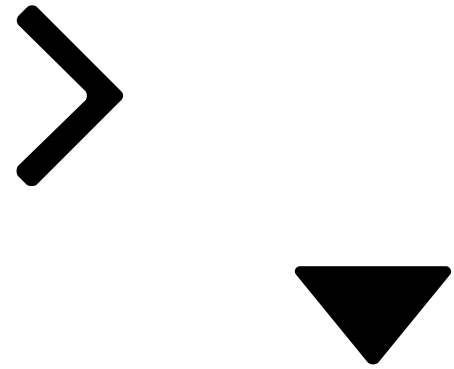
1. Apex Testing



2. Get Started with Apex Unit Tests

Get Started with Apex Unit Tests

Learning Objectives

After completing this unit, you'll be able to:

- Describe the key benefits of Apex unit tests.
- Define a class with test methods.
- Execute all test methods in a class and inspect failures.
- Create and execute a suite of test classes.

Apex Unit Tests

The Apex testing framework enables you to write and execute tests for your Apex classes and triggers on the Lightning Platform. Apex unit tests ensure high quality for your Apex code and let you meet requirements for deploying Apex.

Testing is the key to successful long-term development and is a critical component of the development process. The Apex testing framework makes it easy to test your Apex code. Apex code can only be written in a sandbox environment or a Developer org, not in production. Apex code can be deployed to a production org from a sandbox. Also, app developers can distribute Apex code to customers from their Developer orgs by uploading packages to the Lightning Platform AppExchange. In addition to being critical for quality assurance, Apex unit tests are also requirements for deploying and distributing Apex. The following are the benefits of Apex unit tests.

- Ensuring that your Apex classes and triggers work as expected
- Having a suite of regression tests that can be rerun every time classes and triggers are updated to ensure that future updates you make to your app don't break existing functionality
- Meeting the code coverage requirements for deploying Apex to production or distributing Apex to customers via packages
- High-quality apps delivered to the production org, which makes production users more productive
- High-quality apps delivered to package subscribers, which increase your customers trust

Note

Before each major service upgrade, Salesforce runs all Apex tests on your behalf through a process called Apex Hammer. The Hammer process runs in the current version and next release and compares the test results. This process ensures that the behavior in your custom code hasn't been altered as a result of service upgrades. The Hammer process picks orgs selectively and doesn't run in all orgs. Issues found are triaged based on certain criteria. Salesforce strives to fix all issues found before each new release.

Maintaining the security of your data is our highest priority. We don't view or modify any data in your org, and all testing is done in a copy that runs in a secure data center.

Code Coverage Requirement for Deployment

Before you can deploy your code or package it for the Lightning Platform AppExchange, at least 75% of Apex code must be covered by tests, and all those tests must pass. In addition, each trigger must have some coverage. Even though code coverage is a requirement for deployment, don't write tests only to meet this requirement. Make sure to test the common use cases in your app, including positive and negative test cases, and bulk and single-record processing.

Test Method Syntax

Test methods take no arguments and have the following syntax:

```
@isTest static void testName() {
    // code_block
}
```

Copy

Alternatively, a test method can have this syntax:

```
static testMethod void testName() {
    // code_block
}
```

Copy

Using the `isTest` annotation instead of the `testMethod` keyword is more flexible as you can specify parameters in the annotation. We'll cover one such parameter later.

The visibility of a test method doesn't matter, so declaring a test method as public or private doesn't make a difference as the testing framework is always able to access test methods. For this reason, the access modifiers are omitted in the syntax.

Test methods must be defined in test classes, which are classes annotated with `isTest`. This sample class shows a definition of a test class with one test method.

```
@isTest
private class MyTestClass {
    @isTest static void myTest() {
        // code_block
    }
}
```

Copy

Test classes can be either private or public. If you're using a test class for unit testing only, declare it as private. Public test classes are typically used for test data factory classes, which are covered later.

