Full Stack C# .NET

Unit Testing

Download PDF Copy

Wait... testing?

In software development, it is absolutely imperative that the software that is deployed and being used is working exactly as you and the business intends it to work.

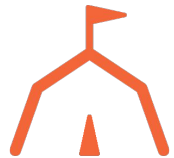
The more complex and used the software becomes, the greater the chance that bugs will sneak into your code base.

Fortunately, industry standards and best practices have been formed to help mitigate these bugs. The following slides will discuss some of the most recognized forms of testing as well as a deep dive on how to unit test.

Types of testing

- Manual Testing
- Automation Testing
- Unit Testing
- Integration Testing
- Smoke Testing
- Regression Testing
- System Testing
- Functional Testing
- Acceptance Testing
- Load Testing

As you can see there are several different types of testing (not all covered here). Each slide will give a high level overview of what these each type of testing is used for.



Manual Testing

As the name suggest, this type of testing is manually done by a Quality Assurance tester or an end user. It's important to know that manual testing is usually used to makes sure the software is behaving as intended for a production (final) environment.

Automated Testing

Automated testing is a set of scripts and premade tools (more software) used to automatically test your software.

Automation VS Manual Testing

When testing software, you will typically see a combination of manual and automated testing, but as technology and toolsets improve, automated testing is becoming the norm.

Although automated testing is becoming more mainstream for all types of testing, it is an investment in time and money. This is why you will see a hybrid.

Unit Testing

The most basic type of testing, where individual units or components are tested during the development (coding) process of an application. Unit testing isolates a part of the code and verifies its correctness.

In C#, unit testing is typically testing the public (or internal) methods of a class.

Integration Testing

Testing where software modules are integrated logically and tested as a group. Usually, full software projects have multiple modules, coded by different developers.

Integration testing is focused on data communication between these modules.

Smoke Testing

This is a build check to make sure major functionality of your code functions before running a major functional or regression test.

This saves time and resources on more time consuming testing processes, and is commonly used as applications grow larger.

Larger apps mean more surface area for errors... even build errors.

Regression Testing

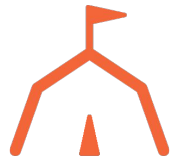
This type of testing is to make sure that any new code changes does not break older code. A common technique would be to run your test before and after commited changes.

This strategy is very useful as business requirements change; you can also use this with automated integration test to make sure one app or other app changes does not break older code.

Functional Testing

This is a type of software testing that verifies that each function (not code) of the application operates in conformance with functional requirements.

- Testing the main functions of an app
- Basic usability testing of the system. Makes sure the user can navigate the app as intended.
- Checks user accessibility
- Checks for suitable error conditions, i.e. if things go wrong, we are letting the user know in the most appropriate/desired way.



System Testing

Full system testing involves testing software for the following:

- Testing full integrated apps including internal and external peripherals in order to see how each component will interact with the entire system.
- Verify through testing of EVERY input to check for desired outputs
- Testing the users experience

Full system testing may be done by a different team or a 3rd party agency. Full system testing is to make sure the software is functioning as expected for production.

Acceptance Testing (UAT)

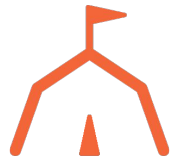
Acceptance testing, or User Acceptance Testing (UAT) is when the main user or clients use your application and test the app to make sure it meets business requirements.

If test pass, then the user has agreed to the software that will be delivered. This is usually the final testing before full production deployment.

Summary

We have covered testing and different ways of testing. You should now be familiar with what goes into testing and you should now be aware of the different strategies that exist.

Many different testing techniques have been discussed; this course will mostly focus on Unit Testing, some integration testing, and a software development principle called TDD (Test Driven Development).



