Rewarding Learning

AS1 LEVEL

FACT FILES

Software Systems Development

For first teaching from September 2013

For first AS award in Summer 2014

For first A Level award in Summer 2015

Introduction to Object Oriented Development Part (vi) – Exception Handling

software systems development

FACT FILE

software systems development

Part (vi) – Exception Handling

Learning Outcomes

Students should be able to:

- explain the need to be able to trap errors in code;
- explain ways that errors can be trapped in an object oriented environment and apply associated techniques;
- try / catch (blocks).

Content

Error Trapping in Code

Error handling refers to the anticipation, detection and resolution of programming errors. The compiler will detect all syntax errors, such as, undeclared variables and invalid structures or methods. Logical errors in the code may only become apparent during runtime. End users can inadvertently cause errors by entering invalid data, storage paths for files and data may have been corrupted or an attempt could be made to divide by zero or the program could run out of memory. These runtime errors are called **exceptions**.

Good software systems prevent errors where possible and recover from them when they arise, without terminating the application. More serious errors that result in termination are dealt with, allowing the application to close cleanly and error information to be stored to a log file.

Poorly designed software systems may freeze up the computer upon encountering an exception, often requiring a reboot.

Exception Handling - Structured Exception Handling (SEH):-
try{} catch{} … finally{}

It is difficult to produce 100% error free applications and it is advisable to include code to handle errors that may arise during execution. These include errors such as invalid data values entered by the end-user or invalid IO file paths. These errors will cause the application to crash and inconvenience the end-user. These type of errors which are beyond the control of the developer should be trapped and dealt with cleanly.

Try blocks surround a section of code for which exception handling is advisable. The **try** block terminates at the point where the exception occurred and branches to the **catch** blocks.

Catch blocks provide code that is executed only if an exception occurs whether custom, common or a general type of exception. Multiple catch blocks can be implemented and are hierarchical.

The **finally** block specifies code that is executed after exception handling has occurred, or after the **try** block finishes. (Note if you include any catch blocks, then the finally block is optional.)

Some common Exceptions provided by C#:-

System.Exception - Base class of exceptions.

System.ArithmeticException - A base class for exceptions that occur during arithmetic operations, such as DivideByZeroException and OverflowException.

System.FormatException - Attempt to store invalid data for the data type specified.

System.IndexOutOfRangeException - Thrown, for example, when an attempt is made to index an array via an index that is less than zero or outside the bounds of the array.

System.OutOfMemoryException - Thrown when an attempt to allocate memory (via new) fails because all available memory has been used up.

System.IOException - A base class for exceptions that occur during Input/Output operations.

System.FileNotFoundException

System.EndOfStreamException

Example of use of Exception Handling:-

1. **using c# Exception**

```
public static void WriteToReport(Car[] cars)
{
        FileStream s = new FileStream("CarReport.txt", FileMode.Create);
        StreamWriter sw;

        try
        {
           // write report
           sw = new StreamWriter(s);
           sw.WriteLine();
           sw.WriteLine("{0,40}", "Car Report");
           sw.WriteLine();
           sw.WriteLine();
           for (int x = 0; x < cars.Length; x++)
           {
              sw.WriteLine(  cars[x].toString()  );
           }
           sw.WriteLine();
           sw.WriteLine("       Number of Makes  - {0}", cars.Length);
           sw.Close();
        }
        catch (Exception e)
        {
           Console.WriteLine(  e.Message );
        }
}
```

2. Throwing Exceptions

It is not always possible to deal with errors in the method where they occur. The solution is to **throw** the error back to the calling method (s).

The simplest format is:- **throw new Exception();**

Or we can pass a string to be used as the Message for the Exception.

throw new Exception(" Error :…….");

These can be caught in the calling method or passed back through several levels.
The levels can be accessed using e.StackTrace

3. Catching Several Exception Types

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;

namespace ProjExceptionExamples
{
   class TestException
   {
      class FileOperations
      {
         public void FileOpen(String filename)
         {
            //.....
            throw new IOException("Error opening file  " + filename);
         }
         public void FileRead()
         {
         }
      }

      public static void Main()
      {
         // Catches the thrown error
         try
         {
            FileOperations fOps = new FileOperations();
            fOps.FileOpen("c:\\myFile.txt");
            fOps.FileRead();
         }
         catch (IOException e)
         {
            Console.WriteLine(e.Message);
         }
         catch (Exception e)     //must be last in series of exceptions  - base exception
         {
            Console.WriteLine(e.Message);
         }
         Console.Read();
      }
   }
```

4. Deriving your own custom Exception Classes

The custom Exception class is derived from class Exception.

```
class MyException : Exception
  {
     //constructors
     public MyException()
        : base()
     {
        LogException();
     }
     public MyException(string message)
        : base(message)
     {
        LogException();
     }
      //method to log error to console
     protected void LogException()
     {
        Console.WriteLine("[MyException.LogException] " + "logging '{0}' ", this.Message);
     }
  }

public class TestException
{

     static void DoWork()
     {
        // code....
        throw new MyException("Error found in doWork");
     }
     public static void Main()
     {
        // Catches the thrown error
        try
        {
           FileOperations fOps = new FileOperations();
           fOps.FileOpen("c:\\myFile.txt");
           fOps.FileRead();

           //Use custom MyException
           DoWork();
        }
        catch (IOException e)
        {
           Console.WriteLine(e.Message);
        }
        catch (Exception e)     //must be last in series of exceptions  - base exception
        {
           //can use the specific catch(MyException e)
           Console.WriteLine(e.Message +"  "+ e.GetType()  +"  "+  e.StackTrace);
        }
        Console.Read();
     }
}
```

Best Practice for Exception Handling

Your catch should…

- Indicate to the end user that an error has occurred, what the error is and what to do next.
- Log it – put some meaningful information in to the program log file, or a message to the console. This is necessary for useful debugging.
- Do not ignore the exception – when the application has thrown an exception, it means that your code flow is not as expected. Don't simply suppress the exception and carry on as though it never happened.

Questions

Q1 Define the following terms:

a. Exception
b. Exception Handling
c. Try block
d. Catch block

Q2 Is it possible to guarantee that the program you have written handles all possible exceptions? Explain your answer.

Q3 Match up the inputs to the resulting outputs in the following code block. The first one has been done for you.

```
try
{
    answer = 2/input;
}
 catch( Exception e)
{
     answer =4;
}
finally
{
      Console.WriteLine(answer);
}
```

Input		Output
2		2
1		4
0		1
4		0.5

Q4 You are writing a program to book clients into a gym. People over 55 get a 25% discount. How would you write the program?

a. Use an **if** statement to test for age 55 or greater. If a person over 55 is detected, throw an exception. A catch{ } block will apply the discount.
b. Use an **if-else** statement.
c. Subtract 55 from the client's age and divide the bill by the result. If an ArithmeticException is thrown, apply the 25% discount in the exception handler.

COUNCIL FOR THE CURRICULUM EXAMINATIONS AND ASSESSMENT
29 Clarendon Road, Clarendon Dock, Belfast BT1 3BG
✆ +44 (0)28 9026 1200 🖷 +44 (0)28 9026 1234 ☎ +44 (0)28 9024 2063 ✉ info@ccea.org.uk 🌐 www.ccea.org.uk