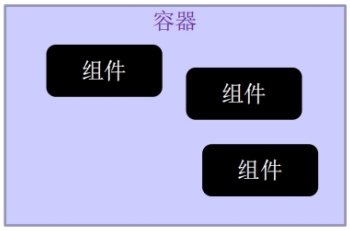
GUI编程

GUI与AWT

- 与以往的控制台程序不同，今天我们所碰到的应用程序都是图形用户接口（GUI，Graphics User Interface）
- 抽象窗口工具包（AWT，Abstract Window Toolkit）提供图形用户接口（GUI，Graphical User Interface）所需要的组件。这些组件支持按照 机器上的风格显示（依赖于平台）。
- GUI的基本类库位于java.awt包中，这个包也被称为抽象窗口工具箱（AWT，Abstract Window Toolkit）
- AWT按照面向对象的思想来创建GUI，它提供了容器类、众多的组件类和布局管理器类
- 请注意AWT中类，以便于继承，实现GUI编程



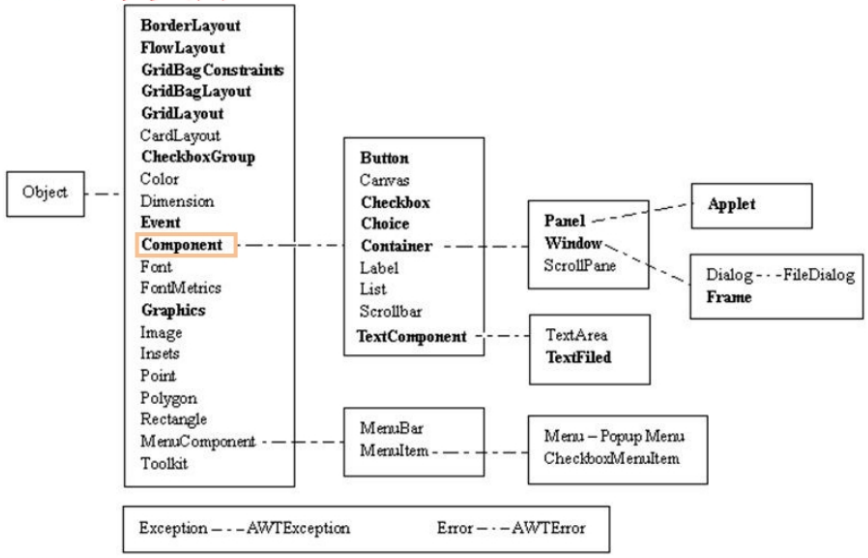
Java的图形工具包：`java.awt.*`(轻量级)，`java.swing.*`

AWT组件

AWT组件

- 在java.awt包中，还提供了可以加入到容器类组件中的各种具体组件，如按钮Button、文本框TextField和文本区域TextArea等
- AWT组件的优点是简单、稳定，兼容于任何一个JDK版本，缺点是依赖于本地操作系统的GUI，缺乏平台独立性
- 由于AWT组件与本地平台的GUI绑定，因此用AWT组件创建的图形界面在不同的操作系统中会有不同的外观
- 为了使用Java语言创建的图形界面也能够跨平台，即在不同操作系统中保持相同的外观，从JDK1.2版本开始引入了Swing组件
- Swing组件位于javax.swing包中

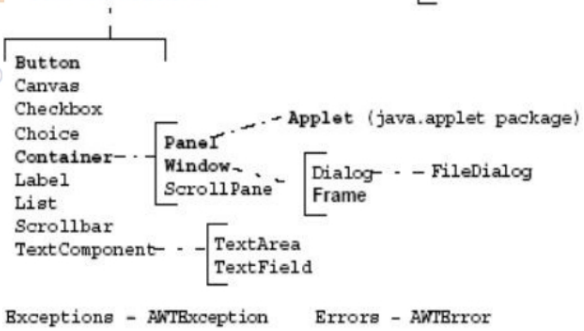
AWT基本类库



AWT中的Component

AWT中的Component

- Component类中声明了所有组件都拥有的方法：
  - setBackground(Color c)/getBackground()：设置/返回组件的背景色
  - getGraphics()：返回组件使用的画笔
  - setVisible(boolean b)/isVisible()：设置/判断组件是否可见
  - setForeground(Color c)：设置组件的前景色



- `setBackground()` 设置组件背景色
- `getBackground()` 返回组件画笔
- `setVisible(boolean b)` 设置是否可见
- `setForeground()` 设置前景色

Container类

- 容器类`Container`是`Component`的子类，它上面允许放置其它组件（包括*包含组件的容器*）
- 容器类可以**嵌套放置**，这对GUI很有用。
- 两个典型窗口类：
  - `Window（窗口）类`：Window是**不依赖于其他容器而独立存在**的容器
    1. `Window` 有两个子类：`Frame（窗体）类和Dialog（对话框）类`
    2. `Frame`带有标题，而且*可以调整大小*；`Dialog`可以被移动，但是*不能改变大小，不可以有菜单栏*
  - `Panel（面板）类`：Panel不能单独存在，只能**存在于其他容器**（`Window`或其子类）中
    `Panel`对象代表了一个长方形的区域，在这区域中可以容纳其他的组件
- `add(Component c)` 添加组件

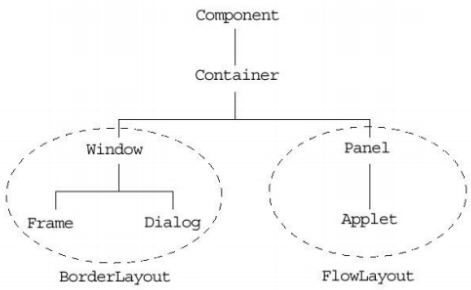
容器的布局管理器

容器的布局管理器

- 组件在容器中的大小和位置由容器的布局管理器（Layout Manager）决定，每一个容器要引用一个特定类型的布局管理器
  - 当容器需要定位和决定组件大小时，激活布局管理器，由布局管理器自动完成
  - 布局管理器全权管理负责容器中所有组件的布局，根据屏幕大小计算组件的最佳尺寸等，最佳尺寸依赖与具体平台的风格。布局管理器能自动根据不同平台管理组件
  - 因为布局管理器负责容器里的组件的位置和大小，因此程序中不需要自己去设定组件的大小或位置
- 如果你修改组件的大小和位置，如setLocation()、setSize()、setBounds()
  - 因为布局管理器在起作用，这些方法不会起作用
  - 如果必须自定义方式控制组件的大小或位置，那就通知容器中来中止布局管理器：setLayout(null);
  - 中止容器缺省的布局管理器后，你必须对容器内所有的组件的显示负责，位置，大小等；这将会是很麻烦的工作
  - 最好方法是生成LayoutManager的子类，覆盖其中的一些功能，定制对组件大小和位置的控制

