Once you have a Developer Edition or sandbox organization, you may want to learn some of the core concepts of Apex. After reviewing the basics, you are ready to write your first Apex program—a very simple class, trigger, and unit test.

Because Apex is very similar to Java, you may recognize much of the functionality.

This tutorial is based on a custom object called Book that is created in the first step. This custom object is updated through a trigger.

This Hello World sample requires custom objects. You can either create these on your own, or download the objects and Apex code as an unmanaged package from the Salesforce AppExchange. To obtain the sample assets in your org, install the Apex Tutorials Package. This package also contains sample code and objects for the Shipping Invoice example.

Note: There is a more complex Shipping Invoice example that you can also walk through. That example illustrates many more features of the language.

IN THIS SECTION:

1. Create a Custom Object
   In this step, you create a custom object called Book with one custom field called Price.
2. Adding an Apex Class
   In this step, you add an Apex class that contains a method for updating the book price. This method is called by the trigger that you will be adding in the next step.
3. Add an Apex Trigger
   In this step, you create a trigger for the `Book__c` custom object that calls the `applyDiscount` method of the `MyHelloWorld` class that you created in the previous step.
4. Add a Test Class
   In this step, you add a test class with one test method. You also run the test and verify code coverage. The test method exercises and validates the code in the trigger and class. Also, it enables you to reach 100% code coverage for the trigger and class.
5. Deploying Components to Production
   In this step, you deploy the Apex code and the custom object you created previously to your production organization using change sets.

Create a Custom Object

In this step, you create a custom object called Book with one custom field called Price.

Prerequisites:

A Salesforce account in a sandbox Professional, Enterprise, Performance, or Unlimited Edition org, or an account in a Developer org.

For more information about creating a sandbox org, see "Sandbox Types and Templates" in the Salesforce Help. To sign up for a free Developer org, see the Developer Edition Environment Sign Up Page.

1. Log in to your sandbox or Developer org.
2. From your management settings for custom objects, if you're using Salesforce Classic, click **New Custom Object**, or if you're using Lightning Experience, select **Create** > **Custom Object**.
3. Enter *`Book`* for the label.
4. Enter *`Books`* for the plural label.
5. Click **Save**.
   Ta dah! You've now created your first custom object. Now let's create a custom field.
6. In the **Custom Fields & Relationships** section of the Book detail page, click **New**.

7. Select Number for the data type and click **Next**.
8. Enter `Price` for the field label.
9. Enter 16 in the length text box.
10. Enter 2 in the decimal places text box, and click **Next**.
11. Click **Next** to accept the default values for field-level security.
12. Click **Save**.

You've just created a custom object called Book, and added a custom field to that custom object. Custom objects already have some standard fields, like Name and CreatedBy, and allow you to add other fields that are more specific to your implementation. For this tutorial, the Price field is part of our Book object and it is accessed by the Apex class you will write in the next step.

SEE ALSO:

Salesforce Help: Find Object Management Settings

Adding an Apex Class

In this step, you add an Apex class that contains a method for updating the book price. This method is called by the trigger that you will be adding in the next step.

Prerequisites:

- A Salesforce account in a sandbox Professional, Enterprise, Performance, or Unlimited Edition org, or an account in a Developer org.
- The Book custom object.

1. From Setup, enter "Apex Classes" in the `Quick Find` box, then select **Apex Classes** and click **New**.
2. In the class editor, enter this class definition:

```
public class MyHelloWorld {

}
```

The previous code is the class definition to which you will be adding one method in the next step. Apex code is generally contained in *classes*. This class is defined as `public`, which means the class is available to other Apex classes and triggers. For more information, see Classes, Objects, and Interfaces on page 58.

3. Add this method definition between the class opening and closing brackets.

```
public static void applyDiscount(Book__c[] books) {
   for (Book__c b :books){
      b.Price__c *= 0.9;
   }
}
```

This method is called `applyDiscount`, and it is both public and static. Because it is a static method, you don't need to create an instance of the class to access the method—you can just use the name of the class followed by a dot (.) and the name of the method. For more information, see Static and Instance Methods, Variables, and Initialization Code on page 67.

This method takes one parameter, a list of Book records, which is assigned to the variable `books`. Notice the `__c` in the object name `Book__c`. This indicates that it is a *custom object* that you created. Standard objects that are provided in the Salesforce application, such as Account, don't end with this postfix.

The next section of code contains the rest of the method definition:

```
for (Book__c b :books){
   b.Price__c *= 0.9;
}
```

Notice the `__c` after the field name `Price__c`. This indicates it is a *custom field* that you created. Standard fields that are provided by default in Salesforce are accessed using the same type of dot notation but without the `__c`, for example, `Name` doesn't end with `__c` in `Book__c.Name`. The statement `b.Price__c *= 0.9;` takes the old value of `b.Price__c`, multiplies it by 0.9, which means its value will be discounted by 10%, and then stores the new value into the `b.Price__c` field. The `*=` operator is a shortcut. Another way to write this statement is `b.Price__c = b.Price__c * 0.9;`. See Expression Operators on page 38.

