158.212 Application Software Development

Suriadi Suriadi

Lecture 7

Defensive Programming

- Input Validation
  - Prevention & Exception Handling
- Assertions

Dialog Boxes

MessageBox

OpenFile/SaveFile

Defensive Programming

- Based on the concepts of “defensive driving”
- Demands a change in the mind-set
  - from *“users will behave with appropriately”,* to …..
  - *“what can users do to break my application!”*
- Must take responsibility even when its NOT your fault

Anticipate unexpected problems in the applications and develop codes to handle those situations accordingly.

Input Validation

- "Garbage in, garbage out" is unacceptable for production systems
- Production systems should never produce garbage or crash due to bogus input
- Input Validation: requires that all data coming from external sources is adequately **validated** and **contingencies** made for erroneous input.
    - Set up **barricades** within code
    - **Validation** and s**anitization** of input

Input Validation

- There are two main methods of doing input validation:

Prevention and Exception Handling

- These methods stop an application crashing if invalid data is used in some operations.

Prevention

- Prevention is the simplest input validation technique
- Prevention stops invalid input causing the application to crash by verifying the data **before** it is used
- The implementation achieved through conditional constructs like "**if/else**" statements

Prevention

For example, a calculator application:

```
While True
   Console.WriteLine("Please enter an expression")
   Dim a, b As Integer
   Dim c As Char
   a = Console.ReadLine()
   c = Console.ReadLine()
   b = Console.ReadLine()
   If c = "+" Then
      Console.WriteLine("= " & a + b)
   ElseIf c = "-" Then
      Console.WriteLine("= " & a - b)
   ElseIf c = "*" Then
      Console.WriteLine("= " & a * b)
   ElseIf c = "/" Then
      Console.WriteLine("= " & a / b)
   End If
End While
```

Prevention

For example, a calculator application:

```
while (true) {
      Console.WriteLine("Please enter an expression");
      int a, b;
      char c;
      a = int.Parse(Console.ReadLine());
      c = char.Parse(Console.ReadLine());
      b = int.Parse(Console.ReadLine());
      if (c == '+') {
       Console.WriteLine("= " + (a + b));
      } else if (c == '-') {
       Console.WriteLine("= " + (a - b));
      } else if (c == '*') {
       Console.WriteLine("= " + a * b);
      } else if (c == '/') {
          Console.WriteLine("= " + a / b);
      }
}
```

Prevention

VB

```
Please enter an expression:
3
+
4
= 7
Please enter an expression:
10
/
5
= 2
Please enter an expression:
3
/
0
= Infinity
```

C#

```
Please enter an expression:
3
+
4
= 7
Please enter an expression:
10
/
5
= 2
Please enter an expression:
3
/
0
DivideByZeroException
```

Prevention

- **Divide-by-zero** causes crashes in C#

- The programmer must check that the second operand for division is not 0

```
} else if(c == '/') {
   if(b == 0) {
    Console.WriteLine("Cannot divide by zero!");
   } else {
    Console.WriteLine("= " + (a/b));
   }
}
```

Prevention

```
Please enter an expression:
3
/
0
Cannot divide by zero!
```

Prevention

- The code above:
  - prevents illegal operator errors
  - prevents divide by zero errors

- Further improvements can be made in the above example…

- What else can go wrong?
  - Invalid integers
  - Too large of number entered
  - Malicious codes?

Prevention

VB

```
Please enter an expression:
3
+
a
InvalidCastException
```

C#

```
Please enter an expression:
3
+
a
FormatException
```

Prevention

VB

```
Dim i As Integer
Dim s As String
s = Console.ReadLine()
while(Integer.TryParse(s,i) = false)
   Console.WriteLine("Invalid Integer")
   s = Console.ReadLine()
end while
```

C#

```
int a;
string s;
s = Console.ReadLine();
while(int.TryParse(s, out a) == false) {
   Console.WriteLine("Invalid Integer");
   s = Console.ReadLine();
}
```

