Trigger Handbook

WiseQuarter Education

WiseQuarter

Apex Trigger

Apex triggers **enable you to perform custom actions before or after changes to Salesforce records, such as insertions, updates, or deletions.**

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.

For example, you can have a trigger run before an object's records are inserted into the database, after records have been deleted, or even after a record is restored from the Recycle Bin.

A trigger is Apex code that executes before or after the following types of operations:

Trigger Events:

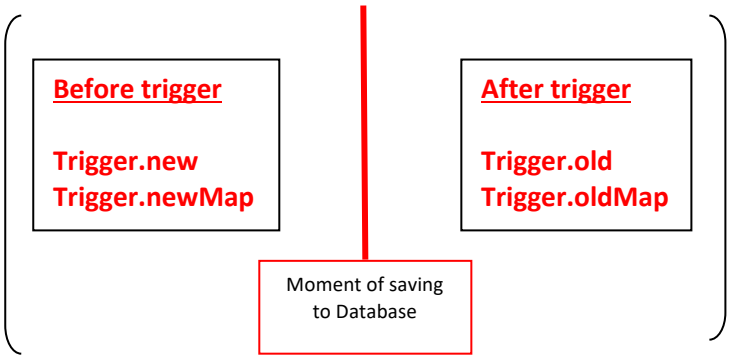
before insert **before** update **before** delete	**after** insert **after** update **after** delete **after** undelete

There are two types of triggers:

- **Before triggers** are used to update or validate record values before they are saved to database.
- **After triggers** are used to access field values that are set by the system (such as a record's 'ID' or 'LastModifiedDate' field), and to affect changes in other records, such as logging into an audit table or firing asynchronous events with a queue. The records that fire the trigger are read-only.

Trigger Syntax:

```
trigger TriggerName on ObjectName (trigger_events) {
                    code_block
}
```



Trigger.new: simply **a list** of the records and holds latest values --> List<sObject>
Trigger.newMap: A map of IDs to the new versions of the sObject records --> Map<Id,sObject>

Trigger.old: holds old copy of current modfying record. --> List<sObject>
Trigger.oldMap : A map of IDs to the old versions of the sObject records. Map<Id,sObject>
You can not make changes old values. (read-only)

	BEFORE				AFTER			
	insert	update	delete	undelete	insert	update	delete	undelete
Trigger.New								
Trigger.newMap								
Trigger.Old								
Trigger.oldMap								

WiseQuarter

TRIGGER CONTEXT VARIABLES

All triggers define implicit variables that allow developers to access run-time context. These variables are contained in the **System.Trigger** class.

	Variable	Usage
Boolean Returning Variables	**Trigger.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.
	Trigger.isInsert	Returns **true** if this trigger was fired due to an insert operation, from the Salesforce user interface, Apex, or the API.
	Trigger.isUpdate	Returns **true** if this trigger was fired due to an update operation, from the Salesforce user interface, Apex, or the API.
	Trigger.isDelete	Returns **true** if this trigger was fired due to a delete operation, from the Salesforce user interface, Apex, or the API.
	Trigger.isBefore	Returns **true** if this trigger was fired before any record was saved.
	Trigger.isAfter	Returns **true** if this trigger was fired after all records were saved.
	Trigger.isUndelete	Returns **true** if this trigger was fired after a record is recovered from the Recycle Bin. This recovery can occur after an undelete operation from the Salesforce user interface, Apex, or the API.
Collection Returning Variables	**Trigger.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.
	Trigger.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.
	Trigger.old	Returns a **list** of the old versions of the sObject records. This sObject list is only available in update and delete triggers.
	Trigger.oldMap	A map of IDs to the old versions of the sObject records. This map is only available in update and delete triggers.
ENUM Returning Variable	**Trigger.operationType**	Returns an enum of type **System.TriggerOperation** corresponding to the current operation. Possible values of the **System.TriggerOperation enum** are: BEFORE_INSERT, BEFORE_UPDATE, BEFORE_DELETE, AFTER_INSERT, AFTER_UPDATE, AFTER_DELETE, and AFTER_UNDELETE. If you vary your programming logic based on different trigger types, consider using the switch statement with different permutations of unique trigger execution enum states.
integer	**Trigger.size**	The total number of records in a trigger invocation, **both old and new. (Returns integer)**

By using
System.debug(Trigger.operationType)

command we can display the operation type on developper console.

https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_triggers_context_variables.htm

WiseQuarter

Bulk Triggers:

All triggers are ***bulk triggers*** by default, and can process multiple records at once. You should always plan on processing more than one record at a time.

Bulkifying Apex code refers to the concept of making sure the code properly handles more than one record at once.

When you use bulk design patterns, your triggers have better performance, consume less server resources, and are less likely to exceed platform limits.

