Lecture 11: Exception and Event Handling

This lecture discusses how programming languages support exceptions. Topics to be covered include:

- Introduction
- Exception Handling in C++
- Exception Handling in Java
- Exception Handling in Ada and Other Pascal-like Languages
- Event Handling

Copyright © 2000 – 2016 by Elvis C. Foster
All rights reserved. No part of this document may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, or otherwise, without prior written permission of the author.

11.1 Introduction

Inexperienced programmers usually think their program will always work as expected. On the other hand, experienced programmers know that things do not always work as expected. Smart programming is about taking care of the expected as well as the unexpected. Programmers refer to the unexpected situations as *exceptions*.

The following are some examples of scenarios that will cause program errors (exceptions):
- The user enters a character where an integer is expected;
- The program uses an array subscript that is outside of the range of valid subscript values for a given array;
- An attempt is made at dividing by zero;
- An attempt is made to write to a file that does not exist.

There are three broad categories of programming errors:
- Syntax errors
- Logic errors
- Runtime errors

We have already shown how programming languages take care of syntax errors through the translation process (review lecture 3). Logic errors are the responsibility of the programmer, but programming languages help by providing debugging features that the programmer can use. Runtime errors are typically handled by exceptions. When an exception occurs, unless the programmer catches and resolves it, the program terminates abnormally and you get a compiler exception message.

Some compilers provide clear exception messages that are easy to track and correct. The Java compiler is particularly outstanding in this area, while the C compiler is perhaps close to the other extreme.

11.2 Exception Handling in C++

C++ treats all exceptions similarly. You can throw an exception from any function. You can also catch an exception, and report it, or re-throw it for another function to negotiate. Exceptions are typically handled in **try-catch** blocks of code. Figure 11.1a provides the basic syntactical structure for C++ exception handling. The essence of the internal workings is as follows: A **try-block** executes code that throws at least one exception; for each exception thrown, there is a **catch-block** that handles recovery from the exception.

Figure 11.1a is followed by an example in figure 11.1b. In this illustration, the function **divide (…)** divides its first parameter by the second parameter, but throws an exception if the second parameter is zero. Because this function throws an exception, it must be called in a **try-catch** block as shown.

Figure 11.1a: Syntactical Structure of C++ Exception Handling

```
try {
// Code from which an exception may be raised; multiple exceptions may be raised
}
// . . .
catch (<Parameter for the exception>)
{
  // Exception-handling code
}
// . . .
catch (<Parameter for the exception>)
{
  // Exception-handling code; there is typically a catch-block for each exception raised in a preceding try-block
}
```

Figure 11.1b: Illustrating Exception Handling in C++

```
const int DIV_BY_ZERO = 10;  // A global constant
// …
double divide (double First, double Second); // Function prototype
// …

int main(int argc, char* argv[])
{
   // …
   try
  {
          double Result;
          double FirstNumber;
          // …
          Result = divide(FirstNumber, 0);
     // …
   }

   catch (int ErrNum)
  {
          if  (ErrNum == DIV_BY_ZERO) cout << "Cannot divide by zero!\n";
          // …
   }

   // …
}  // End of main

double divide (double First, double Second)
{
          if (Second == 0) throw DIV_BY-ZERO;
          return First/Second;
}
// …
```

11.2 Exception Handling in C++ (continued)

Figure 11.2 provides a second example. Here, an array is used to store all error messages that will be sent by the executing program. As scenarios develop, exception messages are pulled from this array and sent to the user. This generic approach can be applied to any program.

