Learning Objectives

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

* Write a trigger for a Salesforce object.
* Use trigger context variables.
* Call a class method from a trigger.
* Use the sObject addError() method in a trigger to restrict save operations.

Writing Apex Triggers

Apex triggers enable you to perform custom actions before or after events to records in Salesforce, such as insertions, updates, or deletions. Just like database systems support triggers, Apex provides trigger support for managing records.

Typically, you use triggers to perform operations based on specific conditions, to modify related records or restrict certain operations from happening. You can use triggers to do anything you can do in Apex, including executing SOQL and DML or calling custom Apex methods.

Use triggers to perform tasks that can’t be done by using the point-and-click tools in the Salesforce user interface. For example, if validating a field value or updating a field on a record, use validation rules and workflow rules instead.

Triggers can be defined for top-level standard objects, such as Account or Contact, custom objects, and some standard child objects. Triggers are active by default when created. Salesforce automatically fires active triggers when the specified database events occur.

Trigger Syntax

The syntax of a trigger definition is different from a class definition’s syntax. A trigger definition starts with the trigger keyword. It is then followed by the name of the trigger, the Salesforce object that the trigger is associated with, and the conditions under which it fires. A trigger has the following syntax:

trigger TriggerName on ObjectName (trigger\_events) {

code\_block

}

Copy

To execute a trigger before or after insert, update, delete, and undelete operations, specify multiple trigger events in a comma-separated list. The events you can specify are:

* before insert
* before update
* before delete
* after insert
* after update
* after delete
* after undelete

**Example**

This simple trigger fires before you insert an account and writes a message to the debug log.

1. In the Developer Console, click **File** | **New** | **Apex Trigger**.
2. Enter HelloWorldTrigger for the trigger name, and then select Account for the sObject. Click **Submit**.
3. Replace the default code with the following.
4. trigger HelloWorldTrigger on Account (before insert) {
5. System.**debug**(**'Hello World!'**);

}

Copy

1. To save, press **Ctrl+S**.
2. To test the trigger, create an account.
   1. Click **Debug** | **Open Execute Anonymous Window**.
   2. In the new window, add the following and then click **Execute**.
   3. Account a = new Account(Name=**'Test Trigger'**);

insert a;

Copy

1. In the debug log, find the Hello World! statement. The log also shows that the trigger has been executed.

Types of Triggers

There are two types of triggers.

* ***Before triggers*** are used to update or validate record values before they’re saved to the database.
* ***After triggers*** are used to access field values that are set by the system (such as a record's Id or LastModifiedDatefield), and to affect changes in other records. The records that fire the *after trigger* are read-only.

Using Context Variables

To access the records that caused the trigger to fire, use context variables. For example, Trigger.New contains all the records that were inserted in insert or update triggers. Trigger.Old provides the old version of sObjects before they were updated in update triggers, or a list of deleted sObjects in delete triggers. Triggers can fire when one record is inserted, or when many records are inserted in bulk via the API or Apex. Therefore, context variables, such as Trigger.New, can contain only one record or multiple records. You can iterate over Trigger.New to get each individual sObject.

This example is a modified version of the HelloWorldTrigger example trigger. It iterates over each account in a for loop and updates the Description field for each.

trigger HelloWorldTrigger on Account (before insert) {

for(Account a : Trigger.New) {

a.Description = **'New description'**;

}

}

Copy

**Note**

The system saves the records that fired the before trigger after the trigger finishes execution. You can modify the records in the trigger without explicitly calling a DML insert or update operation. If you perform DML statements on those records, you get an error.

Some other context variables return a Boolean value to indicate whether the trigger was fired due to an update or some other event. These variables are useful when a trigger combines multiple events. For example:

trigger ContextExampleTrigger on Account (before insert, after insert, after delete) {

if (Trigger.isInsert) {

if (Trigger.isBefore) {

// Process before insert

} else if (Trigger.isAfter) {

// Process after insert

}

}

else if (Trigger.isDelete) {

// Process after delete

}

}

Copy

Trigger Context Variables

The following table is a comprehensive list of all context variables available for triggers.

| **Variable** | **Usage** |
| --- | --- |
| isExecuting | Returns true if the current context for the Apex code is a trigger, not a Visualforce page, a Web service, or an executeanonymous() API call. |
| isInsert | Returns true if this trigger was fired due to an insert operation, from the Salesforce user interface, Apex, or the API. |
| isUpdate | Returns true if this trigger was fired due to an update operation, from the Salesforce user interface, Apex, or the API. |
| isDelete | Returns true if this trigger was fired due to a delete operation, from the Salesforce user interface, Apex, or the API. |
| isBefore | Returns true if this trigger was fired before any record was saved. |
| isAfter | Returns true if this trigger was fired after all records were saved. |
| isUndelete | Returns true if this trigger was fired after a record is recovered from the Recycle Bin (that is, after an undelete operation from the Salesforce user interface, Apex, or the API.) |
| new | Returns a list of the new versions of the sObject records.  This sObject list is only available in insert, update, and undelete triggers, and the records can only be modified in before triggers. |
| newMap | A map of IDs to the new versions of the sObject records.  This map is only available in before update, after insert, after update, and after undelete triggers. |
| old | Returns a list of the old versions of the sObject records.  This sObject list is only available in update and delete triggers. |
| oldMap | A map of IDs to the old versions of the sObject records.  This map is only available in update and delete triggers. |
| size | The total number of records in a trigger invocation, both old and new. |

Calling a Class Method from a Trigger

You can call public utility methods from a trigger. Calling methods of other classes enables code reuse, reduces the size of your triggers, and improves maintenance of your Apex code. It also allows you to use object-oriented programming.

