第15章 事件驱动编程和动画

目 录

contents

- 15.1 引言
- 15.2 事件和事件源
- 15.3 事件处理器和处理事件
- 15.4 内部类及匿名内部类
- 15.5 LAMBDA表达式

15.1 引言

GUI应用程序是基于事件驱动的：用户点击界面控件后引发事件，事件由事件处理器（应用程序代码）进行处理。
程序是通过事件的不断产生往前运行。

一个事件处理器 处理 事件源对象 触发的一个事件。
事件处理器是实现了事件处理接口的类的实例，也是对象

15.1 / 引言

JavaFX对动作事件处理器（handler）的要求：

- ✓ handler必须是实现了EventHandler<T extends Event>泛型接口的类的实例。接口定义了所有处理器的共同行为。<T extends Event>表示T是一个Event及其子类型。Event是所有事件对象的祖先类。
  - ✓ 即事件处理器必须实现EventHandler<T extends Event>泛型接口
- ✓ 事件处理器对象handler需要同事件源对象source绑定起来即，一个事件源对象产生的事件应该由哪个事件处理器来处理，因为我们会有多个事件源对象（如Button）和多个事件处理器。
  - ✓ 通过事件源对象source的setOnAction(handler)方法

```
source.setOnAction(handler)
```

15.1 引言

```
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”); //事件源对象
        Button btCancel = new Button("Cancel"); //事件源对象
        //new事件处理器对象handlerl
        OKHandlerClass handlerl = new OKHandlerClass();
        btOK.setOnAction(handlerl);      //注册事件处理器1到btOK
        //new事件处理器对象handler2
        CancelHandlerClass handler2 = new CancelHandlerClass();
        btCancel.setOnAction(handler2); //注册事件处理器2到btCancel
        pane.getChildren().addAll(btOK, btCancel);
        Scene scene = new Scene(pane);
        primaryStage.setTitle("HandleEvent");
        primaryStage.setScene(scene);
        primaryStage.show();
    }}
```

OKHandlerClass和CancelHandlerClass是实现了EventHandler<T extends Event>泛型接口的具体类

15.1 / 引言

```
class OKHandlerClass implements EventHandler<ActionEvent>{
   public void handle(ActionEvent e){ //实现接口方法handle
        System.out.println( “OK button clicked” );
   }
}
class CancelHandlerClass implements EventHandler<ActionEvent> {
     public void handle(ActionEvent e) {
          System.out.println("Cancel button clicked");
     }
}
```

EventHandler<ActionEvent>是泛型接口EventHandler<T extends Event>的具体接口类型，类型实参为ActionEvent

EventHandler<T extends Event>接口里定义了唯一一个接口方法：
void handle(T e)
当类型实参为ActionEvent时，接口方法就变成
void handle(ActionEvent e)
ActionEvent是Event的子类

事件处理器对象必须是实现了EventHandler<T extends Event> 泛型接口的类的实例
因此定义了二个类OKHandlerClass和CancelHandlerClass实现实例接口EventHandler<ActionEvent>（类型实参为ActionEvent）