TDD & XUNIT

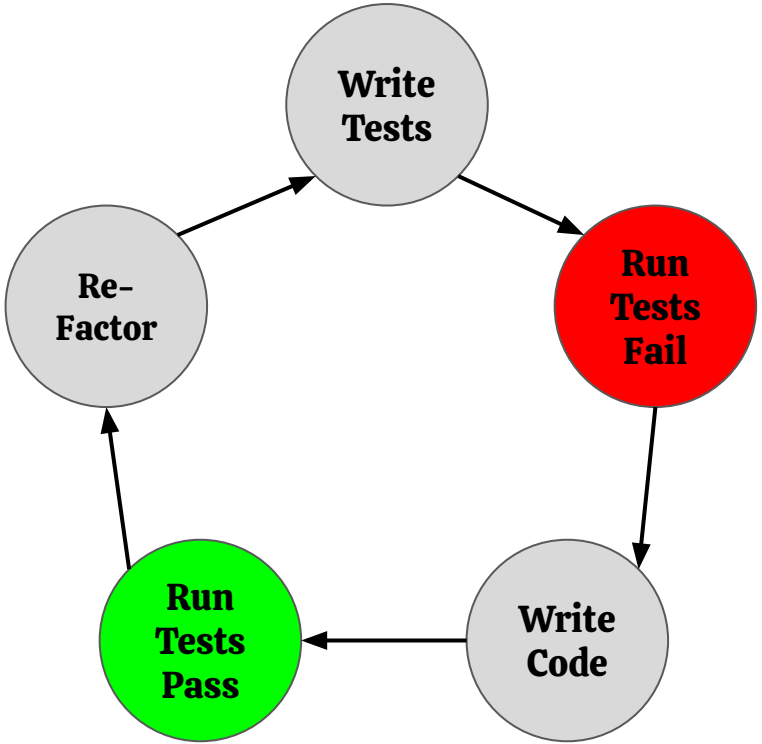
Test Driven Development

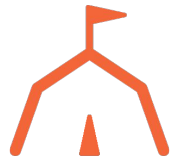
What is TDD?

Test-Driven Development is a method of developing software that involves a very quick cycle. It stresses simplicity of design as well as speed of implementation.

Basic TDD Steps

1. Write Tests
2. Run Tests
3. Write Implementation Code
4. Run Tests
5. Refactor





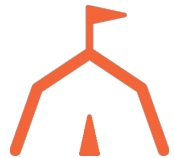
TDD Best Practices

1. You are not allowed to write any production code unless it is to make a failing unit test pass.
2. You are not allowed to write any more of a unit test than is sufficient to fail; and compilation failures are failures.
3. You are not allowed to write any more production code than is sufficient to pass the one failing unit test.

