

Walkthrough: Create and run unit tests for managed code

Article • 03/31/2022 • 11 minutes to read • [16 contributors](#)



In this article

[Create a project to test](#)

[Create a unit test project](#)

[Create the test class](#)

[Create the first test method](#)

[Build and run the test](#)

[Fix your code and rerun your tests](#)

[Use unit tests to improve your code](#)

[See also](#)

This article steps you through creating, running, and customizing a series of unit tests using the Microsoft unit test framework for managed code and Visual Studio **Test Explorer**. You start with a C# project that is under development, create tests that exercise its code, run the tests, and examine the results. Then you change the project code and rerun the tests. If you would like a conceptual overview of these tasks before going through these steps, see [Unit test basics](#).

Create a project to test

1. Open Visual Studio.
2. On the start window, choose **Create a new project**.
3. Search for and select the C# **Console App** project template for .NET Core, and then click **Next**.

Note

If you do not see the **Console App** template, you can install it from the **Create a new project** window. In the **Not finding what you're looking for?** message, choose the **Install more tools and features** link. Then, in the Visual Studio Installer, choose the **.NET Core cross-platform development** workload.

4. Name the project **Bank**, and then click **Next**.

Choose either the recommended target framework or .NET 6, and then choose **Create**.

The Bank project is created and displayed in **Solution Explorer** with the *Program.cs* file open in the code editor.

ⓘ Note

If *Program.cs* is not open in the editor, double-click the file *Program.cs* in **Solution Explorer** to open it.

5. Replace the contents of *Program.cs* with the following C# code that defines a class, *BankAccount*:

C#

 Copy

```
using System;

namespace BankAccountNS
{
    /// <summary>
    /// Bank account demo class.
    /// </summary>
    public class BankAccount
    {
        private readonly string m_customerName;
        private double m_balance;

        private BankAccount() { }

        public BankAccount(string customerName, double balance)
        {
            m_customerName = customerName;
            m_balance = balance;
        }

        public string CustomerName
```

```

{
    get { return m_customerName; }
}

public double Balance
{
    get { return m_balance; }
}

public void Debit(double amount)
{
    if (amount > m_balance)
    {
        throw new ArgumentOutOfRangeException("amount");
    }

    if (amount < 0)
    {
        throw new ArgumentOutOfRangeException("amount");
    }

    m_balance += amount; // intentionally incorrect code
}

public void Credit(double amount)
{
    if (amount < 0)
    {
        throw new ArgumentOutOfRangeException("amount");
    }

    m_balance += amount;
}

public static void Main()
{
    BankAccount ba = new BankAccount("Mr. Bryan Walton", 11.99);

    ba.Credit(5.77);
    ba.Debit(11.22);
    Console.WriteLine("Current balance is ${0}", ba.Balance);
}
}

```

6. Rename the file to *BankAccount.cs* by right-clicking and choosing **Rename** in **Solution Explorer**.

7. On the **Build** menu, click **Build Solution** (or press **Ctrl + SHIFT + B**).

You now have a project with methods you can test. In this article, the tests focus on the `Debit` method. The `Debit` method is called when money is withdrawn from an account.

Create a unit test project

1. On the **File** menu, select **Add > New Project**.

Tip

You can also right-click on the solution in **Solution Explorer** and choose **Add > New Project**.

2. Type **test** in the search box, select **C#** as the language, and then select the **C# MSTest Unit Test Project (.NET Core)** for .NET Core template, and then click **Next**.

Note

In Visual Studio 2019 version 16.9, the MSTest project template is **Unit Test Project**.

3. Name the project **BankTests** and click **Next**.
4. Choose either the recommended target framework or .NET 6, and then choose **Create**.

The **BankTests** project is added to the **Bank** solution.

5. In the **BankTests** project, add a reference to the **Bank** project.

In **Solution Explorer**, select **Dependencies** under the **BankTests** project and then choose **Add Reference** (or **Add Project Reference**) from the right-click menu.

6. In the **Reference Manager** dialog box, expand **Projects**, select **Solution**, and then check the **Bank** item.
7. Choose **OK**.

Create the test class

Create a test class to verify the `BankAccount` class. You can use the `UnitTest1.cs` file that was generated by the project template, but give the file and class more descriptive names.

Rename a file and class


1. To rename the file, in **Solution Explorer**, select the `UnitTest1.cs` file in the `BankTests` project. From the right-click menu, choose **Rename** (or press **F2**), and then rename the file to `BankAccountTests.cs`.
2. To rename the class, position the cursor on `UnitTest1` in the code editor, right-click, and then choose **Rename** (or press **F2**). Type in **BankAccountTests** and then press **Enter**.

