Apex

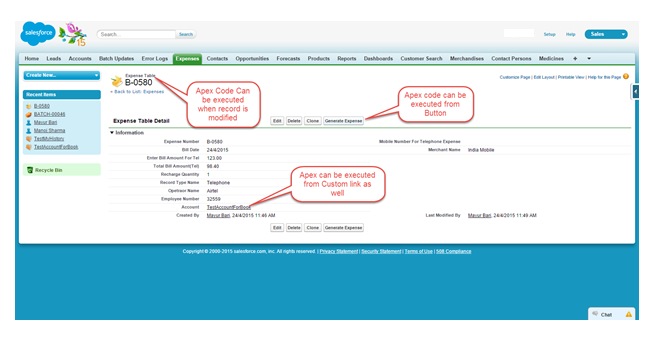
Introduction

Apex is a proprietary language which has been developed by Salesforce.com. Apex is a strongly typed, object-oriented programming language that allows developers to execute flow and transaction control statements on the Force.com platform server in conjunction with calls to the Force.com API.

What is Apex?

Apex is a proprietary language which has been developed by Salesforce.com. As per the official definition, Apex is a strongly typed, object-oriented programming language that allows developers to execute the flow and transaction control statements on the Force.com platform server in conjunction with calls to the Force.com API.

It has a Java-like syntax and acts like database stored procedures. It enables the developers to add business logic to most system events, including button clicks, related record updates, and Visualforce pages.Apex code can be initiated by Web service requests and from triggers on objects. Apex is included in Performance Edition, Unlimited Edition, Enterprise Edition, and Developer Edition.



Features of Apex as a Language

**Integrated**

Apex has built in support for DML operations like INSERT, UPDATE, DELETE and also DML Exception handling. It has support for inline SOQL and SOSL query handling which returns the set of sObject records. We will study the sObject, SOQL, SOSL in detail in future chapters.

**Java like syntax and easy to use**

Apex is easy to use as it uses the syntax like Java. For example, variable declaration, loop syntax and conditional statements.

**Strongly Integrated With Data**

Apex is data focused and designed to execute multiple queries and DML statements together. It issues multiple transaction statements on Database.

**Strongly Typed**

Apex is strongly typed language. It uses direct reference to schema objects like sObject and any invalid reference quickly fails if it is deleted or if is of wrong data type.

**Multitenant Environment**

Apex runs in a multitenant environment. Consequently, the Apex runtime engine is designed to guard closely against runaway code, preventing it from monopolizing shared resources. Any code that violates limits fails with easy-to-understand error messages.

**Upgrades Automatically**

Apex is upgraded as part of Salesforce releases. We don't have to upgrade it manually.

**Easy Testing**

Apex provides built-in support for unit test creation and execution, including test results that indicate how much code is covered, and which parts of your code could be more efficient.

When Should Developer choose Apex?

Apex should be used when we are not able to implement the complex business functionality using the pre-built and existing out of the box functionalities. Below are the cases where we need to use apex over Salesforce configuration.

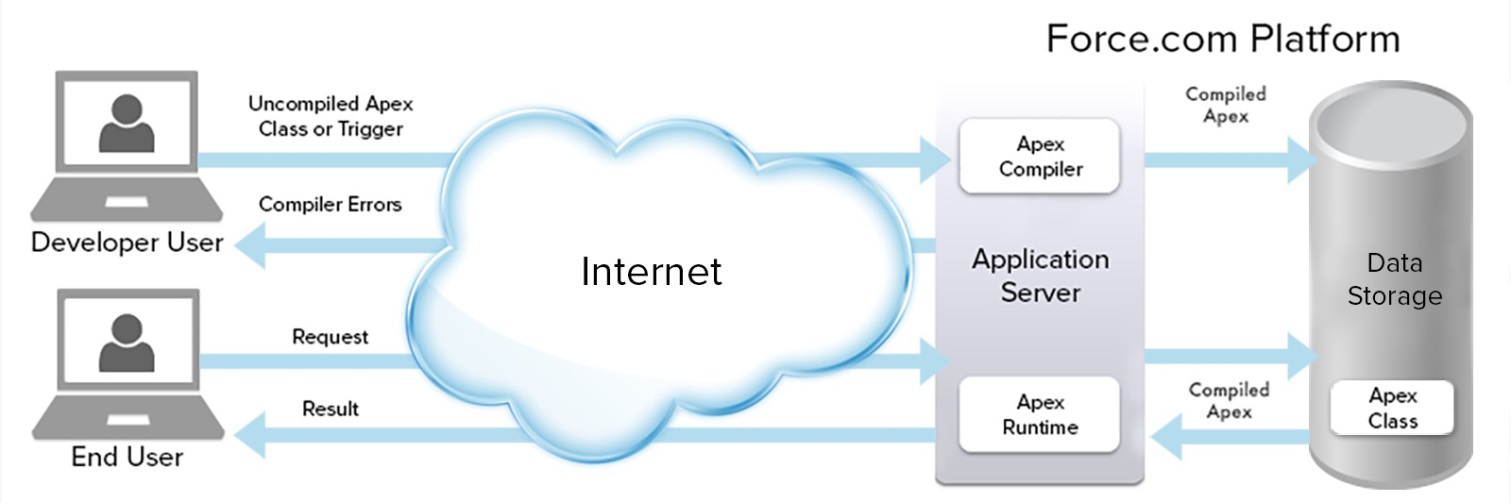
Apex Applications

We can use Apex when we want to:

* Create **Web services (third party application, which can be consumed from any different application)** with integrating other systems.
* Create **email services** for email blast or email setup.
* Perform **complex validation over multiple objects** at the same time and also **custom validation implementation**.
* Create **complex business processes that are not supported** by existing workflow functionality or flows.
* Create **custom transactional logic (logic that occurs over the entire transaction, not just with a single record or object)** like using the Database methods for updating the records.
* Perform some logic when a record is modified or modify the related object's record when there is some event which has caused the trigger to fire.

