次



- Frame 的常用构造方法：
  - Frame()
  - Frame(String s) 创建指定标题的 Frame 对象

Frame 常用方法	说明
setVisible(boolean b)	Frame 默认初始化为不可见的
setSize(int width, int height)	设置窗体的大小
setLocation(int x, int y)	设置窗体的位置，x、y 是左上角的坐标
setBounds(int x, int y, int width, int height)	设置位置、宽度和高度
setTitle(String name)	设置窗体的标题 getTitle()
setResizable(boolean b)	设置是否可以调整大小
setBackground(Color c)	设置背景颜色，参数为 Color 对象
setLayout(LayoutManager mgr)	设置此容器的布局管理器，默认的布局管理器是 BorderLayout
dispose()	释放由此窗体及其拥有的所有子组件所使用的所有本机屏幕资源

示例：

```
import java.awt.Color;
import java.awt.Frame;
public class TestFrame {
    public static void main(String[] args) {
        MyFrame frame = new MyFrame();
    }
}
class MyFrame extends Frame{
    public MyFrame(){
        this.setTitle("第一个窗体");
        this.setSize(200, 300);
        this.setLocation(200, 200);
```

```
        this.setBackground(Color.ORANGE);
        this.setResizable(false);
        this.setVisible(true);
    }
}
```

3.1.3、Panel 类

- Panel 对象可以看成可以容纳其它组件的空间
- Panel 不能独立存在，依赖于其它容器
- Panel 对象可以拥有自己的布局管理器
- Panel 类的继承层次

```
java.lang.Object
  └java.awt.Component
      └java.awt.Container
          └java.awt.Panel
```

- Panel 的构造方法为：
  - Panel()　　//使用默认的 FlowLayout 布局管理器创建新面板
  - Pane(LayoutManager layout)　　//创建具有指定布局管理器的新面板

Panel 类的常用方法	描述
setSize(int width, int height)	设置大小
setLocation(int x, int y)	设置窗体的位置，x、y 是左上角的坐标
setBounds(int x, int y, int width, int height)	设置位置、宽度和高度
setBackground(Color c)	设置背景颜色，参数为 Color 对象
setLayout(LayoutManager mgr)	设置布局管理器

示例：

```
import java.awt.Color;
import java.awt.Frame;
import java.awt.Panel;
public class TestPanel {
    public static void main(String[] args) {
        Frame frame = new Frame();
        frame.setLayout(null);
        frame.setBounds(300,300,500,500);
        frame.setBackground(Color.BLUE);

        Panel panel = new Panel();
        panel.setBounds(100,100,300,300);
        panel.setBackground(new Color(204,204,255));
        frame.add(panel);
```

```
        frame.setVisible(true);
    }
}
```

3.1.4、Toolkit 类

- **Toolkit 抽象类是用于将各种组件绑定到本地系统的工具包。**
  - getDefalutToolkit()　　静态方法可以得到一个 Toolkit 的子类实例
  - Dimension getScreenSize()　　把屏幕尺寸作为一个 Dimension 实例返回
  - Image getImage(URL url)　　返回一幅图像，该图像从指定文件中获取像素数据，图像格式可以是 GIF、JPEG 或 PNG

3.2、布局管理器

- 为了使我们生成的图形用户界面具有良好的平台无关性，Java 语言中，提供了布局管理器这个工具来管理组件在容器中的布局，而不是直接设置组件的位置和大小。
- 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。
- 调用 Container 的 setLayout 方法可以设置容器的布局管理对象。

AWT 提供了 5 种布局管理器类：

- **FlowLayout**
- **BorderLayout**
- **GridLayout**
- **CardLayout**
- **GridBagLayout**

3.2.1、FlowLayout 布局管理器

- FlowLayout 是 Panel 类的默认布局管理器
  - FlowLayout 布局对组件逐行定位，行内从左到右，一行排满后换行
  - 不改变组件的大小，按组件原有尺寸显示组件，可在构造方法中设置不同的组件间距、行距及对齐方式
- FlawLayout 布局默认对齐方式为居中对齐

示例：

```
import java.awt.Button;
import java.awt.FlowLayout;
import java.awt.Frame;
public class TestFlowLayout {
    public static void main(String[] args) {
        Frame f = new Frame("FlowLayout 布局管理器");
        f.setLayout(new FlowLayout());
```

```
            f.add(new Button("按钮"));
            f.add(new Button("按钮"));
            f.add(new Button("按钮"));
            f.add(new Button("按钮"));
            f.add(new Button("按钮"));
            f.setSize(300, 300);
            f.setLocation(400, 300);
            f.setVisible(true);
        }
}
```

FlowLayout 的构造方法

- new FlowLayout(FlowLayout.RIGHT,20,40);
  - 右对齐，组件之间水平间距 20 个像素，竖直间距 40 个像素
- new FlowLayout(FlowLayout.LEFT);
  - 左对齐，水平和竖直间距为缺省值（5 像素）
- new FlowLayout();
  - 使用缺省的居中对齐方式，水平和竖直间距为缺省值（5 像素）

3.2.2、BorderLayout 布局管理器

BorderLayout 是 Frame 类的默认布局管理器

- **BorderLayout 将整个容器的布局划分成**
  - 东（EAST）
  - 西（WEST）
  - 南（SOUTH）
  - 北（NORTH）
  - 中（CENTER）

