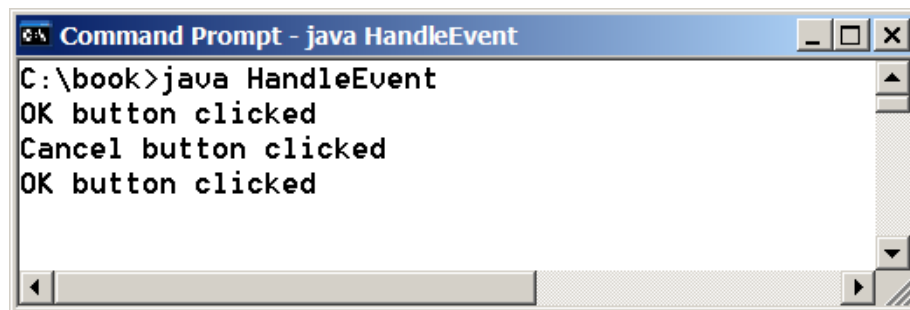
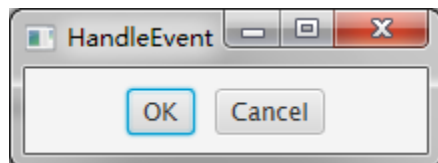


Chapter 15 Event-Driven Programming and Animations

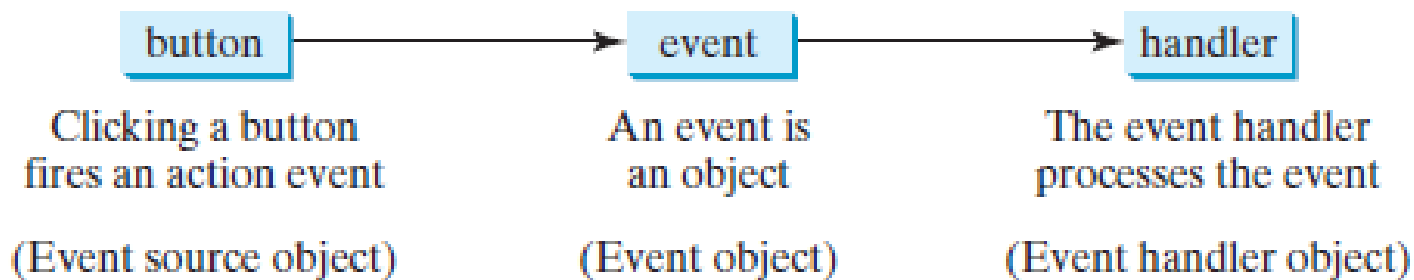
过程式编程（Procedural Programming）与 事件驱动编程（Event-Driven Programming）

- 过程式编程，程序的执行顺序是按照过程所预定义的顺序。
- 事件驱动编程中，某部分的程序是否执行，取决于相应的事件是否被触发。

先看一个例子



- 用户点OK和点Cancel按钮的时候，分别输出提示文字。为了响应按钮，Java需要三个对象参与：1. 有一个事件源对象（按钮对象）；2. 发生了一个事件（事件对象）；3. 有一个处理器处理了这个事件（事件处理器对象）。



如何制作事件处理器？

- 不是所有的对象都能够处理事件。事件处理器也是一个类，但是必须满足两点：
 1. 必须是接口**EventHandler<T extends Event>**的实例，**<T extends Event>**表示T是Event的泛型，且是Event的子类；
 2. 必须利用**source.setAction(handler)**注册到事件源对象上，从而关联事件源对象和事件。

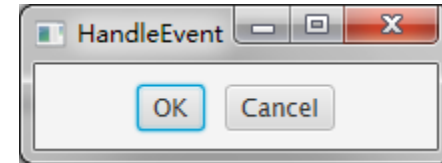
事件处理类的例子

- **EventHandler<ActionEvent>** 接口只定义了一个方法 **handle(ActionEvent)**，这个方法用来处理事件。
- 所以，标准的事件处理器长得象这样：

```
class MyHandlerClass implements EventHandler<ActionEvent> {  
    @Override  
    public void handle(ActionEvent e) {  
        //在这里编写处理事件代码，参数e可以查询事件相关信息  
    }  
}
```

```
import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
public class HandleEvent extends Application {
    public void start(Stage primaryStage) {
        HBox pane = new HBox(10);
        pane.setAlignment(Pos.CENTER);
        Button btOK = new Button("OK");
        Button btCancel = new Button("Cancel");
        OKHandlerClass handler1 = new OKHandlerClass(); //创建处理器
        btOK.setOnAction(handler1); //注册处理器到OK按钮上

        CancelHandlerClass handler2 = new CancelHandlerClass();
        btCancel.setOnAction(handler2);
        pane.getChildren().addAll(btOK, btCancel);
    }
}
```



```
    Scene scene = new Scene(pane);
    primaryStage.setTitle("HandleEvent"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage
}
}

class OKHandlerClass implements EventHandler<ActionEvent> {
    @Override
    public void handle(ActionEvent e) {
        System.out.println("OK button clicked");
    }
}

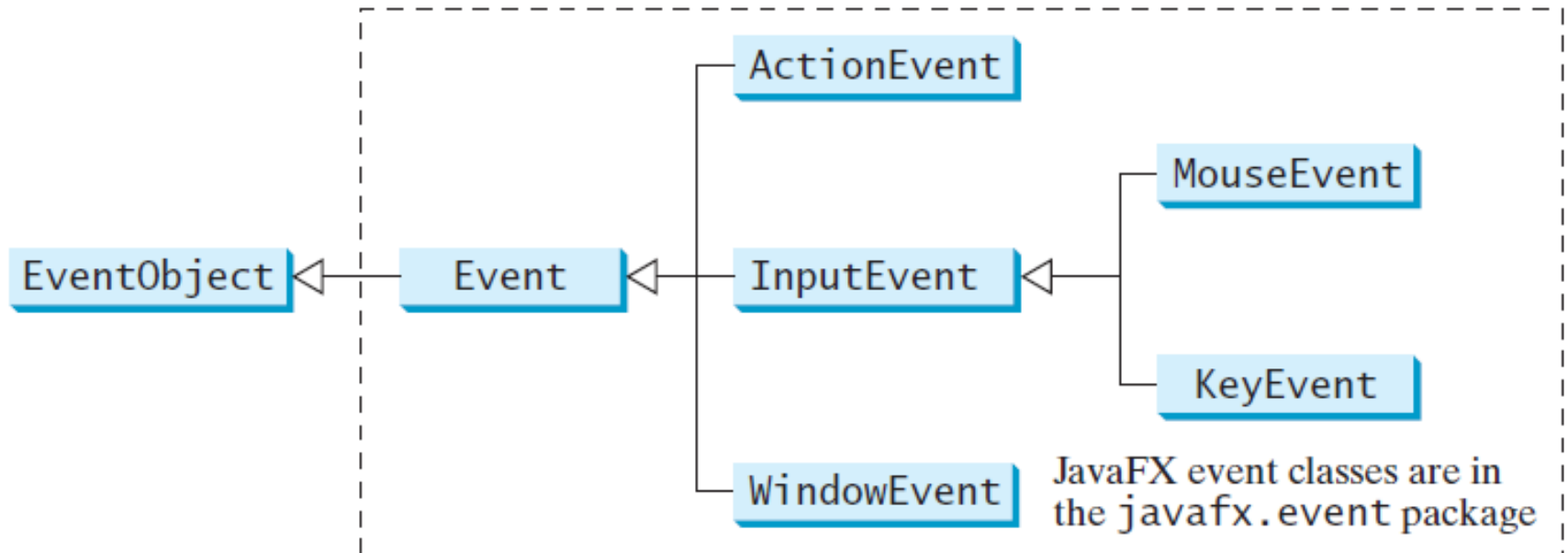
class CancelHandlerClass implements EventHandler<ActionEvent> {
    @Override
    public void handle(ActionEvent e) {
        System.out.println("Cancel button clicked");
    }
}
```