Figure 11.2: Using try-block and catch-block

```
#include <cstdlib>
#include <iostream>
#include <string>
#include <ctype.h>

const int REFERENCE = 1980000;
string ErrMsg[10]; // This array contains all error messages displayed by the program

int main(int argc, char *argv[])
{
  int ThisNumber;
  string ThisName;
  Initialize(); // Initialize the message array called ErrMsg
  // …
  try {
    InputData (ThisNumber, ThisName);
    // …
  }
  catch (int ErrNum)
 {
   cout  << ErrMsg[ErrNum] << end;
  }
 // … Rest of the program
}  // End of main
//…
// Other Functions
void InputData (int &ThisNumber, string &ThisName) // The InputData Function
{
  cout << ”Enter ID Number:”;   cin>> ThisNumber; getchar();
  if (ThisNumber < REFERENCE) throw (0);
  cout << endl;
  cout<<”Enter Name:”;
  cin >> ThisName; char FirstBytes[3] = (ThisName.substr(0,2)).c_str();
  if (!isalpha(FirstBytes[0])) throw (1);
  // …
}
void Initialize () // Initialization Method
{
  // Initialize ErrMsg array…
  ErrMsg[0] = “Invalid ID Number”;
  ErrMsg[1] = “Invalid Name”;
  ErrMsg[2] = “Invalid Date of Birth”;
 // …
} // End of Initialization Method
```

11.3 Exception Handling in Java

Java provides a much more elaborate and stratified exception handling system, consisting of a hierarchy of exception classes. Each Java method that throws an exception must have an adjustment in its heading to indicate what type of exception is thrown. The **try-catch** blocks are used in a manner that is similar to C++. Figures 11.3 – 11.8 provide additional clarification followed by an example.