- Write triggers that work on collections of sObjects
- Write triggers that perform efficient SOQL and DML operations

Developer Console

```
trigger MyTriggerNotBulk on Account(before insert) {
    Account acc = new Account();
    acc.Name = 'New Account of Wisequarter';
}
```

This trigger is only for ONE record

```
trigger BulkTrigger on Account(before insert) {

         for(Account acc: Trigger.new) {
        acc.Name = 'New Name';
} }
```

A Bulkified version:
It works when we want to insert one or more records.
It inserts each record added with the for loop in order. but ERROR occurs when Governor limit is exceeded

Adding the record to a list first and inserting the list at once solves all problems. Millions of records can be processed without exceeding the Governor Limit.

```
trigger BulkTrigger on Account(before insert) {
   List<Account> myList = new List<Account>();
       for(Account acc: Trigger.new) {
         acc.Name = 'New Name';
           myList.add(acc);
       }
}
```

Example:

AccountTrigger Class

```
trigger AccountTrigger on Account(before insert) {
      for(Account acc: Trigger.new) {
        acc.Rating ='Hot';
        acc.BillingState ='ARKANSAS';
 System.debug(acc);
 System.debug(Trigger.operationType); //BEFORE_INSERT
 System.debug(Trigger.isBefore);  // true
 System.debug(Trigger.isInsert); // true
 System.debug(Trigger.isAfter); // false
      }
}
```

Anonymous Windows

```
Account acc = new Account();
Acc.name = 'New Name'

Database.insert(acc);
```

system.debug(acc);
command displays name, Rating, BillingState fields.
But since there is no insert yet (i.e. the save button has not been clicked) The ID field returns **null.**

AccountTrigger Class

```
trigger AccountTrigger on Account(before insert) {
    for(Account acc: Trigger.new) {
        acc.Rating ='Hot';
       acc.BillingState ='ARKANSAS';
    }
}
```

Anonymous Windows

```
List<Account> myList = new List<Account>();

for(integer i=0; i<200; i++) {
 Account myAcc = newAccount;
 myAcc.name ='newAcccount ' + i;
     accList.add(myAcc);
}
Database.insert(myList);
```

```
List<Account> myList = [Select id FROM Account WHERE name LIKE : 'newAcc%'];
Database.delete(myList);
// to delete unnecessary records
```

Instead of creating a list in a class in the developer console, we can also send a list via Anonymous windows.

WiseQuarter

AccountTrigger Class

```
trigger AccountTrigger on Account(before insert,after insert) {

      system.debug('how many times this trigger will run ?');
}
   // it runs twice. Because trigger worked as both before
insert and after insert.
```

Anonymous Windows

```
Account acc = new Account();
Acc.name = 'NewName'

Database.insert(acc);
```

*we can use "**Trigger.operationType**" command to display which type of trigger run.

AccountTrigger Class

```
trigger AccountTrigger on Account(before insert,after insert) {

      system.debug(Trigger.operationType + ' is fired ' );
}
   // BEFORE_INSERT is fired
   // AFTER_INSERT  is fired.
```

Anonymous Windows

```
Account acc = new Account();
Acc.name = 'NewName'

Database.insert(acc);
```

Example:

After creating an Account, assign a contact to account like 'Contact of xxx'.

AccountTrigger Class

```
trigger AccountTrigger on Account(after insert) {
    for(Account acc: Trigger.new) {
       contact newCnt = new Contact();
        newCnt.FirstName ='Contact of '+ acc.name;
        newCnt.LastName ='LastName';
           insert newCnt;
    }
}
```

Anonymous Windows

```
Account acc = new Account();
Acc.name = 'NewName'

Database.insert(acc);
```

** In the multiple DML/SOQL operations we will do, this code will exceed governor limit and fail. (DML-150, SOQL-100/200)

***we have to use LIST to avoid governor limits.**

AccountTrigger Class

```
gger on Account(before insert,after insert) {

    if( Trigger.isBefore && Trigger.isInsert  ) {
           for(Account acc: Trigger.new){

             acc.Rating= 'Hot';
             acc.BillingState = 'ARKANSAS';
          }

    }

    if (Trigger.isAfter && Trigger.isInsert ) {

     List<Contact> newContacts = new List<Contact>();
        for(Account acc: trigger.new){

             Contact cnt = new Contact();
             cnt.FirstName= 'Contact of ' + acc.name ;
             cnt.LastName= 'Last name' ;
             cnt.AccountId = acc.Id;
                newContacts.add(cnt);
         }
              insert newContacts;
    }
}
```

Anonymous Windows

```
Account acc = new Account();
Acc.name = 'NewName'

Database.insert(acc);
```

Trigger.isBefore
Trigger.isInsert
Trigger.isAfter

Returns BOOLEAN.

trigger.new
* Because trigger.new holds a List, for Loop iterates each element of this List..

cnt.AccountId = acc.Id;
In this line contact of the account is set

If we create an Account , the Rating and Billing State of this account are created automatically and also the Contact of the Account is created.