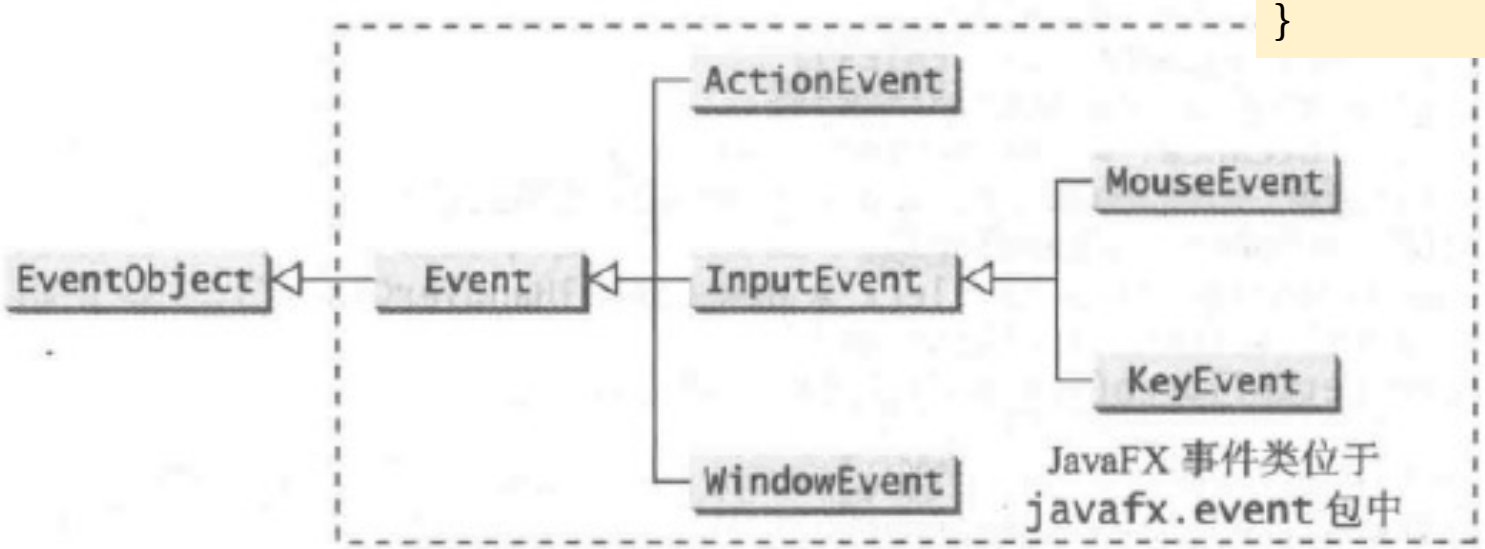
15.2 事件和事件源

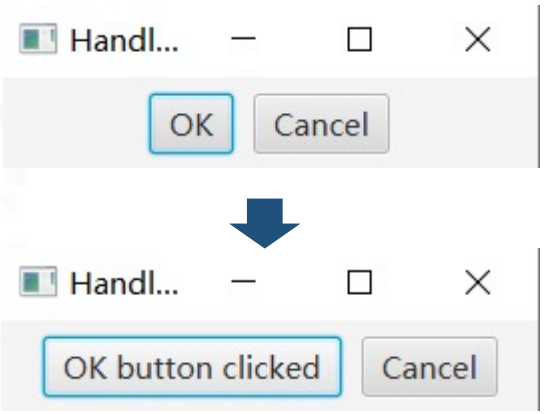
- 事件是从一个事件源上产生的对象（事件本身也是对象）。触发一个事件意味着产生一个事件并委派处理器处理该事件。
- 当运行一个Java GUI 程序的时候，程序和用户进行交互，并且事件驱动它的执行。这被称为事件驱动编程。一个事件可以被定义为一个告知程序某件事发生的信号。事件由外部的用户动作，比如鼠标的移动、单击和键盘按键所触发。程序可以选择响应或者忽略一个事件。
- 产生一个事件并且触发它的组件称为事件源对象。例如，一个按钮是一个按钮单击动作事件的源对象。一个事件是一个事件类的实例。JavaFX 的各种具体事件类都派生自javafx.event.Event，即所有事件都是Event的实例

15.2 事件和事件源

在OKHandlerClass实现接口方法handle时

```
public void handle(ActionEvent e){
        Button btn = (Button)e.getSource();
        btn.setText("OK button clicked");
}
```





- 一个事件对象包含与事件相关的任何属性(比如键盘事件对象包含用户按下了哪个键）。可以通过EventObject 类中的getSource( )实例方法可以得到产生事件的事件源对象的引用。
- EventObject 的子类代表特定类型的事件，比如动作事件、窗口事件、鼠标事件以及键盘事件等。

15.2 事件和事件源

所有ActionEvent事件注册都通过source.setOnAction方法，参数为实现了EvenetHandler<ActionEvent>接口的类的实例

表 15-1 用户动作、源对象、事件类型、处理器接口以及处理器

用户动作	源对象	触发的事件类型	事件注册方法
单击一个按钮	Button	ActionEvent	setOnAction(EventHandler<ActionEvent>)
在一个文本域中回车	TextField	ActionEvent	setOnAction(EventHandler<ActionEvent>)
勾选或者取消勾选	RadioButton	ActionEvent	setOnAction(EventHandler<ActionEvent>)
勾选或者取消勾选	CheckBox	ActionEvent	setOnAction(EventHandler<ActionEvent>)
选择一个新的项	ComboBox	ActionEvent	setOnAction(EventHandler<ActionEvent>)
按下鼠标	Node、Scene	MouseEvent	setOnMousePressed(EventHandler<MouseEvent>)
释放鼠标			setOnMouseReleased(EventHandler<MouseEvent>)
单击鼠标			setOnMouseClicked(EventHandler<MouseEvent>)
鼠标进入			setOnMouseEntered(EventHandler<MouseEvent>)
鼠标退出			setOnMouseExited(EventHandler<MouseEvent>)
鼠标移动			setOnMouseMoved(EventHandler<MouseEvent>)
鼠标拖动			setOnMouseDragged(EventHandler<MouseEvent>)
按下键	Node、Scene	KeyEvent	setOnKeyPressed(EventHandler<KeyEvent>)
释放键			setOnKeyReleased(EventHandler<KeyEvent>)
敲击键			setOnKeyTyped(EventHandler<KeyEvent>)

所有MouseEvent事件的处理器对象都必须是实现了EvenetHandler<MouseEvent>接口的类的实例

MouseEvent具体的类型（单击、释放、移动）通过不同的set方法来区分

KeyEvent事件的处理器对象都必须是实现了EvenetHandler<KeyEvent>接口的类的实例

15.2 事件和事件源

```
public class HandleWindowEvent extends Application {
   public void start(Stage primaryStage) throws Exception {
      HBox pane = new HBox(10);
      Scene scene = new Scene(pane, 300,200);
      primaryStage.setTitle("HandleEvent");
      primaryStage.setOnCloseRequest(new WindowEventHandlerClass());
      primaryStage.setScene(scene);
      primaryStage.show();
   }
}

class WindowEventHandlerClass implements EventHandler<WindowEvent> {
   public void handle(WindowEvent e) {
      new Alert(Alert.AlertType.INFORMATION,"用户试图关闭窗口").showAndWait();
      e.consume();
   }
}
```