◆ 每个容器都有缺省的布局管理器，可以使用setLayout修改容器的布局管理器

- FlowLayout：Panel和Applet的缺省布局管理器
- BorderLayout：Window、Frame和Dialog的缺省布局管理器
- GridLayout：给组件定位提供灵活性的布局管理器
- CardLayout：象扑克牌一样叠在一起，每次只显示其中一个组件
- GridBagLayout：不讨论



Frame 帧

- 帧（Frame）是`Window类`的*子类容器*，有标题栏和调节大小的四角
- 从Window类继承了`add`方法，可以添加组件
- *缺省布局管理器*是`BorderLayout`，可以使用`setLayout`方法修改布局管理器
- 构造器`Frame(String)`创建一个不可见的窗口，标题为`String`指定的字符串，使用`setVisible(true)`可以显示帧

【注意】在Eclipse中由于使用了模块，所以需要修改`module-info.java`，添加一条语句：`requires java.desktop;`

例：生成一个红色的小窗口

JAVA

```java
import java.awt.*;

public class FrameTest {
        private Frame f;

        public FrameTest() {
                f = new Frame("Hello out there"); // Frame名字
        }

        public void launchFrame() {
                f.setSize(170, 170);  // 设置尺寸
                f.setBackground(Color.red);  // 设置颜色
                f.setVisible(true);  // 开了这个才能看到图像
        }

        public static void main(String[] args) {
                // TODO Auto-generated method stub
                FrameTest guiWindow = new FrameTest();
                guiWindow.launchFrame();
        }
}
```

Panel 面板

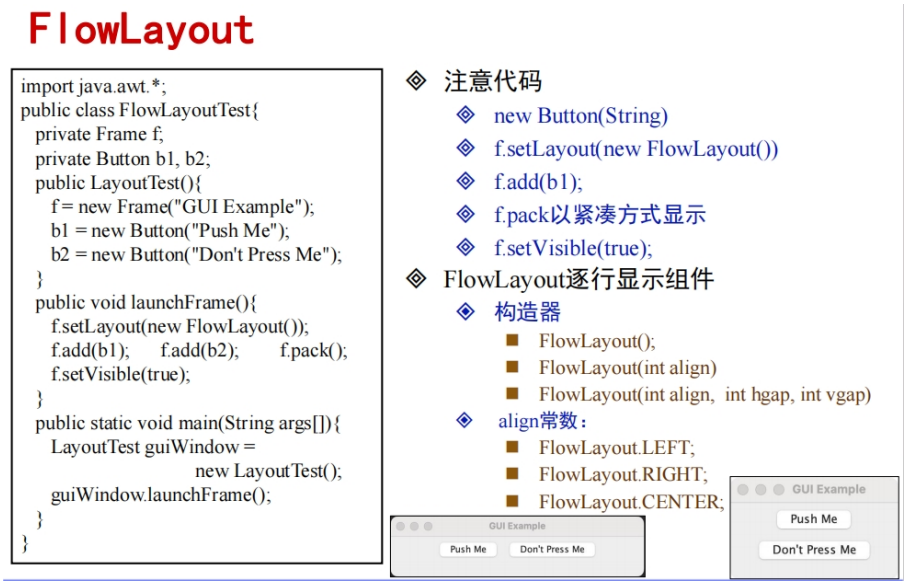
- 面板（Panel）是容器类`Container`的子类容器，是可以在上面**放置其它组件的空白块。**
- Panel的*缺省布局管理器*是`FlowLayout`
- Panel必须添到`Window或其子类（Frame）`容器，这样当显示Frame的时候，Panel也一起显示

JAVA

```
public class FrameWithPanel {
        private Frame f;
        private Panel p;

        public FrameWithPanel(String title) {
                f = new Frame(title);
                p = new Panel();
        }
        public void launchFrame() {
                f.setBackground(Color.blue);
                //f.setLayout(null);
                f.setVisible(true);
                p.setSize(100, 100);
                f.setSize(200, 200);
                p.setBackground(Color.yellow);
                f.add(p);  // 必须将Panel加到Frame中
        }
        public static void main(String[] args) {
                FrameWithPanel guiWindow = new FrameWithPanel("Frame With Panel");
                guiWindow.launchFrame();
        }
}
```

FlowLayout 浮动式

注意构造器是重载过的，可以支持多种传入方式

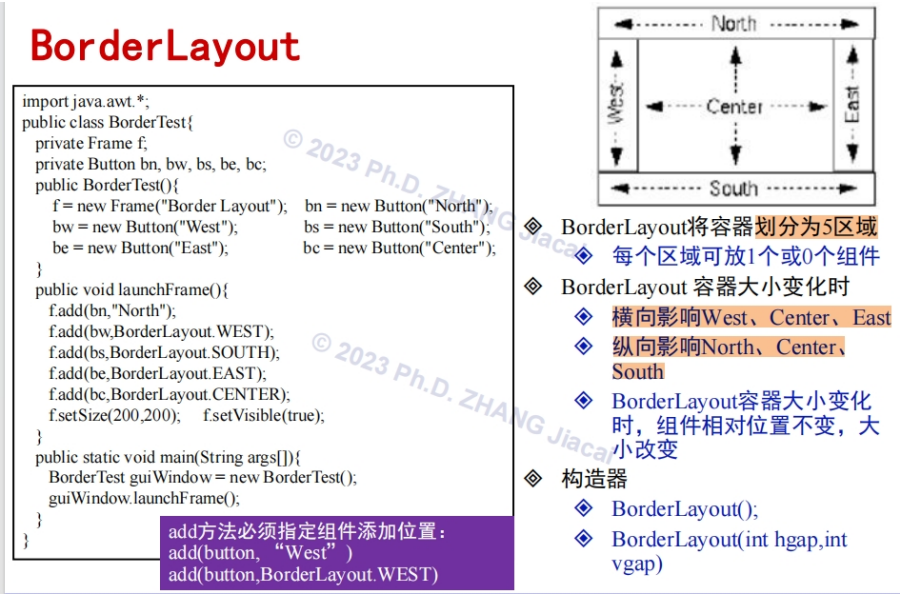


浮动式的原因是可以通过调整窗口大小变换按钮的摆布方式

JAVA

```java
import java.awt.*;
public class FlowLayoutTest {
        private Frame f;
        private Button b1, b2;  // 添加按钮
        public FlowLayoutTest() {
                f = new Frame ("GUI Example");  // 名字
                b1 = new Button("Push Me");  // 设置按钮对象
                b2 = new Button("Don't Press Me");
        }
        public void launchFrame() {
                f.setLayout(new FlowLayout());  // 设置布局
                f.add(b1);   // 在画面上添加按钮
                f.add(b2);
                f.pack();  // 以紧凑方式显示
                f.setVisible(true);
        }
        public static void main(String[] args) {
                FlowLayoutTest guiWindow = new FlowLayoutTest();
                guiWindow.launchFrame();
        }
}
```