Figure11.3: The Java Exception Class Hierarchy

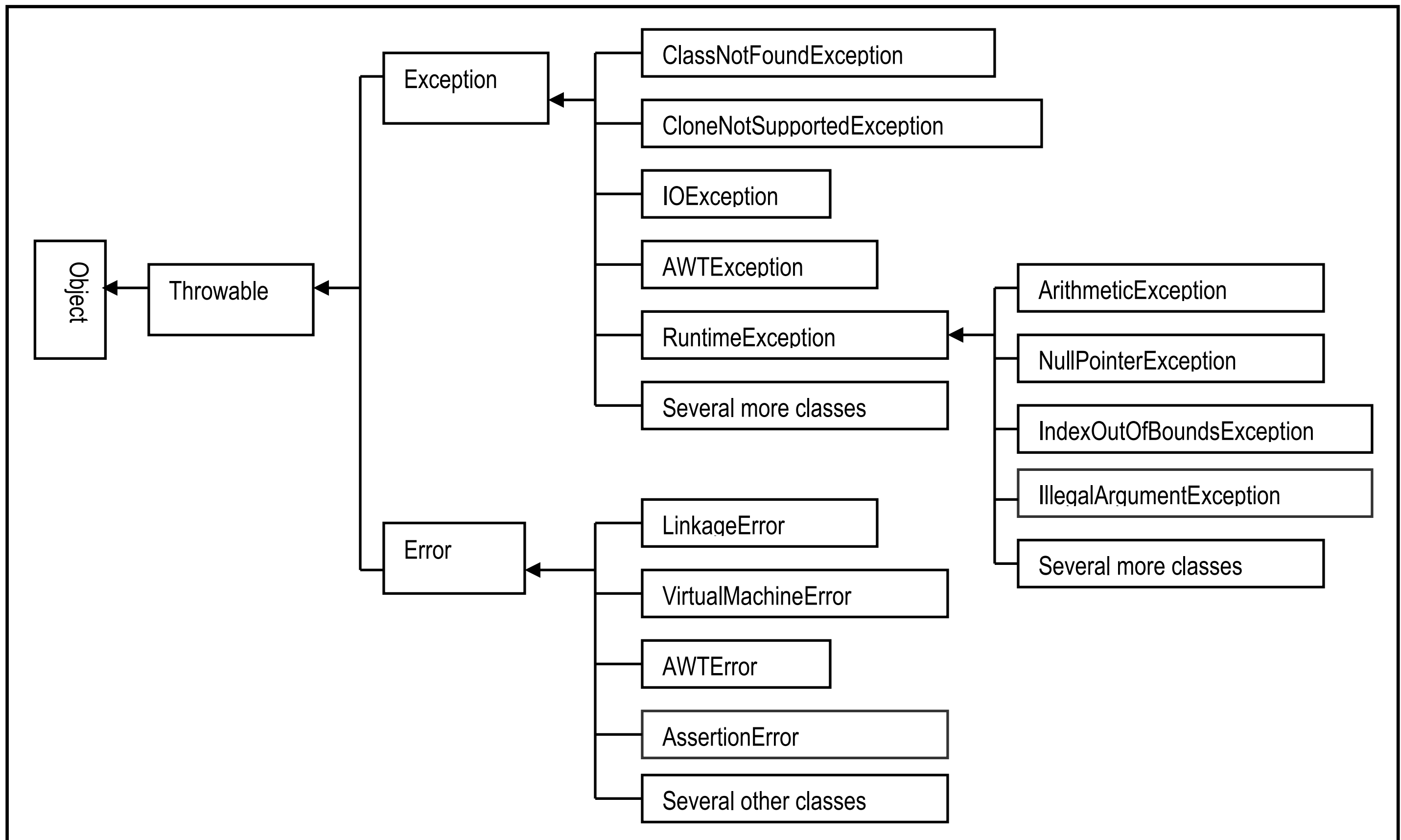

Figure 11.4: Revised Anatomy of a Java Method

```
Method ::=
<Modifier><ReturnType><MethodName>(<Parameters>) throws <Exception1>,…<ExceptionN>
{
  … // Body of Method
}

Modifier ::=
[final] public | private | protected [static] [abstract]
```

Note: Exception1… ExceptionN represent valid exception class(es) as indicated in figure 11.3

Figure 11.5: Simple Illustration of a try-catch Block

```
public void InputData () throws Exception
{
        String InputString;
        try
        {
            int InputInteger = Integer.parseInt(JOptionPane.showInputDialog (null, "Integer Please!",  +
                    "Integer Prompt"));
        }
        catch (Exception Ex)
        {
            System.out.println(Ex.getMessage( )); // Prints the message of the exception
            System.out.println("Invalid integer entered"); // Prints additional message
        }
        …
}
```

Figure 11.6: Illustrating Exception Throwing in Java

```
public void AcceptData() throws Exception
  {
   int x = 0; // Used for input validation
   char SameChar; // Used for input validation
   try
   {
     ID_Number = JOptionPane.showInputDialog(null, "Member ID: ", "Enter College Member Data", JOptionPane.QUESTION_MESSAGE);
     for (x=1; x<= ID_Number.length(); x++) // For each character in ID-Number
     {   // Ensure that the input is numeric
        SameChar = ID_Number.charAt(x-1);  // ThisChar = SameChar;
        if (!(Character.isDigit(SameChar))) // If not numeric
        {
           Exception Ex1 = new Exception("ID_Number must be numeric.");
           throw Ex1;
        }
     } // End-For each character in ID-Number

     FirstName = JOptionPane.showInputDialog(null, "First Name: ", "Enter College Member Data", JOptionPane.QUESTION_MESSAGE);
     for (x=1; x<= FirstName.length(); x++) // For each character in FirstName
     {   // Ensure that the input is alphabetic
        SameChar = FirstName.charAt(x-1); // ThisChar = SameChar;
        if (!(Character.isLetter(SameChar))) // If not alphabetic
        {
           Exception Ex2 = new Exception("FirstName must be alphabetic.");
           throw Ex2;
        }
     } // End-For each character in FirstName

     LastName = JOptionPane.showInputDialog(null, "Last Name: ", "Enter College Member Data",  JOptionPane.QUESTION_MESSAGE);
     for (x=1; x<= LastName.length(); x++) // For each character in LastName
     {   // Ensure that the input is alphabetic
        SameChar = LastName.charAt(x-1); //  ThisChar = SameChar;
        if (!(Character.isLetter(SameChar))) // If not alphabetic
        {
           Exception Ex3 = new Exception("LastName must be alphabetic.");
           throw Ex3;
       }
    } // End-For each character in LastName

     AddressLine1 = JOptionPane.showInputDialog(null, "Address Line 1: ", "Enter College Member Data", JOptionPane.QUESTION_MESSAGE);
     AddressLine2 = JOptionPane.showInputDialog(null, "Address Line 2: ", "Enter College Member Data", JOptionPane.QUESTION_MESSAGE);
     StateProv = JOptionPane.showInputDialog(null, "State or Province: ", "Enter College Member Data", JOptionPane.QUESTION_MESSAGE);
     Zip = JOptionPane.showInputDialog(null, "Zip Code: ", "Enter College Member Data", JOptionPane.QUESTION_MESSAGE);

     Telephone = JOptionPane.showInputDialog(null, "Telephone: ", "Enter College Member Data", JOptionPane.QUESTION_MESSAGE);
     for (x=1; x<= Telephone.length(); x++) // For each character in Telephone
     {   // Ensure that the input is numeric
        SameChar = Telephone.charAt(x-1); // ThisChar = new Character(SameChar);
        if (!((Character.isDigit(SameChar)) || (SameChar == '-'))) // If not numeric
        {
           Exception Ex4 = new Exception("Telephone number must be numeric, delimited with dashes.");
           throw Ex4;
        }
     } // End-For each character in Telephone
     E_Mail = JOptionPane.showInputDialog(null, "E-Mail: ", "Enter College Member Data", JOptionPane.QUESTION_MESSAGE);
   } // End try-block

   catch (Exception Ex) // Catches and re-throws any of the exceptions thrown from above
   {
     throw Ex;
   }
  } // End of AcceptData method
```

