FORCE.COM PLATFORM OVERVIEW

Salesforce provides all the three categories of the Cloud Computing services as below.

1. SAAS (Software as a Service):

As part of SAAS, Salesforce provides Salesforce CRM Application. Which is used to automate the Existing and New Customers information in one place.

2. PAAS (Platform as a Service):

As part of PAAS, Salesforce offered a Platform called as "**Force.com**". By using Force.com Platform, we can

1. Customize the existing Salesforce CRM application
2. We can leverage the Salesforce CRM application features, by adding the additional features based on the need.
3. We can built our own custom applications based on the need. (Ex: Banking, Finance, Healthcare, etc.)
4. We can integrate the Salesforce application with any external systems.

Force.com is a Cloud Platform, which provides a collection of services to build the custom applications.

3. IAAS (Infrastructure as a Service):

As part of IAAS, Salesforce will allocate some amount of memory inside the associated region's available server. In which we can deploy the custom applications. Which is called as a "Salesforce.com Organization".

Force.com platform provides a set of programming languages to build the custom business logic and the custom user interfaces based on the need.

We have 2 types of customizations in Salesforce

1. **Declarative Customization:**

   By using this approach, we can build the application features (Model, View, Controller) without using any programming languages. We can implement the features, by using mouse clicks (i.e. By using Configuration).

   Benefits:

   1. It is very easy to implement the features. (i.e. Developer no need to have the Technical skills)
   2. Reduce the Time
   3. Reduce the Budget / Cost.
   4. Non-Technical persons can implement the application features.

2. **Programmatic Customizations:**

Upon building the application features with Declarative options, we have identified a set of limitations.

To overcome the limitations, we have to implement the application features by writing the code with the help of the programming languages offered by the "Force.com" platform.

Force.com platform provides 2 programming Languages.

1. **Apex Programming:**

   Apex is a Cloud based object oriented programming language. By using Apex programming, we can built our own custom business logic (i.e. Controller)

2. **Visualforce Programming**:

   It is a Web-Development framework, to design attractive and dynamic user interfaces based on the need.

MVC Architecture:

Salesforce Purely Supports MVC style of Development.

MVC is popular as it isolates the application logic from the user interface layer and supports separation of concerns.

The Model-View-Controller (MVC) is an architectural pattern that separates an application into three main logical components: the model, view, and the controller.

Each of these components are built to handle specific development aspects of an application.

MVC is one of the most frequently used industry-standard web development framework to create scalable and extensible projects.

Visual Force supports MVC Architecture for application development.

Model: It describes, how the data has been organized in the backend. Model represents the Data Structure with the help of Database Tables.

Ex: Standard Objects, Custom Objects, Fields and Relationships

View: It represents the Presentation / Front End / View part of the application. It describes, how the Data is represented in the User Interface.

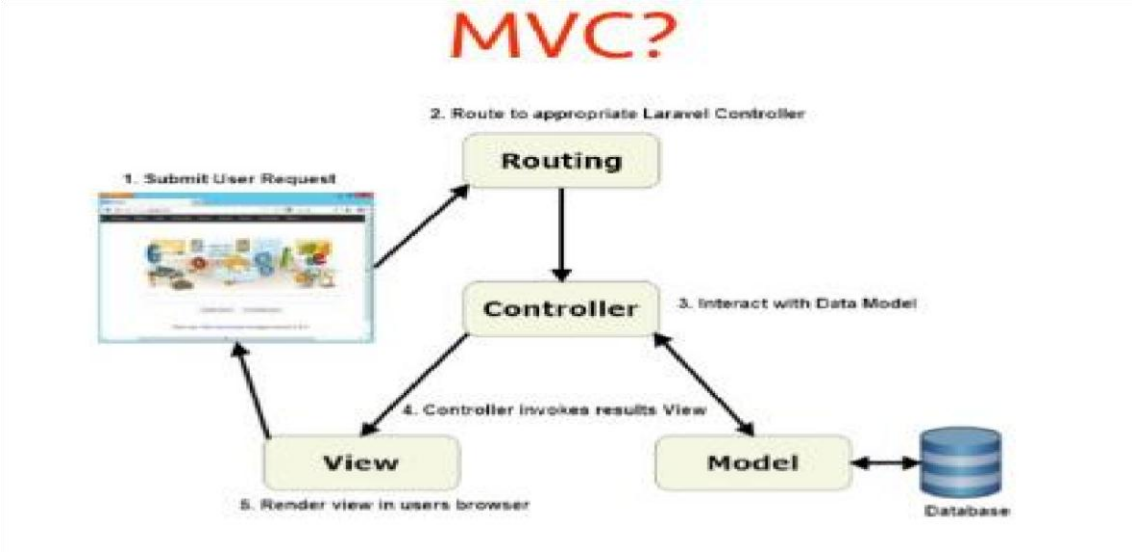
Ex: Page Layouts, Tabs, Search Layouts, Lookup Dialogs --> Standard User Interfaces. VF Pages --> Custom User Interfaces

Controller: Controller is used to perform the actions on the objects and represent the data and results in the user interface. This is also referred as "Business Logic"

Ex: **Standard Business Logic**: Validation Rules, Workflow Rules, Approval processes, sharing

Rules, Record Types, Assignment Rules, Escalation Rules, etc.

Custom Business Logic: Apex Classes, Trigger, SOQL, SOSL, etc.



APEX PROGRAMMING LANGUAGE

Apex is the world's first Cloud based Object Oriented Programming language. Where each business logic feature will be implemented in-terms of Classes and Objects.

By using Apex programming, we can implement our own custom business logic.

Apex is a tightly coupled programming language. (i.e. we need to define the variable first, before its usage in the application.)

Each statement in Apex programming should be ends with a semicolon (;).

By using apex programming language we can implement complex validation rules, and complex transactional flows and Controller Classes.

Apex is tightly integrated with the Salesforce Objects, Which allows us to perform all the DML operations on the Database objects. (i.e. Insert, Update, Upsert, Delete, Undelete)

By using Apex programming, we can retrieve the records from one or more objects by using SOQL Queries. (SOQL - Salesforce Object Query Language)

By using apex programming, we can search for the required content the in the database objects by using SOSL Queries. (SOSL - Salesforce Object Search Language)

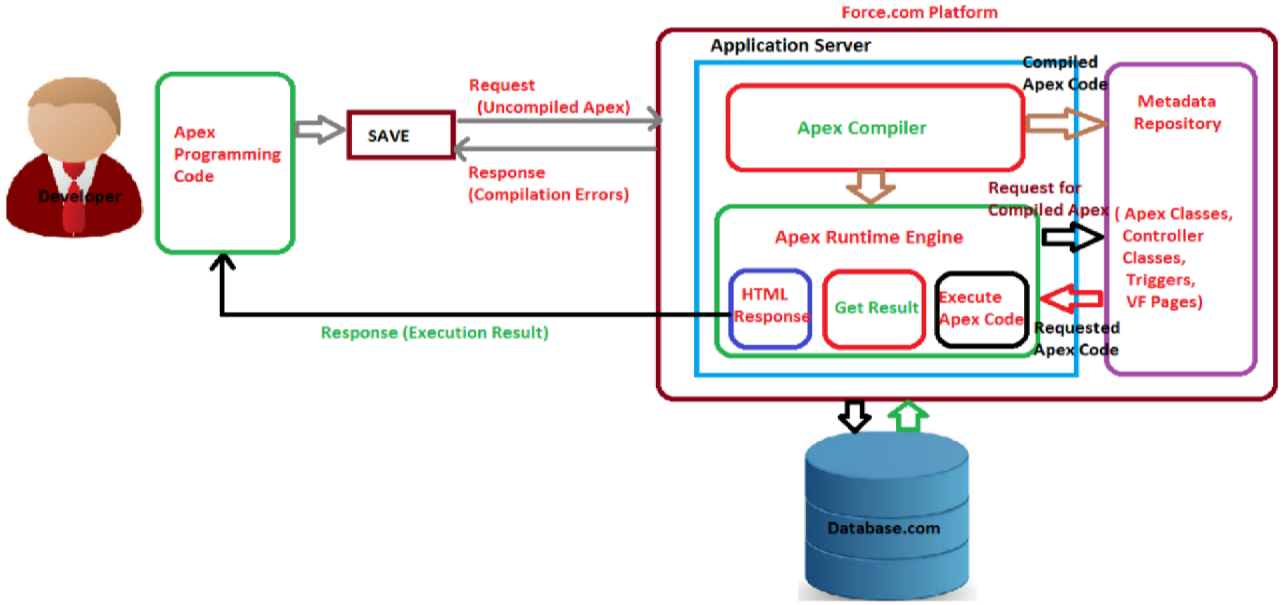
Each statement in apex programming should be compiled by the Apex Compiler, and executed by the "Apex Runtime engine".

Apex is not a case-sensitive programming.

Apex is used to implement the controller classes, which contains the actual business logic.

Apex is comes under programmatic customization.

Apex Programming Life Cycle / Execution Architecture:



Apex Programming provides the fundamental building blocks to prepare the custom business logic as below.

Note: All the Apex Classes will get stored inside the "ApexClass" Object. To Create the Apex Classes, we need "**Author Apex**" permission in the user's profile.

Data Types:

Apex is a Tightly coupled programming language, which requires the variable declaration before it's usage. Upon defining the variable, we have to specify the datatype.

Datatype describes, 2 things as below.

1. What type of data, the variable can hold.
2. How much memory is required to store the value inside the variable.

Apex provides 2 types of Datatypes.

1. **Primitive Types**
2. **SObject Types** (SObject --> Salesforce Objects)

Primitive DataTypes:

These are the fundamental or basic datatypes offered by the Apex Programming language. Which contains the fixed length of memory. Which supports the Static memory allocation. All Primitive datatype values will get resides in "Stack" memory

Note: Primitive datatypes causes the memory wastage.

1. String: This Datatype allows us to store alphanumerical values along with the special characters.

Note: Each Text field, Text Area, Text Area Long, Text Area Rich, Email, Phone, URL, Text Encrypted, Picklist, Picklist-Multiselect, etc. fields will reference the "string" datatype by default.

2. ID : This data type is used to store the 18-characters id value generated by salesforce.

Note: All ID fields, Lookup, Master-Details field types will reference the "ID datatype" by default.

3. Integer: This datatype will store the Numerical values, which doesn't have decimal points inside it.

Note: It will store the values between the Range: -2^31 to +2^31 - 1. Each integer variable will occupy 4 bytes of memory.

Note: Each Number field, Currency field, and percent field will reference the "Integer" Datatype by default.

4. Decimal: This datatype is used to store Decimal values. Which occupies 8-bytes of memory to store each value.

Note: Currency field reference the Decimal datatype by default.

5. Long: This Datatype is used to store the Larger values. Which allows us to store the values between the

Range : -2^63 to +2^63 - 1. Each Long value will occupy 8-bytes of memory.

6. Boolean: This datatype will store the TRUE / FALSE values inside it.

Note: Each Checkbox field will reference the "Boolean" datatype by default.

7. Date: This datatype is used to store "Date" values. Each Date field inside the object will reference the Date Datatype by default.

Note: Each Date value will occupy 8- bytes of memory

8. Time DataType: It allows us store either 12/24 Hours Time value inside the variable**.**

Note: Each Time field type will reference the "Time DataType" by default.
Ex: 10:00 AM, 10:25, 19:45, 5:45:25, 6:10:45:980

9. Dateime: This Datatype is used to store DateTime stamp values. Which should be referenced by "DateTime" field types.

Note: Each DateTime value will occupy 8-bytes of memory.

10. Blob: (Binary Large Objects): This Datatype is used to store Images / Pictures / Audio Files / Videos/ Attachment/ etc.

All the Attachments, Images, pictures will convert into Binary format and the Binary format content will be storing inside the "Blob Datatype". Each Blob Datatype variable can store the value of the size max. 2 GB.

SObject Datatypes:

All salesforce objects (Standard / Custom) are called as "SObjects". We can define a variable with the SObject typs, which can hold the values of each field in the record.

Ex:

```
Account a;
Contact c;
Customer__c cust; Position__c pos;
```

Note:

1. All SObject datatypes will supports Dynamic Memory allocation. (i.e. memory will be allocated dynamically at runtime of the application.)
2. SObject datatype values will get resides in the "Heap Memory".

Variable Declaration:

Variable is nothing but a name to be allocated to a memory address to reference the value inside the memory address.

Upon implementing the business logic, to store the values temporarily, we can define one or more variables inside the application.

Each variable should be defined with the respective datatype.

Syntax: [Access Specifier] <Datatype> <VariableName> [= <Value>];

Note:

1. VariableName should be always starts witha a Character.
2. VariableName name should be a single word always. (It can have one special character "_")
3. We can't define two variables with the same name.

Ex:

integer customerId; string
customerName; string address;
string emailID; Date
joiningDate;

Boolean activeStatus;

Note: Upon defining the variable, if the user didn't assign any value, then apex will fill the memory location with "NULL" value by default. We can assign the value to the variable upon definition as below.

Ex:

integer customerId = 100004;

string customerName = 'Ramesh Kumar'; Boolean offerDiscount = true;

Note: We can define one or more variables in a single statement as below.

Ex:

string customerName, location, industry, rating;

Date joiningDate, birthDate; integer
customerCode;

Decimal annualRevenue, serviceTax; Boolean isCustomerActive;

Operators:

Operator is a symbol, which perform certain operations on the Operands.

Ex:

100 + 20 --> 120

100 * 5 --> 500

Depends on the Operands supplied by the user, the operator behaving differently.

Ex:

10000 + 500 --> 10500

'Welcome' + 'Hyderabd' --> WelcomeHyderabad

'Sampath' + 456 --> Sampath456

Apex provides the below categories of operators.

1. **Mathematical / Arithmetic Operators:** These are used to perform the arithmetic operations on the specified numerical values.

+ --> Addition.

- --> Subtraction

* --> Multiplication

/ --> Division

Math.Mod() --> To get the Reminder value.

2. **Relational / Comparison Operators:** These are used to compare the field values with the specified hardcoded value.

==	--> To check the equality of two numbers.
<	--> Lessthan
<=	--> Lessthan Or Equals
>	--> Greather than
>=	--> Greather or Equals.
!=	--> Not Equals.
Equals() Methods	→ Which uses the "Binary Comparison"

3. **Logical Operators:** These are used to prepare the compound conditions, by clubbing the multiple conditions together.

AND (&&) --> Returns TRUE, if all the conditions are satisfied. Else it returns FALSE.

OR (||) --> It returns TRUE, if any of the condition satisfied. Else it returns FALSE.

NOT (!) --> It will change the sign of the condition result.

4. **Increment / Decrement Operators:** These are used to either increment / decrement the values by 1.

++ --> It will increment the value by 1.

-- --> It will decrement the value by 1.

5. **Assignment Operators:** These are used to assign the values to the variables.

= --> **Assigning the value.**

Ex: integer firstNumber = 100;

+= --> **To increment the value.**

Ex: firstNumber += 230; (i.e. value is : 330)

-= --> To decrement the value with the required number.

Ex: firstNumber -= 20; (i.e. value is : 80)

***=** --> To multiply the value with the required number.

Ex: firstNumber *= 2; (i.e. value is : 200)

/= --> To divide the value with the required number.

Ex: firstNumber /= 2; (i.e. value is : 50)

Printing / Output Statements:

In-order to print the values, salesforce provides a Debug Log file, which will be generated for each program execution.Debug Log file will describes the whole application execution process.

To print the message on the "Debug Log" file, Apex provides a function called as "**System.debug()**".

Syntax: System.debug('Message to Print.');

Ex:

```
System.Debug('Welcome to Apex Programming');

System.debug('This is my first Apex Program.');
```

Ways to Write Apex Programming Code:

Salesforce provides the below 4 Ways to write the Apex Programming code.

1. **By using Execute Anonymous Window:**

   Goto your name and click on it (Ex: Training Batch) and Click on Developer Console" link.

   1. Goto the "Developer Console Editor".
   2. Goto "Debug" menu and Click on it.
   3. Click on "Open Execute Anonymous Window" option.

      Note: From Developer Console Editor, we can use "CTRL+E", as the shortcut.
   4. Write the Code, and Click on "Execute" button.
   5. To execute the Selected Lines of Code, Click on "Execute Highlighted" button.

   Note:

   - Upon click on "Execute" button, it will send the apex code to the "Force.com" platform. It will compile the code and execute the code and the result will be displayed in-terms of a log file called as "Debug Log File".
   - Debug Log file contains the entire execution flow of the apex code along with the results, and error messages if any.
   - Execute Anonymous window, will not save the apex code inside the "Metadata Repository".

Note: It provides the "Intellisence" feature, which exposes the metadata of a specific class.

Note: The Code written inside the Execute Anonymous Window, will not get saved inside the "Metadata Repository".

Note: Select the "Open Log" checkbox inside the "Execute Anonymous Window", which will open the log file automatically upon execution. And Select the checkbox "Debug Only", to see the results inside the window.

2. **By using Standard Navigation:**

By using this approach, we can prepare one or more apex classes and can be stored into Metadata repository.

Click on "Setup" menu.

1. Goto "Build" menu in left panel.
2. Click on "Develop" and expand it.
3. Click on "Apex Classes" link.
4. Click on "New" button, to create a New Apex Class.
5. Write the Code inside the Editor.

Ex:

```
Public Class StudentClass
{
    integer studentId;
    string studdentName, address, emailId;
    boolean isActive;
    decimal feeAmount;
}
```

6. Click on "Save" button.

Note: Upon click on "Save", the code will be send to the "Force.com Platform". It will compile the code and Save the Code in "Metadata Repository".

3. **By using Developer Console:**

By using developer console, we can prepare our own classes, with the required functionalities.

Goto Your Name and expand it.

1. Click on "Developer Console" Editor.
2. Goto "File" menu and expand it.
3. Click on "New" menu item.
4. Click on "Apex Class" from Submenu items.

5. Enter the Apex Class Name inside the Textbox, provided by the Dialog box.

   Ex: StudentClass

6. Click on "Ok" button. Then it will create the class with the specified name.
7. Write the Class Code inside the Editor.
8. Click on "CTRL+S" to save the code.

Note:

Developer Console provides an "AutoSave" option, which will send the apex code to the force.com platform for each 10 fraction of seconds. Once the code gets compiled, it will save the code in "Metadata Repository".