Unit Test Example: Test the TemperatureConverter Class

The following simple example is of a test class with three test methods. The class method that's being tested takes a temperature in Fahrenheit as an input. It converts this temperature to Celsius and returns the converted result. Let's add the custom class and its test class.

1. In the Developer Console, click **File** | **New** | **Apex Class**, and enter `TemperatureConverter` for the class name, and then click **OK**.
2. Replace the default class body with the following.

```
public class TemperatureConverter {
    // Takes a Fahrenheit temperature and returns the Celsius equivalent.
    public static Decimal FahrenheitToCelsius(Decimal fh) {
        Decimal cs = (fh - 32) * 5/9;
        return cs.setScale(2);
    }
}
```

Copy

3. Press Ctrl+S to save your class.
4. Repeat the previous steps to create the `TemperatureConverterTest` class. Add the following for this class.

```
@isTest
private class TemperatureConverterTest {
    @isTest static void testWarmTemp() {
        Decimal celsius = TemperatureConverter.FahrenheitToCelsius(70);
        System.assertEquals(21.11,celsius);
    }

    @isTest static void testFreezingPoint() {
        Decimal celsius = TemperatureConverter.FahrenheitToCelsius(32);
        System.assertEquals(0,celsius);
    }
    @isTest static void testBoilingPoint() {
        Decimal celsius = TemperatureConverter.FahrenheitToCelsius(212);
        System.assertEquals(100,celsius,'Boiling point temperature is not expected.');
    }

    @isTest static void testNegativeTemp() {
        Decimal celsius = TemperatureConverter.FahrenheitToCelsius(-10);
        System.assertEquals(-23.33,celsius);
    }
```

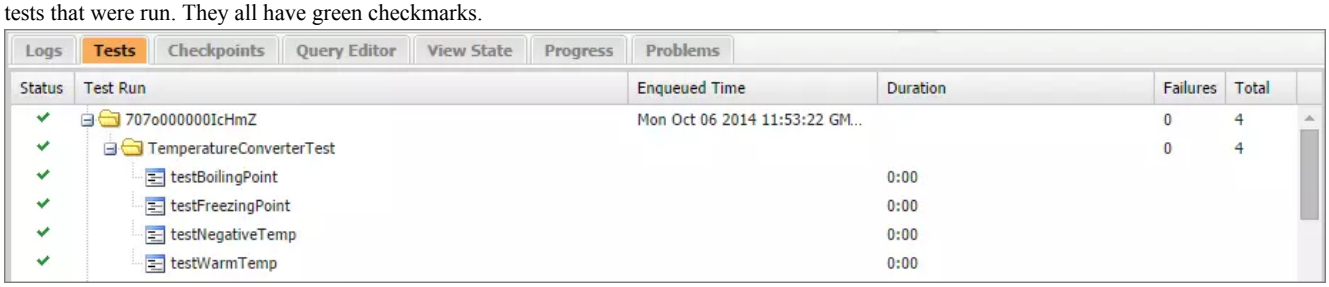
```
}
```

Copy

The `TemperatureConverterTest` test class verifies that the method works as expected by calling it with different inputs for the temperature in Fahrenheit. Each test method verifies one type of input: a warm temperature, the freezing point temperature, the boiling point temperature, and a negative temperature. The verifications are done by calling the `System.assertEquals()` method, which takes two parameters: the first is the expected value, and the second is the actual value. There is another version of this method that takes a third parameter—a string that describes the comparison being done, which is used in `testBoilingPoint()`. This optional string is logged if the assertion fails.

Let's run the methods in this class.

1. In the Developer Console, click **Test** | **New Run**.
2. Under Test Classes, click **TemperatureConverterTest**.
3. To add all the test methods in the `TemperatureConverterTest` class to the test run, click **Add Selected**.
4. Click **Run**.
5. In the Tests tab, you see the status of your tests as they're running. Expand the test run, and expand again until you see the list of individual tests that were run. They all have green checkmarks.



