EXCEPTION HANDLING

For example, our program stops if we enter “cat” instead of “25” for an “int gallons” since gallons is integer.

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

//happens while execution.

These exceptions are child classes derived from the parent class “Exception”

• 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

Understanding Object-Oriented ErrorHandling Methods

• try block

– Contains statements that can produce an error

• Code at least one catch block or finally block

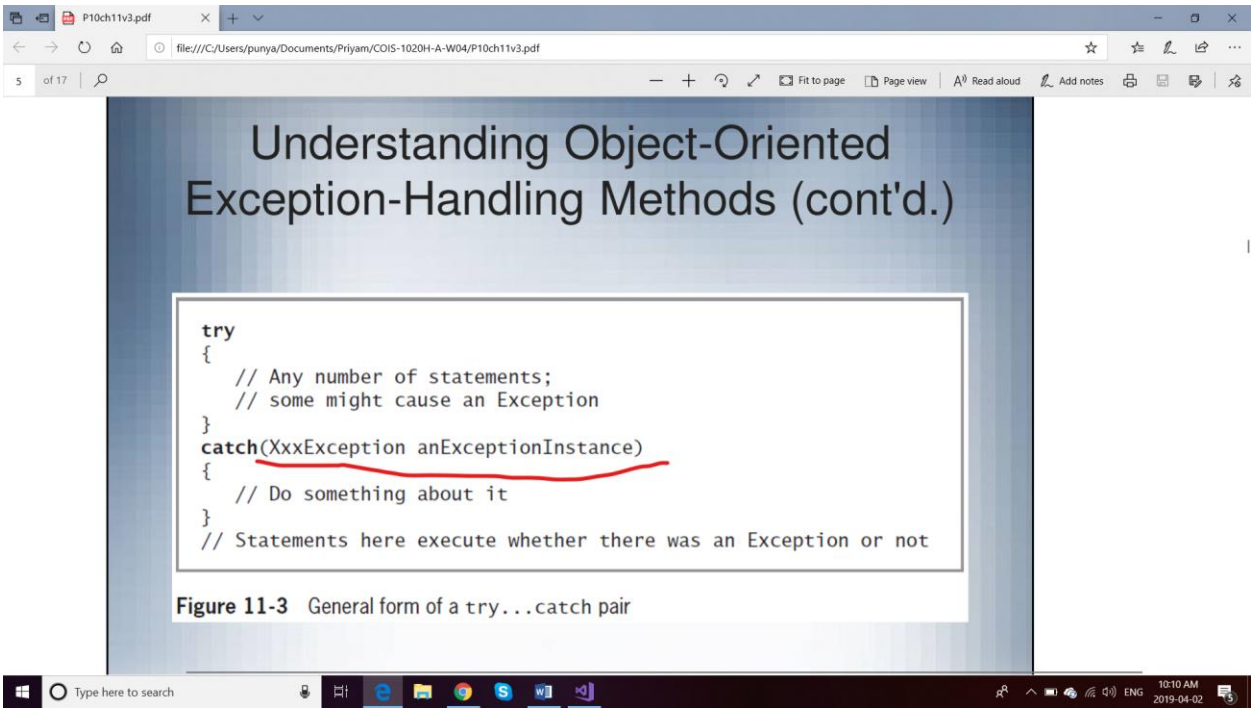
– Immediately following a try block

• catch block

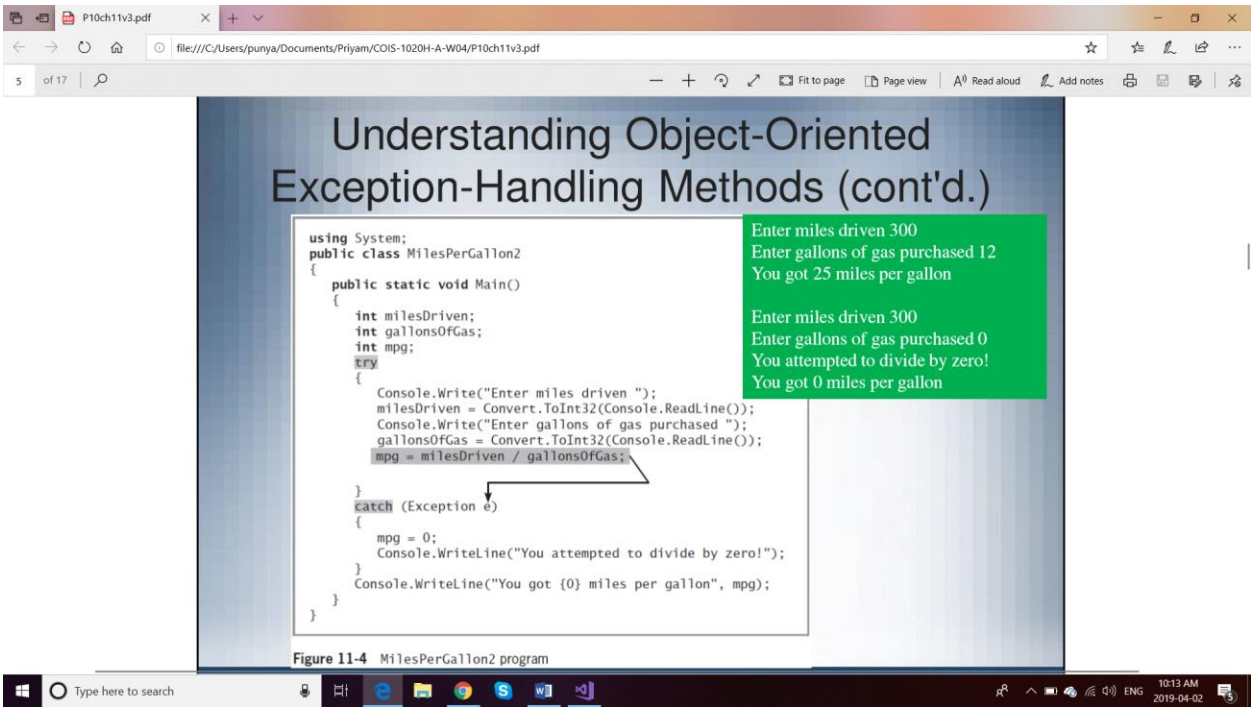
– Can “catch” one type of Exception

Try block does slow down execution.

We can have 2 or 3 catch blocks associated with a try block if we are looking for a specific error.



Any variable name can be given in the catch block



If we enter 0, error occurs at mpg=milesDriven/gallonsOfGas, we need to notice that, the program resumes from the next step where it would have gone if there was an error.

Also, if we enter cat in gallonsOfGas, the error occurs at Convert.ToInt32, it skips all the steps next and reaches catch, then resumes. However, will print that we tried dividing by zero, when error is we entered a string.

