Programming in C# II

Error-handling using structured exceptions

.NET

Contents:

- Types of errors
- Minimizing errors by using control statements and the Debug class
- Exceptions, Throwing and catching exceptions
- Custom exceptions

Farid Naisan, farid.naisan@mah.se

Types of errors

- Three main types of errors:
  - Compile-time errors,
  - Run-time errors
  - Logical errors
- Compilation errors occur when the syntax of a programming language is not followed.
- Run-time errors occur when the program is executing and some invalid operation is performed.
- Logical errors are errors in the program causing wrong output.

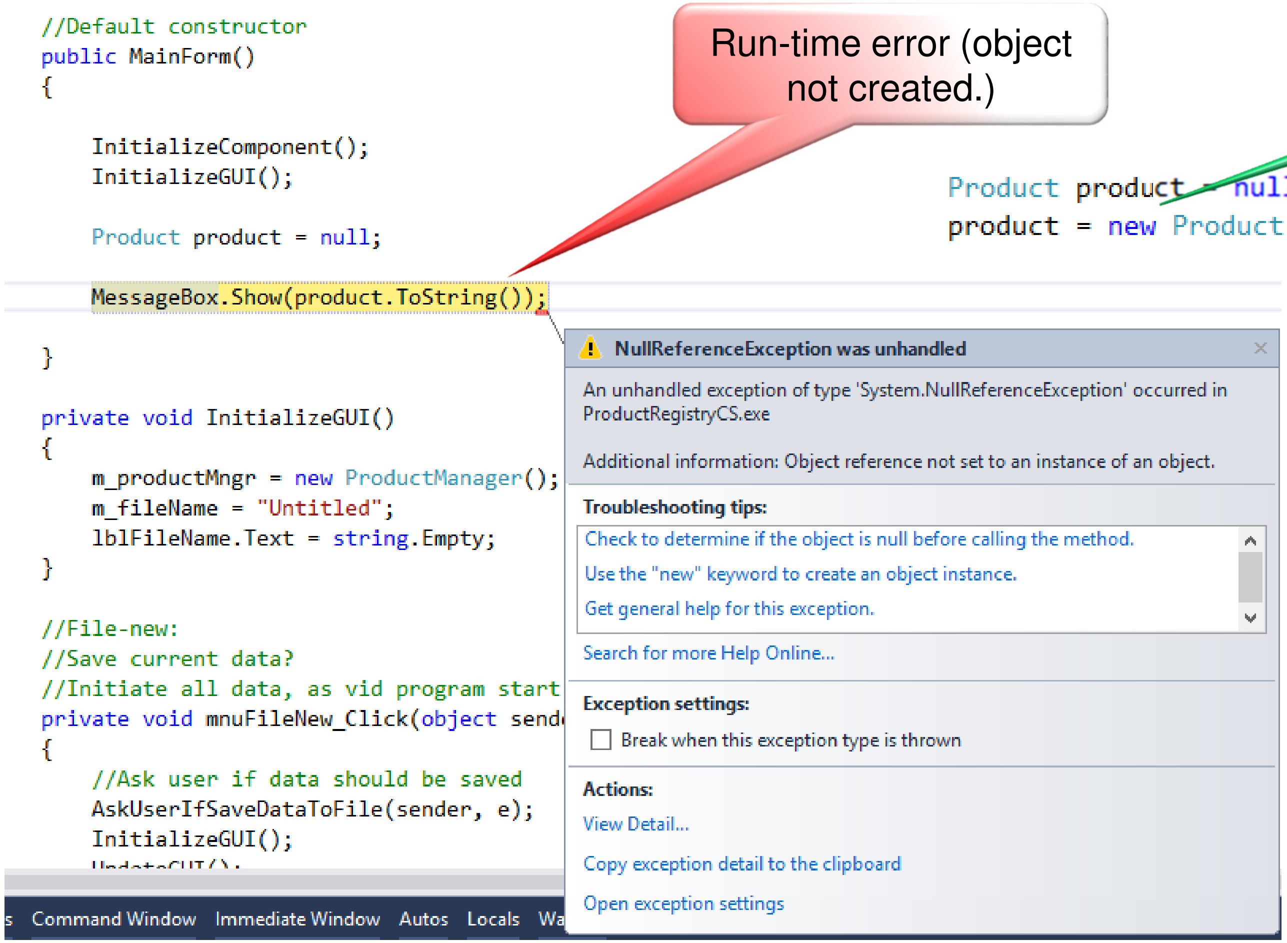
Compilation Error

- The method below fail to compile because of a type error.

```
void Test()
{
    int intNumber = 0;
    decimal amount = 0.0m;
    intNumber = amount / 25.0; //Narrowing not allowed
}
```

Run-time errors

- Valid code resulting in invalid conditions



```
//Default constructor
public MainForm()
{

    InitializeComponent();
    InitializeGUI();

    Product product = null;

    MessageBox.Show(product.ToString());

}

private void InitializeGUI()
{
    m_productMngr = new ProductManager();
    m_fileName = "Untitled";
    lblFileName.Text = string.Empty;
}

//File-new:
//Save current data?
//Initiate all data, as vid program start
private void mnuFileNew_Click(object send
{
    //Ask user if data should be saved
    AskUserIfSaveDataToFile(sender, e);
    InitializeGUI();
    UpdateGUI();
```

NullReferenceException was unhandled

An unhandled exception of type 'System.NullReferenceException' occurred in ProductRegistryCS.exe

Additional information: Object reference not set to an instance of an object.

Troubleshooting tips:

Check to determine if the object is null before calling the method.

Use the "new" keyword to create an object instance.

Get general help for this exception.

Search for more Help Online...

Exception settings:

□ Break when this exception type is thrown

Actions:

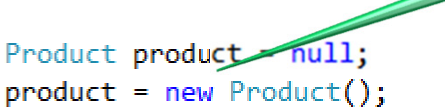
View Detail...

