COIS1020H: Programming for Computing Systems

Chapter 11
Exception Handling

Understanding Exceptions

- **Exception**
  - Any error condition or unexpected behavior in an executing program
    - An excellent way of validating input data
- **Exception handling**
  - Object-oriented techniques used to manage such errors
- Exceptions are objects of the `Exception` class
  - Or one of its derived classes

Class	Description
System.ArgumentException	Thrown when one of the arguments provided to a method is not valid
System.ArithmeticException	Thrown for errors in an arithmetic, casting, or conversion operation
System.ArrayTypeMismatchException	Thrown when an attempt is made to store an element of the wrong type within an array
System.Data.OperationAbortedException	Thrown when an ongoing operation is aborted by the user
System.Drawing.Printing InvalidPrinterException	Thrown when you try to access a printer using printer settings that are not valid
System.FormatException	Thrown when the format of an argument does not meet the parameter specifications of the invoked method
System.IndexOutOfRangeException	Thrown when an attempt is made to access an element of an array with an index that is outside the bounds of the array; this class cannot be inherited
System.InvalidCastException	Thrown for an invalid casting or explicit conversion
System.InvalidOperationException	Thrown when a method call is invalid for the object's current state
System.IO.InvalidDataException	Thrown when a data stream is in an invalid format
System.IO.IOException	Thrown when an I/O error occurs
System.MemberAccessException	Thrown when an attempt to access a class member fails
System.NotImplementedException	Thrown when a requested method or operation is not implemented
System.NullReferenceException	Thrown when there is an attempt to dereference a null object reference
System.OperationCanceledException	Thrown in a thread upon cancellation of an operation that the thread was executing
System.OutOfMemoryException	Thrown when there is not enough memory to continue the execution of a program
System.RankException	Thrown when an array with the wrong number of dimensions is passed to a method
System.StackOverflowException	Thrown when the execution stack overflows because it contains too many nested method calls; this class cannot be inherited

Table 11-1 Selected C# Exceptions

Purposely Generating a `SystemException`

- You can deliberately generate a `SystemException`
  - By forcing a program to contain an error
  - Example
    - Dividing an integer by zero
- You don't necessarily have to deal with exceptions
- Termination of the program is abrupt and unforgiving
- Object-oriented error-handling techniques provide more elegant solutions

Purposely Generating a `SystemException` (cont'd.)

```
using System;
public class MilesPerGallon
{
   public static void Main()
   {
      int milesDriven;
      int gallonsOfGas;
      int mpg;
      Console.Write("Enter miles driven ");
      milesDriven = Convert.ToInt32(Console.ReadLine());
      Console.Write("Enter gallons of gas purchased ");
      gallonsOfGas = Convert.ToInt32(Console.ReadLine());
      mpg = milesDriven / gallonsOfGas;
      Console.WriteLine("You got {0} miles per gallon", mpg);
   }
}
```

Figure 11-1 `MilesPerGallon` program

Purposely Generating a `SystemException` (cont'd.)

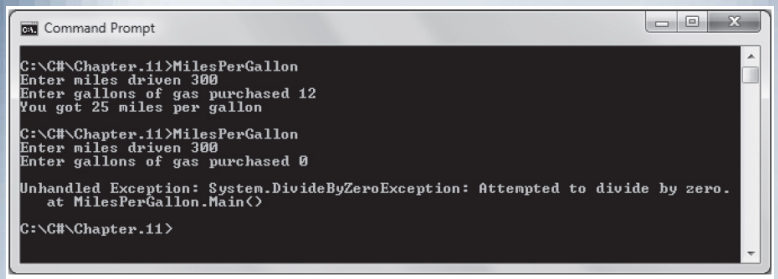

Figure 11-2 Two executions of `MilesPerGallon` program

Understanding Traditional and Object-Oriented Error-Handling Methods

- Check a variable's value with an `if` statement before attempting to divide it into another number
  - Prevents division by zero
    - However, it does not really "handle an exception"
  - Efficient if you think it will be a frequent problem
    - Has little "overhead"
    - Otherwise, create a `SystemException` or `Exception` object
      - `Exception` is the parent class

7

Understanding Object-Oriented Error-Handling Methods

- **`try` block**
  - Contains statements that can produce an error
- Code at least one `catch` block or `finally` block
  - Immediately following a `try` block
- **`catch` block**
  - Can "catch" one type of `Exception`

8

Understanding Object-Oriented Exception-Handling Methods (cont'd.)

```
try
{
   // Any number of statements;
   // some might cause an Exception
}
catch(XxxException anExceptionInstance)
{
   // Do something about it
}
// Statements here execute whether there was an Exception or not
```