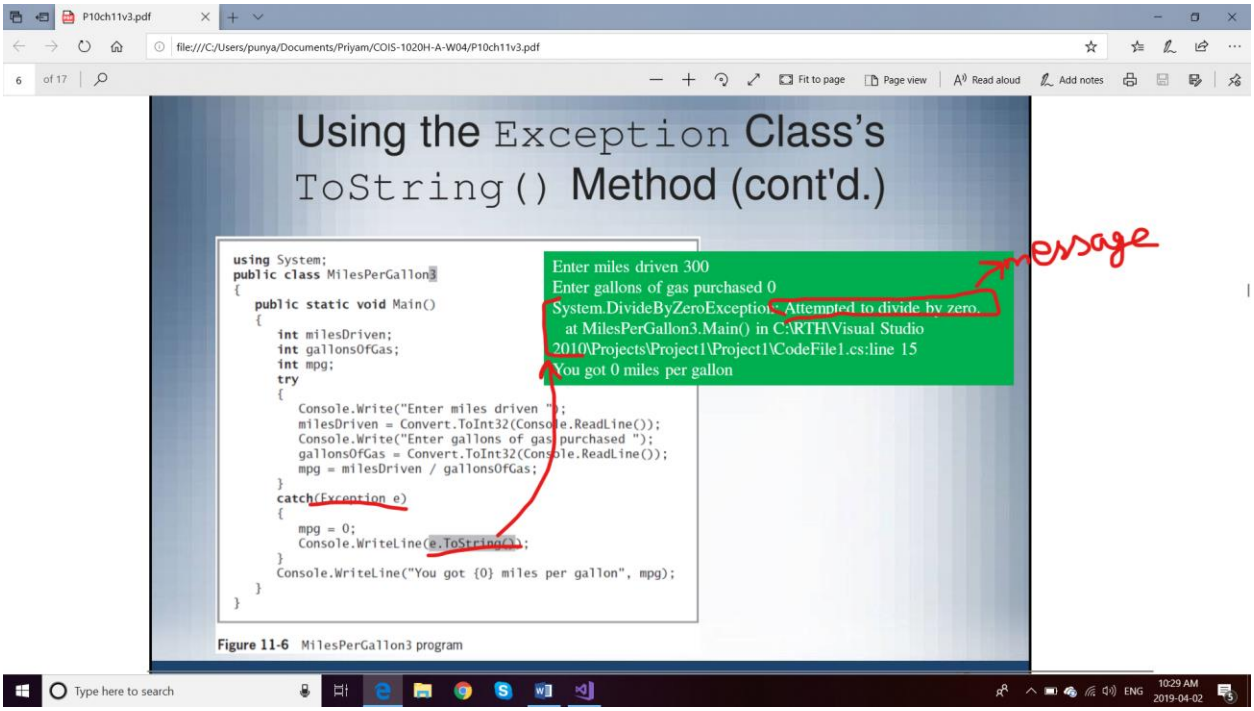
So for such an error, we enter

```
Catch(FormatException e)
{
 Console.WriteLine("You did not enter an integer");
}  //this one is for such a string instead of integer error.
```

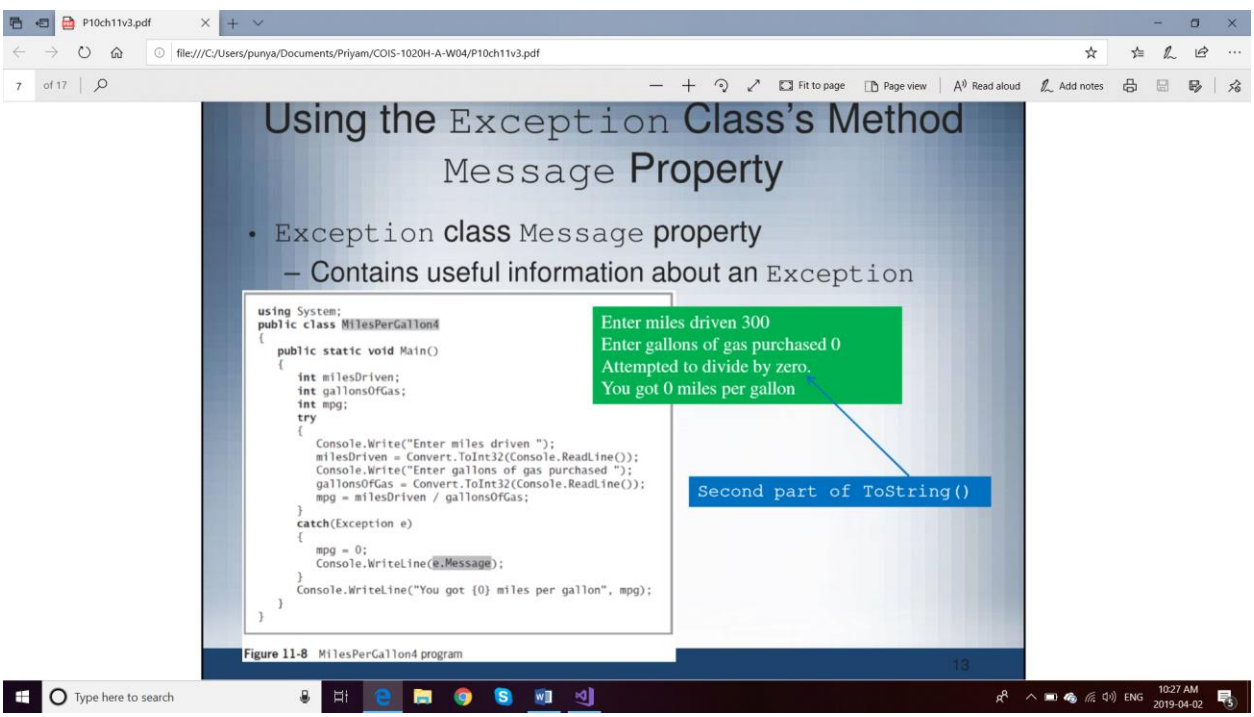
Now let's say, if we keep only FormatException, and enter a 0, then it will show error because, dividing by zero is not a format exception. So, :

```
Catch(DivideByZeroException bob)
{
 Console.WriteLine("You tried dividing by zero");
```