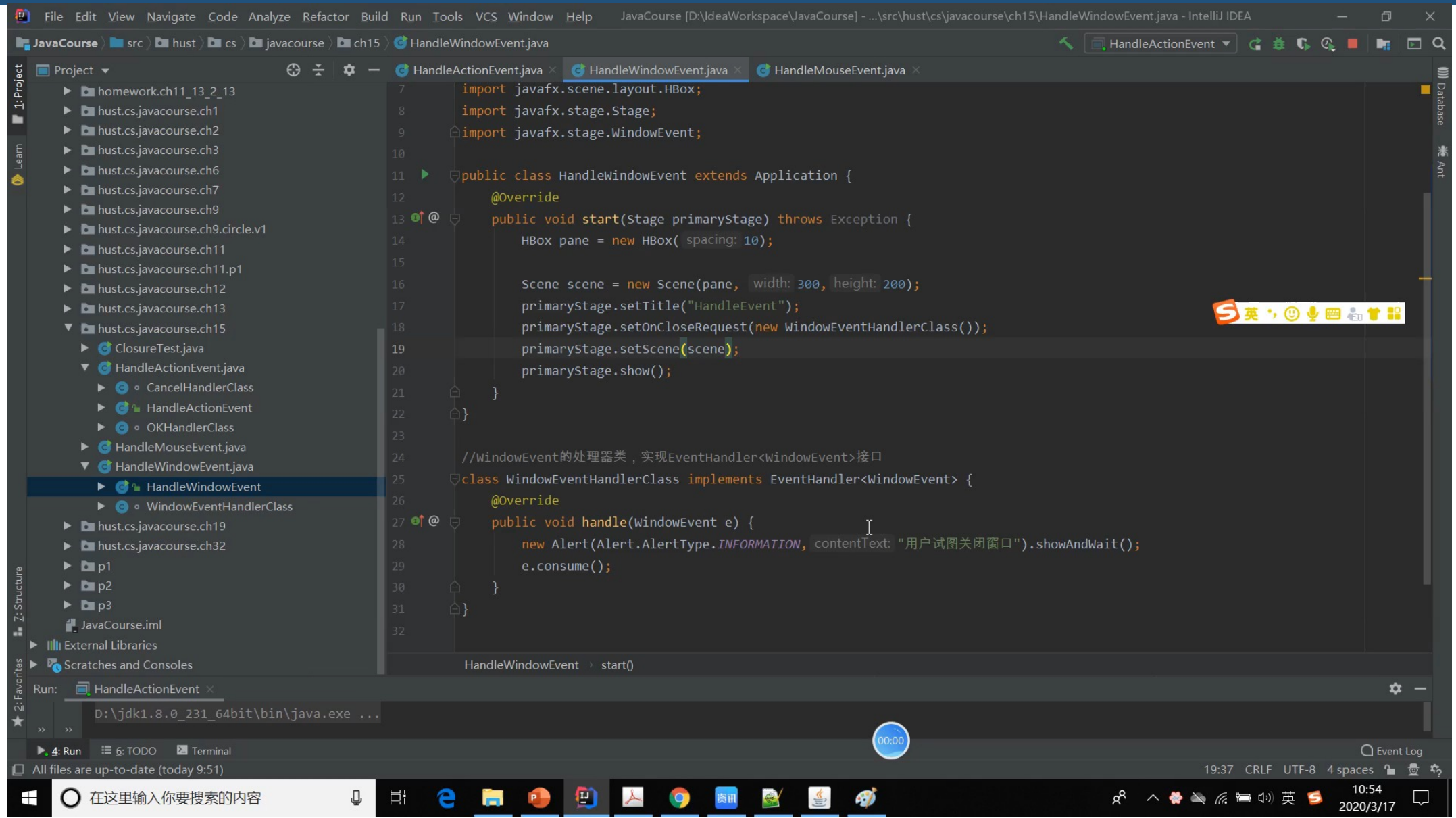
将事件处理器对象绑定到事件源primaryStage，就是程序的主窗口。当用户要关闭主窗口，产生的WindowEvent由下面的handle方法处理。

WindowEvent的处理器类，实现EventHandler<WindowEvent>接口

弹出对话框，提示用户试图关闭窗口

标记这个事件已经处理（消费）了，阻止事件进一步传播，这里就阻止了用户关闭窗口。

15.2 事件和事件源


File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
JavaCourse [D:\IdeaWorkspace\JavaCourse] - ...\src\hust\cs\javacourse\ch15\HandleWindowEvent.java - IntelliJ IDEA
JavaCourse src hust cs javacourse ch15 HandleWindowEvent.java
HandleActionEvent
Project
HandleActionEvent.java
HandleWindowEvent.java
HandleMouseEvent.java
homework.ch11_13_2_13
hust.cs.javacourse.ch1
hust.cs.javacourse.ch2
hust.cs.javacourse.ch3
hust.cs.javacourse.ch6
hust.cs.javacourse.ch7
hust.cs.javacourse.ch9
hust.cs.javacourse.ch9.circle.v1
hust.cs.javacourse.ch11
hust.cs.javacourse.ch11.p1
hust.cs.javacourse.ch12
hust.cs.javacourse.ch13
hust.cs.javacourse.ch15
ClosureTest.java
HandleActionEvent.java
CancelHandlerClass
HandleActionEvent
OKHandlerClass
HandleMouseEvent.java
HandleWindowEvent.java
HandleWindowEvent
WindowEventHandlerClass
hust.cs.javacourse.ch19
hust.cs.javacourse.ch32
p1
p2
p3
JavaCourse.iml
External Libraries
Scratches and Consoles
import javafx.scene.layout.HBox;
import javafx.stage.Stage;
import javafx.stage.WindowEvent;
public class HandleWindowEvent extends Application {
@Override
public void start(Stage primaryStage) throws Exception {
HBox pane = new HBox(spacing: 10);
Scene scene = new Scene(pane, width: 300, height: 200);
primaryStage.setTitle("HandleEvent");
primaryStage.setOnCloseRequest(new WindowEventHandlerClass());
primaryStage.setScene(scene);
primaryStage.show();
}
}
//WindowEvent的处理器类，实现EventHandler<WindowEvent>接口
class WindowEventHandlerClass implements EventHandler<WindowEvent> {
@Override
public void handle(WindowEvent e) {
new Alert(Alert.AlertType.INFORMATION, contentText: "用户试图关闭窗口").showAndWait();
e.consume();
}
}
HandleWindowEvent › start()
Run: HandleActionEvent
D:\jdk1.8.0_231_64bit\bin\java.exe ...
4: Run
6: TODO
Terminal
Event Log
All files are up-to-date (today 9:51)
19:37 CRLF UTF-8 4 spaces
在这里输入你要搜索的内容
10:54
2020/3/17

15.2 事件和事件源

```
public class HandleMouseEvent extends Application {
   public void start(Stage primaryStage) throws Exception {
      Pane pane = new Pane();
      TextField textField = new TextField();
      pane.getChildren().add(textField);
      Scene scene = new Scene(pane, 800,600);
      primaryStage.setTitle("HandleEvent");
      pane.setOnMouseMoved(new MouseEventHandlerClass());
      primaryStage.setScene(scene);
      primaryStage.show();
   }
}
class MouseEventHandlerClass implements EventHandler<MouseEvent> {
   public void handle(MouseEvent e) {
      Pane pane = (Pane)e.getSource();
      TextField t = (TextField)pane.getChildren().get(0);
      t.setText("x = " + e.getSceneX() + ", y = " + e.getSceneY());
   }
}
```

