Swing 2

Window Interfaces Using Swing

Continued

# Introduction

First we will refresh our memory about exception handling in Java. Next we will change the size of our frame to fit in the screen. Then we study Checkboxes and I/O dialog boxes. Finally we go over Radio buttons.

## 1. An overview of exception handling

### Basic exception handling

In a Java program either Java handles the exceptions or the programmer provides a mechanism to handle the exceptions. This is called **throwing exceptions.**

### Exception in Java

We write a simple program to show how the programmer handles an exception, and how one of the features of Java named with **try-catch** block handles exceptions.

### Example 1:

A program to prompt the user for a numerator m, and denominator n. If in m/n division by zero occurs the program terminates with an error message. This error message is the result of the code of the programmer and it is not from an exception:

1. Handling exception by the programmer:

**import** java.util.Scanner;

**public** **class** Main{

**public** **static** **void** main(String[] args){

Scanner key = **new** Scanner(System.*in*);

System.*out*.print("Enter two numbers for numerator(m) and denominator(n): ");

**int** m = key.nextInt();

**int** n = key.nextInt();

**if**(n == 0)

System.*out*.println("Division by zero is not correct.");

**else**

System.*out*.println(m + "/" + n + " = " + (m/n));

System.*out*.println("End of the program.");

}

}

#### A sample dialog:

Enter two numbers for numerator and denominator: 7 0

Division by zero is not correct.

End of the program.

#### Description:

When the denominator is 0 the if-part is executed and we have the above output. This is not a bad solution but sometimes it is hard to catch the error. For example the program is expecting an integer for its input but the user enters a string.

#### Handling exception by Java

We can let the java interpreter catch the exception and terminate the execution of the program.

### Example 2:

In the following program when the control of execution reaches to division and executes it, the Java interpreter throws an exception:

**import** java.util.Scanner;

**public** **class** Main{

**public** **static** **void** main(String[] args){

Scanner key = **new** Scanner(System.*in*);

System.*out*.print("Enter two numbers for numerator and denominator: ");

**int** m = key.nextInt();

**int** n = key.nextInt();

System.*out*.println(m + "/" + n + " = " + (m/n));

System.*out*.println("End of the program.");

}

}

#### A sample dialog:

Enter two numbers for numerator and denominator: 7 0

Exception in thread "main" java.lang.ArithmeticException: / by zero at Main.main(Main.java:9)

#### Description:

When division by zero occurs Java throws an exception message and ***stops***.

Note that the control of execution stopped when the Java interpreter throws the exception. The control of execution did not execute the last System.out. The following approach does not have this problem.

#### Using Try-catch block

We can use this feature of Java to handle exception the way we like to. By using this feature the problem of termination of the program when an exception occurs (above paragraph) goes away. The general format of a try-catch block is as follows:

**try** {

Code\_ To\_ Try

More\_ Code

} **catch**( ExceptionClassName Parameter) {

Process\_Exception

}

Possibly\_ Other\_ Catch\_ Blocks

The try-catch block works as follows:

1. We write part of the code in the try-block that may causes an exception and possibly more code.

2. We write our desired code in an appropriate catch-block such that when an exception occurs this desired code gets executed.

At execution time when an exception occurs the control of execution immediately jumps to the corresponding catch block. Any instruction after the code that causes an exception in the try-block won’t be executed.

If no error occurs in the try-block the control of execution jumps over all the catch-blocks. In other words, in this situation none of the instructions in the catch-blocks get executed. Any code after the catch- blocks gets executed provided we do not have a System.exit(...) in the path of execution.

### Example 3:

The following program is the same division-by-zero program with an application of a try-catch block:

**import** java.util.Scanner;

**public** **class** Main{

**public** **static** **void** main(String[] args){

Scanner key = **new** Scanner(System.*in*);

System.*out*.print("Enter two numbers for numerator and denominator: ");

**int** m = key.nextInt();

**int** n = key.nextInt();

**try**{

System.*out*.println(m + "/" + n + " = " + (m/n));

System.*out*.println("The division is done");

}**catch**(Exception e){

System.*out*.println("Division by zero is not correct");

}

System.*out*.println("End of the program.");

}

}

#### The first sample dialog:

Enter two numbers for numerator and denominator: 6 2

6/2 = 3

The division is done

End of the program.