事件和事件源 Events and Event Sources

- 事件被定义为一个对象，由事件源负责创建。
- 触发一个事件，意味着事件对象会被创建，并且会转发给处理器进行处理（如果有注册自己的处理器，则自己的处理器优先处理，否则由JVM处理）。
- 当一个Java GUI开始运行，程序随着跟用户交互的过程中记录各种事件，并驱动程序运行，这就是所谓的事件驱动编程（*event-driven programming*）
- 事件可能由用户在交互过程中触发，例如鼠标移动，鼠标点击或者敲键盘；事件也可能由系统触发，例如定时器事件，就是每隔一段时间触发一次。

事件类

- 能够创建事件并将其触发的组件，叫做**事件源对象***event source object*，或者**源对象***source object*，或者**源组件***source component*。例如按钮就是一个源对象，因为它会创建按钮点击事件，并将其触发。
- 所有的事件都是**java.util.EventObject**的子类，如图：



事件信息

- 事件对象 **EventObject** 包含了事件发生的相关信息，例如 **getSource()** 方法，可以得到事件发生的事件源对象（例如 **OK** 按钮）。
- **EventObject** 的子类则定义了更加具体的事件对象，例如按钮事件，窗口事件，按键事件等等。
- 如果一个组件可以触发某种事件，它的子类也会触发相同的事件。例如 **JavaFX** 的 **shape**, 布局窗格 **layout pane** 以及控件，都能触发 **MouseEvent** 和 **KeyEvent**，因为结点类 **Node** 是它们的超类。

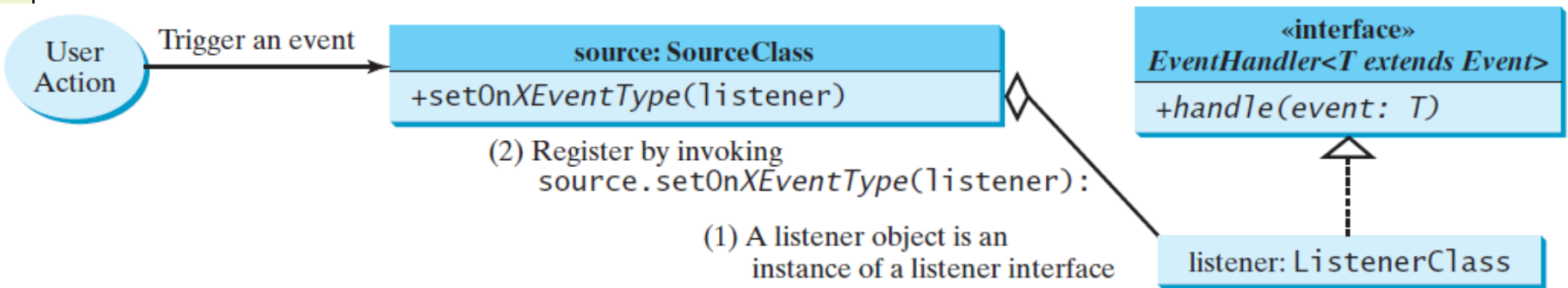
用户事件、源对象、触发事件类型、事件注册方法

<i>User Action</i>	<i>Source Object</i>	<i>Event Type Fired</i>	<i>Event Registration Method</i>
Click a button	Button	ActionEvent	setOnAction(EventHandler<ActionEvent>)
Press Enter in a text field	TextField	ActionEvent	setOnAction(EventHandler<ActionEvent>)
Check or uncheck	RadioButton	ActionEvent	setOnAction(EventHandler<ActionEvent>)
Check or uncheck	CheckBox	ActionEvent	setOnAction(EventHandler<ActionEvent>)
Select a new item	ComboBox	ActionEvent	setOnAction(EventHandler<ActionEvent>)
Mouse pressed	Node, Scene	MouseEvent	setOnMousePressed(EventHandler<MouseEvent>)
Mouse released			setOnMouseReleased(EventHandler<MouseEvent>)
Mouse clicked			setOnMouseClicked(EventHandler<MouseEvent>)
Mouse entered			setOnMouseEntered(EventHandler<MouseEvent>)
Mouse exited			setOnMouseExited(EventHandler<MouseEvent>)
Mouse moved			setOnMouseMoved(EventHandler<MouseEvent>)
Mouse dragged			setOnMouseDragged(EventHandler<MouseEvent>)
Key pressed	Node, Scene	KeyEvent	setOnKeyPressed(EventHandler<KeyEvent>)
Key released			setOnKeyReleased(EventHandler<KeyEvent>)
Key typed			setOnKeyTyped(EventHandler<KeyEvent>)

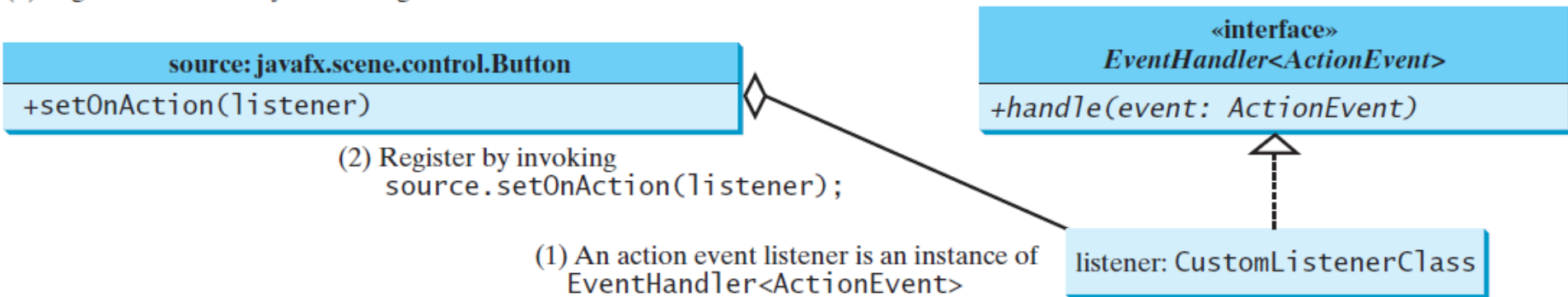
注册事件处理器并处理事件

- Java采用事件代理模型（delegation-based model）来处理事件：源对象触发一个事件，对此事件感兴趣的对象再处理它。后者称为事件处理器或者事件监听器（*event handler* or *event listener*）。事件处理的两个要素如下：
 1. **处理器必须是对应的事件处理接口**。JavaFX为每一种事件T定义了唯一的处理接口**EventHandler<T extends Event>**，处理接口则包含了**handle(T e)**方法用来处理事件。
 2. **处理器必须注册到源对象上**。注册的方法依赖于事件类型。对**ActionEvent**来说，注册方法是**setOnAction**，对鼠标点击事件来说，注册方法则是**setOnMousePressed**，这从上一页的表格可以查到。