将事件源对象pane绑定的到MouseEvent事件处理器对象。setOnMouseMoved方法指定了事件处理器对象处理MouseMoved事件

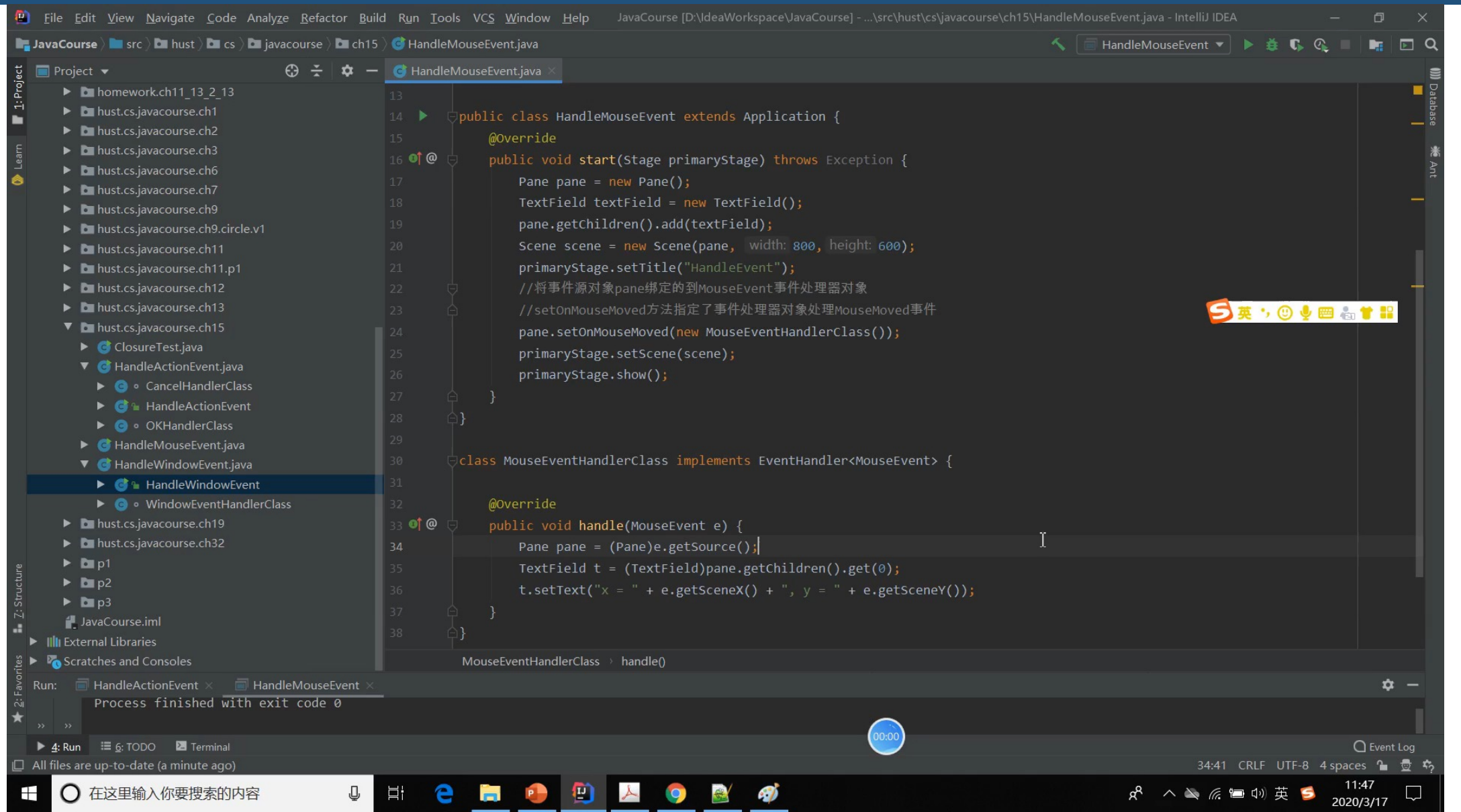
MouseEventHandlerClass类，实现EventHandler< MouseEvent >接口

首先通过getSource得到Pane对象

由于我们想在TextField里显示鼠标当前坐标位置，所以必须通过已经得到的Pane对象，去得到其子对象TextField

e.getScenetX()和e.getSceneY()得到鼠标水平位置坐标和垂直位置坐标（相对于Scene左上角，不是屏幕左上角），将坐标显示在TextField对象里(setText方法)