Working Structure of Apex

As shown in the below diagram (Reference: Salesforce Developer Documentation), Apex runs entirely on demand Force.com Platform:



**Flow of Actions:**

There are two sequence of actions when the developer saves the code and when an end user perform some action which invokes the Apex code as shown below:

* **Developer Action:** When a developer writes and saves Apex code to the platform, the platform application server **first compiles the code into a set of instructions** that can be understood by the **Apex runtime interpreter, and then saves those instructions as metadata (data about data)**.
* **End User Action:** When an end-user triggers the execution of Apex, by clicking a button or accessing a Visualforce page, the platform application server retrieves the compiled instructions from the metadata and sends them through the runtime interpreter before returning the result. The end-user observes no differences in execution time as compared to the standard application platform request.

Since Apex is proprietary language of Salesforce.com, it does not support some features which a general programming language supports. For example, below are some features which Apex does not support:

* It cannot show the elements in User interface.
* You cannot change the standard SFDC provided functionality and also it is not possible to prevent the standard functionality execution.
* Temporary file creation is not supported.
* Creating multiple threads is also not possible as we could do it in other languages.

Understanding the Apex syntax

Apex code typically contains many things that we might be familiar with from other programming languages.

**Variable Declaration:** As strongly typed language, you must declare every variable with data type in Apex. As seen in the code below (screenshot below), lstAcc is declared with data type as List of Accounts.

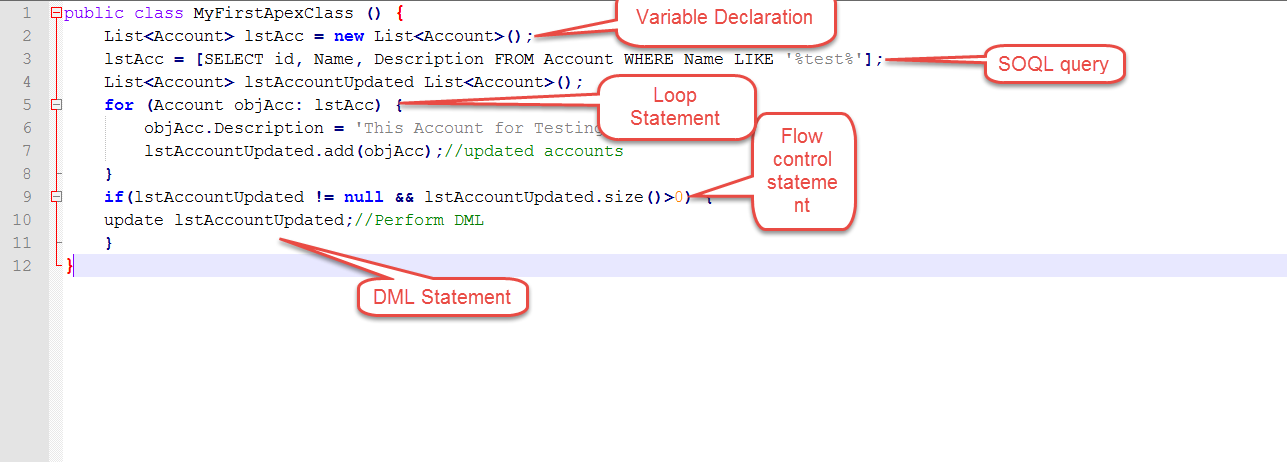
**SOQL Query:** This will be used to fetch the data from Salesforce database. The query shown in screenshot below is fetching data from Account object.

**Loop Statement:** This loop statement is used for iterating over a list or iterating over a piece of code for specified number of times. In this code shown in screenshot below, iteration will be same as the number of records we have in lstAcc.

**Flow Control Statement:** The If statement is used for flow control in this code. Based on certain condition, it is decided whether to go for execution or to stop the execution of the particular piece of code. For example, in the code shown below, it is checking whether the list is empty or it contains records.

**DML Statement:** Performs the records insert, update, upsert, delete operation on the records in database. For example, the below code is updating Accounts with new field value.

Following is a sample of how an Apex code snippet would look like. We are going to study all these Apex programming concepts further in this tutorial.



Understanding the Data Types

As we have studied, the Apex language is strongly typed so every variable in Apex will be declared with the specific data type. All apex variables are initialized to null initially. As a best practice developer has to make sure it should get assigned proper value otherwise such variables when used, will throw null pointer exceptions or any unhandled expections.

Apex supports the following data-types:

* Primitive (Integer, Double, Long, Date, Datetime, String, ID, or Boolean)
* Collections (Lists, Sets and Maps) (To be covered in Chapter 6)
* sObject
* Enums
* Classes, Objects and Interfaces (To be covered in Chapter 11, 12 and 13)

In this chapter, we will look at all the Primitive Data Types, sObjects and Enums. We will be looking at Collections, Classes, Objects and Interfaces in upcoming chapters since they are key topics to be learnt individually.