事件处理模型



(a) A generic source object with a generic event T



(b) A Button source object with an ActionEvent

代理模型的例子

//Step 1 事件源对象

```
Button btOK = new Button("OK");
```

//Step 2 处理器对象

```
OKHandlerClass handler1 = new OKHandlerClass();
```

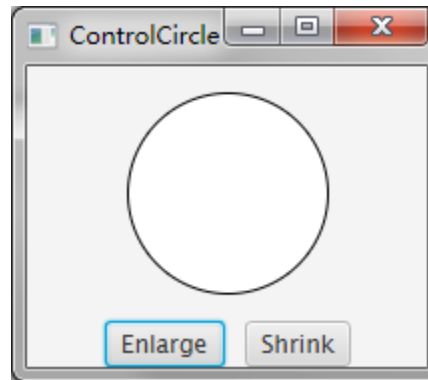
//Step 3 注册处理器，从而关联事件源对象和处理器对象

```
btOK.setAction(handler1);
```

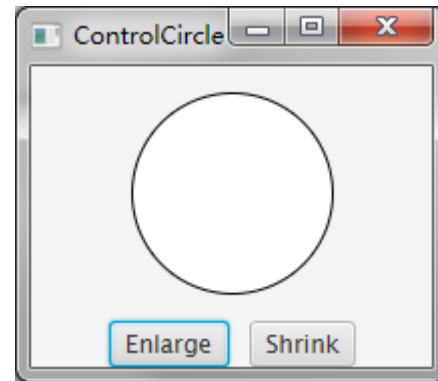
当用户点击按钮时，上面的btOK会触发ActionEvent事件，传递这个事件的信息，并调用处理器的handler1的handle(ActionEvent)方法进行处理。

例子：ControlCircle

- 写一个如图的程序，用两个按钮来控制圆的大小。



思路分析



- 由于Java的源对象和处理器对象可以分开创建，只要最后通过注册关联即可。为了简化，我们先不考虑变大变小问题，搭好界面再说。
- 复杂的界面总是一堆pane的组合，我们这么来设计（有N种方法可以做出相同的界面）：
 1. 主界面用BorderPane，圆放在Center，两个按钮放在Bottom；
 2. 圆放在StackPane上；
 3. 两个按钮放在HBox上。

关键代码

```
15 StackPane pane = new StackPane();
16 Circle circle = new Circle(50);
17 circle.setStroke(Color.BLACK);
18 circle.setFill(Color.WHITE);
19 pane.getChildren().add(circle);
20
21 HBox hBox = new HBox();
22 hBox.setSpacing(10);
23 hBox.setAlignment(Pos.CENTER);
24 Button btEnlarge = new Button("Enlarge");
25 Button btShrink = new Button("Shrink");
26 hBox.getChildren().add(btEnlarge);
27 hBox.getChildren().add(btShrink);
28
29 BorderPane borderPane = new BorderPane();
30 borderPane.setCenter(pane);
31 borderPane.setBottom(hBox);
32 borderPane.setAlignment(hBox, Pos.CENTER);
33
34 // Create a scene and place it in the stage
35 Scene scene = new Scene(borderPane, 200, 150);
36 primaryStage.setTitle("ControlCircle"); // Set the stage title
37 primaryStage.setScene(scene); // Place the scene in the stage
38 primaryStage.show(); // Display the stage
```

接下来添加事件处理

1. 圆的大小是由半径决定的，只要在变大变小的方法中修改圆半径，然后重绘圆即可。为便于修改半径，我们单独定义一个 **CirclePane** 用来在窗格中显示圆，并把变大变小的方法也包装到这个类中。这就是OOP的封装思想。
2. 在主类 **ControlCircle** 中创建一个 **CirclePane** 对象，并声明 **circlePane** 引用这个对象，以便通过这个引用控制圆的缩放。
3. 定义一个处理类 **EnlargeHandler**，实现接口 **EventHandler<ActionEvent>**。为了让 **circlePane** 能够被 **handle** 方法访问到，将 **EnlargeHandler** 定义为 **ControlCircle** 的内部类（关于内部类的进一步阐述见下一节）。
4. 把处理器注册到按钮上，并在 **handle** 方法中调用 **circlePane.enlarge()**。

代码一 1/3

```
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.StackPane;
import javafx.scene.layout.HBox;
import javafx.scene.layout.BorderPane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.stage.Stage;

public class ControlCircle extends Application {
    private CirclePane circlePane = new CirclePane();
    @Override
    public void start(Stage primaryStage) {
        // Hold two buttons in an HBox
        HBox hBox = new HBox();
        hBox.setSpacing(10);
        hBox.setAlignment(Pos.CENTER);
        Button btEnlarge = new Button("Enlarge");
        Button btShrink = new Button("Shrink");
        hBox.getChildren().add(btEnlarge);
        hBox.getChildren().add(btShrink);
    }
}
```

代码—2/3

```
// Create and register the handler
btEnlarge.setOnAction(new EnlargeHandler());

BorderPane borderPane = new BorderPane();
borderPane.setCenter(circlePane);
borderPane.setBottom(hBox);
BorderPane.setAlignment(hBox, Pos.CENTER);

// Create a scene and place it in the stage
Scene scene = new Scene(borderPane, 200, 150);
primaryStage.setTitle("ControlCircle"); // Set the stage title
primaryStage.setScene(scene); // Place the scene in the stage
primaryStage.show(); // Display the stage
}

class EnlargeHandler implements EventHandler<ActionEvent> {
    @Override
    // Override the handle method
    public void handle(ActionEvent e) {
        circlePane.enlarge();
    }
} // End of EnlargeHandler
} // End of ControlCircle
```

