17. GUI事件处理

17.1. GUI组件介绍

- GUI编程（Graphic User Interface，图形用户接口）
- GUI的各种元素，如：容器、按钮、文本框等
- Frame类、Button类、Panel类、Toolkit类、布局管理器、基本组件
- 示例：

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

/**
 * @author xiao儿
 * @date 2019/9/4 11:17
 * @Description MyFrame
 */
public class MyFrame extends Frame implements ActionListener {
    /**
     * 初始化窗体的基本属性
     */
    public MyFrame() {
        this.setSize(600, 400);
        this.setTitle("我的第一个GUI窗体");
        // 创建一个按钮
        Button button = new Button("点我一下，有惊喜");
        // 给按钮添加监听事件
        button.addActionListener(this::actionPerformed);
        // 创建一个线性布局
        FlowLayout flowLayout = new FlowLayout();
        // 把布局应用到窗体上
        this.setLayout(flowLayout);

        // 把按钮添加到窗体上
        this.add(button);
        // 设置窗体可见
        this.setVisible(true);
        // 设置关闭窗口
        this.addWindowListener(new WindowAdapter() {
            @Override
            public void windowClosing(WindowEvent e) {
                super.windowClosing(e);
                System.exit(0);
            }
        });
```

```
    }

    // 单击事件处理的方法
    @Override
    public void actionPerformed(ActionEvent e) {
        System.out.println("惊喜来了，获得100元大红包");
    }

    public static void main(String[] args) {
        new MyFrame();
    }
}
```

17.2. 事件处理

- 事件处理：
  - 事件(</font（Event）：用户对组件的一个操作，称之为一个事件
  - 事件源（Event source)：产生事件的对象
  - 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，事件监听器
- 为了简化编程，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的方法，但不做任何事情
- 接口回调：

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

/**
 * @author xiao儿
 * @date 2019/9/4 14:43
 * @Description Frame2
 *
 * 接口回调：
 * 当一个对象需要给外部对象提供数据时，我们可以定义一个内部接口把数据通过接口传递出去
 * 所有外部对象需要这个数据时，就实现这个接口
 * 好处是传递数据的对象不直接依赖接收数据的对象（低耦合）
 */
public class Frame2 extends Frame {
```

```
16      private TextField textField = new TextField(20);
17      private Button button = new Button("付款");
18
19      public Frame2() {
20          this.setSize(400, 300);
21          this.setLayout(new FlowLayout());
22          this.add(textField);
23          this.add(button);
24          button.addActionListener(new ActionListener() {
25              @Override
26              public void actionPerformed(ActionEvent e) {
27                  String money = textField.getText();
28                  moneyListener.setMoney(money);
29              }
30          });
31          this.setVisible(true);
32      }
33
34      private MoneyListener moneyListener;
35
36      public void setMoneyListener(MoneyListener
    moneyListener) {
37          this.moneyListener = moneyListener;
38      }
39
40      public static interface MoneyListener {
41          public void setMoney(String money);
42      }
43  }
```

17.3. 观察者模式

- 观察者模式原理：简单的说，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

- 观察者模式的作用：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。由于被观察者和观察者没有紧密的耦合在一起，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。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
- 示例：

```
// MessageSubject
/**
 * @author xiao儿
 * @date 2019/9/4 15:33
 * @Description MessageSubject
 *
 * 被观察者的接口
 */
public interface MessageSubject {
    // 注册观察者
    public void registerObserver(Observer o);
    // 移除观察者
    public void removeObserver(Observer o);
    // 通知所有观察者
    public void notifyObserver();
}

// Observer
/**
 * @author xiao儿
 * @date 2019/9/4 15:36
 * @Description Observer
 *
 * 观察者接口
 */
public interface Observer {
    // 更新消息
    public void update(String message);
}
```

```java

// Message
import java.util.ArrayList;
import java.util.List;

/**
 * @author xiao儿
 * @date 2019/9/4 15:41
 * @Description Message
 *
 * 具体的被观察者
 */
public class Message implements MessageSubject {
    // 维护的观察者列表
    private List<Observer> list = new ArrayList<>();

    private String message;

    public void setMessage(String message) {
        this.message = message;
        notifyObserver();
    }

    @Override
    public void registerObserver(Observer o) {
        list.add(o);
    }

    @Override
    public void removeObserver(Observer o) {
        list.remove(o);
    }

    @Override
    public void notifyObserver() {
        for (int i = 0; i < list.size(); i++) {
            Observer observer = list.get(i);
```

```
            observer.update(message);
        }
    }
}

// User
/**
 * @author xiao儿
 * @date 2019/9/4 15:58
 * @Description User
 *
 * 具体的观察者
 */
public class User implements Observer {
    private String name;

    public User(String name) {
        this.name = name;
    }

    @Override
    public void update(String message) {
        System.out.println("[" + name + "]" + "收到消息: " + message);
    }
}

// ObserverDesignPattern
import org.junit.Test;

/**
 * @author xiao儿
 * @date 2019/9/4 16:00
 * @Description ObserverDesignPattern(
 */
public class ObserverDesignPattern {
    @Test
```

```
103     public void testObserver() {
104         Message message = new Message();
105         Observer user = new User("Tom");
106         Observer user1 = new User("Lily");
107         Observer user2 = new User("Job");
108         message.registerObserver(user);
109         message.registerObserver(user1);
110         message.registerObserver(user2);
111 
112         message.setMessage("大家好");
113 
114         message.removeObserver(user);
115 
116         message.setMessage("大家在吗？");
117     }
118 }
```