Object-Oriented Programming I

Topics Covered

- What is OOP?
  - Re-usability
  - Extensibility
  - Polymorphism
- Classes and OOP Applications
  - Example Classes
- Designing a Word-Counting Application
  - Console Startup Arguments
  - An Input Loop
- Refactoring a Procedural Application
- Class Libraries

What is OOP?

Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which may contain data, in the form of fields, often known as attributes; and code, in the form of procedures, often known as methods. A feature of objects is that an object's procedures can access and often modify the data fields of the object with which they are associated (objects have a notion of "this" or "self"). In OOP, computer programs are designed by making them out of objects that interact with one another. There is significant diversity of OOP languages, but the most popular ones are class-based, meaning that objects are instances of classes, which typically also determine their type.

Wikipedia 2018

OOP Characteristics

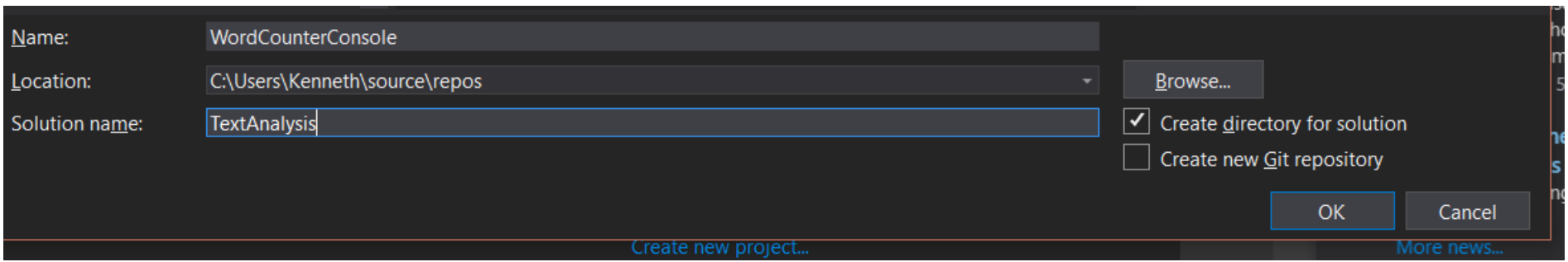
Object Oriented Programming promotes:

- **Re-usability**: Well-designed classes can be used and re-used in multiple contexts.
- **Extensibility**: Through inheritance, functionality of base classes can be extended, refined and re-purposed.
- **Polymorphism**: Objects in an inheritance hierarchy can be used based on their common characteristics and behaviors.
- **Encapsulation**: Each class specializes in one task and exposes just enough information to optimize its utility. Exposing too much information ("implementation details") is a distraction and a risk.
- **Composition**: Applications can bring together the capabilities of multiple classes to compose sophisticated functionality.

A Word-Counting Application

Have you ever wondered how often the word “leviathan” appears in Herman Melville’s novel *Moby Dick*? Probably not :) But nonetheless, lets write an application that can tell us.

Open Visual Studio and create a new “Console App (.NET Framework)” project. In the New Project Dialog, set the name to “WordCounterConsole”, and the Solution Name to “TextAnalysis”.



Program Inputs

Let's not "hard code" our application specifically to Moby Dick and "leviathan". Instead, let's accept these as input. Assume that the text file to be analyzed will be program input, and we will prompt for the search word.

Given this, we must code some reality-checks:

- An argument has been passed to the application
- The argument is a string pointing to a file.

```
static void Main(string[] args)
{
    // Reality Checks
    if (args.Length != 1)
    {
        Console.WriteLine("1 argument is expected.");
        return;
    }
    if (!File.Exists(args[0]))
    {
        Console.WriteLine($"File '{args[0]}' not found.");
        return;
    }
    string document = File.ReadAllText(args[0]);
}
```

When you first type the word "File", it will be given red squiggles, indicating an error.

Hover over the error and select "Show potential fixes." Select the first fix, "using System.IO".

This adds the namespace System.IO that contains File and other classes that we need to use.

The method File.ReadAllText loads the file content as a string.

Asking for Search Text

We now have a (presumably) text file, but we still need the text for which to search. We need to write a prompt asking the user to enter the word. Let's write a method that does this. By putting it into a method, we can reuse it as often as needed.

```
private static string PromptForSearchWord()
{
    Console.Write("Please enter text to search for: ");
    return Console.ReadLine();
}
```

The Input / Search Loop

Let’s create a loop where we prompt for search text, and if the text is non-empty we do the search and output the result. If the entered search text is empty, the program ends. Since this loop must execute at least one time, a do loop serves nicely:

```
do
{
    string input = PromptForSearchWord();
    if (string.IsNullOrWhiteSpace(input)) return;
    // Search for text and output result
} while (true);
```

The FindWordCount Method

Now we need to implement a method that searches a document for a substring and returns a count of the # found. Thanks to the String class's IndexOf method, this is not difficult:

```
private static int FindWordCount(string document, string searchText)
{
    int count = 0;
    int ndx = document.IndexOf(searchText);
    while(ndx >= 0)
    {
        count++;
        ndx = document.IndexOf(searchText, ndx + searchText.Length);
    }
    return count;
}
```

Notice how we utilize two different overloads of the String.IndexOf method.

