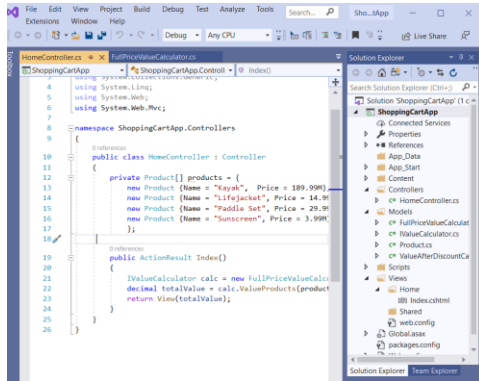


## Unit Testing

Download and unzip SimplifiedShoppingCartApp from Canvas. Note, although the zipped file is called **SimplifiedShoppingCartApp**, the actual project name when you open it is called **ShoppingCartApp**. But this is not the same application as the ShoppingCartApp that we worked with in the last lesson. I have simplified it.



Let's take a look around the application and see what's going on.

```
public class Product
{
    0 references
    public int ProductID { get; set; }
    4 references
    public string Name { get; set; }
    6 references
    public decimal Price { get; set; }
}
```

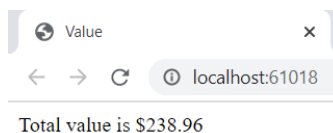
```
public interface IValueCalculator
{
    2 references
    decimal ValueProducts(IEnumerable<Product> products);
}
```

```
public class FullPriceValueCalculator : IValueCalculator
{
    3 references
    public decimal ValueProducts(IEnumerable<Product> products)
    {
        return products.Sum(p => p.Price);
    }
}
```

```
public class HomeController : Controller
{
    private Product[] products = {
        new Product {Name = "Kayak", Price = 189.99M},
        new Product {Name = "Lifejacket", Price = 14.99M},
        new Product {Name = "Paddle Set", Price = 29.99M},
        new Product {Name = "Sunscreens", Price = 3.99M}
    };
    0 references
    public ActionResult Index()
    {
        IValueCalculator calc = new FullPriceValueCalculator();
        decimal totalValue = calc.ValueProducts(products);
        return View(totalValue);
    }
}
```

Alter this application so that it has dependency injection. Now that you've done this a few times, it should be getting easier!

Run the application to make sure it is running correctly and that it is returning the sum of the prices.



## Unit Testing with Visual Studio

You may not have noticed but there are actually two implementations of `IValueCalculator` in the Models folder. The one that we haven't used yet is called **ValueAfterDiscountCalculator**.

```
1 reference
public class ValueAfterDiscountCalculator : IValueCalculator
{
    3 references
    public decimal ValueProducts(IEnumerable<Product> products)
    {
        throw new NotImplementedException();
    }
}
```

I have left the `ValueProducts` method in this implementation fairly empty for now. If the application tries to run it, a “Not Implemented” exception will occur that will remind us that we haven't put the code in there yet.

Eventually though, we will want the following behavior that will apply discounts.

- If the total is greater than \$200, the discount will be 10 percent.
- If the total is between \$100 and \$200 inclusive, the discount will be 5 percent.
- No discount will be applied on totals less than \$100.
- An `ArgumentOutOfRangeException` will be thrown for negative totals.