Figure11.7: Catching and Re-throwing an Exception

```
try
{
   <statement(s)>;
 }
 catch (<ExceptionClass> Ex)
 {
      ...
      throw Ex; // Throws the exception, then exits
 }
 ...
```

Figure 11.8: Illustrating the finally-block

```
try
{
   <statement(s)>;
}
catch (<ExceptionClass> Ex)
{
   <statement(s)>;
}
...
finally
{
   <statement(s)>;
}
...
```

11.4 Exception Handling in Ada and Other Pascal-like Languages

Ada allows you to define exception handlers via the construct shown in figure 11.9. The following guidelines apply:

- The statement(s) to handle the exception can be placed in a block (commenced by the keyword **begin**, and concluded by the keyword **end**), or in a procedure, in which case the **when-statement** simply calls the procedure.
- The keyword **others** is a special keyword that is used to mean any kind of exception.
- The **ExceptionName** must be a valid Ada exception, or a user-defined exception.
- There are four default exceptions: **Constraint-Error**, **Program-Error**, **Storage-Error**, and **Tasking-Error**.
- The **raise-statement** is used to raise (i.e. throw) an exception.
- User-defined exceptions are defined using the construct

```
<ExceptionName> {,<ExceptionName>}: exception
```

Figure 10.9: Defining Exception Handlers in Ada

```
when <ExceptionChoice> {|<ExceptionChoice>}
 then  <Statement(s)>

ExceptionChoice ::= <ExceptionName> | others
```

This approach to exception handling is common to many Pascal-like languages such as Object Pascal and Oracle's PL/SQL. Figure 11.10 provides an illustration in Oracle's PL-SQL. This is a procedure that manages exceptions in that language.

Figure 11.10: A PL-SQL Procedure that Handles Exceptions

```
Create Procedure QueryEmp (ThisEmp In emp.empno%Type)
As
 ThisEmpRec emp%RowType;
 SalaryNull Exception;

 Cursor EmpCursor is
 SELECT * FROM emp WHERE empno >= &ThisEmp;

Begin
 Open EmpCursor;
 <<Hunt>>
 Loop
  Fetch EmpCursor Into ThisEmpRec;
  Exit Hunt when EmpCursor%NOTFOUND;
  If ThisEmpRec.Sal IS NULL Then
     Raise SalaryNull;
  End If;
  DBMS_Output.Put_Line(ThisEmpRec.Empno || ' ' || ThisEmpRec.Ename);
 End Loop;
 Close EmpCursor;

Exception
 When No_Data_Found then  /* Implicit exception */
  DBMS_Output.Put_Line ('No data found');
 When SalaryNull then    /* Explicit exception */
  DBMS_Output.Put_Line ('Salary is null');
 When Others then /* catch all errors */
  DBMS_Output.Put_Line ('There is an execution error');
End;
```

11.5 Event Handling

Event handling is somewhat similar to exception handling in the sense that in both cases, the handler is implicitly called in response to an occurrence. However, there is a subtle but important difference between the two: An exception typically originates from hardware/software interactions external to the immediate code, or from the code itself. In contrast, an event always originates from an external interaction (for instance via a GUI-based user response) with the executing code.