The Input / Output Loop

Our do loop can now be completed:

```
do
{
    string input = PromptForSearchWord();
    if (string.IsNullOrWhiteSpace(input)) return;
    int nOccur = FindWordCount(document, input);
    Console.WriteLine($"The text '{input}' occurs {nOccur} times.");
} while (true);
```

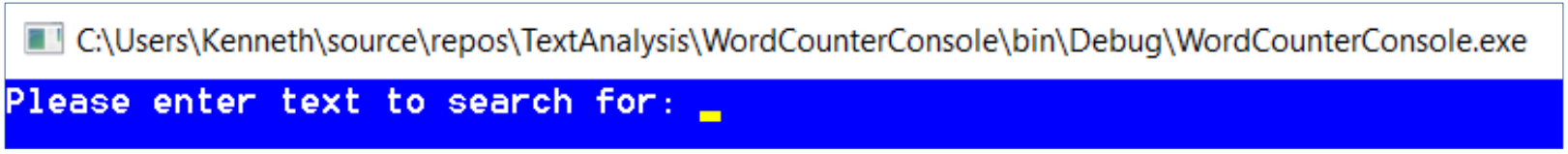
The program is now complete. Use the Build menu to build it, and the next slide will show how to run it.

Running WordCounterConsole

In Solution Explorer, right click on the WordCounterConsole project and select “Open Folder in File Explorer”. In the folder that opens, double-click “bin” then double-click “Debug”. You should see the the executable file “WordCounterConsole.exe”.

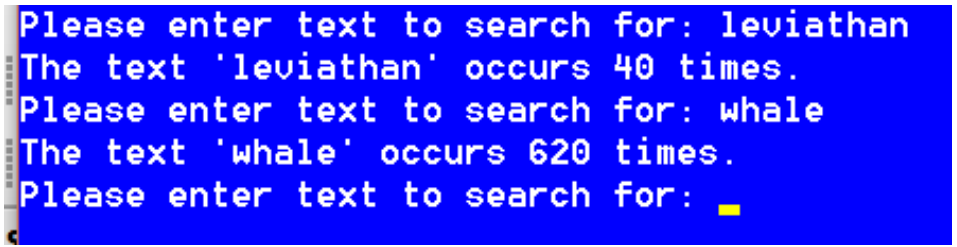
The text of MobyDick should be available in the course resources. If not, grab it from the web. Wherever you choose to save it, open its containing folder.

Now, drag MobyDick.txt onto WordCounterConsole.exe. This will run our program using the file location of MobyDick.txt as the argument. You should see something like this:



Search Results

Now enter our word “leviathan” and press <Enter>. For fun, let’s compare it to a more common word “whale” - enter that and press <Enter>. You should see something like this:



This is good! But … is it *Object Oriented*? Is it re-usable? Can it be extended?

No. This is procedural programming, very effective in its own way, but lacking the advantages of OOP.

Let’s rearrange this program to make it OOP.

Refactoring

Refactoring is a process where we rearrange code to meet new design objectives.

Typical design objectives are:

- **Simplification** – remove duplications and redundancies that make code more complicated than necessary.
- **Reorganization** – if a class becomes too complicated because it tries to do too much, split it into two (or more) classes each with well-defined purpose, then use composition to recreate the desired functionality.
- **Relocation** – if desired functionality is implemented in a scope that is inaccessible, we can move it to a location (a library, for example) where it is re-usable.

Our code is simple already, but we need to reorganize it and relocate it so that it can support other applications.

Refactoring Plan

Here is our plan:

1. Extract our search functionality into a class.
2. This class should be located in a class library where it can be used by multiple applications.

In step 1, we can also enhance the functionality by adding an option as to whether the search should be case sensitive (the current implementation is always case sensitive).

A Class Library Project

In Visual Studio's Solution Explorer, right-click on the TextAnalysis solution and select Add → New Project.

In the New Project dialog, select "Class Library (.NET Framework)", and name the library "TextAnalyzers.Lib". Then click "OK".

In the new library, Visual Studio creates a class called "Class1". In the Solution Explorer, right-click on Class1 and select Rename. Type "WordSearcher" and press <enter>. Visual Studio will prompt you to rename the class. Select 'Yes'.

WordSearcher Class

Our WordSearcher class will have the following members:

- A public **constructor** which accepts a document – a string – which is the target of subsequent searches.
- A public read-only **Document** property – the same document string passed to the constructor.
- A public method named **GetWordCount** which accepts two arguments: a string to search for, and a boolean indicating case-sensitivity. The second argument should be optional, and if omitted it defaults to true. The method returns an integer.
- A static method named **FromFile** which accepts a filename argument and returns a WordSearcher whose Document is the text from that file.

WordSearcher Constructor

Here are the constructor and the Document property:

```
public WordSearcher(string document)
{
    if (string.IsNullOrEmpty(document)) throw new ArgumentNullException(nameof(document));
    Document = document;
}

public string Document { get; private set; }
```

A **constructor** is a special method that allows us to initialize an instance of a class. Constructors always have the same name as the class. Like all methods, constructors can be overloaded – a class can define multiple constructors with distinct parameter lists.