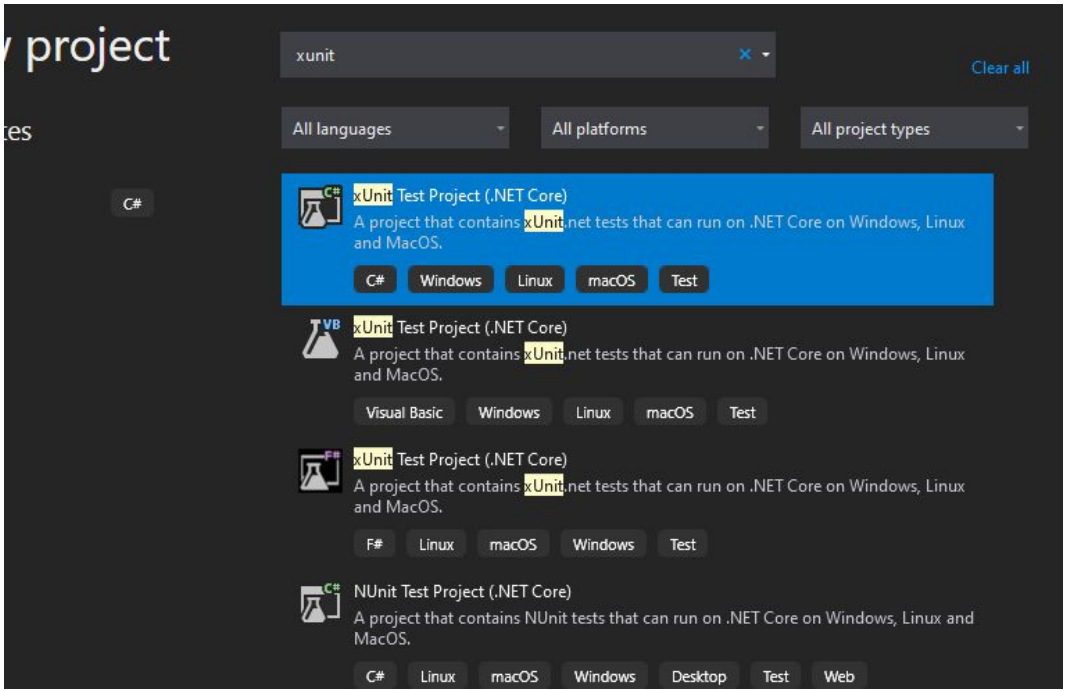
Approaching TDD

TDD is not as focused as you may think on the tests themselves. It's more about having the confidence to reorganize our code knowing that all the previously completed features are working properly.

Getting Started!



Create A Project



- Open up visual studio
- Select Create New Project
- When looking for the type of project, select xUnit Test Project (.NET Core), as displayed on the left.
- Name project and save in desired location (continue with the same steps as making a console app)

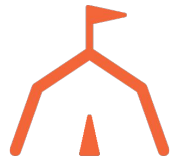
Unit Test Template

```
using System;
using Xunit;

namespace TestingExample
{
    0 references
    public class UnitTest1
    {
        [Fact]
        0 references
        public void Test1()
        {

        }
    }
}
```

- Default template, unit test class with testing method.
- Fact Attribute “[Fact]” tells the testing compiler and the testing framework (XUnit) that this method is a unit test.
- Methods are where your test will be written

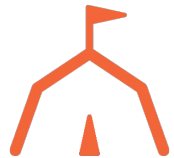
Unit Testing Conventions

- For Each class you are testing, create a testing class for that specific class.
  - Ie. if you have a Math class with math related methods, create a Math Unit test class
- Keep your tests very simple and lean, we are testing logic not making new logic that needs testing! Use the Triple AAA unit testing standard.
  - **Arrange** - setup objects / variables that will be used for testing. The class that will be tested is usually named sut for “System Under Test”.
  - **Act** - run the method you are going to test and store the results in a variable.
  - **Assert** - run a built in Assert method, you will pass in the ACTUAL result of the test method and what YOU expect to make the test pass.



Unit Testing Conventions

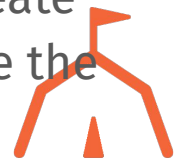
- Don’t repeat the same exact same test cases for different input/output, use the same testing method but with multiple input/output.
- Use TDD, write your unit test first, THEN create the actual class/methods you want to test!
- For OOP and inheritance, make sure to test both the parent and the child
    - If the parent is abstract, then test the public classes of the abstract class with your child classes.