  五个区域，组件只能被添加到指定的区域

如果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覆盖

- **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
  - 东、西两个区域只能在垂直方向缩放
  - 中部可在两个方向上缩放

示例：

```
import java.awt.*;
public class TestBorderLayout {
    public static void main(String args[]) {
        Frame f = new Frame("BorderLayout 布局管理器");
        Button bn = new Button("北(North)");
```

```
        Button bs = new Button("南(South)");
        Button bw = new Button("西(West)");
        Button be = new Button("东(East)");
        Button bc = new Button("中(Center)");
        f.add(bn, BorderLayout.NORTH);
        f.add(bs, BorderLayout.SOUTH);
        f.add(bw, BorderLayout.WEST);
        f.add(be, BorderLayout.EAST);
        f.add(bc, BorderLayout.CENTER);
        f.setSize(300, 300);
        f.setVisible(true);
    }
}
```

3.2.3、GridLayout 布局管理器

- GridLayout 布局管理器将布局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，先从左到右添满一行后换行，再从上到下.
- 在 GridLayout 构造方法中指定分割的行数和列数：
  - 如：new GridLayout(3,3)

示例：

```
import java.awt.*;
public class TestGridLayout {
    public static void main(String args[]) {
        Frame f = new Frame("GridLayout 布局管理器");
        f.setLayout(new GridLayout(3, 3));
        for(int i = 1; i <= 9; i++){
            f.add(new Button("b" + i));
        }
        f.setSize(300, 300);
        //f.pack();
        f.setVisible(true);
    }
}
```

3.2.4、绝对定位方式的布局

通过 setLayout(null)取消该容器的布局管理器，容器内的组件都采用 setBounds(x, y, width,height)来绝对定位。

3.2.5、布局管理小结

- Frame 是一个顶级窗口。Frame 的缺省布局管理器为 BorderLayout。
- Panel 无法单独显示，必须添加到某个容器中。
  - Panel 的缺省布局管理器为 FlowLayout。
  - 当把 Panel 作为一个组件添加到某个容器中后，该 Panel 仍然可以有自己的布局管理器。
- 容器中的布局管理器负责各个组件的大小和位置，因此用户无法在这种情况下设置组件的这些属性。如果试图使用 Java 语言提供的 setLocation()，setSize()，setBounds()等方法，则都会被布局管理器覆盖。
- 如果用户确实需要亲自设置组件大小或位置，则应取消该容器的布局管理器，方法为：setLayout(null)；

3.3、事件处理

事件源、事件监听器、事件类型
事件监听器接口和事件适配器

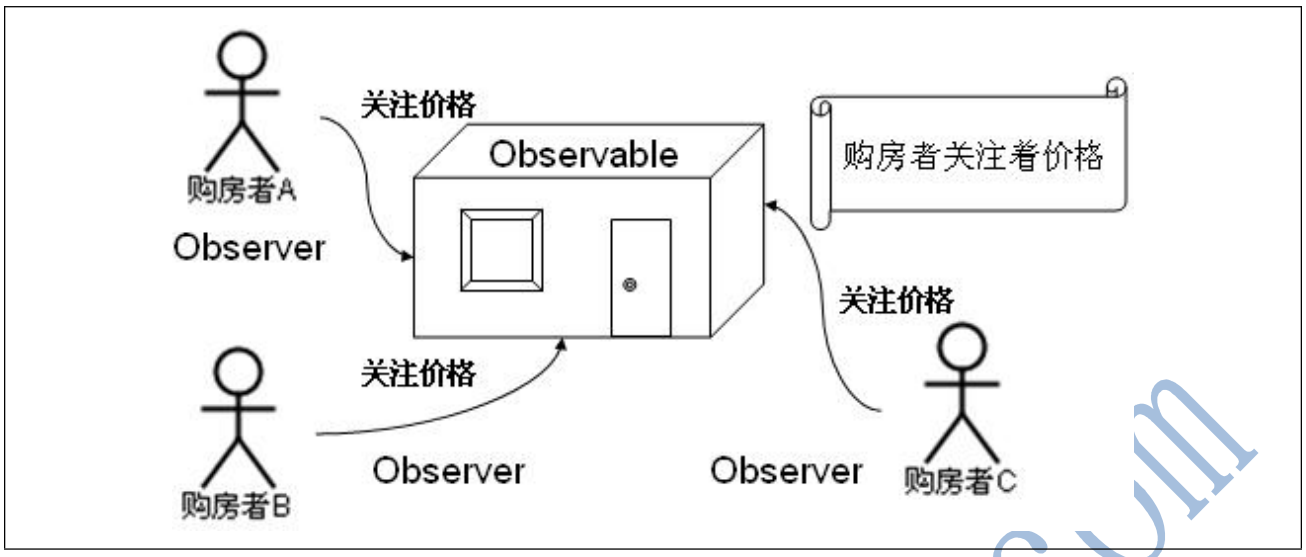
事件处理中的三个概念

- 事件(Event)：用户对组件的一个操作，称之为一个事件
- 事件源(Event source) – 产生事件的组件
- 事件处理方法(Event handler) – 能够接收、解析和处理事件类对象、实现和用户交互的方法
  - Java 程序对事件处理的方法是放在一个类对象中的,这个类对象就是**事件监听器**。
- 必须将一个事件监听器对象同某个事件源的某种事件进行关联，这样，当某个事件源上发生了某种事件后，关联的事件监听器对象中的有关代码才会被执行。这个过程称为向事件源注册事件监听器。