Figure 11-3 General form of a `try...catch` pair

Understanding Object-Oriented Exception-Handling Methods (cont'd.)

```
using System;
public class MilesPerGallon2
{
   public static void Main()
   {
      int milesDriven;
      int gallonsOfGas;
      int mpg;
      try
      {
         Console.Write("Enter miles driven ");
         milesDriven = Convert.ToInt32(Console.ReadLine());
         Console.Write("Enter gallons of gas purchased ");
         gallonsOfGas = Convert.ToInt32(Console.ReadLine());
         mpg = milesDriven / gallonsOfGas;

      }
      catch (Exception e)
      {
         mpg = 0;
         Console.WriteLine("You attempted to divide by zero!");
      }
      Console.WriteLine("You got {0} miles per gallon", mpg);
   }
}
```

```
Enter miles driven 300
Enter gallons of gas purchased 12
You got 25 miles per gallon

Enter miles driven 300
Enter gallons of gas purchased 0
You attempted to divide by zero!
You got 0 miles per gallon
```

Figure 11-4 `MilesPerGallon2` program

Using the `Exception` Class's `ToString()` Method and `Message` Property

- `Exception` class overrides `ToString()`
  - Provides a descriptive error message
  - User can receive precise information about the nature of any `Exception` that is thrown

Using the `Exception` Class's `ToString()` Method (cont'd.)

```
using System;
public class MilesPerGallon3
{
   public static void Main()
   {
      int milesDriven;
      int gallonsOfGas;
      int mpg;
      try
      {
         Console.Write("Enter miles driven ");
         milesDriven = Convert.ToInt32(Console.ReadLine());
         Console.Write("Enter gallons of gas purchased ");
         gallonsOfGas = Convert.ToInt32(Console.ReadLine());
         mpg = milesDriven / gallonsOfGas;
      }
      catch(Exception e)
      {
         mpg = 0;
         Console.WriteLine(e.ToString());
      }
      Console.WriteLine("You got {0} miles per gallon", mpg);
   }
}
```

```
Enter miles driven 300
Enter gallons of gas purchased 0
System.DivideByZeroException: Attempted to divide by zero.
   at MilesPerGallon3.Main() in C:\RTH\Visual Studio
2010\Projects\Project1\Project1\CodeFile1.cs:line 15
You got 0 miles per gallon
```

message

Figure 11-6 `MilesPerGallon3` program

Using the `Exception` Class's Method `Message` Property

- `Exception` class `Message` property
  - Contains useful information about an `Exception`

```
using System;
public class MilesPerGallon4
{
   public static void Main()
   {
      int milesDriven;
      int gallonsOfGas;
      int mpg;
      try
      {
         Console.Write("Enter miles driven ");
         milesDriven = Convert.ToInt32(Console.ReadLine());
         Console.Write("Enter gallons of gas purchased ");
         gallonsOfGas = Convert.ToInt32(Console.ReadLine());
         mpg = milesDriven / gallonsOfGas;
      }
      catch(Exception e)
      {
         mpg = 0;
         Console.WriteLine(e.Message);
      }
      Console.WriteLine("You got {0} miles per gallon", mpg);
   }
}
```

```
Enter miles driven 300
Enter gallons of gas purchased 0
Attempted to divide by zero.
You got 0 miles per gallon
```

`Second part of ToString()`

Figure 11-8 MilesPerGallon4 program

Validating Input Data (Version 1)

```
using System;

public class ValidateInput1
{
   public static void Main()
   {
      int numPennies;
      Console.Write("Enter the number of pennies => ");
      numPennies = Convert.ToInt32(Console.ReadLine());
      Console.WriteLine("The number of pennies entered is {0}", numPennies);
      Console.ReadLine();
   }
}
```

```
Enter the number of pennies => 9
The number of pennies entered is 9

Enter the number of pennies => -4
The number of pennies entered is -4

Enter the number of pennies => cat
Boom!
```

- Standard program which inputs data
  - No validation performed

Validating Input Data (Version 2)

```
using System;

public class ValidateInput2
{
   public static void Main()
   {
      int numPennies;
      do
      {
         Console.Write("Enter the number of pennies => ");
         numPennies = Convert.ToInt32(Console.ReadLine());
      } while (numPennies < 0);
      Console.WriteLine("The number of pennies entered is {0}", numPennies);
      Console.ReadLine();
   }
}
```

```
Enter the number of pennies => 4
The number of pennies entered is 4

Enter the number of pennies => -4
Enter the number of pennies => 9
The number of pennies entered is 9

Enter the number of pennies => cat
Boom!
```