Copy exception detail to the clipboard

Open exception settings

Run-time error (object not created.)

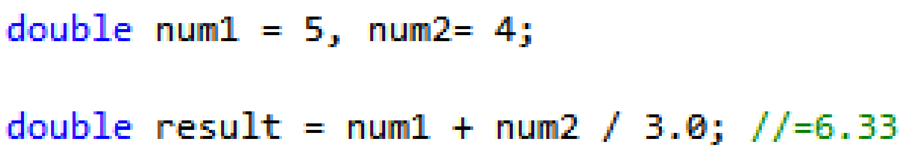
Bug-fix



```
Product product = null;
product = new Product();
```

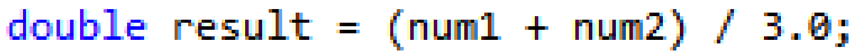
Logic Errors

- Logical errors can be bugs in the code that are errors by the programmer.



```
double num1 = 5, num2= 4;

double result = num1 + num2 / 3.0; //=6.33
```

Let me test with 5 and 4. Why 6.33, it should have given a value 3.0. I get it - lacking parenthesis!



```
double result = (num1 + num2) / 3.0;
```

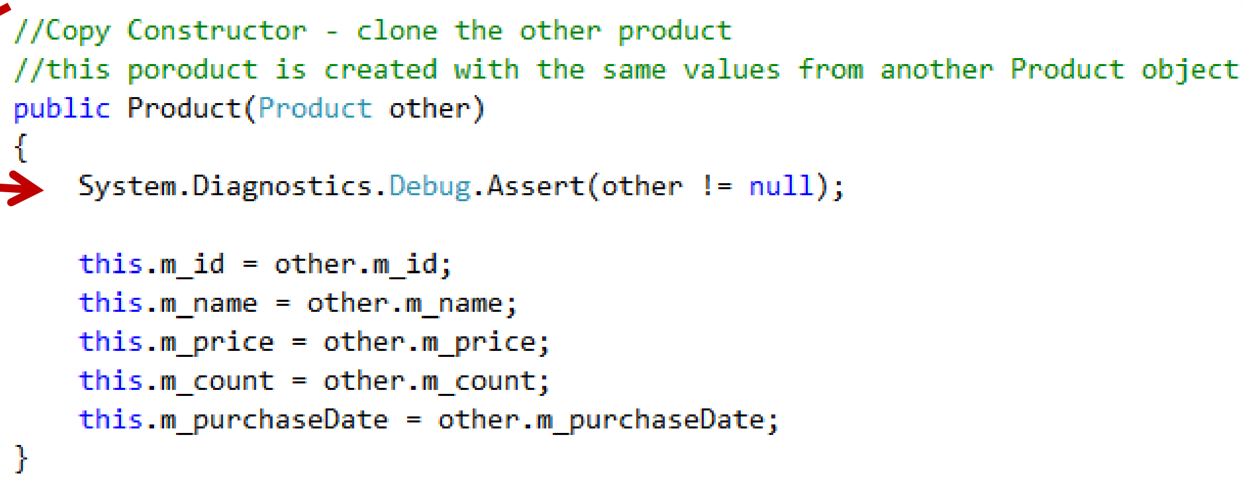
- These types of errors can also be mistakes in specifications given to a programmer.
- To find and fix such problems, the application must be tested and compared with actual calculations.

Error handling

1. Errors can be handled by providing sufficient controls in the source code, wherever there can a risk for wrong results and unwanted behavior due to run-time conditions, using the basic control structures, for example if-statements.

2. Use the Debug.Assert method to place watchdogs.

Import System·Diagnostics

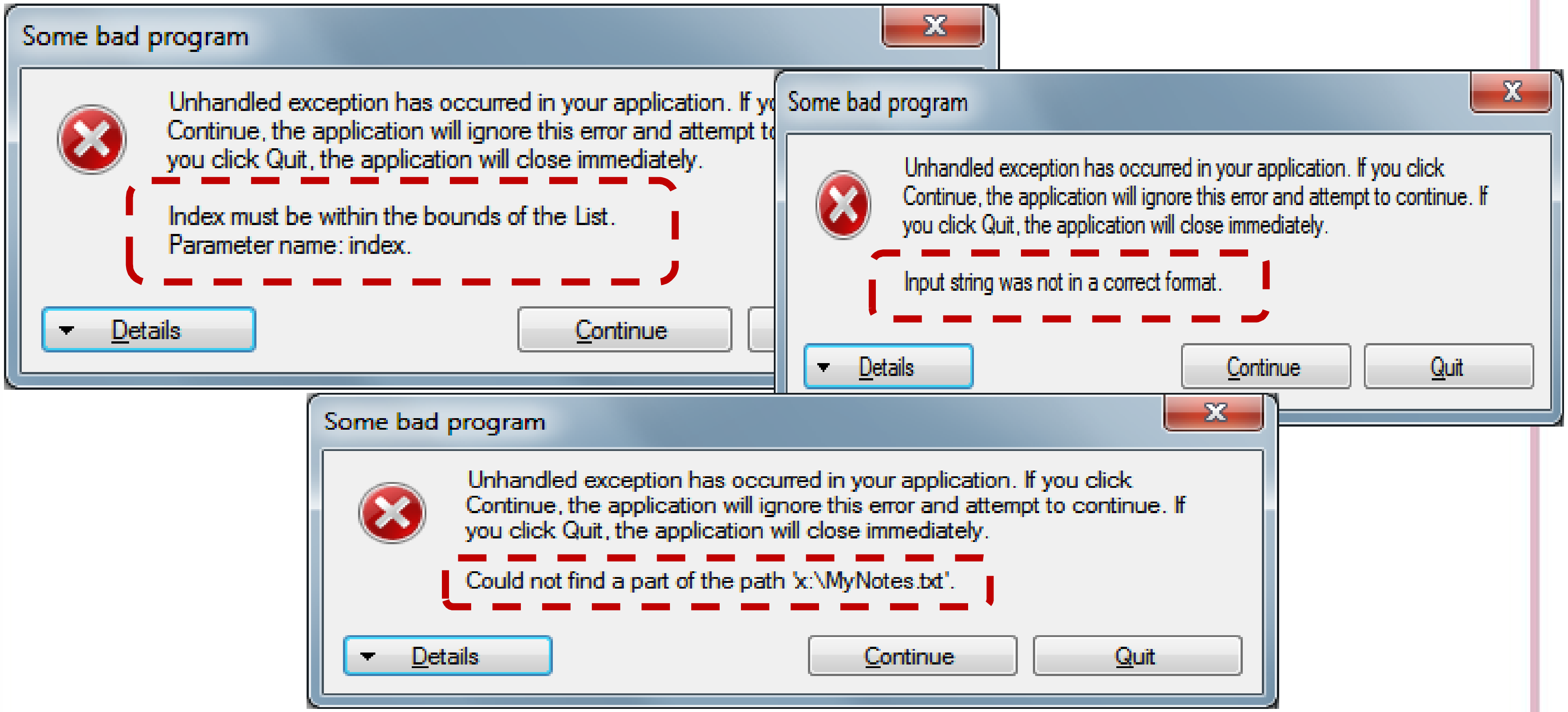


```
//Copy Constructor - clone the other product
//this poroduct is created with the same values from another Product object
public Product(Product other)
{
    System.Diagnostics.Debug.Assert(other != null);

    this.m_id = other.m_id;
    this.m_name = other.m_name;
    this.m_price = other.m_price;
    this.m_count = other.m_count;
    this.m_purchaseDate = other.m_purchaseDate;
}
```

