Practice Minis

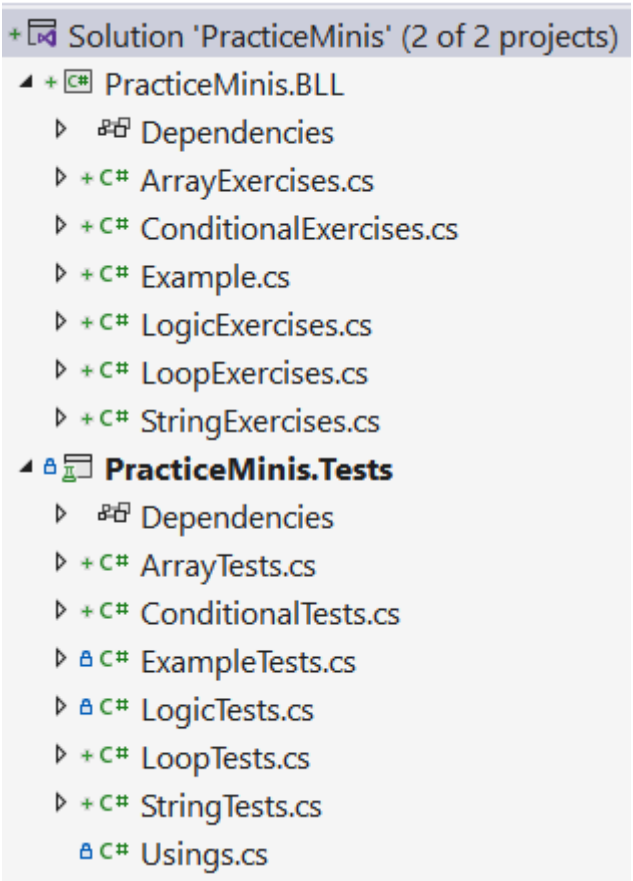
As I've stated during the course, learning the fundamentals is really important and learning to code takes a lot of practice. With this in mind, I have created dozens of practice "minis" which can be solved with small bits of code. Each of the exercises exists in a method and has unit tests running against them to automatically check your solutions for correctness.

Now, I realize that we have not covered unit tests or methods in this course, but fortunately you do not need to understand them to follow the pattern and complete these minis. In these instructions I will walk you through what you need to do to effectively use these exercises in Visual Studio. Not only will this reinforce your learning, but you will get an early experience in the test runner tool, which is used by professional developers!

Opening the Solution

To get started, navigate to the Capstone/PracticeMinis folder where you downloaded the course materials. In it, you will find a solution file **PracticeMinis.sln**

If you double click it it will open in Visual Studio. From here you can use the solution explorer to see that there are two projects: **PracticeMinis.BLL** and **PracticeMinis.Tests**.



The only files we need to be concerned with are the exercise files in PracticeMinis.BLL which are:

- ArrayExercises.cs
- ConditionalExercises.cs
- LogicExercises.cs
- LoopExercises.cs
- StringExercises.cs

Writing Code for an Exercise Mini

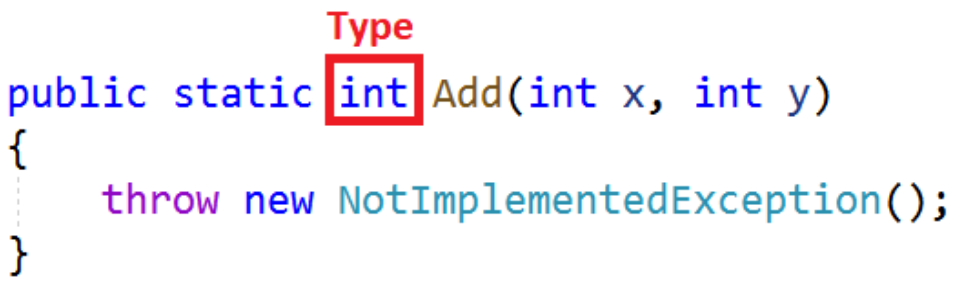
When you open a file, you will be presented with a list of methods with comments for the instructions of the code you should write. We will use the Example.cs file as a demo. Here are the contents of the file:

```
public class Example
{
    /* Given two numbers, x and y, return their sum */
    public static int Add(int x, int y)
    {
        throw new NotImplementedException();
    }
}
```

In the comment, we see that "Given two numbers, x and y, return their sum". Then we see a method with two parameters, x and y.

For now, you just need to understand that parameters are variables that allow data to be passed into a method. You can't see any data here in this file, and that's ok, just trust that x and y will have data when the tests are run.

What you need to do is use the **return** keyword to send a value back from this method of the appropriate type. The type is always right before the name of the method. In this example, the type is int.