The following example trigger shows how to call a static method from a trigger. If the trigger was fired because of an insert event, the example calls the static sendMail() method on the EmailManager class. This utility method sends an email to the specified recipient and contains the number of contact records inserted.

**Note**

The EmailManager class is included in the class example of the Get Started with Apex unit. You must have saved the EmailManager class in your org and changed the sendMail() method to static before saving this trigger.

1. In the Developer Console, click **File** | **New** | **Apex Trigger**.
2. Enter ExampleTrigger for the trigger name, and then select **Contact** for the sObject. Click **Submit**.
3. Replace the default code with the following, and then modify the email address placeholder text in sendMail()to your email address.
4. trigger ExampleTrigger on Contact (after insert, after delete) {
5. if (Trigger.isInsert) {
6. Integer recordCount = Trigger.New.**size**();
7. // Call a utility method from another class
8. EmailManager.**sendMail**(**'Your email address'**, **'Trailhead Trigger Tutorial'**,
9. recordCount + **' contact(s) were inserted.'**);
10. }
11. else if (Trigger.isDelete) {
12. // Process after delete
13. }

}

Copy

1. To save, press **Ctrl+S**.
2. To test the trigger, create a contact.
   1. Click **Debug** | **Open Execute Anonymous Window**.
   2. In the new window, add the following and then click **Execute**.
   3. Contact c = new Contact(LastName=**'Test Contact'**);

insert c;

Copy

1. In the debug log, check that the trigger was fired. Toward the end of the log, find the debug message that was written by the utility method: DEBUG|Email sent successfully
2. Now check that you received an email with the body text 1 contact(s) were inserted.

With your new trigger in place, you get an email every time you add one or more contacts!

Adding Related Records

Triggers are often used to access and manage records related to the records in the trigger context—the records that caused this trigger to fire.

This trigger adds a related opportunity for each new or updated account if no opportunity is already associated with the account. The trigger first performs a SOQL query to get all child opportunities for the accounts that the trigger fired on. Next, the trigger iterates over the list of sObjects in Trigger.New to get each account sObject. If the account doesn’t have any related opportunity sObjects, the for loop creates one. If the trigger created any new opportunities, the final statement inserts them.

1. Add the following trigger using the Developer Console (follow the steps of the HelloWorldTrigger example but useAddRelatedRecord for the trigger name).
2. trigger AddRelatedRecord on **Account**(after insert, after update) {
3. List<Opportunity> oppList = new List<Opportunity>();
5. // Get the related opportunities for the accounts in this trigger
6. Map<Id,Account> acctsWithOpps = new Map<Id,Account>(
7. [SELECT Id,(SELECT Id FROM Opportunities) FROM Account WHERE Id IN :Trigger.New]);
9. // Add an opportunity for each account if it doesn't already have one.
10. // Iterate through each account.
11. for(Account a : Trigger.New) {
12. System.**debug**(**'acctsWithOpps.get(a.Id).Opportunities.size()='** + acctsWithOpps.**get**(a.Id).Opportunities.**size**());
13. // Check if the account already has a related opportunity.
14. if (acctsWithOpps.**get**(a.Id).Opportunities.**size**() == 0) {
15. // If it doesn't, add a default opportunity
16. oppList.**add**(new Opportunity(Name=a.Name + **' Opportunity'**,
17. StageName=**'Prospecting'**,
18. CloseDate=System.**today**().**addMonths**(1),
19. AccountId=a.Id));
20. }
21. }
22. if (oppList.**size**() > 0) {
23. insert oppList;
24. }

}

Copy

1. To test the trigger, create an account in the Salesforce user interface and name it Apples & Oranges.
2. In the Opportunities related list on the account’s page, find the new opportunity. The trigger added this opportunity automatically!

**Beyond the Basics**

The trigger you’ve added iterates over all records that are part of the trigger context—the for loop iterates over Trigger.New. However, the loop in this trigger could be more efficient. We don’t really need to access every account in this trigger context, but only a subset—the accounts without opportunities. The next unit shows how to make this trigger more efficient. In the Bulk Trigger Design Patterns unit, learn how to modify the SOQL query to get only the accounts with no opportunities. Then, learn to iterate only over those records.

Using Trigger Exceptions

You sometimes need to add restrictions on certain database operations, such as preventing records from being saved when certain conditions are met. To prevent saving records in a trigger, call the addError() method on the sObject in question. The addError() method throws a fatal error inside a trigger. The error message is displayed in the user interface and is logged.

The following trigger prevents the deletion of an account if it has related opportunities. By default, deleting an account causes a cascade delete of all its related records. This trigger prevents the cascade delete of opportunities. Try this trigger for yourself! If you’ve executed the previous example, your org has an account called Apples & Oranges with a related opportunity. This example uses that sample account.

1. Using the Developer Console, add the following trigger.
2. trigger AccountDeletion on Account (before delete) {
4. // Prevent the deletion of accounts if they have related opportunities.
5. for (Account a : [SELECT Id FROM Account
6. WHERE Id IN (SELECT AccountId FROM Opportunity) AND
7. Id IN :Trigger.old]) {
8. Trigger.oldMap.**get**(a.Id).**addError**(
9. **'Cannot delete account with related opportunities.'**);
10. }

}

Copy

1. In the Salesforce user interface, navigate to the Apples & Oranges account’s page and click **Delete**.
2. In the confirmation popup, click **OK**.

Find the validation error with the custom error message Cannot delete account with related opportunities.

1. Disable the AccountDeletion trigger. If you leave this trigger active, you can’t check your challenges.
   1. From Setup, search for Apex Triggers.
   2. On the Apex Triggers page, click **Edit** next to the AccountDeletion trigger.
   3. Deselect **Is Active**.
   4. Click **Save**.