3. Handle Exceptions.

Exceptions

- It is assumed that code executes normally, but some times exceptions may occur.



Some bad program

Unhandled exception has occurred in your application. If yo
Continue, the application will ignore this error and attempt to
you click Quit, the application will close immediately.

Index must be within the bounds of the List.
Parameter name: index.

Some bad program

Unhandled exception has occured in your application. If you click Continue, the application will ignore this error and attempt to continue. If you click Quit, the application will close immediately.

Input string was not in a correct format.

Some bad program

Unhandled exception has occurred in your application. If you click Continue, the application will ignore this error and attempt to continue. If you click Quit, the application will close immediately.

Could not find a part of the path 'x:\MyNotes.txt'.

- If you don't trap these exceptions, the CLR takes care of them as shown here, giving users some options.

What are Exceptions

- Exceptions are typically regarded as runtime errors that are difficult to account for while programming an application.
- Examples are when opening a file does not exist or file is incompatible, or trying to calculate a division by zero.
- The programmer (or even the end user) has little control over these “exceptional” circumstances.
- .NET structured exception handling is a technique for dealing with runtime exceptions.
- When a run-time error occurs, CLR generates a corresponding Exception that identifies the problem.
- The .NET base class libraries define numerous exceptions, such as **FormatException, IndexOutOfRangeException, FileNotFoundException, ArgumentOutOfRangeException, etc.**

.NET Exception Handling

- Programming with structured exception handling involves the use of four interrelated entities:
  - A class type that represents the details of the exception.
  - A member that *throws* an instance of the exception class to the caller under the correct circumstances.
  - A block of code on the caller’s side that invokes the exception-prone member.
  - A block of code on the caller’s side that will process (or catch) the exception when it occurs.

C# Keywords

- C# offers four keywords that allow throwing and handling exceptions.
  - try, catch, throw, finally.
- The object that represents the problem at hand is a class extending **System.Exception** (or a descendent thereof).

```
try
{
  // This part of code might result in
  // an Exception thrown and the step-
  // by-step execution might stop
}
catch(Exception ex)
{
  // The execution jumps here only if
  // an Exception is thrown
  // in the try block above here.
  // Corrective actions can/should be
  // taken here.
}
finally
{
  // The finally block is executed no
  // matter an exception is thrown or
  // not. The code here will always run.
}
```

More than one catch can be used. Finally is optional.

