Getting Started with Unit Testing and Test-Driven Development

In most circumstances, you will be unit testing your own functions. However, sometimes you will be writing a function that somebody else wrote the specifications for; or, you will be unit testing a function somebody else wrote.

In every case, though, your function needs **a clear definition ahead of time**, that is, it needs a **clearly defined specification**:

- What the function takes as parameters, including the types
- What type the function returns
- How the function is expected to handle "incorrect" data or "error" conditions, meaning data that doesn't fit into the expected parameters.

For example, a function might take an integer as a parameter, but only numbers greater than 0. In that case, you, or your team, or your project manager, need to clearly state what the function will do when anything *else* is passed in. Will it simply throw an exception? Or will it return some special value? Those are the two most common ways, but regardless, the behavior for an error must be clearly defined up front so that you, as the coder of the function, or as the user of the function, or both, know what to expect.

Tip: When testing a function somebody else wrote, base your tests strictly on what you see in the function's signature and specification and do not assume anything else.

If a function's description does not state, for example, how it will handle error conditions, then *don't make any assumptions*.

For example, suppose a function takes two double parameters and is going to divide the first by the second. But if the function specification does not provide any information on how it would handle an error if the second parameter is a zero, then you do not need to test that situation.

In other words, suppose the function is this:

```
public static double MyDivision(double X, double Y)
```

And the specification states this:

> This function returns the value of X divided by Y.

Then do not bother testing 0 for Y, because the developer did not state in the documentation how the function responds to an error condition such as that.

But if this is the case, and somebody else on your team wrote the function, you will likely want to ask for clarification and request the team member update the specification and clarify what should happen in the event of an error. Does the function throw an exception? Does it return 0? Or perhaps some other value?

Tip 2: What if you wrote the function yourself and you're testing it?

This is a more common scenario, because typically you will write the specification for your own function, and write the function itself, as well as its tests. Now you have full control over it. In that case, you should **clearly define how the function handles bad input or error conditions, and document accordingly.**

Let's continue with the same MyDivision function. Suppose you have written the function and documentation the same was as above:

> public static double MyDivision(double X, double Y)
>
> "This function returns the value of X divided by Y."

Somebody else using this function will not know how to handle the situation of passing a 0 for Y. And further, how will you know what a test result passing 0 should do? You have some choices. You need to modify the documentation. You could modify it as follows:

> "This function returns the value of X divided by Y. **If a 0 is passed for Y, an exception will be thrown.**"

Or, you might decide the function should handle such a situation perhaps like so:

> "This function returns the value of X divided by Y. **If a 0 is passed for Y, the function will return 0.**"

Now you have a case that you must provide a test for both "good" values as well as "bad" values, which in this case would be 0 for a "bad" value. And because you clearly stated what the function should do when a 0 is passed in, you can test for that expected result, such as an exception or simply a 0 returned.

Tip 3: But regarding math functions

Although math functions are often used for practicing testing, they're not very realistic for a lot of reasons. First, most math functions we encounter really only need to be tested for a couple cases. If, for example, you have a function that calculates the value of x + 5, then you really only need to test it with one number. Try, for example, 10, and see if you get back 15. If you don't, then it means you coded it wrong, and it's going to fail no matter what you put in. if you do get back 15, then clearly it works and will certainly work for every other integer you pass into it, assuming you used a very simple calculation such as directly coding x + 5.

Soon we'll be using testing to build up our functions sequentially, an approach called Test Driven Development. This works in a lot of business cases, but is simply not realistic in a scientific setting. Mathematicians do not use test-driven development to come up with a formula to calculate prime numbers, for example. They work at the whiteboard and come up with different formulas that are far more advanced than those of us who don't work as mathematicians would likely understand and once they have a potential formula in place, then they will test it. This is, in a sense, test driven development, but it requires advanced knowledge in a required area, which we, as coders, may not possess.

Instead, our Test Driven Development will usually focus more on business scenarios.

For the rest of this chapter, we will focus on how to test before we explore true Test Driven Development.