So, given two numbers, x and y, return their sum. We know how to calculate a sum, so all we need to do is replace the "throw new" statement with our code, leaving the code block (curly braces). Here is the solution:

```
public class Example
{
    /* Given two numbers, x and y, return their sum */
    public static int Add(int x, int y)
    {
        int sum;

        sum = x + y;

        return sum;
    }
}
```

This is no different than writing code as we have in the course in our program.cs file, except that now we have some pre-existing variables (x and y) to use and we have to use the return keyword to send back an appropriate value that solves the exercise. You can write as much or as little code as required, you just have to use the parameters provided.

Each file will have many mini exercises each with their own instructions in the comments.

Examining the Test Code

Now all we need to do is run the test. If you want to look at the test code, you can open the corresponding *Tests file, but you do not have to in order to complete the exercise. For education sake, let's take a look at the ExampleTests.cs file:

```
public class ExampleTests
{
    // test case values are passed in to the parameter variables, x, y, and expected
    [TestCase(1, 1, 2)]
    [TestCase(5, 10, 15)]
    public void AddTest(int x, int y, int expected)
    {
        int actual = Example.Add(x, y);

        // assert, verify correctness
        Assert.That(actual == expected);
    }
}
```

I have left some comments in here to help explain what is going on, but again, you do not need to understand this now. Some people like to look though, so I will explain what is happening.

The AddTest() method has code to test our Add mini from the Example.cs file. Each [TestCase] contains the values that will be used by the code that we wrote as well as the value of the correct return. So looking at the first case:

[TestCase(1, 1, 2)]

- The x parameter variable will be provided a value of 1.
- The y parameter variable will be provided a value of 1.

- The test will check if we return a value of 2.

Similarly for the second case:

[TestCase(5, 10, 15)]

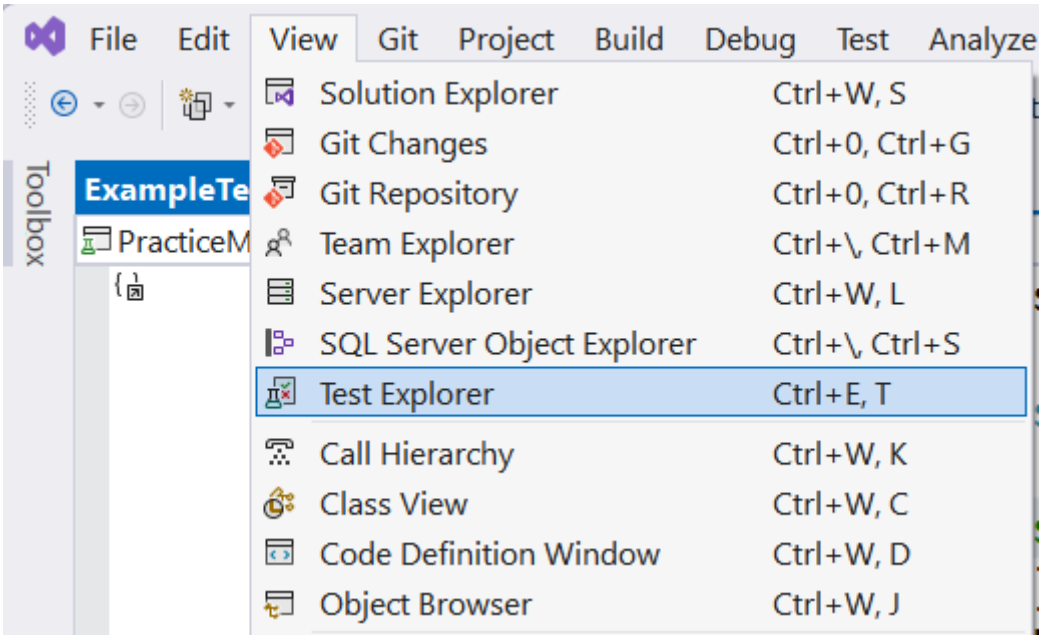
- The x parameter variable will be provided a value of 5.
- The y parameter variable will be provided a value of 10.
- The test will check if we return a value of 15.

If the correct values are returned from the code we wrote, then both tests will be colored green in the test runner. If any of the values returned are incorrect, then the failing tests will be colored red in the test runner with a message of what value was expected, and what we actually returned.