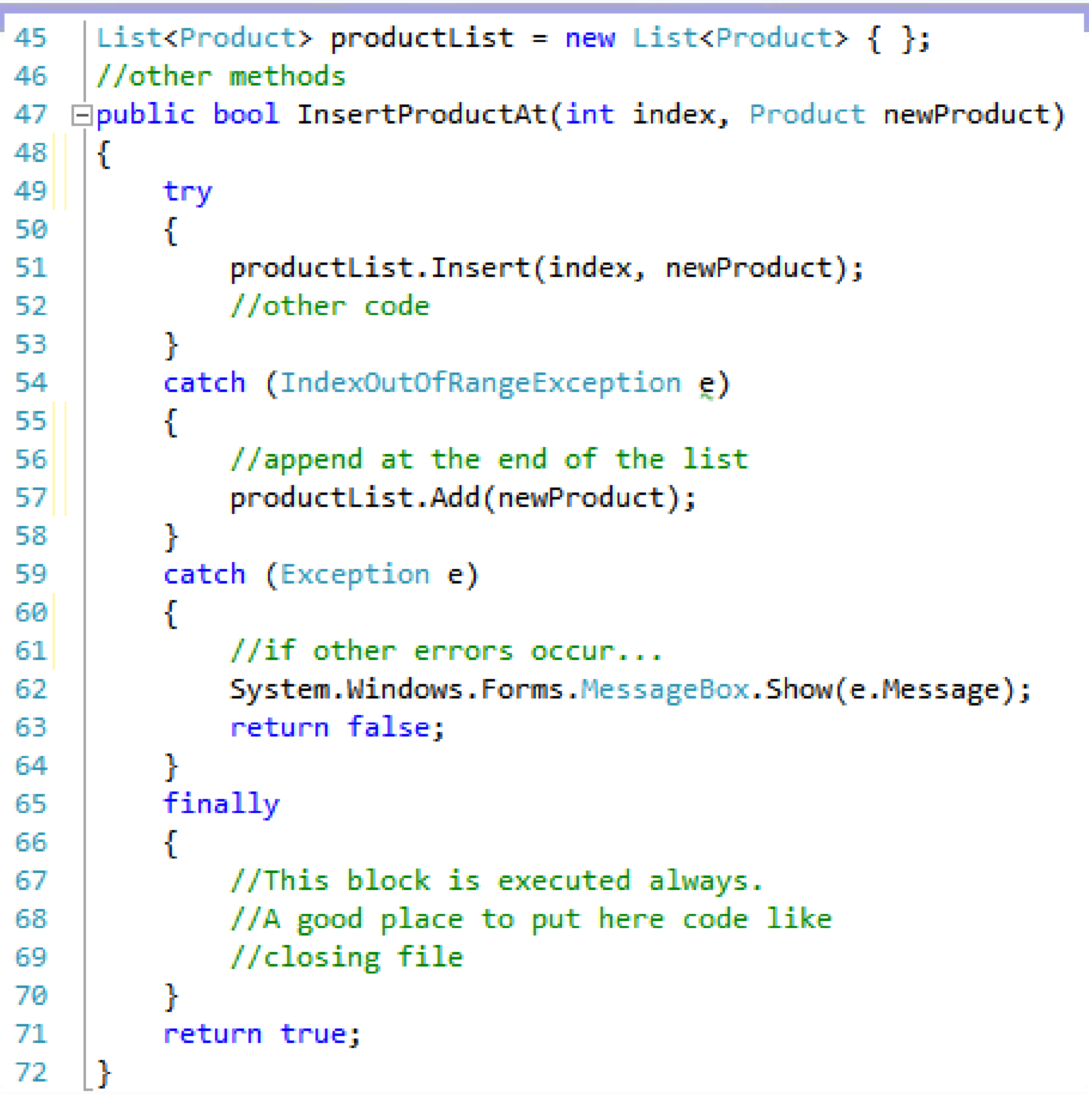
Throw and catch an exception

- An Exception in computer programming is an object that can be created when an exceptional event occurs
- The object is then filled with information about the event and then “thrown” from the situation to some other place in the code where the Exception is “caught”.
- When the Exception is caught, the information about the exceptional event can be used to remedy the situation .
- And in this way, the Exception that occurred is “handled".
- Unhandled errors may cause:
  - Data corruption
  - Application crash
  - Other consequences (blocking resources, etc)

Try

- A try block is a section of statements that may throw an exception during execution. If an exception is detected, the flow of program execution is sent to the appropriate catch block.
  - A try block may be followed by several catch-blocks.
  - CLR sends the program execution tracing back to the caller.
  - If the exception is not handled, program terminates abnormally.
- If the code within a try block does not trigger an exception, the catch block is skipped entirely and the execution continues normally..

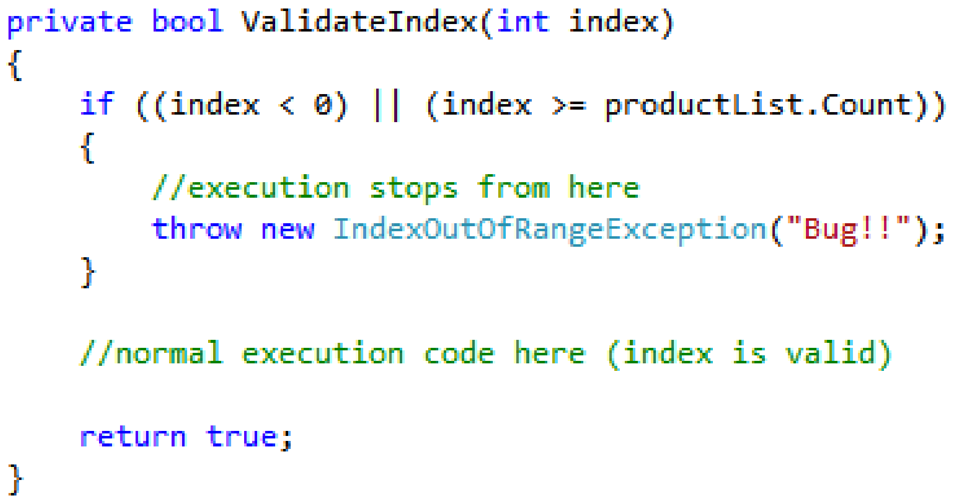
A try-catch example using C#



```
List<Product> productList = new List<Product> { };
//other methods
public bool InsertProductAt(int index, Product newProduct)
{
    try
    {
        productList.Insert(index, newProduct);
        //other code
    }
    catch (IndexOutOfRangeException e)
    {
        //append at the end of the list
        productList.Add(newProduct);
    }
    catch (Exception e)
    {
        //if other errors occur...
        System.Windows.Forms.MessageBox.Show(e.Message);
        return false;
    }
    finally
    {
        //This block is executed always.
        //A good place to put here code like
        //closing file
    }
    return true;
}
```

Throwing an exception

- In addition to catching errors, you can raise errors in your code, i.e. you can throw errors.
- An exception can be thrown simply by using the throw statement.
- In the example here a standard exception is thrown.
- Custom exceptions can be thrown in the same way.
- If you add throw in a try block, the execution jumps directly to the catch block.



```
private bool ValidateIndex(int index)
{
    if ((index < 0) || (index >= productList.Count))
    {
        //execution stops from here
        throw new IndexOutOfRangeException("Bug!!");
    }

    //normal execution code here (index is valid)

    return true;
}
```