The `BankAccountTests.cs` file now contains the following code:

C#	 Copy
<pre>using Microsoft.VisualStudio.TestTools.UnitTesting; namespace BankTests { [TestClass] public class BankAccountTests { [TestMethod] public void TestMethod1() { } } }</pre>	

Add a using statement

Add a [using statement](#) to the test class to be able to call into the project under test without using fully qualified names. At the top of the class file, add:

C#	 Copy
<pre>using BankAccountNS;</pre>	

Test class requirements

The minimum requirements for a test class are:

- The `[TestClass]` attribute is required on any class that contains unit test methods that you want to run in Test Explorer.
- Each test method that you want Test Explorer to recognize must have the `[TestMethod]` attribute.

You can have other classes in a unit test project that do not have the `[TestClass]` attribute, and you can have other methods in test classes that do not have the `[TestMethod]` attribute. You can call these other classes and methods from your test methods.

Create the first test method

In this procedure, you'll write unit test methods to verify the behavior of the `Debit` method of the `BankAccount` class.

There are at least three behaviors that need to be checked:

- The method throws an [ArgumentOutOfRangeException](#) if the debit amount is greater than the balance.
- The method throws an [ArgumentOutOfRangeException](#) if the debit amount is less than zero.
- If the debit amount is valid, the method subtracts the debit amount from the account balance.

Tip

You can delete the default `TestMethod1` method, because you won't use it in this walkthrough.

To create a test method

The first test verifies that a valid amount (that is, one that is less than the account balance and greater than zero) withdraws the correct amount from the account. Add the following method to that `BankAccountTests` class:

--	--

```
[TestMethod]
public void Debit_WithValidAmount_UpdatesBalance()
{
    // Arrange
    double beginningBalance = 11.99;
    double debitAmount = 4.55;
    double expected = 7.44;
    BankAccount account = new BankAccount("Mr. Bryan Walton", beginningBalance);

    // Act
    account.Debit(debitAmount);

    // Assert
    double actual = account.Balance;
    Assert.AreEqual(expected, actual, 0.001, "Account not debited correctly");
}
```

The method is straightforward: it sets up a new `BankAccount` object with a beginning balance and then withdraws a valid amount. It uses the `Assert.AreEqual` method to verify that the ending balance is as expected. Methods such as `Assert.AreEqual`, `Assert.IsTrue`, and others are frequently used in unit testing. For more conceptual information on writing a unit test, see [Write your tests](#).

Test method requirements

A test method must meet the following requirements:

- It's decorated with the `[TestMethod]` attribute.
- It returns `void`.
- It cannot have parameters.

Build and run the test

1. On the **Build** menu, choose **Build Solution** (or press **Ctrl + SHIFT + B**).
2. If **Test Explorer** is not open, open it by choosing **Test > Windows > Test Explorer** from the top menu bar (or press **Ctrl + E, T**).

3. Choose **Run All** to run the test (or press **Ctrl + R, V**).

While the test is running, the status bar at the top of the **Test Explorer** window is animated. At the end of the test run, the bar turns green if all the test methods pass, or red if any of the tests fail.

In this case, the test fails.

4. Select the method in **Test Explorer** to view the details at the bottom of the window.


Fix your code and rerun your tests

The test result contains a message that describes the failure. For the `AreEqual` method, the message displays what was expected and what was actually received. You expected the balance to decrease, but instead it increased by the amount of the withdrawal.


The unit test has uncovered a bug: the amount of the withdrawal is *added* to the account balance when it should be *subtracted*.

Correct the bug

To correct the error, in the `BankAccount.cs` file, replace the line:

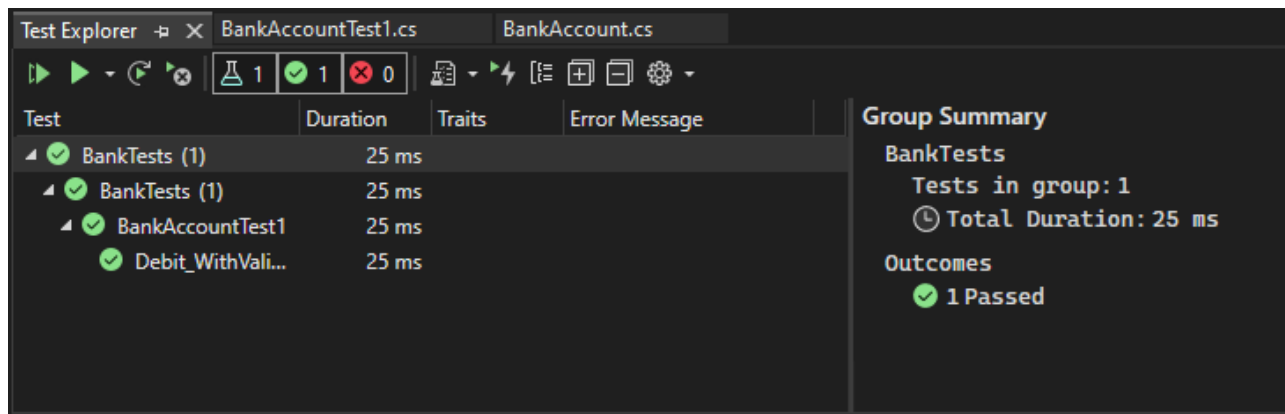
C#	 Copy
<pre>m_balance += amount;</pre>	

with:

C#	 Copy
<pre>m_balance -= amount;</pre>	

Rerun the test

In **Test Explorer**, choose **Run All** to rerun the test (or press **Ctrl + R, V**). The red/green bar turns green to indicate that the test passed.



Use unit tests to improve your code

This section describes how an iterative process of analysis, unit test development, and refactoring can help you make your production code more robust and effective.

Analyze the issues

You've created a test method to confirm that a valid amount is correctly deducted in the `Debit` method. Now, verify that the method throws an [ArgumentOutOfRangeException](#) if the debit amount is either:

- greater than the balance, or
- less than zero.

Create and run new test methods

Create a test method to verify correct behavior when the debit amount is less than zero:

```
C# Copy

[TestMethod]
public void Debit_WhenAmountIsLessThanZero_ShouldThrowArgumentOutOfRangeException()
{
    // Arrange
    double beginningBalance = 11.99;
    double debitAmount = -100.00;
    BankAccount account = new BankAccount("Mr. Bryan Walton", beginningBalance);

    // Act and assert
    Assert.ThrowsException<System.ArgumentOutOfRangeException>(() =>
```

```
account.Debit(debitAmount));  
}
```

Use the [ThrowsException](#) method to assert that the correct exception has been thrown. This method causes the test to fail unless an [ArgumentOutOfRangeException](#) is thrown. If you temporarily modify the method under test to throw a more generic [ApplicationException](#) when the debit amount is less than zero, the test behaves correctly—that is, it fails.

To test the case when the amount withdrawn is greater than the balance, do the following steps:

1. Create a new test method named `Debit_WhenAmountIsMoreThanBalance_ShouldThrowArgumentOutOfRangeException`.
2. Copy the method body from `Debit_WhenAmountIsLessThanZero_ShouldThrowArgumentOutOfRangeException` to the new method.
3. Set the `debitAmount` to a number greater than the balance.

Run the two tests and verify that they pass.

Continue the analysis

The method being tested can be improved further. With the current implementation, we have no way to know which condition (`amount > m_balance` OR `amount < 0`) led to the exception being thrown during the test. We just know that an `ArgumentOutOfRangeException` was thrown somewhere in the method. It would be better if we could tell which condition in `BankAccount.Debit` caused the exception to be thrown (`amount > m_balance` OR `amount < 0`) so we can be confident that our method is sanity-checking its arguments correctly.

Look at the method being tested (`BankAccount.Debit`) again, and notice that both conditional statements use an `ArgumentOutOfRangeException` constructor that just takes name of the argument as a parameter:

C#

 Copy


```
throw new ArgumentOutOfRangeException("amount");
```

There is a constructor you can use that reports far richer information:


[ArgumentOutOfRangeException\(String, Object, String\)](#) includes the name of the argument, the argument value, and a user-defined message. You can refactor the method under test to use this constructor. Even better, you can use publicly available type members to specify the errors.

