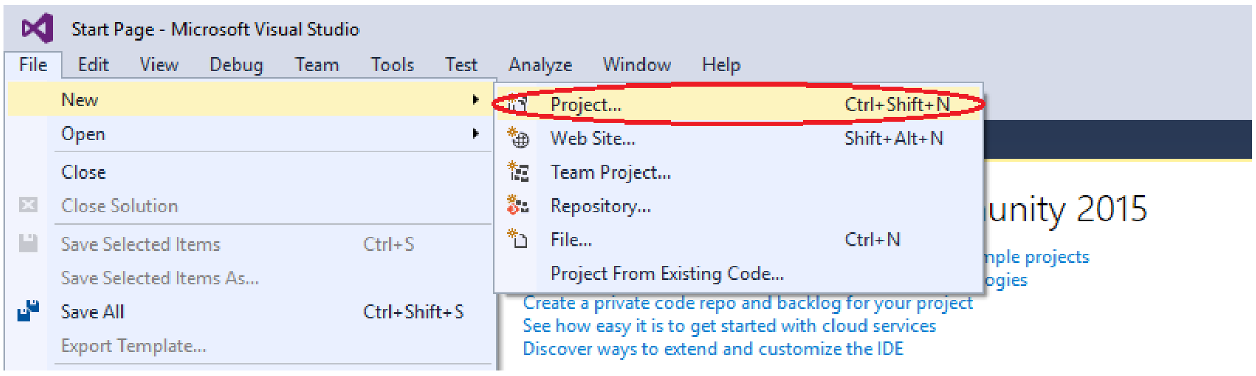
BashSoft piece: Stacks and Queues

Part I: Creating the base functionality needed

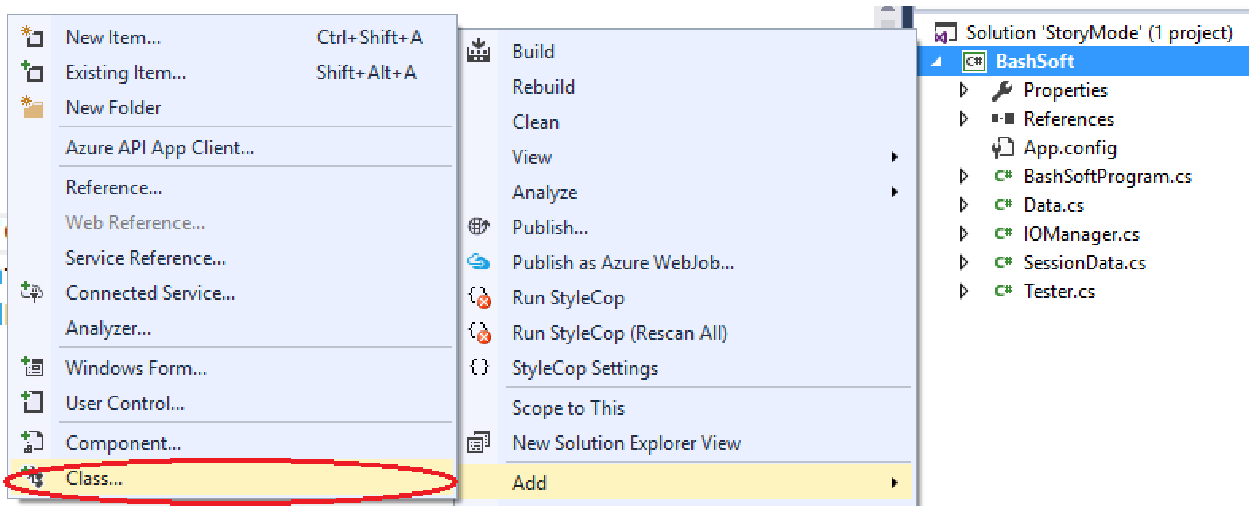
Problem 1. Create a Visual Studio Project

Our first task is to **create** a **project** called **BashSoft**, which we will extend until the end of the course so you might want to **save it** somewhere, where you can **easily find** it and where you can be **sure** you **won't delete** it. You can call the class with the **Main()** method, **Launcher**, because from it we will only call the specific functions we want to execute, but our execution logic will be in other classes.