Primitive Data Type

**Integer**

Any 32 bit number which does not include any decimal point. The value range is -2,147,483,648 and a maximum value of 2,147,483,647.

**Example:** We want to declare a variable which would store the quantity of barrels which needs to be shipped to buyer of chemical processing plant.

Integer barrelNumbers = 1000;

system.debug(' value of barrelNumbers variable: '+barrelNumbers);

System.debug() is function which prints the value of variable so that we could use this to debug or to get to know what value the variable holds currently.

Paste the above code to Developer console and the click on execute. Once logs are generated then it will show the value of variable "barrelNumbers" as 1000.

**Boolean**

This variable can either be true, false or null. Many times, this type of variables can be used as flag in programming to identify the particular condition set or not set.

**Example:** If we would like to set shipmentDispatched as true, then it can be declared as:

Boolean shipmentDispatched;

shipmentDispatched = true;

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

**Date**

This is the variable of type date. This can only store the date not the time. For saving date along with time we would need to store it in variable of DateTime.

**Example:**

//ShipmentDate can be stored when shipment is dispatched.

Date ShipmentDate = date.today();

System.debug('ShipmentDate '+ShipmentDate);

**Long**

This is a 64-bit number without a decimal point. Use this data type when you need a range of values wider than those provided by Integer.

**Example:** If we would like to store the company revenue, then we would use data type as Long.

Long companyRevenue = 21474838973344648L;

system.debug('companyRevenue'+companyRevenue);

**Object**

We can refer this as any data type which is supported in Apex. For example, Class variable can be object of that class, and the sObject generic type is also an object and similarly specific object type like Account is also an Object.

**Example:**

Account objAccount = new Account (Name = 'Test Chemical');

system.debug('Account value'+objAccount);

You can create an object of predefined class as well, as given below:

//Class Name: MyApexClass

MyApexClass classObj = new MyApexClass();

This is the class object which will be used as class variable. No need to execute this code, this is just for reference.

**String**

String is any set of characters within single quotes. It does not have limit of number of characters, but the heap size would be used to determine so that Apex program should not monopolize the resources and does not grow too large.

**Example:**

String companyName = 'Abc International';

System.debug('Value companyName variable'+companyName);

**Time**

This variable is used to store the particular time. This variable should always be declared with system static method.

**Blob**

The Blob is collection of binary data which is stored as object. This will be used when we want to store the attachment in salesforce into a variable. This data type converts the attachments in a single object. When we need to convert the blob into string then we could use toString and valueOf methods to convert it to string when required.

sObject

This is a special data type in Salesforce. It is similar to a table in SQL and contains fields which are similar to columns in SQL. There are two types of sObjects: Standard and Custom.

For example, Account is a standard sObject and any other user defined object (like Customer object that we created) is Custom sObject.

**Example:**

//Declaring an sObject variable of type Account

Account objAccount = new Account();

//Assignment of values to fields of sObjects

objAccount.Name = 'ABC Customer';

objAccount.Description = 'Test Account';

System.debug('objAccount variable value'+objAccount);

//Declaring an sObject for custom object APEX\_Invoice\_c

APEX\_Customer\_c objCustomer = new APEX\_Customer\_c();

//Assigning value to fields

objCustomer.APEX\_Customer\_Decscription\_c = 'Test Customer';

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

Enum

Enum is an abstract data type that stores one value of a finite set of specified identifiers. You could use the keyword Enum to define an Enum. Enum can be used as any other data type in Salesforce.

**Example:**

Suppose, you would like to declare the possible names of Chemical Compound, then you could do something like this:

//Declaring enum for Chemical Compounds

public enum Compounds {HCL, H2SO4, NACL, HG}

Compounds objC = Compounds.HCL;

System.debug('objC value: '+objC);

Apex Variables

Java and Apex are similar in many manners. Variable declaration in Java and Apex is also quite same. Below are some examples to show how to declare local variables.

String productName = 'HCL';

Integer i=0;

Set<string> setOfProducts = new Set<string>();

Map<id, string> mapOfProductIdToName = new Map<id, string>();

Note that all the variables are assigned with the value null.

Declaring Variables

You could declare the variables in Apex like String and Integer as follows:

String strName = 'My String';//String variable declaration

Integer myInteger = 1;//Integer variable declaration

Boolean mtBoolean = true;//Boolean variable declaration

Apex variables are Case-Insensitive

This means that below code will throw an error since the variable 'i' has been declared two times and both will be treated as same.

Integer m = 100;

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

integer m=1; //This statement will throw an error as m is being declared again

System.debug('This code will throw error');

}

Scope of Variables

An Apex variable is valid from the point where it is declared in code. So it is not allowed to redefine the same variable again and in code block. Also, if you declare any variable in a method then that variable scope will be limited to that particular method only. However, class variables can be accessed through out the class.

**Example:**

//Declare variable Products

List<string> Products = new List<strings>();

Products.add('HCL');

//You cannot declare this variable in this code clock or sub code block again

//If you do so then it will throw the error as the previous variable in scope

//Below statement will throw error if declared in same code block

List<string> Products = new List<strings>();

Apex - Strings

String in Apex, as in any other programming language, is any set of characters with no character limit.

**Example:**

String companyName = 'Abc International';

System.debug('Value companyName variable'+companyName);

String Methods

String class in Salesforce has many methods. We will take a look at some of the most important and frequently used string methods in this chapter.

contains

This method will return true if the given string contains the substring mentioned.