4. By using Eclipse IDE / Visual Studio Code Editor.

Debug Log Files:

Upon Click on "Execute" button, Salesforce generates a "Log" file, which indicates the entire application execution flow. We can see the Log files as below.

By using "Developer Console":

Goto Developer Console → Goto "Tab Bar"

1. Click on "Logs" tab.
2. Double click on "Required Log file". Then It will open the Log file with whole execution flow.

Note: To filter the "Printing statements", Select the "Debug Only" checkbox from the Logfile.

Note: From the Execute Anonymous Window Select "Open Log" checkbox, which will open the log file automatically upon executing the code.

Execute Highlighted: This button is used to execute only the selected lines / statements from the Execute Anonymous Window.

Ways to Execute / Invoke an Apex Class: An apex can be invoked in multiple ways as below.

1. By using Execute Anonymous Window.
2. By using "Triggers".
3. By using Visual Force Pages (i.e. Upon click on Button.)
4. By using Batch Apex classes.
5. By using Schedule Apex.
6. By using Email Services.
7. By using APIs' (REST / SOAP API. i.e. methods will be invoked by the external systems.)
8. By using Eclipse IDE / VS Code Editor.
9. By using Process Builder.

Variables:

Variable is nothing but a name to be get assigned to a memory location, in-order to identify a value.

Variables are used to store the values temporarily, inside the application development.

We can define one or more variables inside the application.

By using variables, we can store the values and we can retrieve the values from the memory location.

Syntax:

[Access Specifier] <DataType> <VariableName> [= <Value>];

Rules:

1. VariableName should be always starts with a "Character".
2. VariableName should be always a single word. (i.e. Name should not have any spaces)
3. We can use "_" (Underscore) special character inside the variable name.
4. We can't define two variables with the same name inside the program.

Ex:

```
integer customerCode;
string customerName;
decimal balanceAmount;
boolean isActive;
date birthDate;
```

Assigning the Values:

Upon defining the variables, apex compiler will assign the "Null" value by default to each variable.

The Developer can assign his own value to the variable based on the need as below.

Syntax:

<variableName> = <value>;

Ex:

```
customerCode = 10001;
customerName = 'Tesla Motors';
balanceAmount = 2000000;
isActive = true;
```

Note: Upon defining the variable, we can assign the value as below.

```
integer productCode = 100001;
string productName = 'Laptop';
boolean isInStock = true;
Decimal unitPrice = 25000;
Date manufacturingDate = system.Today();
Date expriryDate = system.Today().AddDays(90);
                    // Date.newinstance(YYYY, MM,DD);
```

expriryDate = Date.NewInstance(2019,09,24);

Note: We can define one or more similar type of variables in a single statement as below.

Ex:

```
string customerName, address, contactNumber;
Date joiningDate, birthDate, relievingDate;
```

Commenting Statements:

Commenting Statements are used to provide the Documentation / description about the business logic.

Commented lines of code will not be Compiled and will not be executed.

Commented Lines of code will not be counted as part of the Apex Characters limit of 60,00,000.

Apex provides 2 types of Commenting Statements

1. **Single Line Comment.** This is used to comment a single line inside the business logic.

Syntax:

```
// Business Logic.
```

Ex:

```
// System.debug('Welcome');
```

2. **Multi-Line Comment:** This is used to comment a group of statements.

Syntax:

```
/*
    ...
    Apex Code...
    ...
*/
```

/* Write an apex program, to define 2 variables and print the values on the result window. */

```
integer customerCode;
string customerName;

system.debug(customerCode);
system.debug(customerName);

system.debug('Customer Code is...: '+ customerCode);
system.debug('Customer Name is...: '+ customerName);

system.debug('-----------------------------------');

// Assigning the values...
customerCode = 100001;
customerName = 'Tesla Motors Inc.';

system.debug('Customer Code is....: '+ customerCode);
system.debug('Customer Name is....: '+ customerName);
```

/* Write an apex program, to calculate the Simple Interest. */

```
integer principleAmount, tenure, rateOfInterest, interestAmount;

principleAmount = 20000;
tenure = 5;
rateOfInterest = 3;

interestAmount = (principleAmount * tenure * rateOfInterest) / 100;

system.debug('Interest Amount is...: '+ interestAmount);
```

/* Write an apex program, to Calculate the Bill Amount based on the Quantity and UnitPrice. */

```
integer quantity, unitPrice, billAmount;

quantity = 25;
unitPrice = 45;

billAmount = quantity * unitPrice;

system.debug('Bill Amount is...: '+ billAmount);
```

Conditional Statements:

Upon developing the Application, once the source code has been written by the developer, then we need to compile the code by using the Compiler.

Upon compiling the code, it may cause few errors, which are called as "Compilation Errors".

Cause:

1. Due to the Syntactical mistakes in the code.
2. Due to Lack of the System Resources.

Remedy: We need to modify the code according to the programming language syntax, and we need to re-compile the code.

Once the code has been compiled successfully, then upon executing the code, it will cause few errors, which are called as "Exceptions".

Exception is nothing but an undesired behavior or an unexpected behavior of an application at runtime.

Cause:

1. Due to the Invalid input to the application.
2. Due to missing the value for the required fields.
3. Due to Wrong logic inside the application.

Once an exception has been occurred, then it will terminate the application abnormally, and will display the error message to the user in technical terms.

Remedy:

We can handle the Runtime Errors / Exceptions by using the below ways.

1. By using "Conditional Statements".
2. By using "Exception Handling Mechanism".

Conditional statements are used to add one or more user defined conditions, to be get verified before executing the statements. i.e. We can execute the statements based on the result of the specified conditions.

If the conditions are satisfied, then it will execute the statements. Else it will skip the statements from the execution.

By using the Conditional Statements, we can change the control flow of the application execution.

Apex provides the below Conditional Statements

1. **IF Condition**
2. **Switch Statement (From Summer'18 Release)**
3. **Ternary Operator (?:)**

IF Condition:

By using IF Condition, we can add one or more user defined conditions before executing the statements.

Upon preparing the conditions, we have to use "Logical Operators(AND / OR / NOT)".

We have the below formats of "IF Conditional Statement".

1. Simple IF Condition
2. IF Else Condition
3. Else IF Condition
4. Nested IF Condition

Simple IF Condition:

Syntax:

```
If(<Conditions>)
{
    // Statements
    ...
    ...
}
```

Once the conditions get satisfied, it will execute all the statement exist inside the block. Else it will skip the execution.

Upon preparing the conditions, we have to follow the below syntax.

Syntax: <VariableName> <Operator> <Value>

Ex:

```
Rating == 'Hot'
Active__C == 'Yes'
AnnualRevenue > 4000000
isInStock == true
```

We can prepare the compound conditions with the help of "Logical Operators".

```
Industry == 'Banking' && Annualrevenue > 4500000
City == 'Hyderabad' && Status == 'Open - Not Contacted'
```

Ex:

```
integer firstNumber, secondNumber , result;

firstNumber = 100;
secondNumber = 5;

result = firstNumber + secondNumber;
system.debug('Addition Result is...: '+ result);
```

```
result = firstNumber - secondNumber;
system.debug('Subtraction Result is...: '+ result);

result = firstNumber * secondNumber;
system.debug('Multiplication Result is...: '+ result);

if(secondNumber > 0)
 {
        system.debug('Division Result is...: '+ (firstNumber / secondNumber));

        system.debug('Reminder Result is...: '+ Math.Mod(firstNumber, secondNumber));
}
```

IF Else Condition:

Syntax:

```
If(<Conditions>)
{
        // True Block Statements
        ...
        ...
}
else
{
        // False Block Statements
        ...
        ...
}
```

In this approach, if the specified conditions are satisfied, then it will execute the immediate block of statements(TRUE Block). Else it will execute the FALSE Block Statements.

Ex:

```
integer firstNumber, secondNumber , result;

firstNumber = 100;
secondNumber = 20;

result = firstNumber + secondNumber;
system.debug('Addition Result is...: '+ result);

result = firstNumber - secondNumber;
system.debug('Subtraction Result is...: '+ result);

result = firstNumber * secondNumber;
system.debug('Multiplication Result is...: '+ result);

if(secondNumber > 0)
  {
        system.debug('Division Result is...: '+ (firstNumber / secondNumber));
```

```
            system.debug('Reminder Result is...: '+ Math.Mod(firstNumber, secondNumber));
        }
    else
        {
            system.debug('Division and Modulus Operations Cannot be Performed.');
        }
```

Assignments:

1. Write an apex program, to find out the biggest number from the given Two Numbers.

2. Write an apex program, to find out the result of a Student.
    2.1. We have 5 Subjects.
    2.2. PASS : If he got more than 40 marks in each subject.
    2.3. FAIL : If he got less than 40 marks in any of the subject.

Else IF Condition:

Syntax:

```
if(<Conditions>)
{
    // Statements..
    ...
}
else if(<Conditions>)
{
    // Statements..
    ...
}
...
...
else
{
    // False Block Statements..
    ...
}
```

In this approach, it will check the value with the multiple conditional branches. If any of the condition get satisfied, then it will execute the immediate block of statements.

If any of the conditions are not satisfied, then it will execute the else block statements.

Note: At a time, Only One Block of statements will get executed in "Else If" condition.

/* Write an apex program, to find out the biggest number from the given two numbers. */

```
integer firstNumber, secondNumber;

firstNumber = 1000;
```

```
secondNumber = 1000;

if(firstNumber == secondNumber)
  {
    system.debug('Both the Numbers are Equals.');
  }
else if(firstNumber > secondNumber)
  {
    system.debug('First Number is the Biggest One.');
  }
else
  {
    system.debug('Second Number is the Biggest One.');
  }
```

/* Write an apex program, to print the Season Name based on the Month Number as below.

Month Number	Season Name
1 - 4	Winter Season
5 - 8	Summer Season
9 - 12	Spring Seson
<1 OR >12	Invalid Month Number.

***/**

```
integer monthNumber = 11;

if(monthNumber >= 1 && monthNumber <= 4)
  {
    system.debug('Winter Season.');
  }
else if(monthNumber >= 5 && monthNumber <= 8)
  {
    system.debug('Summer Season.');
  }
else if(monthNumber >= 9 && monthNumber <= 12)
  {
    system.debug('Spring Season');
  }
else
  {
    system.debug('Invalid Month Number.');
  }
```

Nested IF Condition:

If the application requirement is very complex, then we can insert / include one if condition into an another, which is also called as "Nested IF Condition".

Syntax:

```
If(<Conditions>)
{
    ... Statements...

    if(<Conditions>)
    {
        // Statements
    }
    else
    {
        // Statements
    }
}
else
{
    // False Block Statement
}
```

Switch Statement:

Switch statement has been introduced by salesforce as part of the "Summer'18" release.

It is also one of the conditional statement, to be used to compare a variable / expression value with the multiple branches.

If any of the Branch value is matching with the expression, then it will execute the corresponding branch code.

If any of the branch value is not matching with the expression, then it will execute the else part / block.

Syntax:

```
Switch ON <Expression>
{
    When <Value1>
    {
        // Statements
    }
    When <Value2>
    {
        // Statements
    }
    ...
    ...
    When Else
    {
        // Default Block Statements.
```

```
        }
    }
```

/*Write an apex program, to print the Season Name based on the Month Number as below.

Month Number	Season Name
1 - 4	Winter Season
5 - 8	Summer Season
9 - 12	Spring Seson
<1 OR >12	Invalid Month Number.

***/**

```
integer monthNumber = 15;

Switch ON monthNumber
{
        When 1,2,3,4
        {
                system.debug('Winter Season.');
        }
        When 5,6,7,8
        {
                system.debug('Summer Season');
        }
        When 9,10,11,12
        {
                system.debug('Spring Season');
        }
        When Else
        {
                system.debug('Invalid Month Number.');
        }
}
```


Iterative Statements:

Iterative statements are used to execute a set of statements repeatedly till the specified number of times.

By using Iterative Statements, we can avoid the Redundant / Duplicate code inside the application. And we can make the code generic.

Apex provides the below Iterative Statements

1. While
2. Do-While
3. For
4. Enhanced For Loop

While Loop:

While is a Pre-Checking iterative statements, which will execute the statements based on the condition result.

If the conditions are satisfied, then only it will execute the statements repeatedly till the condition gets satisfied. Once the condition get failed, then it will comes out of the block.

Syntax:

```
While( <Conditions> )
{
        // Statements..
        ...
        ...
}
```

/* Write an apex program, to print the Welcome Message 10 Times. */

```
integer counter = 1;

While(counter <= 5)
  {
    system.debug('Welcome to Apex Programming.');

    counter++;
  }
```

/* Write an apex program, to print the Numbers between 1 - 100. */

```
integer counter=1;

While(counter <= 100)
  {
    system.debug('Number is...: '+ counter);

    counter++;
  }
```

Do-While Loop:

It is a Post-checking iterative statement, which will execute the statements repeatedly till the specified number of times.

Note: Do-While will execute the statements atleast once.

Syntax:

```
Do
{
        // Statements..
        ...
        ...
}While(<Conditions>);
```

/* Write an apex program, to print the Odd numbers between 1- 100. */

```
integer counter = 1;

Do
  {
    system.debug('Odd Number is..: '+ counter);

    counter += 2;
  }While(counter <= 100);
```

/* Write an apex program, to print the Even Numbers between 1 - 50 in reverse order. */

```
integer counter = 50;

do
  {
    system.debug('Even Number is...: '+ counter);

    counter -= 2;
  }While(counter >= 1);
```

FOR Loop:

It is a Pre-Checking iterative Statement, which provides an optimized way of using iterative statements to execute the block of code repeatedly till the specified number of times.

Syntax:

```
        for(<Initialization Part> ; <Conditional Part> ; <Increment / Decrement Part> )
        {
                // Statements..
                ...
                ...
        }
```

/* Write an apex program, to print the Even Numbers between 1 - 100. */

```
for(integer counter = 2; counter <= 100; counter += 2)
{
   system.debug('Even Number is...: '+ counter);
}
```

Arrays:

Upon storing the values inside a variable, it can hold only one value at a time. When the user assigns a new value to the same variable, then it will override the previous value with the new one.

To avoid this problem, we have to use "Arrays".

Array is nothing but a variable, which can hold a collection of Homogeneous Elements(Same Datatype) inside the variable.

Array doesn't supports to store the Heterogeneous elements.

Syntax:

<DataType>[] <arrayName> = new <DataType>[<Size>];

Note: Upon defining the array, we need to specify the size, which indicates the number of elements to be get hold inside the array variable.

Ex: Integer[] customerCodes = new integer[10];

Note: Upon defining the array, it will allocate the memory for the array variable, based on the "Size" and "DataType".

Arrays are purely supporting "Static Memory Allocation", which will reserve the memory upon definition.

The Whole memory will be divided into the various partitions, Where each partition can be identified by using an "Index" position, starting from "Zero".

By default, each partition will be filled with the value "Null".

Ex:
integer[] customerCodes = new Integer[10]; --> It can hold 10 Integer values.

String[] countryNames = new string[50]; --> It can hold 50 string elements.

Id[] recordIds = new ID[20]; --> It can hold 10 records id's.

Account[] lstAccounts = new Account[30]; --> It can hold 30 Account Records. Each Account record will be treated an an element.

Lead[] leadRecords = new Lead[50]; --> It can hold 50 Lead Records inside the variable.

Opportunity[] oppty = new Opportunity[10]; --> It can hold 10 Opportunity records.

Position__c[] pos = new Position__c[5]; --> It can hold 5 Position records.

Candidate__c[] cands = new Candidate__c[25]; --> It can hold 25 Candidate records.

Assigning the Values:
Once the Array has been defined, then we can assign the value for the array variables as below.

Syntax:

```
<arrayName>[<IndexPosition>] = <value>;
```

Ex:

```
Integer[] codes = new integer[5];

codes[0] = 100;
codes[1] = 456;
codes[2] = 40;
codes[3] = 67;
codes[4] = 10;
```

Retrieving the Values:

Once the values has been assigned to the array, then we can retrieve the values / elements from the array as below.

Syntax:

```
<arrayName>[<IndexPosition>];
```

Ex:

```
system.debug(codes[0]);
system.debug(codes[2]);
system.debug(codes[4]);
```

Example:

```
// Defining the array...
Integer[] customerCodes = new Integer[5];

// Get the Array Size..
system.debug('Array Size is....: '+ customerCodes.size());

// Print the array elements..
system.debug('Array Elements are...: '+ customerCodes);

// Assign the values ...
customerCodes[0] = 100;
```

```
customerCodes[1] = 45;
customerCodes[2] = 567;
customerCodes[3] = 9;
customerCodes[4] = 10;

system.debug('After Assigning, the Array Elements are..: '+ customerCodes);

// Display the elements in separate rows...
system.debug(customerCodes[0]);
system.debug(customerCodes[1]);
system.debug(customerCodes[2]);
system.debug(customerCodes[3]);
system.debug(customerCodes[4]);

// Display the values by using FOR Loop...
for(integer i=0; i< customerCodes.size(); i++)
{
        system.debug('Array Element is...: '+ customerCodes[i]);
}
```

Enhanced FOR Loop:

Upon retrieving the elements from the collection based on the index position, it contains more number of operations internally. Which will degrade the application performance.

To avoid this problem, instead of using the "For" Loop, we can use "Enhanced FOR Loop", which reduces the internal operations. Hence it will improve the application performance.

Enhanced FOR Loop will fetch the elements directly, without using the indexes.

Syntax:

```
for( <Datatype> <varName>  : <Collection Name / Array Name> )
{
        system.debug(varName);
}
```

Note:

1. Enhanced FOR Loop will iterate the specified collection from Starting to ending automatically.(i.e. it is a Forward Only Cursor).

2. We can't iterate the collection in the reverse order.

3. We can't get the specific index position elements through Enhanced FOR Loop.

Example:

```
// Defining the array...
Integer[] customerCodes = new Integer[15];
```

```
// Get the Array Size..
system.debug('Array Size is....: '+ customerCodes.size());

// Assign the values ...
customerCodes[0] = 100;
customerCodes[1] = 45;
customerCodes[2] = 567;
customerCodes[3] = 9;
customerCodes[4] = 10;

// Display the Elements by using Enhanced FOR Loop...
for(integer element : customerCodes)
{
        system.debug('Value is...: '+ element);
}
```

Note: Upon defining the array, we can store the elements inside the array as below.