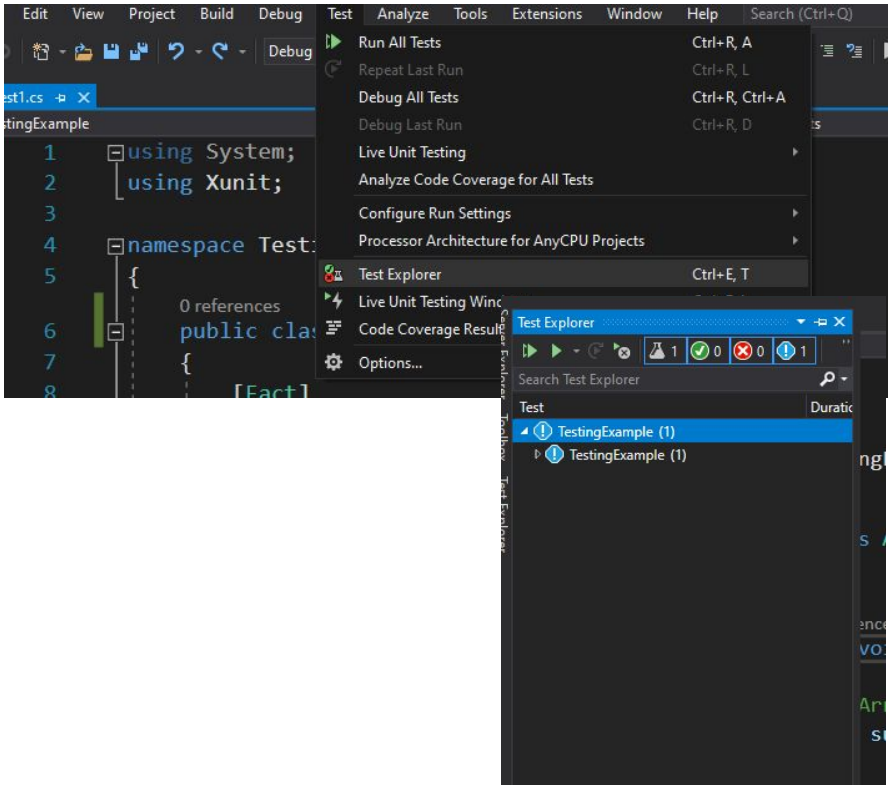
Basic Unit Test

```
0 references
public class ArithmeticTests
{
    [Fact]
    0 references
    public void TwoNumbersAddedUpWillSum()
    {
        // Arrange
        var sut = new Arithmetic();

        // Act
        var result = sut.Sum(2, 3);

        // Assert
        Assert.Equal(5, result);
    }
}
```

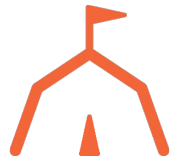
- We have a Testing class for a specific class we want to test, in this case Arithmetic.
- The Testing Method Name explains what unit functionality we are actually testing.
- Use the triple AAA standard to help format the test.
- Arithmetic has an error because it does NOT exist. In TDD, we create the test(s) first, fail, then write the code to pass the test



Running Test / Test Explorer

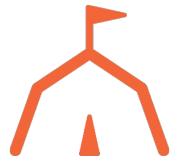


- Open up your test explorer.
  - Click on the Test drop down
  - Select Test Explorer
- Attempt to run your tests by clicking on the green arrow.
- Your test should NOT be able to run because you will have a compiler error because your class you wish to test does NOT exist yet.