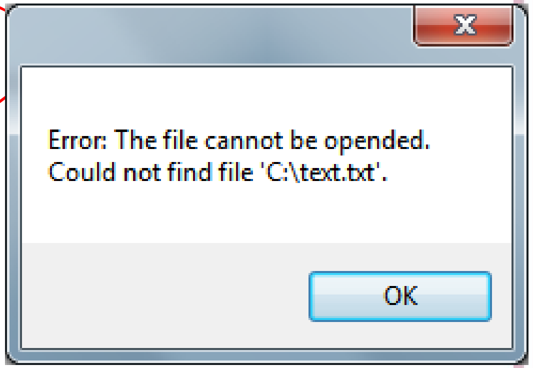
Throwing your own exception

- Using the keyword throw, you send back en error to the caller code with a new message.
- You can also send the original error.

```
public void OpenFileForInput(string fileName)
{
    FileStream reader = null;

    try
    {
        reader = File.Open(fileName, FileMode.Open);
        //do other stuff
    }
    catch (Exception ex)
    {
        //throw your own exception with new text, and the original ex
        throw new FileNotFoundException("The file cannot be opended.", ex);
    }
    finally
    {
        if (reader != null)
           reader.Close();
    }
}
```

Using System.IO

```
private void button1_Click(object sender, EventArgs e)
{
    try
    {
        OpenFileForInput("C:\\text.txt");
    }
    catch (FileNotFoundException ex)
    {
        MessageBox.Show("Error: " + ex.Message + Environment.NewLine + ex.InnerException.Message);
    }
}
```



Error: The file cannot be opended.
Could not find file 'C:\text.txt'.

The using statement

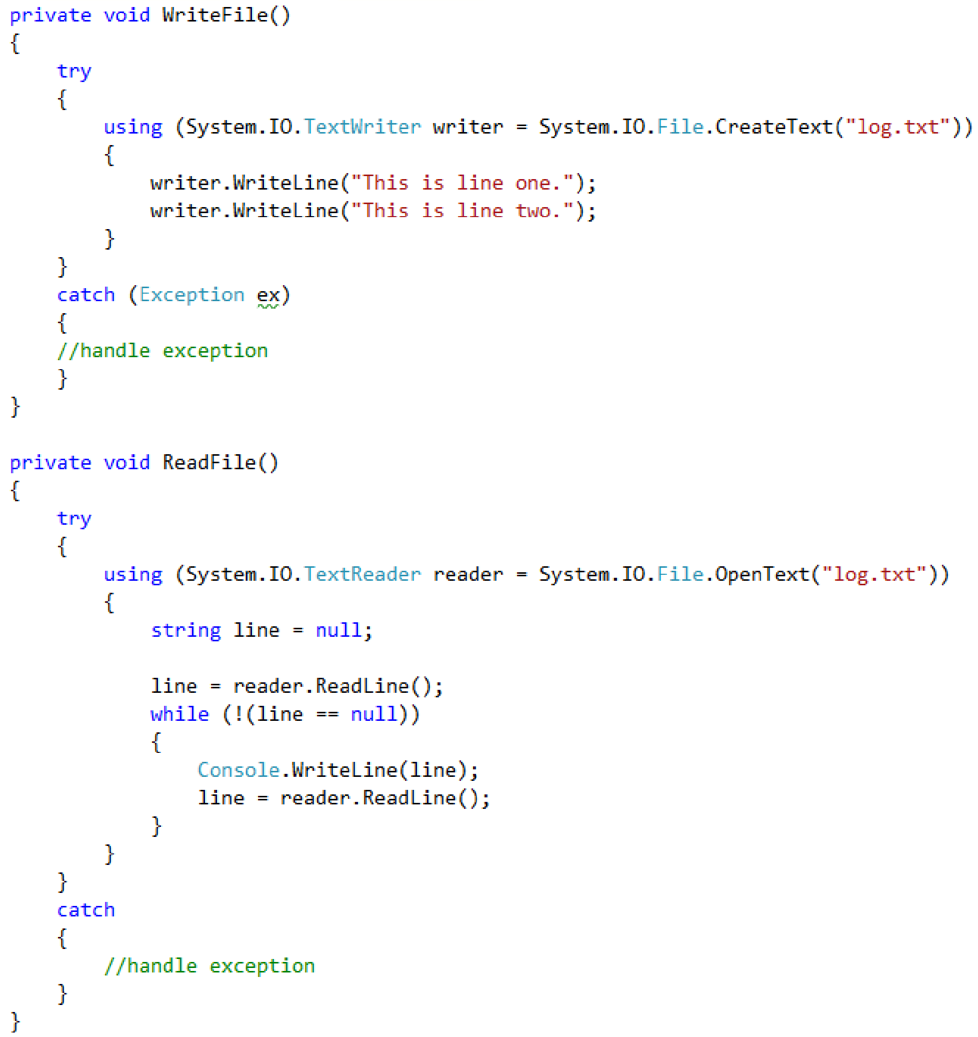
- The finally block should typically contain code for releasing resources, e.g. closing files, etc. to ensure that resources are released regardless of whether an exception was thrown or not.
- The resource must be an object that implements the IDisposable interface.
- An alternative way to automatically release resources is to use the using statement. This using statement is specially practical when working with files and graphical objects.

```
using (Font font1 = new Font("Arial", 16.0f), font2 = new Font("Courier New",
                                                                          12.0f))
{
   // Other code

}
```

- The using statement obtains the resource specified, executes the statements and finally calls the Dispose method of the object to clean up the object.

