



**MALLA REDDY UNIVERSITY**

**R-22**

**III YEAR B.TECH. (CSE) / I – SEM**

**MR22-1CS0155**

**Salesforce Platform Developer**

# Unit 3

**Dr Shaik Hussain Shaik Ibrahim**

**Associate prof/CSE**

**MRU**

## **UNIT-III**

**Apex Testing:** Get Started with Apex Unit Tests, Test Apex Triggers, Create Test Data for Apex Tests.

**Developer Console Basics:** Get Started with the Developer Console, Navigate and Edit Source Code, Generate and Analyze Logs, Inspect Objects at Checkpoints.

# Apex Testing - Get Started with Apex Unit Tests

## Apex Testing :

- Apex testing is the **process of writing and running test code** to ensure that **Apex code components work as expected**.
- It helps you to **create relevant test data in the test classes and run them in Salesforce**.

As a part of Apex testing, you need to test for:

**Positive behaviour:** Test Apex code to verify that it works as per specification when appropriate input is provided to the code.

**Negative behaviour:** Test the limitations of the system if an unexpected value or input is provided to the Apex code.

**Restricted user:** Check that a user whose access to certain objects or records is restricted is barred from accessing them.

# Apex Testing - Get Started with Apex Unit Tests

## Apex Unit Tests:

- Apex provides a **testing framework** that allows you to **write unit tests, run your tests, check test results, and have code coverage results 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.

**Note:** In Salesforce, if the **Apex code is not properly tested, it can't be deployed into the Production environment.**

# Apex Testing - Get Started with Apex Unit Tests

These 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

# Apex Testing - Get Started with Apex Unit Tests

## Code Coverage Requirement for Deployment:

- **At least 75%** of your Apex Code must be covered by unit tests, and all of those tests must complete successfully.
- Every **trigger must have some test coverage**.
- All classes and triggers **must compile successfully**.

## There are two ways of testing a salesforce application:

- **Manual** - through the Salesforce user interface, important, but merely testing through the UI will not catch all the use cases for your application.
- **Automated** - to test for bulk functionality through the code if it's invoked using SOAP API or by Visualforce standard set controller.

# Apex Testing - Get Started with Apex Unit Tests

## How To Write Apex Unit Tests?

- **Unit tests are class methods** that verify whether a particular piece of code is working properly or not.
- Unit test **methods take no arguments, commit no data to the database, send no emails**, and are **flagged with the testMethod keyword or the isTest annotation** in the method definition.
- Also, **test methods must be defined in test classes**, that is, classes annotated with isTest.
- Create **different test methods to test different functionalities**.
- In each test, method writes **different test cases to test your code** whether it is working properly with the different inputs or not.



# Apex Testing - Get Started with Apex Unit Tests

## Test Method Syntax:

Test methods are defined using the **@isTest** annotation or using **testmethod** keyword and have the following syntax,

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

(OR)

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

- 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**.

# Apex Testing - Get Started with Apex Unit Tests

- This sample class shows a definition of a test class with one test method.
- Test classes can be **either private or public**.

**@isTest**

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

- If you're using a test class for unit testing only, declare it as private.

**Note: Test methods aren't allowed in non-test classes**

# Unit Test Example: Steps

1. In the Developer Console, click **File | New | Apex Class**, and enter the class name, and then click **OK**
2. Type the code and Press **Ctrl+S** to save your class
3. Create the test class and the test methods for your respective class and then click **OK to save**

# How to run the test classes and methods

1. In the Developer Console, click **Test | New Run**
2. Under **Test Classes** click the test classes you have written
3. Add all the test methods in your test 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

# Apex Testing - Get Started with Apex Unit Tests

## 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 two positive integer numbers as an input.

It adds the two numbers and returns the summed result.

Let's add the custom class and its test class.

## Steps to create Test class:

1. In the **Developer Console**, click **File | New | Apex Class**, and enter **Add** for the class name, and then click **OK**.