Prevention

- The above code is now safe?
  - Potentially…..
- **Barricades** within the code have now been implemented.
- Prevention as an input validation strategy however has **limitations**.
- *The onus is on the programmer to foresee every possible problem that could arise in an application and write code to prevent the problem.*
- Tip: deny all, then allow specified values

Exception Handling

- Alternative is exception handling
- Rather than attempting to prevent an error from ever occurring, exception handling assumes the input is correct but has code to handle an error if it occurs
- Exception indicates something has gone wrong during execution

Exception Handling

- The term ‘exception’ also suggests that this is infrequent
- Exceptions result in application crashes
- Exception handling allows a program to continue executing
- Promotes development of fault tolerant software

Exception Handling

The anatomy of exception handling:

1. When an error occurs at runtime, an exception is **thrown**
2. Code to **catch** exceptions can be written
3. When a specific exception type is caught, it can be appropriately **handled**, uncaught exceptions break software
4. Once an exception is handled, another block of code can be executed for **finalizing** (tidying up) anything else that generally needs to be performed following exceptions. This block is optional and called regardless of errors.
    - Releases database connections, file handlers etc.

Exception Handling

Exceptions can be **thrown** by a function, subroutine or operation.

Exceptions can be caught by the application using a **try/catch** block.

```
Try
    block that may cause error
Catch variable As exception
    block to handle error
Finally
    block that always executes
End Try
```

```
try {

} catch(exception variable) {

} finally {

}
```

Exception Handling

We can use exception handling to prevent our previous calculator example code from crashing:

```
} else if (c == '/') {
    try {
        Console.WriteLine("= " + a / b);
    } catch (DivideByZeroException e) {
        Console.WriteLine("Cannot Divide by Zero!");
    }
}
```

Exception Handling

- Try/catch blocks can be used to encompass larger blocks of code
- Multiple types of exceptions can be caught by the same try/catch block
- The exception types are organized in an **hierarchical order**, from most specific to most generic

Exception Handling

VB

```
While True
  Try
    Dim a As Integer
    Dim b As Integer
    Dim c As Character
    a = Console.ReadLine()
    c = Console.ReadLine()
    b = Console.ReadLine()
    If c = "+" Then
      Console.WriteLine("= " & a+b)
    ElseIf c = "-" Then
      Console.WriteLine("= " & a-b)
    ElseIf c = "*" Then
      Console.WriteLine("= " & a*b)
    ElseIf c = "/" Then
      Console.WriteLine("= " & a/b)
    End If
```

C#

```
while(true) {
  try {
    int a;
    int b;
    char c;
    a = int.Parse(Console.ReadLine())
    c = char.Parse(Console.ReadLine())
    b = int.Parse(Console.ReadLine())
    if(c == '+') {
      Console.WriteLine("= " & a+b)
    } else if(c == '-') {
      Console.WriteLine("= " & a-b)
    } else if(c == '*') {
      Console.WriteLine("= " & a*b)
    } else if(c == '/') {
      Console.WriteLine("= " & a/b)
    }
```

Exception Handling

VB

```
  Catch e As InvalidCastException
    Console.WriteLine("Invalid Integer!")
  Catch e As Exception
    Console.Writeline("Error: " + e.Message)
  End Try
End While
```

C#

```
  } catch(DivideByZeroException e) {
    Console.WriteLine("Cannot divide by zero!")
  } catch(FormatException e) {
    Console.WriteLine("Invalid Integer!")
  } catch(Exception e) {
    Console.Writeline("Error: " + e.Message)
  }
}
```

Exception Handling

- The class Exception catches **any** abnormal condition
- The class Exception is useful for catching **unpredicted** errors
- Useful mechanisms
  - One can throw exceptions from procedures and allow the calling code to handle the exception
  - Can query the exception object to find out where the error occurred

Exception Handling

- Remember that the ***offending statement*** is not executed

- **Important**: place all clean-up code in the **Finally** clause in order to prevent memory leaks

Using Controls

- **Some limited** input validation can also be performed automatically within the Form controls.