Typical sources of events are mouse-clicks (single or double) by the end-user, mouse-drags, and mouse-drops. Such user actions trigger corresponding signals (i.e. events) to the executing program, for which there must be responses; these responses are coded as event handlers within the code. Following is a brief overview of how event handling in Java, C#, and C++.

11.5.1 Event Handling in Java

As you are probably aware, Java ships with a number of packages. One such package is **javax.swing**, consisting of a number of classes that support GUI programming. The **javax.swing** package contains a number of Java classes that may be employed by the Java programmer; figure 11.11 lists some of the important classes along with important methods in these classes.

Each of these classes contains methods for manipulating instances of the related class, and taking appropriate actions in response to events triggered by end-user actions. Take for instance the **JTextField** class. You will observe that this class has several overloaded constructors for instantiating instances of the class. Observe also that the class has several overloaded methods with the name **getAction()**. These overloaded methods are intended to send events back to the executing program based on end-user interaction with an instance of the **JTextField** class. Depending on the event received, the programmer can code appropriate responsive actions in an event handler for the **JTextField** instance in question.

While this approach is quite straightforward and easy to follow, the main criticism of the language is the multiplicity of classes that the Java programmer has to familiarize himself/herself with, thus defeating the objective of understandability as discussed in lecture 1.

Figure 11.11: Commonly Used Methods of Some Basic GUI Component Classes

The JButton Class (inherits from Object, Component, Container, JComponent and AbstractButton)	
Method	**Comment**
public JButton( )	Creates a **JButton** instance without text or icon.
public JButton(Action anAction)	Creates a **JButton** with properties from the **Action** instance specified.
public JButton(Icon anIcon)	Creates a **JButton** instance with an icon.
public JButton(String aString)	Creates a **JButton** instance with text.
public JButton(String aString, Icon anIcon)	Creates a **JButton** instance with text and icon.
protected String paramString( )	Returns a string representation of the button.
// Other inherited methods	
The JLabel Class (inherits from Object, Component, Container, and JComponent)	
Method	**Comment**
public JLabel( )	Creates a **JLabel** with no specified image or text.
public JLabel(Icon image)	Creates a **JLabel** with an image.
public JLabel(Icon image, int hAlignment)	Creates a **JLabel** with specified image and horizontal alignment.
public JLabel(String text)	Creates a **JLabel** with the specified test.
public JLabel(String text, int hAlignment)	Creates a **JLabel** with specified text and horizontal alignment.
public JLabel(String text, Icon image, int hAlignment)	Creates a **JLabel** with specified text, image and horizontal alignment.
public int getHorizontalAlignment( )	Returns the horizontal alignment along the X axis.
public int getHorizontalTextPosition( )	Returns the horizontal position of the label's text, relative to its image.
public Icon getIcon( )	Returns the icon for the label.
public Component getLabelFor( )	Returns the **labelFor** field (the component for which the label applies).
public int getVerticalAlignment( )	Returns the vertical alignment along the Y axis.
public int getVerticalTextPosition( )	Returns the vertical position of the label's text, relative to its image.
protected String paramString( )	Returns a string representation of the label.
public void setHorizontalAlignment(int x)	Sets the alignment of the label's contents along the X axis.