15.2 事件和事件源

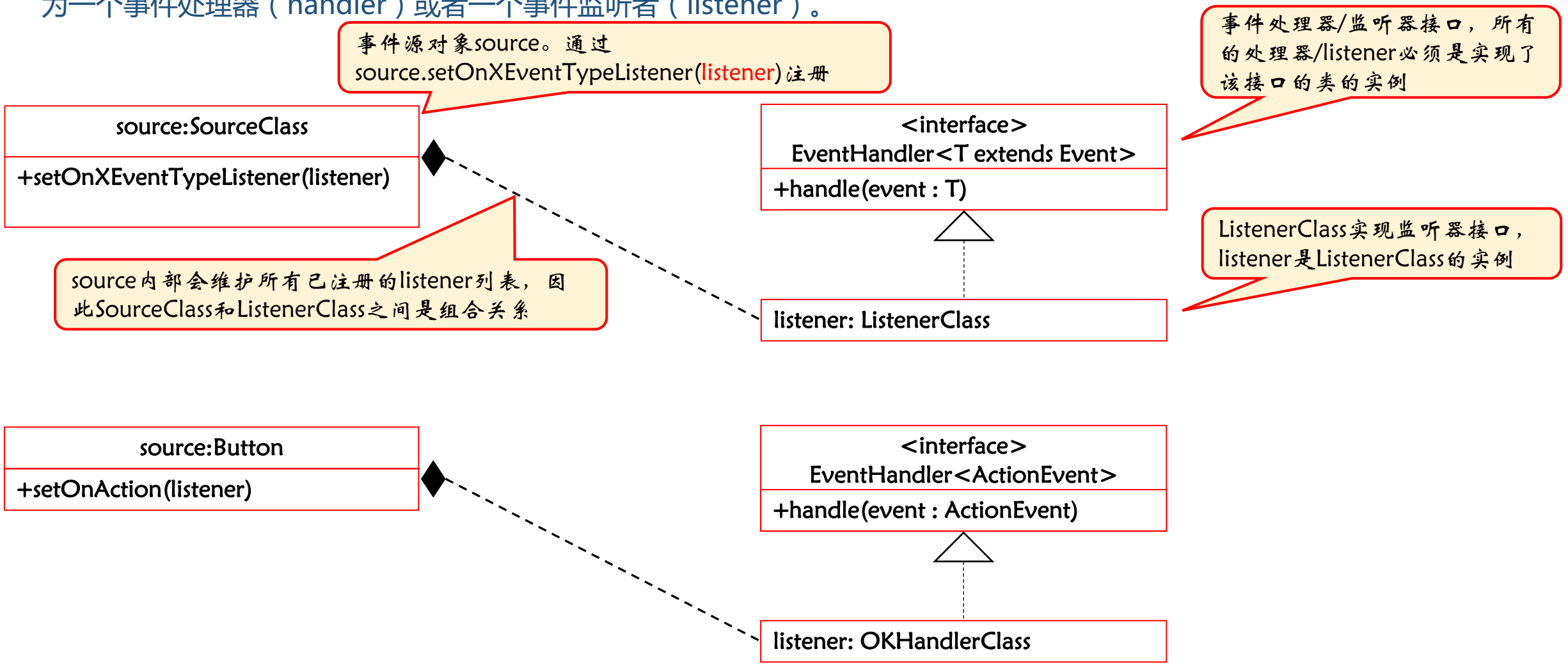

File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
JavaCourse [D:\IdeaWorkspace\JavaCourse] - ...\src\hust\cs\javacourse\ch15\HandleMouseEvent.java - IntelliJ IDEA
JavaCourse src hust cs javacourse ch15 HandleMouseEvent.java
HandleMouseEvent
Project
HandleMouseEvent.java
homework.ch11_13_2_13
hust.cs.javacourse.ch1
hust.cs.javacourse.ch2
hust.cs.javacourse.ch3
hust.cs.javacourse.ch6
hust.cs.javacourse.ch7
hust.cs.javacourse.ch9
hust.cs.javacourse.ch9.circle.v1
hust.cs.javacourse.ch11
hust.cs.javacourse.ch11.p1
hust.cs.javacourse.ch12
hust.cs.javacourse.ch13
hust.cs.javacourse.ch15
ClosureTest.java
HandleActionEvent.java
CancelHandlerClass
HandleActionEvent
OKHandlerClass
HandleMouseEvent.java
HandleWindowEvent.java
HandleWindowEvent
WindowEventHandlerClass
hust.cs.javacourse.ch19
hust.cs.javacourse.ch32
p1
p2
p3
JavaCourse.iml
External Libraries
Scratches and Consoles
13
14 public class HandleMouseEvent extends Application {
15 @Override
16 public void start(Stage primaryStage) throws Exception {
17 Pane pane = new Pane();
18 TextField textField = new TextField();
19 pane.getChildren().add(textField);
20 Scene scene = new Scene(pane, width: 800, height: 600);
21 primaryStage.setTitle("HandleEvent");
22 //将事件源对象pane绑定的到MouseEvent事件处理器对象
23 //setOnMouseMoved方法指定了事件处理器对象处理MouseMoved事件
24 pane.setOnMouseMoved(new MouseEventHandlerClass());
25 primaryStage.setScene(scene);
26 primaryStage.show();
27 }
28 }
29
30 class MouseEventHandlerClass implements EventHandler<MouseEvent> {
31
32 @Override
33 public void handle(MouseEvent e) {
34 Pane pane = (Pane)e.getSource();
35 TextField t = (TextField)pane.getChildren().get(0);
36 t.setText("x = " + e.getSceneX() + ", y = " + e.getSceneY());
37 }
38 }
MouseEventHandlerClass handle()
Run: HandleActionEvent HandleMouseEvent
Process finished with exit code 0
4: Run 6: TODO Terminal
Event Log
All files are up-to-date (a minute ago)
34:41 CRLF UTF-8 4 spaces
在这里输入你要搜索的内容
11:47
2020/3/17

15.3 注册处理器和处理事件

- Java 采用一个**基于委派的模型**来进行事件处理：一个事件源对象触发一个事件，然后一个对该事件感兴趣的对象处理它。后者称为一个事件处理器或者一个事件监听者(实际上就是观察者模式）。
- 一个对象如果要成为一个事件源对象所触发事件的处理器（监听器），那么需要满足两个条件:
  - ✓ 处理器对象（监听器）必须是一个对应的事件处理接口的实例，从而保证该处理器具有处理事件的正确方法。JavaFX 定义了一个对于事件T的统一的处理器泛型接口EventHandler<T extends Event〉。该处理器接口包含handle(T e)方法用于处理事件。
  - ✓ 处理器对象必须通过源对象进行注册。注册方法依赖于事件类型。对ActionEvent 而言，方法是setOnAction。对一个鼠标按下事件来说，方法是setOnMousePressed。对于一个按键事件，方法是set0nKeyPressed。（查PPT第8页的表）

15.3 注册处理器和处理事件

- Java 采用一个**基于委派的模型**来进行事件处理：一个源对象触发一个事件，然后一个对该事件感兴趣的对象处理它。后者称为一个事件处理器（handler）或者一个事件监听者（listener）。