##### Description:

**Exception** is a class in Java. This class is the root of any other java exception classes (see the following figure). The sub-class exceptions are more specific.

The class: *ArithmeticException* in the sample dialog of example 2 is one of these sub-classes. In this round of execution we do not have division by zero. The denominator is 2. All the instructions in the try-block including: System.*out*.println("The division is done")is executed. Since no exception occurs nothing from the catch-block is executed. ***The control of execution jumps over the catch-block*** and executes any instruction after the catch-block which is: System.*out*.println("End of the program.").

#### The second sample dialog:

Enter two numbers for numerator and denominator: 7 0

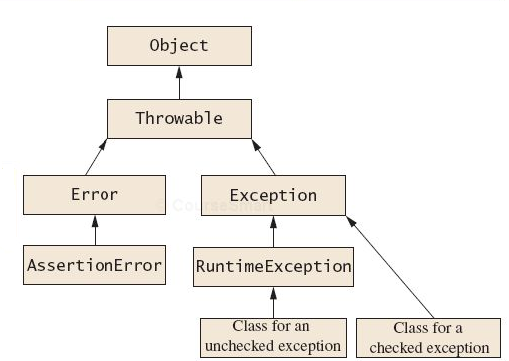
Division by zero is not correct

End of the program.

##### Description:

In this round of execution the denominator is zero. The control of execution executes the division and finds it is a division by zero. The control of execution does not execute the rest of the try-block which is: System.*out*.println("The division is done")and immediately jumps inside the catch-block and executes all the instructions. At this point unlike example 2 the execution of the program does not stop. The control of execution executes any instruction after the catch-block. In this case the instruction: System.*out*.println("End of the program.").

Note that you do not see the line: The division is done in output.



There are situations that we need more than one catch-block. To clarify this situation we run the program of example 2.

### Example 4:

The following sample dialog is the result of the program in example 2. In this round of execution the user enter a non-number data for the nominator which is a wrong data:

Enter two numbers for numerator and denominator: a 2

Exception in thread "main" java.util.InputMismatchException

at java.util.Scanner.throwFor(Unknown Source)

at java.util.Scanner.next(Unknown Source)

at java.util.Scanner.nextInt(Unknown Source)

at java.util.Scanner.nextInt(Unknown Source)

at Main.main(Main.java:7)

As we see we get a different kind of exception from a different exception class. The name of the class is: InputMismatchException.

Every java exception class is a sub-class of the class: Exception. So far we have seen two of them: ArithmeticException and InputMismatchException.

***Note: We do not need to remember all the sub-classes of the Exception class. All we need is to run the java program and get the error message and then use these classes.***

Now we would like to write a program to take care of both of these cases. These are: The user enters zero for denominator or a non-integer for any of the nominator or denominator. Obviously it is harder to include a conditional statement in a single catch-block than using two catch-blocks. When we use more than one catch block we cannot have the type of the parameters in both of them to be of the same type. The following is wrong:

try{

....

....

}catch(Exception e){

....

....

}catch(Exception u){

....

....

}

This is wrong and makes sense why the designer of the language made this situation to be wrong. At the time of an error in the try-block the control of execution does not know to which catch-block should jump.

### Example 5:

The following program is the same program but with two catch-blocks.

**import** java.util.InputMismatchException;

**import** java.util.Scanner;

**public** **class** Main{

**public** **static** **void** main(String[] args){

Scanner key = **new** Scanner(System.*in*);

System.*out*.print("Enter two numbers for numerator and denominator: ");

**try**{

**int** m = key.nextInt();

**int** n = key.nextInt();

System.*out*.println(m + "/" + n + " = " + (m/n));

System.*out*.println("The division is done");

}**catch**(ArithmeticException e){

System.*out*.println("Division by zero is not correct");

}**catch**(InputMismatchException e){

System.*out*.println("You must enter integers");

}

System.*out*.println("End of the program.");

}

}

#### The first sample dialog:

Enter two numbers for numerator and denominator: 7 0

Division by zero is not correct

End of the program.