Figure 10.1: Commonly Used Methods of Some Basic GUI Component Classes (continued)

The JLabel Class (continued)	
Method	**Comment**
public void setHorizontalTextPosition(int x)	Sets the horizontal position of the label's text, relative to its image.
public void setIcon(Icon anIcon)	Defines the icon this component will display.
public void setLabelFor(Component aComp)	Sets the component that this label is labeling.
public void setText(String newText)	Sets the label's text
public void setVerticalAlignment(int y)	Sets the alignment of the label's contents along the Y axis.
public void setVerticalTextPosition(int y)	Sets the vertical position of the label's text, relative to its image.
// Other inherited methods	
The JTextField Class (inherits from Object, Component, Container, JComponent, and JTextComponent)	
Method	**Comment**
public JTextField( )	Creates a new **JTextField** instance.
public JTextField(Document aDoc, String Text, int Cols)	Creates a new **JTextField** instance that uses the given text storage model and the given number of columns.
public JTextField(int Cols)	Creates a new empty **JTextField** instance with the given number of columns.
public JTextField(String Text)	Creates a new **JTextField** instance initialized to the specified text.
public JTextField(String Text, int Cols)	Creates a new **JTextField** instance initialized with the specified text and columns.
public Action getAction( )	Returns the currently set Action for this **ActionEvent** source, or null if no Action is set.
public ActionListener[ ] getActionListeners( )	Returns an array of all the **ActionListeners** added to this **JTextField** with **addActionListener()** method.
public Action[ ] getActions( )	Fetches the command list for the editor.
public int getColumns( )	Returns the number of columns in this **JTextField**.
public int getHorizontalAlignment( )	Returns the horizontal alignment of the text along the X axis..
protected String paramString( )	Returns a string representation of the **TJTextField** object.
public void removeActionListener(ActionListener l)	Removes the specified action listener so that it no longer receives action events from **JTextField** object.
public void setColumns(int Cols)	Sets the number of columns in this **JTextField** object, and then invalidate the layout.
public void setDocument(Document aDoc)	Associates the editor with a text document.
public void setFont(Font aFont)	Sets the current font.
public void setHorizontalAlignment(int x)	Sets the alignment of the **JTextField** object's contents along the X axis.
public void setscrollOffset(int x)	Sets the scroll offset, in pixels.
// Other inherited methods	
The JCheckBox Class (inherits from Object, Component, Container, JComponent, AbstractButton, and JToggleButton)	
Method	**Comment**
public JCheckBox( )	Creates a new unchecked checkbox with no text and no icon.
public JCheckBox(Action anAct)	Creates a check box where properties are taken from the Action supplied.
public JCheckBox(Icon anIcon)	Creates a new unchecked checkbox with an icon and no text.
public JCheckBox(Icon anIcon, boolean Sel)	Creates a check box with an icon and specifies whether or not it is initially selected.

Figure 10.1: Commonly Used Methods of Some Basic GUI Component Classes (continued)