Practicing TDD

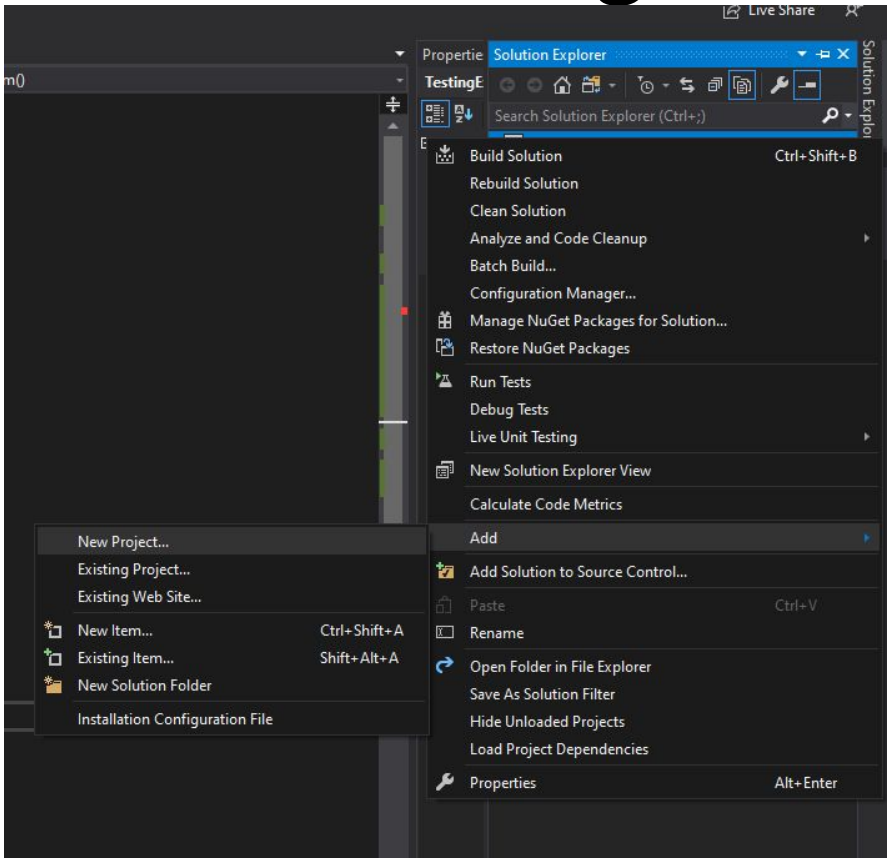
- As stated earlier, your test will NOT run because your class you wish to test does not exist yet.
- Create a separate project for your application, this project can be whatever you want, though library projects are the most common. (we will be using a console project for this example for simplicity).
- Once the project is created, have the Test project reference the newly created project assemblies.
- In your new project, create a class that matches the test case.
- Create a method that matches the test case.



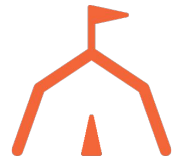
Practicing TDD

- Write logic that will cause you test to pass.
  - This step can potentially take the longest!
- In your test class, make sure you are referencing the SuT (System under Test) class
- Run the test runner, this time it should run
  - If your test fails, that means your logic does NOT meet the test case!
  - This is okay, simply refactor your method until the test passes
- Once the test passes, refactor some more to make the code as clean as possible
- Repeat these steps for every bit of functionality you want to add to your application.

