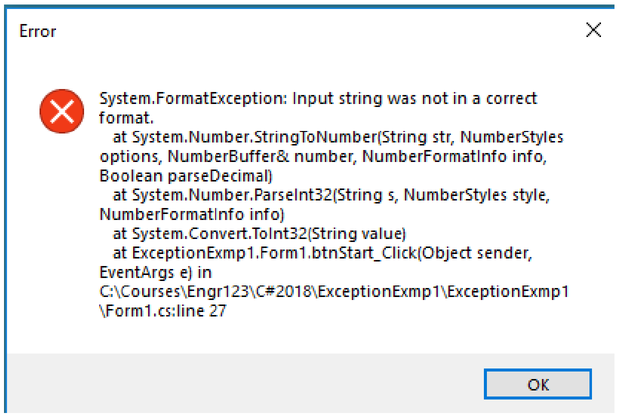
Engr 123 **April 24, 2019**
Day 19

Exceptions
Exception handling in C# provides a way to build a fault tolerant program such that, when the program generates an error it does not stop but issues a notification to the user and allows continuation of the program.

The basic building block for exception handling is the try/catch block.

For example:
```
            string dataIn;
            try
            {
                dataIn = txtNum1.Text;
                n1 = Convert.ToInt32(dataIn);
            }
            catch
            {
                MessageBox.Show("Error on entry.  Renter", "Error",
                     MessageBoxButtons.OK, MessageBoxIcon.Error);
                return;
            }
```
If an exception occurs in the TRY block the user will see the MessageBox and the program will return to the main screen.

You can get more information about the exception like this:
```
           string dataIn;
            try
            {
                dataIn = txtNum1.Text;
                n1 = Convert.ToInt32(dataIn);
            }
            catch (Exception ex)
            {
                string exc = ex.ToString();
                MessageBox.Show(exc, "Error",
                     MessageBoxButtons.OK, MessageBoxIcon.Error);
                return;
            }
```
Now for example, if the txtNum.Text contains some letters the Convert.ToInt32 will not be able to convert it to an int. The Catch statement will get the Exception as a message ex. Convert this to a string and print it with the messagebox. It may look something like this:



```
System.FormatException: Input string was not in a correct
format.
   at System.Number.StringToNumber(String str, NumberStyles
options, NumberBuffer& number, NumberFormatInfo info,
Boolean parseDecimal)
   at System.Number.ParseInt32(String s, NumberStyles style,
NumberFormatInfo info)
   at System.Convert.ToInt32(String value)
   at ExceptionExmp1.Form1.btnStart_Click(Object sender,
EventArgs e) in
C:\Courses\Engr123\C#2018\ExceptionExmp1\ExceptionExmp1
\Form1.cs:line 27
```

There are also a number of more specific exceptions which you can catch. There are hundreds of these. Some examples include:

```
FormatException
DivideByZeroException
FileNotFoundException
```

```
            string dataIn;
             try
             {
                 dataIn = txtNum1.Text;
                 n1 = Convert.ToInt32(dataIn);
             }
             catch(FormatException ex)
             {
                 MessageBox.Show(ex.ToString());
             }
             catch (Exception ex)
             {
                 string exc = ex.ToString();
                 MessageBox.Show(exc, "Error",
                      MessageBoxButtons.OK, MessageBoxIcon.Error);
                 return;
             }
```

Note that in this case we have two catch blocks but the program will execute on the first catch block that matches the exception.

If you put in a specific catch block, for example DivideByZero and an error occurs for some other reason the catch will be ignored and the system will catch the error and end the program.

It is also possible to nest try/catch block in other try blocks in the event that an error occurs in the error processing.

Collections

A collection is a data structure that can store items more generically without regard to how they get implemented. A big advantage is that you need not be concerned about the size of the collection. C# will make the collection whatever size is necessary by automatically getting more memory from the operating system.

There are a number of different collections but in this class we are only going to look at `ArrayList` and `List<T>`

An **ArrayList** can contain anything, int, double, string, or class elements and it need not contain all of the same type. For example:

```
ArrayList aList = new ArrayList(1); //Set initial capacity to 1
int i;
double d;
for(i=0;i<5;i++)
    aList.Add(i);
for(d=0;d<1;d+=.1)
    aList.Add(d);
for(i=0;i<aList.Count;i++)
    Console.WriteLine(aList[i]);
```

ArrayList has a number of functions. A few of them are listed below:

Add – adds an object to the end of the list as in `aList.Add(i);`

Capacity – gets or sets the number of spaces currently available in the list. This is not the same as the number of elements in the array.

Clear – Removes everything from the list as in `aList.Clear();`

Count – gets the number of items currently in the array as in `aList.Count;`

Remove – removes the first occurent of the argument as in `aList.Remove(5);`

Insert inserts an object at a specified index as in `aList.Insert(4, 3.2);`

Sort – Sorts the list but you have to write a separate sort routine to tell it how to sore the objects in the list.

A `List<T>` is the same as ArrayList except that it holds only one type of data – not mixed types. For example

`List<int> list1;`

In this case list1 is an ArrayList that holds only ints. You can make the type anything you want, including instances of a class but the list can only hold one type.