Object Oriented Programming with Applications
Lecture 6

David Šiška & Witold Gawlikowicz

School of Mathematics, University of Edinburgh

2018-19[1]

[1] Last updated 16th October 2018

Lecture 6

Input / output with Excel:

- Library vs. Executable
- ExcelDNA
- Basic input & output.

What makes good code

Input and output

Dealing with input and output (IO) is one of the more tedious aspects of programming.

So far we have seen output to console: `Console.WriteLine`

and input from console: `string x = Console.ReadLine()`.

It is no fun to use such programs.

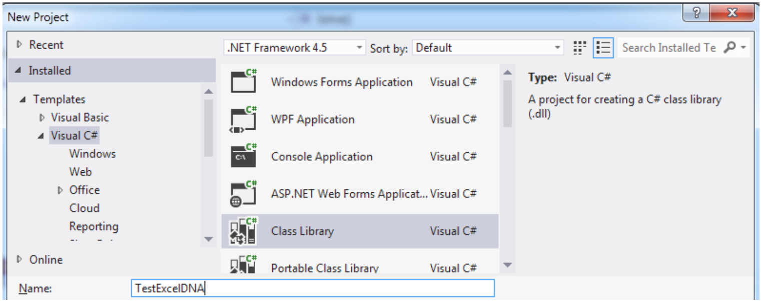
Creating a full windows-based user interface is a lot of work.

Input and output with Excel

We will use MS Excel for data input and output by writing new Excel functions using ExcelDNA library.

Input and output with Excel: Step 1

Create a new Project “C# Class Library”.



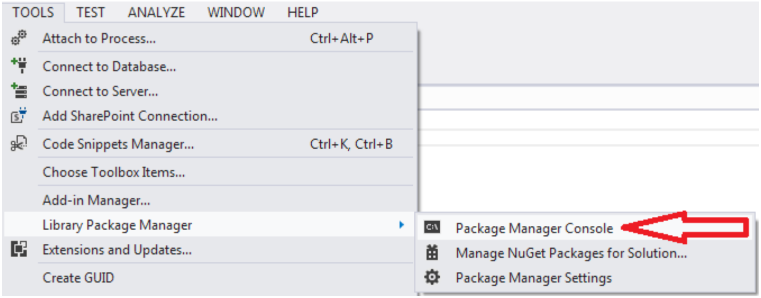
Type: Visual C#

A project for creating a C# class library (.dll)

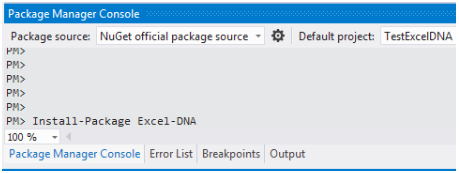
Name: TestExcelDNA

Input and output with Excel: Step 2

Open the package manager console.



Type `Install-Package Excel-DNA` and press Enter:



```
PM>
PM>
PM>
PM>
PM>
PM> Install-Package Excel-DNA
```

Input and output with Excel: Step 3

Modify code in `Class1.cs` to be

```
using ExcelDna.Integration;

namespace TestExcelDNA
{
    public static class MyFunctions
    {
        [ExcelFunction(Description = "My first .NET function")]
        public static string SayHello(string name)
        {
            return "Hello " + name;
        }
    }
}
```

Input and output with Excel: Step 4

Start the project.

You should see Excel.

If asked press “Enable this add-in for this session only”.

In one cell enter `=SayHello("Bob")`.

Input and output with Excel: Step 4

Start the project.

You should see Excel.

If asked press “Enable this add-in for this session only”.

In one cell enter `=SayHello("Bob")`.

What makes good code?

- Correct - it does what it is supposed to do.
- Robust - “program performs well not only under ordinary conditions but also under unusual conditions that stress its designers’ assumptions.” Example: imagine you wrote a function for option pricing using Black–Scholes formula. What happens if the input is $\sigma = 0$? What about $S = 0$?
- Readable - it should be easy to follow what the code does (use indentation, long and descriptive variable, class, method names, comment the code).

What makes good code?

- Modular / reusable - any functionality should only be implemented once. Example: sorting `int`, `double` and `char` is effectively the same. We should not need three different methods.
- Felxible - adding more functionality should be possible.
- Efficient / fast

What makes good code? Continued.

But quoting *Donald Knuth*[2]: “There is no doubt that the grail of efficiency leads to abuse. Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: **premature optimization is the root of all evil.**

Yet we should not pass up our opportunities in that critical 3%. A good programmer will not be lulled into complacency by such reasoning, he will be wise to look carefully at the critical code; but only after that code has been identified.”

[2] Famous for analysis of algorithm complexity. Developed TeX. Wrote “Art of computer programming”.

What makes fast / efficient code?

What to do about those critical 3%?

1. Is there really no formula that will give answer directly?
2. Optimal numerical method. Choose the one with the best "rate of convergence" for the problem you are solving.
3. How much accuracy do you need? No point having one part of code working to 10^{-16} if another, independent part is only accurate to 10^{-6}.
4. Algorithm complexity: e.g. $n \log n$ vs. n^2 makes a massive difference in practice (Quicksort vs. Bubblesort, FFT vs. direct use of numerical integration).
5. Code optimization: do not copy large block of memory unnecessarily, allocate all memory needed in one big chunk rather than in small pieces...

Summary of Lecture 6

We have discussed:

- Debugging
- What makes good code?

Summary of course: what to remember

- Floating point numbers are not real numbers and roundoff errors can lead to incorrect calculations resulting from "correct" formulae.
- Good code is: *correct*, readable / understandable (and commented where necessary), robust, reusable, extendable, ...
- and efficient / fast i.e. optimized in the right places by use of efficient / appropriate numerical method.
- Interfaces, Classes, Methods etc. and more generally the concept of Encapsulation are there for you to create readable, robust, reusable and extendable code.
- Correctly used Exceptions help writing robust code.
- Use libraries for standard algorithms (unless it's a problem sheet where your task is to develop the algorithm).

Summary of course: what we did not cover

Among other things:

- Some very language specific C# features.
- Working with files, databases, network for input / output.
- LINQ - Language Integrated Querry - especially useful for working with large datasets or interacting with SQL databases
- Multi-threading for multi-core CPUs and multi-CPU computers.

Big workshops

What will be the topics:

1. Numerical integration and Newton's method.
2. Finite-difference method for Black–Scholes PDE.
3. Monte Carlo for option pricing.
4. Minimization / optimization / calibration.
5. To be confirmed depending on your interests or weaknesses

There will be two "small" assignments based on the above labs worth 5% and 15% giving a total of 20%. Dates

1. Question details Wednesday Week 6 i.e. 24th October, due Monday Week 8 i.e. 5th November (12:00).
2. Question details Wednesday Week 8 i.e. 7th November, due Monday Week 10 i.e. 19th November (12:00).

Project

Details will be given on Tuesday Week 10 i.e. 20th November (4.00-6.00pm - room TBC)

It will be worth 80% points.

You will be expected to provide a written report, as well as a C# code / solution. The written report should include:

- Why you chose to design the interface the way you did.
- Explanation of the structure of the code. Motivation why it is structured the way it is with: modularity, extendability and robustness in mind.
- Brief explanation of the formulae / methods your code implements

Due Week 1, Semester 2.