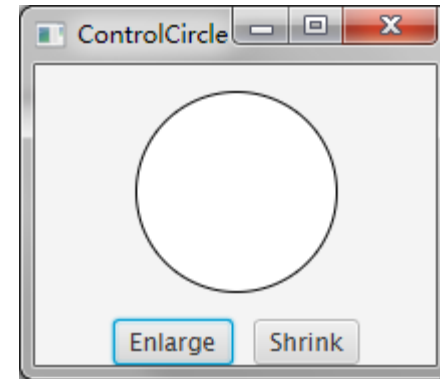
```
class CirclePane extends StackPane {  
    private Circle circle = new Circle(50);
```

```
    public CirclePane() {  
        getChildren().add(circle);  
        circle.setStroke(Color.BLACK);  
        circle.setFill(Color.WHITE);  
    }
```

```
    public void enlarge() {  
        circle.setRadius(circle.getRadius() + 2);  
    }
```

```
    public void shrink() {  
        circle.setRadius(circle.getRadius() > 2 ? circle.getRadius() - 2 :  
circle.getRadius());  
    }  
} // End of CirclePane
```

代码—3/3



- Circle类被重设半径后，Java会自动重绘，不需要另外刷新界面。

内部类Inner Classes

- 内部类：一个类完整放在另外一个类中，作为另外一个类的成员。内部类可以**直接访问**外部类的数据和方法。

```
public class Test {  
    ...  
}  
  
public class A {  
    ...  
}
```

(a)

```
public class Test {  
    ...  
  
    // Inner class  
    public class A {  
        ...  
    }  
}
```

```
// OuterClass.java: inner class demo  
public class OuterClass {  
    private int data;  
  
    /** A method in the outer class */  
    public void m() {  
        // Do something  
    }  
  
    // An inner class  
    class InnerClass {  
        /** A method in the inner class */  
        public void mi() {  
            // Directly reference data and method  
            // defined in its outer class  
            data++;  
            m();  
        }  
    }  
}
```

关于内部类一1

- 内部类会被编译为 *OuterClassName\$InnerClassName.class*
- 例如，内部类 **A** 如果在外部类 **Test** 中定义，将被编译为 **Test\$A.class**
- 由于可以直接访问外部类的成员，减少了参数传递的工作，内部类可以使程序简洁明了。
- 内部类可以是静态的，不过一个静态的内部类只能访问外部类的静态成员。

关于内部类一2

- 内部类可以定义自己的成员为public, protected, 或 private, 其访问规则和前面的学过的普通类没有区别。
- 内部类一般是在外部类中被创建和使用。其实内部类也可以被外部类之外的类使用, 就是用起来麻烦点, 所以一般不推荐这么使用。真要用的话, 可以这样:
 - 对于非静态内部类, 需要这样创建:

```
OuterClass.InnerClass innerObject = outerObject.new InnerClass();
```

- 对于静态内部类, 需要这样创建:

```
OuterClass.InnerClass innerObject = new OuterClass.InnerClass();
```


匿名内部类处理器


Anonymous Inner Class Handlers

- 如果一个内部类没有类名，称之为匿名内部类。使用匿名内部类可以简化事件处理器的写法。匿名内部类的语法如下：

```
new SuperClassName/InterfaceName() {  
    // Implement or override methods in superclass or interface  
    // Other methods if necessary  
}
```

使用匿名内部类改写 ControlCircle 的例子

```
public void start(Stage primaryStage) {  
    // Omitted  
  
    btEnlarge.setOnAction(  
        new EnlargeHandler());  
}  
  
class EnlargeHandler  
    implements EventHandler<ActionEvent> {  
    public void handle(ActionEvent e) {  
        circlePane.enlarge();  
    }  
}
```



左上是内部类版本，右下是匿名内部类版本，注意删除线部分。

```
public void start(Stage primaryStage) {  
    // Omitted  
  
    btEnlarge.setOnAction(  
        new class EnlargeHandler  
            implements EventHandler<ActionEvent>() {  
                public void handle(ActionEvent e) {  
                    circlePane.enlarge();  
                }  
            }  
    ));  
}
```

匿名内部类

- 匿名内部类总是继承自一个类或者实现一个接口，但是不会显式出现`extends` 或 `implements` 的字眼；
- 匿名内部类必须实现超类或者接口所定义的所有抽象方法；
- 匿名内部类总是使用无参构造方法，从它的超类创建一个实例。如果一个匿名内部类实现了一个接口，其构造方法就是`Object()`；
- 匿名内部类会被编译为`OuterClassName$n.class`的形式。例如，假设外部类`Test` 有两个匿名内部类，这两个内部类依照出现的顺序，会被依次编译为`Test$1.class`和`Test$2.class`。

更复杂的例子

- 使用匿名内部类实现以下程序功能。

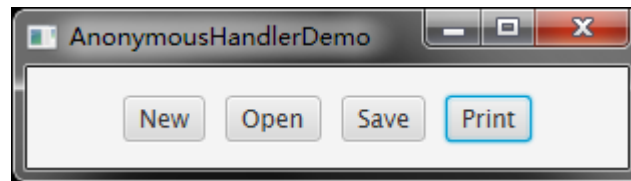
- 输出如下：

Process New

Process Open

Process Save

Process Print



- 思路：定义4个按钮，然后为4个按钮添加4个匿名内部类作为事件处理器。

为New按钮加事件处理器的代码

```
Button btNew = new Button("New");  
btNew.setAction(new EventHandler<ActionEvent>() {  
    @Override // Override the handle method  
    public void handle(ActionEvent e) {  
        System.out.println("Process New");  
    }  
});
```

其实Java 8还支持一种更彪悍的写法，先睹为快：

```
btNew.setAction(e -> System.out.println("Process New"));
```

代码—1/3

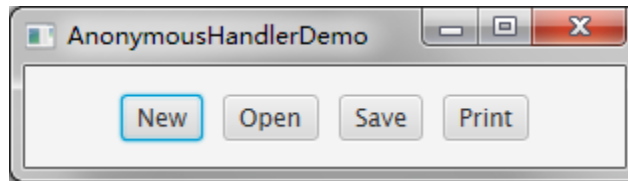
```
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

public class AnonymousHandlerDemo extends Application {
    public void start(Stage primaryStage) {
        // Hold two buttons in an HBox
        HBox hBox = new HBox();
        hBox.setSpacing(10);
        hBox.setAlignment(Pos.CENTER);
        Button btNew = new Button("New");
        Button btOpen = new Button("Open");
        Button btSave = new Button("Save");
        Button btPrint = new Button("Print");
        hBox.getChildren().addAll(btNew, btOpen, btSave, btPrint);
    }
}
```

```
btNew.setOnAction(new EventHandler<ActionEvent>() {  
    public void handle(ActionEvent e) {  
        System.out.println("Process New");  
    }  
});  
  
btOpen.setOnAction(new EventHandler<ActionEvent>() {  
    public void handle(ActionEvent e) {  
        System.out.println("Process Open");  
    }  
});  
  
btSave.setOnAction(new EventHandler<ActionEvent>() {  
    public void handle(ActionEvent e) {  
        System.out.println("Process Save");  
    }  
});  
  
btPrint.setOnAction(new EventHandler<ActionEvent>() {  
    public void handle(ActionEvent e) {  
        System.out.println("Process Print");  
    }  
});
```