##### Description:

The two instructions: **int** m = key.nextInt()and **int** n = key.nextInt()are inside the try-block. We did not include these two statements in the try-block in the previous programs because we were assuming the user only enters integers. Therefore no error occurs. Since no error occurs there was no need to jump into a catch-block. In the next sample dialog box dialog we read about this situation in more detailed.

In this program we have two catch-blocks. At the time of a division by zero the control of execution jumps to the first catch-block. The instruction of this catch block gets executed. After that the control of execution ***jumps over the second catch-block*** and executes the rest of the program.

#### The second sample dialog:

Enter two numbers for numerator and denominator: a 2

You must enter integers

End of the program.

##### Description:

In this case the user enters letter a instead of an integer. When the control of execution reaches to: **int** m = key.nextInt() and executes: key.nextInt() it finds the data is not an integer. An error occurs now. The control of execution does not execute the rest of the try-block. The control of execution also does not jump into the first catch-block. It ***jumps over the first catch-block*** and goes inside the second catch-block. The statement of the second catch-block gets executed and the control executes the rest of the program.

**Question:** Move: **int** m = key.nextInt() before the try-block and run the program. What happens?

There are times that we like to terminate the execution of the program once the control jumped to a catch-block and executed the instructions of the block. In this case we use: ***System.exit(n)***, where n is an optional positive integer.

### Example 6:

The following program terminates after a division by zero but continues its execution after a non-integer input:

**import** java.util.InputMismatchException;

**import** java.util.Scanner;

**public** **class** Main{

**public** **static** **void** main(String[] args){

Scanner key = **new** Scanner(System.*in*);

System.*out*.print("Enter two numbers for numerator and denominator: ");

**try**{

**int** m = key.nextInt();

**int** n = key.nextInt();

System.*out*.println(m + "/" + n + " = " + (m/n));

System.*out*.println("The division is done");

}**catch**(ArithmeticException e){

System.*out*.println("Division by zero is not correct");

}**catch**(InputMismatchException e){

System.*out*.println("You must enter integers");

**System.*exit*(0)**;

}

System.*out*.println("End of the program.");

}

}

#### The first sample dialog:

Enter two numbers for numerator and denominator: 7 0

Division by zero is not correct

End of the program.

##### Description:

No description is necessary. The program exactly behaves like the one in example 5 in its first sample dialog.

#### The second sample dialog:

Enter two numbers for numerator and denominator: a 2

You must enter integers

##### Description:

When the letter a is entered the execution of first: key.nextInt() .causes and exception and the control as in example 5 jumps to the second catch-block. The execution of: System.out displays the message: You must enter integers. The execution of: System.exit(0) terminates the execution of the program. Therefore the control does not go over the last System.out of the program.

Let us combine GUI and exceptions.

### Example 7:

The following adds two integers.

**import** javax.swing.\*;

**public** **class** Main {

**public** **static** **void** main(String[] args) {

**try**{

String a = JOptionPane.*showInputDialog*(

"Enter the first number.");

**int** i = Integer.*parseInt*(a);

String b = JOptionPane.*showInputDialog*(

"Enter the second number.");

**int** j = Integer.*parseInt*(b);

JOptionPane.*showMessageDialog* (**null**,

"The sum of " + i + " and " + j +

"is: " + (i + j));

}**catch**(Exception e){

JOptionPane.*showMessageDialog* (**null**,

"Error: Operands must be integers.");

System.*exit*(0);

}

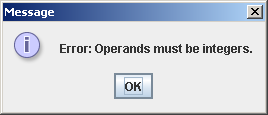
}

}

**A screen dialog**:



When OK is clicked:



#### Description:

The only thing we need to remember is that anything the user enters in box of the ***showInputDialog*** is treated as a string. Therefore in the above case we need to parse it as integer. If the user does not enter an integer an exception happens and we need to take care of this situation (as we did in the example).

**Exercise 1**: Change the program to divide two integers. You should take care of division by zero.

So far we did not use the parameter of an Exception. This parameter is holding the error message. This is the error message that is displayed by the Java interpreter if we do not write a try-catch block.