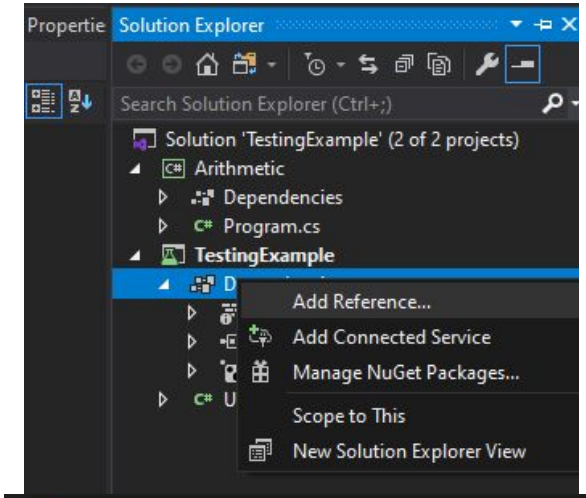
Creating New Console Project

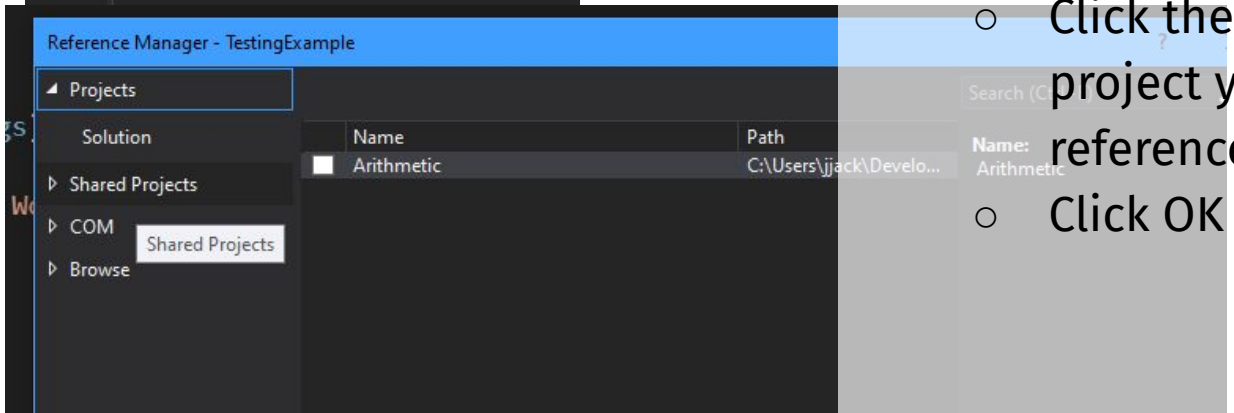


- In your solution explorer
  - Right click on your solution
  - Hover over add
  - Click “New Project...”
- In this window, select a C# dotnet core console application.
- Name it whatever you want.

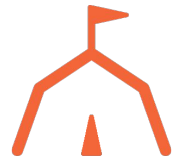


Referencing Console Project





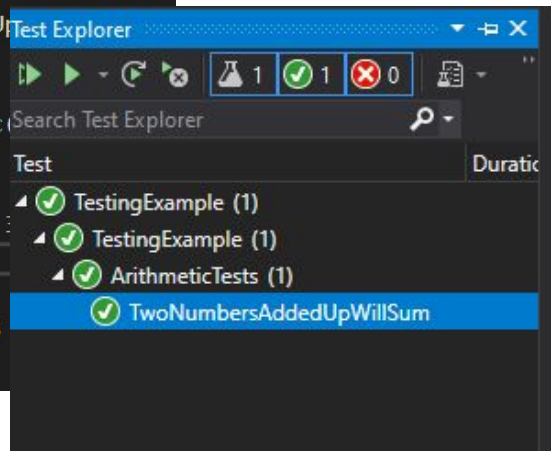
- In your solution explorer
  - Under your test project, right click on Dependencies
  - Click “Add Reference”
- In the new window
  - Click on projects on the top left
  - Click the check box for the project you want to reference
  - Click OK



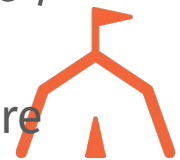
Create class / test class

```
public class Arithmetic
{
    0 references
    public int Sum(int numOne, int numTwo)
    {
        return numOne + numTwo;
    }
}
```

```
[Fact]
0 references
public void TwoNumbersAddedUp
{
    // Arrange
    var sut = new Arithmetic

    // Act
    var result = sut.Sum(2,

    // Assert
    Assert.Equal(5, result);
}
```

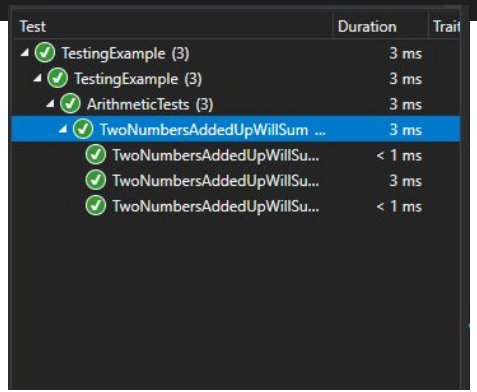


- Create your class in your console project; use the proper access (public)!
- Write the method you wish to test
- Make sure you are “using” the namespace that contains your class in the test class.
- Run the test runner.
- If your test passes, refactor your code to make it more readable / maintainable.
- Run the test again to make sure it’s still passing.