代码—2/3

代码一3/3



```
// Create a scene and place it in the stage
Scene scene = new Scene(hBox, 300, 50);
primaryStage.setTitle("AnonymousHandlerDemo"); // Set title
primaryStage.setScene(scene); // Place the scene in the
stage
primaryStage.show(); // Display the stage
}
}
```


另外一种实现思路

- 如果按钮的功能较为类似，可以考虑共用同一个事件处理器，然后使用ActionEvent的getSource()方法，判断事件由哪个对象触发，再转入相应处理部分。
- 这种写法的优点是，代码比较清晰，尤其是多个事件处理器的功能基本一致的时候（例如做一个计算器，数字按钮0-9的事件处理器）。
- 本题中，4个按钮的功能基本相同，就是显示Process + 按钮上的文字，所以共用同一个事件处理器也是可以的。

将4个按钮注册给同一个handler处理

```
// Create and register the handler  
ButtonHandler handler = new ButtonHandler();  
btNew.setAction(handler);  
btOpen.setAction(handler);  
btSave.setAction(handler);  
btPrint.setAction(handler);
```

- 可以利用上述代码替换P31的代码

另外补一个ButtonHandler类，
是否做成内部类无所谓。

```
class ButtonHandler implements EventHandler<ActionEvent> {  
    public void handle(ActionEvent e) {  
        System.out.println("Process " + ((Button)  
e.getSource()).getText());  
    }  
}
```

- e.getSource() 可以得到触发该事件的源对象，本例中是4个按钮，所以可以将其强行转换为Button类。
- getText() 可以取得控件上的文本，例如按钮上的文本，标签上的文本，输入框的文本，都可以这样获取。

利用Lambda Expression简化事件处理器的写法

- *Lambda*表达式是 Java 8新引入的特性，可以极大简化匿名内部类的写法。例如，左边的代码可以用右边完全代替（1行顶过去5行，有木有？）

```
btEnlarge.setOnAction(  
    new EventHandler<ActionEvent>() {  
        @Override  
        public void handle(ActionEvent e) {  
            // Code for processing event e  
        }  
    }  
);
```

```
btEnlarge.setOnAction(e -> {  
    // Code for processing event e  
});
```

*Lambda*表达式的基本语法

- *Lambda*表达式用法有两种:

(type1 param1, type2 param2, ...) -> expression

- 或者

(type1 param1, type2 param2, ...) -> { statements; }

- 数据类型可以显式指定，也可以由编译器自行确定。
如果只有一个参数，小括号可以省略不写，所以前一个例子才可以这么用：

e -> {

 // Code for processing event e

}

Lambda表达式的实现原理

- 编译器把lambda表达式当作匿名内部类创建的一个对象；然后，编译器知道该对象必须是EventHandler<ActionEvent>的实例，所以会首先自动创建一个实例；接下来，因为EventHandler接口只定义了一个handle方法，并且该方法只接收一个ActionEvent类型的参数，于是编译器自动将e识别成ActionEvent类型，并将语句识别为handle方法的方法体。
- EventHandler 接口**有且只有**一个方法，所以语句全部被作为方法体。如果一个接口定义了多个方法，编译器将无法编译lambda表达式。
- 为了让编译器理解lambda表达式，接口必须**有且只有**一个抽象方法，这种接口就是所谓的**功能接口functional interface**或**单一抽象方法 Single Abstract Method (SAM)接口**。

再次重写P31的代码，注意4种写法略有不同

```
// Create and register the handler
```

```
btNew.setAction((ActionEvent e) -> { //显式指明参数类型  
    System.out.println("Process New");  
});
```

```
btOpen.setAction((e) -> { //由编译器确定参数类型  
    System.out.println("Process Open");  
});
```

```
btSave.setAction(e -> { //单一参数，可省略小括号  
    System.out.println("Process Save");  
});
```

```
btPrint.setAction(e -> System.out.println("Process  
Print")); //只有一条语句，可省略大括号
```

鼠标事件Mouse Events

- 鼠标在结点或者场景上的按下，释放，点击，移动，拖动，进入，离开.....都能触发MouseEvent事件。

`javafx.scene.input.MouseEvent`

```
+getButton(): MouseButton  
+getClickCount(): int  
+getX(): double  
+getY(): double  
+getSceneX(): double  
+getSceneY(): double  
+getScreenX(): double  
+getScreenY(): double  
+isAltDown(): boolean  
+isControlDown(): boolean  
+isMetaDown(): boolean  
+isShiftDown(): boolean
```

Indicates which mouse button has been clicked.

Returns the number of mouse clicks associated with this event.

Returns the x-coordinate of the mouse point in the event source node.

Returns the y-coordinate of the mouse point in the event source node.

Returns the x-coordinate of the mouse point in the scene.

Returns the y-coordinate of the mouse point in the scene.

Returns the x-coordinate of the mouse point in the screen.

Returns the y-coordinate of the mouse point in the screen.

Returns true if the `Alt` key is pressed on this event.

Returns true if the `Control` key is pressed on this event.

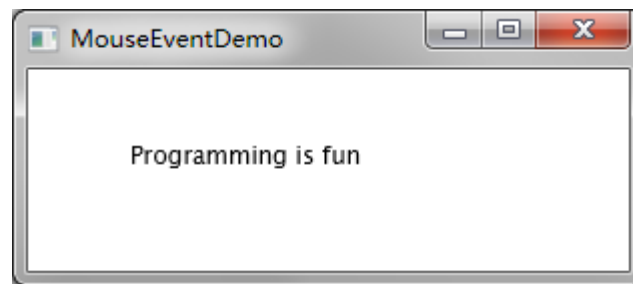
Returns true if the mouse `Meta` button is pressed on this event.

Returns true if the `Shift` key is pressed on this event.

鼠标按键 **MouseButton**

- **MouseButton**定义了4个常量：**PRIMARY**, **SECONDARY**, **MIDDLE**, **NONE**用来表示鼠标按键。4个常量分别表示鼠标的左键，右键，中键和没有按键。
- 可以利用**MouseEvent**的**getButton()**方法，判断用户按下的鼠标键是哪个，例如：
getButton() == MouseButton.SECONDARY
表示用户按下的是右键。

例题



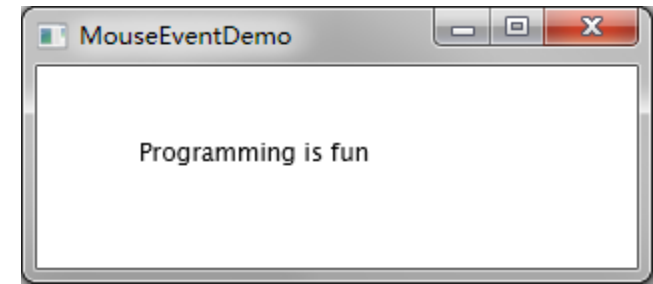
- 写一个程序，实现用鼠标拖动一行文本。
- 解题思路：
 1. 参考表15.1，鼠标拖动需要注册的事件处理器是 **`setOnMouseDragged(EventHandler<MouseEvent>)`**
 2. 文本显示的位置，可以通过`setX`和`setY`改变。只要从鼠标事件`e`中获取鼠标位置，然后用这个鼠标位置去修改文本位置。

```

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.text.Text;
import javafx.stage.Stage;

public class MouseEventDemo extends Application {
    public void start(Stage primaryStage) {
        Pane pane = new Pane();
        Text text = new Text(20, 20, "Programming is fun");
        pane.getChildren().addAll(text);
        text.setOnMouseDragged(e -> { // 注册文本的鼠标拖动事件
            text.setX(e.getX());
            text.setY(e.getY());
        });
        Scene scene = new Scene(pane, 300, 100);
        primaryStage.setTitle("MouseEventDemo"); // Set the stage title
        primaryStage.setScene(scene); // Place the scene in the stage
        primaryStage.show(); // Display the stage
    }
}