**Example 8**: The following program is the similar to the one of example 6

**import** java.util.\*;

**public** **class** Main{

**public** **static** **void** main(String[] args){

Scanner key = **new** Scanner(System.*in*);

System.*out*.print("Enter two numbers for numerator and denominator: ");

**try**{

**int** m = key.nextInt();

**int** n = key.nextInt();

System.*out*.println(m + "/" + n + " = " + (m/n));

System.*out*.println("The division is done");

}**catch**(ArithmeticException e){

System.*out*.println("No zero for denominator" + e);

}**catch**(InputMismatchException e){

System.*out*.println("You must enter integer" + e);

}

}

}

#### The first sample dialog:

Enter two numbers for numerator and denominator: 7 0

No zero for denominator java.lang.ArithmeticException: / by zero

##### Description:

The above underlined message is the result of printing the content of the parameter e of the first catch-block.

#### The second sample dialog:

Enter two numbers for numerator and denominator: a 2

You must enter integer java.util.InputMismatchException

##### Description:

The underlined message this time is the result of printing the content of the parameter e of the second catch-block.

## 2. Changing the size of a frame to fit in the screen

The default size of our frame is (400, 300), where the width is 400 and the height is 300. It would be nice to change our window to fit the screen. For this we need to include:

*import Java.awt.\*;*

Add the following instruction at the end of method: initGUI:

Dimension screenSize = Toolkit.*getDefaultToolkit*().getScreenSize();

setSize(screenSize.width, screenSize.height-50);

### Example 9:

The following program fits a frame to the screen.

**import** java.awt.\*;

**import** javax.swing.WindowConstants;

**import** javax.swing.SwingUtilities;

**public** **class** Main **extends** javax.swing.JFrame {

**public** **static** **void** main(String[] args) {

SwingUtilities.*invokeLater*(**new** Runnable() {

**public** **void** run() {

Main inst = **new** Main();

inst.setLocationRelativeTo(**null**);

inst.setVisible(**true**);

}

});

}

**public** Main() {

**super**();

initGUI();

}

**private** **void** initGUI() {

**try** {

setDefaultCloseOperation(WindowConstants.*DISPOSE\_ON\_CLOSE*);

pack();

//setSize(400, 300);

} **catch** (Exception e) {

e.printStackTrace();

}

Dimension screenSize = Toolkit.*getDefaultToolkit*().getScreenSize();

setSize(screenSize.width, screenSize.height-50);

}

}

#### Description:

SwingUtilities.*invokeLater*(**new** Runnable() This line of code uses an anonymous class that implements Runnable in order to let you specify what you want to execute on the EDT (Event Dispatch Thread). The Runnable interface requires you to implement the run() method. Then the EDT will call on your class object run() when it wants to. It's a way to pass a certain amount of instructions to the AWT Event Dispatch Thread (EDT). This is needed, because it's bad practice to operate on AWT-related things (like JFrames and all of it's children) from other Threads. You better give it to the EDT and let it handle it at the appropriate moment.

setLocationRelativeTo(**null**); This allows you set the location of the window. It normally requires you to call the setSize(). It essentially looks like a single dimensionless point (the top left corner of the window)....which is what it will centre.

The last two instructions will enlarge the frame to fit to the screen. Note that we decrease 50 from the height to be able to have:Icons to minimize, expand, or close a window. on top of the screen. If we remove 50 the frame gets so big that these icons won’t show up.

#### 2. Checkboxes, I/O windows

/\*

\* A simple swing checkbox example with different constructors

\*/

**import** javax.swing.JFrame;

**import** javax.swing.JPanel;

**import** javax.swing.BorderFactory;

**import** javax.swing.JCheckBox;

**public** **class** SimpleCheckbox {

**public** **static** **void** main(String[] args) {

// Create and set up a frame window

JFrame.*setDefaultLookAndFeelDecorated*(**true**);

JFrame frame = **new** JFrame("Simple checkbox demo");

frame.setDefaultCloseOperation(JFrame.***EXIT\_ON\_CLOSE***);

// Define the panel to hold the checkbox

JPanel panel = **new** JPanel();

// Create checkbox with different constructors

JCheckBox checkbox1 = **new** JCheckBox("Apple", **true**);

JCheckBox checkbox2 = **new** JCheckBox("Banana");