15.4 内部类

- 内部类也称为嵌套类，是在一个类的内部定义的类。通常一个内部类仅被其外部类使用时,同时也不想暴露出去，才定义为内部类。内部类不能定义在方法中。
  - 分为实例内部类和静态内部类
- 实例内部类内部不允许定义静态成员。创建实例内部类的对象时需要使用 外部类的实例变量.new 实例内部类类名( )。(即只有当有了外部类的实例，才能实例化 实例内部类的对象)
- 静态内部类用static定义，其内部允许定义实例成员和静态成员。
- 静态内部类的方法不能访问外部类的实例成员变量。
- 创建静态内部类的对象时需要使用new 外部类.静态内部类( )

15.4 内部类

```
class Wrapper{
   private int x=0;
   private static int z = 0;
   //内部静态类
   static class A{
      int y=0;
      //可以定义静态成员，
      //不能访问外部类的实例成员x，可访问外部类静态成员z
      static int q=0;
      int g() {  return ++q + ++y + ++z; }
   }
   //内部实例类,不能定义静态成员，
   //内部实例类可访问外部类的静态成员如z，实例成员如x
   class B{
      int y=0;
      public int g( ) {
         x++; y++;z++;
         return x+y;
      }
      public int getX(){return x;}
   }
}
```

```
public static void main(String[] args){
      Wrapper w = new Wrapper();  //w.x = 0;
      //创建内部静态类实例
      Wrapper.A a = new Wrapper.A();            //a.y=0, a.q=0;
      Wrapper.A b = new Wrapper.A();            //b.y=0, b.q=0;
      a.g();
      //a,b的实例成员彼此无关，因此执行完a.g()后，a.y = 1, b.y = 0;
      //a,b共享静态成员q，所以a.q=b.q = 1;

      //创建内部实例类实例
      //不能用new Wrapper.B();必须通过外部类对象去实例化内部类对象
      Wrapper.B  c = w.new B(); //类型声明还是外部类.内部类
      c.y=0;
      c.g(); //c.y = 1 ,c.gextX() = 1

      //在外部类体外面，不能通过内部类对象访问外部类成员，只能在内部类里面访问，
      //编译器在这里只能看到内部类成员
//       System.out.println(a.z); //错误
//       System.out.println(c.x); //错误
      //不能通过c直接访问外部类的x，可通过c.gextX()
      System.out.println(c.getX());
}
```

内部类可以被成员访问控制符修饰（私有、缺省、包含、公有的），访问控制规则和类成员访问控制一样

一个内部类被编译成名为OuterClassName$InnerClassName的类

15.4 内部类

- 内部类作用：如果一个类A仅仅被某一个类B使用，且A无需暴露出去，可以把A作为B的内部类实现，内部类也可以避免名字冲突：因为外部类多了一层名字空间的限定。例如类Wrapper1、Wrapper2可以定义同名的内部类A而不会导致冲突

```
public class HandleWindowEvent extends Application {
   public void start(Stage primaryStage) throws Exception {
      HBox pane = new HBox(10);
      Scene scene = new Scene(pane, 300,200);
      primaryStage.setTitle("HandleEvent");
      primaryStage.setOnCloseRequest(new WindowEventHandlerClass());
      primaryStage.setScene(scene);
      primaryStage.show();
   }
   class WindowEventHandlerClass implements EventHandler<WindowEvent> {
         public void handle(WindowEvent e) {
               new Alert(Alert.AlertType.INFORMATION,"用户试图关闭窗口").showAndWait();
               e.consume();
         }
   }
}
```

WindowEventHandlerClass现在作为HandleWindowEvent的内部类来实现。对照PPT11页

15.4 匿名内部类：没有名字的内部类

- 匿名内部类可以简化编程。简化时使用匿名内部类的父类或者接口代替匿名内部类。

```
public class HandleWindowEvent extends Application {
   public void start(Stage primaryStage) throws Exception {
      //其它代码省略
      primaryStage.setOnCloseRequest(
                  new WindowEventHandlerClass ( ) );
   }
   class WindowEventHandlerClass implements
                  EventHandler<WindowEvent> {
      public void handle(WindowEvent e) { //处理语句 }
   }
}
```

把WindowEventHandlerClass的完整声明写出，左边代码相当于

```
public class HandleWindowEvent extends Application {
   public void start(Stage primaryStage) throws Exception {
      //其它代码省略
      primaryStage.setOnCloseRequest(
            new ~~class WindowEventHandlerClass~~
               ~~implements~~ EventHandler<WindowEvent>( ) {
                  public void handle(WindowEvent e) { //处理语句 }
            }
      );
   }
}
```

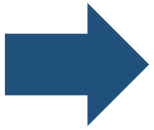
WindowEventHandlerClass这个类名其实不重要，重要的是需要实现EventHandler<WindowEvent>接口，因此想去掉类名，这就是匿名内部类。但new后面必须有一个类型名，就用这个类所实现的接口名作为匿名内部类的类名。

```
public class HandleWindowEvent extends Application {
   public void start(Stage primaryStage) throws Exception {
      //其它代码省略
      primaryStage.setOnCloseRequest(
                  new  EventHandler<WindowEvent>( ) {
                     public void handle(WindowEvent e) { //处理语句 }
                  }
      );
}
```

15.4 匿名内部类：没有名字的内部类

- 匿名内部类可以简化编程。简化时使用匿名内部类的父类或者接口代替匿名内部类。

```
public class HandleWindowEvent extends Application {
   public void start(Stage primaryStage) throws Exception {
      //其它代码省略
      primaryStage.setOnCloseRequest(
                  new WindowEventHandlerClass ( ) );
   }
   class WindowEventHandlerClass implements
                  EventHandler<WindowEvent> {
      public void handle(WindowEvent e) { //处理语句 }
   }
}
```



```
public class HandleWindowEvent extends Application {
   public void start(Stage primaryStage) throws Exception {
      //其它代码省略
      primaryStage.setOnCloseRequest(
            new  EventHandler<WindowEvent>( ) {
                  public void handle(WindowEvent e) { //处理语句 }
            }
      );
}
```

new一个内部匿名类对象时，new 后面直接用这个匿名内部类的父类或者所实现接口作为类型

```
new EventHandler<WindowEvent>( ){
      public void handle(WindowEvent e){   //事件处理语句 }
}
```