- For examples:
  - **Textboxes** allow: max size of input, automatic capitalization of characters etc.

  - **MaskedTextboxes** allow: input of type, email, date, time etc.

Assertions

- Debug.Assert(<condition>) **interrupts execution** when false
  - outputs the type of failure, call stack and dialog on how to proceed
- Used primarily **during development**
- Allows program to **verify itself** during runtime
- **Pinpoints errors** quickly in large systems during modifications
- Can be used to **document** code

Assertions

Lab 3 exercise 5 example:

VB

```
Function CalculateBMI(ByVal weight As Single, ByVal hight As Single) As Single
        Debug.Assert(weight > 0)
        Debug.Assert(hight > 0)

        Return weight / (hight ^ 2)
End Function
```

C#

```
private static double CalculateBMI(double weight, double hight)
{
        Debug.Assert(weight > 0);
        Debug.Assert(hight > 0);

        return weight / (Math.Pow(hight, 2.0));
}
```

Assertions

- Compiled into code at development, can be compiled out of the code for production releases (release mode)

- Assertions are an example of "**offensive programming**"
  - "Highlight" the existence of bugs in your codes

- For highly robust code, **assert first and then handle the exception anyway**

Assertions

In large software, its unrealistic to flush out all *development-time* errors, thus assertions and exception handling must also be used.

"Fail hard during development so that you can fail softer during production"

Steve McConnell, Code Complete

"A dead program normally does a lot less damage than a crippled one"

A Hunt and D Thomas, Pragmatic Programmer

Error-Handling Techniques

How can expected errors be handled?

1. Return a neutral value
2. Substitute the next piece of valid data
3. Return the same answer as previously
4. Substitute the closest legal value
5. Log a warning message to a file
6. Return an error code
7. Call an error-processing routine/object
8. Display error message
9. Shut down

Dialog Boxes

- Dialog Boxes – a dialog box allows for communication or dialog between the user and an application.
- They are often used to display some message, alert, ask the user a question etc.
- Dialog boxes are different to other forms in that they block all other forms until the dialog box closes.

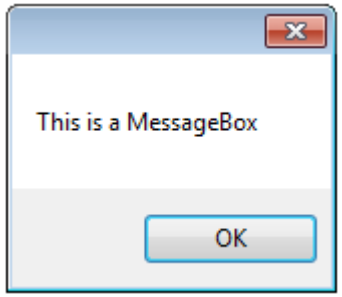
Message Box

- MessageBox is a useful .NET class that can be used to show a **simplified** dialog box with some information or question.
- MessageBox has a method called **Show** which can be used to display the box.
- This can be used to show a MessageBox with a number of options for text, caption, buttons, icon etc.
- This class can be used to display dialog boxes for a number of different uses.

Message Box

The **Show** method with a single string will display a dialog box with a single message

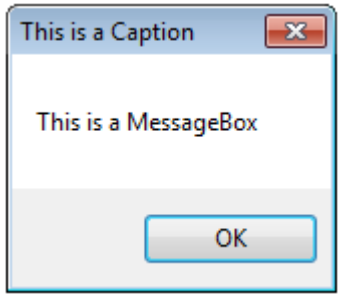
```
MessageBox.Show("This is a MessageBox")
```



Message Box

Two strings will display a box with a caption as well

```
MessageBox.Show("This is a MessageBox", "This is a Caption")
```



Message Box

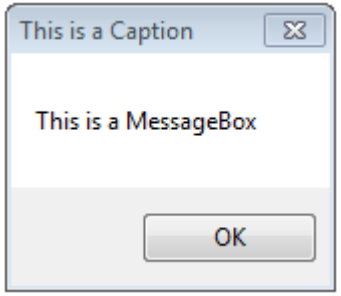
- To use the MessageBox to ask the user for input requires buttons as well.