Using statement with try-catch

- Try-catch can be combined with the using statement if you want to handle an exception.



```
private void WriteFile()
{
    try
    {
        using (System.IO.TextWriter writer = System.IO.File.CreateText("log.txt"))
        {
            writer.WriteLine("This is line one.");
            writer.WriteLine("This is line two.");
        }
    }
    catch (Exception ex)
    {
    //handle exception
    }
}

private void ReadFile()
{
    try
    {
        using (System.IO.TextReader reader = System.IO.File.OpenText("log.txt"))
        {
            string line = null;

            line = reader.ReadLine();
            while (!(line == null))
            {
                Console.WriteLine(line);
                line = reader.ReadLine();
            }
        }
    }
    catch
    {
        //handle exception
    }
}
```

Custom Exceptions

- The System.Exception and its derived classes, defined in the .NET Framework, are very useful classes and can be used in most cases.
- However for some situation, you might want to use more specific exception types related closely to a problem.
- You can provide proper information in such cases by creating a user-defined exception type.
- All exception must derive from the **System.Exception**. Custom exceptions are however recommended to inherit the **ApplicationException** class which in turn inherits the class Exception.
- You can then throw the exception in your code based on some condition.
- The user (caller) of the method can then catch the Exception and handle the situation.

Custom Exception – Best practice

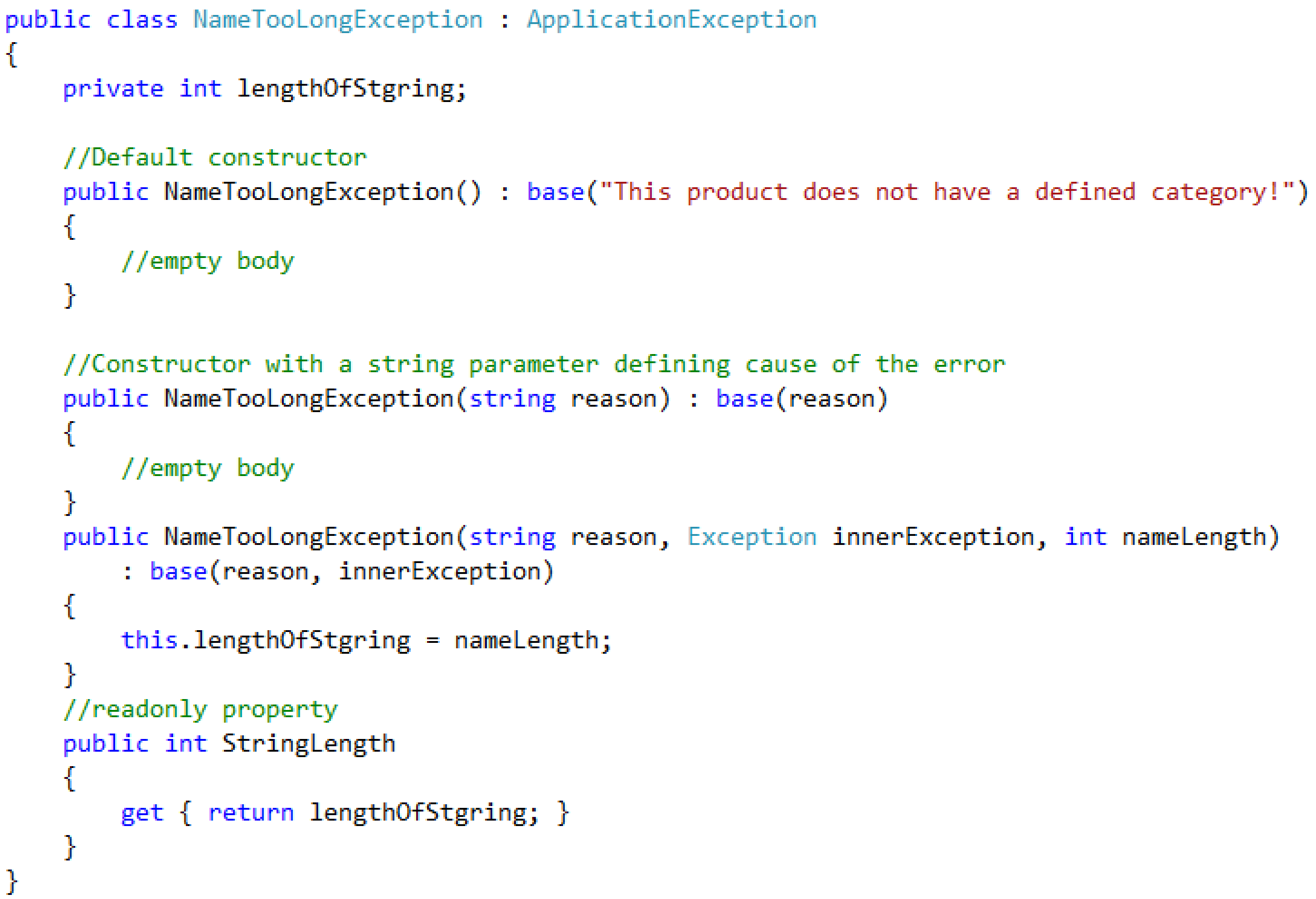
- Your custom exception should have the word Exception as a suffix in the name.
- Provide three constructors:
  - A default constructor
  - A constructor with a string parameter for information about the exception.
  - A constructor that takes a string parameter and an inner exception parameter to get at original exception info.
  - You can define extra parameters for passing more information.

Custom Exception Example

- A custom exception should preferably be derived from the **ApplicationException** which in turn is derived from the class Exception.
- **ApplicationException** is thrown by the user code not by the .NET runtime (CLR).
- Example:
  - Assume we have a class product that has a field name.
  - For some reason, the name should not be longer than 50 chars. If it does, an exception is to be thrown.