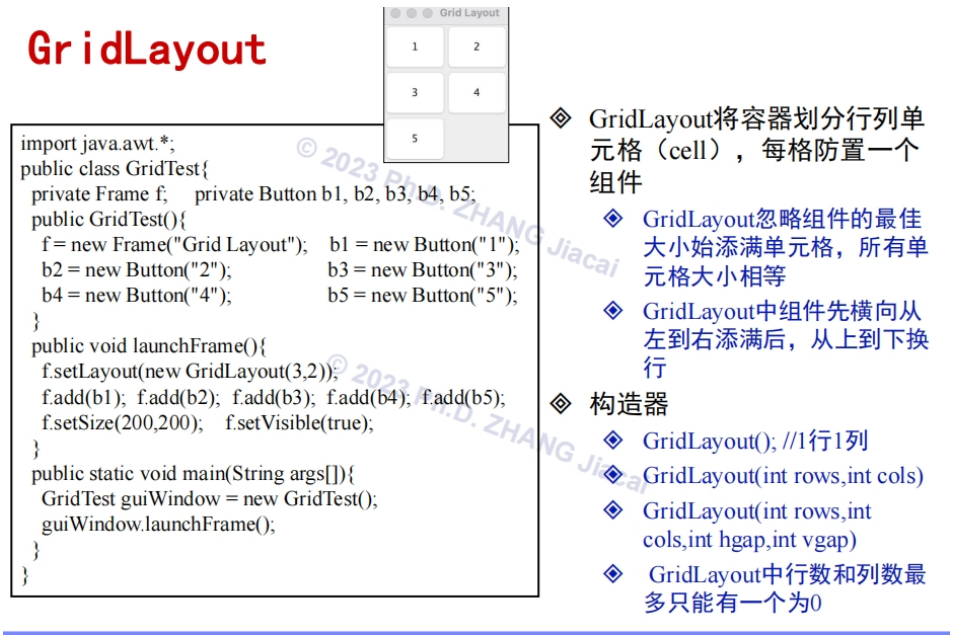
BorderLayout 区域式



JAVA

```java
import java.awt.*;
public class BorderTest {
        private Frame f;
        private Button bn, bw, bs, be, bc;
        public BorderTest() {
                f = new Frame ("Border Layout");  // 名字
                bn = new Button("North");  // 设置按钮对象
                bw = new Button("West");
                bc = new Button("Center");
                bs = new Button("South");
                be = new Button("East");
        }
        public void launchFrame() {
                // f.setLayout(new FlowLayout());
                // 因为缺省默认为BorderLayout，所以不用再进行设置
                f.add(bn, "North");    // 在画面上添加按钮
                f.add(bs, "South");
                f.add(bc, "Center");
                f.add(be, "East");
                f.add(bw, "West");
                f.setSize(200, 200);
                f.setVisible(true);
        }
        public static void main(String[] args) {
                BorderTest guiWindow = new BorderTest();
                guiWindow.launchFrame();
        }
}
```

GridLayout 格子式



JAVA

```java
import java.awt.*;
public class GridTest {
        private Frame f;
        private Button b1,b2,b3,b4,b5;
        public GridTest() {
                f = new Frame("Grid Layout");
                b1 = new Button("1");
                b2 = new Button("2");
                b3 = new Button("3");
                b4 = new Button("4");
                b5 = new Button("5");
        }
        public void launchFrame() {
                f.setLayout(new GridLayout(3, 2));
                // 设置2*2的样式
                f.add(b1);
                f.add(b2);
                f.add(b3);
                f.add(b4);
                f.add(b5);
                f.setSize(200, 200);
                f.setVisible(true);
        }
        public static void main(String[] args) {
                GridTest guiWindow = new GridTest();
                guiWindow.launchFrame();
        }
}
```

CardLayout 卡片式

CardLayout

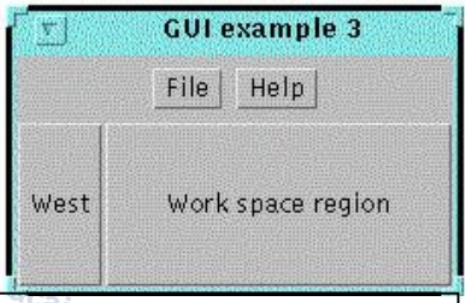
- Card布局管理器像一系列的卡，任何时候只见其中一个。
- add()方法来将卡添加到Card布局中。
- Card布局管理器的show()方法显示指定的一张卡。


```
import java.awt.*;
public class CardTest {
   Panel p1, p2, p3;
   Label l1, l2, l3;
   CardLayout myCard;
   Frame f;
   public static void main (String s[]) {
     CardTest ct = new CardTest ();  ct.init();
   }
   public void init () {
     f = new Frame ("Card Test");
     myCard = new CardLayout();
     f.setLayout(myCard);
     p1 = new Panel();       p2 = new Panel();  p3 = new Panel();
     l1 = new Label("This is the first Panel");
     p1.setBackground(Color.yellow);
     l2 = new Label("This is the second Panel");
     p2.setBackground(Color.green);
     l3 = new Label("This is the third Panel");
     p3.setBackground(Color.magenta);
     p1.add(l1);   p2.add(l2);  p3.add(l3);
     f.add(p1, "First"); f.add(p2, "Second");    f.add(p3, "Third");
     myCard.show(f, "Second");
     f.setSize (200, 200);
     f.setVisible(true);
   }
}
```

JAVA

```java
import java.awt.*;
public class CardTest {
        Panel p1, p2 ,p3;
        Label l1, l2, l3;
        CardLayout myCard;
        Frame f;

        public static void main(String[] args) {
                CardTest ct = new CardTest();
                ct.init(); // 和luanchFrame作用差不多
        }
        public void init() {
                f = new Frame("Card Test");
                myCard = new CardLayout();
                f.setLayout(myCard);
                p1 = new Panel();
                p2 = new Panel();
                p3 = new Panel();
                l1 = new Label("This is the first Panel");
                p1.setBackground(Color.yellow);
                l2 = new Label("This is the Second Panel");
                p2.setBackground(Color.green);
                l3 = new Label("This is the third Panel");
                p3.setBackground(Color.magenta);

                p1.add(l1);
                p2.add(l2);
                p3.add(l3);
                f.add(p1, "First");
                f.add(p2, "Second");
                f.add(p3, "Third");
                myCard.show(f, "Second");
                f.setSize(200, 200);;
                f.setVisible(true);
        }
}
```