- These buttons are available in MessageBoxButtons which can be used as follows:

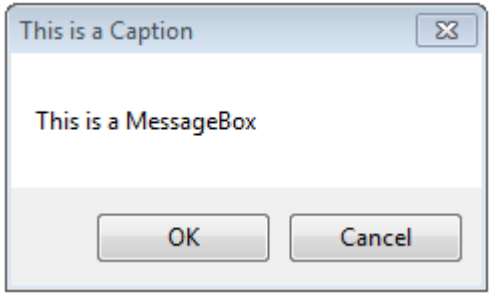
- `MessageBox.Show("MessageBox", "Caption", MessageBoxButtons.<style>)`

Message Box Buttons

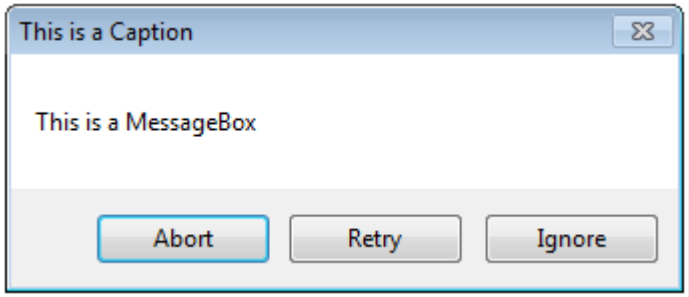
`MessageBoxButtons.OK`



`MessageBoxButtons.OKCancel`

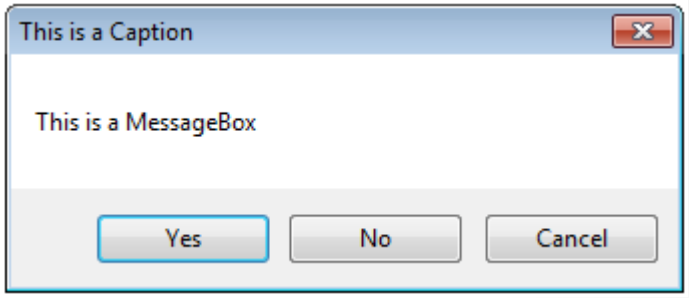


`MessageBoxButtons.AbortRetryIgnore`

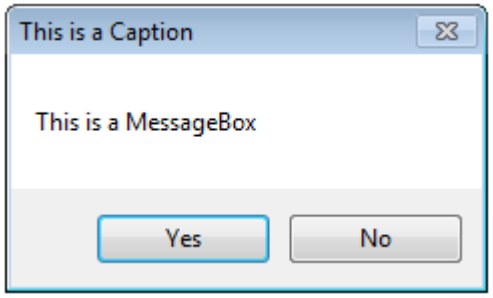


Message Box Buttons

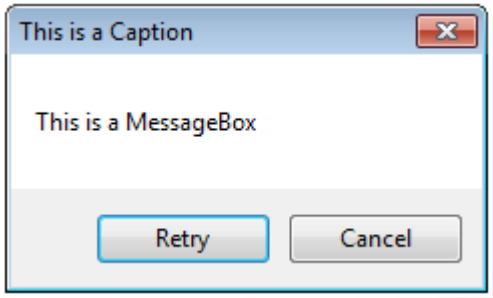
`MessageBoxButtons.YesNoCancel`



`MessageBoxButtons.YesNo`



`MessageBoxButtons.RetryCancel`



Message Box

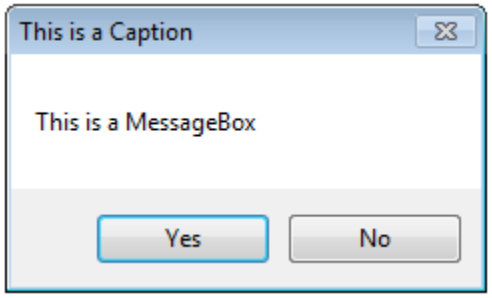
To indicate an error or a warning, a **MessageBoxIcon** can be used.