Custom exception example – create an exception class

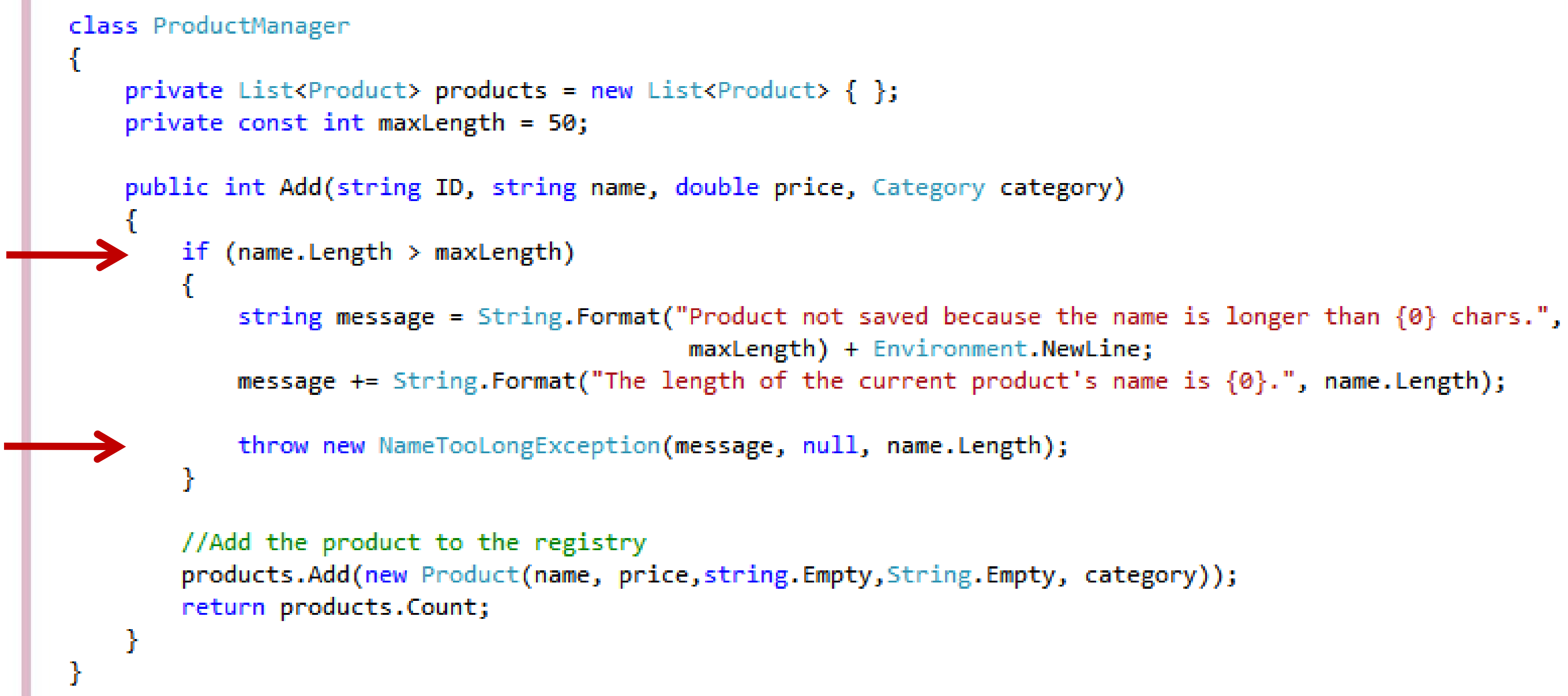
- A custom exception class with three constructors.



```
public class NameTooLongException : ApplicationException
{
    private int lengthOfStgring;

    //Default constructor
    public NameTooLongException() : base("This product does not have a defined category!")
    {
        //empty body
    }

    //Constructor with a string parameter defining cause of the error
    public NameTooLongException(string reason) : base(reason)
    {
        //empty body
    }
    public NameTooLongException(string reason, Exception innerException, int nameLength)
        : base(reason, innerException)
    {
        this.lengthOfStgring = nameLength;
    }
    //readonly property
    public int StringLength
    {
        get { return lengthOfStgring; }
    }
}
```

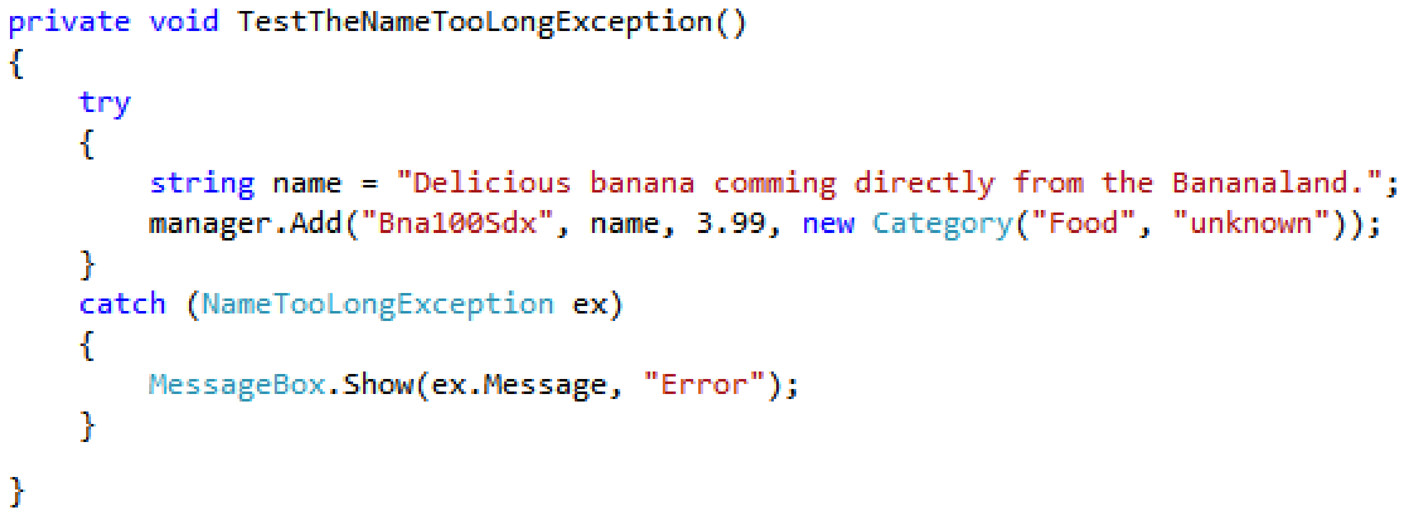
Throw an object of the custom exception

- Throw the NameTooLongException based on a condition.