实例化一个实现了EventHandler<WindowEvent>接口的匿名内部类对象，作为setOnCloseRequest方法的实参

15.4 匿名内部类

- 匿名内部类可以简化编程。简化时使用匿名内部类的父类或者所实现接口代替匿名内部类名字，作为new后面的类型。

匿名内部类的语法如下所示：

```
new SuperClassName/InterfaceName() {
  // Implement or override methods in superclass or interface

  // Other methods if necessary
}
```

- 匿名内部类总是使用父类的无参构造方法产生实例，对于接口使用Object（ ）。
- 匿名内部类必须实现父类或者接口的所有抽象方法。事件处理接口通常只有1个方法。
- 一个匿名内部类被编译成OuterClassName$n.class,如Test$1.class, Test$2.class
- 见教材第15章程序清单15-4

15.5 Lambda表达式

- Lambda表达式可以进一步简化事件处理的程序编写
- 编译器会将lamda表达式看待为匿名内部类对象，将这个对象理解为实现了EventHandler<ActionEvent>接口的实例。下面例子中因为EventHandler接口定义了参数为ActionEvent类型的方法handler，因此编译器可以推断参数e的类型为ActionEvent，并且e->{ }中右边的{ }就是handel方法方法体。
- EventHandler接口只有一个方法，只有一个方法的接口称为功能接口（函数式接口），每个 Lambda 表达式都能隐式地赋值给函数式接口，lamda表达式中的{ }就是函数式接口中接口方法的方法体。

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

a）匿名内部类事件处理器

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

e->{//事件处理代码}是一个lambda表达式，相当于匿名内部类的实例，传给setOnAction方法，绑定到事件源对象btEnlarge

b）lambda 表达式事件处理器

15.5 Lambda表达式

- Lambda表达式本质上更像匿名函数。
- Java里规定Lambda表达式只能赋值给函数式接口。
- Lambda表达式的语法为：

  （type1 para1, ..., typen paran)->expression 或者

  （type1 para1, ..., typen paran)->{ 一条或多条语句}

```
(int a, int b) -> { return a + b; }
() -> System.out.println("Hello World")
(String s) -> { System.out.println(s); }
() -> 42
() -> { return 3.1415 ;}
```

- 当把Lambda表达式赋值给函数式接口时， Lambda表达式的参数的类型是可以推断的；如果只有一个参数，则可以省略圆括弧。从而使Lambda表达式简化为：

  e->处理e的expression 或者

  e->{ 处理e的statements; }

15.5 Lambda表达式

- Lambda 表达式的结构
  - 一个 Lambda 表达式可以有零个或多个参数
  - 参数的类型既可以明确声明，也可以根据上下文来推断。例如：(int a)与(a)效果相同（当可以推断类型时）
  - 所有参数需包含在圆括号内，参数之间用逗号相隔。例如：(a, b) 或 (int a, int b) 或 (String a, int b, float c)
  - 空圆括号代表参数集为空。例如：() -> 42
  - 当只有一个参数，且其类型可推导时，圆括号（ ）可省略。例如：a -> {return a*a；}
  - Lambda 表达式的主体可以是表达式或者是block，如果是表达式，不能有{}；如果是block，则必须加 { }

15.5 Lambda表达式

- 每个 Lambda 表达式都能隐式地赋值给函数式接口
  - Runnable接口就是函数式接口，里面定义接口方法void run( )，我们可以通过 Lambda 表达式创建一个接口实例 。`Runnable r = ( ) -> System.out.println("hello world");`
    - 上面语句的含义是：将一个实现了Runnable接口的类的实例赋值给Runnable接口引用r， Lambda 表达式的主体就是接口方法void run( )的具体实现
  - 当不是显式赋值给函数式接口时，编译器会自动解释这种转化：

```
new Thread(
        () -> System.out.println("hello world")
).start();
```

  - 在上面的代码中，编译器会自动推断：根据线程类的构造函数签名 public Thread(Runnable r) { }，将该 Lambda 表达式赋给 Runnable 接口。

15.5 Lambda表达式

- 函数式接口定义好后，我们可以在 API 中使用它，同时利用 Lambda 表达式。

```
//定义一个函数式接口
public interface WorkerInterface {
        public void doSomeWork();
}

public class WorkerInterfaceTest {
        public static void exec(WorkerInterface worker) {
                worker.doSomeWork();
        }

        public static void main(String [] args) {
                //invoke doSomeWork using Annonymous class
                exec( new WorkerInterface() {
                        @Override public void doSomeWork() {
                        System.out.println("Worker invoked using Anonymous class"); }
                } );
                //invoke doSomeWork using Lambda expression
                exec( () -> System.out.println("Worker invoked using Lambda expression") );
        }
}
```

new一个实现了WorkerInterface接口的匿名类对象传入exec方法

将一个Lambda表达式传入exec方法

15.5 Lambda表达式

```
public class LamdaHandlerDemo extends Application{
        @Override
        public void start(Stage primaryStage) throws Exception {
                HBox hBox = new HBox();
                hBox.setSpacing(10.0);
                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( (ActionEvent e)->{ System.out.println("Process New"); } );
                btOPen.setOnAction( (e)->{System.out.println("Process Open");} ) ;
                btSave.setOnAction( e->{System.out.println("Process Save");} ); //->右边是Statements
                btPrint.setOnAction( e->System.out.println("Process Print") ); //->右边是expression

                Scene secen = new Scene(hBox,300,50);
                primaryStage.setTitle("LamdaHandlerDemo");
                primaryStage.setScene(secen);
                primaryStage.show();
        }
}
```