Syntax:

```
<DataType>[] <arrayName> = new <DataType>[] {<value1>,<Value2>, ......, <ValueN>};
```

Ex:

```
        integer[] codes = new integer[]{100, 45, 234, 23, 56, 789};
```

Example:

```
        string[] countryNames = new string[]{'India','USA','UK','China','Japan','Australia'};

       system.debug('Collection size is...: '+ countryNames.size());

       system.debug('Country Names are...: '+ countryNames);

       for(string country : countryNames)
       {
                system.debug('CountryName is...: '+ country);
       }
```

Drawbacks of Array:

1. Arrays allows to store only homogeneous elements (i.e. Similar Type of elements). We can't store the Heterogeneous elements inside the array.

2. Upon defining the array, we need to specify the size. Which is mandatory.

3. Arrays are purely supporting the "Static Memory Allocation", which causes the memory wastage.

4. Array Elements will get stored inside the "Stack Memory".

5. Array Size cannot be Grow / Shrink at runtime.

6. We can't Insert / Remove an element from the Middle of an array.

7. Arrays will degrade the application performance, upon managing the bulk elements.

8. We don't have any readymade methods to manage the elements inside the array. We have to write the code manually to perform all the operations (Ex: Sort the elements, to insert, to add a new element, etc.)

SObject Datatypes:

Upon storing the values into the primitive type of variables, it reserves the memory during the definition. Which may leads to memory wastage.

To store a collection of similar type of values, we can use "Arrays", which in-terns supports the Static memory allocation.

To store the Heterogeneous elements, which are associated with a Record, salesforce provides "SObject" Datatypes.

SObject --> Salesforce Objects.

For each object, salesforce provides 3 components by default.

Ex: Account

1. Table / Model : Account
2. Tab / View : Accounts
3. Business Logic / Controller: Account

i.e. For each object, salesforce provides a pre-defined class, which has the same name as the object name. Which contains the actual business logic, can be invoked through the User Interfaces (Tabs / Visualforce Pages)

Object Name	Class Name
Account	--> Account
Contact	--> Contact
Opportunity	--> Opportunity
Lead	--> Lead
Case	--> Case
Solution	--> Solution
Campaign	--> Campaign
Position__c	--> Position__c
Candidate__C	--> Candidate__C
Hiring_Manager__C	--> Hiring_Manager__C

Note:

In-order to store an entire record details, we have to create a variable with the respective datatype.

Ex:

Account acc; --> Hold an Account Record
Opportunity oppty; --> Hold an Opportunity Record
Case cs; --> Hold a Case Record
Position__c pos; --> Hold a Position Record
Hiring_Manager__C hr; --> Hold a Hiring Manager Record.

To Store a Collection of Records, we have to define the variable of an Array type as below.

Account[] accRecords = new Account[10]; --> Hold 10 Account Records.

Campaign[] cmpRecords = new Campaign[25]; --> Hold 25 Campaign Records.

Lead[] ldRecords = new Lead[100]; --> Hold 100 Lead Records.

Position__c[] posRecords = new Position__c[5]; --> Hold 5 Position Records

Note:

SObject DataTypes are purely supporting "Dynamic Memory allocation". Hence, we can utilize the memory very efficiently. SObject Datatype Elements, memory will be reserved in "Heap" memory.

OBJECT ORIENTED PROGRAMING

In Procedure Oriented Programming, we have the below drawbacks.

1. It won't provide the Data Security.
2. We don't have the options, to define the level of access of the functionality.

To overcome these drawbacks, we have to use "Object Oriented Programming". Which segregates the entire business logic into the various smaller units called as "Classes".

In Object Oriented Programming, the whole business logic will be divided into the "Classes & Objects".

Class:

A Class is a Blue Print / Physical Entity, which contains a collection members inside it.

(OR)

A Class contains a Set of Data Members (Variables) and Member Functions.

Each class can have one or more variables, functions, procedures, properties, Constructors, etc. All the members defined inside the class are called as "Class Members".

In-order to access the class members, we need a reference / Key called as an "Object".

Whole business logic should be get resides inside the "Class".

Object:

Object is a Logical Entity, which is also called an instance of a class.

Note: Everything in the real-world is an object (Ex: Person, Fan, Marker, Computer, etc.)

By using the Object, we can access each class member and we can perform the "Read / Write" Operations on the class members. (Read / Write --> we can Assign / Retrieve the values)

Object holds the values for the class members. So, the memory will be reserved for the Object.

Note: A class can have one or more Objects. Each object can hold a different value for the class members.

Object Oriented Programming, provides a set of building blocks as below.

1. **Encapsulation**:

Encapsulation is used to group / Wrapping up of a Set of variables and functions together into a single unit called as "Classes".
(i.e. We can group a set of variables, procedures, functions, properties, and Constructors together)

Encapsulation can be achieved with the help of "Classes and Interfaces".

2. Abstraction:

It provides the user interfaces to the user, but without providing the complexity of the Implementation. This is used to achieve the "Data Hiding".

3. **Inheritance**:

Inheritance is used to acquire / access the features of one class into an another.

The Parent class is called as "Base class / Super Class". And the child Class is called as "Derived Class / Sub Class". (i.e. Always a Child class will acquire the features of its parent class. And the child class can have its own additional features.)

4. Polymorphism:

By using this feature, we can define multiple functions with the same name, with Different signature. (i.e. The functions will be differentiated in-terms of their signature (i.e. Number of Parameters, Order of the Parameters and Type of the Parameters).

Apex provides 2 types of Polymorphism

1. **Compile Time Polymorphism:**

This is also called as "Static Binding / Early Binding".
This can be achieved with the help of "Overloading" feature. (Function Overloading / Constructor Overloading)

2. Runtime Polymorphism:

This is also called as "Dynamic Binding / Late Binding".
This can be achieved with the help of "Overriding" feature. (Function Overriding)

Defining Class:

A class is a collection of data members and member functions. We can create one or more classes in an Organization.

Class is used to define the "Business Logic", which can be invoked from the application. By using Class, we can implement the "Custom Controller" inside the Application.

Note: Inside an organization, we can have max. of 60,00,000 apex characters code. (Excluding System.Debug() statements, Commented Lines and Test Classes Code).

We can extend the number of characters limit, by contacting our organization's salesforce Administrator (If we are using Licensed Edition of Salesforce)

All the Apex Classes code will get resides inside the "ApexClass" object. We can query the code as below.

Ex: **Select id, name, body from ApexClass**

Note: Upon defining the Class / Class Members, we have to specify the "Access Specifier".

Syntax:

```
[Access Specifier] Class <ClassName>
{
        // class Members..
        ...
}
```

Access Specifier:

Access Specifier defines, up to what level the class members can be accessible. i.e. It indicates the Level of Access of the Class and class members. Access Specifiers should be applicable for both Class and Class Members.

Apex provides the below 4 Access Specifiers.

1. **Private**:

Private members can be accessible within the class only. We can't access the private members of the class from Outside of the class. This access specifier can be applicable for only class members and Test Classes.

Note: Class should be always defined with either "Public / Global" access specifier.

2. **Public**:

Public members can be accessible from within the class and from out side of the class also.

Note: Public members can be accessible within the entire organization.

3. **Protected**:

Protected members can be accessible within the class and from all the derived Classes (Child classes).

4. **Global**:

Global members can be accessible within the class, within the organization and from out side of the organization also.

Ex: All the Batch Apex Classes, Schedule Apex Classes, API's, Webservices should be defined with "Global" access specifier.

Note: Upon defining the class member, if the user didn't specify any access specifier, then apex will consider the members as "**Private**" by default.

Note:

We can define the Class, by using any of the below ways.

1. By using Standard Navigation
2. By using Developer Console
3. By using "Eclipse IDE / Visual Studio Code Editor".

// Define an apex Class, to manage the StudentDetails.

```
public class StudentDetails
{
        Public integer studentId;
        Public string studentName, address;
        public decimal feeAmount;
        Public Date birthDate;
        Public boolean isActive;
}
```

// Define an apex class To manage the Product Details.

```
public class ProductDetails
{
        Public integer productCode;
        public string productName, manufacturerName;
        Public Date expiryDate, manufacturingDate;
        Public Decimal unitPrice;
        Public boolean isInStock;
}
```

Creating Object:

Object is an instance of the class, which is used to access the class members, and we can assign / retrieve the values of the class members.

Syntax: <ClassName> <objectName> = new <ClassName>();

(OR)

<ClassName> <objectName>;
<objectName> = new <ClassName>();

Ex: StudentDetails s = new StudentDetails();

ClassName : Specify the name of the class for which we need to create an object.

ObjectName: Specify a Name for the object.

1. Object name should be always starts with a character
2. It should be a single word.

New: It will allocate the memory to the object dynamically at runtime.

ClassName(): This statement will call the default constructor of the class. Which take care the responsibility of assigning default values for the class members.

If the user specify the default constructor for the class, then it will get execute. Else the compiler will prepare a default constructor for the class and execute it to assign the values.

Note: Upon creating the object, it will assign the "Null" value for all the class members by default by the default constructor.

We can't create multiple objects with the same name. (i.e. Object name should be always unique.)

A class can have one or more objects. Each object contains its own memory location, which holds a separate value for each class member.

Ex:

```
StudentDetails s = new StudentDetails();
StudentDetails s1 = new StudentDetails();
```

By using the object, we can assign / retrieve the values from the class members.

Assigning the Values to Class Variables:

Syntax: <objectName>.<MemberName> = <value>;

Ex:

```
StudentDetails s = new StudentDetails();

s.studentId = 100001;
s.studentName = 'Ramesh Kumar';
s.isActive = true;
s.feeAmount = 15000;
s.birthDate = Date.NewInstance(2009, 10, 23);
```

Retrieving the values:

We can get the values from the variable as below.

Syntax: <ObjectName>.<MemberName>;

Ex:

```
StudentDetails s = new StudentDetails();

system.debug('Student Id is...: '+ s.studentId);
system.debug('Student Name is...: '+ s.studentName);
system.debug('Fee Amount is...: '+ s.feeAmount);
```

/* Create an Apex class to store the Employee Details, and assign the values to the members by defining an object. */

Class Code: (Developer Console)

```
public class EmployeeDetails
{
    public integer empCode;
    public string empName, designation;
    public decimal empSalary;
```

```
        public boolean isActive;
        public date joiningDate;
}
```

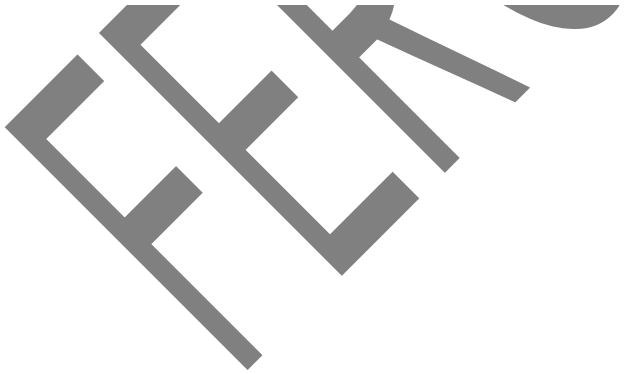
Execution: (Execute Anonymous Window)

```
// Create the object of the class..
        EmployeeDetails emp = new EmployeeDetails();

// Access the Class members...
        system.debug('Employee ID is...: '+ emp.empCode);
        system.debug('Employee Name is...: '+ emp.EmpName);
        system.debug('Designation is....: '+ emp.designation);
        system.debug('Salary is....: '+ emp.empsalary);
        system.debug('Joining Date is....: '+ emp.joiningDate);

// Assigning the values...
        emp.empCode = 2001;
        emp.empName = 'Kiran Kumar';
        emp.empSalary = 45000;
        emp.designation = 'Sales Manager';
        emp.joiningDate = Date.newInstance(2017, 02, 15);
        emp.isActive = true;

// After assginment, Access the Class members...
        system.debug('Employee ID is...: '+ emp.empCode);
        system.debug('Employee Name is...: '+ emp.EmpName);
        system.debug('Designation is....: '+ emp.designation);
        system.debug('Salary is....: '+ emp.empsalary);
        system.debug('Joining Date is....: '+ emp.joiningDate);
        system.debug('Is Employee in Active ? : '+ emp.isActive);
```



ORM Technology:

ORM stands for **Object Relationship and Mapping**

Salesforce has been purely built-upon "ORM Technology", Hence it provides all the 3 Layers of the Application upon creating an object. i.e. Upon creating an object, salesforce provides the below 3 layers.

Layer 1: It creates a Table with the specified name.

Layer 2: It creates a "Tab" with the required user interface controls, to manage the records inside the table.

Layer 3: It provides the Business Logic, in the form of a Class. Where the Class name should be exactly same as the object name.

Ex:

Object Name: Customer

Table Name: Customer__c
Tab Name: Customers
Business Logic (Class): Customer__c

Salesforce provides a Business Logic / Controller class for each object to mange the records in the table with the help of the user interface (Tab) or through apex programming as below.

Object Name	Class Name
Account	--> Account
Contact	--> Contact
Lead	--> Lead
Opportunity	--> Opportunity
Case	--> Case
Order	--> Order
Task	--> Task
Event	--> Event
...	--> ...
Position__c	--> Position__c
Candidate__C	--> Candidate__C
Hiring_Manager__c	--> Hiring_Manager__c
...	

Ex:
Account Object will be represented in terms of an Account Class we below..

```
Public Class Account
{
        // Members...
}
```

Lead Object will be represented in the form of a "Lead" class as below..

```
Public Class Lead
{
        // Members..
}
```

Position Object will be represented in the form of a "Position__c" class as below..

```
Public Class Position__c
{
        // Members...
}
```

Hiring Manager Object will be represented in the form of a class as below..

```
Public Class Hiring_Manager__c
{
        // Members..
}
```


Note: Each Object in salesforce will be represented in-terms of a "Class" by default. Where the Table Name and Class Name should be exactly similar.

Each field inside the Object will be represented in-terms of a Variable inside the Class. Along with the variables, salesforce provides the below 10 standard methods in each class by default.

1. Save()
2. QuickSave()
3. Cancel()
4. Delete()
5. First()
6. Next()
7. Previous()
8. Last()
9. HasPrevious()
10. HasNext()

By using these classes we can interact with the object through programming. And we can perform all the DML operations on the object records.

Ex: Account Object will be represented as below.

```
Public Class Account
{
        // Members...
        Public Id id, ownerid;

        Public string name, rating, industry, type, phone, fax,
                website, BillingCity, BillingState, BillingStreet, BillingCountry, BillingPostalCode,
                 active__C,....etc.

        Public Decimal Annualrevenue;

        Public Boolean Both_Addresses_are_same__c;

        Public DateTime createdby, lastmodifiedby;
        ...

        //Standard Methods

        Public void Save()
        {
                // It will save the record into the associated object and will re-direct the user to the
                detailed page of the record.
        }

        Public void Cancel()
        {
                // It will cancels the current operation..
        }

        Public void Delete()
        {
                // It will remove the current record from the object.
        }

        Public void QuickSave()
        {
                // It will Save the record into the associated object, and the user will be stay in the
                same page.
        }

        //Pagination Methods..
        Public void First()
        {
```

```
                //It will loads the First Set of records in the Pagination
        }

        Public void Next()
        {
                //It will loads the Next Set of records in the Pagination
        }

        Public void Previous()
        {
                //It will loads the Previous Set of records in the Pagination
        }

        Public void Last()
        {
                //It will loads the Last Set of records in the Pagination
        }

        Public Boolean HasPreivous()
        {
                //It returns TRUE, when the Previous Set of records exist in the pagination. Else it
                returns FALSE.
        }

        Public Boolean HasNext()
        {
                //It returns TRUE, when the Next Set of records exist in the pagination. Else it
                returns FALSE.
        }
}
```

Note: Upon creating the records into the objects through Programming, we have to follow the below steps.

Step 1: Create the Object for the associated Class.

Ex: Account acc = new Account();

Step 2: Assign the values for the required fields inside the object.

Ex:

```
acc.Name = 'Sample Test Account';
acc.Rating = 'Hot';
acc.Industry = 'Banking';
acc.Annualrevenue = 4500000;
acc.Active__C = 'Yes';
acc.Both_Addresses_are_same__c = true;
```

Step 3: Insert the Record into the object.

Ex: insert acc;

Step 4: Get the Confirmation of the Insert Operation.

Ex:

```
if(acc.id != null)
{
        system.debug('Account Record Inserted with the ID : '+ acc.id);
}
```

/* Write an apex program, to insert an Account Record into the object. */

```
// Step 1: Create the object of Account Class..
    Account acc = new Account();

// Step 2: Assign the value for the fields..
    acc.Name = 'Sample Apex Account Record';
    acc.rating = 'Hot';
    acc.industry = 'Banking';
    acc.annualrevenue = 5000000;
    acc.type = 'Prospect';
    acc.website = 'www.gmail.com';
    acc.phone = '9900998877';
    acc.fax = '9900887766';
    acc.active__C = 'Yes';
    acc.customerPriority__C = 'High';

// Step 3 : Insert the Record..
    insert acc;

// Step 4: Get the Record Confirmation.
    if(acc.id != null)
    {
        system.debug('Account Record inserted with the Id ...: '+ acc.id);
    }
```

/* Write an apex program, to insert a Hiring Manager Record into the object. */

```
// Step 1: Create the object of Account Class..
    Hiring_Manager__c hr = new Hiring_Manager__C();

// Step 2: Assign the value for the fields..
    hr.Name = 'Srikanth Rathode';
    hr.Location__C = 'Bangalore';
    hr.Contact_Number__C = '9900998877';
    hr.email_id__c = 'srikanth@gmail.com';
```

```
// Step 3 : Insert the Record..
        insert hr;

// Step 4: Get the Record Confirmation.
        if(hr.id != null)
        {
                system.debug('Hiring Manager Record id is...: '+ hr.id);
        }
```

Creating Bulk Records:

By using apex programming, we can insert one or more records into the associated object, by iterating the code till the required number of times by using any of the iterative statement.

