Event Handling and Exception Handling
Introduction

GUI-oriented programs are event-driven. In this module you will learn how to write Java programs that can react to user interface events, such as button pushes and mouse clicks. The Java window toolkit (awt) has a very sophisticated mechanism that allows a program to specify the events in which it is interested and which objects to notify when one of these events occurs.

Programs can fail for a variety of reasons: bad input and programmer errors are just two of many possible causes. A program should deal with failure in a predictable manner. In Java, exception handling provides a flexible mechanism for passing control from the point of error detection to a competent recovery handler. This module also discusses the exception handling mechanism and shows you how to use it appropriately in your programs.

Objectives

- To understand the Java event model
- To install action event listeners
- To accept input from buttons, text fields
- To familiarize with JButton, JLabel, JPanel, JTextField and JFrame
- To learn how to throw exceptions
- To be able to design your own exception classes
- To understand the difference between checked and unchecked exceptions
- To learn how to catch exceptions
- To learn when and where to catch an exception

Events, Event Sources, and Event Listeners

User interface events include key presses, mouse moves, button clicks, menu selections and so on. Every program must indicate which events it needs to receive. It does that by installing event listener objects. An event listener object belongs to a class that is provided by the application programmers. Its methods describe the actions to be taken when an event occurs. To install a listener, you need to know the event source. The event source is the user interface component that generates a particular event. You add an event listener object to the appropriate event source.

The general structure for defining an event driven application is:

1. Define interface components (event sources) to be used for generating events, such as buttons, text fields, check box etc.
2. Add Event Listeners to those event sources.
3. Define actions performed once the events occur (by implementing an actionPerformed method).

If no event listener object is added to an event source, the event source is inactive; If no action is defined, nothing will happen when the event occurs.

Please evaluate file ClickListener.java and ButtonTester.java programs on page 445 and 446 of the text.
Building Applications with Buttons

Java’s javax.swing package defines various GUI components classes and their methods. In this section, you are required to familiarize with JButton, JLabel and JPanel classes and the process to build GUI applications. Buttons are the most commonly used components in a Graphic User Interface (GUI).

Use a JPanel container to group multiple user interface components together. For some complex GUI’s, you may overlap panels to achieve a certain layout. Finally, all components have to be put into a main panel; the main panel then has to be put into a frame to form a GUI.

For example:

```java
JButton button = new JButton (“Click Me“); //to create a JButton object
JLabel label = new JLabel (“This is ”+ n + “ clicks!”); //to create a JLabel object
JPanel panel = new JPanel(); //to create a JPanel object
JFrame frame = new JFrame(); //to create a JFrame object
panel.add(button);
panel.add(label);
frame.add(panel);
```

Once the components are defined and settled in the frame, the next step is to add an event listener that to handle events (button click for a JButton component). You can define another class to implement the ActionListener interface like the example in the text book. You place the button action into the actionPerformed method (see class AddInterestListener on page 447). You can also install event listeners as inner classes to be able to use fields, methods and final variables. The example, InvestmentViewer1.java, is on page 448.

You may find the following program is easier to understand. You can evaluate it to see how it works.
import java.awt.event.*;  
import javax.swing.*;  
public class TestButton  
{  
    private static int n = 0;  
    public static void main(String[] args)  
    {  
        JFrame frame = new JFrame(); //to create a JFrame object  
        JButton button = new JButton("Click Me"); //to create a JButton object  
        final JLabel label = new JLabel("This is "+ n +" Click!"); //to create a JLabel object  
        JPanel panel = new JPanel(); //to create a JPanel object  
        panel.add(button);  
        panel.add(label);  
        frame.add(panel);  
        class AddListener implements ActionListener  //inner class to implement ActionListener  
        {  
            public void actionPerformed(ActionEvent e)  
            {  
                n++;  
                label.setText("This is "+n+" Click!");  
            }  
        }  
        ActionListener listener = new AddListener(); //create an object of AddListener  
        button.addActionListener(listener); //add Listener to the button  
        frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);  
        frame.setVisible(true);  
    }  
}  
private static final int FRAME_WIDTH=400;  
private static final int FRAME_HEIGHT=100;  
}  

Processing Text Input

Except for the showInputDialog method of the JOptionPane class, most GUI’s collect text input through text fields. The JTextField class (in javax.swing) provides a text field. For example:

```java
final int FIELD_WIDTH = 10;
final JTextField rateField = new JTextField(FIELD_WIDTH);
```

To each text field, you generally need to provide a label so that the user knows what to type into it. Also, you should supply a button that the user can press to indicate that the input is ready for processing.

When the button is clicked, its actionPerformed method reads the user input from the text fields, using the getText method of the JTextField class. Note the getText method returns a String object.