# Apex Testing - Get Started with Apex Unit Tests

2. Write the following coding in the add class body.

## Apex Class – Add.apxc

```
public class Add
{
    public static integer addvalue (integer a, integer b)
    {
        if(a<0 || b<0)
        {
            return -1;
        }
        else
        {
            integer result = a + b;
            return result;
        }
    }
}
```

# Apex Testing - Get Started with Apex Unit Tests

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

## Apex Class – AddTest.apxc

```
@isTest public class AddTest {  
  
    @isTest private static void addValueTest1(){  
        integer res = Add.addvalue(10,-12);  
        system.assertEquals(-1,res);  
  
    @isTest private static void addValueTest2(){  
        integer res = Add.addvalue(10,20);  
        system.assertEquals(30,res);    }  
  
    @isTest private static void addValueTest3(){  
        integer res = Add.addvalue(32,20);  
        system.assertEquals(10,res);    }    }
```

# Apex Testing - Get Started with Apex Unit Tests

- The **AddTest** test class verifies that the method works as expected by calling it with different inputs.
- Each test method verifies one type of input:
  - Given input is **Positive or Negative**,
  - Result by adding two number has **Positive Behaviour**,
  - Result by adding two number has **Negative Behaviour**,

The verifications are done by calling the **System.assertEquals()** method, which takes two parameters:

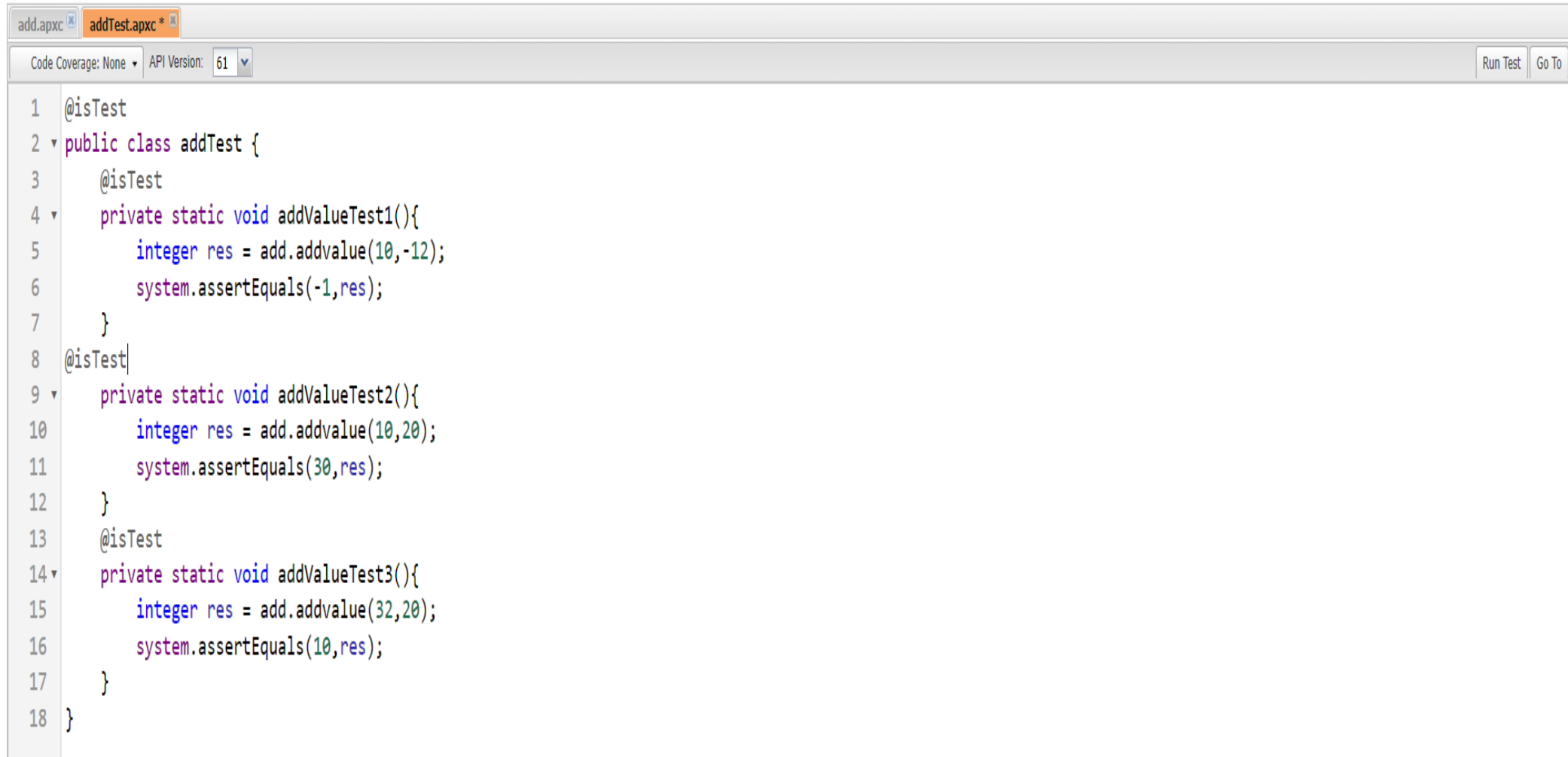
- 1. Expected value, and**
- 2. Actual value.**