WiseQuarter

One Trigger Per Object: (Best Practices)

- It's exactly as it sounds! You combine all possible triggers on a specific object into just one trigger.
- Best practice is creating a single trigger on each object. Multiple triggers on the same object can cause the conflict and errors if it reaches the governor limits.
- Trigger event should contain all events

(before insert, before update, before delete, after insert, after update, after delete, after undelete)

```
trigger TriggerName on ObjectName (trigger_events) {

SWITCH on Trigger.operationType {

WHEN BEFORE_INSERT{ }
WHEN BEFORE_UPDATE{ }
WHEN BEFORE_DELETE{ }
WHEN AFTER_INSERT{ }
WHEN AFTER _UPDATE{ }
WHEN AFTER _DELETE{ }
WHEN AFTER _UNDELETE{ }
}
}
```

```
trigger TriggerName on ObjectName (trigger_events) {

if(Trigger.IsBefore && Trigger.isInsert) { }
if(Trigger.IsBefore && Trigger.isUpdate) { }
if(Trigger.IsBefore && Trigger.isDelete) { }
if(Trigger.IsAfter && Trigger.isInsert) { }
if(Trigger.IsAfter && Trigger. isUpdate) { }
if(Trigger.IsAfter && Trigger. isDelete) { }
if(Trigger. IsAfter && Trigger. isUndelete) {
}

}
```

```
trigger TriggerName on ObjectName (trigger_events)
{
 if(Trigger.IsBefore) {
    if(Trigger.isInsert){ }
    if(Trigger.isUpdate){ }
    if(Trigger.isDelete){ }
  }
 if(Trigger.IsAfter {
    if(Trigger.isInsert){ }
    if(Trigger.isUpdate){ }
    if(Trigger.isDelete){ }
    if(Trigger.isUndelete){ }
  }
}
```

Example:

Before creating an Account, if the industry filed is null assign 'Banking' to Industry field

AND

Before updating an Account, assign 'Education' to Industry field.

AccountTrigger Class

```
trigger AccountTrigger on Account(before insert, before update,){

        List<Account> newAccounts = new List<Account>();
              for(Account acc: Trigger.new){
                  if(acc.industry== null){
                  acc.industry= 'Banking';
                   newAccounts.add(acc);
     System.debug('Trigger.operationType: ' + Trigger.operationType);
     System.debug('Trigger.isBefore: ' + Trigger.isBefore);
     System.debug('Trigger.isInsert: ' + Trigger. isInsert);
                  }
              }

        List<Account> newAccounts2 = new List<Account>();
              for(Account acc: Trigger.new){
                  acc.industry= 'Education';
                   newAccounts2.add(acc);
     System.debug('Trigger.operationType: ' + Trigger.operationType);
     System.debug('Trigger.isBefore: ' + Trigger.isBefore);
     System.debug('Trigger.isUpdate: ' + Trigger. isUpdate);
              }
}
```

Anonymous Windows

```
Account acc = new Account();
Acc.name = 'NewName'

Database.insert(acc);
```

```
update [SELECT Id FROM Account WHERE …];
```

system.debug()
allows us to display the result on the screen

Execution Log-Debug

```
Trigger.operationType: BEFORE_INSERT
Trigger.isBefore: true
Trigger.isInsert: true
Trigger.operationType: BEFORE_INSERT
Trigger.isBefore: true
Trigger.isUpdate: false
```

The code works, new Account is inserted

BUT!!!

While the newly Account created, Industry supposed to be Banking, however it appeares to be Education .

It is necessary to determine which code will run .
Becuse of this we need SWITCH or SWITCH or IF STATEMENT.

WiseQuarter

The Same Example with Switch On Methods:

- Before creating an Account, if the industry filed is null assign 'Banking' to Industry field
- Before updating an Account, assign 'Education' to Industry field.
- After creating an Account, create a contact of Account

AccountTrigger Class

```
trigger AccountTrigger on Account(before insert, before update, before delete, after insert, after update, after delete, after undelete){

    SWITCH on Trigger.operationType {

      WHEN BEFORE_INSERT{

       List<Account> newAccounts = new List<Account>();
            for(Account acc: Trigger.new){
                if(acc.industry== null){
                acc.industry= 'Banking';
                 newAccounts.add(acc);
            }
      }
      WHEN BEFORE_UPDATE{

       List<Account> newAccounts2 = new List<Account>();
            for(Account acc: Trigger.new){
              if(acc.industry== null){
                acc.industry= 'Edcation';
                 newAccounts2.add(acc);
              }
            }
      }
      WHEN BEFORE_DELETE{}
      WHEN AFTER_INSERT{
       List<Contact> yeniContacts = new List<Contact>();
            for(Account acc: trigger.new){

             Contact cnt = new Contact();
             cnt.FirstName= 'Contact of ' + acc.name;
             cnt.LastName= 'Last name';
             cnt.AccountId = acc.Id;
               yeniContacts.add(cnt);
            }
              insert yeniContacts;

      }
      WHEN AFTER_UPDATE{}
      WHEN AFTER_DELETE{}
      WHEN AFTER_UNDELETE{}
    }
}
```

Anonymous Windows

```
Account acc = new Account();
Acc.name = 'NewName'

Database.insert(acc);
```

We can write more readable code using SWITCH ON.