After you run tests, code coverage is automatically generated for the Apex classes and triggers in the org. You can check the code coverage percentage in the Tests tab of the Developer Console. In this example, the class you've tested, the `TemperatureConverter` class, has 100% coverage, as shown in this image.

Note

Whenever you modify your Apex code, rerun your tests to refresh code coverage results.

A known issue with the Developer Console prevents it from updating code coverage correctly when running a subset of tests. To update your code coverage results, use **Test** | **Run All** rather than **Test** | **New Run**.

While one test method would have resulted in full coverage of the `TemperatureConverter` class, it's still important to test for different inputs to ensure the quality of your code. Obviously, it isn't possible to verify every data point, but you can test for common data points and different ranges of input. For example, you can verify passing positive and negative numbers, boundary values, and invalid parameter values to verify negative behavior. The tests for the `TemperatureConverter` class verify common data points, like the boiling temperature, and negative temperatures.

The `TemperatureConverterTest` test class doesn't cover invalid inputs or boundary conditions. Boundary conditions are about minimum and maximum values. In this case, the temperature conversion method accepts a Decimal, which can accept large numbers, higher than Double values. For invalid inputs, there is no invalid temperature but the only invalid input is `null`. How does the conversion method handle this value? In this case, when the Apex runtime dereferences the parameter variable to evaluate the formula, it throws a `System.NullPointerException`. You can modify the `FahrenheitToCelsius()` method to check for an invalid input and return `null` in that case, and then add a test to verify the invalid input behavior.

Up to this point, all tests pass because the conversion formula used in the class method is correct. But that's boring! Let's try to simulate a failure just to see what happens when an assertion fails. For example, let's modify the boiling point temperature test and pass in a false expected value for the boiling point Celsius temperature (0 instead of 100). This causes the corresponding test method to fail.

1. Change the `testBoilingPoint()` test method to the following.

```
    @isTest static void testBoilingPoint() {
        Decimal celsius = TemperatureConverter.FahrenheitToCelsius(212);
        // Simulate failure
        System.assertEquals(0,celsius,'Boiling point temperature is not expected.');
    }
```