# Apex Testing - Get Started with Apex Unit Tests

Let's run the methods in this class.

1. Under **Test Classes**, click **Run Test**.



```
add.apxc | addTest.apxc *
Code Coverage: None | API Version: 61 | Run Test | Go To

1  @isTest
2  public class addTest {
3      @isTest
4      private static void addValueTest1(){
5          integer res = add.addvalue(10,-12);
6          system.assertEquals(-1,res);
7      }
8      @isTest
9      private static void addValueTest2(){
10         integer res = add.addvalue(10,20);
11         system.assertEquals(30,res);
12     }
13     @isTest
14     private static void addValueTest3(){
15         integer res = add.addvalue(32,20);
16         system.assertEquals(10,res);
17     }
18 }
```

2. 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.

Logs Tests Checkpoints Query Editor View State Progress Problems					
Status	Test Run	Enqueued Time	Duration	Failures	Total
✖	TestRun @ 3:24:43 pm			1	3
✖	addTest			1	3
✖	addValueTest3		0:00		
✔	addValueTest1		0:00		
✔	addValueTest2		0:00		

# Code Coverage

- 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

Overall Code Coverage			>>
Class	Percent	Lines	
Overall	100%		
TemperatureConverter	100%	3/3	

# To Increase the code coverage

- To increase the code coverage, cover all the test code for your user defined class in the written test class

# Apex Testing - Get Started with Apex Unit Tests

- 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 **Add class, has 100% coverage**, as shown in this image.

Overall Code Coverage >>		
Class	Percent	Lines
Overall	15%	
AccountDeletion	0%	0/4
add	100%	5/5
Person	0%	0/2
TeacherClass	0%	0/8
TeacherTrigger	0%	0/5
Trigger1	0%	0/9

# Apex Testing - Get Started with Apex Unit Tests

## Example:

```
public class BranchInsert {  
    public static void insertBranch(String BranchName) {  
        Branch__c br = new Branch__c ( BName__c = BranchName);  
        insert br;    }    }
```

```
@isTest public class BranchInsertTest {  
    @isTest static void testInsertBranch() {  
        String testBranchName = 'IOT';  
        BranchInsert.insertBranch(testBranchName);  
        Branch__c resultBranch = [SELECT Id, BName__c FROM Branch__c  
                                WHERE BName__c = :testBranchName LIMIT 1];  
        System.assertNotEquals(null, resultBranch);  
        System.assertEquals(testBranchName, resultBranch.BName__c);    }    }
```

# Apex Testing - System-Defined Methods

## System-Defined Methods:

The common system-defined unit test methods are:

**startTest:** startTest method marks the point in your test code when the test actually begins.

**stopTest:** stopTest method comes after the startTest method and marks the end point of an actual test code.

## Purpose:

- Any code that executes after the call to start test() and before the stop test () is assigned a new set of governor limits.
- Any code that executes after the call to stop test() is arranged with the original limits that were in effect before the start test() was called.