**Syntax:**

public Boolean contains(String substring)

**Example:**

String myProductName1 = 'HCL';

String myProductName2 = 'NAHCL';

Boolean result = myProductName2.contains(myProductName1);

System.debug('O/p will be true as it contains the String and Output is: '+result );

equals

This method will return true if the given string and the string passed in the method have the same binary sequence of characters and they are not null. You could compare the SFDC record id as well using this method. This method is case sensitive.

**Syntax:**

public Boolean equals(Object string)

**Example:**

String myString1 = 'MyString';

String myString2 = 'MyString';

Boolean result = myString2.equals(myString1);

System.debug('Value of Result will be true as they are same and Result is:'+result);

equalsIgnoreCase

This method will return true if stringtoCompare has the same sequence of characters as the given string. However, this method is not case sensitive.

**Syntax:**

public Boolean equalsIgnoreCase(String stringtoCompare)

**Example:**

Below code will return true as string characters and sequence are same, ignoring the case sensitivity.

String myString1 = 'MySTRING';

String myString2 = 'MyString';

Boolean result = myString2.equalsIgnoreCase(myString1);

System.debug('Value of Result will be true as they are same and Result is:'+result);

remove

This method removes the string provided in stringToRemove from given string. This is useful when you want to remove some specific characters from string and don't know the exact index of the characters to remove. This method is case sensitive and will not work if the same character sequence occurs but case is different.

**Syntax:**

public String remove(String stringToRemove)

**Example:**

String myString1 = 'This Is MyString Example';

String stringToRemove = 'MyString';

String result = myString1.remove(stringToRemove);

System.debug('Value of Result will be 'This Is Example' as we have removed the MyString and Result is :'+result);

removeEndIgnoreCase

This method removes the string procvided in stringToRemove from the given string but only if it occurs at the end. This method is not case sensitive.

**Syntax:**

public String removeEndIgnoreCase(String stringToRemove)

**Example:**

String myString1 = 'This Is MyString EXAMPLE';

String stringToRemove = 'Example';

String result = myString1.removeEndIgnoreCase(stringToRemove);

System.debug('Value of Result will be 'This Is MyString' as we have removed the 'Example' and Result is :'+result);

startsWith

This method will return true if the given string starts with the prefix provided in the method.

**Syntax:**

public Boolean startsWith(String prefix)

**Example:**

String myString1 = 'This Is MyString EXAMPLE';

String prefix = 'This';

Boolean result = myString1.startsWith(prefix);

System.debug(' This will return true as our String starts with string 'This' and the

Apex - Arrays

Arrays in Apex are basically the same as Lists in Apex. There is no logical distinction between the Arrays and Lists as their internal data structure and methods are also same but the array syntax is little traditional like Java.

Below is the representation of an Array of Products:   
**Index 0** - HCL   
**Index 1** - H2SO4   
**Index 2** - NACL   
**Index 3** - H2O   
**Index 4** - N2   
**Index 5** - U296

**Syntax:**

<String> [] arrayOfProducts = new List<String>();

**Example:**

Suppose, we would like to store name of our Products, then we could use the Array in which we could store the Product Names as shown below. You could access the particular Product by specifying the index.

//Defining array

String [] arrayOfProducts = new List<String>();

//Adding elements in Array

arrayOfProducts.add('HCL');

arrayOfProducts.add('H2SO4');

arrayOfProducts.add('NACL');

arrayOfProducts.add('H2O');

arrayOfProducts.add('N2');

arrayOfProducts.add('U296');

for (Integer i = 0; i<arrayOfProducts.size(); i++) {

//This loop will print all the elements in array

system.debug('Values In Array: '+arrayOfProducts[i]);

}

**Accessing array element by using index:**

You could access any element in array by using the index as shown below:

//Accessing the element in array

//We would access the element at Index 3

System.debug('Value at Index 3 is :'+arrayOfProducts[3]);

Apex - Constants

As in any other programming language, Constants are the variables which do not change their value once declared or assigned a value.

In Apex, Constants are used when we want to define variables which should have constant value throughout the program execution. Apex constants are declared with the keyword 'final'.

**Example:**

Consider a CustomerOperationClass class and a constant variable regularCustomerDiscount inside it:

public class CustomerOperationClass {

static final Double regularCustomerDiscount = 0.1;

static Double finalPrice = 0;

public static Double provideDiscount (Integer price) {

//calculate the discount

finalPrice = price - price\*regularCustomerDiscount;

return finalPrice;

}

}

To see Output of above class, you have to execute the following code in developer console anonymous window:

Double finalPrice = CustomerOperationClass.provideDiscount(100);

System.debug('finalPrice '+finalPrice);

Apex - Decision Making

Decision making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

In this chapter we will be studying the basic and advanced structure of decision making and conditional statements in Apex. Decision making is necessary to control the flow of execution when certain condition is met or not. Following is the general from of a typical decision making structure found in most of the programming languages:



|  |  |
| --- | --- |
| **Statement** | **Description** |
| **if statement** | An if statement consists of a boolean expression followed by one or more statements. |
| **if...else statement** | An if statement can be followed by an optional else statement, which executes when the boolean expression is false. |
| **if...elseif...else statement** | An if statement can be followed by an optional else if...else statement, which is very useful to test various conditions using single if...else if statement. |
| **nested if statement** | You can use one if or else if statement inside another if or else if statement(s). |

Apex - Loops

Loops are used when a particular piece of code should be repeated with desired number of iteration. Apex supports the standard traditional for loop as well as other advanced types of Loops. Lets have a more detailed look on Loops In Apex.