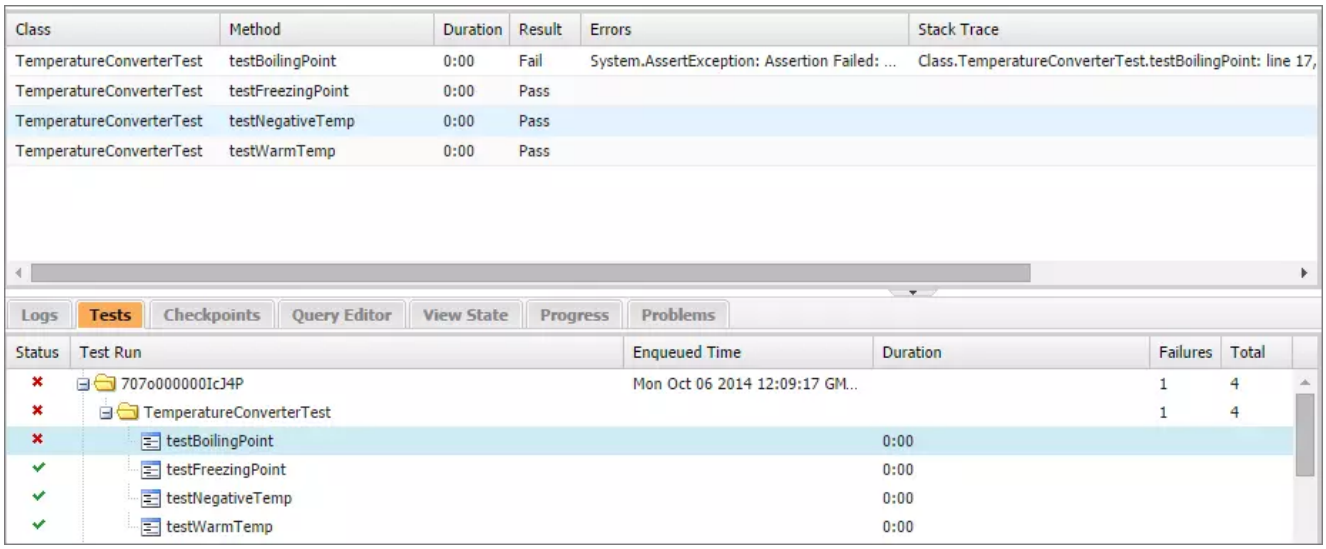
Copy

2. To execute the same test run, click the latest run in the Tests tab, and then click **Test** | **Rerun**.

The assertion in `testBoilingPoint()` fails and throws a fatal error (an AssertException that can't be caught).

3. Check the results in the Tests tab by expanding the latest test run. The test run reports one out of four tests failed. To get more details about the failure, double-click the test run.

   Detailed results appear in a separate tab as shown in this image.



4. To get the error message for the test failure, double-click inside the Errors column for the failed test. You'll see the following. The descriptive text next to `Assertion Failed:` is the text we provided in the `System.assertEquals()` statement.

```
System.AssertException: Assertion Failed: Boiling point temperature is not expected.: Expected: 0, Actual: 100.00
```

The test data in these test methods are numbers and not Salesforce records. You'll find out more about how to test Salesforce records and how to set up your data in the next unit.

Increase Your Code Coverage

When writing tests, try to achieve the highest code coverage possible. Don't just aim for 75% coverage, which is the lowest coverage that the Lightning Platform requires for deployments and packages. The more test cases that your tests cover, the higher the likelihood that your code is robust. Sometimes, even after you write test methods for all your class methods, code coverage is not at 100%. One common cause is not covering all data values for conditional code execution. For example, some data values tend to be ignored when your class method has `if` statements that cause different branches to be executed based on whether the evaluated condition is met. Ensure that your test methods account for these different values.

This example includes the class method, `getTaskPriority()`, which contains two `if` statements. The main task of this method is to return a priority string value based on the given lead state. The method validates the state first and returns `null` if the state is invalid. If the state is CA, the method returns 'High'; otherwise, it returns 'Normal' for any other state value.

```
public class TaskUtil {
    public static String getTaskPriority(String leadState) {
        // Validate input
        if (String.isBlank(leadState) || leadState.length() > 2) {
            return null;
        }

        String taskPriority;

        if (leadState == 'CA') {
            taskPriority = 'High';
        } else {
            taskPriority = 'Normal';
        }

        return taskPriority;
    }
}
```

Copy

Note

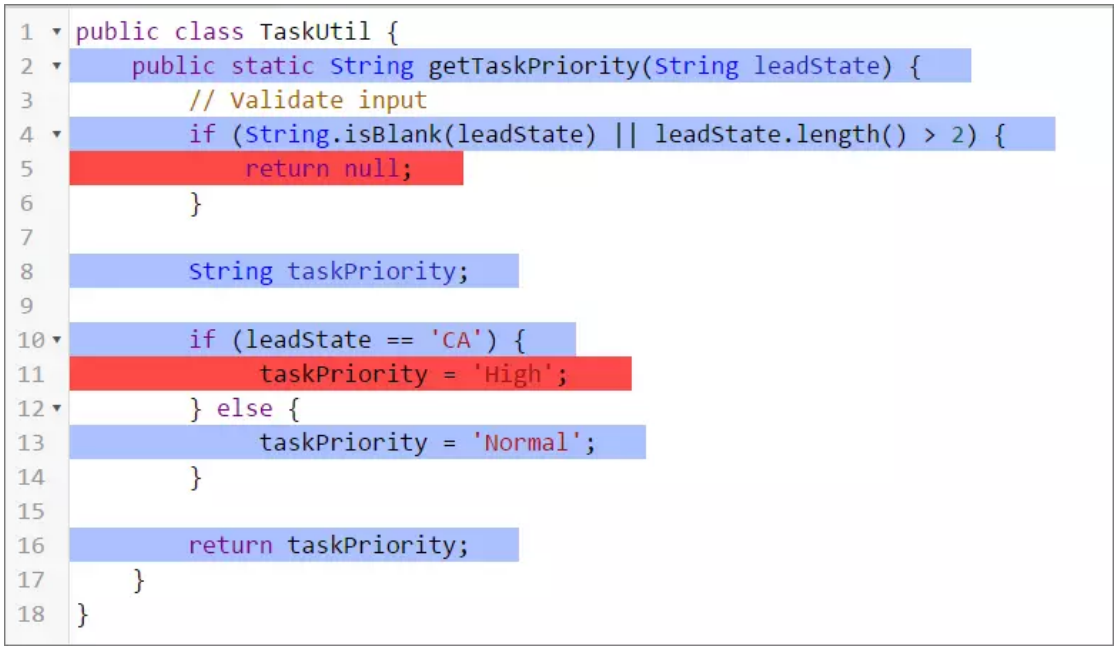
The equality operator (`==`) performs case-insensitive string comparisons, so there is no need to convert the string to lower case first. This means that passing in `'ca'` or `'Ca'` will satisfy the equality condition with the string literal `'CA'`.