## What is Test-Driven Development? (TDD)

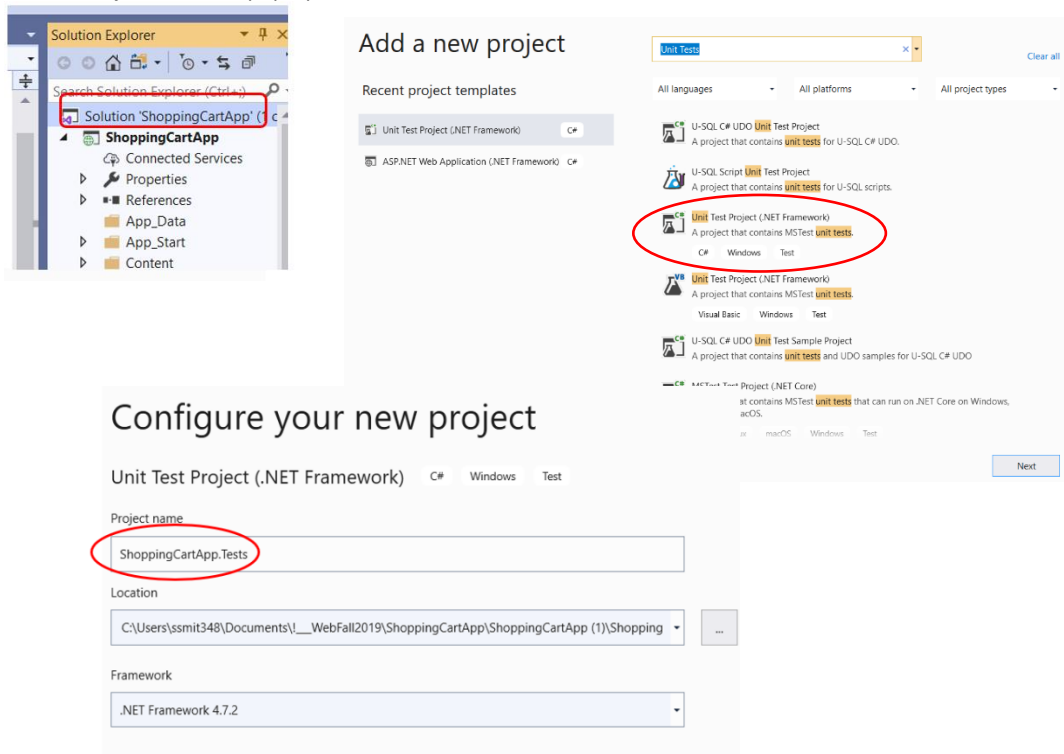
The Test-Driven Development (TDD) is a software engineering practice in which unit tests are written first, **before** the code you will test is actually written. When the unit tests are first run, they will always fail because you haven't written the code. Next, you write your code and check the tests again. Continue working on your code until all of your unit tests pass.

*“When the tests all pass, you're done.”*

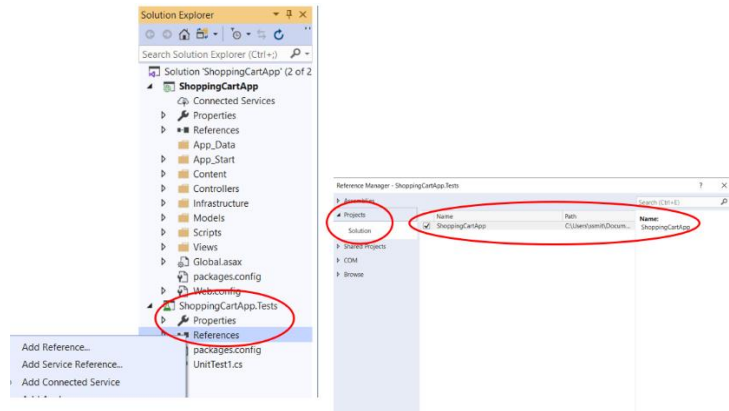
We will use this practice now. So we will write our unit tests first, before we write our code.

## Creating Unit Tests

Right-click the top-level item in the Solution Explorer (which is labeled **Solution 'ShoppingCartApp'**) and selecting **Add ► New Project** from the pop-up menu.



Right-click the References item for the ShoppingCartApp.Tests project in the Solution Explorer, and then select **Add Reference** from the pop-up menu.



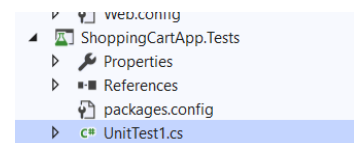
Download UnitTestCode.txt from Canvas, open using Notepad and copy and paste the code inside the UnitTest1 class which should now be found in your ShoppingCartApp.Tests project.

```
public class UnitTest1
{
    4 references | 3/4 passing
    private IValueCalculator getTestObject()
    {
        return new ValueAfterDiscountCalculator();
    }
    [TestMethod]
    0 references
    public void Total_Above_200()
    {
        // arrange
        IValueCalculator target = getTestObject();
        Product[] productsAtThreeHundred = {
            new Product {Name = "Product1", Price = 100M},
            new Product {Name = "Product2", Price = 200M}
        };
        // act
        var discounted300 = target.ValueProducts(productsAtThreeHundred);
        // assert
        Assert.AreEqual(270M, discounted300);
    }
    [TestMethod]
    0 references
    public void Total_Between_100_And_200()
    {
```

**Arrange (set up a scenario)**

**Act (attempt the operation)**

**Assert (verify the result)**



The behavior for the ValueProducts method is listed below and we'll start with the first line.