A loop statement allows us to execute a statement or group of statements multiple times and following is the general from of a loop statement in most of the programming languages:



Apex programming language provides following kinds of loop to handle looping requirements. Click the following links to check their detail.

|  |  |
| --- | --- |
| **Loop Type** | **Description** |
| **for loop** | This loop performs a set of statements for each item in a set of records. |
| **SOQL for loop** | Execute a sequence of statements directly over the returned set of SOQL query. |
| **Java-like for loop** | Execute a sequence of statements in traditional Java-like syntax. |
| **while loop** | Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body. |
| **do...while loop** | Like a while statement, except that it tests the condition at the end of the loop body. |

Apex - Collections

Collections are the type of variable which can store multiple number of records. For example, List can store multiple number of Account object's records. Let's have a detailed overview of all collection types.

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, but we can do that. Below is the example for 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 level. This is called as 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 utilize while programming to achieve some functionlity like calculating the size of List, adding an element, etc.

Below are some most frequently used methods.

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

The following example demonstrates 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 could use 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

Set is 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 Product Name with 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

Below are some examples which demostrates the methods which we could use 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 standard definition of Class. Apex Classes are similar to Java Classes.

**Example:**

For example, InvoiceProcessor class describes the class which has all the methods and actions which we could do with the Invoice. If you create an instance of this class then it will represent the single invoice which is currently in context.