创建4个Button对象，事件源对象

4种方式注册事件处理器

15.5 Lambda表达式

- Lambda 神奇功能
  - 计算给定数组中每个元素平方后的总和。请注意，Lambda 表达式只用一条语句就能达到此功能，这也是 MapReduce 的一个初级例子。我们使用 map() 给每个元素求平方，再使用 reduce() 将所有元素计入一个数值：
  - java.util.stream.Stream 接口包含许多有用的方法，能结合 Lambda 表达式产生神奇的效果。

```
//Old way:
List<Integer> list = Arrays.asList(1,2,3,4,5,6,7);
int sum = 0;
for(Integer n : list) {
            int x = n * n;
            sum = sum + x;
 }
System.out.println(sum);

//New way:
List<Integer> list = Arrays.asList(1,2,3,4,5,6,7);
 int sum = list.stream().map(x -> x*x).reduce((x,y) -> x + y).get();
System.out.println(sum);
```

15.5 Lambda表达式

- Java Lambda 表达式的由来
  - Java 中的一切都是对象（除了基本数据类型），即使数组也是一种对象，每个类创建的实例也是对象。在 Java 中定义的函数或方法不可能完全独立，也不能将方法作为参数或返回一个方法给实例。为此，Java 8 增加了一个语言级的新特性，名为 Lambda 表达式。
  - 在函数式编程语言中，函数是一等公民，它们可以独立存在，你可以将其赋值给一个变量，或将他们当做参数传给其他函数。JavaScript 是最典型的函数式编程语言（当然也是面向对象的）。函数式语言提供了一种强大的功能——闭包。当一种编程语言支持函数返回类型为函数时，这种语言天然就支持闭包。
  - Java虽然不支持函数返回类型为函数，但可以用匿名内部类实现闭包，但这种闭包多了一个限制：要求捕获的的自由变量必须是final的。用Lambda表达式同样如此： Lambda表达式捕获的自由变量必须是final的
  - 为什么Java里的闭包多了这个限制：在Java的经典著作《Effective Java》、《Java Concurrency in Practice》大神们这么解释：如果Java闭包捕获的自由变量是非final的，会导致线程安全问题。 Python和Javascript则不用考虑这样的问题，所以它的闭包捕获的自由变量是可以任意修改的。

15.5 Lambda表达式

- 闭包(Closure)
  - 闭包(Closure)并不是一个新鲜的概念，很多函数式语言中都使用了闭包。例如在JavaScript中，当你在内嵌函数中使用外部函数作用域内的变量时，就是使用了闭包。用一个常用的类比来解释闭包和类（Class）的关系：类是带函数的数据，闭包是带数据的函数。

  - 闭包的本质：代码块+上下文

15.5 Lambda表达式

- 闭包(Closure)
  - 闭包的本质：代码块+上下文

```
function generator() {
        var i = 0; //被闭包捕获的自由变量
        return function() {//返回一个嵌套匿名函数，使用了外部函数作用域里变量i
                return i++;
        };
}
var gen1 = generator();     //得到一个自然数生成器，        gen1是一个闭包
var gen2 = generator();     //得到另一个自然数生成器，      gen2是一个闭包
var r1 = gen1();            //r1 = 0
var r2 = gen1();            //r2 = 1
var r3 = gen2();            //r3 = 0
var r4 = gen2();            //r4 = 1
console.log(r4);
```

15.5 Lambda表达式

- 闭包(Closure)
  - 闭包中捕获的自由变量有一个特性，就是它们不在父函数返回时释放，而是随着闭包生命周期的结束而结束。gen1和gen2分别使用了相互独立的变量i（在gen1的i自增1的时候，gen2的i并不受影响，反之亦然），只要gen1或gen2这两个变量没有被JavaScript引擎垃圾回收，他们各自的变量i就不会被释放。

```
function generator() {
        var i = 0; //被闭包捕获的自由变量
        return function() {//返回一个嵌套匿名函数，使用了外部函数作用域里变量i
                return i++;
        };
}
var gen1 = generator();     //得到一个自然数生成器，        gen1是一个闭包
var gen2 = generator();     //得到另一个自然数生成器，      gen2是一个闭包
var r1 = gen1();            //r1 = 0
var r2 = gen1();            //r2 = 1
var r3 = gen2();            //r3 = 0
var r4 = gen2();            //r4 = 1
console.log(r4);
```

闭包被创造出来显然是因为有场景需要的。一个最为普遍和典型的使用场合是：延迟执行。我们可以把一段代码封装到闭包里，你可以等到“时机”成熟时去执行它。

15.5 Lambda表达式

```
interface Closure<T>{
   T get();
}

class Dog{

}
```

```
Closure<Dog> c1 = testClosure1();  //返回闭包1
Closure<Dog> c2 = testClosure1();  //返回闭包2

//二个闭包里面还有dog，而且是不同的dog，这个时候testClosure1方法已经结束了
//二个闭包都捕获了局部变量dog，延长了dog的生命周期
System.out.println(c1.get() == c2.get()); //false
```

```
public static Closure<Dog> testClosure1(){
    //匿名内部类需要访问匿名内部类所在方法中的局部变量的时候，
    //必须给局部变量加final进行修饰
     final Dog  dog = new Dog();
     //Java里，匿名内部类方法要捕获的外部闭包环境的自由变量必须是final的
     return new Closure<Dog>() {
        @Override
        public Dog get() {
           //匿名对象的get方法捕获了外面的自由变量dog，
           //使得testClosure1中的局部变量dog生命周期延长
           return dog;
        }
     };
}
```

15.5 Lambda表达式

```
interface Closure<T>{
   T get();
}

class Dog{

}
```

```
Closure<Integer> c3 = testClosure2();  //返回闭包1
Closure<Integer> c4 = testClosure2();  //返回闭包2

//二个闭包里面还有Integer，而且是不同的Integer
 System.out.println(c3.get() == c4.get());
```

```
public static Closure<Integer> testClosure2(){
    //从JDK1.8开始8加了一个语法糖：在lambda表达式以及匿名类内部，如果捕获某局部变量，则直接将其视为final。
    int i = 0; //不用final修饰，但是一旦在lambda表达式里修改i，立刻编译报错，换句话说，捕获的自由变量还是不可改
    return () -> { return i;}; //注意装箱操作，返回的是Integer对象。
}
```