4. Click **Save** to save the new class. You should now have this full class definition.

```
public class MyHelloWorld {
   public static void applyDiscount(Book__c[] books) {
      for (Book__c b :books){
         b.Price__c *= 0.9;
      }
   }
}
```

You now have a class that contains some code that iterates over a list of books and updates the Price field for each book. This code is part of the `applyDiscount` static method called by the trigger that you will create in the next step.

Add an Apex Trigger

In this step, you create a trigger for the `Book__c` custom object that calls the `applyDiscount` method of the `MyHelloWorld` class that you created in the previous step.

Prerequisites:

- A Salesforce account in a sandbox Professional, Enterprise, Performance, or Unlimited Edition org, or an account in a Developer org.
- The MyHelloWorld Apex class.

A *trigger* is a piece of code that executes before or after records of a particular type are inserted, updated, or deleted from the Lightning platform database. Every trigger runs with a set of context variables that provide access to the records that caused the trigger to fire. All triggers run in bulk; that is, they process several records at once.

1. From the object management settings for books, go to Triggers, and then click **New**.
2. In the trigger editor, delete the default template code and enter this trigger definition:

```
trigger HelloWorldTrigger on Book__c (before insert) {

   Book__c[] books = Trigger.new;

   MyHelloWorld.applyDiscount(books);
}
```

The first line of code defines the trigger:

```
trigger HelloWorldTrigger on Book__c (before insert) {
```

It gives the trigger a name, specifies the object on which it operates, and defines the events that cause it to fire. For example, this trigger is called HelloWorldTrigger, it operates on the `Book__c` object, and runs before new books are inserted into the database.

The next line in the trigger creates a list of book records named `books` and assigns it the contents of a trigger context variable called `Trigger.new`. Trigger context variables such as `Trigger.new` are implicitly defined in all triggers and provide access to the records that caused the trigger to fire. In this case, `Trigger.new` contains all the new books that are about to be inserted.

```
Book__c[] books = Trigger.new;
```

The next line in the code calls the method `applyDiscount` in the `MyHelloWorld` class. It passes in the array of new books.

```
MyHelloWorld.applyDiscount(books);
```

You now have all the code that is needed to update the price of all books that get inserted. However, there is still one piece of the puzzle missing. Unit tests are an important part of writing code and are required. In the next step, you will see why this is so and you will be able to add a test class.

SEE ALSO:

Salesforce Help: Find Object Management Settings

Add a Test Class

In this step, you add a test class with one test method. You also run the test and verify code coverage. The test method exercises and validates the code in the trigger and class. Also, it enables you to reach 100% code coverage for the trigger and class.

Prerequisites:

- A Salesforce account in a sandbox Professional, Enterprise, Performance, or Unlimited Edition org, or an account in a Developer org.
- The HelloWorldTrigger Apex trigger.

Note: Testing is an important part of the development process. Before you can deploy Apex or package it for AppExchange, the following must be true.

- Unit tests must cover at least 75% of your Apex code, and all of those tests must complete successfully.

  Note the following.

  - When deploying Apex to a production organization, each unit test in your organization namespace is executed by default.
  - Calls to `System.debug` aren't counted as part of Apex code coverage.
  - Test methods and test classes aren't counted as part of Apex code coverage.
  - While only 75% of your Apex code must be covered by tests, don't focus on the percentage of code that is covered. Instead, make sure that every use case of your application is covered, including positive and negative cases, as well as bulk and single records. This approach ensures that 75% or more of your code is covered by unit tests.

- Every trigger must have some test coverage.
- All classes and triggers must compile successfully.

1. From Setup, enter *Apex Classes* in the `Quick Find` box, then select **Apex Classes** and click **New**.
2. In the class editor, add this test class definition, and then click **Save**.

```
@IsTest
private class HelloWorldTestClass {
    @IsTest
    static void validateHelloWorld() {
       Book__c b = new Book__c(Name='Behind the Cloud', Price__c=100);
       System.debug('Price before inserting new book: ' + b.Price__c);
```

```
        // Insert book
        insert b;

        // Retrieve the new book
        b = [SELECT Price__c FROM Book__c WHERE Id =:b.Id];
        System.debug('Price after trigger fired: ' + b.Price__c);

        // Test that the trigger correctly updated the price
        System.assertEquals(90, b.Price__c);
    }
}
```

This class is defined using the `@IsTest` annotation. Classes defined this way should only contain test methods and any methods required to support those test methods. One advantage to creating a separate class for testing is that classes defined with `@IsTest` don't count against your org's limit of 6 MB of Apex code. You can also add the `@IsTest` annotation to individual methods. For more information, see `@IsTest` Annotation on page 98 and Execution Governors and Limits.