JCheckBox checkbox3 = **new** JCheckBox("Grape", **true**);

JCheckBox checkbox4 = **new** JCheckBox("Orange");

JCheckBox checkbox5 = **new** JCheckBox("Pear", **true**);

//Set up the title for the panel

panel.setBorder(BorderFactory.*createTitledBorder*("Fruits"));

//Add the checkbox into the panels

panel.add(checkbox1);

panel.add(checkbox2);

panel.add(checkbox3);

panel.add(checkbox4);

panel.add(checkbox5);

//Add the panel into the frame

frame.add(panel);

//Set the window to visible. Default is false.

frame.pack();

frame.setVisible(**true**);

}

}

Run the program and you get the following output:

A window with title "Simple checkbox demo" and five check boxes under Fruits:
Apple - checked
Banana - not checked
Grape - checked
Orange - not checked
Pear - checked

### Example 11:

We would like to write a program to enter two numbers via input dialog boxes and display their addition by an output dialog box.

**import** javax.swing.\*;

**public** **class** Main {

**public** **static** **void** main(String[] args) {

String a = "", b = "";

**int** i = 0, j = 0;

a = JOptionPane.*showInputDialog*("Enter the first number.");

b = JOptionPane.*showInputDialog*("Enter the second number.");

i = Integer.*parseInt*(a);

j = Integer.*parseInt*(b);

JOptionPane.*showMessageDialog* (**null**, "The sum of " + i +

" and " + j + " is: " + (i + j));

}

}

This program works as long as the user enters two integers in the input dialog boxes. But if the user enters in dialog boxes something else (string, floating point numbers, etc) JVM (Java Virtual Machine) issues an exception and terminate the program.

From the first lecture we decided to control the exceptions in our program by using try-catch block. The following program accomplishes this.

### Example 12:

Example 11 with a try-catch block.

**import** javax.swing.\*;

**public** **class** Main {

**public** **static** **void** main(String[] args) {

String a = "", b = "";

**int** i = 0, j = 0;

a = JOptionPane.*showInputDialog*("Enter the first number.");

b = JOptionPane.*showInputDialog*("Enter the second number.");

**try**{

i = Integer.*parseInt*(a);

j = Integer.*parseInt*(b);

}**catch**(Exception e){

System.*out*.println("Error: Program terminated because: " + e);

System.*exit*(0);

}

JOptionPane.*showMessageDialog* (**null**, "The sum of " + i +

" and " + j + "is: " + (i + j));

}

}

Any data, other than integers such as entering string axy and buv an exception will be caught with the message:

Error: Program terminated because: java.lang.NumberFormatException: For input string: "axy"

The program terminates because of the execution of the instruction:

*System*.*exit*(0);

## 3. Radio buttons

Radio buttons are similar to check boxes, but we cannot check (have a dot) more than one radio button. Let us explain this concept with the following example.

### Example 13:

We would like to write a program similar to the example 10. We use two radio buttons rather than two textboxes.

Adding simply radio buttons to the frame does not work. And the reason it doesn’t work because we as users run the program and click on the first radio button, a dot appears in it. But the problem is when the user clicks on the second radio button a dot appears in the button without the dot on the first button disappearing. The goal is to have only one dot at a time.

The solution is to group each set of radio buttons with an object of the type JButtonGroup.

Here is an example:

The solution is to group each set of radio buttons with an object of the type ButtonGroup.

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

import javax.swing.ButtonGroup;

import javax.swing.JButton;

import javax.swing.JRadioButton;

import javax.swing.WindowConstants;

import javax.swing.event.AncestorEvent;

import javax.swing.event.AncestorListener;

import javax.swing.SwingUtilities;

public class Radiobtn extends javax.swing.JFrame {

private JRadioButton jRadioButton1;

private JRadioButton jRadioButton2;

private JButton jButton1;

private ButtonGroup buttonGroup;

/\*\*

\* Auto-generated main method to display this JFrame

\*/

public static void main(String[] args) {

SwingUtilities.invokeLater(new Runnable() {

public void run() {

Radiobtn inst = new Radiobtn();

inst.setLocationRelativeTo(null);

inst.setVisible(true);

}

});