3.3.1、观察者设计模式（理解）

观察者模式定义：简单地说，观察者模式定义了一个一对多的依赖关系，让一个或多个观察者对象监察一个主题对象。这样一个主题对象在状态上的变化能够通知所有的依赖于此对象的那些观察者对象，使这些观察者对象能够自动更新。

观察者设计实际上从生活中很好理解。



中国的老百姓最关心房子问题，每次房子价格的上涨或下降，都会引起关注，那么这个时候如果要想实现此类操作，则必须使用观察者设计模式。

房子的价格属于被观察者，每一个购房者都应该表示一个观察者。

所以，如果要想实现观察者设计，则每一个观察者都必须实现 Observer 接口，表示可以观察。此接口定义方法如下：

No.	方法名称	类型	描述
1	void update(Observable o,Object arg)	普通	当被观察者发生改变时调用

所有的被观察对象都必须继承 Observable 类，表示此类的某个内容可以被观察。此类的定义如下：

No.	方法名称	类型	描述
1	public void addObserver(Observer o)	普通	增加观察者
2	protected void setChanged()	普通	观察的内容发生了改变
3	public void notifyObservers()	普通	发生改变之后要通知所有观察者，调用 update 方法

范例：定义房子的类

```
package org.obserdemo;
import java.util.Observable;
public class House extends Observable {
    private float price; // 房子价格
    public House(float price) {
        this.price = price;
    }
    public float getPrice() {
        return price;
    }
    public void setPrice(float price) { // 调用此方法的时候才会更改房子价格
        super.setChanged(); // 告诉java，此处发生改变了
        super.notifyObservers();// 告诉所有的观察者
        this.price = price;
    }
    public String toString() {
        return "房子价格为: " + this.price;
    }
}
```

之后定义观察者的类，此类实现 Observer 接口。

范例：定义观察者

```
package org.obserdemo;
import java.util.Observable;
import java.util.Observer;
public class HousePriceObserver implements Observer {
    private String name;
    public HousePriceObserver(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public void update(Observable o, Object arg) {
        House h = (House) o;
        System.out.println(this.name + "观察到了价格的变化，价格变化为：" + h.getPrice());
    }
}
```

只要一更新房子价格，则会调用 update()方法。

范例：测试

```
package org.obserdemo;
public class TestObserver {
    public static void main(String[] args) {
        House h = new House(1000000);
        HousePriceObserver hpo1 = new HousePriceObserver("张三");
        HousePriceObserver hpo2 = new HousePriceObserver("李四");
        HousePriceObserver hpo3 = new HousePriceObserver("王五");
        h.addObserver(hpo1);// 加入观察者
        h.addObserver(hpo2);// 加入观察者
        h.addObserver(hpo3);// 加入观察者
        System.out.println(h);
        h.setPrice(110000); // 更新价格
    }
}
```

观察者模式的效果

观察者模式的效果有以下的优点：

第一、观察者模式在被观察者和观察者之间建立一个抽象的耦合。被观察者角色所知道的只是一个具体观察者列表，每一个具体观察者都符合一个抽象观察者的接口。被观察者并不认识任何一个具体观察者，它只知道它们都有一个共同的接口。