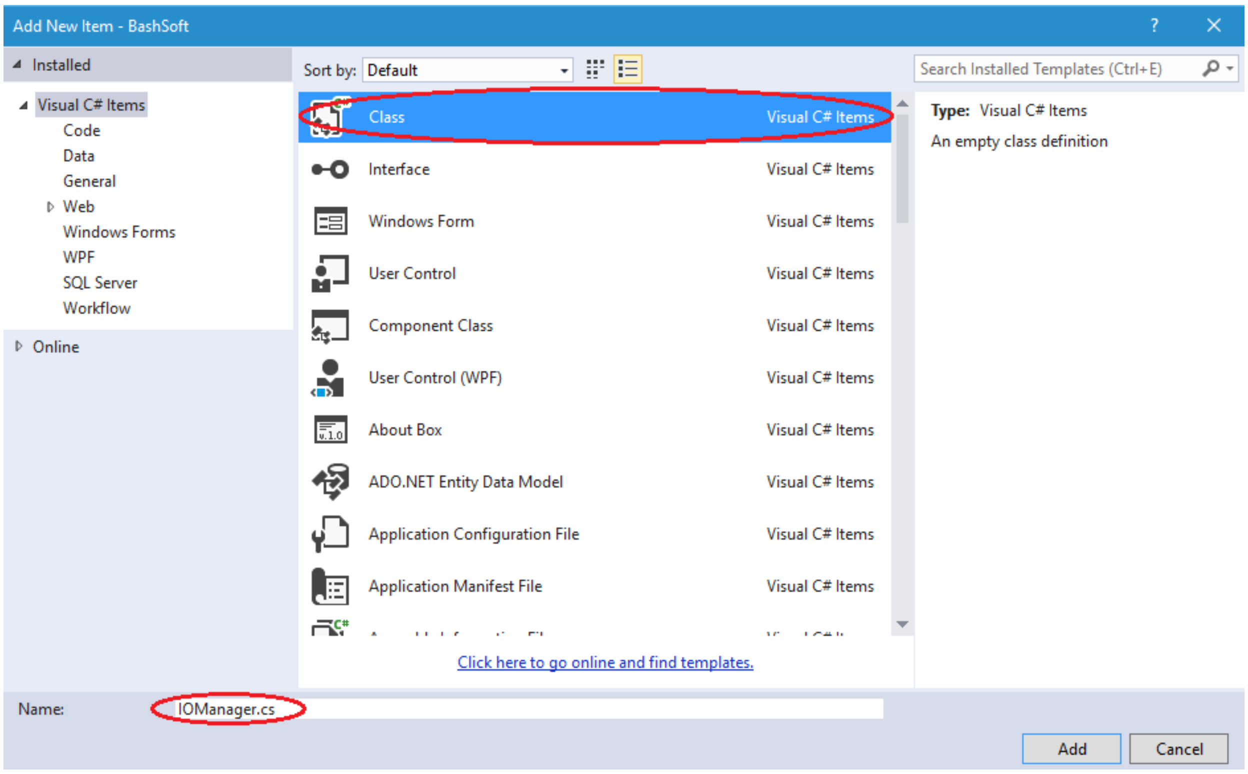


Start Page - Microsoft Visual Studio
File Edit View Debug Team Tools Test Analyze Window Help
New
Open
Close
Close Solution
Save Selected Items Ctrl+S
Save Selected Items As...
Save All Ctrl+Shift+S
Export Template...
Project... Ctrl+Shift+N
Web Site... Shift+Alt+N
Team Project...
Repository...
File... Ctrl+N
Project From Existing Code...

Once you have created the project, you have to add a class that we will call **IOManager** and it will give us the functionality for traversing the folders and other behaviors.



In the next menu you have to choose to create a new class with the name "**IOManager**"


Add New Item - BashSoft
Installed
Visual C# Items
Code
Data
General
Web
Windows Forms
WPF
SQL Server
Workflow
Online
Search Installed Templates (Ctrl+E)
Class — Visual C# Items
Interface — Visual C# Items
Windows Form — Visual C# Items
User Control — Visual C# Items
Component Class — Visual C# Items
User Control (WPF) — Visual C# Items
About Box — Visual C# Items
ADO.NET Entity Data Model — Visual C# Items
Application Configuration File — Visual C# Items
Application Manifest File — Visual C# Items
Click here to go online and find templates.
Add
Cancel

Sort by: Default

Type: Visual C# Items

An empty class definition

Name: IOManager.cs

Next the only things we have to **change** over the **generated class** is **to** add it "**public static**" before the keyword **class**. The keyword "**public**" means we can **use** our class **everywhere** in our project. Sometimes we will leave some methods **private**, because we may want to **hide** some of the **functionality** of our class, in front of the other world. The other keyword "**static**" means that we can do "**general/global**" stuff with it. Example: "**Math**", "**Console**".

The **opposite of static** we can say are classes like "**Stack**, **List**, **StringBuilder**" for which we have to say "**new List<T>**" in order **to create** a **new** list. The **static** classes **do not need to be created** like we don't say "new Math", instead we just use **Math.Sqrt().**

So now your class should look something like this:

```
public static class IOManager
{
}
```

Problem 2. Create a flexible interface for output to the user

So now that we have our first class we are going to have to implement some functionality that this class should have. But before that, first we have to decide **how** are we going **to communicate with the user efficiently** and if this is something that we have to use in many places, how can we change it or replace it easily using doing only a few changes in one place. The solution behind such a problem give us one of the **Design Patterns** which are a topic of the **next course**, but the main idea of this one is that we can **hide** some **functionality** (The writing to the console, which can easily be changed for writing in a file), **by using** a **class** that only gives us **base functionality** for communication with a user.

