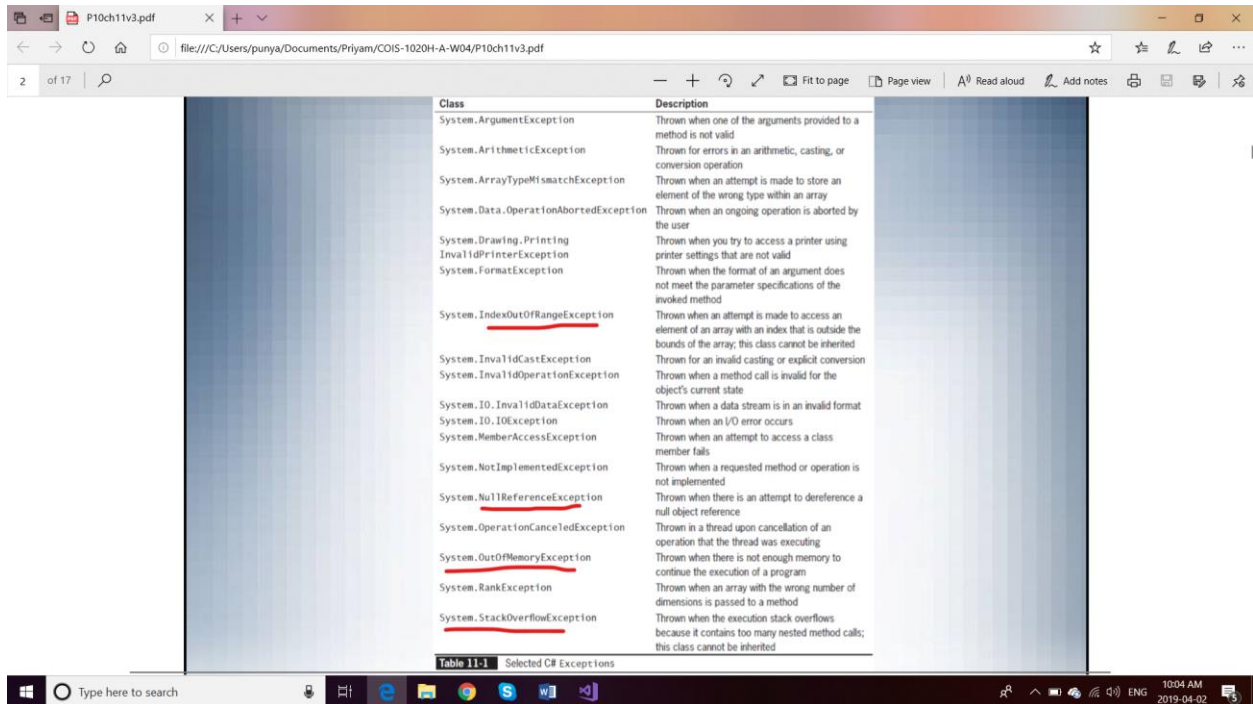


# EXCEPTION HANDLING

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



The screenshot shows a PDF viewer window with a table titled "Table 11-1 Selected C# Exceptions". The table lists various C# exception classes and their descriptions. The classes listed are: System.ArgumentException, System.ArithmeticException, System.ArrayTypeMismatchException, System.Data.OperationAbortedException, System.Drawing.Printing.InvalidPrinterException, System.FormatException, System.IndexOutOfRangeException, System.InvalidCastException, System.InvalidOperationException, System.IO.InvalidDataException, System.IO.IOException, System.MemberAccessException, System.NotImplementedException, System.NullReferenceException, System.OperationCanceledException, System.OutOfMemoryException, System.RankException, and System.StackOverflowException. The descriptions for each class are provided in the second column. The table is titled "Table 11-1 Selected C# Exceptions".

| 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                            |