Syntax:

```
For(<Initialization Part>; <ConditionsPart>;<Increment/Decrement Part?)
{
        // Write the Code to insert Records.
}
```

/* Write an apex program, to insert 10 Hiring Manager Records into the object. */

```
for(integer counter = 1; counter <= 10; counter ++)
  {
     Hiring_Manager__C hr = new Hiring_Manager__c();

        hr.Name = 'Apex HR - '+ counter;
        hr.location__c = 'Hyderabad';
        hr.contact_number__C = '9900998877';
        hr.email_id__C = 'apexhr'+counter+'@gmail.com';

     insert hr;

     if(hr.id != null)
     {
        system.debug('Hiring Manager Record Created Successfully...: '+ hr.id);
     }
  }
```

Creating Associated Records:

By using Apex programming, we can insert the associated records into the objects.

Step 1: Write the Code to insert the Parent Record.

Step 2: Write the code to insert the Child Record.

2.1: Upon creating the Child record, make the child record to be get associated with the Parent Record. (i.e. We need to supply the Parent Record Id, inside the Relationship field in the Child object)

/* Write an apex program, to insert an Account Record and an associated Contact record.

Account Record
^
|
--- Contact Record
***/**

```
// Step 1: Create the Parent Record.
Account acc = new Account();

acc.Name = 'Parent Account Record';
acc.rating = 'Hot';
acc.industry = 'Banking';
acc.annualrevenue = 5600000;
acc.type = 'Prospect';
acc.phone = '8899889999';
acc.fax = '6655440000';
acc.website = 'www.salesforce.com';
acc.active__C = 'Yes';

insert acc;

if(acc.id != null)
{
        system.debug('Account Record Inserted. Record Id is..: '+ acc.id);

        // Step 2: Create the Associated Contact Record..

        Contact con = new Contact();

                con.firstname = 'Child';
                con.lastname = 'Contact Record';
```

```
            con.email = 'sample@test.com';
            con.title = 'Project Manager';
            con.Phone = '9900990000';
            con.fax = '9955443311';
            con.homephone = '9900087777';
            con.otherphone = '6655665566';

            // make the contact to be associated with the account.
            con.accountid = acc.id;

        insert con;

        if(con.id != null)
        {
            system.debug('Contact Record id is...: '+ con.id);
        }
}
```

/* Write an apex program, to insert a "Hiring Manager Record". And an associated "Position Record" into the object. */

```
// Step 1: Create the Parent Record.
Hiring_Manager__c hr = new Hiring_Manager__c();

hr.name = 'Abhishek Rana';
hr.Email_id__C = 'rana@gmail.com';
hr.contact_Number__C = '9900990000';
hr.location__C = 'Hyderabad';

insert hr;

if(hr.id != null)
  {
    system.debug('Hiring Manager Record ID is..: '+ hr.id);

    // Step 2: Create the Associated Contact Record..
    Position__c pos = new Position__c();

        pos.name = 'SFDC Consultant';
        pos.Location__C = 'Hyderabad';
        pos.Position_status__C = 'New Position';
        pos.Minimum_pay__C = 1000000;
        pos.maximum_budget__C = 1500000;
        pos.number_of_positions__c = 5;
```

```
        pos.open_date__C = system.today();
        pos.milestone_date__C = system.today().AddDays(30);
        pos.hr_email_id__C = 'rana@gmail.com';
        pos.contact_number__C = '9900990000';
        pos.position_description__C = 'Required 4+ years of experience in Salesforce Development.';
        pos.skills_required__C = 'Required 4+ years of experience in Salesforce Development.';

        // make the position to be assigned to HR person..
        pos.HiringManager__c = hr.id;

        insert pos;

        if(pos.id != null)
       {
          system.debug('Position Record ID is...: '+ pos.id);
       }
  }
```

/* Write an apex program, to insert a Hiring Manager Record and an associated Task Record. */

```
// Step 1: Create the Parent Record (Hiring Manager)
Hiring_Manager__c hr= new Hiring_Manager__c();

hr.Name = 'Paramesh Kumar';
hr.Location_del__c = 'Hyderabad';
hr.Email_id__c = 'paramesh@gmail.com';
hr.contact_Number__C = '9900998877';

insert hr;

if(hr.id != null)
  {
     system.debug('HR Record Inserted Successfully. HR Id is..: '+ hr.id);

     // Step 2: Create the Associated Child Record (Task Record)
     Task t = new Task();

        t.Subject = 'Please Update your Contact Details in the Portal.';
        t.Status = 'New';
        t.Priority = 'High';
        t.activityDate = system.today();

        // Make the Task to be associated with the Parent.
```

```
        t.whatid= hr.id;

    insert t;
    if(t.id != null)
    {
       system.debug('Task Record Created Successfully with the Id..: '+ t.id);
    }
}
```

DEFINING SUB PROGRAMS

Upon implementing the business logic, we can divide the whole business logic into the various smaller pieces called as "Sub Programs".

Sub Program is a Part of a Program, which contains the business logic, to be get used to perform a particular task.

Sub Programs should be get resides inside the "Class". We can have one or more sub programs inside a class.

Sub programs has been divided into 2 Types.

1. Procedures:
These are the Fire & Forgot options. Which will perform the operation based on the business logic. And it won't return the result back to the calling environment.

2. Functions.
These are the Request-Reply features, which will perform the operation based on the business logic. And it will return the result to the calling environment.

Procedures:

These are the part of the Program, which contains the business logic to perform certain operations inside the application.

Syntax:

```
[Access Specifier] void <ProcedureName>([<Parameters>])
{
    // Write the Business Logic..
}
```

Note: Always we have to write the procedures inside the class. A class can have one or more procedures.

Salesforce provides 2 types of procedures

1. **Non-Parameterized Procedures.**
These type of procedures, doesn't requires any input values. It will execute the business logic based on the statements.

Syntax:

```
[Access Specifier] void <ProcedureName>()
{
    // Write the Business Logic...
```

```
}
```

Ex:

```
Public Class MathClass
{
    Public void Addition()
    {
        system.debug('Addition Operation.');
    }

    Public void Multiply()
    {
        system.debug('Multiply Method.');
    }

    Public void InsertAccountRecord()
    {
        // Write the Code to insert Account Record..
    }
}
```

Invoke the Procedures:

We can invoke the members of the class, by creating the object of the class.

Syntax:

```
<ObjectName>.<ProcedureName>();
```

Ex:

```
MathClass mClass = new MathClass();

mClass.Addition();
mClass.Multiply();
mClass.InsertAccountRecord();
```

/* Write an apex Class, to insert an Account Record and an associated Case Record into the object. */

Class Code (Developer Console)

```
public class AccountsHelper
{
Public void CreateNewAccount()
  {
    // Insert the Account Record..
    Account acc = new Account();
```

```
        acc.Name = 'Apex Account Record';
        acc.Rating = 'Hot';
        acc.Industry = 'Finance';
        acc.AnnualRevenue = 3500000;
        acc.Type = 'Customer Direct';
        acc.Phone = '9900990000';
        acc.Fax = '9988778888';
        acc.Website = 'www.salesforce.com';
        acc.BillingCity = 'Kochi';
        acc.Active__c = 'Yes';

        insert acc;

        if(acc.id != null)
        {
           system.debug('Account Record has been Inserted Successfully. Record Id is...: '+ acc.id);

           // Create the Associated Case Record..
           Case cs = new Case();

           cs.Subject = 'My Webcam is Not Working';
           cs.Description = 'Dear Customer Support, My Webcam is not working. Please look into my issue
                             ASAP.';
           cs.Priority = 'High';
           cs.Status = 'New';
           cs.Origin = 'Phone';

           cs.AccountId = acc.id;

           insert cs;
           if(cs.id != null)
           {
              system.debug('Case Record has been created successfully. Case id is...: '+ cs.id);
           }
        }
     }
}
```

Execution Process (Execute Anonymous Window):

```
// Creating the object of the class and invoke the method..
        AccountsHelper helper = new AccountsHelper();
        helper.CreateNewAccount();
```

Parameterized Procedures:

These are used to supply the input values to the procedures at runtime. Based on the input values supplied at runtime, it will behave differently.

So that we can make the procedure functionality dynamic.

Syntax:

```
<Access Specifier> void <ProcedureName>([Parameters List])
{
        // Business Logic..
}
```

Note: We can supply "N" number of parameters to the procedure. Each parameter should be supplied with the "DataType and Parameter Name". We can supply the multiple parameters by seperating a comma.

Syntax:

```
<Access Specifier> void <ProcedureName>(<DataType> <ParamName1>,
                                        <DataType> <ParamName2>, .....,
                                        <DataType> <ParamNameN>)
{
        // Business Logic..
}
```

Ex:

```
Public Class MathOperations
{
        Public void Addition(integer value1, integer value2)
        {
            system.debug('Addition REsult is...: '+ (value1 + value2));
        }

        Public void Multiply(integer value1, integer value2, integer value3)
        {
            system.debug('Multiply Result is...: '+ (value1 * value2 * value3));
        }

        Public void Division(integer value1, integer value2)
        {
            if(value2 > 0)
            {
                system.debug('Division Result is...: '+ (value1 / value2));
            }
```

```
            else
            {
                system.debug('Division Operation Cannot be performed');
            }
        }
}
```

Invoke the Procedures:

Upon calling the procedure by using the object, we need to supply the input values to the procedure at runtime. Based on the input values, it will produce the results.

Syntax:

```
<ObjectName>.ProcedureName(<inputvalue1>,<inputvalue2>,....,<inputvalueN>);
```

Ex:

```
MathOperations mOp = new MathOperations();

mOp.Addition(2000, 34); --> O/p: 2034
mOp.Addition(45000, 5000); --> O/p: 50000
```

/* Write an apex class to insert the Hiring Manager Records based on the specified number. */

Class Code:

```
public class HiringManagerHelper
{
  // Note: Static methods can be called directly without the object of the class.
  Public static void CreateRecruiterRecords(integer maxRecords)
  {
    if(maxRecords > 0)
    {
      for(integer counter = 1; counter <= maxRecords; counter++)
      {
        Hiring_Manager__c hr = new Hiring_Manager__C();

            hr.Name = 'Sample HR'+ counter;
            hr.Location__c = 'Chennai';
            hr.Email_ID__c = 'samplehr'+counter+'@gmail.com';
            hr.Contact_Number__c = '9900990000';

        insert hr;

        if(hr.Id != null)
        {
```

```
            system.debug('Record Inserted Successfully. Record id is..: '+ hr.id);
          }
        }
      }
    }
}
```

Execution:

```
        // Call the method...
        HiringManagerHelper.CreateRecruiterRecords(50);
```

/* Write an apex program, to Create an Account record along with the associated case record by preparing the two separate procedures. */

Class Code:

```
public class AccountsHelper
{
   Public void CreateAccountRecord()
   {
      Account acc = new Account();

         acc.name = 'Parent Account Record';
         acc.Rating = 'Hot';
         acc.Industry = 'Banking';
         acc.AnnualRevenue = 4500000;
         acc.Type = 'Prospect';
         acc.Ownership = 'Public';
         acc.Phone = '9999999999';
         acc.Fax = '9988889988';
         acc.Website = 'www.gmail.com';
         acc.Active__c = 'Yes';
         acc.CustomerPriority__c = 'High';

      insert acc;

      if(acc.id != null)
      {
         system.debug('Account Record Created. Record Id is...: '+ acc.id);

         // Call the Procedure to create the case record..
         CreateRelatedCase(acc.id);
      }
   }
```

```
   Public void CreateRelatedCase(Id accId)
   {
      Case cs = new Case();

           cs.Status = 'New';
           cs.Priority = 'High';
           cs.Origin = 'Phone';
           cs.Type = 'Mechanical';
           cs.Reason = 'Performance';
           cs.Subject = 'My Printer is not working properly.';
           cs.Description = 'Dear Customer Support, I am unable to print the documents properly. Please
look into my issue. ';

           cs.AccountId = accId;
      insert cs;

      if(cs.Id != null)
      {
         system.debug('Case Record Created Successfully with the id..: '+ cs.id);
      }
   }
}
```

Execution:

```
//Create the Object of the Class..
        AccountsHelper accHelper = new AccountsHelper();

// Calling the Procedure...
        accHelper.CreateAccountRecord();
```

Assignments:

1. Write an apex class, to Insert 20 Lead Records into the Lead Object.

Ex:

```
Public Class LeadHelper
{
        Public void CreateBulkLead()
        {
                // Write the Business Logic...
        }
}
```

2. Write an apex class, to insert 50 Contact records into the object.

Ex:

```
Public Class ContactsHelper
{
        Public void CreateContacts()
        {
                // Write the Business Logic..
        }
}
```

3. Write an apex class, to insert a Hiring Manager Record. And an associated Position Record.

Ex:

```
Public Class HiringManagerHelper
{
        Public void CreateAssociatedRecords()
        {
                // Write the Business Logic..
        }
}
```

4. Write an apex class, to insert the Lead Records, based on the specified number.

5. Write an apex class, to insert the Contact Records based on the specified number.

6. Write an apex program, to insert 5 Hiring Manager Records.

7. Write an apex program, to insert an Account Record, and Two Associated Contacts, and 5 Associated Opportunity Records.

```
Account Record
        |
        | --> Contacts (2)
        |
        --> Opportunity (5)
```

8. Write an apex program, to insert a Hiring Manager Record and an Associated "Position Record", And Two associated Candidates for the position.

```
Hiring Manager
        |
        --> Position (1)
                        |
                        --> Candidates (2)
```

DEFINING CONSTRUCTORS

Constructors are the Special Member functions which will get executes upon creating the object of the class.

Note: Constructors are used to assign the pre-defined values to the class members.

Constructors will be executed automatically upon creating the object of the class by using "New" keyword.

Upon creating the Constructor, we have to follow the below rules.

1. Constructor Name should be same as the class name
2. Constructor should be always defined with the Access Specifier "Public".
3. Constructor doesn't returns any value to the calling environment. Hence Constructor doesn't have any return type even void.
4. We can define one or more constructors inside a class. (i.e. We can overload the Constructors)
5. We can make the Constructor Chaining. (i.e. One Constructor can invoke the another constructor)

Syntax:

```
Public <ConstructorName>
{
    // Write the Business Logic.
}
```

Ex:

```
Public Class ProductHelper
{
    Public ProductHelper()
    {
        // Write the Business Logic..
    }
}
```

Note: Upon creating the object of the class, If the class doesn't have any constructor, then, Apex Compiler will creates a Default Constructor for the class and will get executes.

Types of Constructors:

We have 2 Types of constructors in Apex Programming.

1. **Non-Parameterized Constructors:**

   This type of Constructor doesn't expect any input values. This is also called as the Default Constructor.

   Syntax:

```
Public Class ProductHelper
{
    Public ProductHelper()
    {
        // Write the Business Logic..
    }
```

```
}
```

2. **Parameterized Constructors:**

It will collect the input values from the user. We need to supply the input values to the constructor upon creating the object of the class.

Syntax:

```
Public Class ProductHelper
{
    Public ProductHelper(integer prodCode, string prodName)
    {
        // Write the Business Logic..
    }
}
```

UseCase: Create an Apex Class to define a Default Constructor and assign the values to the Customer.

Class Code:

```
public class CustomerDetailsHelper
{
  Private integer customerCode;
   Private string customerName, address, phoneNumber;
   Private decimal balanceAmount;

   // Default Constructor to assign the pre-defined values for the members.
   Public CustomerDetailsHelper()
   {
      customerCode = 100001;
      customerName = 'Sampath Kumar';
      address = 'JP Nagar, Bangalore';
      phoneNumber = '9988776655';
      balanceAmount = 12500;
   }

   // Method to display the customer details..
   Public void ShowCustomerDetails()
   {
      system.debug('Customer Code is....: '+ customerCode);
      system.debug('Customer Name is...: ' + customerName);
      system.debug('Balance Amount is....: '+ balanceAmount);
      system.debug('Customer Address is...: '+ address);
   }
}
```

Execution:

```
// Invoking the Method..
CustomerDetailsHelper custHelper = new CustomerDetailsHelper();
```

```
custHelper.ShowCustomerDetails();
```

Parameterized Constructors:

UseCase: Create an Apex Class to define a Parameterized Constructor and assign the values to the Products. Supply the Product values to the constructor at runtime.

Class Code:

```
public class ProductDetailsHelper
{
   Private integer productCode;
   Private string productName, manufacturer;
   Private decimal unitPrice;
   Private Date manufacturedDate, expiryDate;

   // Parameterized Constructor to assign the values.
   Public ProductDetailsHelper(integer pCode, string pName, string pManufacturer, decimal pPrice, Date pmDate,
                               Date peDate)
   {
      // Assign the Parameter values to class variables..
      productCode = pCode;
      productName = pName;
      manufacturer = pmanufacturer;
      unitPrice = pPrice;
      manufacturedDate = pmDate;
      expiryDate = peDate;
   }

   // Method to Display the Product Details..
   Public void ShowProductsInfo()
   {
      system.debug('Product Code is...: '+ productCode);
      system.debug('Product Name is...: '+ productName);
      system.debug('Unit Price is...: '+ unitPrice);
      system.debug('Expiry Date is....: '+ expiryDate.format());
   }
}
```

Execution:

```
// Create the object and supply the product values to the Constructor..
ProductDetailsHelper prodHelper = new ProductDetailsHelper(2001, 'Desktop', 'DELL Inc.', 24000,
Date.newInstance(2019,04,23), Date.newInstance(2034, 04, 22));

// Invoke the method to print the product details...
        prodHelper.ShowProductsInfo();
```

COLLECTION CLASSES

Upon managing the elements by using Arrays, it causes the below issues.

1. Arrays Supports to store only Homogeneous elements. (i.e. It can hold similar type of elements. We can't store
   Heterogeneous elements in arrays)

2. Arrays Supports Static Memory allocation, as upon defining the array, we need to specify the size.

3. It doesn't provides the proper memory management, which causes the memory wastage.

4. Array size cannot be grow / shrink at runtime.

5. We don't have the readymade methods to manage the elements inside the array.

6. We can't insert / remove the element from the Middle of the array.

To avoid the drawbacks exist inside the Arrays, salesforce provides a set of readymade classes to manage a collection of elements efficiently, called as "Collection Classes".

Collections are used to manage a group of elements of type either Primitive type / SObject Type / Apex Type / Collection Type or a User Defined Type.

Collection allows us to store both Homogeneous and Heterogeneous elements.