复杂布局



- BorderLayout每个区域只能显示一个组件，看上去却放置了两个组件
- North区放Panel容器，其中包含两个按钮

```
import java.awt.*;
public class ComplexTest{
  private Frame f;
  private Panel p;
  private Button bw, bc;
  private Button bfile, bhelp;

  public ComplexTest(){
    f = new Frame("Complex Layout");
    bw = new Button("West");
    bc = new Button("Work Space");
    bfile = new Button("File");
    bhelp = new Button("Help");
  }
```

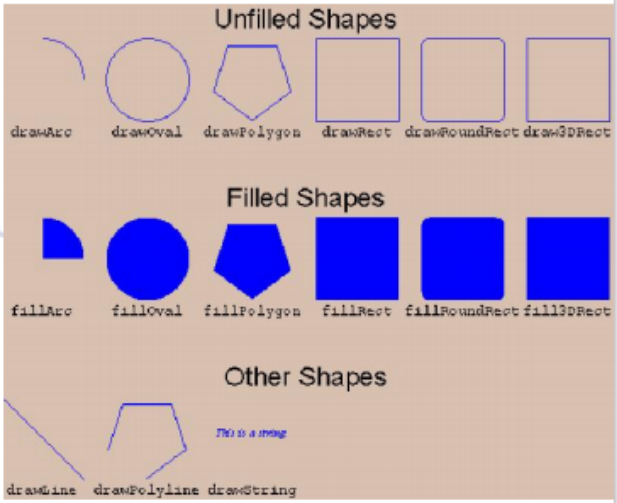
```
  public void launchFrame(){
    f.add(bw, BorderLayout.WEST);
    f.add(bc, BorderLayout.CENTER);
    p = new Panel();
    p.add(bfile); p.add(bhelp);
    f.add(p, BorderLayout.NORTH);
    f.pack();
    f.setVisible(true);
  }
  public static void main(String args[]){
    ComplexTest guiWindow = new ComplexTest();
    guiWindow.launchFrame();
  }
}
```

JAVA

```java
import java.awt.*;
public class ComplexTest {
        private Frame f;
        private Panel p;
        private Button b1, b2, b3, b4;

        public ComplexTest() {
                b1 = new Button("File");
                b2 = new Button("Help");
                b3 = new Button("West");
                b4 = new Button("Work space region");
        }
        public void launchFrame() {
                f = new Frame("GUI example 3");
                p = new Panel();

                p.setSize(100, 200);
                f.setSize(200, 200);

                p.add(b1);
                p.add(b2);
                // Panel的默认缺省是FlowLayout类型，而Frame的默认缺省是BorderLayout类型，所以这里都不需要设置布局管理器
                f.add(p, BorderLayout.NORTH); //不知道为什么我的会报错
                f.add(b3, "West");
                f.add(b4, "East");
                f.pack();
                f.setVisible(true);
        }
        public static void main(String[] args) {
                ComplexTest guiWindow = new ComplexTest();
                guiWindow.launchFrame();
        }
}
```

AWT画图

- 每个组件都有一个paint()方法，用于画出特定的组件。用户可以在该方法中使用组件的Graphics对象直接在屏幕上画图
- AWT提供了画布（Canvas）和面板（Panel）用于画图
  - 每当一个Canvas或Panel需要重新绘制时，调用paint方法
- 具体什么时候调用由AWT线程决定
  - Canvas或Panel第一次显示；
  - 遮挡窗口移开时；
  - 包含Canvas或Panel的Frame最下化后，恢复原来大小时；
  - repaint方法高用paint方法
- paint方法的唯一参数是一个Graphics对象，它包含有许多画各种形状图形的方法，封闭图形还可以着色。




```
public void paint(Graphics g){
    g.setColor(Color.red);
    g.fillOval(1,1,getSize(). width, getSize().height);
  }
```

定制显示方式组件

```java
import java.awt.*;
class MyPanel extends Panel{ // 通过继承Panel类对画出的Panel进行修改
        public void paint(Graphics g){  // 对paint进行重写
                g.setColor(Color.red);
                g.fillOval(1, 2, getSize().width, getSize().height);
                //在面板上绘制一个填充的椭圆。椭圆的左上角坐标是(1，2)，宽度和高度分别等于面板的宽度和高度。
        }
}
public class MyPanleTest{
        private Frame f;
        private MyPanel p1, p2, p3, p4;
        public MyPanelTest(){
                f = new Frame("Grid Layout");
                p1 = new MyPanel();
                p2 = new MyPanel();
                p3 = new MyPanel();
                p4 = new MyPanel();
        }
        public void main launchFrame(){
                f.setLayout(new GridLayout(2,2));  // 通过new方式设置Layout
                f.add(p1);
                f.add(p2);
                f.add(p3);
                f.add(p4);
                f.setSize(200, 200);
                f.setVisible(true);
        }
        public static void main(String args[]){
                MyPanelTest guiWindow = new MyPanelTest();
                guiWindow.launchFrame();
        }
}
```

事件处理

- Java语言将每一个键盘或鼠标的操作定义为一个**“事件”**，在编程中只需定义每个特定事件发生时程序应该做出何种响应即可。这就是图形用户界面中的*“事件”*和*“事件响应”*
- 除了鼠标和键盘的操作，又如系统的状态改变、标准图形界面元素等都可以引发事件，系统对这些事件分别定义处理代码，就可以保证应用程序系统有条不紊地工