由于被观察者和观察者没有紧密地耦合在一起，因此它们可以属于不同的抽象化层次。如果被观察者和观察者都被扔到一起，那么这个对象必然跨越抽象化和具体化层次。

第二、观察者模式支持广播通讯。被观察者会向所有的登记过的观察者发出通知

观察者模式有下面的缺点：

第一、如果一个被观察者对象有很多的直接和间接的观察者的话，将所有的观察者都通知到会花费很多时间。

第二、如果在被观察者之间有循环依赖的话，被观察者会触发它们之间进行循环调用，导致系统崩溃。在使用观察者模式是要特别注意这一点。

第三、如果对观察者的通知是通过另外的线程进行异步投递的话，系统必须保证投递是以自恰的方式进行的。

第四、虽然观察者模式可以随时使观察者知道所观察的对象发生了变化，但是观察者模式没有相应的机制使观察者知道所观察的对象是怎么发生变化的。

观察者模式与其它模式的关系

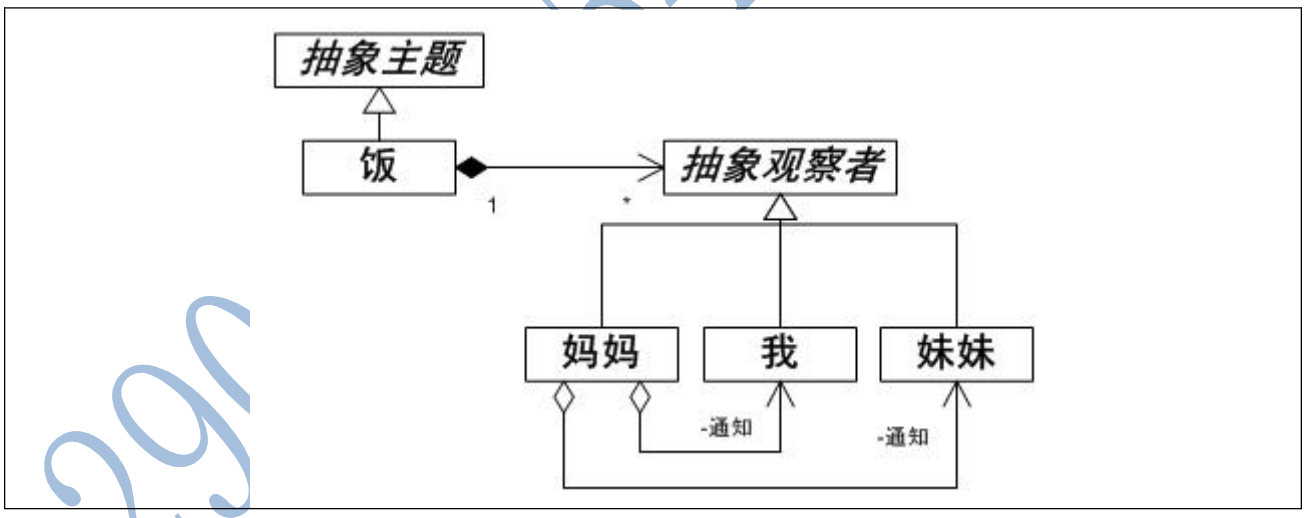
观察者模式使用了备忘录模式(Memento Pattern)暂时将观察者对象存储在被观察者对象里面。

问答题

第一题、我和妹妹跟妈妈说：“妈妈，我和妹妹在院子里玩；饭做好了叫我们一声。”请问这是什么模式？能否给出类图说明？

问答题答案

第一题答案、这是观察者模式。我和妹妹让妈妈告诉我们饭做好了，这样我们就可以来吃饭了。换用较为技术化的语言来说，当系统的主题（饭）发生变化时，就告诉系统的其它部份（观察者们，也就是妈妈、我和妹妹），使其可以调整内部状态（有开始吃饭的准备），并采取相应的行动（吃饭）。



3.3.2、事件处理机制

- 向组件(事件源)注册事件监听器后，事件监听器就与组件建立关联，当组件接受外部作用(事件)时，组件会产生一个相应的事件对象，并把这个对象传给与之关联的事件监听器，事件监听器就会被启动并执行相关的代码来处理该事件。
- 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。

示例：

```
import java.awt.*;
public class TestActionEvent {
    public static void main(String[] args) {
        Frame frame = new Frame();
        frame.setSize(300, 300);
        frame.setVisible(true);
        frame.setLayout(null);
        Button btn = new Button("点我看看!!");
        btn.addActionListener(new Monitor());
        btn.setBounds(110, 130, 80, 30);
        frame.add(btn);
    }
}
//动作事件监听器
class Monitor implements ActionListener{
    public void actionPerformed(ActionEvent e) {
        Button btn = (Button)e.getSource();
        System.out.println("呵呵,你点击了" + btn.getName());
    }
}
```

3.3.3、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) mouseReleased(MouseEvent) mouseEntered(MouseEvent) mouseExited(MouseEvent) mouseClicked(MouseEvent)
Mouse Motion	MouseMotionListener	mouseDragged(MouseEvent) mouseMoved(MouseEvent)
Key	KeyListener	keyPressed(KeyEvent) keyReleased(KeyEvent) keyTyped(KeyEvent)
Focus	FocusListener	focusGained(FocusEvent) focusLost(FocusEvent)
Adjustment	AdjustmentListener	adjustmentValueChanged (AdjustmentEvent)
Component	ComponentListener	componentMoved(ComponentEvent) componentHidden (ComponentEvent)

		componentResized(ComponentEvent) componentShown(ComponentEvent)
Window	WindowListener	windowClosing(WindowEvent) windowOpened(WindowEvent) windowIconified(WindowEvent) windowDeiconified(WindowEvent) windowClosed(WindowEvent) windowActivated(WindowEvent) windowDeactivated(WindowEvent)
Container	ContainerListener	componentAdded(ContainerEvent) componentRemoved(ContainerEvent)
Text	TextListener	textValueChanged(TextEvent)

示例：给窗体注册一个窗体事件监听器，当用户点击了关闭按钮时，把窗体关闭掉。

```
import java.awt.Frame;
import java.awt.event.*;
public class TestCloseFrame {
    public static void main(String[] args) {
        Frame frame = new Frame("关闭窗体事件");
        frame.setSize(300, 300);
        frame.setVisible(true);
        frame.addWindowListener(new MyWindowLinstener());
    }
}
class MyWindowLinstener implements WindowListener{     //窗体事件
    public void windowClosing(WindowEvent e) {
        ((Frame)e.getComponent()).dispose();
        System.exit(0);
    }
    public void windowActivated(WindowEvent e) {}
    public void windowClosed(WindowEvent e) {}
    public void windowDeactivated(WindowEvent e) {}
    public void windowDeiconified(WindowEvent e) {}
    public void windowIconified(WindowEvent e) {}
    public void windowOpened(WindowEvent e) {}
}
```


3.3.4、事件适配器(Event Adapter)

- 为简化编程，针对大多数事件监听器接口定义了相应的实现类----事件适配器类，在适配器类中，实现了相应监听器接口中所有的方法，但不做任何事情。
- 在定义监听器类时就可以继承事件适配器类，并只重写所需要的方法
- **常用的适配器**
  - ComponentAdapter (组件适配器)