Collections supports Dynamic Memory allocation, Hence the Collection size can grow / shrink at runtime.

Collections allows us to manage the memory very efficiently. To manage the elements inside the Collection (Adding / Removing/ etc.) we have a set of readymade methods available in each collection class.

Apex Provides the Below Collection Classes

1. **List Class**
2. **Set Class**
3. **Map Class**

List Collection Class:

List is an ordered collection, which preserves the insertion order. i.e. It will store the elements inside the collection, in the same order in which they were inserted.

List Collection allows us to store both Homogeneous and Heterogeneous elements.

List Collection supports to store Primitive type, SObject Type, Apex Type, User Defined Type and Collection type of elements.

List Collection supports Dynamic Memory allocation, Hence the Collection size can grow/shrink at runtime.

Note: List Collection allows us to store the Duplicate Elements inside it.

List Collection provides a set of readymade methods to manage the elements inside it.

Syntax: **List<Datatype> <objectName> = new List<Datatype>();**

Ex:

List<Integer> customerCodes = new List<Integer>(); --> It holds a collection of integer elements

List<string> countryNames = new List<string>(); --> It holds a collection of string elements

List<ID> recordIds = new List<ID>(); --> It holds a Collection of ID values.

List<Account> lstAccounts = new List<Account>(); --> It holds a collection of Account records.

List<Case> lstCases = new List<Case>(); --> It holds a Collection of Case Records.

List<Position__c> lstPos = new List<Position__c>(); --> It holds a collection of Position Records

List<Candidate__C> lstCands = new List<Candidate__C>(); --> It holds a collection of Candidate Records.

List<SObject> lstRecords = new List<SObject>(); --> It can hold any Object record.

List<Object> lstValues = new List<Object>(); --> It can hold any Primitive type values. (Heterogeneous)

Note: List Collection supports the Nested List Collection upto 5 levels.

Ex: List<List<List<List<List<string>>>>> lstValues = new List<List<List<List<List<string>>>>>();

List Collection class provides a set of readymade methods to manage the elements as below.

Methods:

Ex: List<string> lstNames = new List<string>();

1. **Add(<ElementName>):** This method is used to add a New element to the collection. Add method will add the new element at the end of collection.

Note: Each element inside the List collection will be identified by using an index position, which starts from "Zero".

```
Ex:     lstNames.Add('Welcome');
        lstNames.Add('1234');
        lstNames.Add('Hyderabad');
```

2. **AddAll(<CollectionName>):** This method is used to add a collection of elements to the list collection at a time.

```
Ex:     string[] countryNames = new string[]{'India','USA','UK','Australia','China'};
        lstNames.AddAll(countryNames);
```

3. **Add(<Integer IndexPosition>, <ElementName>):** This method is used to insert the element at the specified index position.

Ex: lstNames.Add(2, 'Bangalore');

4. **Size():** This method returns an integer value, which indicates the number of elements exist inside the collection.

Ex: lstNames.Size();

5. **Boolean IsEmpty():** This method is used to identify, whether the collection is Empty or not. It returns TRUE, when the collection is empty. Else it returns FALSE.

```
Ex:         if( lstNames.isEmtpy())
            {
            system.debug('Collection is Empty.');
            }
            else
            {
                    system.debug('Collection is Not Empty.');
            }
```

6. **Sort():** This method will arrange the collection elements in ascending order.

Ex: lstNames.sort();

7. **Remove(<indexPosition>):** This method is used to remove the element from the collection exist at the specified index position.

Ex: lstNames.Remove(1);

8. **Clear():** This method will remove all the elements from the collection.

Ex: lstNames.Clear();

9. **Get(<indexPosition>):** This method will return the element value exist at the specified index position.

```
Ex:     if(! lstNames.isEmtpy())
        {
                lstNames.Get(4);
        }
```

10. **Set(<indexPosition>,<ElementName>):** This method is used to Update the element value with the new name, which is exist in the specified index position.

Ex: lstNames.Set(2, 'Mumbai');

11. **Clone():** This method is used to create a copy of the collection.

Ex: lstNames.Clone();

12. **Boolean Contains(<ElementName>):** This method is used to find an element in the collection. It returns TRUE, when the specified element is found. Else it returns FALSE.

Ex: lstNames.Contains('Apex');

13. **Integer IndexOf(<ElementName>):** This method will returns the index of the specified element in the collection. If the element Not found in the collection, then it will returns "-1". If the element is repeated multiple times in the collection, then it will returns the index of the "First Occurrence" of the element.

Example:

```
// Creating the Object for List Collection Class..
        List<string> lstNames = new List<String>();

// Display the Collection size..
        system.debug('Collection Size is...: '+ lstNames.size());

// Adding the Elements to collection..
        lstNames.Add('Apex');
        lstNames.Add('Welcome');
        lstNames.Add('India');
        lstNames.Add('Australia');
        lstNames.Add('Hyderabad');
        system.debug('After Adding Elements, Collection Size is...: '+ lstNames.size());

        system.debug('Elements Are...: '+ lstNames);

// Inserting an element..
        lstNames.Add(1, 'Ameerpet');
        system.debug('After Insert, Collection Elements..: '+ lstNames);

// Adding Multiple Elements..
        string[] countryNames = new string[]{'USA','UK','China','Japan','Middle East'};
        lstNames.AddAll(countryNames);

        system.debug('Collection size...: '+ lstNames.size());
        system.debug('Collection Elements...: '+ lstNames);


// Adding Duplicate Elements..
        lstNames.Add('Banana');
        lstNames.Add('India');

        system.debug('After Adding Duplicates, Collection size is...: '+ lstNames.size());

// Check for collection is Empty or not..
        if(lstNames.isEmpty())
                system.debug('Collection is Empty.');
        else
                system.debug('Collection is Not Empty.');
```

```
// Search for an Element..
        if(lstNames.contains('Banana'))
                system.debug('Element Found.');
        else
                system.debug('Element Not Found.');

// Get the Index of an element..
        system.debug('Index of element Banana is...: '+ lstNames.IndexOf('Banana'));

// Get the Element based on Index position..
        system.debug('Element exist at the index position 4 is...: '+ lstNames.Get(4));

// Print the elements in seperate rows..
        for( string element : lstNames )
        {
                system.debug('Element Name is...: '+ element);
        }

// Removing an Element..
        if(lstNames.Contains('Hyderabad'))
        {
                lstNames.Remove(lstNames.IndexOf('Hyderabad'));
                system.debug('After Removing, collection size is..: '+ lstNames.size());
        }
        else
                system.debug('Element Not Found in the colleciton.');

//Replace the Element Value..
        lstNames.Set(2, 'Welcome to Hyderabad');
        system.debug('Elements are...: '+ lstNames);

// Sorting the Elements..
        lstNames.sort();
        system.debug('After Sorting...: ');

        for(string element : lstNames)
        {
                system.debug('Element Name...: '+ element);
        }

// Remove all elements..
        lstNames.Clear();
        system.debug('After Removing all, collection size..: '+ lstNames.size());
```

Set Collection Class:

Set is an unordered collection, which doesn't preserve the insertion order. Set collection will arrange the elements inside the collection in sorting order by default.

Set collection allows to store "Primitive Type elements, SObject Type, Apex Type, Collection type and User defined type" elements.

Set collection supports the Dynamic memory allocation, Hence the memory size can Grow / Shrink at runtime.

Set collection doesn't allow the Duplicate elements. Upon adding the new elements, it will maintains the uniqueness with the help of "Binary Comparison".

Set collection provides a set of readymade methods, to manage the elements inside the collection.

Syntax: **Set<Datatype> <objectName> = new Set<Datatype>();**

```
Ex:     Set<integer> customerCodes = new Set<Integer>();

        Set<string> countryNames = new Set<string>();
        Set<ID> recordIds = new Set<ID>();
        Set<Account> accountsSet = new Set<Account>();
```

Methods:

Ex: **Set<string> countryNames = new Set<string>();**

1**. Add(<ElementName>):** This method will add a new element to the collection. Upon adding the element, it will arrange the elements in the sorting order by default.

```
Ex:     countryNames.Add('India');
        countryNames.Add('Australia');
        countryNames.Add('USA');
        countryNames.Add('China');
```

2. **AddAll(<CollectionName>):** This method is used to add a collection of elements from the source collection.

```
Ex:  string[] countries = new string[]{'UK','Middle East', 'Japan','Bangladesh','India'};
     countryNames.AddAll(countries);
```

3. **Integer Size():** It returns an integer, which indicates the number of elements exist inside the collection.

```
Ex: countryNames.Size();
```

4. **Boolean isEmpty():** This method returns TRUE, when the collection is Empty. Else it returns FALSE.

```
Ex:     if(countryNames.isEmpty())
        {
                system.debug('Collection is Empty.');
        }
        else
```

```
{
        system.debug('Collection is Not Empty.');
}
```

5. **Boolean Contains(<ElementName>):** This method is used to search for the specified element in the collection. It returns TRUE, when the element found. Else it returns FALSE.

Ex: countryNames.Contains('USA')

6. **Remove(<ElementName>):** This method is used to remove the specified element from the collection.

Ex: countryNames.Remove('china');

7. **Clear():** This method will remove all the elements from the collection.

Ex: countryNames.Clear();

8. **Equals(<Collection>):** This method is used to compare the two collections. It returns TRUE, when both the collections are same. Else it returns FALSE.

Ex: countryNames.Equals(<CollectionName>);

MAP Collection Class:

Map collection allows us to manage a collection of elements, where each element contains "Key" and "Value". i.e. Map collection allows us to store "Key-Value" pair collections.

Upon storing the elements, Key should be unique and "Value" may be unique / duplicate.

Each element can be identified by using "Key". It will arrange the elements inside the collection based on the "Key" in sorting order. Hence, Map is an Unordered collection.

Map collection supports Dynamic Memory allocation, which makes the Collection Size can Grow / Shrink at runtime.

Map collection maintains the uniqueness, based on the "Key". Hence map collection doesn't allow Duplicate elements.

Key and Value can be either Primitive / SObject / Apex / Collection / User Defined type.

Syntax:

```
Map<KeyDatatype, ValueDatatype> <objectName> = new Map<KeyDatatype, ValueDatatype>();
```

Ex:

```
Map<Integer,integer> customerCodes = new Map<Integer, Integer>();
Map<Id, string> recordDetails = new Map<ID,string>();
Map<string,string> countryCodes = Map<string, string>();
Map<Id, Account> accountsMap = new Map<ID, Account>();
Map<Id, Position__c> positionsMap = new Map<Id,Position__c>();
Map<Account,List<Contact>> contactsMap = new Map<Account, List<Contact>>();
```

Methods:

Map collection provides a set of instance methods, to manage the elements as below.

Ex: **Map<string, string> fruitsMap = new Map<string, string>();**

1. **Put(<Key>,<Value>):** This method is used to add a New Element to the collection.

```
Ex:     fruitsMap.Put('Green','Apple');
        fruitsMap.Put('Red','Apple');
        fruitsMap.Put('Black','Grapes');
        fruitsMap.Put('Yellow','Banana');
```

2. **PutAll(<Map Collection>):** This method is used to add a collection of elements from one Map to another Map collection.

```
Ex: fruitsMap.PutAll(<MapCollectionName>);
```

3**. Integer Size():** This method is used to get the collection size.

```
Ex: fruitsMap.Size();
```

4. **Boolean isEmpty():** This method is used to check, wether the specified collection is Empty or not.

```
Ex: fruitsMap.isEmpty();
```

5. **Get(<KeyName>):** This method is used to get the value of the Specified Key.

```
Ex:     if(! fruitsMap.isEmpty())
        {
                system.debug('Element value is..: '+ fruitsMap.Get('Yellow'));
        }
```

6. **Boolean ContainsKey(<KeyName>):** This method is used to check, wether the specified key is exist inside the collection or not. It returns TRUE, when the key found. Else it returns FALSE.

```
Ex:     if(fruitsMap.ContainsKey('Red'))
        {
                system.debug('Element value is...: '+ fruitsMap.Get('Red'));
        }
        else
        {
                system.debug('Element Not Found.');
        }
```

7. **Remove(<KeyName>):** By using this method, we can remove the element associated with the specified "Key".

```
Ex:     if( fruitsMap.ContainsKey('Red'))
        {
```

```
        fruitsMap.Remove('Red');
    }
    else
    {
        system.debug('Invalid Key.');
    }
```

8. **Clear():** This method will remove all the elements from the collection.

    Ex: fruitsMap.Clear();

9. **Set<Datatype> KeySet():** This method will returns all the Key's exist inside the collection.

    Ex: Set<String> keysCollection = fruitsMap.KeySet();

10. **List<Datatype> Values():** This method will return all element values exist in the collection.

    Ex: List<string> valuesCollection = fruitsMap.Values();

11. **Boolean Equals(<SourceCollection>):** This method is used to compare the Two Map collecitons. And it will returns TRUE, when both the Map collecitons are Identical. Else it returns FALSE.

    Ex: fruitsMap.Equals(<SourceMapCollectionName>);

Example:

```
// Defining the Map Collection...
Map<string, string> fruitsMap = new Map<string,string>();

// display the collection size...
system.debug('Collection size is....: '+ fruitsMap.Size());

// Adding Elements to collection...
fruitsMap.Put('Yellow','Banana');
fruitsMap.Put('Orange','Orange');
fruitsMap.Put('Black','Grapes');
fruitsMap.Put('Green','Apple');

// Display the Collection Elements..
system.debug('Collection Elements are...: '+ fruitsMap);

// Display the elements in seperate rows...
for(string keyName  : fruitsMap.KeySet())
{
    system.debug('Key is : '+ keyName + '  ---> Value is...: '+ fruitsMap.Get(keyName));
}
```

```
// Display the Keys Collection..
system.debug('Key Names are....: '+ fruitsMap.KeySet());

// Display the Values...
system.debug('Values are.....: '+ fruitsMap.Values());

// Adding the Elements...
fruitsMap.Put('Red','Apple');
system.debug('After adding elements, collection size is...: '+ fruitsMap.size());

// Adding Duplicate Elements..
fruitsMap.Put('Red','Cherry');
system.debug('After adding Duplicates, collection size is...: '+ fruitsMap.size());
system.debug('Collection Elements are....: '+ fruitsMap);

// Check for the element existance...
if(fruitsMap.ContainsKey('Yellow'))
{
        system.debug('Element Value is....: '+ fruitsMap.Get('Yellow'));
}
else
{
        system.debug('Element Not Found.');
}

// Remove an element from the collection...
if(fruitsMap.ContainsKey('Red'))
{
        fruitsMap.Remove('Red');
        system.debug('After Deleting, Collection elements are...: '+ fruitsMap);
}
else
{
        system.debug('Invalid Key Name.');
}

// Remove all elements..
fruitsMap.Clear();
system.debug('After removing all elements, collection size is....: '+ fruitsMap.size());
```

Bulkification / Bulkify Process:

Upon performing the DML operations on the Records, salesforce provides the Governor Limits, that we can have max. of 150 DML statements inside a single transaction.

If the user tried to perform more than 150 DML operations inside a transaction, it will causes an exception called as "**System.Limit Exception Too Many DML Statements: 151**".

To avoid this problem, always we have to make our code "Bulkify". i.e. Instead of performing the operations on each record seperately, we can use the collections.

Add all the records to collection. Once all the records has been added to the collection, then perform the DML operations on the Collection of records at a time outside the FOR loop.

Note: As a best practice, it is always recommended to avoid the usage of DML statements inside the "FOR Loop".

/* Write an apex program, to insert 300 Contact records into the contact object. */

Class Code:

```
public class ContactsHandler
{
   Public static void InsertBulkContacts(integer noOfcontacts)
   {
      // Creating a Collection...
      List<Contact> lstContacts = new List<Contact>();

      // Creating the Contact Records...
      if(noOfContacts > 0)
      {
         for(integer counter = 1; counter <= noOfContacts; counter++)
         {
            // Create a contact record...
            Contact con= new Contact();

            con.FirstName = 'Apex';
            con.LastName = 'Contact - '+ counter;
            con.Email = 'apexContact'+counter+'@gmail.com';
            con.Title = 'Sales Rep.';
            con.Department = 'Sales';
            con.Phone = '9900990099';
            con.Fax = '9988776655';
            con.MobilePhone = '9900990000';
            con.mailingStreet = 'Kukatpally, Hyderabad';
            con.MailingCity = 'Hyderabad';
            con.MailingState = 'Telangana';
```

```
            // add the record to collection..
            lstContacts.Add(con);
        }

        if(! lstContacts.isEmpty())
        {
            insert lstContacts;
        }
      }
    }
}
```

Execution:

```
// Calling the method...
ContactsHandler.InsertBulkContacts(300);
```

SALESFORCE OBJECT QUERY LANGUAGE (SOQL Queries)

SOQL Stands for Salesforce Object Query Language.

By using SOQL Queries, we can fetch either one or more records from the Salesforce Objects based on the need.

Upon fetching the records, we can add the required user defined conditions to filter the data.

By using SOQL Queries, we can fetch the records from one or more associated objects. (i.e. We can fetch the data from the related objects)

Query: Query is nothing but a request to the Database.com, to fetch the required records from one or more objects. SOQL Query should be prepared with the help of "**SELECT**" statement as below

Syntax:

```
SELECT <FieldName / API Names> from <ObjectName>
        [ WHERE <Conditions> ]
        [ GROUP BY <Column Names> ]
        [ HAVING <Conditions> ]
        [ ORDER BY <Column Names> ]
        [ LIMIT <Number of Records To Return> ]
        [ OFFSET <Number of Records to Skip> ]
        [ FOR UPDATE ]
        [ ALL ROWS ]
```

Governor Limits:

1. We can have max. of 100 SOQL Queries inside a Transaction.

   When the user tries to execute more than 100 SOQL Queries inside a Transaction, then SalesForce will hit the Governor Limit and will cause an exception "System.LimitException Too Many SOQL Queries : 101".

2. Each SOQL Query can returns max. of 50,000 records.

Best Practices:

1. Upon writing the SOQL Queries, we need to avoid the usage of SOQL Query inside the "FOR Loop".

   If the FOR Loop is repeating for more than 100 times, then it will hit the governor limit, which will cause the exception.

2. While writing the SOQL Queries, we need to specify the required columns inside the Query. Don't fetch the unnecessary fields information from the object.