事件

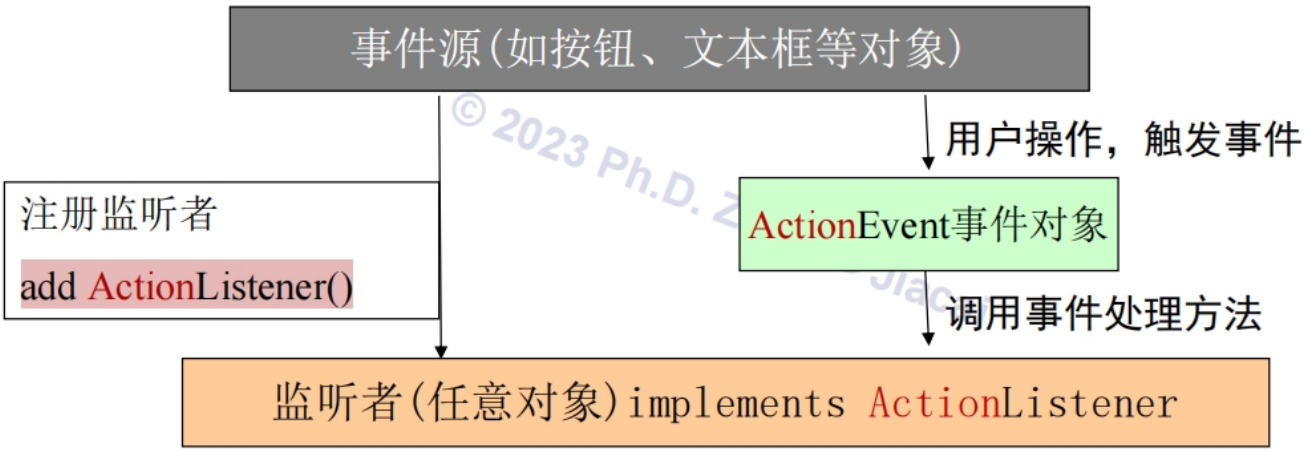
- 当用户在用户界面上单击鼠标或敲键盘时，向外发出事件（Event）对象
- **事件源**（Event Source）是事件生成者，比如在按钮上单击鼠标，就会生成一个`ActionEvent`对象，它的**事件源就是按钮**

- 事件对象描述发生的事情，`ActionEvent`对象包含一些方法可以提取事件发生时的一些信息：
  - `getActionCommand() ;` 获得设置的Command信息
  - `getModifier();`

事件处理的模式

◈ JDK1.1后，采用事件授权（Delegation）机制。每个对象源获得产生的事件后，将事件上传给一个或多个注册的类（监听者，Listeners）。监听者包含事件接收和处理程序。

◈ 事件处理程序是一个接收、翻译事件的方法



◈ 采用事件授权（Delegation）机制。每个事件源获得产生的事件后，将事件上传给一个或多个注册的类（监听者，Listeners）。监听者包含事件接收和处理程序。

◈ 事件授权机制中，事件可以授权给其它对象（监听者）处理。监听者是实现了EventListener接口的类，不同事件有不同监听者，每个事件的监听者实现对应的接口（包含处理对应事件的对应方法）。

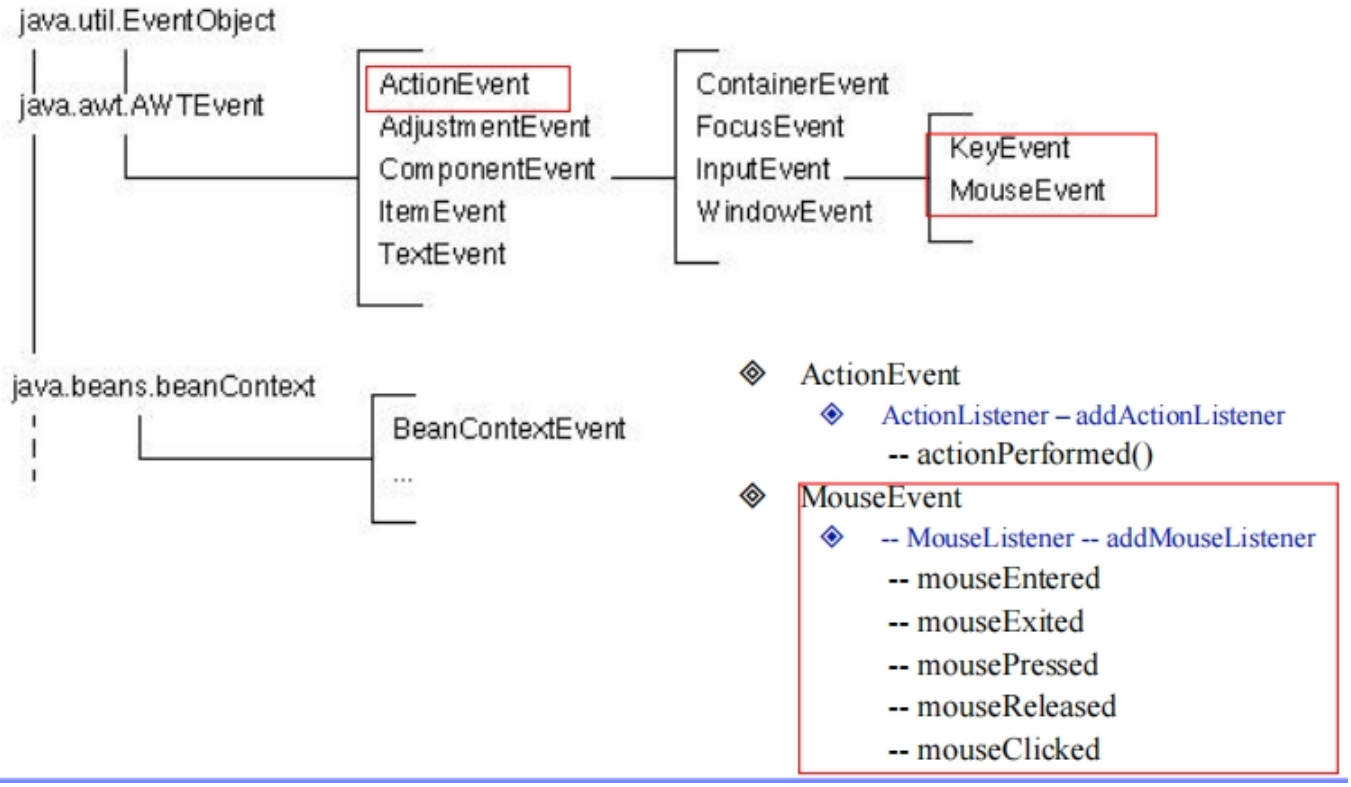
◈ 事件只报告给监听者，没有监听者的事件没有响应。

事件的授权处理机制

◈ 向事件源（谁生成事件）注册事件监听者（谁处理事件）
  - ◈ 只有在组件中注册了的监听者才能够观察到组件中发生的事件，并对事件进行响应
  - ◈ 使用事件的授权机制，可以将事件处理和组件显示独立，也便于将组件的工作分布给多个对象处理
  - ◈ 同一组件的事件会很多，如窗口有关闭事件，打开事件，最小化，最大化，鼠标移动等不同类型事件，可以将这不同的事件分别交给不同对象处理
  - ◈ 即便是同一事件也可以让多个监听者来重复处理，这个时候就要为组件注册多个事件监听者。教官喊立正（事件），多个士兵（监听者）都要站好
  - ◈ 同一事件有多个监听者时，这些监听者的处理方法执行的先后顺序不确定。如果要使这些处理按顺序进行，只有一个方法。那就是只注册第一个监听器，在它的事件处理中，调第二个事件处理；依次类推