```



按键事件Key Events

- 在结点或场景上按下，释放，键入，都能够触发 KeyEvent 事件。
- 控件按键事件的捕捉有个前提，就是控件需要拥有键盘焦点。

javafx.scene.input.KeyEvent

```
+getCharacter(): String  
+getCode(): KeyCode  
+getText(): String  
+isAltDown(): boolean  
+isControlDown(): boolean  
+isMetaDown(): boolean  
+isShiftDown(): boolean
```

Returns the character associated with the key in this event.

Returns the key code associated with the key in this event.

Returns a string describing the key code.

Returns true if the Alt key is pressed on this event.

Returns true if the Control key is pressed on this event.

Returns true if the mouse Meta button is pressed on this event.

Returns true if the Shift key is pressed on this event.

按键常量KeyCode

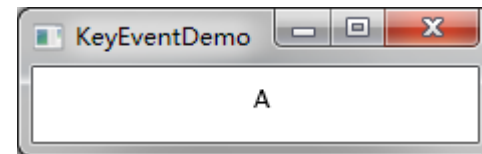
- 为区分每一个按键，KeyCode为每个按键定义了唯一的编码（如图），可以通过按键事件的getCode()得到用户按下的按键。

<i>Constant</i>	<i>Description</i>	<i>Constant</i>	<i>Description</i>
HOME	The Home key	CONTROL	The Control key
END	The End key	SHIFT	The Shift key
PAGE_UP	The Page Up key	BACK_SPACE	The Backspace key
PAGE_DOWN	The Page Down key	CAPS	The Caps Lock key
UP	The up-arrow key	NUM_LOCK	The Num Lock key
DOWN	The down-arrow key	ENTER	The Enter key
LEFT	The left-arrow key	UNDEFINED	The <code>keyCode</code> unknown
RIGHT	The right-arrow key	F1 to F12	The function keys from F1 to F12
ESCAPE	The Esc key	0 to 9	The number keys from 0 to 9
TAB	The Tab key	A to Z	The letter keys from A to Z

```

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.text.Text;
import javafx.stage.Stage;
public class Hello extends Application {
    public void start(Stage primaryStage) {
        Pane pane = new Pane();
        Text text = new Text(20, 20, "A");
        pane.getChildren().add(text);
        text.setOnKeyPressed(e -> {
            switch (e.getCode()) {
                case DOWN: text.setY(text.getY() + 10); break;
                case UP:   text.setY(text.getY() - 10); break;
                case LEFT: text.setX(text.getX() - 10); break;
                case RIGHT: text.setX(text.getX() + 10); break;
                default: if (Character.isLetterOrDigit(e.getText().charAt(0)))
                        text.setText(e.getText());
            }
        });
        Scene scene = new Scene(pane);
        primaryStage.setTitle("KeyEventDemo"); // Set the stage title
        primaryStage.setScene(scene); // Place the scene in the stage
        primaryStage.show(); // Display the stage
        text.requestFocus(); // text is focused to receive key input
    }
}

```



此行重要，如果文本
没有焦点，键盘事件
不会交给文本处理

可观察对象的监听器Listeners

- 可以用监听器来监听可观察对象，这样当其值改变时，监听器可以收到事件通知。
- 这种监听器必须实现InvalidationListener接口，并覆盖接口所定义的invalidated(Observable o)方法。
- 看个例子。

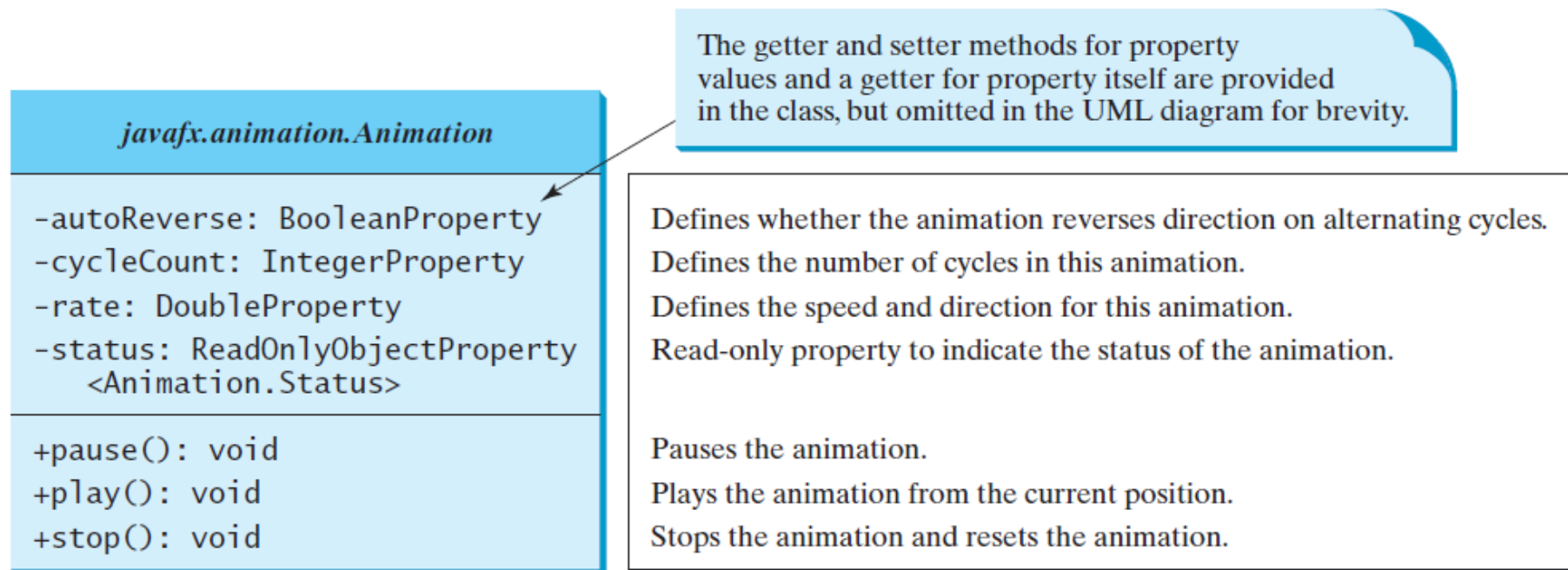
给balance加上监听器的例子

```
1 import javafx.beans.InvalidationListener;
2 import javafx.beans.Observable;
3 import javafx.beans.property.DoubleProperty;
4 import javafx.beans.property.SimpleDoubleProperty;
5
6 public class ObservablePropertyDemo {
7     public static void main(String[] args) {
8         DoubleProperty balance = new SimpleDoubleProperty();
9         balance.addListener(new InvalidationListener() {
10             public void invalidated(Observable ov) {
11                 System.out.println("The new value is " +
12                     balance.doubleValue());
13             }
14         });
15
16         balance.set(4.5);
17     }
18 }
```