**Beyond the Basics**

Calling addError() in a trigger causes the entire set of operations to roll back, except when bulk DML is called with partial success.

* If a DML statement in Apex spawned the trigger, any error rolls back the entire operation. However, the runtime engine still processes every record in the operation to compile a comprehensive list of errors.
* If a bulk DML call in the Lightning Platform API spawned the trigger, the runtime engine sets the bad records aside. The runtime engine then attempts a partial save of the records that did not generate errors.

Triggers and Callouts

Apex allows you to make calls to and integrate your Apex code with external Web services. Apex calls to external Web services are referred to as callouts. For example, you can make a callout to a stock quote service to get the latest quotes. When making a callout from a trigger, the callout must be done asynchronously so that the trigger process doesn’t block you from working while waiting for the external service's response.The asynchronous callout is made in a background process, and the response is received when the external service returns it.

To make a callout from a trigger, call a class method that executes asynchronously. Such a method is called a future method and is annotated with @future(callout=true). This example class contains the future method that makes the callout.

**Note**

The example uses a hypothetical endpoint URL for illustration purposes only. You can’t run this example unless you change the endpoint to a valid URL and add a remote site in Salesforce for your endpoint.

public class CalloutClass {

@future(callout=true)

public static void **makeCallout**() {

HttpRequest request = new HttpRequest();

// Set the endpoint URL.

String endpoint = **'http://yourHost/yourService'**;

request.**setEndPoint**(endpoint);

// Set the HTTP verb to GET.

request.**setMethod**(**'GET'**);

// Send the HTTP request and get the response.

HttpResponse response = new HTTP().**send**(request);

}

}

Copy

This example shows the trigger that calls the method in the class to make a callout asynchronously.

trigger CalloutTrigger on Account (before insert, before update) {

CalloutClass.**makeCallout**();

}

DML

## DML Statements

Let us now consider the instance of the Chemical supplier company again. Our Invoice records have fields as Status, Amount Paid, Amount Remaining, Next Pay Date and Invoice Number. Invoices which have been created today and have their status as 'Pending', should be updated to 'Paid'.

## Insert Operation

Insert operation is used to create new records in Database. You can create records of any Standard or Custom object using the Insert DML statement.

**Example**

We can create new records in APEX\_Invoice\_\_c object as new invoices are being generated for new customer orders every day. We will create a Customer record first and then we can create an Invoice record for that new Customer record.

// fetch the invoices created today, Note, you must have at least one invoice

// created today

List<apex\_invoice\_\_c> invoiceList = [SELECT id, Name, APEX\_Status\_\_c,

createdDate FROM APEX\_Invoice\_\_c WHERE createdDate = today];

// create List to hold the updated invoice records

List<apex\_invoice\_\_c> updatedInvoiceList = new List<apex\_invoice\_\_c>();

APEX\_Customer\_\_c objCust = new APEX\_Customer\_\_C();

objCust.Name = 'Test ABC';

//DML for Inserting the new Customer Records

insert objCust;

for (APEX\_Invoice\_\_c objInvoice: invoiceList) {

if (objInvoice.APEX\_Status\_\_c == 'Pending') {

objInvoice.APEX\_Status\_\_c = 'Paid';

updatedInvoiceList.add(objInvoice);

}

}

// DML Statement to update the invoice status

update updatedInvoiceList;

// Prints the value of updated invoices

System.debug('List has been updated and updated values are' + updatedInvoiceList);

// Inserting the New Records using insert DML statement

APEX\_Invoice\_\_c objNewInvoice = new APEX\_Invoice\_\_c();

objNewInvoice.APEX\_Status\_\_c = 'Pending';

objNewInvoice.APEX\_Amount\_Paid\_\_c = 1000;

objNewInvoice.APEX\_Customer\_\_c = objCust.id;

// DML which is creating the new Invoice record which will be linked with newly

// created Customer record

insert objNewInvoice;

System.debug('New Invoice Id is '+objNewInvoice.id+' and the Invoice Number is'

+ objNewInvoice.Name);

## Update Operation

Update operation is to perform updates on existing records. In this example, we will be updating the Status field of an existing Invoice record to 'Paid'.

**Example**

// Update Statement Example for updating the invoice status. You have to create

and Invoice records before executing this code. This program is updating the

record which is at index 0th position of the List.

// First, fetch the invoice created today

List<apex\_invoice\_\_c> invoiceList = [SELECT id, Name, APEX\_Status\_\_c,

createdDate FROM APEX\_Invoice\_\_c];

List<apex\_invoice\_\_c> updatedInvoiceList = new List<apex\_invoice\_\_c>();

// Update the first record in the List

invoiceList[0].APEX\_Status\_\_c = 'Pending';

updatedInvoiceList.add(invoiceList[0]);

// DML Statement to update the invoice status

update updatedInvoiceList;

// Prints the value of updated invoices

System.debug('List has been updated and updated values of records are'

+ updatedInvoiceList[0]);

## Upsert Operation

Upsert Operation is used to perform an update operation and if the records to be updated are not present in database, then create new records as well.

**Example**

Suppose, the customer records in Customer object need to be updated. We will update the existing Customer record if it is already present, else create a new one. This will be based on the value of field APEX\_External\_Id\_\_c. This field will be our field to identify if the records are already present or not.