Multiple test cases

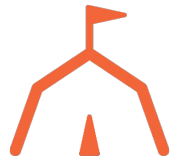
```
[Theory]
[InlineData(2, 3, 5)]
[InlineData(11, 13, 24)]
[InlineData(-5, -3, -8)]

0 references
public void TwoNumbersAddedUpWillSum(int numOne, int numTwo, int expected)
{
    // Arrange
    var sut = new Arithmetic();

    // Act
    var result = sut.Sum(numOne, numTwo);

    // Assert
    Assert.Equal(expected, result);
}
```



- If you want to test multiple inputs / outputs, you can with Theory + Inline Data.
- Refactor attributes on initial test to use Theory instead of Fact
- Under the Theory Attribute, add InlineData attributes that take in arguments.
- The arguments should match one-to-one to the arguments the test method can take
- Make sure to use the arguments in the test method instead of hard data

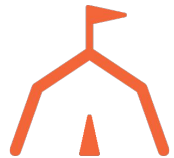
Approaching Testing

TDD is not as focused as you may think on the tests themselves. It's more about having the confidence to reorganize our code knowing that all the previously completed features are working properly.



Documentation Using TDD

Tests provide an excellent record of what the application should do as well as what it actually does.



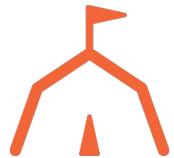
Visual Studio Test Explorer

Examples of methods in the assert class:

- Assert.AreSame: Checks that they are the exact same object, memory reference and all.
- Assert.AreEqual: Checks that objectOne.Equals(objectTwo).

Visual Studio Test Explorer

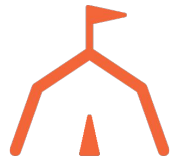
We can receive immediate feedback test results within VS.



TDD Exercises

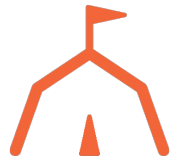
TDD is a process; it takes a lot of focus to think about the functionality you want to add to your software first and write the test cases for. The following slides will contain exercises to help get you in the mindset for for practicing test driven development. As a recap remember...

- Write your test cases first
- Write the classes / methods in a separate project afterwords
- Refactor your classes / methods until your test pass
- Refactor some more
- Rinse and repeat for ALL functionality



Arithmetics!

- Create a class that performs the 4 (+ 1) basic arithmetic functions.
  - Add
  - Subtract
  - Divide
  - Multiply
  - Remainder
- Make sure each method takes in two arguments
- Make sure to account for decimals

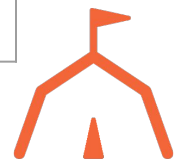


Temp Converter

A function that takes a temperature and a unit (either "C" or "F") and converts to that unit.

TEST CASES:

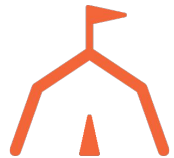
Input Temp	Input Target Unit	Output Temp
32 (f)	“C”	0(C)
68 (f)	“C”	20(C)
100 (c)	“F”	212(F)
-40 (c)	“F”	-40(F)



FizzBuzzBaz

Write a program that prints the number 1 to 100, but...

- Numbers that are exact multiples of 3, or that contain 3, must print a string containing “Fizz”
  - I.e. 9 -> “Fizz”, 31 -> “Fizz”
- Numbers that are exact multiples of 5, or that contain 5, must print a string containing “Buzz”
  - I.e. 10 -> “Buzz”, 53 -> “Buzz”
- Numbers that are exact multiples of 7, or that contain 7, must print a string containing “Baz”
  - I.e. 14 -> “Baz”, 71 -> “Baz”
- If more than one rule applies to a number, apply each applicable rule
  - I.e. 15 -> “FizzBuzz”, 35 -> “FizzBuzzBaz”



What should I wear?

Write a program that will tell you what to wear based on the temperature and type of event.

Event Type	Suggestion
casual	“Something Comfy”
semi-formal	“A polo”
formal	“A suit”

Temperature	Suggestion
Less than 54 degrees	“A Coat”
54 - 70 degrees	“A jacket”
More than 70 degrees	“No Jacket”

Summary

- You should be able to identify what TDD is.
- You should know how to develop software using TDD
- You should be able to write unit test using XUNIT