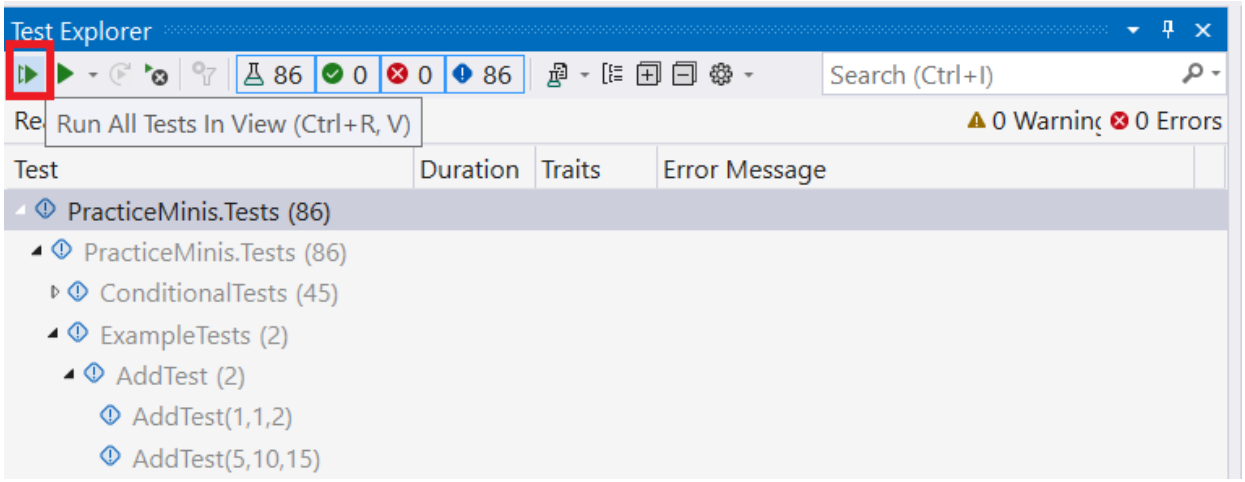
Again, you should not modify the test files at all, they will work as-is.

Running the Tests

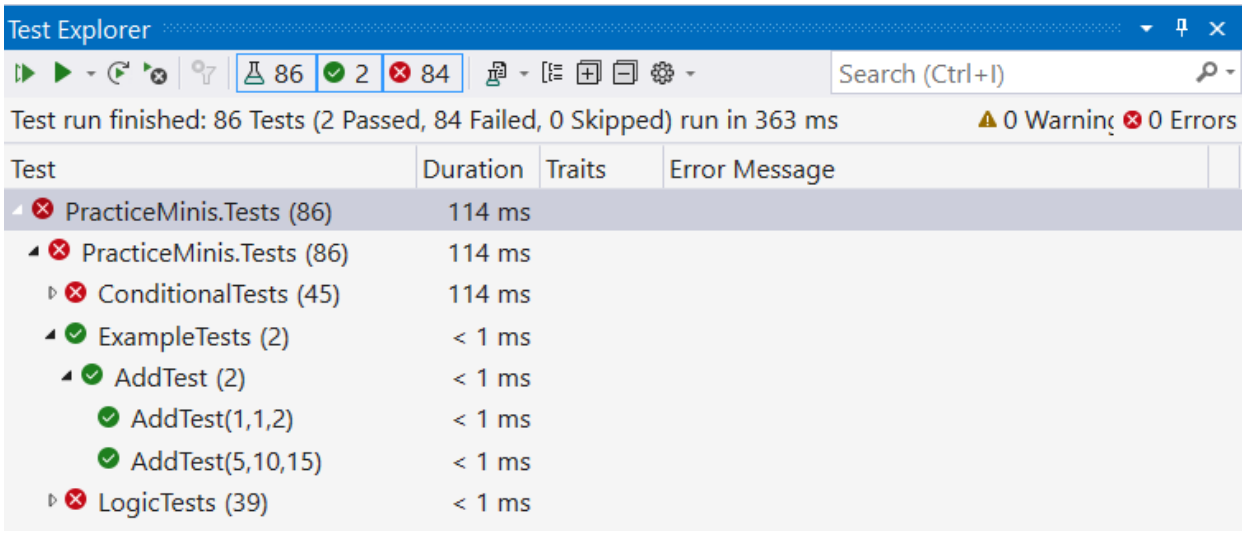
In Visual Studio, there is a tool called the Test Explorer, which can be opened from the View menu:



Once open, you will see a window pane with a list of all the tests. The easiest way to run the tests is to click the run all tests button which is a green play button, highlighted in red in the top left of the explorer.