**Note** − Before executing this code, please create a record in Customer object with the external Id field value as '12341' and then execute the code given below −

// Example for upserting the Customer records

List<apex\_customer\_\_c> CustomerList = new List<apex\_customer\_\_c>();

for (Integer i = 0; i < 10; i++) {

apex\_customer\_\_c objcust=new apex\_customer\_\_c(name = 'Test' +i,

apex\_external\_id\_\_c='1234' +i);

customerlist.add(objcust);

} //Upserting the Customer Records

upsert CustomerList;

System.debug('Code iterated for 10 times and created 9 records as one record with

External Id 12341 is already present');

for (APEX\_Customer\_c objCustomer: CustomerList) {

if (objCustomer.APEX\_External\_Id\_c == '12341') {

system.debug('The Record which is already present is '+objCustomer);

}

}

## Delete Operation

You can perform the delete operation using the Delete DML.

**Example**

In this case, we will delete the invoices which have been created for the testing purpose, that is the ones which contain the name as 'Test'.

You can execute this snippet from the Developer console as well without creating the class.

// fetch the invoice created today

List<apex\_invoice\_\_c> invoiceList = [SELECT id, Name, APEX\_Status\_\_c,

createdDate FROM APEX\_Invoice\_\_c WHERE createdDate = today];

List<apex\_invoice\_\_c> updatedInvoiceList = new List<apex\_invoice\_\_c>();

APEX\_Customer\_\_c objCust = new APEX\_Customer\_\_C();

objCust.Name = 'Test';

// Inserting the Customer Records

insert objCust;

for (APEX\_Invoice\_\_c objInvoice: invoiceList) {

if (objInvoice.APEX\_Status\_\_c == 'Pending') {

objInvoice.APEX\_Status\_\_c = 'Paid';

updatedInvoiceList.add(objInvoice);

}

}

// DML Statement to update the invoice status

update updatedInvoiceList;

// Prints the value of updated invoices

System.debug('List has been updated and updated values are' + updatedInvoiceList);

// Inserting the New Records using insert DML statement

APEX\_Invoice\_\_c objNewInvoice = new APEX\_Invoice\_\_c();

objNewInvoice.APEX\_Status\_\_c = 'Pending';

objNewInvoice.APEX\_Amount\_Paid\_\_c = 1000;

objNewInvoice.APEX\_Customer\_\_c = objCust.id;

// DML which is creating the new record

insert objNewInvoice;

System.debug('New Invoice Id is' + objNewInvoice.id);

// Deleting the Test invoices from Database

// fetch the invoices which are created for Testing, Select name which Customer Name

// is Test.

List<apex\_invoice\_\_c> invoiceListToDelete = [SELECT id FROM APEX\_Invoice\_\_c

WHERE APEX\_Customer\_\_r.Name = 'Test'];

// DML Statement to delete the Invoices

delete invoiceListToDelete;

System.debug('Success, '+invoiceListToDelete.size()+' Records has been deleted');

## Undelete Operation

You can undelete the record which has been deleted and is present in Recycle bin. All the relationships which the deleted record has, will also be restored.

**Example**

Suppose, the Records deleted in the previous example need to be restored. This can be achieved using the following example. The code in the previous example has been modified for this example.

// fetch the invoice created today

List<apex\_invoice\_\_c> invoiceList = [SELECT id, Name, APEX\_Status\_\_c,

createdDate FROM APEX\_Invoice\_\_c WHERE createdDate = today];

List<apex\_invoice\_\_c> updatedInvoiceList = new List<apex\_invoice\_\_c>();

APEX\_Customer\_\_c objCust = new APEX\_Customer\_\_C();

objCust.Name = 'Test';

// Inserting the Customer Records

insert objCust;

for (APEX\_Invoice\_\_c objInvoice: invoiceList) {

if (objInvoice.APEX\_Status\_\_c == 'Pending') {

objInvoice.APEX\_Status\_\_c = 'Paid';

updatedInvoiceList.add(objInvoice);

}

}

// DML Statement to update the invoice status

update updatedInvoiceList;

// Prints the value of updated invoices

System.debug('List has been updated and updated values are' + updatedInvoiceList);

// Inserting the New Records using insert DML statement

APEX\_Invoice\_\_c objNewInvoice = new APEX\_Invoice\_\_c();

objNewInvoice.APEX\_Status\_\_c = 'Pending';

objNewInvoice.APEX\_Amount\_Paid\_\_c = 1000;

objNewInvoice.APEX\_Customer\_\_c = objCust.id;

// DML which is creating the new record

insert objNewInvoice;

System.debug('New Invoice Id is '+objNewInvoice.id);

// Deleting the Test invoices from Database

// fetch the invoices which are created for Testing, Select name which Customer Name

// is Test.

List<apex\_invoice\_\_c> invoiceListToDelete = [SELECT id FROM APEX\_Invoice\_\_c

WHERE APEX\_Customer\_\_r.Name = 'Test'];

// DML Statement to delete the Invoices

delete invoiceListToDelete;

system.debug('Deleted Record Count is ' + invoiceListToDelete.size());

System.debug('Success, '+invoiceListToDelete.size() + 'Records has been deleted');

// Restore the deleted records using undelete statement

undelete invoiceListToDelete;

System.debug('Undeleted Record count is '+invoiceListToDelete.size()+'. This should

be same as Deleted Record count');

# Apex - SOSL

## SOSL

Searching the text string across the object and across the field will be done by using SOSL. This is Salesforce Object Search Language. It has the capability of searching a particular string across multiple objects.

SOSL statements evaluate to a list of sObjects, wherein, each list contains the search results for a particular sObject type. The result lists are always returned in the same order as they were specified in the SOSL query.

### SOSL Query Example

Consider a business case wherein, we need to develop a program which can search a specified string. Suppose, we need to search for string 'ABC' in the Customer Name field of Invoice object. The code goes as follows −

First, you have to create a single record in Invoice object with Customer name as 'ABC' so that we can get valid result when searched.