Document is a property. It is public – any user of the class can read it. But the setter is private, meaning that it can only be set from within methods of the class itself.

WordSearcher.GetWordCount

The GetWordCount method implementation is shown below. It uses a default value for the second argument, and makes use of overloads of String.IndexOf that accept a StringComparison argument.

```
public int GetWordCount(string searchText, bool isCaseSensitive = true)
{
  StringComparison scomp = isCaseSensitive ? StringComparison.InvariantCulture :
    StringComparison.InvariantCultureIgnoreCase;
  int count = 0;
  int ndx = Document.IndexOf(searchText, scomp);
  while (ndx >= 0)
  {
    count++;
    ndx = Document.IndexOf(searchText, ndx + searchText.Length, scomp);
  }
  return count;
}
```

WordSearcher.FromFile

Ideally, we might add a second constructor that accepts a filePath string argument. But we already have a constructor that accepts a single string, so we will instead use a static factory method:

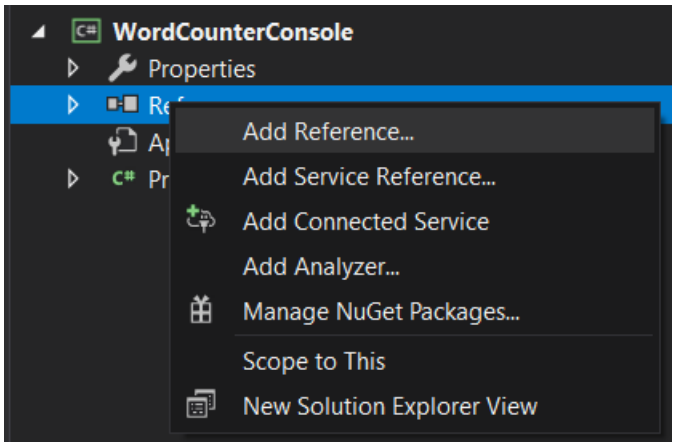
```
public static WordSearcher FromFile(string filePath)
{
    if (!File.Exists(filePath)) throw new ArgumentException($"File not found: {filePath}");
    return new WordSearcher(File.ReadAllText(filePath));
}
```

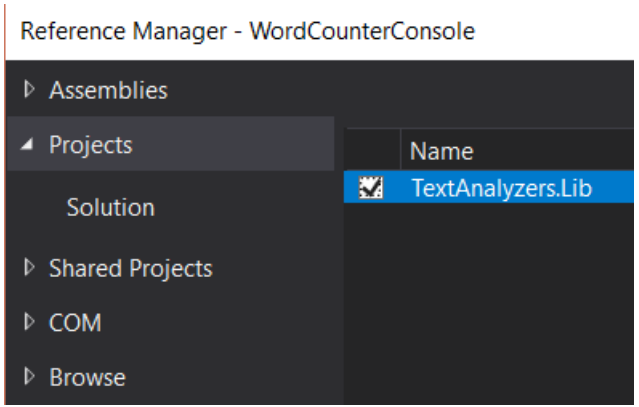
The method first ensures that the file referenced by filePath exists, and then returns a WordSearcher constructed with the text content of that file.

Using WordSearcher

We're now ready to plug our WordSearcher into our console application. First, we need to add a reference to the class library we just created.

In Solution Explorer, right-click on our application's References folder and select 'Add Reference'. In the Reference Manager, check the box next to our library TextAnalyzers.Lib, then click 'OK'.





Refactoring Program

We can now replace two lines in our Main method as shown below. We can also remove our static FindWordCount method.

```
//string document = File.ReadAllText(args[0]);
WordSearcher searcher = WordSearcher.FromFile(args[0]);
do
{
  string input = PromptForSearchWord();
  if (string.IsNullOrWhiteSpace(input)) return;
  //int nOccur = FindWordCount(document, input);
  int nOccur = searcher.GetWordCount(input);
  Console.WriteLine($"The text '{input}' occurs {nOccur} times.");
} while (true);
```

One last point: we did not move our 'PromptForSearchWord' method into WordSearcher. Why? *It does not belong there*. It is not word-search functionality, and it is tightly-coupled to our Console application. It would not work, for example, in a windowed application.

OOP Part I: Recap

We've barely scratched the surface of OOP programming, but let's recap what we've done:

- We have a class WordSearcher.
- WordSearch has state – it represents a searchable text document.
- We can create a WordSearcher by calling its constructor, passing in the document text;
- or we can create a WordSearcher by calling it's FromFile factory method with the path to a text file.
- WordSearcher currently has just has one instance method, GetWordCount.
- WordSearcher is defined in a library, where we can re-use it in multiple applications.

In the next presentation we will take our new class a few steps further.

Topics Covered

- What is OOP?
  - Re-usability
  - Extensibility
  - Polymorphism
- Classes and OOP Applications
  - Example Classes
- Designing a Word-Counting Application
  - Console Startup Arguments
  - An Input Loop
- Refactoring a Procedural Application
- Class Libraries