- Protects against negative input only

15

Validating Input Data (Better Version1)

```
using System;

public class ValidateInput3
{
   public static void Main()
   {
      int numPennies=0;
      bool notValid = true;
      do
      {
         try
         {
            Console.Write("Enter the number of pennies => ");
            numPennies = Convert.ToInt32(Console.ReadLine());
            if (numPennies < 0)
               Console.WriteLine("The number of pennies must be positive");
            else
               notValid = false;
         }
         catch (FormatException e)
         {
            Console.WriteLine("The number of pennies must be a positive integer");
         }
      } while (notValid);
      Console.WriteLine("The number of pennies entered is {0}", numPennies);
      Console.ReadLine();
   }
}
```

```
Enter the number of pennies => -4
The number of pennies must be positive
Enter the number of pennies => cat
The number of pennies must be a positive integer
Enter the number of pennies => 9
The number of pennies entered is 9
```

16

Validating Input Data (Better Version2)

```
using System;

public class ValidateInput3
{
   public static void Main()
   {
      int numPennies=0;
      bool notValid = true;
      do
      {
         try
         {
            Console.Write("Enter the number of pennies => ");
            numPennies = Convert.ToInt32(Console.ReadLine());
            if (numPennies < 0)
               throw (new ApplicationException());
            else
               notValid = false;
         }
         catch (FormatException e)
         {
            Console.WriteLine("The number of pennies must be a positive integer");
         }
          catch (ApplicationException e)
         {
            Console.WriteLine("The number of pennies must be positive");
         }
      } while (notValid);
      Console.WriteLine("The number of pennies entered is {0}", numPennies);
      Console.ReadLine();
   }
}
```

```
Enter the number of pennies => -4
The number of pennies must be positive
Enter the number of pennies => cat
The number of pennies must be a positive integer
Enter the number of pennies => 9
The number of pennies entered is 9
```


Catching Multiple `Exceptions`

- You can place as many statements as you need within a `try` block
  - Only the first error-generating statement throws an `Exception`
- Multiple `catch` blocks are examined in sequence
  - Until a match is found for `Exception` that occurred
- Various `Exceptions` can be handled by the same `catch` block

Catching Multiple `Exceptions` (cont'd.)

```
using System;
public class TwoErrors
{
   public static void Main()
   {
      int num = 13, denom = 0, result;
      int[] array = {22, 33, 44};
      try
      {
         result = num / denom;
         result = array[num];
      }
      catch(DivideByZeroException error)
      {
         Console.WriteLine("In first catch block: ");
         Console.WriteLine(error.Message);
      }
      catch(IndexOutOfRangeException error)
      {
         Console.WriteLine("In second catch block: ");
         Console.WriteLine(error.Message);
      }
   }
}
```

In first catch block:
Attempted to divide by zero.

Figure 11-10 TwoErrors program with two catch blocks

Catching Multiple `Exceptions` (cont'd.)

```
using System;
public class TwoErrors2
{
   public static void Main()
   {
      int num = 13, denom = 0, result;
      int[] array = {22, 33, 44};
      try
      {
         result = array[num];
         result = num / denom;
      }
      catch(DivideByZeroException error)
      {
         Console.WriteLine("In first catch block: ");
         Console.WriteLine(error.Message);
      }
      catch (IndexOutOfRangeException error)
      {
         Console.WriteLine("In second catch block: ");
         Console.WriteLine(error.Message);
      }
   }
}
```

In second catch block:
Index was outside the bounds of the array.

Figure 11-12 The TwoErrors2 program

Catching Multiple `Exception`s (cont'd.)

- Poor coding style for a method to `throw` more than three or four types
  - Method is trying to accomplish too many diverse tasks
  - `Exception` types thrown are too specific and should be generalized
- **Unreachable** blocks
  - Contain statements that can never execute under any circumstances
    - Because the program logic "can't get there"
    - Eg. Having two `catch` blocks where first catches an `Exception` and the second a `DivideByZeroException`
      - the second would never be selected

Using the `finally` Block

- **`finally` block**
  - Contains actions to perform at the end of a `try...catch` sequence
  - Executes whether the `try` block identifies any `Exception`s or not
  - Used to perform clean-up tasks
- A `finally` block executes after:
  - The `try` ends normally
  - The `catch` executes
  - The `try` ends abnormally and the `catch` does not execute

Using the `finally` Block (cont'd.)

```
try
{
   // Statements that might cause an Exception
}
catch(SomeException anExceptionInstance)
{
   // What to do about it
}
finally
{
   // Statements here execute
   // whether an Exception occurred or not
}
```

Figure 11-15 General form of a `try...catch` block with a `finally` block

Using the `finally` Block (cont'd.)

```
try
{
   // Open the file
   // Read the file
   // Place the file data in an array
   // Calculate an average from the data
   // Display the average
}
catch(IOException e)
{
   // Issue an error message
   // Exit
}
finally
{
   // If the file is open, close it
}
```