// Program To Search the given string in all Object

// List to hold the returned results of sObject generic type

List<list<SObject>> invoiceSearchList = new List<List<SObject>>();

// SOSL query which will search for 'ABC' string in Customer Name field of Invoice Object

invoiceSearchList = [FIND 'ABC\*' IN ALL FIELDS RETURNING APEX\_Invoice\_c

(Id,APEX\_Customer\_r.Name)];

// Returned result will be printed

System.debug('Search Result '+invoiceSearchList);

// Now suppose, you would like to search string 'ABC' in two objects,

// that is Invoice and Account. Then for this query goes like this:

// Program To Search the given string in Invoice and Account object,

// you could specify more objects if you want, create an Account with Name as ABC.

// List to hold the returned results of sObject generic type

List<List<SObject>> invoiceAndSearchList = new List<List<SObject>>();

// SOSL query which will search for 'ABC' string in Invoice and in Account object's fields

invoiceAndSearchList = [FIND 'ABC\*' IN ALL FIELDS RETURNING APEX\_Invoice\_\_c

(Id,APEX\_Customer\_\_r.Name), Account];

// Returned result will be printed

System.debug('Search Result '+invoiceAndSearchList);

// This list will hold the returned results for Invoice Object

APEX\_Invoice\_\_c [] searchedInvoice = ((List<APEX\_Invoice\_c>)invoiceAndSearchList[0]);

// This list will hold the returned results for Account Object

Account [] searchedAccount = ((List<Account>)invoiceAndSearchList[1]);

System.debug('Value of searchedInvoice'+searchedInvoice+'Value of searchedAccount'

+ searchedAccount);

## SOQL

This is almost the same as SOQL. You can use this to fetch the object records from one object only at a time. You can write nested queries and also fetch the records from parent or child object on which you are querying now.

# Apex - SOQL

## SOQL Example

Consider our ongoing example of Chemical Company. Suppose, we need a list of records which are created today and whose customer name is not 'test'. In this case, we will have to use the SOQL query as given below −

// fetching the Records via SOQL

List<apex\_invoice\_\_c> InvoiceList = new List<apex\_invoice\_\_c>();

InvoiceList = [SELECT Id, Name, APEX\_Customer\_\_r.Name, APEX\_Status\_\_c FROM

APEX\_Invoice\_\_c WHERE createdDate = today AND APEX\_Customer\_\_r.Name != 'Test'];

// SOQL query for given criteria

// Printing the fetched records

System.debug('We have total '+InvoiceList.size()+' Records in List');

for (APEX\_Invoice\_\_c objInvoice: InvoiceList) {

System.debug('Record Value is '+objInvoice);

// Printing the Record fetched

}

You can run the SOQL query via the Query Editor in the Developer console as shown below.

Run the query given below in the Developer Console. Search for the Invoice records created today.

SELECT Id, Name, APEX\_Customer\_\_r.Name, APEX\_Status\_\_c FROM APEX\_Invoice\_\_c

WHERE createdDate = today

You must select the fields for which you need the values, otherwise, it can throw run time errors.

## Traversing Relationship Fields

This is one of the most important parts in SFDC as many times we need to traverse through the parent child object relationship

Also, there may be cases when you need to insert two associated objects records in Database. For example, Invoice object has relationship with the Customer object and hence one Customer can have many invoices.

Suppose, you are creating the invoice and then you need to relate this invoice to Customer. You can use the following code for this functionality −

// Now create the invoice record and relate it with the Customer object

// Before executing this, please create a Customer Records with Name 'Customer

// Creation Test'

APEX\_Invoice\_\_c objInvoice = new APEX\_Invoice\_\_c();

// Relating Invoice to customer via id field of Customer object

objInvoice.APEX\_Customer\_\_c = [SELECT id FROM APEX\_Customer\_\_c WHERE Name =

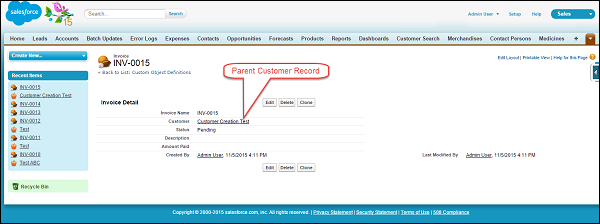
'Customer Creation Test' LIMIT 1].id;

objInvoice.APEX\_Status\_\_c = 'Pending';

insert objInvoice; //Creating Invoice

System.debug('Newly Created Invoice'+objInvoice); //Newly created invoice

Execute this code snippet in the Developer Console. Once executed, copy the Id of invoice from the Developer console and then open the same in SFDC as shown below. You can see that the Parent record has already been assigned to Invoice record as shown below.



## Fetching Child Records

Let us now consider an example wherein, all the invoices related to particular customer record need to be in one place. For this, you must know the child relationship name. To see the child relationship name, go to the field detail page on the child object and check the "Child Relationship" value. In our example, it is invoices appended by \_\_r at the end.

### Example

In this example, we will need to set up data, create a customer with name as 'ABC Customer' record and then add 3 invoices to that customer.

Now, we will fetch the invoices the Customer 'ABC Customer' has. Following is the query for the same −

// Fetching Child Records using SOQL

List<apex\_customer\_\_c> ListCustomers = [SELECT Name, Id,

(SELECT id, Name FROM Invoices\_\_r) FROM APEX\_Customer\_\_c WHERE Name = 'ABC Customer'];

// Query for fetching the Child records along with Parent

System.debug('ListCustomers '+ListCustomers); // Parent Record

List<apex\_invoice\_\_c> ListOfInvoices = ListCustomers[0].Invoices\_\_r;

// By this notation, you could fetch the child records and save it in List

System.debug('ListOfInvoices values of Child '+ListOfInvoices);

// Child records

You can see the Record values in the Debug logs.