The new value is 4.5

动画Animation

- JavaFX提供了一个抽象类Animation，专门用于制作各种动画效果。下面是Animation的UML图。



路径过渡PathTransition

- 路径过渡可以让一个结点沿着指定路径移动。它是Animation的子类。
- 这个类有一个duration成员，用来描述动画的持续时间。
- 还有一个成员是orientation，用来描述移动过程中结点是否需要和移动方向（路径的切线）保持垂直。

旋转的矩形

— 1/2

```
import javafx.animation.PathTransition;
import javafx.animation.Timeline;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.scene.shape.Circle;
import javafx.stage.Stage;
import javafx.util.Duration;

public class PathTransitionDemo extends Application {
    public void start(Stage primaryStage) {
        Pane pane = new Pane();
        Rectangle rectangle = new Rectangle(0, 0, 25, 50);
        rectangle.setFill(Color.ORANGE);
        Circle circle = new Circle(125, 100, 50);
        circle.setFill(Color.WHITE);
        circle.setStroke(Color.BLACK);

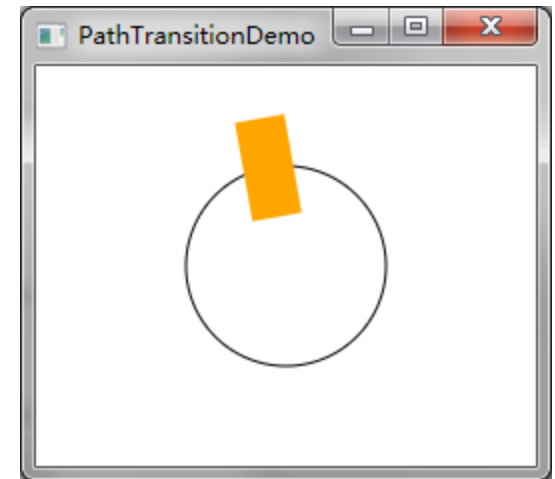
        pane.getChildren().add(circle);
        pane.getChildren().add(rectangle);
    }
}
```

旋转的矩形

— 2/2

```
// Create a path transition
PathTransition pt = new PathTransition();
pt.setDuration(Duration.millis(4000));
// 定义移动路径，只要是Shape都能当路径
// 例如要让矩形上下移动，可以用: new Line(125, 200, 125, 0)
pt.setPath(circle);
pt.setNode(rectangle);
pt.setOrientation(PathTransition.OrientationType.ORTHOGONAL_TO_TANGENT);
pt.setCycleCount(Timeline.INDEFINITE);
pt.setAutoReverse(true);
pt.play(); // Start animation
```

```
// 在circle上定义鼠标事件处理器，按下左键暂停，松开继续
circle.setOnMousePressed(e -> pt.pause());
circle.setOnMouseReleased(e -> pt.play());
// Create a scene and place it in the stage
Scene scene = new Scene(pane, 250, 200);
primaryStage.setTitle("PathTransitionDemo"); // Set the stage title
primaryStage.setScene(scene); // Place the scene in the stage
primaryStage.show(); // Display the stage
}
}
```



淡入淡出过渡FadeTransition

- 淡入淡出过渡可以让指定结点从一个透明度变化到另一个透明度，它也是Animation的子类。

淡入淡出的圆

— 1/2

```
import javafx.animation.FadeTransition;
import javafx.animation.Timeline;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Ellipse;
import javafx.stage.Stage;
import javafx.util.Duration;

public class FadeTransitionDemo extends Application {

    @Override
    // Override the start method in the Application class
    public void start(Stage primaryStage) {
        Pane pane = new Pane();
        Ellipse ellipse = new Ellipse(10, 10, 100, 50);
        ellipse.setFill(Color.RED);
        ellipse.setStroke(Color.BLACK);
        ellipse.centerXProperty().bind(pane.widthProperty().divide(2));
        ellipse.centerYProperty().bind(pane.heightProperty().divide(2));
        ellipse.radiusXProperty().bind(pane.widthProperty().multiply(0.4));
        ellipse.radiusYProperty().bind(pane.heightProperty().multiply(0.4));
        pane.getChildren().add(ellipse);
    }
}
```

```
// Apply a fade transition to ellipse
```

```
FadeTransition ft = new FadeTransition(Duration.millis(3000), ellipse);
```

```
ft.setFromValue(1.0);
```

```
ft.setToValue(0.1);
```

```
ft.setCycleCount(Timeline.INDEFINITE);
```

```
ft.setAutoReverse(true);
```

```
ft.play(); // Start animation
```

```
// Control animation
```

```
ellipse.setOnMousePressed(e -> ft.pause());
```

```
ellipse.setOnMouseReleased(e -> ft.play());
```

```
// Create a scene and place it in the stage
```

```
Scene scene = new Scene(pane, 200, 150);
```

```
primaryStage.setTitle("FadeTransitionDemo"); // Set the stage title
```

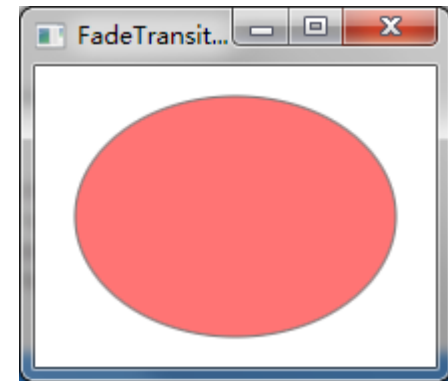
```
primaryStage.setScene(scene); // Place the scene in the stage
```

```
primaryStage.show(); // Display the stage
```

```
}
```

```
}
```

淡入淡出的圆 — 2/2



时间轴Timeline

- 如果需要更好的动画效果，可以利用时间轴Timeline类。这个类和Flash一样，能够让你定义关键帧**KeyFrame**，并指定每一个关键帧之间的时间间隔。Timeline类也是Animation的子类。
- Timeline的构造方法是：
 - Timeline(KeyFrame...keyframes)
- 关键帧的构造方法是：
 - KeyFrame(Duration duration, EventHandler<ActionEvent> onFinished)
- 当每一帧的持续时间到，事件处理器onFinished会被自动触发。所以Timeline可以当定时器用。

