# Exercise Set 0: Geometrical Objects

Software Development 2017
Department of Computer Science
University of Copenhagen

Oleks Shturmov <oleks@oleks.info>

**Due:** Friday, February 10, 15:00

This is the first exercise set in the course Software Development, B3-4 2017. The exercise sets give you hands-on experience with the topics covered during the week. You should complete the exercises individually, but you are welcome to discuss your work with your peers.

## 1 Install Party (5%)

In this course, we will be programming in C#, atop of Mono, the open-source implementation of Microsoft's .NET platform. Unlike your previous course, Programming and Problem Solving, we will also be using an *integrated development environment* (IDE) known as MonoDevelop on Linux, and Xamarin Studio on Windows and macOS. Follow the guides the following guides to get started.

1.1. Install MonoDevelop / Xamarin Studio.

See our online guide for further details:

`https://git.dikunix.dk/su17/Guides/blob/master/IDE.md`

1.2. Read our C# style guide apply our policy in your IDE.

See our online guide for further details:

`https://git.dikunix.dk/su17/Guides/blob/master/CSharpStyle.md`

## 2 Geometry I (15%)

The following tasks are aimed to get you started with the IDE.

### Console Project (5%)

The humble `Hello World!` program is often used to show off a language, or a software development environment. With MonoDevelop / Xamarin Studio, this is mostly a point-and-click exercise:

2.1. Open MonoDevelop.

2.2. Select File, New, Solution.

2.3. Select .NET, Console Project (C#), and press Next.

2.4. Name the solution Geometry, and the project TUI.

Choose a location for the solution on your machine.

For now, always choose the following options by default:

- ☑ Create a project within the solution directory.
- ☐ Use git for version control.

2.5. Press Create.

This will setup a `HelloWorld` solution and project, and drop you into a file `Program.cs`, with the following code:

```
using System;

namespace TUI
{
    class MainClass
    {
        public static void Main (string[] args)
        {
            Console.WriteLine ("Hello World!");
        }
    }
}
```

2.6. Right-click on the project and press Run Item.

You should see a console window popping up and saying:

```
Hello World!

Press any key to continue...
```

2.7. Try to answer as many of these follow-up questions as you can:

2.7.1. What is `"Hello World!"`?

2.7.2. What is `Console.WriteLine`?

2.7.3. What is `Main`?
*Hint:* Try running with the method name `HelloWorld` instead.

2.7.4. What is `MainClass`?
*Hint:* Try running with the class name `HelloWorld` instead.

2.7.5. What is `HelloWorld`?
*Hint:* Try running without the namespace declaration.

2.7.6. What are `args`?
*Hint:* Try running with an empty argument list.

2.7.7. What is `System`?
*Hint:* Try building without the `using` statement.

2.7.8. What is `Console`?
*Hint:* Try running with `System.Console` instead.

2.8. Close the console window.

2.9. Restart the IDE, and run `HelloWorld` again.

## Library Project (5%)

We will experiment with multiple user interfaces, but will expose the same underlying library.

Begin by creating a Geometry project:

2.10. Right-click on the solution.

2.11. Select Add, Add New Project...

2.12. Select Library (C#), and press Next.

2.13. Let the project be called “Geometry”.
This will create a library project with a class “MyClass”.

2.14. Right-click on the class name “MyClass”.

2.15. Select Refactor, Rename.

2.16. Change the name to “Vertex”, and tick off the following:

☑ Rename file that contains public class.

Now, let the TUI project reference the Geometry project.

2.17. Right-click on References under the TUI project.

2.18. Select Edit References...

2.19. Tab over to Projects, check “Geometry”, and press OK.

2.20. To make sure that things work, try to print a new instance of `Vertex` in you `Main`, build, and run the TUI. How would you describe the text that is printed?

## Gtk# 2.0 Project (5%)

2.21. Add a new Gtk# 2.0 Project to your solution. Call it GUI.

Unlike with the TUI project, you now also get a `MainWindow.cs`.

2.22. Right-click on the project and select Run Item to see it run.

2.23. Right-click on the solution and select Options.

2.24. Under Run, and Startup Project, you can now select GUI as the start-up project instead of TUI by default.

(You might want to change this back later.)

2.25. Open-up `MainWindow.cs`. You will see that this is a *partial* class. This means that the definition of this class is split across several files. This is conventional in GUI applications, allowing for one of the files to be generated by a GUI design tool.

The tool that comes built into MonoDevelop / Xamarin Studio is called Stetic GUI Designer. This name is for your reference only; the designer is *integrated* into the IDE.

To start it, simply tab over to the "Designer" rather than "Source" when having the `MainWindow.cs` open. (Tabs are at the bottom.)

2.26. Use the designer to add a Drawing Area to the window. A Drawing Area is a GUI widget that you can draw on. On your right-hand side you will find a toolbox. Let the drawing area fill the window (default behaviour).

2.27. Rename the drawing area as `darea`.

*Hint:* You need to modify the widget "properties".

2.28. For now, when you start the application, you see the same, boring, gray colour. To actually draw something on the drawing area, we need to attach an event handler to it's expose event: the event called whenever the drawing area has to be (re)drawn on the screen.