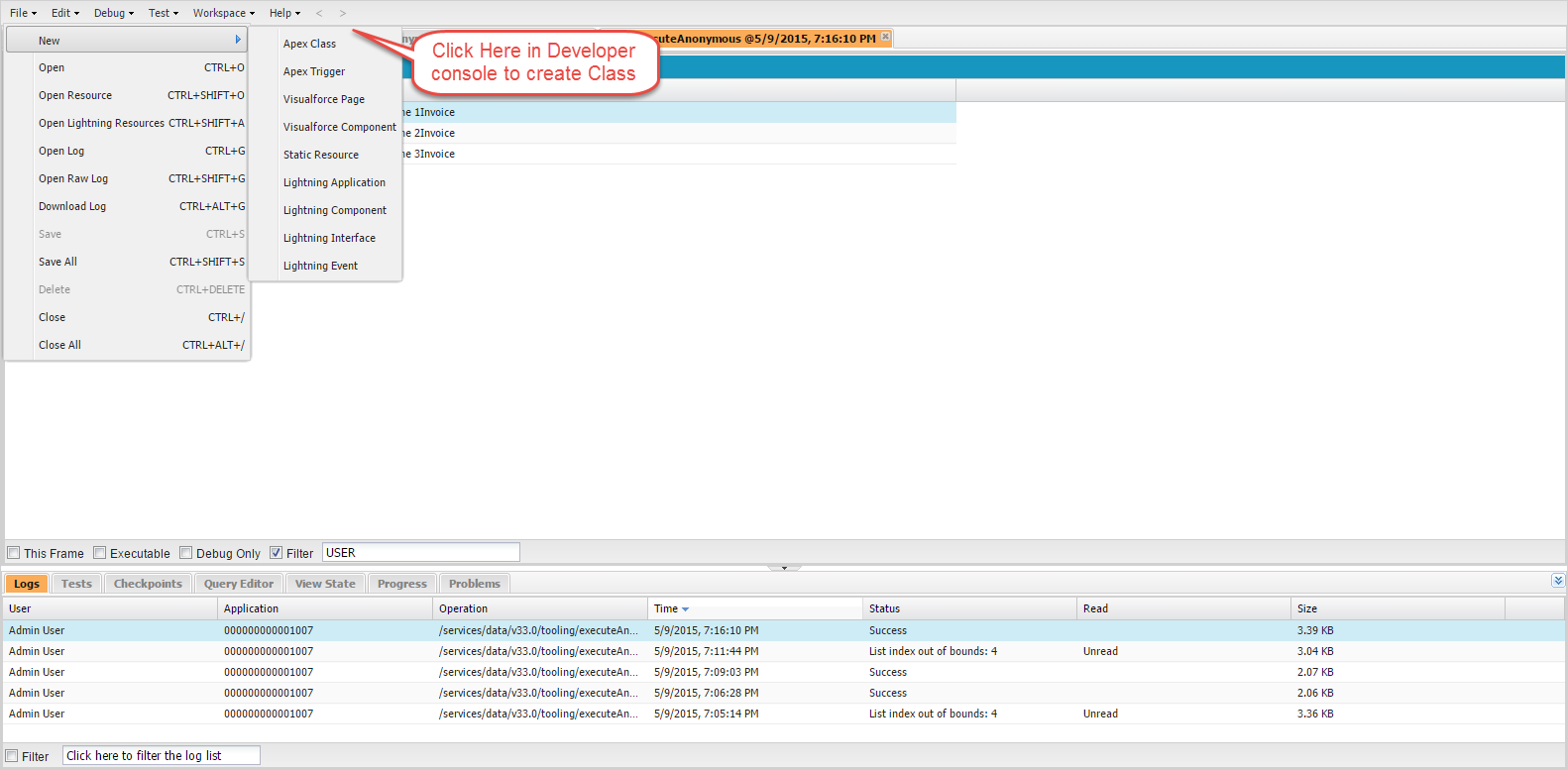
Creating Classes

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

**From Developer Console:**

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

**Step 2:** Click on File => New and then click on Apex 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.

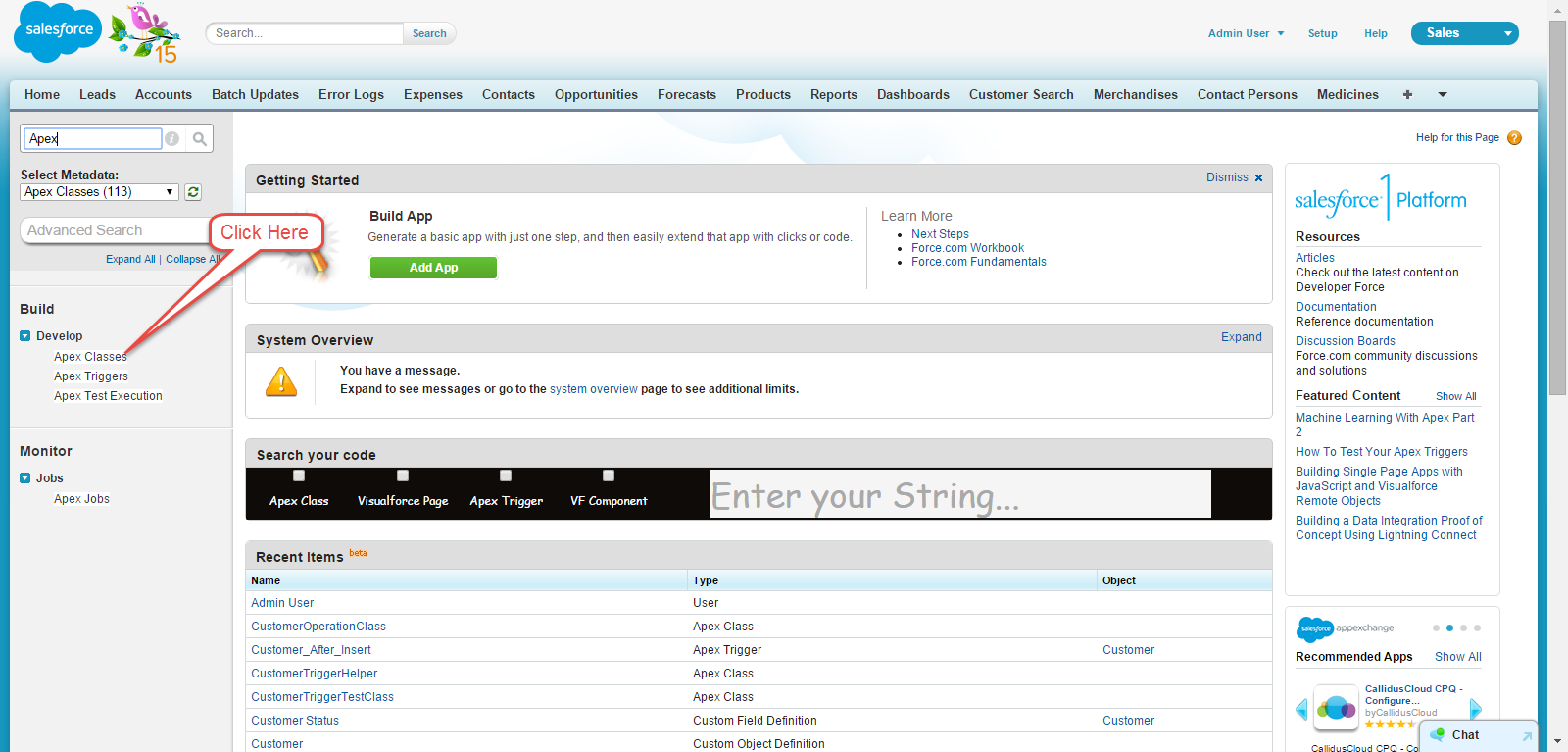
**Step 3:** Provide the Name for the Class and click on OK.