It can be used in a MessageBox using:

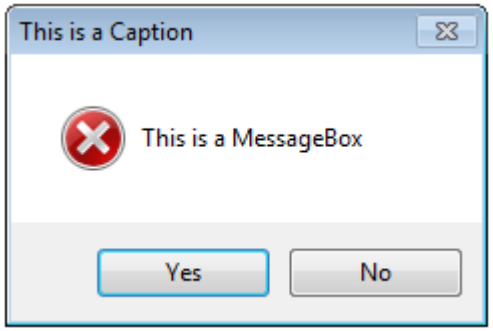
```
MessageBox.Show(“MessageBox”, “Caption”,
MessageBoxButtons.<Style>,
MessageBoxIcon.<Style>)
```

Message Box Icon

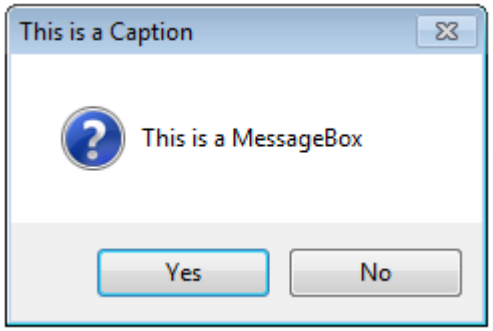
`MessageBoxIcon.None`



`MessageBoxIcon.Hand`

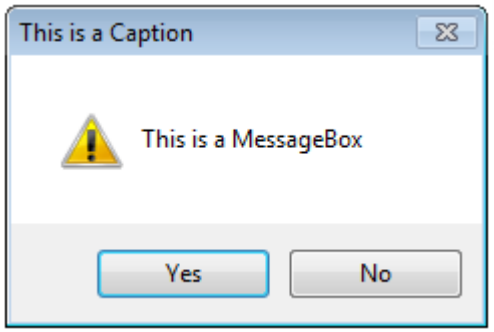


`MessageBoxIcon.Question`

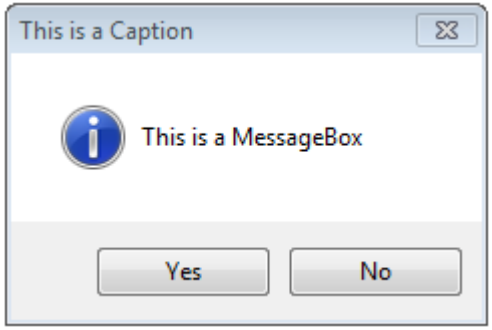


Message Box Icon

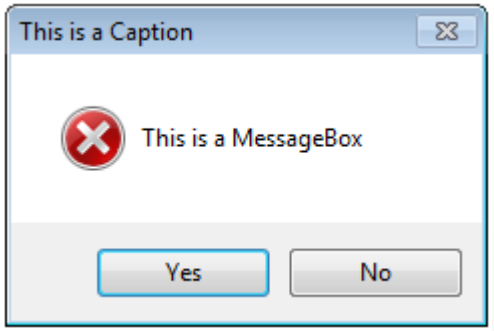
`MessageBoxIcon.Exclamation`



`MessageBoxIcon.Asterisk`

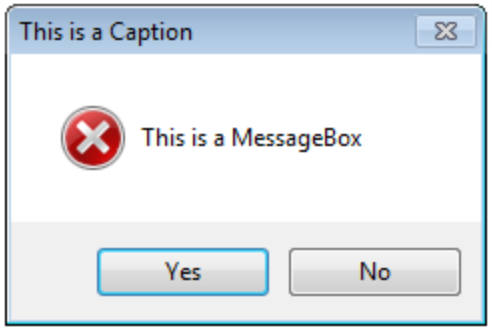


`MessageBoxIcon.Stop`

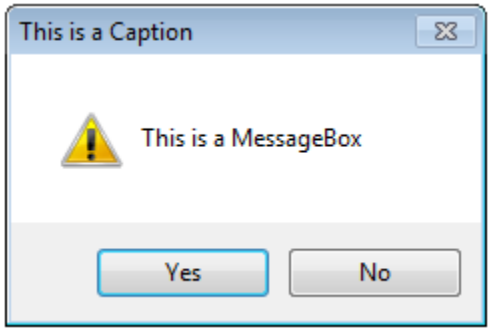


Message Box Icon

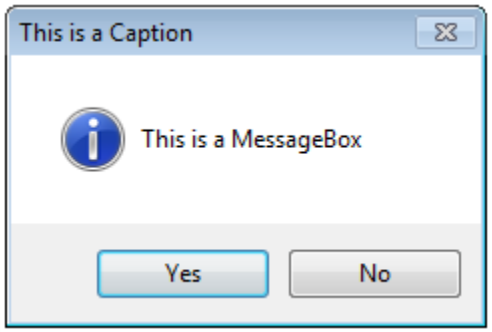
`MessageBoxIcon.Error`



`MessageBoxIcon.Warning`



`MessageBoxIcon.Information`



Message Box

- When a Message Box is displayed, it will return a result.