Figure 11-16 Format of code that tries reading a file and handles an `Exception`

Creating Your Own `Exception` Classes

- To create your own `Exception`
  - Extend the `ApplicationException` or the `Exception` class

```
public class NegativeBalanceException : Exception
{
   private static string msg = "Bank balance is negative. ";
   public NegativeBalanceException() : base(msg)
   {
   }
}
```

Figure 11-28 The `NegativeBalanceException` class

Creating Your Own `Exception` Classes (cont'd.)

```
public class BankAccount
{
   private double balance;
   public int AccountNum {get; set;}
   public double Balance
   {
      get
      {
         return balance;
      }
      set
      {
         if(value < 0)
         {
            NegativeBalanceException nbe =
               new NegativeBalanceException();
            throw(nbe);
         }
         balance = value;
      }
   }
}
```

Figure 11-29 The `BankAccount` class

Creating Your Own `Exception` Classes (cont'd.)

```
using System;
public class TryBankAccount
{
    public static void Main()
    {
        BankAccount acct = new BankAccount();
        try
        {
            acct.AccountNum = 1234;
            acct.Balance = -1000;
        }
        catch (NegativeBalanceException e)
        {
            Console.WriteLine(e.Message);
        }
    }
}
```

```
The bank balance is negative.
```


Validating Input Data (One More Time)

```
using System;
public class ValidateInput4
{
    public static void Main()
    {
        int numPennies = 0;
        bool valid;
        do
        {
            try
            {
                valid = true;
                Console.Write("Enter the number of pennies => ");
                numPennies = Convert.ToInt32(Console.ReadLine());
                if (numPennies < 0)
                {
                    throw (new InvalidRangeException("Number of pennies must be positive"));
                }
            }
            catch (FormatException e)
            {
                valid = false;
                Console.WriteLine("The number of pennies must be a positive integer");
            }
            catch (InvalidRangeException e)
            {
                valid = false;
                Console.WriteLine(e.Message);
            }
        } while (!valid);
        Console.WriteLine("The number of pennies entered is {0}", numPennies);
        Console.ReadLine();
    }
}
```

```
using System;
public class InvalidRangeException : Exception
{
    public InvalidRangeException(string errMsg): base(errMsg)
    { }
}
```

```
Enter the number of pennies => -4
Number of pennies must be positive
Enter the number of pennies => cat
The number of pennies must be a positive integer
Enter the number of pennies => 9
The number of pennies entered is 9
```

Rethrowing an `Exception`

- **Rethrow the `Exception`**
  - Let the calling method handle the problem
  - Method that catches an `Exception` does not have to handle it
- Within a `catch` block, you can rethrow the `Exception` that was caught to the method that called your method
  - By using the keyword `throw` with no object after it

Rethrowing an `Exception` (cont'd.)

```
using System;
public class ReThrowDemo
{
   public static void Main()
   {
      try
      {
         Console.WriteLine("Trying in Main() method");
         MethodA();
      }
      catch(Exception ae)
      {
         Console.WriteLine("Caught in Main() method --\n   {0}",
            ae.Message);
      }
      Console.WriteLine("Main() method is done");
   }
```

Figure 11-32 The ReThrowDemo program *(Continues)*

(Continued)

```
    public static void MethodA()
    {
       try
       {
          Console.WriteLine("Trying in method A");
          MethodB();
       }
       catch(Exception)
       {
          Console.WriteLine("Caught in method A");
          throw;
       }
    }
    public static void MethodB()
    {
       try
       {
          Console.WriteLine("Trying in method B");
          MethodC();
       }
       catch(Exception)
       {
          Console.WriteLine("Caught in method B");
          throw;
       }
    }
    public static void MethodC()
    {
       Console.WriteLine("In method C");
       throw(new Exception("This came from method C"));
    }
}
```

```
Trying in Main method
Trying in method A
Trying in method B
In method C
Caught in method B
Caught in method A
Caught in Main method --
 This came from method C
Main() method is done
```

Figure 11-32 The ReThrowDemo program

Summary

- An exception is any error condition or unexpected behavior in an executing program
- You can purposely generate a `SystemException` by forcing a program to contain an error
- When you think an error will occur frequently, it is most efficient to handle it in the traditional way
- In object-oriented terminology, you “try” a procedure that may not complete correctly
- Every `Exception` object contains a `ToString()` method and a `Message` property

Summary (cont'd.)

- You can place as many statements as you need within a `try` block
  - Catch as many different `Exception`s as you want
- When you have actions to perform at the end of a `try...catch` sequence, use a `finally` block
- When methods throw `Exception`s, they don't have to catch them
  - can be rethrown to the calling method
- To create your own `Exception` that you can throw, you can extend the `Exception` class