Chapter 11: Exception Handling
Understanding Exceptions

• **Exception**
  – Any error condition or unexpected behavior in an executing program

• **Exception handling**
  – Object-oriented techniques used to manage such errors

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

Table 11-1  Selected C# Exception subclasses
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.)

```csharp
using System;

class MilesPerGallon
{
    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 The MilesPerGallon program
Purposely Generating a SystemException (cont’d.)

Figure 11-2 Two executions of the 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”
  – Is efficient if you think it will be a frequent problem
    • Has little “overhead”
    • Otherwise, create an `Exception` object
Understanding Object-Oriented Exception-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**
Understanding Object-Oriented Exception-Handling Methods (cont’d.)

```csharp
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 11-5  General form of a try...catch pair
```
Understanding Object-Oriented Exception-Handling Methods (cont’d.)

```csharp
using System;

class MilesPerGallon2
{
    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);
    }
}

Figure 11-6  The MilesPerGallon2 program
```
Understanding Object-Oriented Exception-Handling Methods (cont’d.)

Figure 11-7  Two executions of the MilesPerGallon2 program
Using the **Exception Class's** **ToString()** Method and **Message** Property

• **Using The** Exception class and **ToString()**
  – Provides a descriptive error message
  – The user can receive precise information about the nature of any Exception that is thrown
Using the **Exception Class’ s** `ToString()` Method and `Message` Property (cont’d.)

```csharp
using System;
class MilesPerGallon3
{
    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);
    }
}
```

*Figure 11-9  The MilesPerGallon3 program*
Using the **Exception Class**’s `ToString()` Method and `Message` Property (cont’d.)

![Command Prompt window showing execution of the `MilesPerGallon3` program. The program prompts for miles driven and gallons of gas purchased, then throws a `DivideByZeroException` when attempting to divide by zero.]()

---

**Figure 11-10** Execution of the `MilesPerGallon3` program
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 the Exception that occurred

• Various Exceptions can be handled by the same catch block
Catching Multiple **Exceptions** (cont’d.)

```csharp
using System;
class TwoErrors
{
    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);
        }
    }
}
```

**Figure 11-13**  The TwoErrors program with two catch blocks
Catching Multiple Exceptions (cont’d.)

Figure 11-14    Output of the TwoErrors program
Catching Multiple Exceptions (cont’d.)

```csharp
using System;
class TwoErrors2
{
    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(OutOfRange error)
        {
            Console.WriteLine("In second catch block: ");
            Console.WriteLine(error.Message);
        }
    }
}
```

*Figure 11-15* The TwoErrors2 program
Catching Multiple Exceptions (cont’d.)

![Command Prompt output](image)

**Figure 11-16** Output of the TwoErrors2 program
using System;
class TwoErrors3
{
    static void Main()
    {
        int num = 13, denom = 0, result;
        int[] array = {22, 33, 44};
        try
        {
            result = array[num];
            result = num / denom;
        }
        catch(Exception error)
        {
            Console.WriteLine(error.Message);
        }
    }
}

Figure 11-17 The TwoErrors3 class with one catch block
Catching Multiple Exceptions (cont’d.)

• It is poor coding style for a method to throw more than three or four types of Exception
  – The method is trying to accomplish too many diverse tasks
  – The 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”
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 **Exceptions** 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.)

```csharp
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 11-20** General form of a `try...catch` block with a `finally` block
Using the finally Block (cont’d.)

```csharp
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 11-21  Format of code that tries reading a file and handles an exception