Instead of (acc.industry== null)

We can write **(!String.isBlank(Lead.Industry))**

Instead of writing long code inside the trigger like this, We can write simpler and more organized code by creating Handler classes. BEST PRACTİCE

WiseQuarter

Handler Class (Best Practices):

Apex trigger handler is an apex class to handle complexity in trigger logic set.

It is used to provide a better way of writing complex logic that's required for trigger code and also avoid creating more than one trigger per object.

Use the Trigger handler to contain all the code and create your triggers as logic-less triggers.

(Trigger Apex trigger içerisinde yazılırken, Handler Apex class içerisinde yazılır.)

Example:

1. Before creating an Account, if the industry filed is null assign 'Banking' to Industry field
2. After creating an Account, create a contact of Account
3. Before updating an Account, assign 'Education' to Industry field.
4. Before deleting an account write '… company is deleted' on the console

AccountTrigger Class

```
trigger AccountTrigger on Account(before insert, before update, before delete, after insert, after update, after delete, after undelete){

    SWITCH on Trigger.operationType {

      WHEN BEFORE_INSERT{
AccountTriggerHandler.beforeInsertHandler(Trigger.new);
      }
      WHEN BEFORE_UPDATE{
AccountTriggerHandler.beforeUpdateHandler(Trigger.new);
      }
      WHEN BEFORE_DELETE{
AccountTriggerHandler.beforeDeleteHandler(Trigger.old);
      }
      WHEN AFTER_INSERT{
AccountTriggerHandler.afterInsertHandler(Trigger.new);
      }
      WHEN AFTER_UPDATE{}
      WHEN AFTER_DELETE{}
      WHEN AFTER_UNDELETE{}
  }
}
```

Anonymous Windows

```
Account acc = new Account();
Acc.name = 'NewName'

Database.insert(acc);
```

BY wrting AccountTriggerHandler.beforeInsertHandler(Trigger.new);
We call handler class in this Trigger.
As we know **trigger.new** Holds the list which we are processing.

When an Account is wanted to be inserted, the WHEN BEFORE_INSERT{} trigger starts to work and does the action written in {}.

AccountTriggerHandler.beforeInsertHandler(Trigger.new);

We send the list kept in Trigger.new to the beforeInsertHandler method of the AccountTriggerHandler Class.

List<Account> takes this sent list with accList and processes it in the Method.

Each element which processed in the for loop is put it in the newAccounts list which is created with line List<Account> newAccounts= new List<Account>() ;
Finally, it is inserted into the database with the previously triggered insert command.

WiseQuarter

AccountTriggerHandler Class

```
Public class AccountTriggerHandler{

 Public static void beforeInsertHandler(List<Account> accList){

       List<Account> newAccounts= new List<Account>();

            for(Account acc: accList){

               if(acc.industry== null){
                acc.industry= 'Banking';
                acc.Rating='Hot';
                 newAccounts.add(acc);
               }
            }
 }
  Public static void beforeUpdateHandler(List<Account> accList){

       List<Account> newAccounts2= new List<Account>();

            for(Account acc: accList){
                if(acc.industry== null){
                acc.industry= 'Edcation';
                 newAccounts2.add(acc);
                }
            }
 }
 Public static void afterInsertHandler(List<Account> accList){

       List<Contact> yeniContacts = new List<Contact>();

            for(Account acc: accList){

              Contact cnt = new Contact();
              cnt.FirstName= 'Contact of ' + acc.name ;
              cnt.LastName= 'Last name' ;
              cnt.AccountId = acc.Id;
                yeniContacts.add(cnt);
            }
             insert yeniContacts;
 }
 Public static void afterInsertHandler(List<Account> accList){

      List<Account> newAccounts3= new List<Account>();

            for(Account acc: accList){

     System.debug(acc.name +' company is deleted');
             }
 }
}
```

Inside the AccountTriggerHandler Class we have a method(function) named beforeInsertHandler. This method accept an account list **(List<Account> accList)** as a parameter **.**

forLoop
Cannot accept trigger.new list because trigger.new can only be used inside trigger class
In here accList holds the List which is hold by Trigger.new **in**

Example:

if phone number is being updated,write the old phone number on the OldPhone field.(firstly; we need to create 'Old Phone' field on Account object.)

AccpuntTrigger Class

```
trigger AccountTrigger on Account (before insert, before update, before delete, after
insert, after update, after delete, after undelete){
    SWITCH on Trigger.operationType{
      WHEN BEFORE_INSERT{}
      WHEN BEFORE_UPDATE{
   AccountTriggerHandler.OldPhoneRecord(Trigger.new, Trigger.oldMap);
      }
      WHEN BEFORE_DELETE{}
      WHEN AFTER_INSERT{}
      WHEN AFTER_UPDATE{}
      WHEN AFTER_DELETE{}
      WHEN AFTER_UNDELETE{}
  }
}
```

AccountTriggerHandler Class

```
Public class AccountTriggerHandler{

 Public static void OldPhoneRecord(List<Account> accList, Map<Id,Account> AccOldMap){

   For(Account newAccount: accList){

      Account oldAcc= AccOldMap.get(newAccount.Id);
      newAccount.Old_Phone__c = oldAcc.Phone;
   }
 }
}
```

Trigger.oldMap: A map of IDs to the old versions of the sObject records.