}



We don't always need to write what message to be printed. If we just write .ToString() then it will print whatever is stored inside.



Just like the ToString() Method, we have a Message property. Which just extracts the important part : for instance, here, attempt to divide by zero.

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
```

No “Boom!” even if we enter “cat”. Ta-daaa!!

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
```

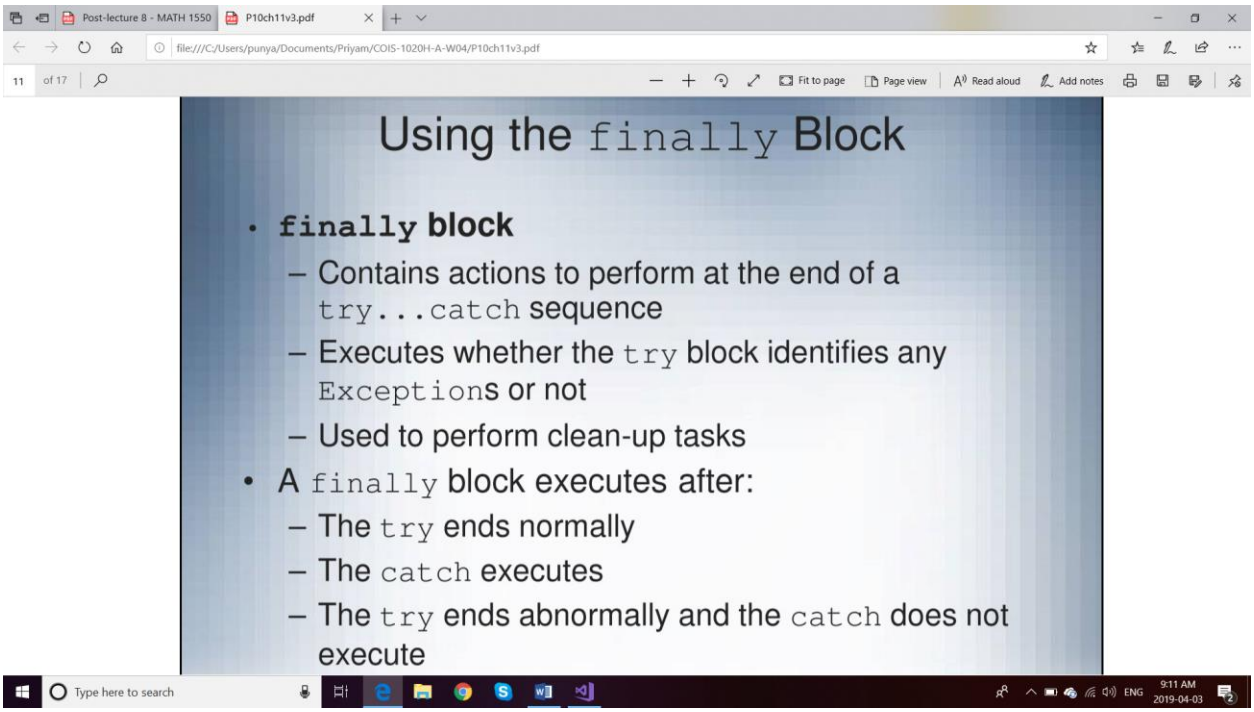
3 exceptions in the above example :

1) FormatException

2)Application Exception

3) overloaded value

It’s good to have catch at the end input so it goes through the entire program at least.