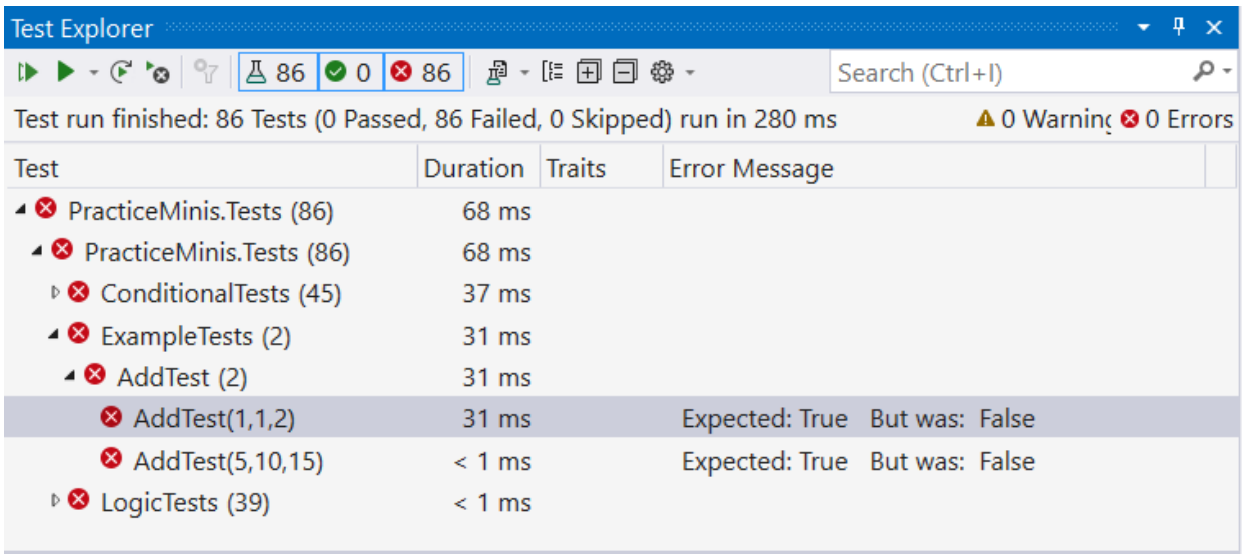


You can also see in the tree view under ExampleTests there are 2 tests, one for each [TestCase] from the ExampleTests.cs file. When we click the button, the tests will run and we get our results:

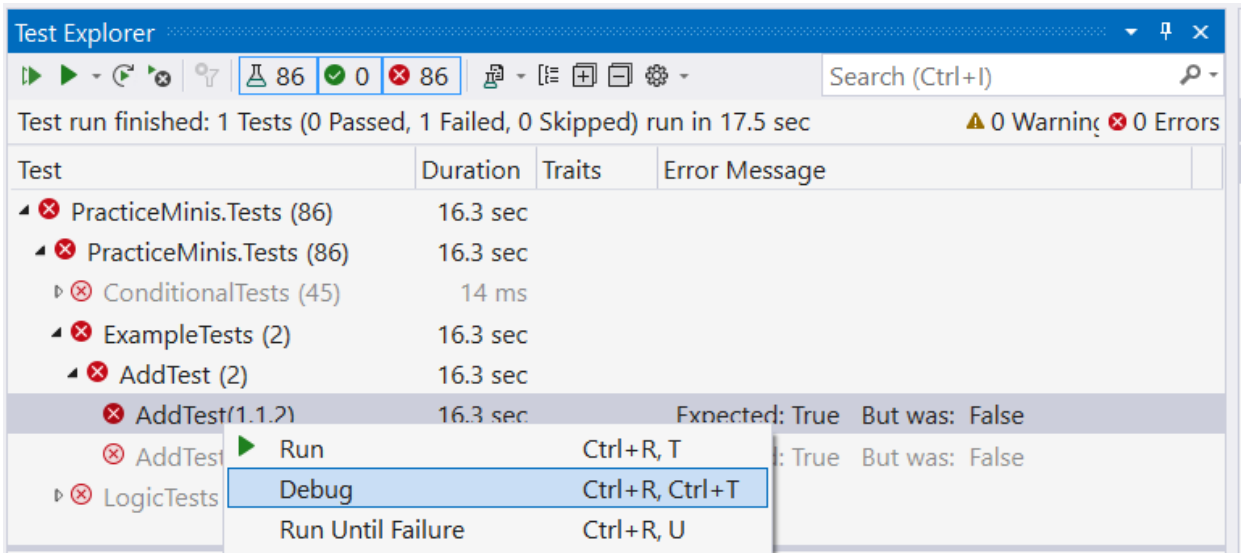


Our AddTest tests passed and turned green. Naturally since we have not completed the code for the other exercises, they are all red. This allows you to chip away at the mini exercises until all of the tests turn green.

Now, let's say we write the Example.cs code incorrectly, by subtracting instead of adding. Our test explorer results would then look like this:



This means the actual value from our code did not match the expected value from the test case. This isn't all that helpful, but luckily we can put in breakpoints and debug our code just like we demonstrated in the course. All you have to do is start debugging for the specific test you want to look at by right clicking it in the test explorer and choosing the debug menu option:



Conclusion

These minis should take a good amount of time to complete. They are designed to challenge you and stretch your understanding of the fundamental types and control structures of the C# language. Don't get frustrated if it takes many attempts to get a mini correct and don't be afraid to ask for help or use internet resources for inspiration!