Since we cannot use Trigger.oldMap in Handler class
We used Map<Id,Account> AccOldMap.
This Map holds records of the Account object before the update according to the ID.
Key: ID
Value: Lead object (prior value)

Example:

Whenever phone field updated on Account
then name field should also get updated with name and old phone number on account.

AccountTrigger Class

```
trigger AccountTrigger on Account (before insert, before update, before delete, after
insert, after update, after delete, after undelete){
    SWITCH on Trigger.operationType{
      WHEN BEFORE_INSERT{}
      WHEN BEFORE_UPDATE{
   AccountTriggerHandler.OldPhoneAdd(Trigger.new, Trigger.oldMap);
      }
      WHEN BEFORE_DELETE{}
      WHEN AFTER_INSERT{}
      WHEN AFTER_UPDATE{}
      WHEN AFTER_DELETE{}
      WHEN AFTER_UNDELETE{}
  }
}
```

AccountTriggerHandler Class

```
Public class AccountTriggerHandler{

 Public static void OldPhoneAdd(List<Account> accList, Map<Id,Account> AccOldMap){

   For(Account newAccount: accList){

      Account oldAcc= AccOldMap.get(newAccount.Id);

      If(newAccount.phone != oldAcc.phone){

        newAccount.name = newAccount.name + oldAcc.Phone;
      }
   }
 }
}
```

Example:

When the account's Billing City is updated,
assign the Billing City field of Account to the Mailing City field in the contacts of that account.
(**Since we will make changes on another object after the update, we write After Update..)

AccountTrigger Class

```
trigger AccountTrigger on Account (before insert, before update, before delete, after
insert, after update, after delete, after undelete){
    SWITCH on Trigger.operationType{
      WHEN BEFORE_INSERT{}
      WHEN BEFORE_UPDATE{}
      WHEN BEFORE_DELETE{}
      WHEN AFTER_INSERT{}
      WHEN AFTER_UPDATE{
      AccountTriggerHandler.MailingCityUpdate(Trigger.new, Trigger.oldMap);
      }
      WHEN AFTER_DELETE{}
      WHEN AFTER_UNDELETE{}
  }
}
```

AccountTriggerHandler Class

```
Public class AccountTriggerHandler{

 Public static void MailingCityUpdate(List<Account> accList, Map<Id,Account> AccOldMap){

      Set<Id> accIdset = new Set<Id>();

   for(Account newAccount: accList){
       Account oldAccount= AccOldMap.get(newAccount.Id);

      if(newAccount.BillingCity != oldAccount.BillingCity){
            accIdset.add(newAccount.Id);
      }

    }

List<Contact> contList=[SELECT Id, MailingCity, Account.BillingCity FROM Contact WHERE AccountId IN : accIdset];

    for(Contact cont: contList){
      cont.MailingCity = cont.Account.BillingCity;
    }
     Database.update(contList);
}
```

Set<Id> accIdset = new Set<Id>();

We put the Id of each Account created with forLoop into SET.
Because; The elements in the SET must be unique, means , there must be no more than one of the same element.

Example:

Create a 'Number of contact' field on the Account and
develop a trigger that updates this field on the Account every time a new Contact is created.

ContactTrigger Class

```
trigger ContactTrigger on Contact (before insert, before update, before delete, after
insert, after update, after delete, after undelete){
    SWITCH on Trigger.operationType{
      WHEN BEFORE_INSERT{
      ContactTriggerHandler.NumberOfContact(Trigger.new);
      }
      WHEN BEFORE_UPDATE{}
      WHEN BEFORE_DELETE{}
      WHEN AFTER_INSERT{}
      WHEN AFTER_UPDATE{}
      WHEN AFTER_DELETE{}
      WHEN AFTER_UNDELETE{}
  }
}
```

ContactTriggerHandler Class

```
Public class ContactTriggerHandler{

 Public static void NumberOfContact(List<Contact> cntList){

      Set<Id> accIdset = new Set<Id>();

   for(Contact newContact: cntList){

       if(newContact.AccountId != null){

          accIdset.add(newContact.AccountId);
       }
   }

   if(accIdset.size()>0){

List<Account> accList=[SELECT Id, (SELECT Id FROM Contacts) FROM Account WHERE Id IN : accIdset];

    for(Account acc: accList){
      acc.Number_Of_Contact__c = acc.Contacts.size();
    }
     Database.update(accList);
  }
}
```

It gives the following error when using AccountId instead of ID

FROM Contacts) FROM Account WHERE AccountId IN : accIdset
^

ERROR at Row:1:Column:57

No such column 'AccountId' on entity 'Account'. If you are attempting to use a custom field, be sure to append the '__c' after the custom field name. Please reference your WSDL or the describe call for the appropriate names.