Example 1

Let's consider a function that takes a string as input and returns a string. Suppose you are testing a function with a clear specification such as this::

Talk Function

```
public static string Talk(string animal)
```

This function takes as a parameter the name of an animal, as a string, and can be either **cat**, **CAT**, **dog**, **DOG**, **cow**, or **COW**. The parameter must be all uppercase or all lowercase. If the parameter is **cat** or **CAT**, the function will return **MEOW**. If the parameter is **dog** or **DOG**, the function will return **BARK**. If the function is **cow** or **COW**, the function will return **MOO**. In all other cases the function will return **null**.

Please list at least six test functions. Show the parameter and expected return value. The following are a couple to get you started; please provide at least six more. You should include tests that return a correct response (such as MEOW) as well as tests that return incorrect responses.

Parameter	Expected Return
"cat"	"MEOW"
"Cat"	null
"hello"	null
null	null

After completing the above exercise, ask yourself this question:

> Is it possible to list **every** test that returns one of the expected return values, e.g. MEOW, BARK, MOO, or null?

Let’s consider each of these. How many ways can you return MEOW? There are only two ways that can result in MEOW. Please list them here:

Now how many ways can you return BARK? There are two possible ways. List them here:

How many ways can you return MOO? Again, there are two possible ways. List them here:

So far we have six inputs that we must test, because we know there’s a limited, fixed number of ways to return these three values, MEOW, BARK, and MOO.

But what about null?

Because any other possible parameter will return null, that means there are infinitely many ways to return null. We can’t do an infinite number of tests. So what do we do? First, we need to draw attention to some special cases. We allow either CAT or cat as a parameter to return MEOW. But we don’t allow mixed case, as in Cat or cAT. So we should try at least one of these. In theory, there’s a limited number of cases for these, but realistically we can probably just try one or two. Similarly with DOG and COW; we could test at least one or two of mixed cases of those as well, but that might also be more than we need. So here are a few we could include in our tests:

"Cat"

"cAT"

"Dog"

"dOG"

"Cow"

"cOW"

Next, we should consider other words. What other words are possible? Pretty much anything. But as coders we know that **all** other cases will likely be the result of **a single “else” block** in the function. So we might just try a couple extras just to be safe, such as:

"Hello"

"World"

And finally, there are two very special cases we should always try when dealing with strings, specifically the empty string and null:

""

null

Looking back at the above, you can see we came up with at least sixteen test cases. Is that how many we should always strive for? No; it will vary with each function we're testing. But we should always strive for this:

> **Test as many of the known cases as we can, or, when possible, all (e.g. cat, CAT, dog, DOG, cow, COW).**
>
> **Then test several "other cases" that most likely get handled in an else block, of which there may be millions or infinitely many. So pick a few words such as HELLO or World.**
>
> **Finally test specific edge cases: Empty strings, nulls. If our function takes numbers, consider testing 0 and 1 as possible edge cases, depending on what the function does. (We have to think intelligently about the function.)**

Let's look at the edge case concept in more detail next.

Edge Cases

Edge cases depend on the function and the types it can take. With strings, our edge cases are usually the empty string and null.

With numbers, the edge cases depend on what the function does, but will typically involve the number 0 and 1, which are always good numbers to include in the tests.

With objects, the edge cases will vary, but should always include null, which means there's no object at all. You might also include a case where the object exists, but its members are all empty, such as an empty string or the number 0.

Back to Exceptions

As always, if somebody else wrote the function and the specification states that it does not handle error conditions and will throw an exception for edge cases or any incorrect input, then you typically won't need to test for those.

For example, if the specification states that passing null will result in an exception, then you probably don't need to test for null. Or for example, the documentation for the Talk function might look like this, with an added note about all other cases:

> **Talk Function**
>
> `public static string Talk(string animal)`
>
> This function takes as a parameter the name of an animal, as a string, and can be either **cat**, **dog**, or **cow**. The parameter must be all uppercase or all lowercase. If the parameter is **cat** or **CAT**, the function will return **MEOW**. If the parameter is **dog** or **DOG**, the function will return **BARK**. If the function is **cow** or **COW**, the function will return **MOO**. *These are the only valid input, and any other input will result in an exception.*

In this situation, we really only need to test for the specified "good" values: CAT, cat, DOG, dog, COW, cow. We could, however, test if anything else throws an exception. But as before, we really only need to test a couple cases, because most likely all other cases are handled by a single else block.