- This can be used to determine which button was pressed. The result is returned in a **DialogResult**. The possible result values are:

```
DialogResult.None
DialogResult.OK
DialogResult.Cancel
DialogResult.Abort
DialogResult.Retry
DialogResult.Ignore
DialogResult.Yes
DialogResult.No
```

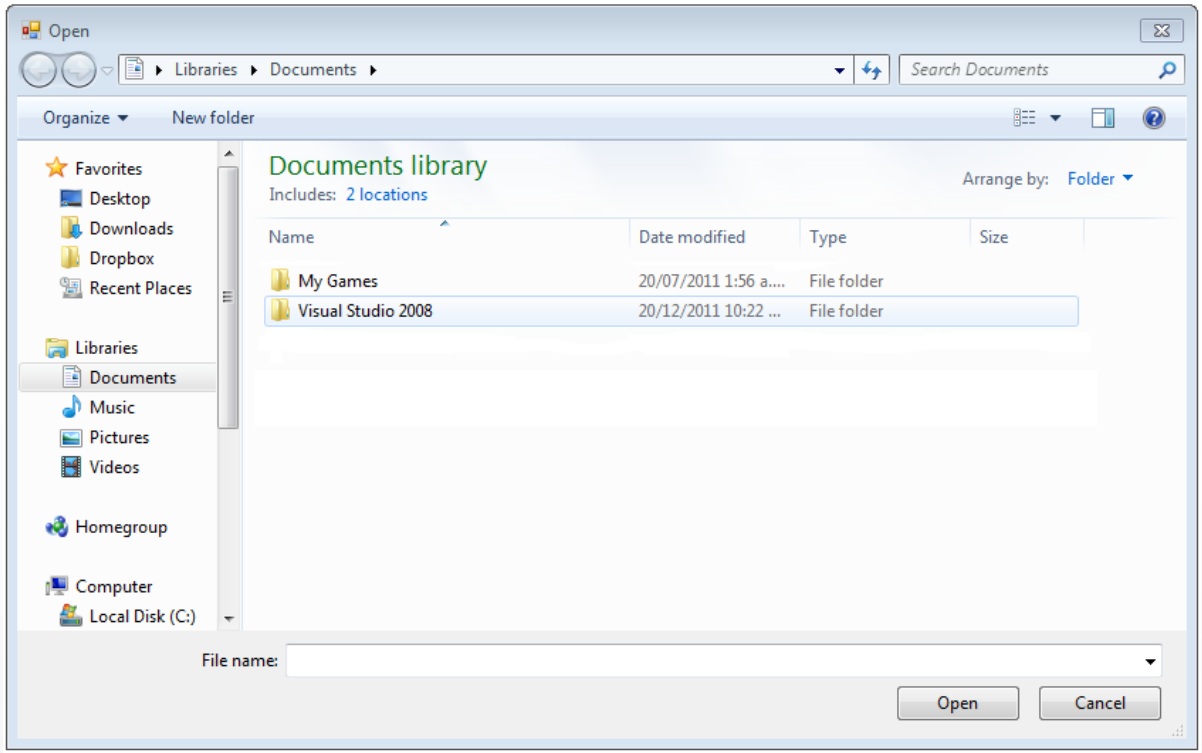
Message Box

These **DialogResult** values can be used to control the behaviour of the application based on what buttons the user pressed.

```
Dim result As DialogResult
result = MessageBox.Show("Are you sure you want to exit?",
                         "Exit?",
                         MessageBoxButtons.YesNo)
If result = DialogResult.Yes Then
   Me.Close()
End If
```

OpenFileDialog

Another very useful dialog box is the OpenFileDialog. An instance of this dialog box must be created and shown. This shows a standard dialog box that allows the user to select a file to open.