Example:

- If a new lead is created or updated and if 'Industry' field is null, give a warning.
- Prevent deleting record and give a warning if 'Industry' field is null.

LeadTrigger Class

```
trigger LeadTrigger on Lead(before insert, before update,
before delete, after insert, after update, after delete, after
undelete){

    if( Trigger.IsBefore && Trigger.isInsert){
LeadTriggerHandler.IndustryCheck(Trigger.new);
    }
    if( Trigger.IsBefore && Trigger.isUpdate){
LeadTriggerHandler.IndustryCheck(Trigger.new);
    }
    if( Trigger.IsBefore && Trigger.isDelete){
LeadTriggerHandler.IndustryCheck(Trigger.old);
    }
    if( Trigger.IsAfter && Trigger.isInsert){}
    if( Trigger.IsAfter && Trigger.isUpdate){}
    if( Trigger.IsAfter && Trigger.isDelete){}
    if( Trigger.IsAfter && Trigger.isUndelete){}
}
```

Anonymous Windows

```
Lead leadObj=new Account();
leadObj.name = 'NewName'

Database.insert(acc);
```

We used Trigger.old
Because,
We will take action for the old version of the record (saved, existing record)

LeadTriggerHandler Class

```
Public class LeadTriggerHandler{

 Public static void IndustryCheck(List<Lead> leadList){

        List<Lead> leadList= new List<Lead>();

          for(Lead leadObj: leadList){

              if(leadObj.Industry == null){
                leadObj.Industry.addError('Warning!!');
              }
          }
 }
}
```

addError
is uded to create arror

leadOb.Industry.addError('Warning!!');

this code displays the warning text on Industry field
If we don't write the field name it will displays the error on the page.

* With this trigger, we have created a Validation Rule so that the record that does not have an industry can not

Ex:

- When a lead record is Restored from a recycle bin, type 'Restored' in the console.

LeadTrigger Class

```
trigger LeadTrigger on Lead(before insert, before update,
before delete, after insert, after update, after delete, after
undelete){

    if( Trigger.IsBefore && Trigger.isInsert){}
    if( Trigger.IsBefore && Trigger.isUpdate){}
    if( Trigger.IsBefore && Trigger.isDelete){}
    if( Trigger.IsAfter && Trigger.isInsert){}
    if( Trigger.IsAfter && Trigger.isUpdate){}
    if( Trigger.IsAfter && Trigger.isDelete){}
    if( Trigger.IsAfter && Trigger.isUndelete){
       for(Lead leadObj: leadList){
      system.debug('Restored: '+ lead.FirstName);
       }
    }
}
```

When we manually restore a record from the Recycle bin

system.debug('Restored: '+ lead.FirstName);
displays
'Restored: '+ lead.FirstName
on the console

Example:

- Create a field on lead named ‘Recycled’ and update the field as ‘Restored’ when a lead is recycled.

LeadTrigger Class

```
trigger LeadTrigger on Lead(before insert, before update,
before delete, after insert, after update, after delete,
after undelete)
{

    if( Trigger.IsBefore && Trigger.isInsert){}
    if( Trigger.IsBefore && Trigger.isUpdate){}
    if( Trigger.IsBefore && Trigger.isDelete){}
    if( Trigger.IsAfter && Trigger.isInsert){}
    if( Trigger.IsAfter && Trigger.isUpdate){}
    if( Trigger.IsAfter && Trigger.isDelete){}
    if( Trigger.IsAfter && Trigger.isUndelete){
        for(Lead leadObj: leadList){
            leadObj.Recycled__c='Restored';
        }
    }
}
```

leadObj.Recycled__c='Restored';

This code does not work, the system gives an error.
Because after triggers are READ ONLY

Example:

Create an error (Warning) trigger when a Lead is updated from 'Open - Not Contacted' status to 'Closed - Converted' or 'Closed - Not Converted' status. (Validation).

LeadTrigger Class

```
trigger LeadTrigger on Lead(before insert, before update, before delete, after insert,
after update, after delete, after undelete){
    SWITCH on Trigger.operationType{
      WHEN BEFORE_INSERT{}
      WHEN BEFORE_UPDATE{
   LeadTriggerHandler.StatusCheck(Trigger.new, Trigger.oldMap);
      }
      WHEN BEFORE_DELETE{}
      WHEN AFTER_INSERT{}
      WHEN AFTER_UPDATE{}
      WHEN AFTER_DELETE{}
      WHEN AFTER_UNDELETE{}
  }
}
```

LeadTriggerHandler Class

```
Public class LeadTriggerHandler{

 Public static void StatusCheck(List<Lead> leadList, Map<Id,Lead> leadOldMap){

   For(Lead newLead: leadList){

      Lead oldLead= leadOldMap.get(newLead.id);

     If(oldLead.Status=='Open – Not Connected' && (newLead.Status=='Closed – Converted'||
newLead.Status=='Closed – Not Converted')){

        Lead.Status.addError('Warning!!');
     }
   }
 }
}
```

Trigger.oldMap: A map of IDs to the old versions of the sObject records.