Ways to invoke SOQL Queries:

Salesforce provides the below 2 ways to invoke the SOQL queries.

1. By using "Query Editor" from the Developer Console.

   Click on Your Name and expand it.

   1. Click on "Developer Console" link.
   2. Goto the "Developer Console Editor".
   3. Click on "Query Editor" tab from the TabBar.
   4. Type the SOQL Query inside the Query Window.
   5. Click on "Execute Button".
   6. See the result in the "Result Window".

2. By using "Apex Programming".

/* Write an SOQL Query to fetch the Account id, Name, rating, industry, annualrevenue and Active Status values from the Account Object. */

```
Select id, name, rating, industry, annualrevenue, active__c
    from Account
```

/* Write an SOQL Query, to fetch all the Lead Records with te columns 1d, FirstName, LastName, Company, Status, Phone, Fax, City, Lead Source. */

```
Select id, firstname, lastname, email, title, company, status, phone, fax, city, leadsource from Lead
```

/* Write an SOQL Query to fetch all the Email Templates information. */

```
Select id, name, description, subject, body, templatetype from EmailTemplate
```

/* Write an SOQL Query to fetch the User Details. */

```
Select id, firstname, lastname, email, username,
                profile.Name, isactive
                from User
```

/* Write an SOQL Query, to fetch all the Hiring Manager Details. */

Select id, name, location_del__c, email_id__c, contact_number__C from Hiring_Manager__C

By using Apex Programming also, we can invoke the SOQL Queries based on the need.

Apex provides 2 types of SOQL Queries.

1. **Static SOQL Queries**
2. **Dynamic SOQL Queries.**

Static SOQL Queries:

In this approach, the SOQL Query should be enclosed inside the "[ ]" (i.e. in Square Braces)

Syntax: [SOQL Query]

Note: Static SOQL Queries will be executed automatically without manual execution.

Upon preparing the Query, we need to specify all the required columns and Conditions.
We can't add any more columns and conditions to the Static SOQL Query at runtime.

Ex: [Select id, name, rating, industry, annualrevenue from Account]

Once the Query gets executed, then it will return the results.

1. If the Query Returns Only One Record:

If the SOQL Query returns Only One record as a result, then we need to store the result, by creating a variable of the associated type.

Syntax: <SObjectDataType> <variableName> = [SOQL Query Returns Only 1 Record];

Ex:

```
Account acc = [Select id, name, rating, industry, annualrevenue from Account];

system.debug('Account Record is...: '+ acc);

if(acc != null)
{
```

```
        system.debug('Account id..: '+ acc.id);
        system.debug('Account Name ...: '+ acc.name);
        system.debug('Industry Name...: '+ acc.industry);
    }
```

2. <u>If SOQL Query Returns Multiple Records:</u>

If the Query returns Multiple records as the result, then we need to store the results inside a "List Collection".

Syntax:

List<DataType> <objectName> = [SOQL Query];

Once the records has been stored inside the collection, then we need to iterate the collection and display the records.

Ex:

```
List<Hiring_Manager__c> lstHrs = [Select id, name,
                                  location__C, contact_Number__C
                                  from Hiring_Manager__c];

system.debug('Collection Size is..: '+ lstHrs.Size());

if(! lstHrs.isEmpty())
{
    for(Hiring_Manager__c hr : lstHrs)
    {
        system.debug('Id is..: '+ hr.id);
        system.debug('Name is...: '+ hr.name);
        system.debug('Location is..: '+ hr.location__c);
    }
}
```

/* Write an apex program, to fetch all the Records from the Lead Object. */

<u>Class Code:</u>

```
public class DatabaseUtility
{
   Public static void GetAllLeadRecords()
   {
      List<Lead> lstLeads = [Select id, firstname, lastname, company, status, email, phone,fax
                             from Lead];

      system.debug('Collection Size is...: '+ lstLeads.size());
```

```
        if(! lstLeads.isEmpty())
        {
           for(Lead ld : lstLeads)
           {
              system.debug('Lead Id...: '+ ld.id);
              system.debug('Lead Name ...: '+ (ld.FirstName + ' '+ ld.LastName));
              system.debug('Company Name is...: '+ ld.Company);
              system.debug('Lead Status....: '+ ld.Status);
              system.debug('Contact Number...: '+ ld.Phone);
              system.debug('Fax Number...: '+ ld.Fax);
              system.debug('Email Id .....: '+ ld.Email);

              system.debug('---------------------------------------');
           }
        }
     }
}
```

Execution:

```
// Create the object of the Class.
          DatabaseUtility.GetAllLeadRecords();
```

/* Write an apex program, to get All the User Records from the User Object. */

Class Code:

```
public class DatabaseUtility
{
   Public static void GetUserRecords()
   {
      for(User u : [Select id, firstname, lastname, email,username, profile.name, isactive
                                          from user])
         {
            system.debug('User Record is...: '+ u);
         }
   }
}
```

Execution:

```
// Invoke the Method..
          DatabaseUtility.GetUserRecords();
```

/* Write an apex program, to fetch all the Position Records and store the results inside the "Map Collection".*/

Class Code:

```
public class DatabaseUtility
{
   Public static void GetPositionDetails()
   {
      Map<id, Position__c> mapPositions = new Map<id, Position__c>([Select id, name,
                                          Location_Name__c, position_status__c,
                                                maximum_budget__C, milestone_date__C
                                                from Position__c]);
     system.debug('Collection Size is...: '+ mapPositions.size());

     if(! mapPositions.isEmpty())
     {
        /*
        // Case 1: Get Positions
        for(Id recordId : mapPositions.KeySet())
        {
          system.debug('Position Record is..: Id...: '+ recordId + '  --> Record: '+ mapPositions.Get(recordId));
          system.debug('Position Name is...: '+ mapPositions.Get(recordId).Name);
        }
       */

        // Case 2: Get Positions
        for(Position__C p:  mapPositions.Values())
        {
           system.debug('Position Record is..: '+ p);
        }
     }
   }
}
```

Execution:

```
// Invoke the Method..
        DatabaseUtility.GetPositionDetails();
```

Assignments:

1. Write an apex program to fetch All the Email Templates from the Object and display on the Debug Log File.

2. Write an apex program, to fetch all the Cases in the form of a "Map Collection". And display the records.

3. Write an apex program, to fetch all the Public Groups information from the Object.

4. Write an apex class to get all the Opportunity Records into the Map collection.

Limit Clause:

This clause is used to restrict the number of records to be get returned by the SOQL Query. It is an optional statement in SOQL Query.

Syntax: [SOQL Query Limit <Integer>]

Ex:

Limit 1
Limit 5
Limit 10

Note: If the SOQL Query has been restricted to return Only One Record (i.e. Limit 1), then we need to store the result by defining a variable of the associated type.

Ex: Account acc = [Select id, name, rating, industry, annualrevenue, active__c
from Account Limit 1];

Assignments:

1. Write an apex program, to fetch the 5 Records from the Contact Object and store the results in the Map collection.

2. Write an apex program, to fetch only One User Record from the object.

ALL ROWS:

This clause is used to get all the records from the object including "Deleted Records" also. It is an optional statement in SOQL Query.

Syntax:

[SOQL Query ALL ROWS]

/* Write an apex program, to fetch all the Account Records from the Object, including Deleted Records also.*/

Class Code:

```
public class DatabaseUtility
{
  Public static void getAllAccountRecords()
  {
    List<Account> lstAccounts = [Select id, name, rating, industry
                                 from Account
                                       ALL ROWS];

    system.debug('Collection Size is...: '+ lstAccounts.size());
  }
}
```

Execution:

```
// Invoke the Method..
        DatabaseUtility.getAllAccountRecords();
```

ORDER By Clause:

By using this clause we can arrange the resultset records either in "Ascending / Descending" order based on one or more columns.

Syntax:

```
[SOQL Query ORDER BY <ColumnName> [ASC (Default) / DESC] ]
```

Note: If the user didn't specify any sorting order, then it will arrange the records in "Ascending Order" by default.

/* Write an apex program, to fetch all the Account Records from the object and arrange in Ascending order based on the Name. */

```
List<Account> lstAccounts = [Select id, name, rating, industry, annualrevenue, active__c, phone, fax
                                                        from Account ORDER BY name];
```

/* Write an SOQL Query, to fetch all the Position Records from the Object. And arrange the records in the form of "Recently created Records first". */

```
List<Position__C> lstPositions = [Select id, name, location__C, position_status__c, createddate,
        maximum_budget__c, close_date__C
                from Position__C
                        Order by createddate desc ];
```

FOR UPDATE:

By using this clause, we can apply the lock on the required records. So that other users can't perform any other operations on my records.

Once the transaction has been done, then the records will gets unlocked automatically.

Ex:

```
List<Lead> lstLeads = [Select id, firstname, lastname, email, company, phone, fax, status
                                                        from Lead FOR Update];
```

OFFSET Clause:

This clause is used to skip the specified number of records from the resultset. This is an optional statement in SOQL Query.

Syntax:

[SOQL Query OFFSET <Number>]

Ex:

OFFSET 10
OFFSET 20

This clause should be used upon implementing the "**Pagination / Paging**" feature.

Note: Always "OFFSET" clause should be followed by "LIMIT" clause.

/* Write an apex program, to fetch te records from the object, based on the specified Limit and Offset values at runtime. */

Class Code:

```
public class DatabaseUtility
{
   Public static void GetRecordsByLimits(integer limitValue, integer offsetValue)
   {
         List<Account> lstAccounts = [Select id, name, rating, industry, annualrevenue,
                                       active__C, phone, fax
                                       from Account
                                             ORDER BY Name
                                                   Limit : limitValue
                                                   OFFSET : offsetValue];

      if(! lstAccounts.isEmpty())
      {
         for(Account acc : lstAccounts)
         {
           system.debug('Record is...: '+ acc);
         }
      }
   }
}
```

Execution:

DatabaseUtility.GetRecordsByLimits(10, 5);

WHERE Clause:

This clause is used to add one or more user defined conditions to be get applied along with the SOQL Query, to filter the records from the resultset.

Syntax:

[SOQL Query Where <Conditions>]

Note: We can add multiple conditions along with the Query, with the help of "Logical Operators".

Ex:

```
[Select id, name, rating, industry from Account
                    Where rating = 'Hot']
```

/* Write an SOQL Query, to fetch all the Banking Account Records. */

```
List<Account> lstAccounts = [Select id, name, rating, industry, annualrevenue,
            phone, fax, active__C
                    from Account
                        Where industry = 'Banking'];
```

/* Write an SOQL Query, to fetch all the Lead Records whose status is "Closed - Converted".*/

```
List<Lead> lstLeads = [Select id, firstName, lastname, email, company, status, phone, fax
            from Lead
                    Where status = 'Closed - Converted'];
```

/* Write an SOQL Query to fetch all the Hyderabad Related Position and arrange the records in Alphabetical order based on the name. */

```
List<Position__C> lstPos = [Select id, name, location__C, position_status__c,
                    maximum_budget__c, close_date__C
                        from Position__C
                            Where location__c = 'Hyderabad'
                                ORDER by name    Limit 10];
```

/* Write an SOQL Query, to fetch only the Deleted Records from the Account Object. */

```
List<Account> deletedAccounts = [Select id, name, rating, industry,
            isDeleted, active__c
                    from Account
                        Where isDeleted = true
                        ALL ROWS];
```

/* Write an SOQL Query to fetch all the Opportunies whose stage is "Prospecting" and Amount value is more than 15000. */

```
List<Opportunity> lstOppty = [Select id, name, amount, stagename, closedate
            from Opportunity
                    Where stagename = 'Prospecting' and amount > 15000];
```

/* Write an SOQL Query to fetch all the Active Users who are associated to "CIS Platform Profile". */

```
List<User> activeUsers = [Select id, firstname, lastname, isactive, profile.name, email
                          from User
                              Where profile.name = 'CIS Platform Profile' and
                                  isActive = true];
```

Note: We can supply the value for the Condition inside the SOQL Query dynamically at runtime with the help of "Parameterized Procedures" as below.

Syntax:

```
[SOQL Query Where <ColumnName> = : <parameterName> ]
```

Ex:

```
string indName = 'Finance';

List<Account> lst = [SELECT id, name, rating, industry from Account
                                                  Where industry =: indName];
```

/* Write an Apex program, to fetch all the Positions based on the specified Location at runtime. */

Class Code:

```
public class DatabaseUtility
{
   Public static void GetPositionsByLocation(string cityName)
   {
      if(cityName != null && cityName != '')
      {
         List<Position__C> lstPos = [Select id, name, location_name__C,
                                     position_status__C, maximum_budget__c, milestone_date__C
                                     from Position__C
                                         Where location_name__c =: cityName
                                             Order by name];

         system.debug('Collection size is...: '+ lstPos.size());

         if(! lstPos.isEmpty())
         {
           for(Position__C pos : lstPos)
           {
              system.debug('Position...: '+ pos);
           }
         }
      }
   }
}
```

Execution:

```
DatabaseUtility.GetPositionsByLocation('Chennai');
DatabaseUtility.GetPositionsByLocation('Hyderabad');
```

LIKE Operator:

By using Like Operator, we can fetch the similar kind of information from the Object, based on the specified expression.

Note: Like Operator should be always used along with the "WHERE Clause".

Syntax:

```
WHERE <ColumnName> LIKE 'Expression'
                    (OR)
WHERE <ColumnName> LIKE : <Expression>
```

Note: Upon preparing the expression, we have to use the "Wild Card Characters" as below.

% --> Represents Zero or More Characters

_ (UnderScore) --> Represents Only One Character.

Ex:

Where Name like 'R%' --> Returns all accounts whose name is starting with "R"

Where Name like '%Kumar' --> Returns all accounts whose name ending with "Kumar"

Where name like '_a%' --> Returns all accounts, whose name's second character "a".

Where name like '____' --> Returns all accounts whose name contains 4 characters.

Where name like '%ram%' --> Returns All accounts, whose name contains "ram".

/* Write an SOQL Query, to fetch all the Accounts, whose name is starting with the word 'United'. */

```
List<Account> lstAccounts = [Select id, name, rating, industry,
                                annualrevenue, active__C
                            from Account
                                Where name like 'United%'];
```

/* Write an SOQL Query, to fetch all the Leads Information, whose Street contains "pally". */

```
List<Lead> lstLeads = [Select id, firstname, lastname, company,
                            status, street
                        from Lead
```

```
Where street like '%pally%'];
```

/* Write an SOQL Query, to fetch all the Position Details, Whose name's second character is 'a'. */

```
List<Position__C> lstPos = [Select id, name, location__c, position_status__C,
                            from Position__C
                                Where name like '_a%'];
```

/* Write an apex program, to fetch all the Account Records, based on the starting character at runtime. */

Class Code:

```
public class DatabaseUtility
{
   Public static void GetAccountsByStartingChar(string startingChar)
   {
      if(startingChar != null && startingChar != '')
      {
          startingChar += '%';

        List<Account> lstAccounts = [Select id, name, rating, industry,
                                    annualrevenue, active__c
                                        from Account
                                    Where name like : startingChar];

        system.debug('Collection Size...: '+ lstAccounts.size());

        if(! lstAccounts.isEmpty())
        {
          for(Account acc : lstAccounts)
          {
              system.debug('Account Record...: ' + acc);
          }
        }
      }
   }
}
```

Execution:

```
DatabaseUtility.GetAccountsByStartingChar('a');

DatabaseUtility.GetAccountsByStartingChar('Un');

DatabaseUtility.GetAccountsByStartingChar('Bulk');
```

IN Operator:

/* Write an apex program, to fetch all the Contacts associated with the "Banking Industry Accounts". */

Class Code:

```
public class DatabaseUtility
{
   Public static void GetContactsByIndustryName(string industryName)
   {
      if(industryName != null && industryName != '')
      {
         // Get All the Accounts based on the specified industry...
         Map<ID, Account> mapAccounts = new Map<Id,Account>([Select id, name, industry
                                    from Account
                                          where industry =: industryName]);

         system.debug('Collection size...: '+ mapAccounts.size());

         if(! mapAccounts.KeySet().isEmpty())
         {
            // Get the Related Contacts of the Accounts..
            List<Contact> lstContacts= [Select id, firstname, lastname,
                                    email, phone, fax, accountid
                                          from Contact
                                          Where accountid IN : mapAccounts.KeySet()];

            if(! lstContacts.isEmpty())
            {
              system.debug('Contacts Collection...: '+ lstContacts.size());

              // Display the Contact Records...
              for(Contact con : lstContacts)
              {
                 system.debug('Contact Record...: '+ con);
              }
            }
         }
      }
   }
}
```

Execution:

```
DatabaseUtility.GetContactsByIndustryName('Energy');
```

Dynamic SOQL Queries:

By using Dynamic SOQL Queries, we can include the Column names / Field Names inside the Query at runtime. And we can add the required conditions to the Query at runtime based on the requirement.

To achieve this, we need to store the SOQL Query inside a string variable.

Syntax: string <variableName> = 'SOQL Query';

Ex:

```
string accountsQuery = 'Select id, ';
        ...
        ...
        ...
                accountsQuery += 'name, industry,';
        ....
        ....
                accountsQuery += 'annualrevenue, active__C';

        ...
        ...
                accountsQuery += ' from Account';
```

Once the Query has been prepared, then we need to execute the Query manually by using "Database.Query()" method.

Note: Database.Query() method will always returns the results in the form of "List Collection".

Syntax: List<SObjectType> <objectName> = Database.Query(SOQL Query);

Ex: List<Account> lstAccounts = Database.Query(accountsQuery);

/* Write an apex program, to fetch all the Candidate Records from the Object based on the specified Location name. */

Class Code:

```
public class DatabaseUtility
{
   Public static void GetCandidatesByLocation(string cityName)
   {
      if(cityName != null && cityName != '')
      {
         string candidatesQuery = 'Select id, name, location__C, contact_number__C, email_id__C from
Candidate__c where location__c =: cityName';

         List<Candidate__C> lstCands = Database.query(candidatesQuery);
```

```
        system.debug('Collection Size is...: '+ lstCands.size());

        if(! lstCands.isEmpty())
        {
           for(Candidate__c cnd : lstCands)
           {
              system.debug('Candidate Record...: '+ cnd);
           }
        }
     }
  }
}
```

Execution:

```
DatabaseUtility.GetCandidatesByLocation('Chennai');
```

/* Write an apex program, to fetch all the Lead Records from the Objects based on the Input Values supplied by the user at runtime.