We can use XUnit to test for exceptions; however, we need to know which exceptions to test for. We'll take that up later when we get into using XUnit to test.

Testing Practice

In each of these, list several tests, including parameters and expected results.

Do not attempt to code these functions! We will do that later. Also, do not write XUnit tests. Simply list in a notepad document the parameters and expected results.

WordJoin function

```
public static string WordJoin(string)
```

This function takes a single string parameter that contains words separated by a single space, such as "Hello there everybody".

It then combines the words together without spaces and capitalizes the first letter of each word. Thus "Hello there everybody" will become "HelloThereEverybody". The function will accept as a “word” any sequence of characters other than space. For words starting with non-alphabetic characters, no change for “uppercase” will be done. Thus the string “Abc %def !%” will return “Abc%def1%”. If either a null or empty string is passed in, the function will return null.

Coupon function

```
public static decimal CalculateCoupon(decimal purchase)
```

This function takes as a parameter the amount purchased, and calculates a coupon the customer should receive. When a customer purchases at least $10 up to and including $19.99, the customer will get a $2 coupon back. When a customer purchases $20 or more, the customer will get a $3 coupon back. If the customer purchases under $10, then no coupon will be given.

Thus, if the parameter is between 10.00 and 19.99 inclusive, the function will return 2.00.

If the parameter is 20.00 or higher, the function will return 3.00.

If the parameter is under 10.00, then the function will return 0.00.

Note: This function does not deal with tax calculations.

IntegersInString function

```
public static int[] IntegersInString(string nums)
```

This function takes as a parameter a single string consisting of integers separated by commas and returns an array of integers.

For example, the input string "10,3,6,0,-5,100" would return an array of integers consisting of 10, 3, 6, 0, -5, and 100.

Any string containing characters other than integers and commas will throw an exception of type System.FormatException. Thus a string such as “Hello,1,5” would throw FormatException.

IsFactor function

```
public static bool IsFactor(int X, int Y)
```

This function takes two parameters and determines if the second is a factor of the first. If the second is a factor of the first, it returns true. If not, it returns false.

Tip: For this final exercise, think carefully about the situation if the second parameter is zero. Technically, the specification does indeed say what to return when the second parameter is zero.

Building Tests in XUnit

Testing if two things are equal

[COMING SOON]

Testing if one thing is true

Consider this: You have built a function called IsEven, which returns true when a function is even, and false when it is odd. You want to then build some tests around it.

One test might pass 2 into the function, in which case you would expect to get true.

Although you might expect to use the Assert.Equal function and simply pass true for the expected value, XUnit in fact prefers you use a different function called simply True.

In other words, do **not** do this:

```
Assert.Equal(true, actual);
```

XUnit will run this, but it will issue a warning. Instead, call Assert.True like so:

```
[Fact]
public void TestIsEven()
{
	bool actual = Program.IsEven(2);
	Assert.True(actual);
}
```

Testing if one thing is false

Similarly, if you need to test if a result is false, you can use the False method. Again, with the IsEven function, we could pass a 3 in and expect a false:

```
[Fact]
public void TestIsEven()
{
	bool actual = Program.IsEven(3);
	Assert.False(actual);
}
```

```
}
```

Testing for null or not null

Just like when testing whether a single item is true or false, testing for null requires a different method. We do not call Assert.Equal and pass in null for the expected value; instead we simply call Assert.Null like so:

```
[Fact]
public void TestSomeFunction()
{
	string actual = Program.SomeFunction();
	Assert.Null(actual);
}
```

For example, you might have a function that returns a string under normal conditions, but returns null if invalid data was passed in (such as an empty string). You could include a test such as the above, intentionally pushing invalid data into your function, and verifying that it returns null.