- If the total is greater than \$200, the discount will be 10 percent.
- If the total is between \$100 and \$200 inclusive, the discount will be 5 percent.
- No discount will be applied on totals less than \$100.
- An ArgumentOutOfRangeException will be thrown for negative totals.

```
public class UnitTest1
{
    private IValueCalculator getTestObject()
    {
        return new ValueAfterDiscountCalculator();
    }
    [TestMethod]
    public void Total_Above_200()
    {
        // arrange
        IValueCalculator target = getTestObject();
        Product[] productsAtThreeHundred = {
            new Product {Name = "Product1", Price = 100M},
            new Product {Name = "Product2", Price = 200M}
        };
        // act
        var discounted300 = target.ValueProducts(productsAtThreeHundred);
        // assert
        Assert.AreEqual(300M * .90M, discounted300);
    }
}
```

Now, we'll move to the next requirement.

- If the total is greater than \$200, the discount will be 10 percent.
- **If the total is between \$100 and \$200 inclusive, the discount will be 5 percent.**
- No discount will be applied on totals less than \$100.
- An `ArgumentOutOfRangeException` will be thrown for negative totals.

```
[TestMethod]
public void Total_Between_100_And_200()
{
    //arrange
    IValueCalculator target = getTestObject();
    Product[] productsAt100 = {
        new Product {Name = "Product1", Price = 50M},
        new Product {Name = "Product2", Price = 50M}
    };
    Product[] productsAt150 = {
        new Product {Name = "Product3", Price = 100M},
        new Product {Name = "Product4", Price = 50M}
    };
    Product[] productsAt200 = {
        new Product {Name = "Product5", Price = 100M},
        new Product {Name = "Product6", Price = 100M}
    };
    // act
    decimal discounted100 = target.ValueProducts(productsAt100);
    decimal discounted150 = target.ValueProducts(productsAt150);
    decimal discounted200 = target.ValueProducts(productsAt200);
    // assert
    Assert.AreEqual(100M * .95M, discounted100, "Discount for $100 is wrong");
    Assert.AreEqual(150M * .95M, discounted150, "Discount for $150 is wrong");
    Assert.AreEqual(200M * .95M, discounted200, "Discount for $200 is wrong");
}
```

Now, we'll move to the next requirement.

- If the total is greater than \$200, the discount will be 10 percent.
- If the total is between \$100 and \$200 inclusive, the discount will be 5 percent.
- **No discount will be applied on totals less than \$100.**
- An `ArgumentOutOfRangeException` will be thrown for negative totals.

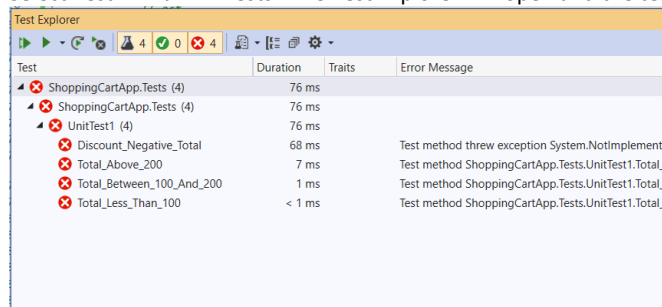
```
[TestMethod]
public void Total_Less_Than_100()
{
    //arrange
    IValueCalculator target = getTestObject();
    Product[] productsAtZero = {
        new Product {Name = "Product1", Price = 0M},
        new Product {Name = "Product2", Price = 0M}
    };
    Product[] productsAtFifty = {
        new Product {Name = "Product3", Price = 25M},
        new Product {Name = "Product4", Price = 25M}
    };
    // act
    decimal discounted0 = target.ValueProducts(productsAtZero);
    decimal discounted50 = target.ValueProducts(productsAtFifty);
    // assert
    Assert.AreEqual(0M, discounted0, "Discount for $0 is wrong");
    Assert.AreEqual(50M, discounted50, "Discount for $50 is wrong");
}
```

Now, we'll move to the last requirement.

- If the total is greater than \$200, the discount will be 10 percent.
- If the total is between \$100 and \$200 inclusive, the discount will be 5 percent.
- No discount will be applied on totals less than \$100.
- **An `ArgumentOutOfRangeException` will be thrown for negative totals.**

```
[TestMethod]
[ExpectedException(typeof(ArgumentOutOfRangeException))]
public void Discount_Negative_Total()
{
    //arrange
    IValueCalculator target = getTestObject();
    Product[] productsAtNegative = {
        new Product {Name = "Product1", Price = -10M},
        new Product {Name = "Product2", Price = 0M}
    };
    // act
    decimal discounted0 = target.ValueProducts(productsAtNegative);
}
```

Select Test ► Run All Tests. The Test Explorer will open and the tests will be run.