- ContainerAdapter (容器适配器)
- FocusAdapter (焦点适配器)
- KeyAdapter(键盘适配器)
- MouseAdapter (鼠标适配器)
- MouseMotionAdapter (鼠标运动适配器)
- WindowAdapter (窗口适配器)

示例：

```
import java.awt.Frame;
import java.awt.event.*;
public class TestCloseFrame2 {
    public static void main(String[] args) {
        Frame frame = new Frame("关闭窗体事件");
        frame.setSize(300, 300);
        frame.setVisible(true);
        frame.setLocation(300, 400);
        frame.addWindowListener(new MyWindowAdapter());
    }
}
class MyWindowAdapter extends WindowAdapter{
    public void windowClosing(WindowEvent e) {
        ((Frame)e.getComponent()).dispose();
        System.exit(0);
    }
}
```

用成员内部类定义事件监听器

示例：

```
import java.awt.Frame;
import java.awt.event.*;
public class TestInnerClassEvent {
    public static void main(String[] args) {
        MyFrame3 frame = new MyFrame3();
    }
}
class MyFrame3 extends Frame {
    public MyFrame3(){
        this.setSize(300, 300);
        this.setVisible(true);
        this.setLocation(300, 400);
        this.addWindowListener(new MyWindowAdapter());
    }
    class MyWindowAdapter extends WindowAdapter {
        public void windowClosing(WindowEvent e) {
```

```
                ((Frame) e.getComponent()).dispose();
                System.exit(0);
            }
        }
}
```

用匿名内部类定义事件监听器

示例：

```
import java.awt.Frame;
import java.awt.event.*;
public class TestInnerClassEvent2 {
    public static void main(String[] args) {
        MyFrame4 frame = new MyFrame4();
    }
}
class MyFrame4 extends Frame {
    public MyFrame4() {
        this.setSize(300, 300);
        this.setVisible(true);
        this.setLocation(300, 400);
        this.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                ((Frame) e.getComponent()).dispose();
                System.exit(0);
            }
        });
    }
}
```

3.4、AWT 常用组件

3.4.1、Label

- **标签组件：可放置文本的组件**
- **构造方法**
  - Label(String text)　使用指定的字符串构造一个标签，对齐方式为左对齐
  - public　Label(String text, int alignment)
    - Alignment -对齐方式的值：可选值 Label.LEFT、Label.RIGHT、 Label.CENTER
- **常用方法**
  - public setAlignment(int alignment)　　设置对齐方式
  - public void setText(String text)　　设置标签的文本

3.4.2、TextField

- **单行文本框：允许编辑单行文本的文本组件**
- **构造方法**
  - public TextField()　构造新文本字段
  - public TextField(String text)
  - public TextField(int columns)　构造具有指定列数的新空文本字段
- **常用方法**
  - public void setEchoChar(char c)　设置此文本字段的回显字符
  - public String getText()　返回此文本组件表示的文本
  - public void setEditable(boolean b)　设置此文本组件是否可编辑

3.4.3、Button

- **标签按钮：当按下该按钮时，应用程序能执行某项动作**
- **构造方法：**
  - Button()
  - Button(String label)
- **常用方法**
  - void setActionCommand(String command)　设置此按钮激发的动作事件的命令名称。在默认情况下，此动作命令设置为与按钮标签相匹配
  - public String getActionCommand()　返回此按钮激发的动作事件的命令名称
  - public void addActionListener(ActionListener l)　添加指定的动作侦听器

3.4.4、Checkbox

- **复选框组件：用来创建单选按钮和多选按钮**
- **构造方法**
  - Checkbox(String label, boolean state)　创建多选按钮，state 指定是否默认选中
  - Checkbox(String label, boolean state, CheckboxGroup group)　创建单选按钮，第三个参数指定这个按钮所属的组
- **常用方法**
  - public void addItemListener(ItemListener l)　添加指定的项侦听器
  - void setState(boolean state)　将此复选框的状态设置为指定状态
  - boolean getState()

3.4.5、Choice

- **单选的下拉式列表框，内容不可改变。**

- **常用方法**
  - public void add(String item)　将一个项添加到此 Choice 菜单中
  - public String getSelectedItem()　获取当前选择的字符串表示形式
  - public int getSelectedIndex()　返回当前选定项的索引
  - public void remove(int position)　从选择菜单的指定位置上移除一个项
  - public void removeAll()　从选择菜单中移除所有的项
  - public void addItemListener(ItemListener l)　添加指定的项侦听器

3.4.8、List

- **内容可变的列表框组件，可单项或多项选择**
- **构造方法**
  - public List(int rows)　创建一个用指定可视行数初始化的新滚动列表，不允许进行多项选择
  - public List(int rows, boolean multipleMode)
    - rows - 可视行数
    - multipleMode - 如果为 true，则允许进行多项选择；否则，一次只能选择一项
- **常用方法**
  - add(String item)　向滚动列表的末尾添加指定的项
  - public String getSelectedItem()　获取此滚动列表中选中的项
  - public String[] getSelectedItems()　获取此滚动列表中选中的项