## Fetching Parent Record

Suppose, you need to fetch the Customer Name of Invoice the creation date of which is today, then you can use the query given below for the same −

### Example

Fetch the Parent record's value along with the child object.

// Fetching Parent Record Field value using SOQL

List<apex\_invoice\_\_c> ListOfInvoicesWithCustomerName = new List<apex\_invoice\_\_c>();

ListOfInvoicesWithCustomerName = [SELECT Name, id, APEX\_Customer\_\_r.Name

FROM APEX\_Invoice\_\_c LIMIT 10];

// Fetching the Parent record's values

for (APEX\_Invoice\_\_c objInv: ListOfInvoicesWithCustomerName) {

System.debug('Invoice Customer Name is '+objInv.APEX\_Customer\_\_r.Name);

// Will print the values, all the Customer Records will be printed

}

Here we have used the notation APEX\_Customer\_\_r.Name, where APEX\_Customer\_\_r is parent relationship name, here you have to append the \_\_r at the end of the Parent field and then you can fetch the parent field value.

## Aggregate Functions

SOQL does have aggregate function as we have in SQL. Aggregate functions allow us to roll up and summarize the data. Let us now understand the function in detail.

Suppose, you wanted to know that what is the average revenue we are getting from Customer 'ABC Customer', then you can use this function to take up the average.

### Example

// Getting Average of all the invoices for a Perticular Customer

AggregateResult[] groupedResults = [SELECT

AVG(APEX\_Amount\_Paid\_\_c)averageAmount FROM APEX\_Invoice\_\_c WHERE

APEX\_Customer\_\_r.Name = 'ABC Customer'];

Object avgPaidAmount = groupedResults[0].get('averageAmount');

System.debug('Total Average Amount Received From Customer ABC is '+avgPaidAmount);

Check the output in Debug logs. Note that any query that includes an aggregate function returns its results in an array of **AggregateResult**objects. AggregateResult is a readonly sObject and is only used for query results. It is useful when we need to generate the Report on Large data.

There are other aggregate functions as well which you can be used to perform data summary.

**MIN()** − This can be used to find the minimum value

**MAX()** − This can be used to find the maximum value.

## Binding Apex Variables

You can use the Apex variable in SOQL query to fetch the desired results. Apex variables can be referenced by the Colon (:) notation.

### Example

// Apex Variable Reference

String CustomerName = 'ABC Customer';

List<apex\_customer\_\_c> ListCustomer = [SELECT Id, Name FROM APEX\_Customer\_\_c

WHERE Name = :CustomerName];

// Query Using Apex variable

System.debug('ListCustomer Name'+ListCustomer); // Customer Name

# Apex - Collections

Lists

List can contain any number of records of primitive, collections, sObjects, user defined and built in Apex type. This is one of the most important type of collection and also, it has some system methods which have been tailored specifically to use with List. List index always starts with 0. This is synonymous to the array in Java. A list should be declared with the keyword 'List'.

**Example**

Below is the list which contains the List of a primitive data type (string), that is the list of cities.

List<string> ListOfCities = new List<string>();

System.debug('Value Of ListOfCities'+ListOfCities);

Declaring the initial values of list is optional. However, we will declare the initial values here. Following is an example which shows the same.

List<string> ListOfStates = new List<string> {'NY', 'LA', 'LV'};

System.debug('Value ListOfStates'+ListOfStates);

List of Accounts (sObject)

List<account> AccountToDelete = new List<account> (); //This will be null

System.debug('Value AccountToDelete'+AccountToDelete);

We can declare the nested List as well. It can go up to five levels. This is called the Multidimensional list.

This is the list of set of integers.

List<List<Set<Integer>>> myNestedList = new List<List<Set<Integer>>>();

System.debug('value myNestedList'+myNestedList);

List can contain any number of records, but there is a limitation on heap size to prevent the performance issue and monopolizing the resources.

Methods for Lists

There are methods available for Lists which we can be utilized while programming to achieve some functionalities like calculating the size of List, adding an element, etc.

Following are some most frequently used methods −

* size()
* add()
* get()
* clear()
* set()

The following example demonstrates the use of all these methods

// Initialize the List

List<string> ListOfStatesMethod = new List<string>();

// This statement would give null as output in Debug logs

System.debug('Value of List'+ ListOfStatesMethod);

// Add element to the list using add method

ListOfStatesMethod.add('New York');

ListOfStatesMethod.add('Ohio');

// This statement would give New York and Ohio as output in Debug logs

System.debug('Value of List with new States'+ ListOfStatesMethod);

// Get the element at the index 0

String StateAtFirstPosition = ListOfStatesMethod.get(0);

// This statement would give New York as output in Debug log

System.debug('Value of List at First Position'+ StateAtFirstPosition);

// set the element at 1 position

ListOfStatesMethod.set(0, 'LA');

// This statement would give output in Debug log

System.debug('Value of List with element set at First Position' + ListOfStatesMethod[0]);

// Remove all the elements in List

ListOfStatesMethod.clear();

// This statement would give output in Debug log

System.debug('Value of List'+ ListOfStatesMethod);

You can use the array notation as well to declare the List, as given below, but this is not general practice in Apex programming −

String [] ListOfStates = new List<string>();

Sets

A Set is a collection type which contains multiple number of unordered unique records. A Set cannot have duplicate records. Like Lists, Sets can be nested.

**Example**

We will be defining the set of products which company is selling.