Runs every time we run through the try and catch block.

➔hard disk speed is in nanoseconds and database is milliseconds. Hard disk I a million times slower and its not a good usage of CPU to make it wait for million cycles. Thus, the data is first store in main memory database then copied to hard disk for sustaining info.

- ⇨ Basically, finally block is to close a file or run a last buffer.
- ⇨ There is only one finally block which is optional.

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

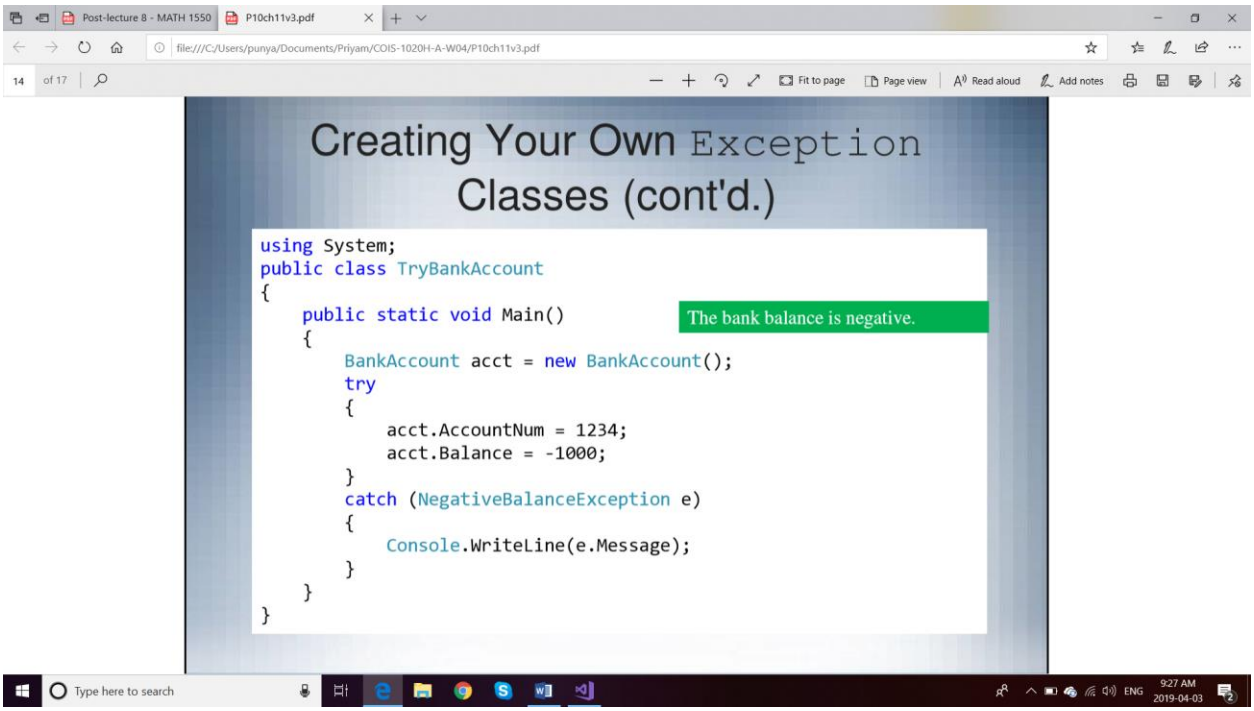
Since all exceptions are child classes of a parent class Exception, so we can create our own exception class.

We need to create an object of the class and throw the exception to where it is called from, basically catch in Main Program :

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



We could just : throw(new NegativeBalanceException()); instead of creating the new object in a different line.