3.4.9、Scrollbar

- **滚动条组件**
- **构造方法**
  - public Scrollbar(int orientation, int value, int visible, int minimum, int maximum)　构造一个新的滚动条
    - orientation ：方向
      - Scrollbar.HORIZONTAL　水平
      - Scrollbar.VERTICAL　垂直
    - value：初始值
    - visible：可见量
    - minimum：最小值
    - maximum：最大值
- **常用方法**
  - public int getValue()　获取此滚动条的当前值
  - public void addAdjustmentListener(AdjustmentListener l)
    - 添加指定的调整侦听器

3.4.10、ScrollPane

- **带水平及垂直滚动条的容器组件**

- **构造方法**
  - public **ScrollPane**()　创建一个滚动窗格容器
    - ScrollPane.SCROLLBARS_AS_NEEDED: 只在滚动窗格需要时显示
    - ScrollPane.BARS_ALWAYS:　总是显示滚动条
    - ScrollPane.SCROLLBARS_NEVER:　永远不创建或显示滚动条
- **常用方法**
  - public Component **add**(Component comp)　将指定组件追加到此容器的尾部
  - public final void setLayout(LayoutManager mgr)　设置此容器的布局管理器

3.4.11、TextArea

- **多行文本域：**
- **构造方法**
  - public TextArea()
  - public TextArea(int rows, int columns)
  - public TextArea(String text, int rows, int columns, int scrollbars)
    - scrollbars：滚动条类型
- **常用方法**
  - public void append(String str)　将给定文本追加到文本区的当前文本
  - public void replaceRange(String str, int start, int end)　用指定替换文本替换指定开始位置与结束位置之间的文本
  - public int getCaretPosition()　返回文本插入符的位置
  - public String getSelectedText()　返回此文本组件所表示文本的选定文本
  - public void setEditable(boolean b)　设置此文本组件是否可编辑

3.4.12、菜单相关

- MenuBar　菜单栏组件
- Menu　菜单组件
- MenuItem　菜单项（二级菜单）组件
- CheckboxMenuItem　带复选框的菜单项组件
- PopupMenu　弹出式菜单组件

如何创建菜单

- 创建一个 MenuBar 对象，并将其置于一个可容纳菜单的容器(如 Frame 对象)中。
  - 调用 Frame 类的 setMenuBar(MenuBar mb)方法。
- 创建一个或多个 Menu 对象，并将它们添加到先前创建的 MenuBar 对象中。
- 创建一个或多个 MenuItem 对象，再将其加入到各个 Menu 对象中
  - 设置快捷键：　menuItem.setShortcut(new MenuShortcut(KeyEvent.VK_N));

弹出式菜单

- 能够在组件中的指定位置上动态弹出的菜单
- 弹出的方法
  - public void show(Component origin, int x, int y)
    - 在相对于初始组件的 x、y 位置上显示弹出式菜单

示例：

```
PopupMenu popup = new PopupMenu();
popup.add("a");
popup.add("b");
popup.add("c");
textArea.add(popup);
textArea.addMouseListener(new MouseAdapter() {
        public void mouseReleased(MouseEvent e) {
            if(e.getButton() == MouseEvent.BUTTON3){
                popup.show(e.getComponent(), e.getX(), e.getY());
            }
        }
});
```

3.5、Java 图形

Graphics 类和 Canvas 组件
paint 方法

Graphics

- Graphics 类是所有图形操作的抽象基类，提供了在组件上进行图形、文字绘制及显示图像等操作方法：
  - drawLine(int x1, int y1, int x2, int y2)　画线
  - drawOval(int x, int y, int width, int height)　画椭圆
  - drawRect(int x, int y, int width, int height)　画矩形
  - drawString(String str, int x, int y)　画字符串
  - drawImage(Image img, int x, int y, ImageObserver observer)　画图像
  - fillOval(int x, int y, int width, int height)　填充椭圆
  - fillRect(int x, int y, int width, int height)　填充指定的矩形
  - …
- java.awt.Component 类中提供了一个 getGraphics 方法来返回一个包含有该组件的屏幕显示信息的 Graphics 类对象

Canvas 组件

- Canvas 组件表示屏幕上一个空白矩形区域，应用程序可以在该区域内绘图。

示例：

```
public class GraphicsTest {
    public static void main(String[] args) {
Frame f = new Frame("画图示例");
//创建一个区域
Canvas canvas = new Canvas();
f.add(canvas);
f.setSize(300, 400);
f.setVisible(true);
//要先让组件显示后，才会返回 Graphics 对象，否则会返回 null
Graphics g = canvas.getGraphics();
g.drawLine(50, 50, 200, 50);  //画线
g.drawString("Graphics 画图", 50, 70);  //画字符串
g.drawRect(50, 100, 200, 50);  //画矩形
g.drawOval(50, 160, 200, 50);  //画椭圆（矩形中填充椭圆）
    }
}
```

组件重绘

- 在组件大小改变或隐藏后又显示，AWT 线程都会重新绘制组件，组件上原来绘制的图形也就不复存在了。解决办法：
  - AWT 组件在重绘时，会立即调用组件的 paint 方法，我们只需要在这这个方法中编写我们的绘图代码。