The JCheckBox Class (continued)	
Method	**Comment**
public JCheckBox(String Text)	Creates an initially unchecked check box with text.
public JCheckBox(String Text, boolean Sel)	Creates a check box with text; specifies whether it is initially selected.
public JCheckBox(String Text, Icon anIcon)	Creates an initially unchecked check box with text and an icon.
public JCheckBox(String Text, Icon anIcon, boolean Sel)	Creates an initially unchecked check box with text and an icon; specifies whether it is initially selected.
protected String paramString( )	Returns a string representation of the check box.
// Other inherited methods	
The JRadioButton Class (inherits from Object, Component, Container, JComponent, AbstractButton, and JToggleButton)	
Method	**Comment**
public JRadioButton( )	Creates a new unchecked radio button with no text and no icon.
public JRadioButton (Action anAct)	Creates a radio button where properties are taken from the Action supplied.
public JRadioButton (Icon anIcon)	Creates a new unchecked radio button with an icon and no text.
public JRadioButton (Icon anIcon, boolean Sel)	Creates a radio button with an icon and specifies whether or not it is initially selected; no text.
public JRadioButton (String Text)	Creates an initially unchecked radio button with text.
public JRadioButton (String Text, boolean Sel)	Creates a radio button with text; specifies whether it is initially selected.
public JRadioButton (String Text, Icon anIcon)	Creates an initially unchecked radio button with text and an icon.
public JRadioButton (String Text, Icon anIcon, boolean Sel)	Creates an initially unchecked radio button with text and an icon; specifies whether it is initially selected.
protected String paramString( )	Returns a string representation of the radio button.
The JComboBox Class (inherits from Object, Component, Container, and JComponent)	
Method	**Comment**
public JComboBox( )	Creates a combo box with a default data model.
public JComboBox(ComboBoxModel aModel)	Creates a combo box that takes it's items from an existing **ComboBoxModel** instance.
public JComboBox(Object[ ] Items)	Creates a combo box that contains the elements in the specified array.
public void addActionListener(ActionListener aListener)	Adds an ActionListener instance.
public void addItem(Object anObject)	Adds an item to the item list.
public void addItemListener(ItemListener aListener)	Adds an ItemListener instance
public void addPopupMenuListener (PopupMenuListener aListener)	Adds a PopupMenuListener which will listen to notification messages from the popup portion of the combo box.
protected void fireActionEvent( )	Notifies all listeners that have registered interest for notification on this event type of an action event.
protected void fireItemStateChanged (ItemEvent anEvent)	Notifies all listeners that have registered interest for notification on this event type of an item state change.
public void firePopupMenuCancelled( )	Notifies PopupMenuListeners that the popup portion of the combo box has been canceled.

Figure 10.1: Commonly Used Methods of Some Basic GUI Component Classes (continued)

The JComboBox Class (continued)	
Method	**Comment**
public void firePopupMenuWillBecomeInvisible( )	Notifies PopupMenuListeners that the popup portion of the combo box has become invisible.
public void firePopupMenuWillBecomeVisible ( )	Notifies PopupMenuListeners that the popup portion of the combo box has become visible.
public Action getAction ( )	Returns the currently set Action for this ActionEvent source, or null if no Action is set.
public String getActionCommand ( )	Returns the action command that is included in the event sent to action listeners.
public ActionListener [ ] getActionListeners( )	Returns an array of all the ActionListeners added to this JComboBox with the **addActionListener()** method.
public int getItemCount( )	Returns the number of items in the list.
public ItemListener [ ] getItemListeners ( )	Returns an array of all the ItemListeners added to this JComboBox with addItemListener().
public int getMaximumRowCount ( )	Returns the maximum number of items the combo box can display without a scrollbar.
public ComboBoxModel getModel ( )	Returns the data model currently used by the combo box.
public PopupMenuListener [ ] getPopupMenuListeners ( )	Returns an array of all the PopupMenuListeners added to this combo box with the **addPopupMenuListener()** method.
public int getSelectedIndex ( )	Returns the first item in the list that matches the given selected item.
public Object getSelectedItem ( )	Returns the current selected item.
public void insertItemAt (Object anObj, int x)	Inserts an item into the item list at a given index.
public boolean isEditable ( )	Returns true if the combo box is editable; false otherwise.
public boolean isPopupVisible ( )	Returns true if the popup is visible; false otherwise.
protected String paramString( )	Returns a string representation of the combo box.
public void removeActionListener (ActionListener aListener)	Removes an ActionListener.
public void removeAllItems ( )	Removes all items from the item list.
public void removeItem (Object anObj)	Removes an item from the item list.
public void removeItemAt (int x)	Removes the item at position **x**. This method works only if the combo box uses a mutable data model.
public void removeItemListener (ItemListener aListener)	Removes an ItemListener.
public void removePopupMenuListener (PopupMenuListener aListener)	Removes a PopupMenuListener.
public void setEditable ( )	Determines whether the combo box field is editable.
public void setEditor (ComboBoxEditor anEditor)	Sets the editor used to paint and edit the selected item in the combo box field.
public void setEnabled ( )	Enables the combo box so that items can be selected.
public void setMaximumRowCount (int x)	Sets the maximum number of rows the combo box displays.
public void setModel (ComboBoxModel aModel)	Sets the data model that the combo box uses to obtain the list of items.
public void setPopupVisible (boolean vFlag)	Sets the visibility of the popup.
public void setSelectedIndex (int x)	Selects the item at index **x**.
public void setSelectedItem (Object anObj)	Sets the selected item in the combo box display area to the object in the argument.
public void showPopup ( )	Causes the combo box to display its popup window.
// Other inherited methods	