◈ 对于每种事件，它们的监听者必须实现对应的处理程序（方法），这是能够在监听器类实现相应的接口来保证的
  - ◈ Java的与事件处理相关的类都定义在包java.awt.event中。
  - ◈ 每个事件都会对应有一个接口，这个接口中所定义的方法可能有多个，当一个事件发生时，某个方法将会被调用

Event和EventListener



各个`Event`对应的监听者：

Category	Interface Name	Methods
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)

Category	Interface Name	Methods
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)

例子：按钮事件的响应

JAVA

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

public class TestButton {
        private Frame f;
        private Button b;

        public TestButton(){
                f = new Frame("Test Button");
                b = new Button("Press Me"); // 默认反馈为Press Me
                b.setActionCommand("ButtonPressed"); //修改ActionCommand的反馈
        }
        public void launchFrame() {
                b.addActionListener(new ButtonHandler()); // 注册监听者，不会触发垃圾回
收
                f.add(b, "Center");
                f.pack();
                f.setVisible(true);
        }
        public static void main(String[] args) {
                TestButton guiApp = new TestButton();
                guiApp.launchFrame();
        }
}
class ButtonHandler implements ActionListener{
        // 监听者需要实现接口
        public void actionPerformed(ActionEvent e) { // 触发反馈
                System.out.println("Action occered");
                System.out.println(e.getActionCommand());
                System.exit(0);
        }
}
```

执行程序后，会显示一个写着"Press Me"按钮，点击后即可退出程序，并在控制台显示"Action occered"和"ButtonPressed"语句

- `ActionEvent`传给`ActionListener`接口中定义的方法`actionPerformed`中，其中的参数`e`中就能捕获到当前发生的事件
- 注册监听者：`b.addActionListener(ActionListener al)`
- `ActionListener`接口只有一个抽象方法

```java
public void actionPerformed(ActionEvent e)
```

捕获鼠标移动的例子

- 捕获鼠标的运动
  - 按下鼠标的拖动（mouseDraged）
  - 松开鼠标的移动(mouseMoved)
- 鼠标运动事件可以交给MouseMotionListener接口
  - 定义了mouseDraged方法和mouseMoved方法。
- 要捕获更多鼠标事件，就要实现MouseListener接口
  - mouseEntered
  - mouseExited
  - mousePressed
  - mouseReleased
  - mouseClicked
- 当鼠标按下或运动时，有关鼠标位置的信息就包含在鼠标事件对象MouseEvent中
  - getX(); getY();

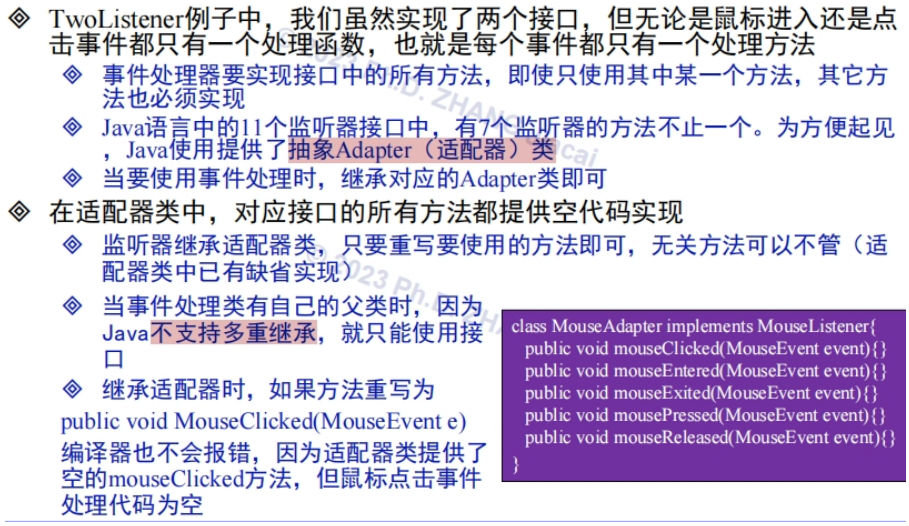
JAVA

```java
package lesson13;
import java.awt.*;
import java.awt.event.*;
public class TwoListener implements MouseMotionListener, MouseListener {  // 实现两个接口
        private Frame f;
        private TextField tf;

        public TwoListener() {
                f = new Frame("Two Listener");
                tf = new TextField(30);  // 显示文本区域
        }

        public void launchFrame() {
                Label label = new Label("Click and drag");
                f.add(label,"North");
                f.add(tf,"South");
                f.addMouseListener(this);  // 把自己加入到监听者
                f.addMouseMotionListener(this);
                f.setSize(200, 200);
                f.setVisible(true);
        }
        public void mouseDragged(MouseEvent e) {
                String s = "Mouse: x = " + e.getX() + "y = " + e.getY();
                tf.setText(s); // 文本栏显示文本
        }
        public void mouseEntered(MouseEvent e) {
                String s = "The mouse entered";
                tf.setText(s);
        }
        public void mouseExited(MouseEvent e) {
                String s = "The mouse exited";
                tf.setText(s);
        }
        public void mouseClicked(MouseEvent e) {
                System.exit(0);
        }
        public void mouseMoved(MouseEvent e) {}
        public void mousePressed(MouseEvent e) {}
        public void mouseReleased(MouseEvent e) {}

        public static void main(String[] args) {
                TwoListener two = new TwoListener();
                two.launchFrame();
        }
}
```

代码实现了鼠标移动、点击、拖动、退出的触发反馈，并会在窗口中的文本框内显示

Q: 在代码中，Moved，Pressed和Released方法实际上并没有用到，但因为接口必须实现这些方法，如果不想实现那么多个方法怎么办?

监听器实现所有接口方法



JAVA

```java
import java.awt.*;
import java.awt.event.*;
public class TwoListener extends MouseAdapter implements MouseMotionListener{  // 使用继承 + 实现接口
        //...//
        public void mouseDragged(MouseEvent e) {
                String s = "Mouse: x = " + e.getX() + "y = " + e.getY();
                tf.setText(s);
        }
        public void mouseEntered(MouseEvent e) {
                String s = "The mouse entered";
                tf.setText(s);
        }
        public void mouseExited(MouseEvent e) {
                String s = "The mouse exited";
                tf.setText(s);
        }
        public void mouseClicked(MouseEvent e) {
                System.exit(0);
        }
        public static void main(String[] args) {
                TwoListener two = new TwoListener();
                two.launchFrame();
        }
}
```

因为采用了继承，这样就能不用重新实现`MouseMoved, MouseReleased, MousePressed`方法

其他修改方法

创建内部类

JAVA