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

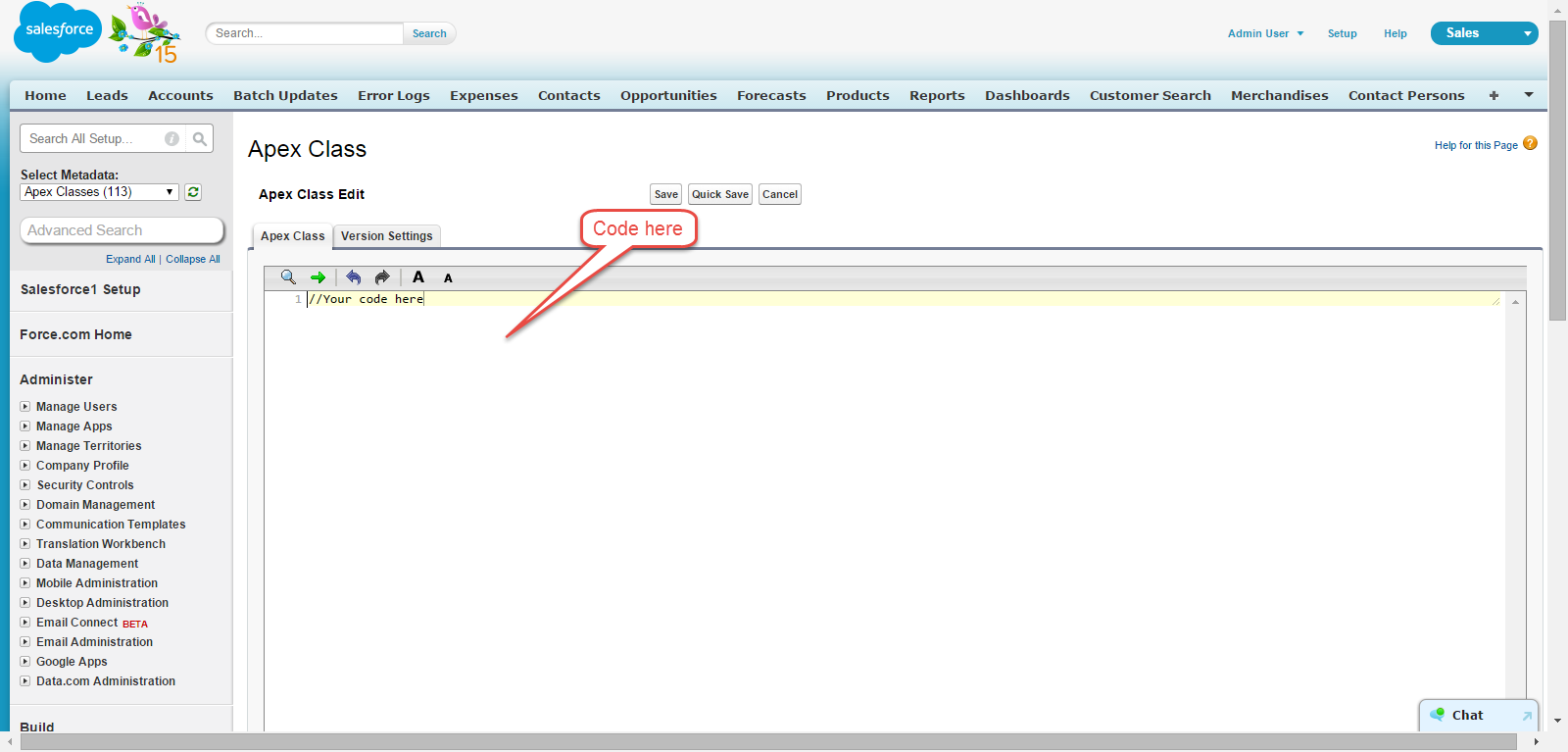
**From Apex Class Detail Page:**

**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:**

Below is the 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 accessible by all apex code irrespective of your organization. If you have method defined with webservice keyword then you must declare the containing class with global keyword.

Sharing Modes

**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 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 System mode, i.e. it has been defined with Without Sharing keyword. This is called System Mode.

**Virtual:**

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

Apex - Classes

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 return type. Body is 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, below components are required:

* return\_data\_type
* method\_name

Access Modifiers for Class Methods:

Using access modifiers, you could specify access level for 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 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 below 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 class name.

We don't need to define the constructor for every class, as by default a no-argument constructor gets called. Constructors are useful when we would like to have some initialization of variables or process to be done at the time of class initialization. For example: You would 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 could 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 constructors 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 could 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);

Apex - Triggers

Apex triggers are like stored procedures which execute when a particular event occurs. It executes before and after an event occurs on record.

Syntax

trigger triggerName on ObjectName (trigger\_events) { Trigger\_code\_block }

Executing the trigger

Below are the events on which we could fire trigger:

* insert
* update
* delete
* merge
* upsert
* undelete

Trigger Example 1

Suppose we received a business requirement that we need to create an Invoice Record when Customer's 'Customer Status' field changes to Active from Inactive. For this, we will create a trigger on APEX\_Customer\_\_c object by following steps:

**Step 1:** Go to sObject

**Step 2:** Click on Customer

**Step 3:** Click on 'New' button in Trigger related list and add the trigger code as give below.

//Trigger Code

trigger Customer\_After\_Insert on APEX\_Customer\_\_c (after update) {

List InvoiceList = new List();

for (APEX\_Customer\_\_c objCustomer: Trigger.new) {

if (objCustomer.APEX\_Customer\_Status\_\_c == 'Active') {

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

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

InvoiceList.add(objInvoice);

}

}

//DML to insert the Invoice List in SFDC

insert InvoiceList;

}

**Explanation:**

**Trigger.new:** This is the context variable which stores the records which are currently in context of trigger, either being inserted or updated. In this case, this variable has Customer object's records which has been updated.

There are other context variables which are available in the context: trigger.old, trigger.newMap, trigger.OldMap.

Trigger Example 2

The above trigger will execute when there is update operation on Customer records. But suppose we wanted to insert invoice record only when Customer Status changes from Inactive to Active and not every time.

For this, we could use another context variable trigger.oldMap which will store the key as record id and value as old record values.

//Modified Trigger Code

trigger Customer\_After\_Insert on APEX\_Customer\_\_c (after update) {

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

for (APEX\_Customer\_\_c objCustomer: Trigger.new) {

//condition to check the old value and new value

if (objCustomer.APEX\_Customer\_Status\_\_c == 'Active' && trigger.oldMap.get(objCustomer.id).APEX\_Customer\_Status\_\_c == 'Inactive') {

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

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

InvoiceList.add(objInvoice);

}

}

//DML to insert the Invoice List in SFDC

insert InvoiceList;

}

**Exaplnation:**

We have used Trigger.oldMap variable which as explained earlier, is a context variable which stores the Id and old value of records which are being updated.

Apex - Trigger Design Patterns

Design patterns are used to make our code more efficient and to avoid hitting the governor limits. Often developers can write inefficient code that can cause repeated instantiation of objects. This can result in inefficient, poorly performing code, and potentially the breaching of governor limits. This most commonly occurs in triggers, as they can operate against a set of records.