Set<string> ProductSet = new Set<string>{'Phenol', 'Benzene', 'H2SO4'};

System.debug('Value of ProductSet'+ProductSet);

Methods for Sets

Set does support methods which we can utilize while programming as shown below (we are extending the above example) −

// Adds an element to the set

// Define set if not defined previously

Set<string> ProductSet = new Set<string>{'Phenol', 'Benzene', 'H2SO4'};

ProductSet.add('HCL');

System.debug('Set with New Value '+ProductSet);

// Removes an element from set

ProductSet.remove('HCL');

System.debug('Set with removed value '+ProductSet);

// Check whether set contains the particular element or not and returns true or false

ProductSet.contains('HCL');

System.debug('Value of Set with all values '+ProductSet);

Maps

It is a key value pair which contains the unique key for each value. Both key and value can be of any data type.

**Example**

The following example represents the map of the Product Name with the Product code.

// Initialize the Map

Map<string, string> ProductCodeToProductName = new Map<string, string>

{'1000'=>'HCL', '1001'=>'H2SO4'};

// This statement would give as output as key value pair in Debug log

System.debug('value of ProductCodeToProductName'+ProductCodeToProductName);

Methods for Maps

Following are a few examples which demonstrate the methods that can be used with Map −

// Define a new map

Map<string, string> ProductCodeToProductName = new Map<string, string>();

// Insert a new key-value pair in the map where '1002' is key and 'Acetone' is value

ProductCodeToProductName.put('1002', 'Acetone');

// Insert a new key-value pair in the map where '1003' is key and 'Ketone' is value

ProductCodeToProductName.put('1003', 'Ketone');

// Assert that the map contains a specified key and respective value

System.assert(ProductCodeToProductName.containsKey('1002'));

System.debug('If output is true then Map contains the key and output is:'

+ ProductCodeToProductName.containsKey('1002'));

// Retrieves a value, given a particular key

String value = ProductCodeToProductName.get('1002');

System.debug('Value at the Specified key using get function: '+value);

// Return a set that contains all of the keys in the map

Set SetOfKeys = ProductCodeToProductName.keySet();

System.debug('Value of Set with Keys '+SetOfKeys);

Map values may be unordered and hence we should not rely on the order in which the values are stored and try to access the map always using keys. Map value can be null. Map keys when declared String are case-sensitive; for example, ABC and abc will be considered as different keys and treated as unique.

# Apex - Classes

What is a Class?

A class is a template or blueprint from which objects are created. An object is an instance of a class. This is the standard definition of Class. Apex Classes are similar to Java Classes.

For example, **InvoiceProcessor** class describes the class which has all the methods and actions that can be performed on the Invoice. If you create an instance of this class, then it will represent the single invoice which is currently in context.

Creating Classes

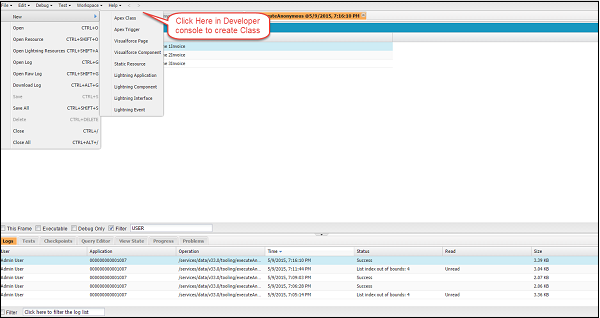
You can create class in Apex from the Developer Console, Force.com Eclipse IDE and from Apex Class detail page as well.

From Developer Console

Follow these steps to create an Apex class from the Developer Console −

**Step 1** − Go to Name and click on the Developer Console.

**Step 2** − Click on File ⇒ New and then click on the Apex class.



From Force.com IDE

Follow these steps to create a class from Force.com IDE −

**Step 1** − Open Force.com Eclipse IDE

**Step 2** − Create a New Project by clicking on File ⇒ New ⇒ Apex Class.

**Step 3** − Provide the Name for the Class and click on OK.

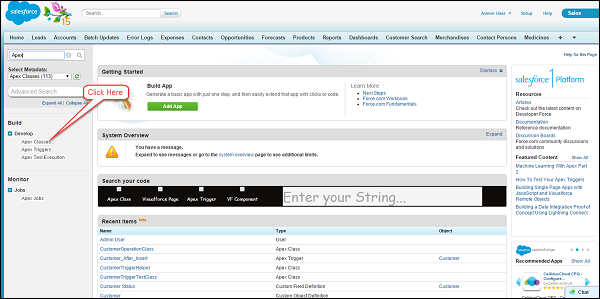
Once this is done, the new class will be created.

From Apex Class Detail Page

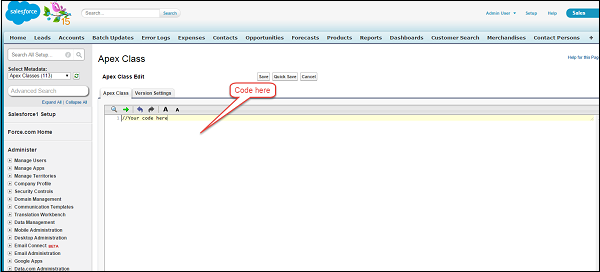
Follow these steps to create a class from Apex Class Detail Page −

**Step 1** − Click on Name ⇒ Setup.

**Step 2** − Search for 'Apex Class' and click on the link. It will open the Apex Class details page.



**Step 3** − Click on 'New' and then provide the Name for class and then click Save.



Apex Class Structure

Below is the sample structure for Apex class definition.

**Syntax**

private | public | global

[virtual | abstract | with sharing | without sharing]

class ClassName [implements InterfaceNameList] [extends ClassName] {

// Classs Body

}

This definition uses a combination of access modifiers, sharing modes, class name and class body. We will look at all these options further.