//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

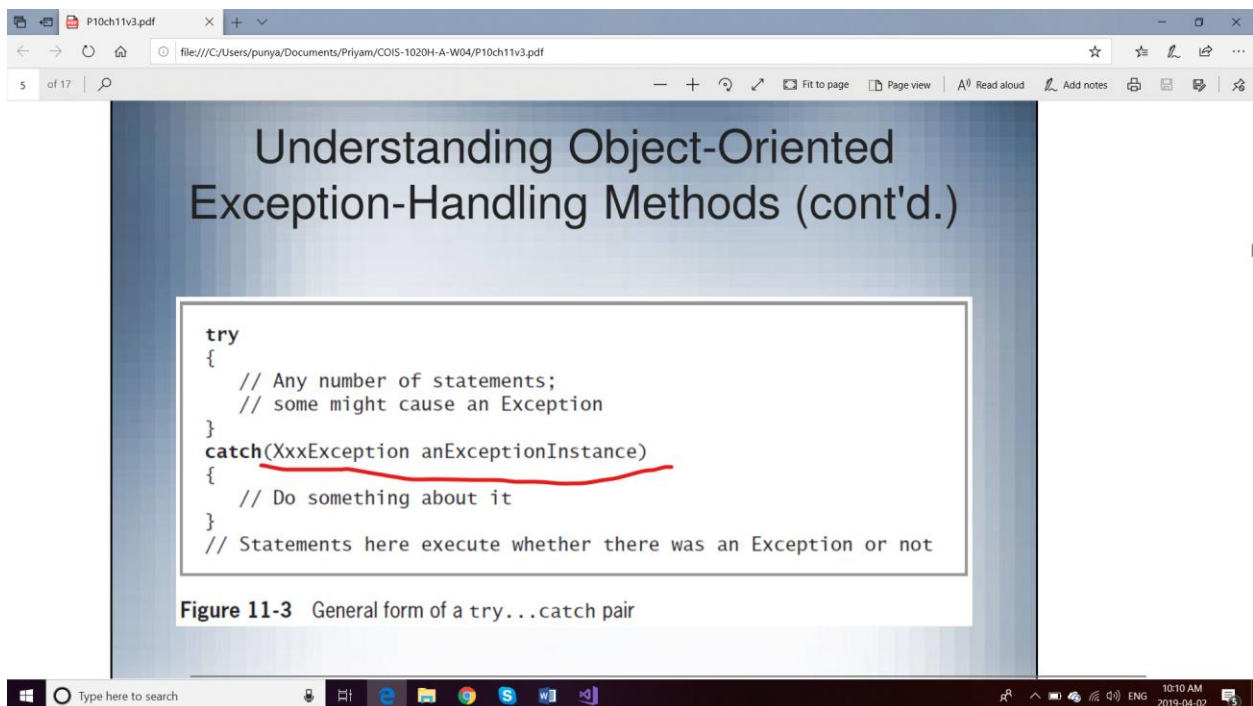
- try block
  - Contains statements that can produce an error

- Code at least one catch block or finally block
- Immediately following a try block

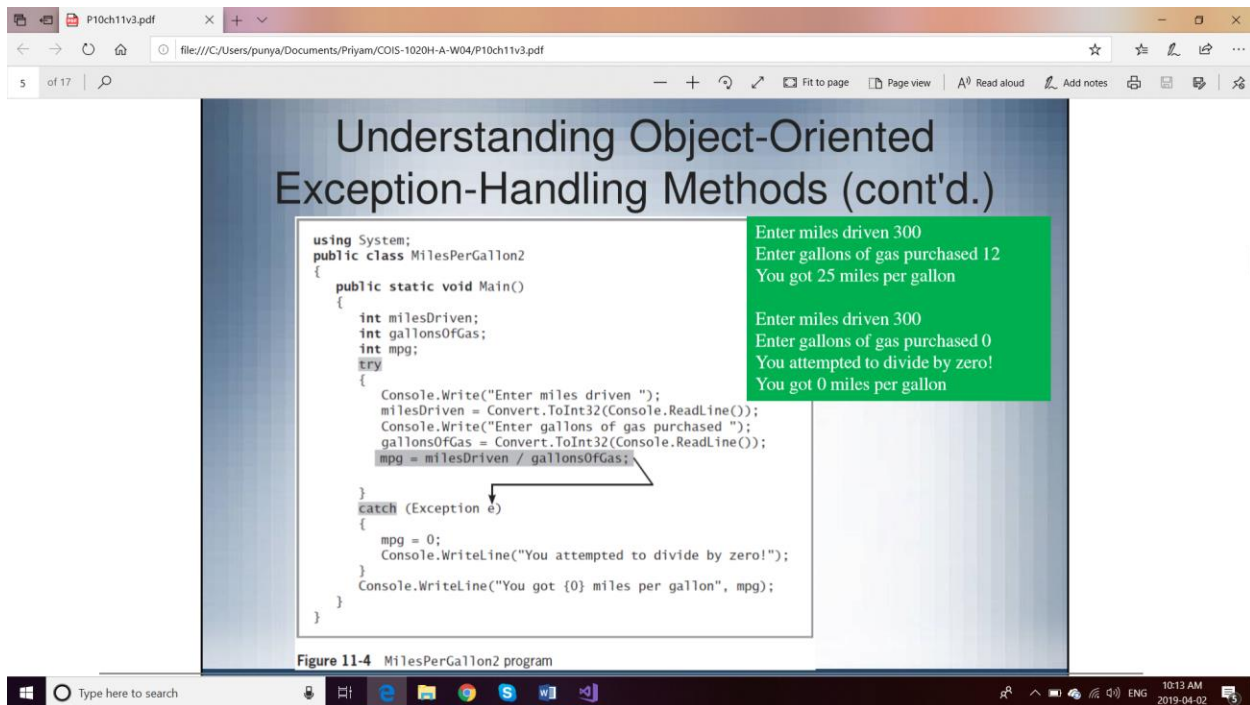
- catch block
- Can “catch” one type of Exception

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");`

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.WriteLine("Enter miles driven ");
            milesDriven = Convert.ToInt32(Console.ReadLine());
            Console.WriteLine("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:\ARTH\Visual Studio 2010\Projects\Project1\Project1\CodeFile1.cs:line 15  
You got 0 miles per gallon

message

Figure 11-6 MilesPerGallon3 program

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.