# Apex Testing - System-Defined Methods

The **startTest** method marks the point in your test code when your test actually begins. Each test method is allowed to call this method only once. All of the code before this method should be used to initialize variables, populate data structures, and so on, allowing you to set up everything you need to run your test. Any code that executes after the call to startTest and before stopTest is assigned a new set of governor limits.

The startTest method does not refresh the context of the test: it adds a context to your test. **For example**, if your class makes 98 SOQL queries before it calls startTest, and the first significant statement after startTest is a DML statement, the program can now make an additional 100 queries. Once stopTest is called, however, the program goes back into the original context, and can only make 2 additional SOQL queries before reaching the limit of 100.



# Apex Testing - System-Defined Methods

The **stopTest** method marks the point in your test code when your test ends. Use this method in conjunction with the startTest method. Each test method is allowed to call this method only once. Any code that executes after the stopTest method is assigned the original limits that were in effect before startTest was called. All asynchronous calls made after the startTest method are collected by the system. When stopTest is executed, all asynchronous processes are run synchronously. An exception encountered during stopTest halts the synchronous processing.

**For example,** an unhandled exception in a batch job's execute method will prevent the finish method from running in a test context.

# Apex Testing - System-Defined Methods

Governor limits are reset when the `Test.startTest` appears and the code between `Test.startTest` and `Test.stopTest` executes in fresh set of governor limits (Context changes). Also `Test.stopTest` appears, the context is again moved back to the original code.

**For example,** you have a scenario, if during the setup of your test you needed to execute 99 SOQL queries and insert 9,999 records to seed the org with the data your code required for proper testing, if Salesforce did not offer a mechanism to reset the governor limits the code which you are testing would only have room for one more SOQL query and one more record in a DML statement before it would hit one of those two limits (100 queries and 10,000 records processed by DML statements respectively) and throw an exception.

In the above scenario, if you were to call `Test.startTest()` after your 99 queries were complete and your 9,999 rows were DML'd – the transaction limits within your test would be back to zero and at that point the code which you are testing would be running in a context that more closely resembles a single transaction's limits in real life. This mechanism allows you to “ignore” the work that had to be done to set up the test scenario.

# Apex Testing - System-Defined Methods

## Example:

```
@isTest

private class myClass {

    static testMethod void myTest() {

        // Create test data

        .....Test.startTest();

        // Actual apex code testing

        .....Test.stopTest();

    }
}
```

# Apex Testing - System-Defined Methods

## Example:

```
public class BranchInsert {  
    public static void insertBranch(String BranchName) {  
        Branch__c br = new Branch__c ( BName__c = BranchName);  
        insert br;    }    }  
  
@isTest public class BranchInsertTest {  
    @isTest static void testInsertBranch() {  
        String testBranchName = 'IOT';  
  
        Test.startTest();           // Start test context  
        BranchInsert.insertBranch(testBranchName);  
        Test.stopTest();           // Stop test context  
  
        Branch__c resultBranch = [SELECT Id, BName__c FROM Branch__c  
                                WHERE BName__c = :testBranchName LIMIT 1];  
  
        System.assertNotEquals(null, resultBranch);  
        System.assertEquals(testBranchName, resultBranch.BName__c);    }    }
```

# 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.

# Test Suite – working scenario

- You 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.

# Steps to create test suite

1. In the Developer Console, select **Test | New Suite**.
2. Enter a test suite name and click ok
3. Select test class 1 and press Ctrl key to add more test classes
4. To add the selected test classes to the suite, click >.
5. Click **Save**
6. Select **Test | New Suite Run**.
7. Select the respective test suite and click > to move 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

TempConverterTaskUtilSuite

Available Test Classes

Name ▲

AccountViewerControllerTest

>

<

Selected Test Classes

Name ▲

TaskUtilTest

TemperatureConverterTest

Filter test classes (\* = any)

[My Namespace]

▼

Cancel

Save



# Apex Testing - Test Apex Triggers

## Test Apex Triggers:

- Before deploying a trigger, write unit tests to perform the actions that fire the trigger and verify expected results.
- Let's test a trigger that we worked with in the Apex Triggers module.