}

public Radiobtn() {

super();

initGUI();

}

private void initGUI() {

try {

setDefaultCloseOperation(WindowConstants.DISPOSE\_ON\_CLOSE);

getContentPane().setLayout(null);

{

jRadioButton1 = new JRadioButton();

getContentPane().add(jRadioButton1);

jRadioButton1.setText("First Radio Button");

jRadioButton1.setBounds(32, 26, 135, 20);

getButtonGroup().add(jRadioButton1);

jRadioButton1.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent evt) {

jRadioButton1ActionPerformed(evt);

}

});

}

{

jRadioButton2 = new JRadioButton();

getContentPane().add(jRadioButton2);

jRadioButton2.setText("Second Radio Button");

jRadioButton2.setBounds(32, 61, 144, 20);

getButtonGroup().add(jRadioButton2);

jRadioButton2.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent evt) {

jRadioButton2ActionPerformed(evt);

}

});

}

{

jButton1 = new JButton();

getContentPane().add(jButton1);

jButton1.setText("Click on a Radio Button");

jButton1.setBounds(32, 107, 173, 23);

jButton1.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent evt) {

jButton1ActionPerformed(evt);

}

});

jButton1.addAncestorListener(new AncestorListener() {

public void ancestorMoved(AncestorEvent evt) {

System.out.println("jButton1.ancestorMoved, event="+evt);

//TODO add your code for jButton1.ancestorMoved

}

public void ancestorAdded(AncestorEvent evt) {

jButton1AncestorAdded(evt);

}

public void ancestorRemoved(AncestorEvent evt) {

System.out.println("jButton1.ancestorRemoved, event="+evt);

//TODO add your code for jButton1.ancestorRemoved

}

});

}

pack();

setSize(400, 300);

} catch (Exception e) {

//add your error handling code here

e.printStackTrace();

}

}

private ButtonGroup getButtonGroup() {

if(buttonGroup == null) {

buttonGroup = new ButtonGroup();

}

return buttonGroup;

}

private void jRadioButton1ActionPerformed(ActionEvent evt) {

System.out.println("jRadioButton1.actionPerformed, event="+evt);

//TODO add your code for jRadioButton1.actionPerformed

}

private void jRadioButton2ActionPerformed(ActionEvent evt) {

System.out.println("jRadioButton2.actionPerformed, event="+evt);

//TODO add your code for jRadioButton2.actionPerformed

}

private void jButton1AncestorAdded(AncestorEvent evt) {

System.out.println("jButton1.ancestorAdded, event="+evt);

//TODO add your code for jButton1.ancestorAdded

}

private void jButton1ActionPerformed(ActionEvent evt) {

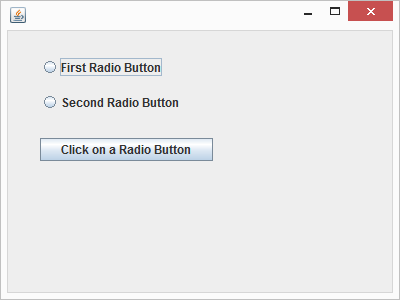
System.out.println("jButton1.actionPerformed, event="+evt);

//TODO add your code for jButton1.actionPerformed

}

}

The output looks like this:



Here is another example for Radio Button:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40 | import java.awt.FlowLayout;  import javax.swing.ButtonGroup;  import javax.swing.JFrame;  import javax.swing.JRadioButton;  import javax.swing.SwingUtilities;    public class SwingJRadioButtonDemo extends JFrame {        public SwingJRadioButtonDemo() {          super("Swing JRadioButton Demo");            JRadioButton option1 = new JRadioButton("Linux");          JRadioButton option2 = new JRadioButton("Windows");          JRadioButton option3 = new JRadioButton("Macintosh");            ButtonGroup group = new ButtonGroup();          group.add(option1);          group.add(option2);          group.add(option3);            setLayout(new FlowLayout());            add(option1);          add(option2);          add(option3);            pack();      }        public static void main(String[] args) {          SwingUtilities.invokeLater(new Runnable() {                @Override              public void run() {                  new SwingJRadioButtonDemo().setVisible(true);              }          });      }  } |