示例：

```
class MyFrame extends Frame {
    public MyFrame(){
this.setBounds(100, 100, 300, 500);
this.setLayout(null);
this.setVisible(true);
```

```
    }
    public void paint(Graphics g) {
//要先让组件显示后，才会返回 Graphics 对象，否则会返回 null
//Graphics g = g.getGraphics();
g.drawLine(50, 50, 200, 50);   //画线
g.drawString("Graphics 画图", 50, 70);   //画字符串
g.drawRect(50, 100, 200, 50);      //画矩形
g.drawOval(50, 160, 200, 50);      //画椭圆
//画图像
URL url = Thread.currentThread()
.getContextClassLoader()
.getResource("ubuntulogo.png");
Image img = this.getToolkit().getImage(url);
g.drawImage(img, 50, 300, this);
    }
}
```

3.6、Swing

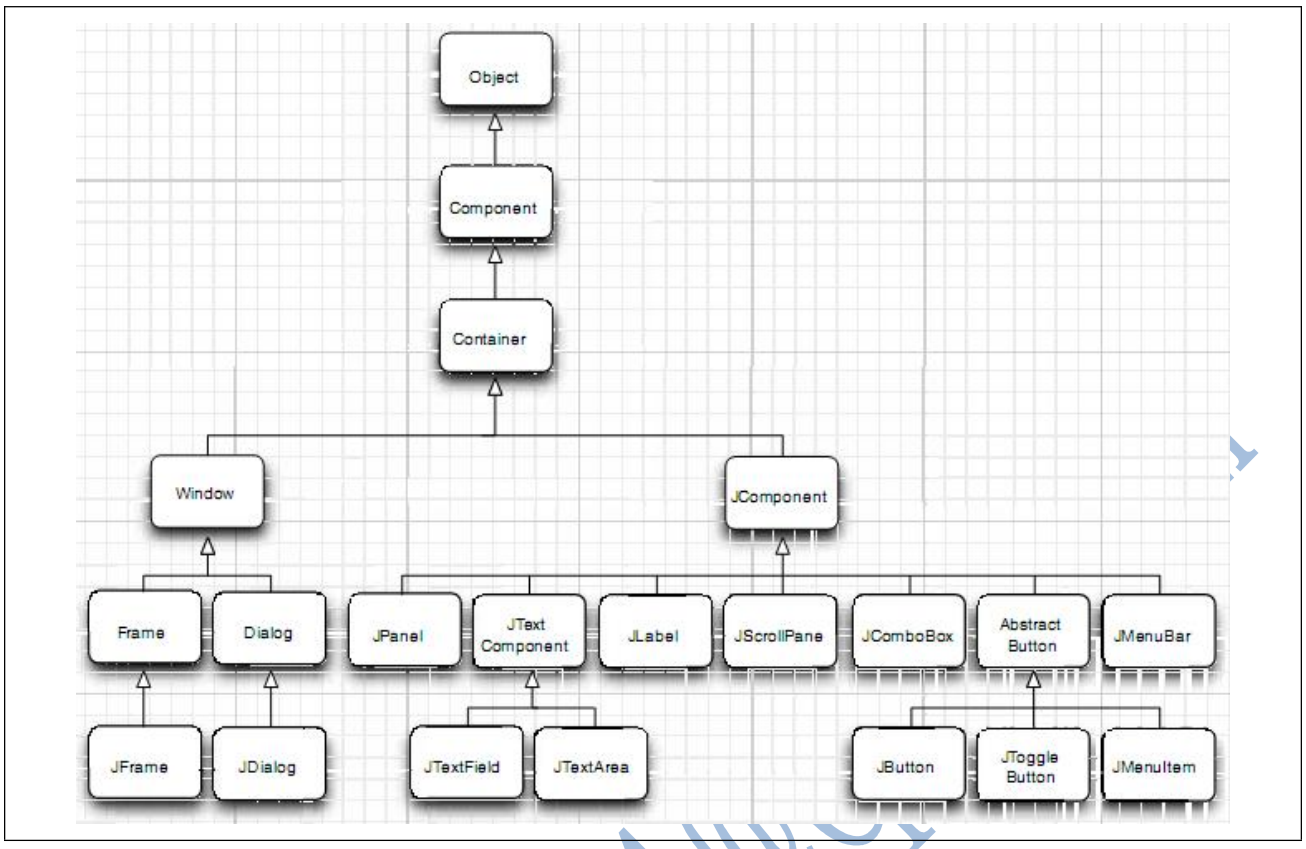
- Swing 介绍
- Swing 常用组件使用

3.6.1、Swing 介绍

Swing 工具集

- Swing 是第二代 GUI 开发工具集
- AWT 采用了与特定平台相关的实现，而绝大多数 Swing 组件却不是
- Swing 是构筑在 AWT 上层的一组 GUI 组件的集合，为保证可移植性，它完全用 Java 语言编写
- 和 AWT 相比，Swing 提供了更完整的组件(放置在 javax.swing 包下)，引入了许多新的特性和能力
- Swing 增强了 AWT 中组件的功能，这引起增强的组件命名通常是在 AWT 组件名前增加了一个“J”字母；同时也提供了更多的组件库，如：JTable、JTree、JComboBox 等。