OpenFileDialog

- The **OpenFileDialog** will return a **DialogResult** which determines whether the user selected a file or cancelled.

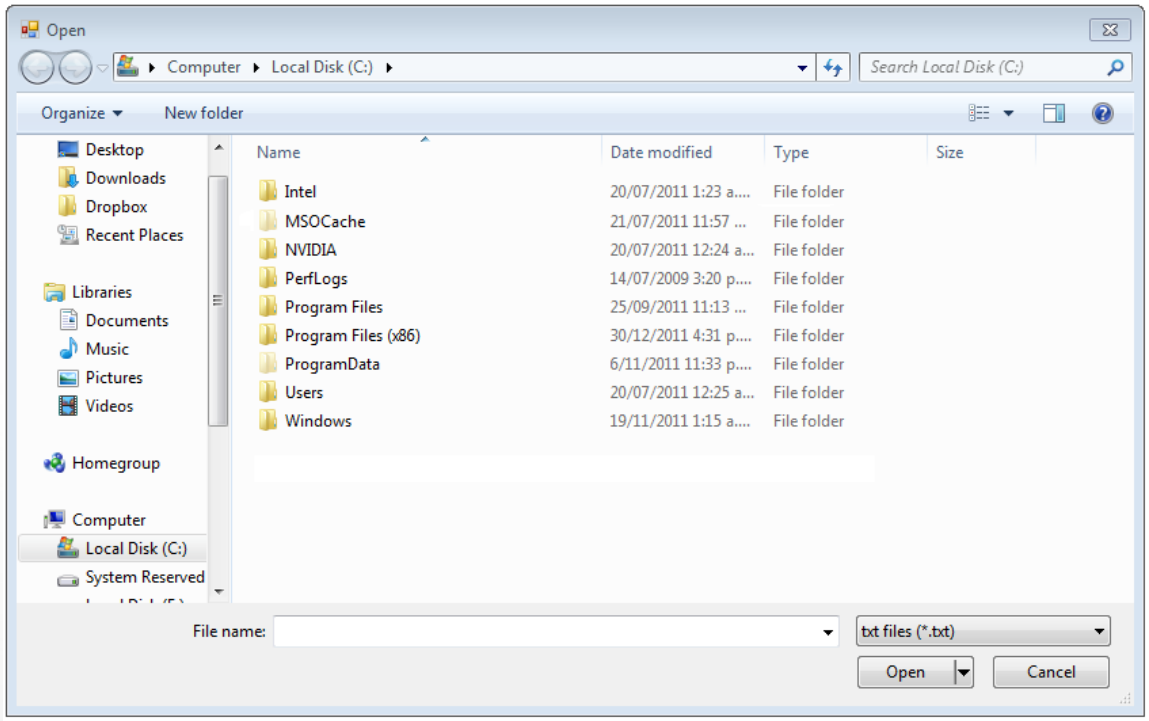
- The selected file can be accessed using:

```
Dim result As DialogResult
Dim openFileDialog1 As New OpenFileDialog()
result = openFileDialog1.ShowDialog()
If result = DialogResult.OK Then
   Dim name As String = openFileDialog1.FileName
   openFileDialog1.OpenFile() 'The file
End If
```