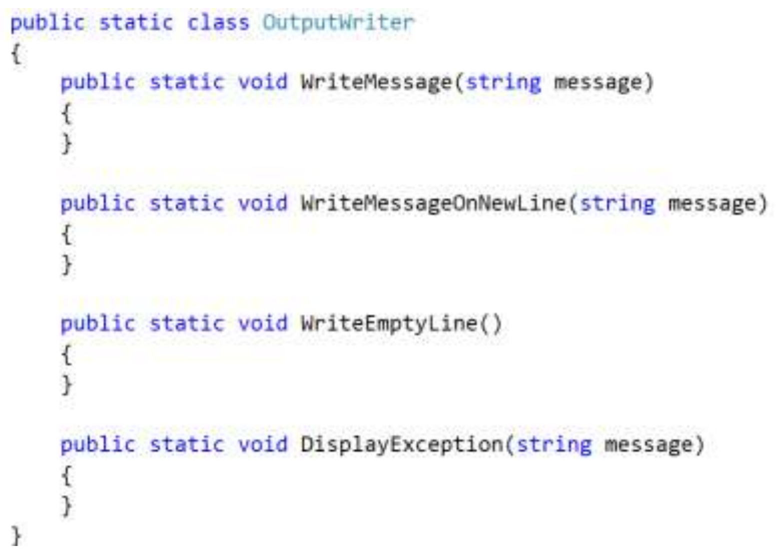
Our new class can be called **OutputWriter** and you should make it following the steps above as described for the **IOManager**. The **new class** again has to be **public** and **static** and after you've created it, it should look something like this:

```
public static class OutputWriter
{

}
```

So now we can add a few methods that we will use throughout our whole app that write to the currently set output.

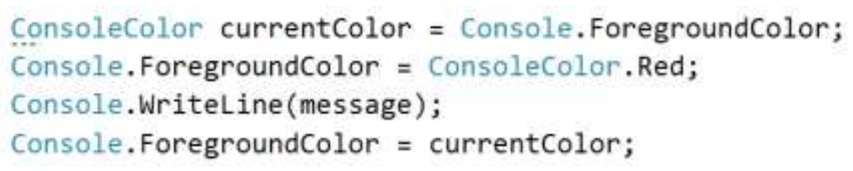
- The first method gives us the ability to **write a message.**
- The second method to implement is a method for **writing a message on a new line**.
- The third method is to **write a new empty line**.
- The fourth method is to **write a different kind of message** which is an **error/exception**.

The class with the three methods inside it should look something like this:



```
public static class OutputWriter
{
    public static void WriteMessage(string message)
    {
    }

    public static void WriteMessageOnNewLine(string message)
    {
    }

    public static void WriteEmptyLine()
    {
    }

    public static void DisplayException(string message)
    {
    }
}
```

The implementation of the first three methods is pretty common. The **first** one only **writes the message on the console**, and the **second** one **writes the message** and goes to the **next line after that**. The **third** only **writes an empty line on the console**. The **fourth** method however has some small specifics. The specifics are that we need to **get** the **current foreground color**(font color), **save it**, **change the foreground color to red**, **write the given message** and finally **change** the **foreground color to** the **one before**. Here is how this has to look in code:



```
ConsoleColor currentColor = Console.ForegroundColor;
Console.ForegroundColor = ConsoleColor.Red;
Console.WriteLine(message);
Console.ForegroundColor = currentColor;
```

Now that we are ready with the user output. It's time to implement the traversal of the folders and in the future, **if** we want to **change** the **output destination**, we **only** need to **change** it here in the **class** we **just made**, and not everywhere where we've written **Console.WriteLine()**.

Part II: Implementing the traversal alorithm

Problem 3.   Traversing the folder of the project

Our next task is to learn how to **traverse folders** in order to be able to do all kinds of operations with files that are stored on the hard drive. This is our first small step into the big picture.

We will **traverse the folder** of the project **using queue** with a technique called **BFS**. Here is a animation that can probably help you understand how **BFS** works, however this is not the main point, so you may just use it, without going into too much depth about how it works.

Shortly we will create a method **TraverseFolder (string path)**. How does it traverse a folder? **First** it **enqueues** the **folder** that we **pass as parameter** in the method signature. After that it **dequeues every folder** in the queue one at a time **until** the **queue** becomes **empty**, **while at the sam**e time **enqueues all** of its **subfolders** at the end of the queue.