We will be looking at some important design pattern strategies in this chapter.

Bulk Triggers Design Patterns

In real business case, it will be possible that you may need to process thousands of records in one go. If your trigger is not designed to handle such situations, then it may fail while processing the records. There are some best practices which you need to follow while implementing the triggers. All triggers are bulk triggers by default, and can process multiple records at a time. You should always plan to process more than one record at a time.

Consider a business case, where you would like to process large number of records and you have written the below trigger. This is the same example which we had taken for inserting the invoice record when Customer Status changes from Inactive to Active.

//Bad Trigger Example

trigger Customer\_After\_Insert on APEX\_Customer\_\_c (after update) {

for (APEX\_Customer\_\_c objCustomer: Trigger.new) {

if (objCustomer.APEX\_Customer\_Status\_\_c == 'Active' && trigger.oldMap.get(objCustomer.id).APEX\_Customer\_Status\_\_c == 'Inactive') {//condition to check the old value and new value

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

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

insert objInvoice;//DML to insert the Invoice List in SFDC

}

}

}

If you have a closer look, then you could see that the DML Statement has been written in for loop block which will work when processing only few records but when you are processing some hundreds of records, it will reach the DML Statement limit per transaction which is governor limit. We will have a detailed look on Governor Limits in later chapter.

To avoid this, we have to make the trigger efficient for processing multiple records at a time.

Here is the best practice example for the same:

//Modified Trigger Code-Bulk Trigger

trigger Customer\_After\_Insert on APEX\_Customer\_\_c (after update) {

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

for (APEX\_Customer\_\_c objCustomer: Trigger.new) {

if (objCustomer.APEX\_Customer\_Status\_\_c == 'Active' && trigger.oldMap.get(objCustomer.id).APEX\_Customer\_Status\_\_c == 'Inactive') {//condition to check the old value and new value

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

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

InvoiceList.add(objInvoice);//Adding records to List

}

}

insert InvoiceList;//DML to insert the Invoice List in SFDC, this list contains the all records which need to be modified and will fire only one DML

}

This trigger will only fire 1 DML statement as it will operating over a List and List has all the records which needs to be modified.

By this way, you could avoid the DML statement governor limits.

Trigger Helper Class

Writing the whole code in trigger is also not a good practice. Hence you should call the Apex class and delegate the processing from Trigger to Apex class as shown below. Trigger Helper class is the class which does all the processing for trigger.

Let's take our invoice record creation example again.

//Below is the Trigger without Helper class

trigger Customer\_After\_Insert on APEX\_Customer\_\_c (after update) {

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

for (APEX\_Customer\_\_c objCustomer: Trigger.new) {

if (objCustomer.APEX\_Customer\_Status\_\_c == 'Active' && trigger.oldMap.get(objCustomer.id).APEX\_Customer\_Status\_\_c == 'Inactive') {//condition to check the old value and new value

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

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

InvoiceList.add(objInvoice);

}

}

insert InvoiceList;//DML to insert the Invoice List in SFDC

}

//Below is the trigger with helper class

//Trigger with Helper Class

trigger Customer\_After\_Insert on APEX\_Customer\_\_c (after update) {

CustomerTriggerHelper.createInvoiceRecords(Trigger.new, trigger.oldMap);//Trigger calls the helper class and does not have any code in Trigger

}

**Helper Class:**

public class CustomerTriggerHelper {

public static void createInvoiceRecords (List<apex\_customer\_\_c> customerList, Map<id, apex\_customer\_\_c> oldMapCustomer) {

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

for (APEX\_Customer\_\_c objCustomer: customerList) {

if (objCustomer.APEX\_Customer\_Status\_\_c == 'Active' && oldMapCustomer.get(objCustomer.id).APEX\_Customer\_Status\_\_c == 'Inactive') {//condition to check the old value and new value

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

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

InvoiceList.add(objInvoice);

}

}

insert InvoiceList;//DML to insert the Invoice List in SFDC

}

}

In this, all the processing has been delegated to the helper class and when we need a new functionality we can simply add the code to helper class without modifying the trigger.

Single Trigger on Each sObject

Always create a single trigger on each object. Multiple triggers on the same object could cause the conflict and errors if it reaches the governor limits.

You could use the context variable to call the different methods from helper class as per the requirement. Consider our previous example. Suppose that our createInvoice method should be called only when the record is updated and on multiple events. Then we could control the execution as below:

//Trigger with Context variable for controlling the calling flow

trigger Customer\_After\_Insert on APEX\_Customer\_\_c (after update, after insert) {

if (trigger.isAfter && trigger.isUpdate) {//This condition will check for trigger events using isAfter and isUpdate context variable

CustomerTriggerHelper.createInvoiceRecords(Trigger.new);//Trigger calls the helper class and does not have any code in Trigger and this will be called only when trigger ids after update

}

}

//Helper Class

public class CustomerTriggerHelper {

//Method To Create Invoice Records

public static void createInvoiceRecords (List<apex\_customer\_\_c> customerList) {

for (APEX\_Customer\_\_c objCustomer: customerList) {

if (objCustomer.APEX\_Customer\_Status\_\_c == 'Active' && trigger.oldMap.get(objCustomer.id).APEX\_Customer\_Status\_\_c == 'Inactive') {//condition to check the old value and new value

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

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

InvoiceList.add(objInvoice);

}

}

insert InvoiceList;//DML to insert the Invoice List in SFDC

}

}