Here are some steps for adding a test method to verify an Apex trigger:

1. In the Developer Console, click **File | New | Apex Class**
2. Enter the **class name**
3. Replace the **default class body**
4. **Set up a test**
5. Do the **custom actions**
6. **Verify** that the trigger performed the relevant custom action.

# Apex Testing - Test Apex Triggers

Apex triggers are special types of classes that execute custom actions before or after changes to Salesforce records, such as insertions, updates, or deletions.

## Example:

- Let's say you have a custom object called **Teacher\_\_c** with the following fields:
  - **Teacher\_Name\_\_c** (Text)
  - **Experience\_\_c** (Number)
- You have a trigger on **Teacher\_\_c** that automatically updates the **Experience\_\_c** as **5** whenever a **Teacher\_\_c** record is inserted with **Teacher\_Name\_\_c** = 'Teacher5' otherwise updates the **Experience\_\_c** as **0** .

# Apex Testing - Test Apex Triggers

## Apex Trigger- TeacherRecord.apxt

```
trigger TeacherRecord on Teacher__c (before insert) {  
    for(Teacher__c a:Trigger.new) {  
        if(a.Teacher_Name__c=='Teacher5')  
        {  
            a.Experience__c = 5; }  
        else  
        {  
            a.Experience__c = 0;  
        }  
    }  
}
```

## Apex Test Class- OrderTriggerTest.apxc

```
@isTest public class TeacherRecordTest {
@isTest static void testteacherTrigger1()
{ Teacher__c testteacher = new Teacher__c(Teacher_Name__c = 'Teacher5');
  insert testteacher;
  Teacher__c insertedteacher = [SELECT Experience__c FROM Teacher__c WHERE Id
                                                                    =:testteacher.Id];

  System.assertEquals(5, insertedteacher.Experience__c); }
@isTest static void testteacherTrigger2()
{ Teacher__c testteacher = new Teacher__c(Teacher_Name__c = 'Teacher');
  insert testteacher;
  Teacher__c insertedteacher = [SELECT Experience__c FROM Teacher__c WHERE Id
                                                                    =:testteacher.Id];

  System.assertEquals(0, insertedteacher.Experience__c); }
@isTest static void testteacherTrigger3()
{ Teacher__c testteacher = new Teacher__c(Teacher_Name__c = 'Teacher');
  insert testteacher;
  Teacher__c insertedteacher = [SELECT Experience__c FROM Teacher__c WHERE Id
                                                                    =:testteacher.Id];

  System.assertEquals(5, insertedteacher.Experience__c); } }
```

## Explanation of the Test Class:

**1.@isTest Annotation:** Marks the class and the method as a test class and method, which are used only for testing purposes.

**2.Setup:** Create an instance of **Teacher\_\_c** with **Teacher\_Name\_\_c** set to '**Teacher5**'.

This is the data you'll be using to test the trigger.

**3.Act:** Insert the **Teacher\_\_c** record. This action fires the **TeacherRecord** trigger, which should automatically set the **Experience\_\_c** field based on the trigger's logic.

**4.Assert:** After inserting the record, query it from the database and verify:

- The **Teacher\_Name\_\_c** field is as expected (**Teacher5**).
- The **Experience\_\_c** field is updated with **5**.

# Apex Testing - Create Test Data for Apex Tests

## Create Apex Test Data:

- The **data from Apex tests is only temporary** and is **not saved in the database**.
- 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.
- It is recommended that the **test utility classes be created to add reusable methods** for test data setup.

# Apex Testing - Create Test Data for Apex Tests

- 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.
- **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**.