This is the test class for the `getTaskPriority()` method. The test method simply calls `getTaskPriority()` with one state (`'NY'`).

```
@isTest
private class TaskUtilTest {
    @isTest static void testTaskPriority() {
        String pri = TaskUtil.getTaskPriority('NY');
        System.assertEquals('Normal', pri);
    }
}
```

Copy

Let's run this test class (`TaskUtilTest`) in the Developer Console and check code coverage for the corresponding `TaskUtil` class that this test covers. After the test run finishes, the code coverage for `TaskUtil` is shown as 75%. If you open this class in the Developer Console, you see six blue (covered) lines and two red (uncovered) lines, as shown in this image.



```
1  public class TaskUtil {
2      public static String getTaskPriority(String leadState) {
3          // Validate input
4          if (String.isBlank(leadState) || leadState.length() > 2) {
5              return null;
6          }
7
8          String taskPriority;
9
10         if (leadState == 'CA') {
11             taskPriority = 'High';
12         } else {
13             taskPriority = 'Normal';
14         }
15
16         return taskPriority;
17     }
18 }
```

The reason why line 5 wasn't covered is because our test class didn't contain a test to pass an invalid state parameter. Similarly, line 11 wasn't covered because the test method didn't pass `'CA'` as the state. Let's add two more test methods to cover those scenarios. The following shows the full test class after adding the `testTaskHighPriority()` and `testTaskPriorityInvalid()` test methods. If you rerun this test class, the code coverage for `TaskUtil` is now at 100%!

```
@isTest
private class TaskUtilTest {
    @isTest static void testTaskPriority() {
        String pri = TaskUtil.getTaskPriority('NY');
        System.assertEquals('Normal', pri);
    }

    @isTest static void testTaskHighPriority() {
        String pri = TaskUtil.getTaskPriority('CA');
        System.assertEquals('High', pri);
    }

    @isTest static void testTaskPriorityInvalid() {
        String pri = TaskUtil.getTaskPriority('Montana');
        System.assertEquals(null, pri);
    }
}
```

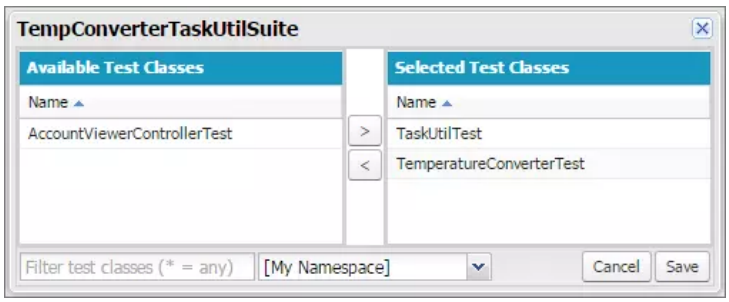
Copy

Create and Execute a Test Suite

A test suite is a collection of Apex test classes that you run together. For example, create a suite of tests that you run every time you prepare for a deployment or Salesforce releases a new version. Set up a test suite in the Developer Console to define a set of test classes that you execute together regularly.