11.5.2 Event Handling in C#

Event handling in C# is similar to the Java implementation described above. Obviously, the language-shipped class-names that the programmer uses are different. Additionally, the C# programmer has at his/her disposal a number of scaffoldings of the .NET framework. Moreover, the event-handling exercise appears to be a bit more simplified than in a Java environment. This is not surprising, since the designers of C# benefitted significantly from observing the strengths and challenges of the Java and C++ environments. In the recommended readings, you will observe that there are two entries that provide further clarification on event handling in C#.

11.5.3 Event Handling in C++

As is the case for multithreading, the C++ language does not inherently treat the matter of event handling, but instead defers to IDEs that provide such mechanisms. Fortunately, there are several excellent products that are available in the marketplace, including C++ Builder, Qt, and Simple DirectMedia Layer (SDL). For each of these products, you would need to learn the various nuances of the user interface and its interaction with C++, but that should not pose a significant challenge, especially since they all embrace the basic concepts of events and event-handling as described earlier. Information on these products is readily available on the Worldwide Web (WWW). Additionally, there is no shortage of C++ enthusiasts who provide useful tips on solving common programming problems using the language; one such reference is included in the recommended readings.

11.6 Summary and Concluding Remarks

Let us summarize what has been covered in this lecture:

- Exception handling is a common strategy that programming languages provide for recovering from runtime errors.
- C++ employs a very simple but highly effective and efficient exception-handling mechanism that involves the use of **try-blocks** and **catch-blocks**.
- Java adopts the basic C++ approach to exception handling, but adds more sophistication by managing a complex hierarchy of exception classes for various identified categories of exceptions.
- The approach in Ada and other Pascal-like languages is somewhat different from the approach in C-based languages: An exception may be raised (i.e. thrown) from various points in the executing program. Once raised, recovery from the exception must be made by one or more exception handling statements, each triggered by a **when** statement. These exception handling statements typically appear toward the end of the related subprogram in an exception block.
- Event handling is somewhat similar to exception handling in the sense that in both cases, the handler is implicitly called in response to an occurrence. However, the important difference is that whereas an exception typically originates from hardware/software interactions external to the immediate code, or from the code itself, an event always originates from an external interaction (for instance via a GUI-based user response) with the executing code.
- An event handler provides the code necessary for responding to events received.
- Java provides an elaborate repository of classes whose instances may trigger events. The Java programmer has the responsibility of coding for these events and taking appropriate actions.
- C# adopts the basic Java approach to event handling, but benefits from scaffoldings provided by the .NET framework.
- C++ does not inherently treat event handling, but relies on host IDEs running the language to provide such event handling facilities.

11.6 Summary and Concluding Remarks (continued)

The information provided here should give you a solid foundation for understanding how issues related to exception handling and event handling are treated in contemporary programming languages. In our next lecture, we will examine procedural and functional languages.

11.7 Recommended Readings

[Code Project 2014] Code Project. 2014. "Events and handling in C#." Accessed on December 26, 2014. http://www.codeproject.com/Articles/1474/Events-and-event-handling-in-C

[Landheer-Cieslak 2010] Landheer-Cieslak, Ronald. 2010. "Making Life Easier: Event handling in C and C++." Accessed December 26, 2014. http://rlc.vlinder.ca/blog/2010/08/event-handling-in-c-and-c/

[Microsoft 2014] Microsoft. 2014. "Handling and Raising Events." Accessed December 26, 2014. http://msdn.microsoft.com/en-us/library/edzehd2t%28v=vs.110%29.aspx

[Sebesta 2012] Sebesta, Robert W. 2012. *Concepts of Programming Languages* 10th Edition. Colorado Springs, Colorado: Pearson. See chapter 14.