If the user supplies a normal string value, then get the data based on the "Last Name" of the Lead Record.

If the user supplies an email id, then get the data based on the email id from the object.

***/**

Class Code:

```
public class DatabaseUtility
{
   Public static void GetLeadRecordsBySearchText(string searchText)
   {
     if(searchText != null && searchText != '')
     {
        string leadsQuery = 'Select id, firstname, lastname, email, status, company, phone from Lead Where ';

        if(pattern.matches('[a-zA-Z0-9.+_-]+@[a-zA-Z.-]+.[a-zA-Z]{2,4}', searchText))
        {
           system.debug('Input Value is in the form of Email ID.');
           leadsQuery += ' email =: searchText';
        }
        else
        {
           system.debug('Input Value is in the form of a String.');

           searchText += '%';
```

```
            leadsQuery += ' lastname like : searchText order by lastname';
        }

        List<Lead> lstLeads = Database.query(leadsquery);
        system.debug('ResultSet size...: '+ lstLeads.size());

        if(! lstLeads.isEmpty())
        {
            for(Lead ld : lstLeads)
            {
                system.debug('Lead Record...: '+ ld);
            }
        }
    }
  }
}
```

Execution:

```
DatabaseUtility.GetLeadRecordsBySearchText('bulktestlead1@gmail.com');
DatabaseUtility.GetLeadRecordsBySearchText('tes');
```

Group By Clause:

By using Group By clause, we can segregate the records into the various groups based on one or more columns inside the object.

Note: Group By Clause should be always used, along with the "Aggregate Functions".

Aggregate Functions:

Aggregate Functions will receive a collection of values as the input and will perform the operations on the records and it return only one value as the result.

SOQL Supports the below Aggregate Functions

1. Count():

This function will returns an integer, which indicates the number of records exist inside the object.

2. Count(<ColumnName>):

This function will returns an integer, which indicates the number values exist inside the specified column.

Note:

In this function, "NULL" values will not be countable.

3. Count_Distinct(<ColumnName>):

This function will returns an integer, which indicates the number of values exist inside the object.

Note:

But, in this function it won't count the "NULL values" and "Repeated / Duplicate" values.

4. Sum(<ColumnName>):

This function will returns the "Sum / Total" of all the values exist inside the specified column.

Note:

We can supply the "Number / Percent / Currency" columns as the input.

5. Min(<ColumnName>):

This function will returns the smallest value exist inside the specified column.

Note: We can supply either "Number / Percent / Currency / Date / Time / DateTime" fields as the input.

6. Max(<ColumnName>):

This function will returns the Highest value exist inside the specified column.

Note: We can supply either "Number / Percent / Currency / Date / Time / DateTime" fields as the input.

/* Write an SOQL Query, to findout the Number of records exist inside the "Account" object. */

Class Code:

```
public class DatabaseUtility
{
    Public static void GetRecordsCount()
   {
      // Static SOQL Query..
      integer recordsCount =       [Select count() from Account];

      system.debug('Account Records Count is..: '+ recordsCount);

      // Dynamic SOQL Query..

      string accountsQuery = 'Select count() from Account';

      integer accountRecordsCount = Database.countQuery(accountsquery);

      system.debug('Account Records Count is...: '+ accountRecordsCount);

   }
}
```

Execution:

DatabaseUtility.GetRecordsCount();

AggregateResult Class:

Upon using the "Aggregate functions" inside the SOQL Query, the result should be of type an "AggregateResult" class.

Syntax: AggregateResult[] results = [SOQL Qury with Group By Clause];

We can fetch each record inside the resultset by iterating the collection.

We can fetch each value inside the resultant record by using "Get" function.

Syntax: <objectName>.Get('FieldName / AliasName');

Note:

For Aggregate functions inside the SOQL Query, we have to provide an "Alias".
Alias is a Temporary name allocated to a column, to fetch the value.

Ex:

[select industry, count(id) recordsCount from Account
Group By Industry]

/*

Write an Apex Program, to list out all the Industry Names and the number of associated accounts count with each industry.

*/

Class Code:

```
Public Class DatabaseUtility
{
    Public static void GetAccountIndustryWithRecordsCount()
  {
    AggregateResult[] aggResult = [Select industry, count(id) recordsCount
                                   from Account
                                   Group By industry];

    system.debug('Collection size is...: '+ aggResult.size());

    for(AggregateResult res : aggResult)
    {
      system.debug('Industry Name...: '+ res.Get('Industry') + '    ---> Count : '+ res.Get('recordsCount'));
    }
  }
}
```

Execution:

DatabaseUtility.GetAccountIndustryWithRecordsCount();

Having Clause:

Having Clause is used to add one or more user defined conditions along with the Group By clause, to filter the data.

Note:

Always, Having Clause should be used along with the "Group By" Clause. Having Clause should be always followed by the GroupBy Clause.

/*

Write an apex program, to display the Industry names, and the Number of records in each industry. Which are having minimum 2 records inside the group.

*/

Class Code:

```
Public Class DatabaseUtility
{
   Public static void GetAccountIndustryWithRecordsCount()
   {

         AggregateResult[] aggResult = [Select industry, count(id) recordsCount
                                                from Account
                                                Group By industry
                                                Having count(id) >= 3];


      system.debug('Collection size is...: '+ aggResult.size());

      for(AggregateResult res : aggResult)
      {
         system.debug('Industry Name...: '+ res.Get('Industry') + '    ---> Count : '+ res.Get('recordsCount'));
      }
   }
}
```

Execution:

DatabaseUtility.GetAccountIndustryWithRecordsCount();

/* Write an apex program, to fetch all the Duplicate Account records based on the Account Name. */

Class Code:

```
public class DatabaseUtility
{
```

```
  Public static void GetDuplicateAccounts()
  {
    AggregateResult[] results = [Select name, count(id) recordsCount
                                 from Account
                                      Group By name
                                           having count(id) > 1 ];

    system.debug('Duplicate Accounts Count...: '+ results.size());

    for(AggregateResult res : results)
    {
      system.debug('Account Name..: '+ res.get('name') + '  ---> Count : '+ res.get('recordsCount'));
    }
  }
}
```

Execution:

```
DatabaseUtility.GetDuplicateAccounts();
```

RELATIONSHIP QUERIES

Upon fetching the data from the multiple objects, we need to write a separate SOQL Query to fetch the data from each object.

Which will increase the number of SOQL Queries inside the application, which may leads to the exception "Too Many SOQL Queries: 101".

To avoid this problem, we can fetch the data from multiple objects by using a single SOQL Query.

Note: Upon using the Relationship Queries, both the objects should be associated by using either "Lookup / Master-Detail".

We can fetch the Data from the Related objects with the below 4 scenarios.

1. **Parent To Child (Standard Relationship)**
2. **Child To Parent (Standard Relationship)**
3. **Parent To Child (Custom Relationship)**
4. **Child To Parent (Custom Relationship)**

Parent To Child (Standard Relationship):

By using this approach, we can fetch the Parent Records and the associated Child records by using a single SOQL Query.

The objects should be associated, with the Standard / Readymade association. We can implement this features with the help of "Inner Queries / Sub Queries".

Syntax:

[Parent SOQL Query (Child SOQL Query)]

Note: To fetch the associated child records, we need to write the "Inner Query", which should be enclosed inside the braces "( )" .

Note: In Inner Query, the Child object name should be in "Plural format".

Ex:

(Select id, firstname, lastname, email, phone from Contacts)
(Select id, name, amount, stagename from Opportunities)
(Select id, subject, description from Cases)

Note: We can Write one or more Inner Queries inside a single SOQL Query.

Note: SOQL supports Only One Level of Inner Queries. (i.e. Inside an Inner Query, we can't write an another Inner Query)

/* Write an apex program, to fetch the Account Records based on the specified name. And fetch all the associated Contacts, Opportunities, and Case Records. */

Class Code:

```
public class DatabaseUtility
{
   Public static void GetParentAndChildRecordsByName(string accountName)
   {
      if(accountName != null && accountName != '')
      {
        List<Account> lstAccounts =      [Select id, name, rating, industry,
                          annualrevenue, active__C, phone, fax,
                            (Select id, firstname, lastname, email, phone, fax from Contacts),
                            (Select id, name, amount, stagename from Opportunities),
                            (Select id, casenumber, priority, status, origin from Cases)
                                                    from Account where name =: accountName];

        system.debug('Collection Size....: '+ lstAccounts.size());

        if(! lstAccounts.isEmpty())
        {
           for(Account acc : lstAccounts)
           {
              system.debug('Account Record is..: '+ acc);
```

```
            // Get the Related Contacts...
            system.debug('Number of Contacts...: '+ acc.Contacts.size());

            for(Contact con : acc.Contacts)
            {
               system.debug('Contact Record is...: '+ con);
            }

            // Get the Related Cases..
            system.debug('Number of Related Cases...: '+ acc.Cases.Size());
            for(Case cs : acc.Cases)
            {
               system.debug('Case Record is..: '+ cs);
            }

            // Get the Related Opportunities..
            system.debug('Number of Opportunities....: '+ acc.opportunities.size());
            for(Opportunity oppty : acc.Opportunities)
            {
               system.debug('Opportunity Record is..: '+ oppty);
            }
          }
        }
      }
    }
}
```

Execution:

```
DatabaseUtility.GetParentAndChildRecordsByName('Edge Communications');
```

Child To Parent (Standard Relationship):

By using Relationship Queries, we can fetch the Data from Child To Parent also. (i.e. Bottom to Top approach)

i.e. Upon fetching the Child Record information, we can fetch the associated Parent Record details, and the associated Grand Parent record details, etc.

By using Child To Parent Query, we can traverse the Objects from Child to the associated parent objects.

```
Grand Grand Parent
        ^
        |
        Grand Parent
                ^
                |
```

-- Parent Object
^
|
-- Child Object

Note: We can traverse the objects from Bottom to Top upto "5 Levels"

Upon fetching the Parent Record details, we need to Query the Parent Object fields with the below syntax.

Syntax:

<ParentObjectName>.<FieldName>

Ex:

Account.Name
Account.Rating
Account.Active__c
Account.Pan_Number__C

We can fetch the Grand Parent details as below.

Syntax: <ParentObjectName>.<GrandParentObjectName>.FieldName

Ex:

Account.Owner.FirstName
Account.Owner.Email
Account.Owner.UserName

Note: Upon fetching the Data, we can filter the records by adding the conditions based on any of the object fields. (i.e. On Child Object, Parent Object, Grand Parent Object)

Ex:

Where Account.Active__c == 'Yes'
Where Account.Owner.Email = 'sample@gmail.com'

/* Write an apex program, to fetch the Case Record based on the Case Number and get the associated Contact Details and the related Account and Owner Details along with the "Owners Profile Information".*/

Class Code:

```
public class DatabaseUtility
{
   Public static void GetCaseAndRelatedParentDetails(string csNumber)
   {
      if(csNumber != null && csNumber != '')
      {
         List<Case> lstCases = [Select id, caseNumber, priority, status,
```

```
                origin, Contact.FirstName, contact.Email,
                contact.Account.Name, contact.Account.Industry,
                contact.Account.Owner.Email,
                        contact.Account.Owner.Profile.Name
            from Case
                where caseNumber =: csNumber];

        system.debug('Collection size...: '+ lstCases.size());

        if(! lstCases.isEmpty())
        {
           for(Case cs : lstCases)
           {
              system.debug('Case Record is...: '+ cs);
              system.debug('Contact Email Id ...: '+ cs.Contact.Email);
              system.debug('Account Name...: '+ cs.Contact.Account.Name);
              system.debug('Owner Details....: '+ cs.Contact.Account.Owner.Email);
              system.debug('Profile Name....: '+ cs.Contact.Account.Owner.Profile.Name);

           }
        }
     }
  }
}
```

Execution:

```
DatabaseUtility.GetCaseAndRelatedParentDetails('00001016');
```

Parent To Child (Custom Relationship):

By using this approach, we can fetch the data from Parent objects and from all the associated child objects which having the custom association.

Note: This can be implemented with the help of "Sub Queries". (i.e. We need to write the child object query as a Sub Query).

Inside the SubQuery, the child object name should be specified with "Plural Format" and by post-fixing with "__r". ("__r" represents the Custom Relationship)

/* Write an apex program, to fetch all the Hiring Manager Records and the associated Position records.*/

Class Code:

```
public class DatabaseUtility
{
   Public static void GetHRRecordsWithPositions()
   {
```

```
        List<Hiring_Manager__C> lstHrs = [Select id, name, location_del__C,
                                contact_number__C, email_id__c,
                                (Select id, name, location_name__c, position_status__c, milestone_date__C,
maximum_budget__C from Positions__r)
                                from Hiring_Manager__c];

    system.debug('Collection size is...: '+ lstHrs.size());

    if(! lstHrs.isEmpty())
    {
      for(Hiring_Manager__C hr : lstHrs)
      {
        system.debug('Hiring Manager...: '+ hr);

        system.debug('Related Positions Count..:  '+ hr.Positions__r.size());

        for(Position__C pos : hr.Positions__R)
        {
          system.debug('Position Record...: '+ pos);
        }
       system.debug('---------------------------------------');
      }
    }
  }
}
```

Execution:

```
        DatabaseUtility.GetHRRecordsWithPositions();
```

DML OPERATIONS THROUGH APEX

DML Stands for Data Manipulation Language.

Salesforce provides a set of Standard Objects, and the user can create one or more Custom objects based on the application need, to store the application specific / organization specific data.

All the Salesforce objects will get resides inside the "Database.com". And all the records will get stored inside the objects.

By using Page Layout, we can perform the operations on Only One record at a time.

By using Apex Programming, we can perform the operations on either one or more records at a time.

Apex provides the below ways to perform the DML operations.

1. **By using DML Statements**
2. **By using Database Class Methods.**

By using DML Statements, we can interact with the SObjects through programatically. As the DML statements are tightly integrated with the "Database.com".

Apex provides the below DML Statements

1. **INSERT**
2. **DELETE**
3. **UNDELETE**
4. **UPDATE**
5. **UPSERT (INSERT + UPDATE)**
6. **MERGE**

Governor Limits:

1. We can have max. of 150 DML statements inside a Transaction.
If the user tries to perform more than 150 DML statements in a transaction, salesforce server will raise "**System.Limit Exception. Too many DML Statements: 151**" exception.

2. By using Each DML Statement, we can process max. of **10,000** records at a time.

Best Practices:

1. Upon performing the DML operations, we have to make the operation as Bulkify.
(i.e. We need to add all the records to the Collection. And perform the DML operations on the Collection of records at a time).

2. It is always recommended to avoid the usage of DML statements, inside the "FOR Loop".

Note:
All the DML Statements are "Atomic".
(i.e. Upon performing the operations on the collection of records. If any of the operation fails, then it will Rollback the Whole Transaction. Because, each DML statement, will use the "Implicit Transaction" by default.)

INSERT Statement:

By using this Statement, we can insert either One / More records into the specified object.
It supports to insert the records into both "Standard and Custom" objects.

Inserting One Record:

Syntax: Insert <objectName>;

Inserting Multiple Records:

Syntax: Insert <collectionObjectName>;

/* Write an apex Class, to Insert specified number of records into the Contact object. */

Class Code:

```
public class DMLUtility
{
Public static void InsertBulkContacts(integer maxRecords)
  {
    if(maxRecords > 0)
    {
      List<Contact> lstContacts = new List<Contact>();

      for(integer counter = 1; counter <= maxRecords; counter++)
      {
        // Create the Contact Records...
        Contact con = new Contact();

        con.FirstName = 'Bulk';
        con.LastName = 'Contacts - '+ counter;
        con.Email = 'bulkContact' + counter + '@gmail.com';
        con.Title = 'Sales Manager';
        con.Department = 'Sales Department';
        con.Phone = '9900990000';
        con.Fax = '9900990000';
        con.MailingStreet = 'Gayathri Nagar';

        if(counter != 198)
        {
              con.Pan_Number__C = 'ALPPG9090E';
```

```
            }

            // Add the Record to Collection..
            lstContacts.Add(con);
          }

          if(! lstContacts.isEmpty())
          {
            insert lstContacts;
          }
        }
    }
}
```

Execution:

```
// Invoke the Method...
        DMLUtility.InsertBulkContacts(200);
```

Database Class:

Upon performing the DML operations, by using DML Statements (like Insert , Update, Delete, Undelete, etc), If any of the operation has been failed. Then it will Rollback the whole Transaction. (i.e. Data will not be commited to the Database)

And we don't have any such options in DML statements to allow the "Partial Processing".

To avoid these problems, Apex provides "Database" Class, to perform all the DML operations through the "Readymade Methods".

By using Database Class methods, we can perform all the DML operations on One or More records in the object.

Benefits:

1. Database Class methods allows us to perform "Partial Processing".
(i.e. Upon prcessing the records, if any of the record fails. Then it will continue the operations with the rest of the records)

2. It provides a Tracking mechanism, through which we can track the Failed records, and the reason for the failure.

Database.Insert() Method:

This method is used to insert One / More records into the associated object.

Syntax: Database.Insert(<CollectionName>):

Note: This method will act as the "Insert Statement". Which maintains the Implicit Transaction by default.

Database.SaveResult[] <object> = Database.Insert(<CollectionName>, Boolean AllORNothing):

--> This method allows us to perform the Partial Processing.

1. CollectionName: Describes the name of the collection, which needs to be get inserted.

2. Boolean AllORNothing: It is used to allow the Partial Processing.

TRUE --> It will maintains the Transactions. Which is similar like as "Insert Statement".
FALSE --> It allows the Partial Processing.

Upon processing the Records by using "Database.Insert()" method. To track the each record result, Apex provides a class called as "Database.SaveResult" class.

Database.SaveResult Class:

This class contains the result of one record insertion. It provides the below methods to track the result.

1. Boolean IsSuccess():

This method is used to track, that the Insertion operation is Success or not. It returns TRUE, if the operation successfull. Else it returns FALSE.

2. **ID GetID():**

This method is used to Get the "Record ID", when the operation is successfull.