There is also a NotNull function that you can call, although you might not need to use this one very often:

```
[Fact]
public void TestSomeFunction()
{
	string actual = Program.SomeFunction();
	Assert.NotNull(actual);
}
```

Testing Multiple Items with a Single Function

[COMING SOON - Theory/InlineData]

Testing for Exceptions

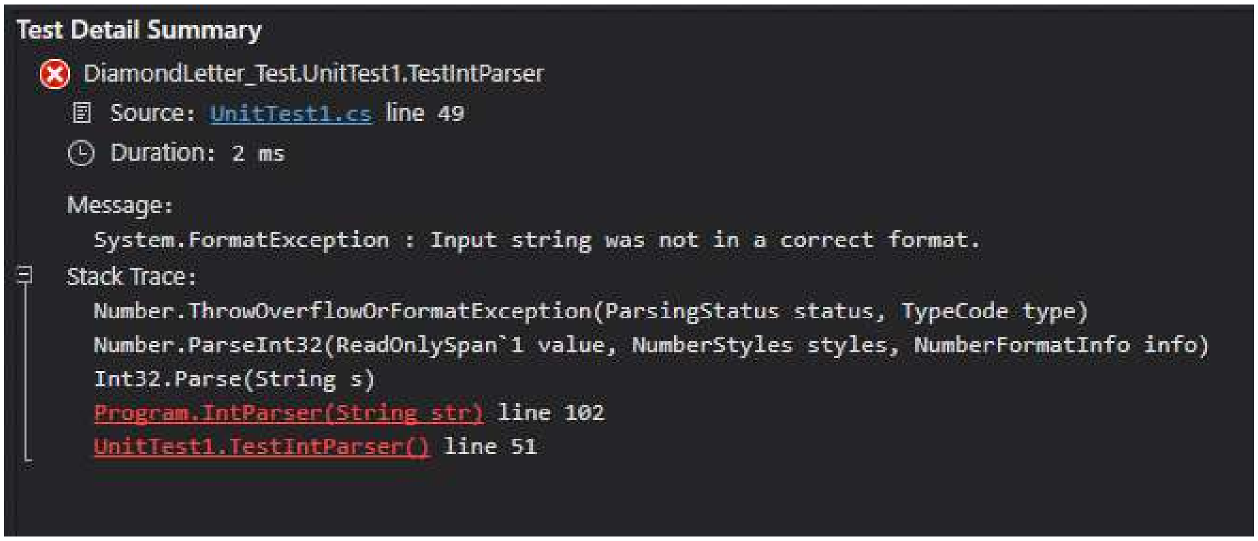
In order to test for an exception, you have to know what kind of exception you will be catching. For example, let's say you're writing a function that calls int.Parse. The int.Parse function throws an exception of type System.FormatException. Suppose this is the function you're writing and subsequently testing:

```
public static int IntParser(string str)
{
	return int.Parse(str);
}
```

(If you're coding along, you can just put this in your Program class to try it out.) This function, of course, really doesn't do much, simply calling another function, but it's enough for us to learn how to test for exceptions. In your XUnit code, add the following Fact function:

```
[Fact]
public void TestIntParser()
{
      int actual = Program.IntParser("3");
      int expected = 3;
      Assert.Equal(actual, expected);
}
```

When you run this test, you will see it passes as expected. But how do we test invalid data? What if we want to test IntParser("Hello")? Try changing the code above to exactly that. But… what should the expected be? For now, just leave it 3 so we can see how the test runner deals with exceptions. Run the test. Click on the test result and you'll see the following:



```
Test Detail Summary
  DiamondLetter_Test.UnitTest1.TestIntParser
    Source: UnitTest1.cs line 49
    Duration: 2 ms

    Message:
      System.FormatException : Input string was not in a correct format.
    Stack Trace:
      Number.ThrowOverflowOrFormatException(ParsingStatus status, TypeCode type)
      Number.ParseInt32(ReadOnlySpan`1 value, NumberStyles styles, NumberFormatInfo info)
      Int32.Parse(String s)
      Program.IntParser(String str) line 102
      UnitTest1.TestIntParser() line 51
```

Although this is in fact what we expected to see (an exception), this gets flagged as a **failing test**. But in fact, an exception is what we *expected* to see and indeed received, so technically this should be a *passing* test! So let's see how to fix this.