OpenFileDialog

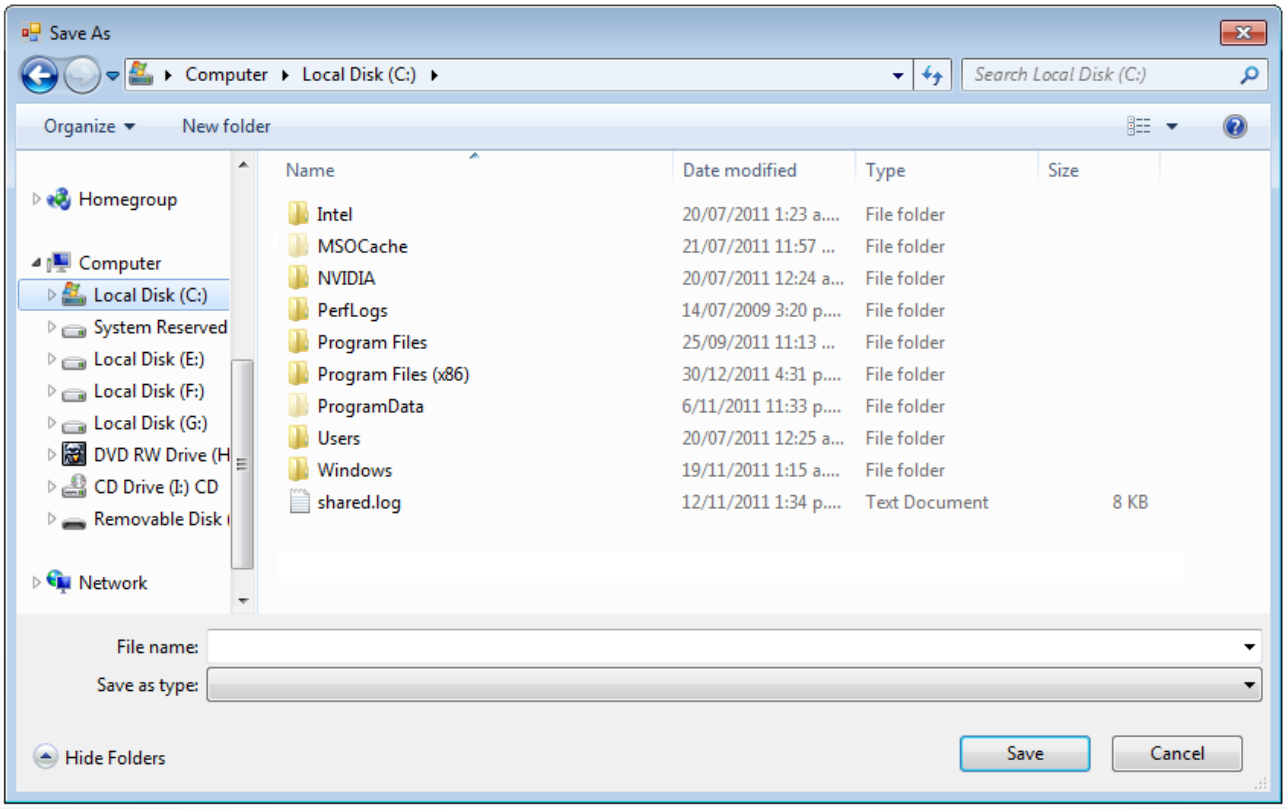
The OpenFileDialog can be configured in a number of ways such as the initial directory, file filters etc. These can be set as follows:

```
openFileDialog1.InitialDirectory = "c:\"
openFileDialog1.Filter = "txt files (*.txt)|*.txt"
```



SaveFileDialog

The SaveFileDialog is almost the same as the OpenFileDialog except it can be used to create new files as well.



SaveFileDialog

```
Dim result As DialogResult
Dim saveFileDialog1 As New SaveFileDialog()
result = saveFileDialog1.ShowDialog()
If result = DialogResult.OK Then
    Dim name As String = saveFileDialog1.FileName
    saveFileDialog1.OpenFile() 'The file
End If
```

Summary

- Defensive Programming
  - Input Validation
    - Prevention and Exception Handling
  - Assertions
- Dialog Boxes
- MessageBox
- OpenFile/SaveFile