```
package lesson13;
import java.awt.*;
import java.awt.event.*;
public class TwoListener {
        /*...*/

        public void launchFrame() {
                Label label = new Label("Click and drag");
                f.add(label,"North");
                f.add(tf,"South");
                // 加入监听者的时候改为加入自己重写后的监听者
                f.addMouseListener(new MouseHandler());
                f.addMouseMotionListener(new MouseMotionHandler ());
                f.setSize(200, 200);
                f.setVisible(true);
        }

        public static void main(String[] args) {
                TwoListener two = new TwoListener();
                two.launchFrame();
        }
        // 创建继承Adapter的内部类，然后对需要重写的方法在内部重写
        class MouseMotionHandler extends MouseMotionAdapter{
                public void mouseDragged(MouseEvent e) {
                        String s = "Mouse: x = " + e.getX() + "y = " + e.getY();
                        tf.setText(s);
                }
        }

        class MouseHandler extends MouseAdapter{
                public void mouseEntered(MouseEvent e) {
                        String s = "The mouse entered";
                        tf.setText(s);
                }
                public void mouseExited(MouseEvent e) {
                        String s = "The mouse exited";
                        tf.setText(s);
                }
                public void mouseClicked(MouseEvent e) {
                        System.exit(0);
                }
        }
}
```

采用内部类处理事件，这样内部类可直接访问外部类的私有成员TextField对象

采用匿名内部类

可以在表达式内定义匿名内部类，并立即生成实例对象，处理事件

匿名内部类没有构造器！！！

JAVA

```java
import java.awt.*;
import java.awt.event.*;
public class TwoListener {
        private Frame f;
        private TextField tf;

        public TwoListener() {
                f = new Frame("Two LIstener");
                tf = new TextField(30);
        }

        public void launchFrame() {
                Label label = new Label("Click and drag");
                f.add(label,"North");
                f.add(tf,"South");
                // 匿名内部类
                f.addMouseListener(new MouseAdapter (){
                        public void mouseEntered(MouseEvent e) {
                                String s = "The mouse entered";
                                tf.setText(s);
                        }
                        public void mouseExited(MouseEvent e) {
                                String s = "The mouse exited";
                                tf.setText(s);
                        }
                        public void mouseClicked(MouseEvent e) {
                                System.exit(0);
                        }
                });

                f.addMouseMotionListener(new MouseMotionAdapter () {
                        public void mouseDragged(MouseEvent e) {
                                String s = "Mouse: x = " + e.getX() + "y = " +
e.getY();
                                tf.setText(s);
                        }
                });
                f.setSize(300, 200);
                f.setVisible(true);
        }

        public static void main(String[] args) {
                TwoListener two = new TwoListener();
                two.launchFrame();
        }
}
```

CardLayout事件处理举例

JAVA

```java
package lesson13;
import java.awt.*;
import java.awt.event.*;
public class CardTest2 extends WindowAdapter implements MouseListener{
        private Panel[] p = new Panel[3];
        private Label[] lb = new Label[3];
        private Color[] c = {Color.red, Color.green, Color.blue};
        private CardLayout myCard;
        private Frame f;

        public void go() {
                f = new Frame("Card Test");
                myCard = new CardLayout();
                f.setLayout(myCard);
                for(int i =0; i < 3; i++) {
                        p[i] = new Panel();
                        lb[i] = new Label("This is the Panel " + i);
                        p[i].setBackground(c[i]);
                        p[i].add(lb[i]);
                        p[i].addMouseListener(this);
                        f.add(p[i]);
                }
                myCard.show(f, "First");
                f.addWindowFocusListener(this);
                f.setSize(200, 200);
                f.setVisible(true);
        }

        public void mousePressed(MouseEvent e) {
                if(e.isControlDown()) myCard.previous(f);
                else myCard.next(f);
        }

        public void windowClosing(WindowEvent e) {
                System.exit(0);
        }

        public void mouseReleased(MouseEvent e) {}
        public void mouseClicked(MouseEvent e) {}
        public void mouseEntered(MouseEvent e) {}
        public void mouseExited(MouseEvent e) {}

        public static void main(String[] args) {
                CardTest2 ct = new CardTest2();
                ct.go();
        }
}
```

这份代码实现了按按钮切换`Card`界面的功能

Swing

基本组件

基本组件是`JFrame`
`JFrame`窗体类有一个构造方法`JFrame(String title)`，通过它可以创建一个以参数为标题的`JFrame`对象。

当`JFrame`被创建后，它是**不可见**的，必须通过以下方式使JFrame成为可见的：

- 先调用`setSize(int width,int height)`显式设置JFrame的大小，或者调用`pack()`方法自动确定JFrame的大小，`pack()`方法会确保JFrame容器中的组件都会有与布局相适应的合理大小
- 然后调用`setVisible(true)`方法使JFrame成为可见的

```java
import javax.swing.*;
public class SimpleFrame{
        public static void main(String args[]){
                JFrame jFrame=new JFrame("Hello");
                JButton jButton = new JButton("Swing Button");
                jButton.setMnemonic('i'); // 快捷键Alt-i键
                jButton.setToolTipText("Press me"); //鼠标移动提示
                jFrame.add(jButton);
                jFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                //jFrame.setSize(200,100); //设置JFrame的宽和高
                jFrame.pack();
                jFrame.setVisible(true); //使JFrame变为可见
        }
}
```

JFrame的setDefaultCloseOperation(int operation)方法:
参数

- `JFrame.DO_NOTHING_ON_CLOSE`：什么也不做
- `JFrame.HIDE_ON_CLOSE` ：隐藏窗体，这是JFrame的默认选项。
- `JFrame.DISPOSE_ON_CLOSE`：销毁窗体。
- `JFrame.EXIT_ON_CLOSE` ：结束程序

JComponent类的绘图方法

- `Swing`组件`JComponent`类覆盖了`Component`类的`paint()`方法。`JComponent`类的`paint()`方法把绘图任务委派给3个`protected方法`来完成
  - `paintComponent()`：画当前组件
  - `paintBorder()`：画组件的边界
  - `paintChildren()`：如果组件为容器，则画容器所包含的组件
- 自定义的Swing组件的图形绘制只要覆盖`paintComponent()`
- JComponent类的`paintComponent()`方法会以组件的背景色来覆盖整个组件

JAVA