The XUnit library includes another assertion we can use called Assert.Throws. However, the syntax for using it is a bit tricky. We have to use what are called lambda functions. Modify your test to look like this:

```
[Fact]
public void TestIntParser()
{
      Assert.Throws<FormatException>(() => Program.IntParser("Hello"));
}
```

The Throws function requires a type (in the same way as when we create lists, such as List<string>). It then takes a parameter which is itself a function. Don’t worry too much about the syntax here, but know that the stuff inside the parentheses is an entire function itself containing no parameters and a single line of code:

```
() => Program.IntParser("Hello")
```

This function will run and we know it will throw an exception, because we’re passing Hello into IntParser.

So now run the test and you’ll see the test passes. We expected an exception, and we indeed received an exception. (If you want to be sure, try changing the "Hello" back to "3" and notice that now the function fails! It does not generate an exception, but expected one. Then change it back to "Hello".)

Learning TDD

Now let's explore how to use tests to build functions. This is called Test-Driven Development.

Dividing a problem into smaller problems.

Let's revisit this one and think about a function that we might write that this function would need, and then how we would test it:

WordJoin function

```
public static string WordJoin(string)
```

This function takes a single string parameter that contains words separated by a single space, such as "Hello there everybody".

It then combines the words together without spaces and capitalizes the first letter of each word. Thus "Hello there everybody" will become "HelloThereEverybody". The function will accept as a "word" any sequence of characters other than space. For words starting with non-alphabetic characters, no change for "uppercase" will be done. Thus the string "Abc %def !%" will return "Abc%def1%". If either a null or empty string is passed in, the function will return null.

In order to break this problem down, think about the basic things it needs to do to complete its job. To do this, consider its functionality in a bit more detail by thinking about how you might do this manually on paper.

Suppose you received this string: "Hello there everybody". Think about how you would do it manually on paper. You would look at each word individually, and you would change the first letter of each to uppercase, and then combine them all into one big long word. Think of each of those steps separately:

1. Gather each word individually.
2. Change the first letter of each word to uppercase
3. Combine the words together into one long word

In terms of programming, we can define these same three steps a bit more precisely:

1. Split the string into individual words
2. Capitalize the first letter of each word
3. Join the words back together

In order to code this, we would create a separate function for each of these three steps, and we will code and test each one individually. However, if we look at the .NET framework documentation for the string class, we can find pre-made functions that accomplish steps 1 and 3.

The first can be accomplished through the String.Split function, as shown here.
https://docs.microsoft.com/en-us/dotnet/api/system.string.split?view=net-5.0

The third can be accomplished through the String.Join function, as shown here.
https://docs.microsoft.com/en-us/dotnet/api/system.string.join?view=net-5.0

Therefore, of these three steps, we only need to code one function, one that capitalizes the first letter of a word. Then in our main WordJoin function we will call this new function. We'll call this new function Capitalizer.

In order to accomplish all this, we will divide our problem into these steps:

1. Code any "helper" functions and test each one individually. Make sure these functions work completely before moving to step 2.
2. Code the primary function, which in turn calls our helper functions. Test this function completely.

In our case, step 1 consists of coding our Capitalizer function. Step 2 consists of coding the primary function we're building, which is called WordJoin:

1. Code and test the Capitalizer function
2. Code and test the WordJoin function, which makes use of Capitalizer.

Tip: When you're building Capitalizer, you're **coding** it. Later, when you're building WordJoin, you're **using** the Capitalizer function and should not need to look at the code inside Capitalizer. Focus only on the task at hand!

We will do these next.

Create the Capitalizer Function's Specification

Before building this function, we need to formalize it by creating a complete specification. What parameters will it take? What type will it return? What will it do with invalid data being passed in?

Parameters: The function will take a single string as a parameter containing a single word in any case.