Since we cannot use Trigger.oldMap in Handler class
We used Map<Id,Lead> leadOldMap.
This Map keeps records of the lead object according to ID before it is updated.
Key: ID
Value: Lead object (prior value)

Example:

- While creating a lead, assign 'Other' if Lead Source is null.
- After creating Lead record. Create a Task.

LeadTrigger Class

```
trigger LeadTrigger on Lead(before insert, before update, before delete, after insert,
after update, after delete, after undelete){

    SWITCH on Trigger.operationType{
      WHEN BEFORE_INSERT{
            for(Lead newLead: Trigger.new){
              if(String.isBlank(newLead.LeadSource){
                  newLead.LeadSource= 'Other';
              }
            }
      }
      WHEN BEFORE_UPDATE{}
      WHEN BEFORE_DELETE{}
      WHEN AFTER_INSERT{
         List<Task> taskList=new List<Task>();

            for(Lead newLead: Trigger.new){

              Task newTask = new Task();
                newTask.Subject='new Lead Created';
                newTask.Status='Not Started';
                newTask.WhoId=newLead.Id;
                newTask.OwnerId= newLead.OwnerId;

              taskList.add(newTask);
            }
            Database.insert(taskList);
      }
      WHEN AFTER_UPDATE{}
      WHEN AFTER_DELETE{}
      WHEN AFTER_UNDELETE{}
  }
}
```

WiseQuarter

Example:

- Populate Opportunity description as 'Test purpose Opportunity' when user creates Opportunity.
- Whenever Opportunity is updated, show name of user who updates opportunity on description of Opportunity
- Prevent deleting of an Opportunity if Opportunity is 'Closed Won'.
- When the Opportunity stage is updated to ''Negotiation/Review'', develop a trigger that creates a task for the owner.

OpportunityTrigger Class

```
trigger OpportunityTrigger on Opportunity(before insert, before update, before delete, after
insert, after update, after delete, after undelete){

    SWITCH on Trigger.operationType{
      WHEN BEFORE_INSERT{
            for(Opportunity opp: Trigger.new){
               Opp.Description = 'Test purpose Opportunity';
            }
      }
      WHEN BEFORE_UPDATE{
            for(Opportunity opp: Trigger.new){
  opp.Description='This opp is updated by: '+System.userInfo.getFirstName()+' '+ System.userInfo.getLastName();
            }
      }
      WHEN BEFORE_DELETE{
            For(Opportunity opp: Trigger.old){
               If(opp.stageName == 'Closed Won'){
                  Opp.addError('You can not delete Won Deals');
               }
            }
      }
      WHEN AFTER_INSERT{}
      WHEN AFTER_UPDATE{
         List<Task> taskList = new List<Task>();
            for(Opportunity opp: Trigger.new){

If(opp.StageName=='Negotiation/Review' && Trigger.oldMap.get(opp.Id).StageName!= 'Negotiation/Review'){
            Task newTask = new Task();
               newTask.Subject='Opportunity stage is updated to Negotiation/Review';
               newTask.Status='Not Started';
               newTask.OwnerId= opp.OwnerId;
               newTask.WhatId=opp.Id;
            taskList.add(newTask);
}
             }
            Database.insert(taskList);
      }
      WHEN AFTER_DELETE{}
      WHEN AFTER_UNDELETE{}
  }
}
```

where the user's record is kept.

Contains methods for obtaining information about the context user

```
If(opp.StageName=='Negotiation/Review' && Trigger.oldMap.get(opp.Id).StageName!= 'Negotiation/Review'){}

New version  -> (opp.StageName) -> 'Negotiation/Review' ve(&&)
Old version  -> (Trigger.oldMap.get(opp.Id).StageName) -> 'Negotiation/Review'
İf this two are not same ( means stage is updated to Negotiation/Review' )
The statement is true.
```

```
Instead of Trigger.oldMap.get(opp.Id)
We coluld use  Opportunity oldOpp = Trigger.oldMap.get(opp.Id); before
If statement
```

Previous Example With the Handler Class:

OpportunityTrigger Class

```
trigger OpportunityTrigger on Opportunity(before insert, before update, before delete, after insert, after update, after delete, after undelete){

    SWITCH on Trigger.operationType{
       WHEN BEFORE_INSERT{
       OpportunityTriggerHandler.TestDescription(Trigger.new);
       }
       WHEN BEFORE_UPDATE{
       OpportunityTriggerHandler.UpdateDescription(Trigger.new);
       }
       WHEN BEFORE_DELETE{
       OpportunityTriggerHandler.StageCheck(Trigger.old);
       }
       WHEN AFTER_INSERT{}
       WHEN AFTER_UPDATE{
       OpportunityTriggerHandler.StageTask(Trigger.new, Trigger.oldMap);
       }
       WHEN AFTER_DELETE{}
       WHEN AFTER_UNDELETE{}
   }
}
```