Add the following line after `Build()` in the `MainWindow` constructor:

```
this.darea.ExposeEvent += this.OnExposeEvent;
```

Then, declare the following method:

```
protected void OnExposeEvent (object sender,
    ExposeEventArgs a)
```

To set the background to black:

```
this.darea.ModifyBg (StateType.Normal,
    new Gdk.Color (0, 0, 0));
```

2.29. Try running the application. It should result in a black window.

## 3 Geometry II (30%)

The following tasks expand on the Geometry project initialized above. As you progress, you will want to test your code. To do this, we suggest to gradually compose a long list of `Console.WriteLine` calls in the entry point of your TUI project. Don't delete old calls to make sure you don't break old functionality as you make modifications.

### Vertex (15%)

3.1. Let `Geometry.Vertex`[1] be a struct (it is currently a class). The struct should represent a two-dimensional vertex, exposing two fields of type `int`: `x` and `y`. The struct should also have just one constructor:

`Vertex(int x, int y)`
Initializes a new vertex with the given coordinates.

3.2. Override the method `ToString` for `Geometry.Vertex` to yield the string `(x, y)`, where `x` is the string-representation of the field `x`, and similarly for `y`. How can you test that this works as intended?

3.3. Overload the operators `==` and `!=` for `Geometry.Vertex`.

In your report, argue that your implementation meets the guarantees required for every implementation of `==` and `!=`:

For any `v` and `w`, `(v != w) == !(v == w)`.

*Hint:* One of the operators should use the other in its implementation.

3.4. Override the `Equals` method for Geometry.Vertex.

This method must (eventually) use the `==` operator above.

In your report, argue that your implementation meets the guarantees required for every implementation of `Equals`[2] (except for those relating to the following task).

3.5. Override the `GetHashCode` method for Geometry.Vertex.

The implementation does not have to take "hashing" into account, but it must adhere to the following property:

```
v.Equals(w) ==> (v.GetHashCode() == w.GetHashCode())
```

You should read `==>` as a left-to-right implication (i.e., if `v` equals `w`, then they have the same hash code, but if they have the same hash code, they are *not necessarily equal*).

[1] `Geometry.Vertex` is a *fully qualified name*. It means that the struct/class `Vertex` must be declared under the `Geometry` namespace. Due to our style guide, this also means that the struct/class should defined in a file `Vertex.cs` inside the Geometry project.

[2] See also `https://msdn.microsoft.com/en-us/library/ms173147(v=vs.90).aspx`.

### 3.1 Polygons (15%)

To be announced after the second C# lecture.

## 4 Testing (30%)

In the previous section (Section 3 on page 5), we suggested that you test by gradually composing a long list of `Console.WriteLine` calls in the entry point of your TUI project.

Although such test results might be comprehensible to you, they are not likely to be so to anyone else.

4.1. Convert all your tests to yield a boolean (rather than rely on the overloading of the `Console.WriteLine` method). (For now, you can just use `Console.WriteLine` to report the booleans.)

4.2. Create a class `TUI.Tester` that maintains a state enum, indicating whether the tests conducted thus far have been successful or not.

Define the enum as follows:

```
public enum StateType { Successful, Failed };
```

4.3. Expose the value of the enum via a get-only property `State`.

4.4. Let the `TUI.Tester` have the following public instance method:

```
void Test(object actual, object expected, string message)
```
Tests if `actual.Equals(expected)`. If not, the state is changed to `Failed`, and `message` is written to standard error[3].

4.5. Let your entry-point begin with the following line:

```
var tester = new Tester ();
```

Followed by a long number of calls to `tester.Test`, but ending with:

```
Console.WriteLine (tester.State);
```

## 5 Submission (20%)

You should submit the following via Absalon:

- A ZIP archive containing *just* a folder containing your solution.
- A short PDF report documenting your solution.

[3]Use `Console.Error.WriteLine` instead of merely `Console.WriteLine`.

## Getting Started with our LaTeX Template (5%)

Reporting is a standard part of software development in practice.

LaTeX is one of the most versatile frameworks to this end. You should be familiar with LaTeX from your previous courses, and you will find it easier to produce high-quality documents going forth, if you work on your skill. Furthermore, good use of LaTeX is a good exercise in software development.

This is why we provide a LaTeX template for this course, and make it an explicit *requirement* that you use LaTeX as the binding framework for all of your reports. Our template is optional. We highly recommend it as a starting point, but you are free to explore the art of programming (LaTeX) at your leisure.

5.1. See our online guide to get started:

`https://git.dikunix.dk/su17/Guides/blob/master/LaTeX.md`

## Report (10%)

5.2. Write a report.

Your report should at least:

- Give an overview of (the extent of) your solution.
- Discuss the non-trivial parts of your implementation.
- Discuss your design decisions, if any.
- Disambiguate any ambiguities in the exercise set.
- If you deviate from the exercise text, tell us where, and why.

## Refactoring (5%)

5.3. To keep your TA happy, and receive more valuable feedback, you should:

- Make sure the code compiles without errors and warnings.
- Make the code comprehensible (perform adequate renaming, separate long methods into several methods, add comments where appropriate).
- Make sure the code follows our style guide.