The method `validateHelloWorld` is defined using the `@IsTest` annotation. This annotation means that if changes are made to the database, they're rolled back when execution completes. You don't have to delete any test data created in the test method.

Note: The `@IsTest` annotation on methods is equivalent to the `testMethod` keyword. As best practice, Salesforce recommends that you use `@IsTest` rather than `testMethod`. The `testMethod` keyword may be versioned out in a future release.

First, the test method creates a book and inserts it into the database temporarily. The `System.debug` statement writes the value of the price in the debug log.

```
Book__c b = new Book__c(Name='Behind the Cloud', Price__c=100);
System.debug('Price before inserting new book: ' + b.Price__c);

// Insert book
insert b;
```

After the book is inserted, the code retrieves the newly inserted book, using the ID that was initially assigned to the book when it was inserted. The `System.debug` statement then logs the new price that the trigger modified.

```
// Retrieve the new book
b = [SELECT Price__c FROM Book__c WHERE Id =:b.Id];
System.debug('Price after trigger fired: ' + b.Price__c);
```

When the `MyHelloWorld` class runs, it updates the `Price__c` field and reduces its value by 10%. The following test verifies that the method `applyDiscount` ran and produced the expected result.

```
// Test that the trigger correctly updated the price
System.assertEquals(90, b.Price__c);
```

3. To run this test and view code coverage information, switch to the Developer Console.
4. In the Developer Console, click **Test** > **New Run**.
5. To select your test class, click **HelloWorldTestClass**.
6. To add all methods in the `HelloWorldTestClass` class to the test run, click **Add Selected**.
7. Click **Run**.
   The test result displays in the Tests tab. Optionally, you can expand the test class in the Tests tab to view which methods were run. In this case, the class contains only one test method.

8. The Overall Code Coverage pane shows the code coverage of this test class. To view the percentage of lines of code in the trigger covered by this test, which is 100%, double-click the code coverage line for **HelloWorldTrigger**. Because the trigger calls a method from the `MyHelloWorld` class, this class also has coverage (100%). To view the class coverage, double-click **MyHelloWorld**.
9. To open the log file, in the Logs tab, double-click the most recent log line in the list of logs. The execution log displays, including logging information about the trigger event, the call to the `applyDiscount` method, and the price before and after the trigger.

By now, you've completed all the steps necessary for writing some Apex code with a test that runs in your development environment. In the real world, after you've tested your code and are satisfied with it, you want to deploy the code and any prerequisite components to a production org. The next step shows you how to do this deployment for the code and custom object you've created.

SEE ALSO:

Salesforce Help: Open the Developer Console

Deploying Components to Production

In this step, you deploy the Apex code and the custom object you created previously to your production organization using change sets.

Prerequisites:

- A Salesforce account in a sandbox **Performance**, **Unlimited**, or **Enterprise** Edition organization.
- The HelloWorldTestClass Apex test class.
- A deployment connection between the sandbox and production organizations that allows inbound change sets to be received by the production organization. See "Change Sets" in the Salesforce online help.
- "Create and Upload Change Sets" user permission to create, edit, or upload outbound change sets.

This procedure doesn't apply to Developer organizations since change sets are available only in **Performance**, **Unlimited**, **Enterprise**, or Database.com Edition organizations. If you have a Developer Edition account, you can use other deployment methods. For more information, see Deploying Apex.

1. From Setup, enter *`Outbound Changesets`* in the `Quick Find` box, then select **Outbound Changesets**.
2. If a splash page appears, click **Continue**.
3. In the Change Sets list, click **New**.
4. Enter a name for your change set, for example, *`HelloWorldChangeSet`*, and optionally a description. Click **Save**.
5. In the Change Set Components section, click **Add**.
6. Select Apex Class from the component type drop-down list, then select the MyHelloWorld and the HelloWorldTestClass classes from the list and click **Add to Change Set**.
7. Click **View/Add Dependencies** to add the dependent components.
8. Select the top checkbox to select all components. Click **Add To Change Set**.
9. In the Change Set Detail section of the change set page, click **Upload**.
10. Select the target organization, in this case production, and click **Upload**.
11. After the change set upload completes, deploy it in your production organization.
    a. Log into your production organization.
    b. From Setup, enter *`Inbound Change Sets`* in the `Quick Find` box, then select **Inbound Change Sets**.
    c. If a splash page appears, click **Continue**.
    d. In the change sets awaiting deployment list, click your change set's name.