OpportunityTriggerHandler Class

```
public class OpportunityTriggerHandler(){
       public static void TestDescription(List<Opportunity> oppList){
          List<Opportunity> newOpps= new List<Opportunity>();
              for(Opportunity opp: oppList){
                 Opp.Description = 'Test purpose Opportunity';
                 newOpps.add(opp);
              }
       }
       public static void UpdateDescription(List<Opportunity> oppList){
          List<Opportunity> newOpps2= new List<Opportunity>();
              for(Opportunity opp: oppList){

opp.Description='This opp is updated by: '+System.userInfo.getFirstName()+' '+ System.userInfo.getLastName();
                 newOpps2.add(opp);
              }
       }
       public static void StageCheck(List<Opportunity> oppList){
          List<Opportunity> newOpps3= new List<Opportunity>();
              For(Opportunity opp: oppList){
                 If(opp.stageName == 'Closed Won'){
                    Opp.addError('You can not delete Won Deals');
                     newOpps3.add(opp);
                 }
              }
       }
       public static void StageTask(List<Opportunity> oppList Map<Id,Opportunity> oldOppMap){
          List<Task> taskList = new List<Task>();
              for(Opportunity opp: oppList){
If(opp.StageName=='Negotiation/Review' && oldOppMap.get(opp.Id).StageName!= 'Negotiation/Review'){
              Task newTask = new Task();
                 newTask.Subject='Opportunity stage is updated to Negotiation/Review';
                 newTask.Status='Not Started';
                 newTask.OwnerId= opp.OwnerId;
                 newTask.WhatId=opp.Id;
             taskList.add(newTask);
}
              }
               Database.insert(taskList);
       }
```

WiseQuarter

Recursion (Best Practices):

- To avoid the recursion on a trigger, make sure your trigger is getting executed only one time.
- Avoid Recursion of a Trigger by calling static boolean variable from another class.

Example:

Generate a trigger that after creating a new lead, creates another lead (FirstName='new Lead Created').

LeadTrigger Class

```
trigger LeadTrigger on Lead(before insert, before update, before delete, after insert, after update, after delete, after undelete)
{
    if( Trigger.IsBefore && Trigger.isInsert){}
    if( Trigger.IsBefore && Trigger.isUpdate){}
    if( Trigger.IsBefore && Trigger.isDelete){}
    if( Trigger.IsAfter && Trigger.isInsert){
   LeadTriggerHandler.newLeadCreation(trigger.old);
    }
    if( Trigger.IsAfter && Trigger.isUpdate){}
    if( Trigger.IsAfter && Trigger.isDelete){}
    if( Trigger.IsAfter && Trigger.isUndelete){}
}
```

Anonymous Windows

```
Lead myLead=new Account();
myLead.FirstName='...';
myLead.LastName='...';
myLead.Status='...';
myLead.Company='...';
myLead.Industry='...';
Database.insert(myLead);
```

**we can create a new lead.

LeadTriggerHandler Class-(**will give ERROR**)

```
Public class LeadTriggerHandler{

 Public static void newLeadCreation(List<Lead> leadList){

       List<Lead> UpdatedleadList= new List<Lead>();

          for(Lead myLead: UpdatedleadList){
              Lead myLead= new Lead();
                myLead.FirstName='new Lead Created';
                myLead.LastName='myLead Last Name';
                myLead.Status='Open – Not Contacted';
                myLead.Company='New Company';
               myLead.Industry='Education';
              Updatedleadlist.add(myLead);
          }
           Database.insert(UpdatedleadList);
   }
}
```

We created a new Lead via the handler inside the AFTER INSERT trigger.
After a Lead is created (insert), another lead will be created which will create the next lead.
This will give us an infinite loop.
This will create **'Maximum trigger depth exceeded'**. error

LeadTriggerHandler Class (Recursion Stop)

```
Public class LeadTriggerHandler{
      Public static Boolean RecursionStop=true;
 Public static void newLeadCreation(List<Lead> leadList){

     List<Lead> UpdatedleadList= new List<Lead>();
      If(RecursionStop){

            RecursionStop = false;
        for(Lead myLead: UpdatedleadList){
            Lead myLead= new Lead();
              myLead.FirstName='new Lead Created';
              myLead.LastName='myLead Last Name';
              myLead.Status='Open – Not Contacted';
              myLead.Company='New Company';
             myLead.Industry='Education';
            Updatedleadlist.add(myLead);
        }
         Database.insert(UpdatedleadList);
      }
   }
```

```
Public static Boolean RecursionStop=true;
```

Boolean RecursionStop variable is assigned as **true**

```
If(RecursionStop){

RecursionStop = false;

}
```

Becasue RecursionStop is true the code in if statement will run only once.

```
RecursionStop = false;
```

Whit this line we convert RecursionStop to false and then **if()** will not run again.
Which means we did not create recursion

**RecursionStop = false;

```
Public static Boolean RecursionStop=true;
```

** It has to be outside of our method.
Because every time the method is called, it must enter the method as true so that it will