Returns: The function will change the first letter of the word to uppercase.

Invalid Data: Here we get to decide what we want to do. We want to make our function as easy to use as possible when we're calling it from WordJoin. According to the specification, WordJoin will return null if it encounters invalid data. So let's do the same with this function. Therefore, our specification will state: The function will return null when it receives invalid data.

Question: Should the function change the other letters to lowercase? Answer: The original specification for WordJoin did not say we should do that, so we won't.

We can now write part of the function. For now, simply have it return an empty string so that it compiles.

```
public static string Capitalize(string word) {
    return "";
}
```

Before we code the function, however, let's create the tests.

Create the Capitalize function's Tests

Put together a list of possible tests to send to this function. Include good tests (such as changing "hello" to "Hello". Also include strings that don't have letters for the first character, since that was part of the WordJoin specification. Finally, remember our edge cases: an empty string, as well as null. According to the specification, what do we expect the function to return in those cases?

Create the tests in XUnit. You may use the [Theory] attribute with several [InlineData] attributes to combine these tests into a single test function.

Run the tests. Most, or perhaps all, will fail. That's okay, as we haven't built the function yet. Now we can build the function.

Build the Capitalize Function

Inside the function, we will take the first letter of the word and capitalize it. We can access that letter through word[0], and then call ToUpper. (Tip: The char type has its own ToUpper function, just like the String type does.) How do we access the remaining characters? We could loop through them, but better might be a call to String's Substring method. You can read about it here. https://docs.microsoft.com/en-us/dotnet/api/system.string.substring?view=net-5.0

Spend time on this function itself, and force yourself to stay away from the original function, WordJoin. Focus your attention on this Capitalize function itself!

As you build the function, make sure it compiles, and test it out. As you add more code, test it out again. Keep coding, testing, coding, testing, until the function works.

After all the tests pass, including your edge cases, and you're confident the function works, you may shift your focus to the WordJoin function. At that point you will simply use the Capitalize function you just built.

Note that because we already have a specification for WordJoin, we don't need to write one. Let's begin coding WordJoin. But wait! First we need some tests.

Create the WordJoin Tests

This should be a very similar process as with Capitalize. First, let's get a basic function so we have something to call. Simply copy the earlier header, and add the curly brackets, and a return statement. This will go in your main program's code, not the XUnit code.

```
public static string WordJoin(string) {
    return "";
}
```

Then in our XUnit code we can start adding tests. As before, you can use [Theory] and [InlineData]. After you've created the tests, run them, and most or all should fail. And now we're ready to code the inside of the function.

Building the WordJoin function

Tip: Ideally Capitalize should work perfectly. However, it's possible that you forgot to test some things with Capitalize and you might run into problems. In that case you'll have to backtrack and work on Capitalize again, covering these additional cases you discovered that caused it to break.

Think back to the original three steps. We should now have three functions at our disposal to accomplish the WordJoin:

- String.Split (built into the .NET framework)
- Capitalize (you built this!)
- String.Join (built into the .NET framework)

Piece together your function using these three functions. Do you need to loop? The first function returns an array that you'll save into a variable and need to loop through. Will you need a second array or will you replace the individual members of the array? Think about either and see if any one is better than the other. Then you can piece the items in the array back together with the Join function.

Coding the Other Practice Exercises

For the other three exercises, you'll follow a similar approach. Do you need to break the problem up into a smaller problem? When you have smaller problems, is there already a .NET framework method you can use? And when there isn't, plan to code a separate function and provide all the tests you need for it.

More stuff

Here are a few final things that might help you along as you code the other three functions.

Tip: How to throw an exception

If your code needs to throw an exception, you can do so with a keyword called throw. Here's an example of a function that checks if a positive number is even or odd, and for non-positive numbers it throws an exception.

```
static bool PositiveEvenOdd(int num)
{
      if (num < 1)
      {
            throw new System.ArgumentOutOfRangeException();
      }
      if (num % 2 == 0)
      {
            return true;
      }
      else
      {
            return false;
      }
}
```