```java
import javax.swing.*;
import java.awt.Color;
import java.awt.Graphics;
public class OvalDrawer extends JPanel implements Runnable{
        private Color[] colors = {Color.RED, Color.BLACK, Color.BLUE, Color.GREEN,
Color.DARK_GREY};
        private Color color;
        private int x = 10, y = 10, width = 10, height = 10;

        public OvalDrawer(){
                new Thread(this).start();  // 创建线程
        }

        public void run(){
                while(true){  // 不断执行此程序
                        x = (int)(Math.random() * 300);
                        y = (int)(Math.random() * 300);
                        width = (int)(Math.random() * 100);
                        height = (int)(Math.random() * 100);
                        color = colors[(int)(Math.random() * (color.length -1)];
                        repaint(); // 请求重绘面板,导致 `paintComponent` 方法被调用
                        try{Thread.sleep(400);
                        }catch(InterruptedException e){
                                throw new RuntimeException(e);
                        }
                }
        }
        public void paintComponent(Graphics g){
                super.paintComponent(g);
                g.setColor(color);
                g.fillOval(x, y, width, height);
        }
        public static void main(String args[]){
                JPanel panel = new OvalDrawer();
                JFrame frame = new JFrame("Hello");
                frame.setContentPane(panel);
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); // 设置了当用户
关闭窗口时应采取的操作
                frame.setSize(300, 300);
                frame.setVisible(true);
        }
}
```

1. `frame.setContentPane(panel);`
   这行代码设置 `JFrame` 的内容面板（content pane）。在这里，`panel` 是一个 `JPanel` 类型的对象，它被设置为 `JFrame` 的内容面板。内容面板是 `JFrame` 中用于放置GUI组件的主要区域。通过将 `panel` 设置为内容面板，你可以在 `JFrame` 中添加和显示 `OvalDrawer` 类中定义的椭圆绘制的面板。
   2.`frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);`

这行代码设置了当用户关闭窗口时应采取的操作。具体而言，`JFrame.EXIT_ON_CLOSE` 表示当用户关闭窗口时，Java应用程序应该终止。这是为了确保当用户关闭窗口时，程序会优雅地退出而不是继续运行。

2. `public void paintComponent(Graphics g)`
   这段代码是在 `OvalDrawer` 类中重写的 `paintComponent` 方法，用于在面板上绘制椭圆。

解释：

1. `super.paintComponent(g);`：首先调用了父类的 `paintComponent` 方法。这是因为在 Swing 中，通常建议在绘制组件之前调用父类的 `paintComponent` 方法。这确保了任何需要在组件上进行的标准绘制操作都会被执行，同时也清理了旧的绘图状态。
2. `g.setColor(color);`：设置绘图上下文的颜色为 `color`，即当前椭圆的颜色。在这个例子中，`color` 是在 `run` 方法中根据随机选择的颜色数组中的一个确定的。
3. `g.fillOval(x, y, width, height);`：使用当前的位置（`x`，`y`）和尺寸（`width`，`height`）绘制一个填充的椭圆。这是实际绘制椭圆的部分，`x` 和 `y` 是椭圆的左上角的坐标，`width` 和 `height` 是椭圆的宽度和高度。
   总体而言，这段代码的作用是在面板上绘制一个填充的椭圆，位置和颜色由 `OvalDrawer` 类的成员变量控制，而这些变量在 `run` 方法中被随机更新。因为 `run` 方法是在单独的线程中运行的，所以椭圆的位置、尺寸和颜色会在一定的时间间隔内不断地变化，从而产生动画效果。

缺省适配器模式

缺省适配器模式是一种设计模式，用于**在一个接口中提供默认的空实现**，使得实现这个接口的类可以**只覆盖感兴趣的方法而不必实现所有方法。**
这样，实现类就不必实现所有接口方法，而是只需要覆盖它们感兴趣的方法，提高了代码的可维护性和灵活性。

举个例子：
和尚的接口是：

```java
interface 和尚{
        public void 吃斋();
        public void 念经();
        public void 撞钟();
        public void 习武();
        public void 打坐();
        public String getName();
}
```

但鲁智深类只实现了习武：

```JAVA
class 鲁智深 implements 和尚{
        public void 习武(){
                拳打镇关西;
                大闹五台山;
                大闹桃花村;
                倒拔垂杨柳;
                火烧瓦官寺;
        }
        public String getName(){
                return "鲁智深";
        }
}
```

那么很明显，由于类没有实现所有的方法，会报错，但是鲁智深确实是一个和尚，那么应该怎么对代码进行修改呢？
于是想到可以用一个抽象类先实现这个接口，再让鲁智深类进行继承

```JAVA
abstract class 天星 interface 和尚{
        public void 吃斋(){}
        public void 念经(){}
        public void 撞钟(){}
        public void 习武(){}
        public void 打坐(){}
        public String getName(){return null;}
}
```

那么抽象类里面的方法均是已实现的内部方法，而且由于在抽象类实现，所以也不用在鲁智深类里面对每一个方法进行重写。使得在未来接口发生变化时，不必修改所有实现类，只需修改适配器即可。

例子：

JAVA

```java
// 接口定义
interface MyListener {
    void method1();
    void method2();
    void method3();
}

// 适配器类提供默认空实现
abstract class MyListenerAdapter implements MyListener {
    @Override
    public void method1() {
        // 默认空实现
    }

    @Override
    public void method2() {
        // 默认空实现
    }

    @Override
    public void method3() {
        // 默认空实现
    }
}

// 实际使用适配器的类
class MyListenerImpl extends MyListenerAdapter {
    // 只需要覆盖感兴趣的方法
    @Override
    public void method2() {
        System.out.println("Overridden method2");
    }
}

public class DefaultAdapterPatternExample {
    public static void main(String[] args) {
        MyListener myListener = new MyListenerImpl();
        myListener.method1(); // 使用默认空实现
        myListener.method2(); // 使用实现类覆盖的方法
        myListener.method3(); // 使用默认空实现
    }
}
```

注意：

课堂小问题(4)

以下哪些说法正确？

a)在AWT的事件处理模型中，一个组件只能注册一个监听器

b)AWT的事件处理采用委托模式

c)用户点击界面上的一个JButton按钮，将触发一个ActionEvent事件

d)在AWT类库中，每个监听接口都有一个相应的适配器类

答案：b c

a）不正确。在AWT的事件处理模型中，一个组件可以注册多个监听器。

b）正确。AWT的事件处理采用委托模式，事件被委托给注册的监听器来处理。

c）正确。当用户点击界面上的 JButton 按钮时，将触发一个 ActionEvent 事件，需要相应的 ActionListener 来处理。

d）不正确。在 AWT 类库中，并不是每个监听接口都有一个相应的适配器类。适配器类主要用于简化实现某个监听器接口时，如果该接口中有多个方法而我们只想实现其中的一部分时。例如，`WindowAdapter` 适配器类用于实现 `WindowListener` 接口的一组方法。然而，并非所有的监听器接口都有相应的适配器类。只有超过1个事件触发的监听器接口有适配器类(7个)