闪烁的文本 — 1/2

```
import javafx.animation.Animation;
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.animation.KeyFrame;
import javafx.animation.Timeline;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.layout.StackPane;
import javafx.scene.paint.Color;
import javafx.scene.text.Text;
import javafx.util.Duration;
public class Hello extends Application {
    public void start(Stage primaryStage) {
        StackPane pane = new StackPane();
        Text text = new Text(20, 50, "Programming is fun");
        text.setFill(Color.RED); pane.getChildren().add(text);
        EventHandler<ActionEvent> eventHandler = e ->
            if (text.getText().length() != 0) {
                text.setText("");
            } else {
                text.setText("Programming is fun");
            }
    };
};
```

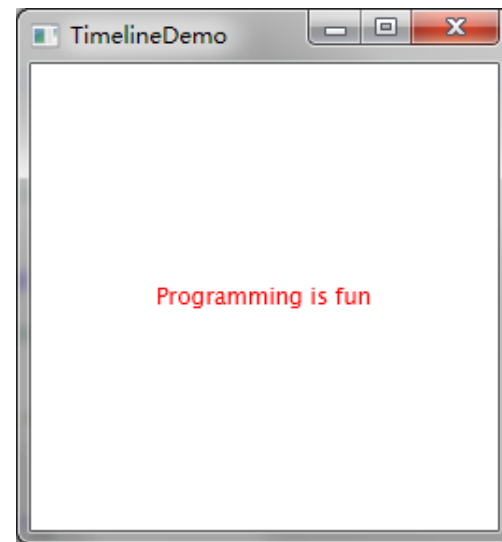
这个事件处理器的功能是切换文本内容。

```
// Create an animation for alternating text
Timeline animation = new Timeline(new KeyFrame(Duration.millis(500),
eventHandler));
animation.setCycleCount(Timeline.INDEFINITE);
animation.play(); // Start animation
// Pause and resume animation
text.setOnMouseClicked(e -> {
    if (animation.getStatus() == Animation.Status.PAUSED) {
        animation.play();
    } else {
        animation.pause();
    }
});
// Create a scene and place it in the stage
Scene scene = new Scene(pane, 250, 250);
primaryStage.setTitle("TimelineDemo"); // Set the stage title
primaryStage.setScene(scene); // Place the scene in the stage
primaryStage.show(); // Display the stage
}
```

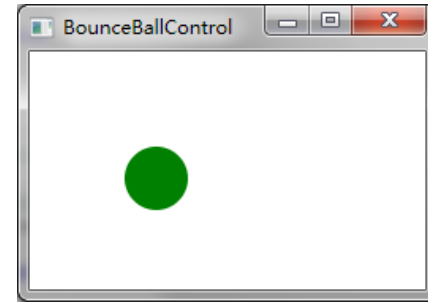
关键帧时间到的事件处理器

这个事件处理器的功能是暂停/继续动画。

闪烁的文本 — 2/2



弹跳小球的例子



- 解题思路：将球的显示和球的控制分开实现
 - 球的显示用Circle，直接将其画在Pane上即可，为此从Pane上继承一个BallPane类；为了方便控制球，这个类需要开放一些方法，如控制球速，暂停/启动。
 - 球的控制直接在测试类的main方法中调用即可。也就是我们希望BallPane已经完整封装了弹跳小球的所有功能。

```
import javafx.animation.KeyFrame;
import javafx.animation.Timeline;
import javafx.beans.property.DoubleProperty;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.util.Duration;
```

```
public class BallPane extends Pane {
    public final double radius = 20;
    private double x = radius, y = radius;
    private double dx = 1, dy = 1;
    private Circle circle = new Circle(x, y, radius);
    private Timeline animation;

    public BallPane() {
        circle.setFill(Color.GREEN); // Set ball color
        getChildren().add(circle); // Place a ball into this pane
        // Create an animation for moving the ball
        animation = new Timeline(new KeyFrame(Duration.millis(50), e -> moveBall()));
        animation.setCycleCount(Timeline.INDEFINITE);
        animation.play(); // Start animation
    }
}
```

BallPane类

— 1/3

利用时间轴动画，每50毫秒触发一次时间到事件。

BallPane类

— 2/3

```
public void play() {  
    animation.play();  
}
```

小球暂停/继续的功能直接通过时间轴动画实现。

```
public void pause() {  
    animation.pause();  
}
```

```
public void increaseSpeed() {  
    animation.setRate(animation.getRate() + 0.1);  
}
```

小球的速度功能直接通过时间轴帧率实现。

```
public void decreaseSpeed() {  
    animation.setRate(animation.getRate() > 0 ? animation.getRate() - 0.1 : 0);  
}
```

```
public DoubleProperty rateProperty() {  
    return animation.rateProperty();  
}
```

BallPane类

—3/3

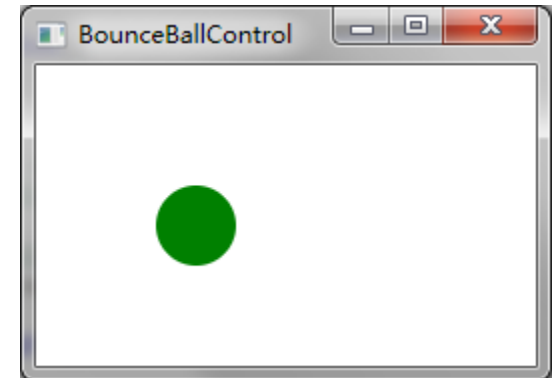
```
protected void moveBall() {  
    // Check boundaries  
    if (x < radius || x > getWidth() - radius) {  
        dx *= -1; // Change ball move direction  
    }  
    if (y < radius || y > getHeight() - radius) {  
        dy *= -1; // Change ball move direction  
    }  
  
    // Adjust ball position  
    x += dx;  
    y += dy;  
    circle.setCenterX(x);  
    circle.setCenterY(y);  
}  
}
```

边界检测，别让小球跑出界了。若抵达边界，将增量取反，强行掉头。其实这里用 $dx = -dx$ 和 $dy = -dy$ 更好， $*$ 运算比较花时间。

重设小球中心，实现小球的移动。

BounceBallControl

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.input.KeyCode;
public class BounceBallControl extends Application {
    public void start(Stage primaryStage) {
        BallPane ballPane = new BallPane(); // Create a ball pane
        ballPane.setOnMousePressed(e -> ballPane.pause());
        ballPane.setOnMouseReleased(e -> ballPane.play());
        ballPane.setOnKeyPressed(e -> {
            if (e.getCode() == KeyCode.UP) {
                ballPane.increaseSpeed();
            } else if (e.getCode() == KeyCode.DOWN) {
                ballPane.decreaseSpeed();
            }
        });
        Scene scene = new Scene(ballPane, 250, 150);
        primaryStage.setTitle("BounceBallControl"); // Set the stage title
        primaryStage.setScene(scene); // Place the scene in the stage
        primaryStage.show(); // Display the stage
        ballPane.requestFocus(); // 让小球取得输入焦点，这样键盘才能控制小球
    }
}
```



THE END