The tests failed because we have **no code** in the method yet.

Remember that we expected this to happen!

```
public class ValueAfterDiscountCalculator : IValueCalculator
{
    public decimal ValueProducts(IEnumerable<Product> products)
    {
        throw new NotImplementedException();
    }
}
```

← Nothing there!

Now we'll write the actual code that will figure the discounted values.

```
public class ValueAfterDiscountCalculator : IValueCalculator
{
    public decimal ValueProducts(IEnumerable<Product> products)
    {
        decimal fullPrice = products.Sum(p => p.Price);

        if (fullPrice < 0)
        {
            throw new ArgumentOutOfRangeException();
        }
        else if (fullPrice > 200)
        {
            return fullPrice * 0.9M;
        }
        else if (fullPrice > 100 && fullPrice <= 200)
        {
            return fullPrice * 0.95M;
        }
        else
        {
            return fullPrice;
        }
    }
}
```

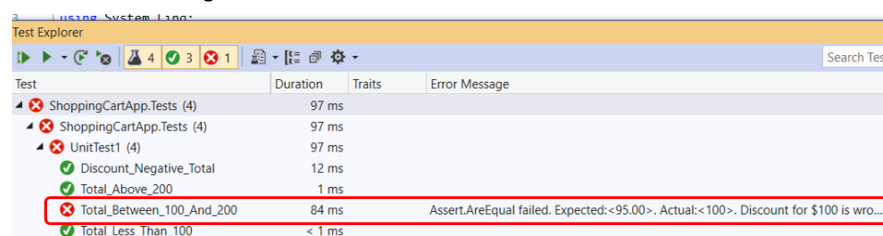
Download  
CodeForValueProducts.txt.

Open in NotePad and copy  
and paste the code into the  
**ValueProducts** method of  
**ValueAfterDiscountCalculator**

Don't forget that there are two **ValueProduct** methods because we have **two** implementations.

Change the code in the  
**ValueAfterDiscountCalculator** implementation.

Now run the tests again. Test > Run All Tests



Test	Duration	Traits	Error Message
ShoppingCartApp.Tests (4)	97 ms		
ShoppingCartApp.Tests (4)	97 ms		
UnitTest1 (4)	97 ms		
Discount_Negative_Total	12 ms		
Total_Above_200	1 ms		
Total_Between_100_And_200	84 ms		Assert.AreEqual failed. Expected:<95.00>. Actual:<100>. Discount for \$100 is wro...
Total_Less_Than_100	< 1 ms		

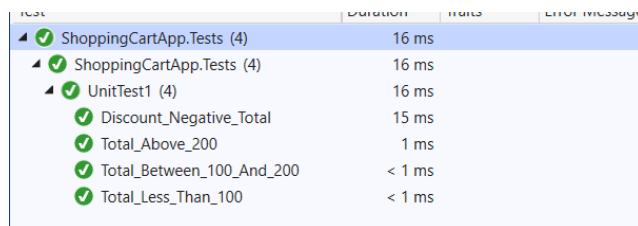
One of our tests didn't pass! We are told that the "Discount for \$100 is wrong" Our code is wrong when figuring the discount for a total of \$100.

If the total is between \$100 and \$200 inclusive, the discount will be 5 percent.

Change

```
else if (fullPrice > 100 && fullPrice <= 200)
to
else if (fullPrice >= 100 && fullPrice <= 200)
```

Now all of the tests pass.



Test	Duration	Traits	Error Message
ShoppingCartApp.Tests (4)	16 ms		
ShoppingCartApp.Tests (4)	16 ms		
UnitTest1 (4)	16 ms		
Discount_Negative_Total	15 ms		
Total_Above_200	1 ms		
Total_Between_100_And_200	< 1 ms		
Total_Less_Than_100	< 1 ms		

So again -

The Test-Driven Development (TDD) is a software engineering practice in which unit tests are written first, **before** the code you will test is actually written.

When the unit tests are first run, they will always fail because you haven't written the code.

Next, you write your code and check the tests again. Continue working on your code until all of your unit tests pass.

"When the tests all pass, you're done."

And this is exactly what we did.

Now that we've verified that our new implementation is working, we can go into NinjectDependencyResolver.cs and switch our implementation. Do that and then run the application. Verify that a discount has been applied.