Refactor the code under test

First, define two constants for the error messages at class scope. Put these in the class under test, `BankAccount`:

C#	 Copy
<pre>public const string DebitAmountExceedsBalanceMessage = "Debit amount exceeds balance"; public const string DebitAmountLessThanZeroMessage = "Debit amount is less than zero";</pre>	


Then, modify the two conditional statements in the `Debit` method:

C#	 Copy
<pre>if (amount > m_balance) { throw new System.ArgumentOutOfRangeException("amount", amount, DebitAmountExceedsBalanceMessage); } if (amount < 0) { throw new System.ArgumentOutOfRangeException("amount", amount, DebitAmountLessThanZeroMessage); }</pre>	

Refactor the test methods

Refactor the test methods by removing the call to [Assert.ThrowsException](#). Wrap the call to `Debit()` in a try/catch block, catch the specific exception that's expected, and verify its associated message. The [Microsoft.VisualStudio.TestTools.UnitTesting.StringAssert.Contains](#) method provides the ability to compare two strings.

Now, the `Debit_WhenAmountIsMoreThanBalance_ShouldThrowArgumentOutOfRangeException` might look like this:

C# 

```
[TestMethod]
public void Debit_WhenAmountIsMoreThanBalance_ShouldThrowArgumentOutOfRangeException()
{
    // Arrange
    double beginningBalance = 11.99;
    double debitAmount = 20.0;
    BankAccount account = new BankAccount("Mr. Bryan Walton", beginningBalance);

    // Act
    try
    {
        account.Debit(debitAmount);
    }
    catch (System.ArgumentOutOfRangeException e)
    {
        // Assert
        StringAssert.Contains(e.Message,
            BankAccount.DebitAmountExceedsBalanceMessage);
    }
}
```

Retest, rewrite, and reanalyze

Currently, the test method doesn't handle all the cases that it should. If the method under test, the `Debit` method, failed to throw an `ArgumentOutOfRangeException` when the `debitAmount` was larger than the balance (or less than zero), the test method would pass. This is not good, because you want the test method to fail if no exception is thrown.

This is a bug in the test method. To resolve the issue, add an `Assert.Fail` assert at the end of the test method to handle the case where no exception is thrown.

Rerunning the test shows that the test now *fails* if the correct exception is caught. The catch block catches the exception, but the method continues to execute and it fails at the new `Assert.Fail` assert. To resolve this problem, add a `return` statement after the `StringAssert` in the catch block. Rerunning the test confirms that you've fixed this problem. The final version of the `Debit_WhenAmountIsMoreThanBalance_ShouldThrowArgumentOutOfRangeException` looks like this:

```
[TestMethod]
public void Debit_WhenAmountIsMoreThanBalance_ShouldThrowArgumentOutOfRangeException()
{
    // Arrange
    double beginningBalance = 11.99;
    double debitAmount = 20.0;
    BankAccount account = new BankAccount("Mr. Bryan Walton", beginningBalance);

    // Act
    try
    {
        account.Debit(debitAmount);
    }
    catch (System.ArgumentOutOfRangeException e)
    {
        // Assert
        StringAssert.Contains(e.Message,
            BankAccount.DebitAmountExceedsBalanceMessage);
        return;
    }

    Assert.Fail("The expected exception was not thrown.");
}
```

Conclusion

The improvements to the test code led to more robust and informative test methods. But more importantly, they also improved the code under test.

Tip

This walkthrough uses the Microsoft unit test framework for managed code. **Test Explorer** can also run tests from third-party unit test frameworks that have adapters for **Test Explorer**. For more information, see [Install third-party unit test frameworks](#).

See also

For information about how to run tests from a command line, see [VSTest.Console.exe command-line options](#).

Recommended content

[Unit testing fundamentals - Visual Studio \(Windows\)](#)

Learn how Visual Studio Test Explorer provides a flexible and efficient way to run your unit tests and view their results.

[Unit testing C# with NUnit and .NET Core - .NET](#)

Learn unit test concepts in C# and .NET Core through an interactive experience building a sample solution step-by-step using dotnet test and NUnit.

[Get started with unit testing - Visual Studio \(Windows\)](#)

Use Visual Studio to define and run unit tests to maintain code health, and to find errors and faults before your customers do.

[Debug unit tests with Test Explorer - Visual Studio \(Windows\)](#)

Learn how to debug unit tests with Test Explorer in Visual Studio.

[StringAssert.Contains Method \(Microsoft.VisualStudio.TestTools.UnitTesting\)](#)

Tests whether the specified string contains the specified substring and throws an exception if the substring does not occur within the test string.

[Assert Class \(Microsoft.VisualStudio.TestTools.UnitTesting\)](#)

A collection of helper classes to test various conditions within unit tests. If the condition being tested is not met, an exception is thrown.

[Create a unit test project - Visual Studio \(Windows\)](#)

Learn how to create a unit test project. The test project can be in the same solution as the production code, or it can be in a separate solution.

Show more 