**Example**

Following is a sample structure for Apex class definition −

public class MySampleApexClass { //Class definition and body

public static Integer myValue = 0; //Class Member variable

public static String myString = ''; //Class Member variable

public static Integer getCalculatedValue () {

// Method definition and body

// do some calculation

myValue = myValue+10;

return myValue;

}

}

Access Modifiers

Private

If you declare the access modifier as 'Private', then this class will be known only locally and you cannot access this class outside of that particular piece. By default, classes have this modifier.

Public

If you declare the class as 'Public' then this implies that this class is accessible to your organization and your defined namespace. Normally, most of the Apex classes are defined with this keyword.

Global

If you declare the class as 'global' then this will be accessible by all apex codes irrespective of your organization. If you have method defined with web service keyword, then you must declare the containing class with global keyword.

Sharing Modes

Let us now discuss the different modes of sharing.

With Sharing

This is a special feature of Apex Classes in Salesforce. When a class is specified with 'With Sharing' keyword then it has following implications: When the class will get executed, it will respect the User's access settings and profile permission. Suppose, User's action has triggered the record update for 30 records, but user has access to only 20 records and 10 records are not accessible. Then, if the class is performing the action to update the records, only 20 records will be updated to which the user has access and rest of 10 records will not be updated. This is also called as the User mode.

Without Sharing

Even if the User does not have access to 10 records out of 30, all the 30 records will be updated as the Class is running in the System mode, i.e., it has been defined with Without Sharing keyword. This is called the System Mode.

Virtual

If you use the 'virtual' keyword, then it indicates that this class can be extended and overrides are allowed. If the methods need to be overridden, then the classes should be declared with the virtual keyword.

Abstract

If you declare the class as 'abstract', then it will only contain the signature of method and not the actual implementation.

Class Variables

**Syntax**

[public | private | protected | global] [final] [static] data\_type

variable\_name [= value]

In the above syntax −

* Variable data type and variable name are mandatory
* Access modifiers and value are optional.

**Example**

public static final Integer myvalue;

Class Methods

There are two modifiers for Class Methods in Apex – Public or Protected. Return type is mandatory for method and if method is not returning anything then you must mention void as the return type. Additionally, Body is also required for method.

**Syntax**

[public | private | protected | global]

[override]

[static]

return\_data\_type method\_name (input parameters) {

// Method body goes here

}

Explanation of Syntax

Those parameters mentioned in the square brackets are optional. However, the following components are essential −

* return\_data\_type
* method\_name

Access Modifiers for Class Methods

Using access modifiers, you can specify access level for the class methods. For Example, Public method will be accessible from anywhere in the class and outside of the Class. Private method will be accessible only within the class. Global will be accessible by all the Apex classes and can be exposed as web service method accessible by other apex classes.

**Example**

//Method definition and body

public static Integer getCalculatedValue () {

//do some calculation

myValue = myValue+10;

return myValue;

}

This method has return type as Integer and takes no parameter.

A Method can have parameters as shown in the following example −

// Method definition and body, this method takes parameter price which will then be used

// in method.

public static Integer getCalculatedValueViaPrice (Decimal price) {

// do some calculation

myValue = myValue+price;

return myValue;

}

Class Constructors

A constructor is a code that is invoked when an object is created from the class blueprint. It has the same name as the class name.

We do not need to define the constructor for every class, as by default a no-argument constructor gets called. Constructors are useful for initialization of variables or when a process is to be done at the time of class initialization. For example, you will like to assign values to certain Integer variables as 0 when the class gets called.

**Example**

// Class definition and body

public class MySampleApexClass2 {

public static Double myValue; // Class Member variable

public static String myString; // Class Member variable

public MySampleApexClass2 () {

myValue = 100; //initialized variable when class is called

}

public static Double getCalculatedValue () { // Method definition and body

// do some calculation

myValue = myValue+10;

return myValue;

}

public static Double getCalculatedValueViaPrice (Decimal price) {

// Method definition and body

// do some calculation

myValue = myValue+price; // Final Price would be 100+100=200.00

return myValue;

}

}

You can call the method of class via constructor as well. This may be useful when programming Apex for visual force controller. When class object is created, then constructor is called as shown below −

// Class and constructor has been instantiated

MySampleApexClass2 objClass = new MySampleApexClass2();

Double FinalPrice = MySampleApexClass2.getCalculatedValueViaPrice(100);

System.debug('FinalPrice: '+FinalPrice);

Overloading Constructors

Constructors can be overloaded, i.e., a class can have more than one constructor defined with different parameters.

**Example**

public class MySampleApexClass3 { // Class definition and body

public static Double myValue; // Class Member variable

public static String myString; // Class Member variable

public MySampleApexClass3 () {

myValue = 100; // initialized variable when class is called

System.debug('myValue variable with no Overaloading'+myValue);

}

public MySampleApexClass3 (Integer newPrice) { // Overloaded constructor

myValue = newPrice; // initialized variable when class is called

System.debug('myValue variable with Overaloading'+myValue);

}

public static Double getCalculatedValue () { // Method definition and body

// do some calculation

myValue = myValue+10;

return myValue;

}

public static Double getCalculatedValueViaPrice (Decimal price) {

// Method definition and body

// do some calculation

myValue = myValue+price;

return myValue;

}

}

You can execute this class as we have executed it in previous example.

// Developer Console Code

MySampleApexClass3 objClass = new MySampleApexClass3();

Double FinalPrice = MySampleApexClass3.getCalculatedValueViaPrice(100);

System.debug('FinalPrice: '+FinalPrice);