Swing 继承体系



3.6.2、JFrame

- **继承自 java.awt.Frame 类，功能也相当。**
- **与 Frame 区别之处：**
  - JFrame 上自带一个放置内容的面板 JContentPane，我们应该在这个面板上进行增加组件和设置布局管理器。
    - 调用 JFrame 的 getContentPane()方法可获得它的 JContentPane 对象。
    - 从 JDK5.0 之后，重写了 add(Component comp)和 setLayout(LayoutManager l)方法，直接调用这两个方法也是在操作 JContentPane 对象。
  - 当用户点击 JFrame 的关闭按钮时，JFrame 会自动隐藏，但没有关闭，可以在 windowClosing 事件中关闭。但更常用的方式是调用 JFrame 的方法来关闭
    - setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);

3.6.3、其它组件

- JPanel　面板
- JLabel　标签
- Icon, ImageIcon：图像图标
  - 路径要是绝对路径，或图片的 URL
- JButton　按钮
- JCheckBox　复选框
- JRadioButton　单选按钮

- 要放置到 javax.swing.ButtonGroup 中

❖ JScrollPane 带滚动条的面板

```
URL imgURL = Thread.currentThread().getContextClassLoader().getResource("javalogo.png");
Icon icon = new ImageIcon(imgURL);
JScrollPane panel = new JScrollPane(textArea,
ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS,
ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);
```

❖ JTextField、JTextArea

❖ JOptionPane 标准对话框

- public static int showConfirmDialog(…)
- public static int showOptionDialog(…)

❖ JPasswordField 密码框

❖ JScrollBar 滚动条

❖ JComboBox 下拉列表

❖ JMenuBar

- JMenu、JPopupMenu
- JMenuItem、JCheckBoxMenuItem、JRadioButtonMenuItem

❖ JFileChooser 文件选择器

❖ JColorChooser 颜色选择器

❖ JProgressBar 进度条

❖ JToolBar 工具条

3.6.4、设置 Swing 感观

```
class MyFrame2 extends JFrame {
    public MyFrame2() {
setTitle("程序标题");
setBounds(200, 200, 300, 300);
setDefaultCloseOperation(EXIT_ON_CLOSE);
setLayout(null);
JButton btn = new JButton("按钮");
btn.setBounds(10, 10, 60, 25);
this.add(btn);
// 设置程序感观
setupLookAndFeel();
// 显示窗口
setVisible(true);
    }
    // 设置程序感观
    private void setupLookAndFeel() {
// 取得系统当前可用感观数组
```

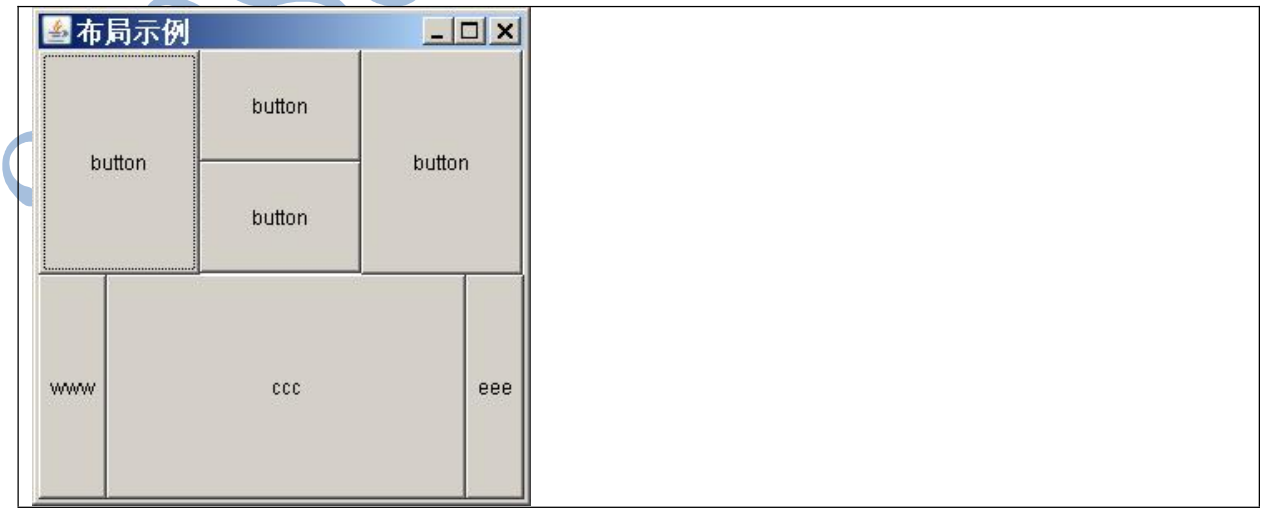
```
UIManager.LookAndFeelInfo[] arr = UIManager.getInstalledLookAndFeels();
Random random = new Random();
String strLookFeel = arr[random.nextInt(arr.length)].getClassName();
//得到可跨平台的感观
//String strLookFeel2 = UIManager.getCrossPlatformLookAndFeelClassName();
try {
        UIManager.setLookAndFeel(strLookFeel);
        SwingUtilities.updateComponentTreeUI(this);
} catch (Exception e) {        System.out.println("Can't Set Lookandfeel Style to " + strLookFeel);    }
    }
}
```

4、总结

AWT 组件的使用
Swing 组件的使用
布局管理器
事件处理
了解 Java 图形
熟悉 AWT 和 SWING 的常用组件，反复练习

5、作业

1、使用布局管理器实现下图效果：



2、使用布局管理器实现计算器界面效果：

3、使用 Swing 构建记事本界面