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.WriteLine("Enter miles driven ");
            milesDriven = Convert.ToInt32(Console.ReadLine());
            Console.WriteLine("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

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.WriteLine("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.WriteLine("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

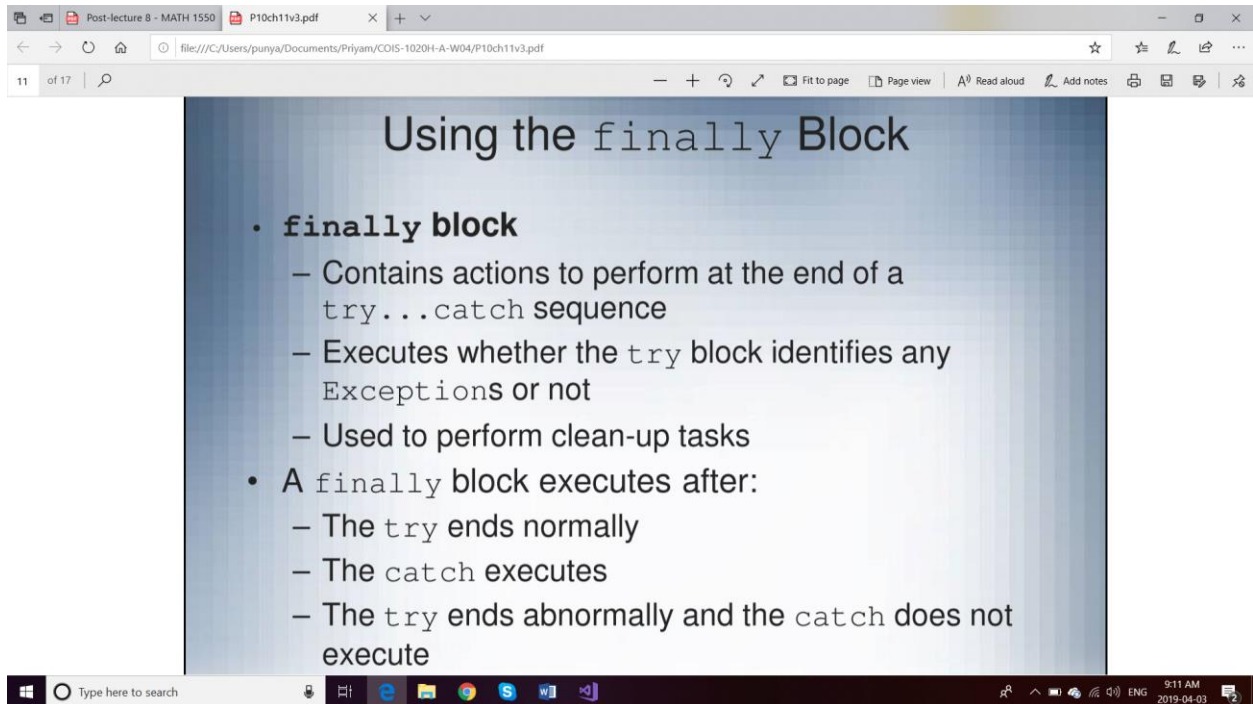
3 exceptions in the above example :

- 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 is a million times slower and it's not a good usage of CPU to make it wait for million cycles. Thus, the data is first stored 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.

The screenshot shows a presentation slide titled "Creating Your Own Exception Classes". The slide content includes a bullet point: "To create your own Exception – Extend the ApplicationException or the Exception class". Below this, a code block shows the implementation of the `NegativeBalanceException` class, which inherits from `Exception`. The code is as follows:

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

Below the code block, the slide is captioned "Figure 11-28 The NegativeBalanceException class". The presentation window's taskbar at the bottom shows the Windows search bar and various application icons.

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 :

The screenshot shows a presentation slide titled "Creating Your Own Exception Classes (cont'd.)". The slide content includes a code block showing the implementation of the `BankAccount` class. The code is as follows:

```
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;
        }
    }
}
```

Below the code block, the slide is captioned "Figure 11-29 The BankAccount class". The presentation window's taskbar at the bottom shows the Windows search bar and various application icons.

The screenshot shows a presentation slide titled "Creating Your Own Exception Classes (cont'd.)". The slide contains C# code for a `TryBankAccount` class with a `Main` method. The code uses a `try-catch` block to handle a `NegativeBalanceException`. A green message box displays the exception message: "The bank balance is negative." The slide is viewed in a PDF reader, with the file path `file:///C:/Users/punya/Documents/Priyam/COIS-1020H-A-W04/P10ch11v3.pdf` visible in the address bar. The Windows taskbar at the bottom shows the time as 9:27 AM on 2019-04-03.

## 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.

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