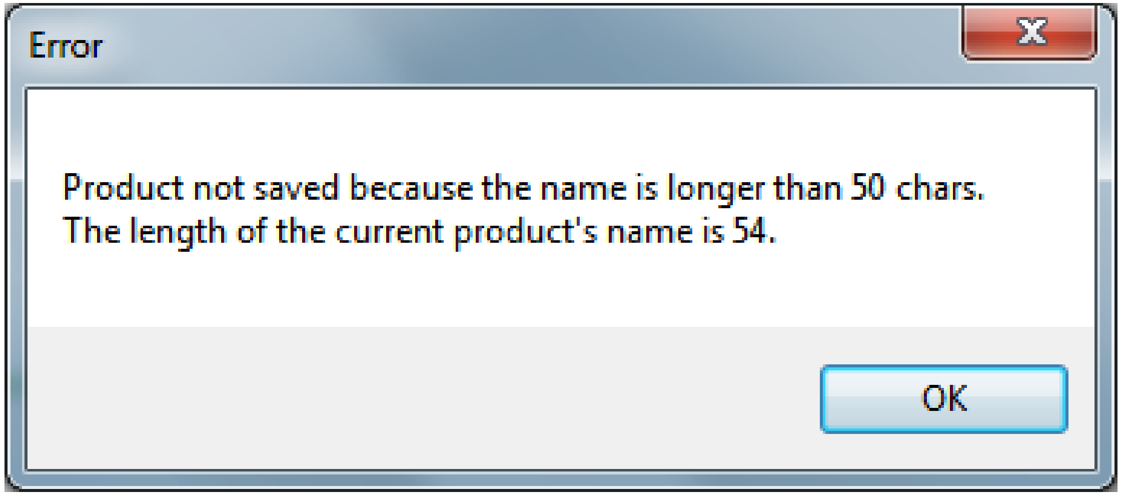


```
class ProductManager
{
    private List<Product> products = new List<Product> { };
    private const int maxLength = 50;

    public int Add(string ID, string name, double price, Category category)
    {
        if (name.Length > maxLength)
        {
            string message = String.Format("Product not saved because the name is longer than {0} chars.",
                                            maxLength) + Environment.NewLine;
            message += String.Format("The length of the current product's name is {0}.", name.Length);

            throw new NameTooLongException(message, null, name.Length);
        }

        //Add the product to the registry
        products.Add(new Product(name, price,string.Empty,String.Empty, category));
        return products.Count;
    }
}
```

Test the custom exception



```
private void TestTheNameTooLongException()
{
    try
    {
        string name = "Delicious banana comming directly from the Bananaland.";
        manager.Add("Bna100Sdx", name, 3.99, new Category("Food", "unknown"));
    }
    catch (NameTooLongException ex)
    {
        MessageBox.Show(ex.Message, "Error");
    }

}
```



Product not saved because the name is longer than 50 chars.
The length of the current product's name is 54.

Call Stack

- A call stack is a list of methods that has been executed by CLR to get to the current position in the program code.
- When debugging, this information is usually available and it can be traced back to the very first method which the method Main.
- When an exception is thrown, the normal step by step execution of the program is stopped.
- The Exception is propagated back (thrown) on the call stack until it reaches a catch statement.
- CLR goes through the stack and if it does not find an exception handler registered for the exception, the unhandled-exception for the current application domain is fired and the program could be terminated.

VS's Debugger

- Visual Studio has a very advance but easy to use Debugger.
- Some quick keys to remember:
  - F9 Toggle a breakpoint
  - F5 Start a debug session
  - F10 Step over
  - F11 Step into
  - Shift+F5: Cancel/Stop debug session
  - Ctrl+F5 : Run application without debugging

Some guidelines

- Where ever in your code, you suspect a source of error at run time to user input or otherwise, provide code to handle the situation to prevent unwanted behavior at runtime.
  - Correct the error, or simply continue execution in a save way.
- Do not use exceptions to control the flow of code execution.
- Do not use exception where you can handle errors using ordinary control statements such if else.
- .NET defines a large number of exceptions, but the parent to all exceptions is the class Exception.
- Try to use the correct exception type but if you are unsure about the type of error you get, use an object of the general Exception class.
- A finally block is a good place for clean up code as it is always executed no matter an error occurs or not.

Summary

- Software errors and bugs cost money, but they exist
- Every good programmer takes cares of all the situations that can cause errors.
- Use simple ways using if-statements, Debug object, etc. of checking for possible error.
- When simple checking is not sufficient, use structured exception handling. You have these options:
  - No error handling – CLR applies default error handling
  - Catch all types of error using objects of the general Exception class.
  - Use specific Exceptions that handles specific errors, such IndexOutOfRangeException.
  - Write your own exception class – user defined exception class

Links

- Exceptions
  - http://msdn.microsoft.com/en-us/library/5b2yeyab(v=vs.110).aspx
- Custom Exceptions:
  - http://msdn.microsoft.com/en-us/library/87cdya3t(v=vs.110).aspx