You now have two test classes in your org. These two classes aren't related, but let's pretend for the moment that they are. Assume that there are situations when you want to run these two test classes but don't want to run all the tests in your org. Create a test suite that contains both classes, and then execute the tests in the suite.

1. In the Developer Console, select **Test** | **New Suite**.
2. Enter `TempConverterTaskUtilSuite` for the suite name, and then click **OK**.
3. Select **TaskUtilTest**, hold down the Ctrl key, and then select **TemperatureConverterTest**.
4. To add the selected test classes to the suite, click >.



5. Click **Save**.
6. Select **Test** | **New Suite Run**.
7. Select **TempConverterTaskUtilSuite**, and then click **>** to move `TempConverterTaskUtilSuite` to the Selected Test Suites column.
8. Click **Run Suites**.
9. On the Tests tab, monitor the status of your tests as they're running. Expand the test run, and expand again until you see the list of individual tests that were run. Like in a run of individual test methods, you can double-click method names to see detailed test results.

Creating Test Data

Salesforce records that are created in test methods aren't committed to the database. They're rolled back when the test finishes execution. This rollback behavior is handy for testing because you don't have to clean up your test data after the test executes.

By default, Apex tests don't have access to pre-existing data in the org, except for access to setup and metadata objects, such as the User or Profile objects. Set up test data for your tests. Creating test data makes your tests more robust and prevents failures that are caused by missing or changed data in the org. You can create test data directly in your test method, or by using a utility test class as you'll find out later.

Note

Even though it is not a best practice to do so, there are times when a test method needs access to pre-existing data. To access org data, annotate the test method with `@isTest(SeeAllData=true)`. The test method examples in this unit don't access org data and therefore don't use the `SeeAllData` parameter.

Tell Me More...

- You can save up to 6 MB of Apex code in each org. Test classes annotated with `@isTest` don't count toward this limit.
- Even though test data rolls back, no separate database is used for testing. As a result, for some sObjects that have fields with unique constraints, inserting duplicate sObject records results in an error.
- Test methods don't send emails.
- Test methods can't make callouts to external services. You can use mock callouts in tests.
- SOSL searches performed in a test return empty results. To ensure predictable results, use `Test.setFixedSearchResults()` to define the records to be returned by the search.

Resources

Documentation

Check out the following in the *Apex Developer Guide*.

- Testing Best Practices
- What Are Apex Unit Tests?
- Isolation of Test Data from Organization Data in Unit Tests

Check out the following in the Salesforce Help.

- Checking Code Coverage

Get Ready

You'll be completing this challenge in your own personal Salesforce environment. Choose from the dropdown menu, then click **Launch** to get started. If you use Trailhead in a language other than English, set the language of your Trailhead Playground to English before you attempt this challenge. Want to find out more about using hands-on orgs for Trailhead learning? Check out the Trailhead Playground Management module.

Your Challenge

Create a unit test for a simple Apex class.
Install a simple Apex class, write unit tests that achieve 100% code coverage for the class, and run your Apex tests.

- The Apex class to test is called 'VerifyDate', and the code is available here. Copy and paste this class into your Developer Edition via the Developer Console.
- 'VerifyDate' is a class which tests if a date is within a proper range, and if not will return a date that occurs at the end of the month within the range.
- The unit tests must be in a separate test class called 'TestVerifyDate'.
- The unit tests must cover scenarios for all lines of code included in the Apex class, resulting in 100% code coverage.
- Run your test class at least once (via 'Run All' tests the Developer Console) before attempting to verify this challenge.

Launch

Check challenge to earn 500 points