An example is given below, pay particular attention to the actionPerformed method.
import java.awt.event.*;
import javax.swing.*;

public class TestTextField
{
    public static void main(String[] args)
    {
        JFrame frame = new JFrame(); //to create a JFrame object
        JButton button = new JButton("Submit"); //to create a JButton object
        final JLabel label = new JLabel("Hello"); //to create a JLabel object
        JPanel panel = new JPanel(); //to create a JPanel object
        final JTextField inputName = new JTextField(10); //to create a JTextField object
        JLabel label1 = new JLabel("Input name:");
        panel.add(label1);
        panel.add(inputName);
        panel.add(button);
        panel.add(label);
        frame.add(panel);
        class AddListener implements ActionListener //inner class to implement ActionListener
        {
            public void actionPerformed(ActionEvent e)
            {
                label.setText("Hello " + inputName.getText());
            }
        }
        ActionListener listener = new AddListener(); //create an object of AddListener
        button.addActionListener(listener); //add ActionListener to the button
        frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
        frame.setVisible(true);
    }
    private static final int FRAME_WIDTH=400;
    private static final int FRAME_HEIGHT=100;
}

Please read and understand the program InvestmentViewer2.java on page 452, 453.

**Mouse Events**

Self study. There is no requirement to do this topic for this course.
Throwing Exceptions

An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions. What should your program do when it detects an error condition? You need to signal the exceptional condition and then provide code to handle it. One mechanism, in Java, is very easy: you just throw an appropriate exception object by using the throw statement. When you throw an exception, execution does not continue with the next statement but in an exception handler.

The Java platform provides numerous exception classes. All these classes are descendants of the Throwable (refer to Figure 1 on page 554) class and all allow programs to differentiate among the various types of exceptions that can occur during the execution of a program. You also can create your own exception classes to represent problems that can occur within the classes that you write. A later section in this chapter explains how you can create your own exception classes. For now, all you need to remember is that you can throw only objects that inherit from the java.lang.Throwable class.

```java
public class BankAccount {
    public void withdraw (double amount)
    {
        if (amount > balance)
        {
            throw (new IllegalArgumentException("Amount exceeds balance!"));
        }
        balance = balance – amount;
    }
    ...
}
```

A method example to throw a Throwable exceptional object

Checked and Unchecked Exceptions

There are two kinds of exceptions: Checked and Unchecked.

- **Checked exceptions:**

  All subclasses of the class Exception except for subclasses of RuntimeException are checked exceptions. When you call a method that throws a checked exception, the compiler checks that you don’t ignore it. You must tell the compiler what you are going to do about the exception if it is ever thrown. If you take a look at the Figure 1 on page 554, you will see the majority of checked exceptions occur when you deal with input and output. This is a fertile ground for external failures beyond your control. Therefore, you will need to deal with checked exceptions principally when programming with files and streams.

  You have two options to handle a checked exception:

  - To handle the exception by catching exceptions (in the next topic)
  - To tell the compiler that you are aware of this exception and that you want your method to be able to be terminated when it occurs. In this case, you need to add a throws
specifier to a method that can throw a checked exception. The method can throw different types.

For example:

```java
generate void read (String filename) throws FileNotFoundException
{  
    FileReader reader = new FileReader(filename)  
    Scanner in = new Scanner(reader);  
    ...  
    throw new FileNotFoundException("file not found!"); // FileNotFoundException is checked  
    ...  
}
generate```

Please note: It is “throws” in the method declaration, while it is a “throw” (no “s”) in the body of method.

- **Unchecked exceptions:**

  All subclasses of the class RuntimeException are unchecked exceptions. The compiler doesn’t require you to keep track of unchecked exceptions. You can throw this kind of exception directly in your method without informing complier about anything.

### Catching Exceptions

In a method that is ready to handle a particular exception type, place the statements that can cause the exception inside a try block, and the handler inside a catch clause.

For example, consider the following pseudocode method that reads an entire file into memory:

```java
readFile  
{  
    open the file;  
    determine its size;  
    allocate that much memory;  
    read the file into memory;  
    close the file;  
}
```

Exceptions enable you to write the main flow of your code and to deal with the exceptional cases elsewhere. If the `readFile` function used exceptions techniques, it would look more like this:

```java
readFile  
{  
    try {  
    open the file;  
    determine its size;  
    allocate that much memory;  
    read the file into memory;  
    close the file;  
} catch (fileOpenFailed) {  
    doSomething;  
} catch (sizeDeterminationFailed) {  
    doSomething;  
```
The finally Clause

Sometimes, you need to take some action whether or not an exception is thrown. The finally construct is used to handle this situation. Once a try block is entered, the statement in a finally clause are guaranteed to be executed, whether or not an exception is thrown.

The finally block is optional and provides a mechanism to clean up regardless of what happens within the try block. Use the finally block to close files or to release other system resources.

Designing Your Own Exception Types

When faced with choosing the type of exception to throw, you can either use one written by someone else—the Java platform provides a lot of exception classes that you can use—or you can write one of your own. You should write your own exception classes if you answer yes to any of the following questions. Otherwise, you can probably use someone else’s.

- Do you need an exception type that isn’t represented by those in the Java platform?
- Would it help your users if they could differentiate your exceptions from those thrown by classes written by other vendors?
- Does your code throw more than one related exception?
- If you use someone else’s exceptions, will your users have access to those exceptions? A similar question is: should your package be independent and self-contained?

Your own exception classes must extend to Exception or RuntimeException class.

Reading
Text book:

Chapter 12: Event Handling (Chapter sections required: 12.1, 12.2, 12.3)

Chapter 15: Exception Handling (whole chapter required)

Review questions

Review exercises:

Page 462 - 463: Exercise R12.1, R12.2, R12.9, R12.10
Page 571 - 572: Exercise R15.1, R15.2, R15.5, R15.8

Programming exercises:

Page 463: Exercise P12.3, P12.4
Page 572: Exercise P15.1

References

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

/**
 * An action listener that prints a message.
 */
public class ClickListener implements ActionListener
{
    public ClickListener(String theName)
    {
        name = theName;
    }

    public void actionPerformed(ActionEvent event)
    {
        System.out.println("Button " + name + " was clicked!");
    }

    private String name;
}

import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;

/**
 * This program demonstrates how to install an action listener.
 */
public class ExP12_3
{
    public static void main(String[] args)
    {
        JFrame frame = new JFrame();
        JPanel panel = new JPanel();

        JButton buttonA = new JButton("A");
        panel.add(buttonA);

        JButton buttonB = new JButton("B");
        panel.add(buttonB);

        frame.add(panel);

        ActionListener listenerA = new ClickListener("A");
        buttonA.addActionListener(listenerA);

        ActionListener listenerB = new ClickListener("B");
        buttonB.addActionListener(listenerB);

        frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

/**
 * An action listener that prints a message.
 */
public class ClickListener implements ActionListener
{
    public void actionPerformed(ActionEvent event)
    {
        System.out.println("Button " + event.getActionCommand() + " was clicked!");
    }
}

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;

/**
 * This program demonstrates how to install an action listener.
 */
public class ExP12_4
{
    public static void main(String[] args)
    {
        JFrame frame = new JFrame();
        JPanel panel = new JPanel();

        JButton buttonA = new JButton("A");
        panel.add(buttonA);

        JButton buttonB = new JButton("B");
        panel.add(buttonB);

        frame.add(panel);

        ActionListener listener = new ClickListener();
        buttonA.addActionListener(listener);
        buttonB.addActionListener(listener);

        frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        frame.setVisible(true);
    }
}

private static final int FRAME_WIDTH = 100;
private static final int FRAME_HEIGHT = 70;
P15-1
/**
 * A bank account has a balance that can be changed by
 * deposits and withdrawals.
 */
public class BankAccount
{
    /**
     * Constructs a bank account with a zero balance.
     */
    public BankAccount()
    {
        balance = 0;
    }

    /**
     * Constructs a bank account with a given balance.
     * @param initialBalance the initial balance
     */
    public BankAccount(double initialBalance)
    {
        if (initialBalance < 0)
            throw new IllegalArgumentException("Cannot enter negative balance");
        balance = initialBalance;
    }

    /**
     * Deposits money into the bank account.
     * @param amount the amount to deposit
     */
    public void deposit(double amount)
    {
        if (amount < 0)
            throw new IllegalArgumentException("Cannot deposit negative amount");
        double newBalance = balance + amount;
        balance = newBalance;
    }

    /**
     * Withdraws money from the bank account.
     * @param amount the amount to withdraw
     */
    public void withdraw(double amount)
    {
        if (amount < 0 || amount > balance)
            throw new IllegalArgumentException("Cannot withdraw this amount");
        double newBalance = balance - amount;
        balance = newBalance;
    }
}
```java
balance = newBalance;
}

/**
   * Gets the current balance of the bank account.
   * @return the current balance
   */
public double getBalance()
{
    return balance;
}

private double balance;

/**
   * A class to test the BankAccount class.
   */
public class ExP15_1
{
    public static void main(String[] args)
    {
        BankAccount harrysChecking = new BankAccount();

        try
        {
            harrysChecking = new BankAccount(-500);
        }
        catch (IllegalArgumentException e)
        {
            e.printStackTrace();
        }

        try
        {
            harrysChecking.deposit(-100);
        }
        catch (IllegalArgumentException e)
        {
            e.printStackTrace();
        }

        try
        {
            harrysChecking.withdraw(100);
        }
        catch (IllegalArgumentException e)
        {
            e.printStackTrace();
        }
    }
}