For our purposes we will **use** the static class **DirectoryInfo**, which **will give** us all the **information** we need **for** the **directories** we work with, don't worry you'll get familiar with it in a few lectures. Here is the initialization of the method with the queue. We **enqueue** the **root folder** we wanted to traverse first and also **create** a **variable for** the **indentation** of the first path, so it can be later **used for displaying** the **levels of depth** we've entered while traversing.

```
public static void TraverseDirectory(string path)
{
    OutputWriter.WriteEmptyLine();
    int initialIdentation = path.Split('\\').Length;
    Queue<string> subFolders = new Queue<string>();
    subFolders.Enqueue(path);
```

Next we need to make sure we will **traverse all** of the **subfolders** that we have **in** the **queue** so we will traverse **while** the **queue** is **not empty** (that is why we **push** the **initial element** in the queue).

For each iteration of the **while loop** we want to **dequeue** a **folder** that we are going to traverse and to **print** its **path**, but in order to know how many level in depth we have entered, we are going to **use another indentation variable** and **take the delta between the two**.

```
while (subFolders.Count != 0)
{
    //TODO: Dequeue the folder at the start of the queue
    //TODO: Print the folder path
    //TODO: Add all it's subfolders to the end of the queue
```

Also **for each folder** we need to **iterate all** its **subfolders** and **add them to** the end of the **queue**. We can do this with a simple foreach loop:

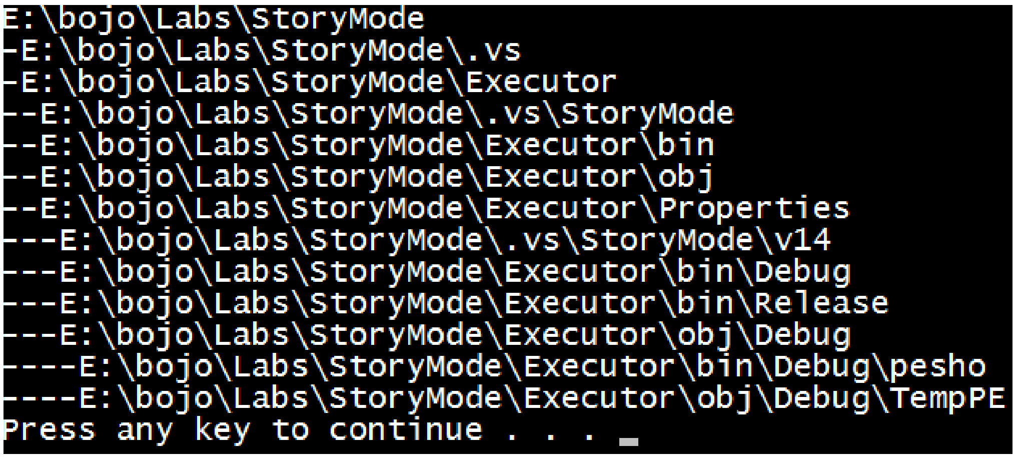
```
string currentPath = subFolders.Dequeue();
int identation = currentPath.Split('\\').Length - initialIdentation;

//TODO: Print the folder path

foreach (string directoryPath in Directory.GetDirectories(currentPath))
{
    //TODO: Add all it's subfolders to the end of the queue

}
```

You can **print** the **full name of** the **directory** with the following line of code:

```
OutputWriter.WriteMessageOnNewLine(string.Format("{0}{1}",
                                                  new string('-', identation),
                                                  currentPath));
```

Now if you filled your TODOs properly when you run your code you should get some output like this if we call the method through the **Main()**

```
static void Main()
{
    IOManager.TraverseDirectory(@"E:\bojo\Labs\StoryMode");
}
```



```
E:\bojo\Labs\StoryMode
-E:\bojo\Labs\StoryMode\.vs
-E:\bojo\Labs\StoryMode\Executor
--E:\bojo\Labs\StoryMode\.vs\StoryMode
--E:\bojo\Labs\StoryMode\Executor\bin
--E:\bojo\Labs\StoryMode\Executor\obj
--E:\bojo\Labs\StoryMode\Executor\Properties
---E:\bojo\Labs\StoryMode\.vs\StoryMode\v14
---E:\bojo\Labs\StoryMode\Executor\bin\Debug
---E:\bojo\Labs\StoryMode\Executor\bin\Release
---E:\bojo\Labs\StoryMode\Executor\obj\Debug
----E:\bojo\Labs\StoryMode\Executor\bin\Debug\pesho
----E:\bojo\Labs\StoryMode\Executor\obj\Debug\TempPE
Press any key to continue . . .
```

You are now ready with your first tool for the wanted bash. Soon you will be able to easily change your position in the file system and do different operations with other files.