3. **GetErrors():**

This method is used to get the "Error Messages" has been raised upon failing the records.
It returns the Error Messages in terms of a Class called as "Database.Error".

Syntax:

Database.Error[] errors = <objectName>.GetErrors();

Database.Error Class:

This class is used to track the error messages has been occured upon processing the records.

We can get each record Failure Messages by using the below methods.

1. GetMessage():

This method will returns the "Error Message" has been occured, upon failing the record.

2**. GetStatusCode():**

This method will returns the Status Code of the Error Message.

Ex: REQUIRED_FIELD_MISSING, DUPLICATE_DETECTED, etc.

3. GetFields():

This method will returns the list of fields, which causes the error.

/*

Write an apex program, to Insert Bulk Contact Records, by allowing the Partial Processing. And Track the Record Results and Failure Messages.

***/**

Class Code:

```
public class DMLUtility
{
   Public static void InsertBulkContacts(integer maxRecords)
   {
      if(maxRecords > 0)
      {
         List<Contact> lstContacts = new List<Contact>();

         for(integer counter = 1; counter <= maxRecords; counter++)
         {
            // Create the Contact Records...
            Contact con = new Contact();

            con.FirstName = 'Bulk';
            con.LastName = 'Insert - '+ counter;
            con.Email = 'bulkContact' + counter + '@gmail.com';
            con.Title = 'Sales Manager';
            con.Department = 'Sales Department';
            con.Phone = '9900990000';
            con.Fax = '9900990000';
            con.MailingStreet = 'Gayathri Nagar';

            if(counter != 198)
            {
                  con.Pan_Number__C = 'ALPPG9090E';
            }

            // Add the Record to Collection..
            lstContacts.Add(con);
         }

         if(! lstContacts.isEmpty())
         {
            Database.SaveResult[] results = Database.Insert(lstContacts, false);
```

```
        for(Database.SaveResult res : results)
         {
            if(res.isSuccess())
            {
               system.debug('Record has been inserted Successfully. Record Id is...: '+ res.getID());
            }
            else
            {
                                                    //Database.Error[] errors = res.GetErrors();
               for(Database.Error err : res.GetErrors())
               {
                  system.debug('Error Message is...: '+ err.GetMessage());
                  system.debug('Get Status Code ....: '+ err.GetStatusCode());
                  system.debug('Effected Fields are....: '+ err.GetFields());
               }
            }
         }

      }
   }
}
```

Execution:

```
// Invoke the Method...
DMLUtility.InsertBulkContacts(200);
```

DELETE Statement:

By using this Statement, we can remove either One / More records from the object. Upon deleting the record, we need the "Record ID".

Deleted Records will get resides inside the "ReCycle Bin" for 15 Days. The user can re-store the deleted records back to the actual object.

Remove One Record:

```
Delete <objectName>;
```

Remove Multiple Records:

```
Delete <collectionObjectName>;
```

Database Class:

```
Database.DeleteResult[] results =
        Database.Delete(<collectionObjectName>, Boolean AllORNothing);
```

/*
Write an Apex Program, to delete the Contact Records based on the specified "Starting Characters".
***/**

Class Code:

```
public class DMLUtility
{
   Public static void RemoveContactsByStartingCharacters(string startingChar)
   {
      /*
      if(startingChar != null && startingChar != '')
      {
         startingChar += '%';
         List<Contact> contactsToRemove = [select id, lastname
                                           from Contact
                                           where lastname like : startingChar];
         if(! contactsToRemove.isEmpty())
         {
            Delete contactsToRemove;

            //Database.DeleteResult[] results = Database.delete(contactsToRemove, false);
         }

      }
                  */

      startingChar += '%';

      Delete[select id, lastname from Contact where lastname like : startingChar];
   }
}
```

Execution:

```
// Invoke the Method...
DMLUtility.RemoveContactsByStartingCharacters('Contact');
```

UNDELETE Statement:

This statement is used to Re-Store the deleted records back to the actual object. We can fetch the Deleted Record from the object by using "IsDeleted" field.

Syntax:

Re-Store Only One Record:

```
Undelete <objectName>;
```

Re-Store Multiple Records:

Undelete <collectionObjectName>;

By using Database Class:

Database.UndeleteResult[] results =
Database.Undelete(<collectionObjectName>, Boolean);

/*
Write an apex program, to Re-Store all the Contacts to the actual object. Whose last name starts with "Contact".
***/**

Class Code:

```
Public static void ReStoreContctsByStatingCharacters(string startingChar)
  {
    if(startingChar != null && startingChar != '')
    {
      startingChar += '%';

      Undelete [Select id, lastname from Contact
                Where lastname like : startingChar and isDeleted = true
                ALL ROWS];
    }
  }
}
```

Execution:

```
// Invoke the Method...
DMLUtility.ReStoreContctsByStatingCharacters('Contact');
```

Assignments:

1. Write an Apex Program, to insert 450 Lead Records into the Lead Object by allowing the Partial Processing.

2. Write an apex program, to Delete the Contact Records created in the Last Week.
   (Where createddate = LAST_WEEK)

3. Write an apex program, to remove all the Contacts which are not associated with any Account Record.

4. Write an apex program, Remove all the Hiring Managers, Whose email id starts with 'test'.

5. Write an apex program, to re-store the Hiring Manager Records, whose Location is associated with "Hyderabad".

UPDATE Statement:

By using this statement, we can update one or more records into the specified object.

Syntax:

Updating Only One Record:

```
Update <objectName>;
```

Updating Multiple Records:

```
Update <collectionObjectName>;
```

By using Database Class Method:

```
Database.SaveResult[] results =
        Database.Update(<CollectionName>, Boolean AllORNothing);
```

Upon updating the records, we have to use the below steps.

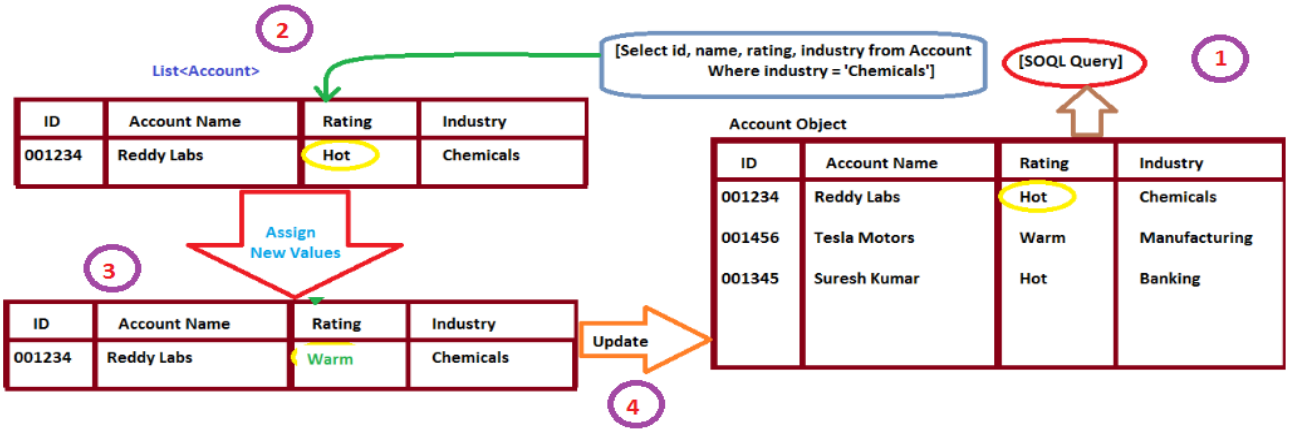
Step 1: Get the Required records to update by using SOQL Query

```
List<Account> lstAccounts = [select id, name, rating, industry
                             from Account where rating = 'Hot'];
```

Step 2: Assign the New values to the records and Commit the changes back to the Database.

```
if(! lstAccounts.isEmpty())
{
    for(Account acc : lstAccounts)
    {
        acc.industry = 'Finance';
    }

    update lstAccounts;
}
```



/* Write an Apex program, to update the AnnualRevenue as "20,00,000" and the Rating as "Hot" for All the Banking Accounts. */

Class Code:

```
public class DMLUtility
{
   Public static void UpdateBankingAccountDetails()
   {
      List<Account> accountsToUpdate = new List<Account>();

      for(Account acc : [Select id, name, rating, annualrevenue, industry
                  from Account
                        Where industry = 'Banking'])
      {
         acc.Rating = 'Hot';
         acc.AnnualRevenue = 2000000;

         accountsToUpdate.Add(acc);
      }

      if(! accountsToUpdate.isEmpty())
      {
         update accountsToUpdate;
      }

   }
}
```

Execution:

```
// Invoke the Method...
DMLUtility.UpdateBankingAccountDetails();
```

```
/*
      Write an apex program, to De-Activate the Specified User based on the UserName.
*/
```

Class Code:

```
public class DMLUtility
{
   Public static void DeActivateUser(string uName)
   {
      if( uName != null && uName != '')
      {
          User userToDeactivate = [select id, username, isActive
                        from User
```

```
                                Where username =: uName
                                    Limit 1];
        if(userToDeactivate != null)
        {
           userToDeactivate.isactive = false;

           update userToDeactivate;
        }
     }
  }
}
```

Execution:

```
// Invoke the Method...
DMLUtility.DeActivateUser('mgruser.2017@gmail.com');
```

Assignments:

1. Write an apex program, to assign all the Unassociated contacts to "Edge Communications" account.

2. Write an apex program, to assign the Lead Status as "Closed - Converted", which are having the annualrevenue > 20,00,000.

3. Write an apex program, to assign the Ownership of the Accounts to "Manager User". Which are associated with "Banking" industry.

/* Write an apex program up synchronize the Contact record's Phone and Fax Numbers, with the associated Account Record's Phone and Fax values. */

Class Code:

```
public class DMLUtility
{

   Public static void SynchronizeContactDetails()
   {
      // Get All the Contacts, associated with the Accounts...
        List<Contact> lstContacts = [Select id, accountId, phone, fax, pan_Number__c
                                from Contact
                                    Where accountId != null];


      Set<ID> accountIDsSet = new Set<ID>();

      // Get All the associated Account Record Id's..
      if(! lstContacts.isEmpty())
      {
```

```
  for(Contact con : lstContacts)
{
   accountIDsSet.Add(con.AccountId);
}

// Get the associated Account Records...
Map<ID, Account> accountsMap = new Map<Id,Account>([Select id, name,
                                    phone, fax
                                    from Account
                                    Where id IN : accountIdsSet]);


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

// Assign the New values (Phone and Fax)
for(Contact con : lstContacts)
{
   /*
            con.Phone = accountsMap.Get(con.AccountId).Phone;
            con.Fax = accountsMap.Get(con.AccountId).fax;

    */

  Account acc = accountsMap.Get(con.AccountId);
            con.Phone = acc.Phone;
   con.Fax = acc.Fax;
   con.PAN_Number__c = 'ALHUH3456E';

   contactsToUpdate.Add(con);
  }
  // Commit the changes back to the database...
  if(! contactsToUpdate.isEmpty())
  {
   Database.update(contactsToUpdate);
  }
 }
}
```

Execution:

```
// Invoke the Method...
DMLUtility.SynchronizeContactDetails();
```

UPSERT Statement:

Upsert is the combination of both "Update + Insert". i.e. We can perform both insert and update operations in an object by using a single DML Statement (Upsert).

Note: This statement will Update the records into the object, which are holding the Record Id. And it will insert the records, which are not having the "Record ID".

Syntax:

Upserting Single Record:

Upsert <objectName>;

Upserting Multiple Records:

Upsert <collectionObjectName>;

By using Database Class Method:

Database.UpsertResult[] results =
Database.Upsert(<collectionObjectName>, Boolean);

Method: **Boolean ISCreated():**
It returns TRUE, if the upsert method creates a New record into the object.
Else it returns FALSE(i.e. Record is updated)

/*
Write an apex program, to Change all New Case Priorities as "High" and Create a New Case Record also in the object.
***/**

Class Code:

```
public class DMLUtility
{
  Public static void UpsertCaseRecords()
   {
     // Prepare a Collection to Hold All Case Records..
     List<Case> casesToUpsert = new List<Case>();

     // Get All the New Case Records and update priority as "High".
     for(Case newCase : [Select id, status, priority
                                 from Case Where status = 'New'])
     {
        newCase.Priority = 'High';

        casesToUpsert.Add(newCase);
     }
```

```
// Create a New Case Record...
        Case cs = new Case();

                cs.Status = 'New';
                cs.Priority = 'High';
                cs.Origin = 'Phone';
                cs.Type = 'Mechanical';
                cs.Reason = 'Perfomance';
                cs.Subject = 'My Printer is Not Working as Expected.';
                cs.Description = 'Dear Customer Support, My Newly Purchansed Printer is unable to Print the
Documements. Please Resolve the issue ASAP.';

        // Add the record to the collection..
                casesToUpsert.Add(cs);

    // Upsert the records..
    if(! casesToUpsert.isEmpty())
    {
        Database.UpsertResult[] results = Database.Upsert(casesToUpsert, false);

      for(Database.UpsertResult res  : results )
      {
        if(res.isSuccess())
        {
          if(res.isCreated())
          {
                system.debug('Record has been inserted Successfully.');
                system.debug('Hiring Manager Record ID is...: '+ res.getId());
          }
          else
          {
            system.debug('Record Updated Successfully with the ID ..: '+ res.getId());
          }
        }
        else
        {
          for(Database.Error err : res.getErrors())
          {
                        system.debug('Error Message is...: '+ err.getMessage());
            system.debug('Effected Fields are...: '+ err.getFields());
            system.debug('Error Status Code is....: '+ err.Getstatuscode());
          }
        }
        system.debug('------------------------------------------');
      }
```

```
    }
  }
}
```

Execution:

```
// Invoking the Method..
        DMLUtility.UpsertCaseRecords();
```

MERGE Statement:

This Statement is used to Merge / Combine the multiple records into a single record.

Note: Merge Statement will allows us to combine max. of 3 records into a single record. Once the records has been merged, then the remaining records will be removed from the object permanently.

Note: Merge statement will allows us to merge the records available in either "Account / Contact / Lead" object.

Syntax:

```
        Merge <Target / Parent Record> <Source / Child Record>;

        Merge <Target Record> List<Source Records>;
```

Note:

Upon merging the records, if the Child record contains any associated record. Then the associated records will be get transfered to the Target records. (i.e. Related Records also will get merged.)

```
/*
        Write an apex class, to merge the Two Account Records into a single one.
*/
```

Class Code:

```
public class DMLUtility
{
   Public static void MergeAccountRecords()
   {
        Account acc1 = [Select id, name, rating, industry from Account
                        Where  name like '%250'
                        LIMIT 1];

        Account acc2 = [Select id, name, rating, industry from Account
                          Where name like '%249'
                          LIMIT 1];

     system.debug('Account 250 Record is...: '+ acc1);
     system.debug('Account 249 Record is...: '+ acc2);
```

```
        if(acc1 != null && acc2 != null)
        {
           Merge acc1 acc2;
        }
     }
}
```

Execution:

```
// Invoke the Method...
DMLUtility.MergeAccountRecords();
```

SALESFORCE OBJECT SEARCH LANGUAGE (SOSL Queries)

By using SOQL Queries, we can search for the specific content in Only One Object at a time, by adding the required conditions with the help of "WHERE" clause.

To search for the keyword, in multiple objects, we need to write multiple SOQL Queries. Which causes the "Too Many SOQL Queries: 101" Exception.

To avoid this problem, salesforce provides a Search Engine called as "SOSL Queries".

SOSL --> Salesforce Object Search Language.

By using SOSL Queries, we can search for a specific content in multiple objects in the salesforce Database.

It will search for the content in all fields inside the object. If any of the Object record, maching the specified search text criteria, then it will return the records to the user.

Note: SOSL Queries should be prepare with the help of "FIND" keyword.

Syntax:

```
FIND 'searchtext*' IN ALL FIELDS RETURNING
        <ObjectName1>(<Colum1>, <Colum2>,<Colum3> ,.....<ColumN>),
        <ObjectName2>(<Colum1>, <Colum2>,<Colum3> ,.....<ColumN>),
        ...
        ...
        <ObjectName20>(<Colum1>, <Colum2>,<Colum3> ,.....<ColumN>)
```

Ex:

```
FIND 'test*' IN ALL FIELDS RETURNING
        Account(id, name, rating, industry, active__c),
        Lead(id, firstname, lastname, title, city, state),
        Position__C(id, name, Location__c, Position_status__C),
        Contact(Id, firstname, lastname, email, phone, fax),
        Candidate__C (id, name, location__c, notice_period__C)
```

Governor Limits:

1. A transaction can have max. of 20 SOSL Queries.

2. Each SOSL Query can return max. of 2,000 records as the result.

3. Each SOSL Query can search for the records in max. of 20 Objects.

Note: We can't perform the "DML Operations" on the SOSL Query results.

Salesforce provides 2 types of SOSL Queries.

1. Static SOSL Queries:

All the Static SOSL Queries should be enclosed inside the square braces "[ ]".

We can't add the "Search Text" in the SOSL Query dynamically at runtime. It should be hardcoded upon writing the query.

Ex: [SOSL Query];

Static SOSL Queries will be executed automatically by default.

SOSL Queries will returns the results in the form of "List<List<SObject>>".

Syntax:

List<List<SObject>> results = [SOSL Queries];

2. Dynamic SOSL Queries.

By using this approach, we can include the searchtext dynamically at runtime inside the SOSL Query.

Each Dynamic SOSL Query should be get prepared and stored inside a string variable.

Ex:

string searchQuery = ' SOSL Query ';

We need execute the SOSL Query manually with the help of "Search.Query()" method.

Dynamic SOSL Query will returns the result in the form of "List<List<SObject>>".

Ex:

List<List<SObject>> lstResults = Search.Query(searchQuery);

Note: Upon searching for the keyword in multiple objects, we can specify the Search Group. Which indicates, in what type of fields the keyword has to be get searched.

Options:

IN ALL FIELDS:

It will search in All the Fields in the Object.

IN NAME FIELDS:

It will search for the Keyword only in "Name Fields".

IN EMAIL FIELDS:

It will search for the Keyword only in "Email Fields".

IN PHONE FIELDS:

It will search for the Keyword, only in "Phone fields".

Ways to invoke the SOSL Queries:

We can invoke the SOSL Queries with the below options.

1. By using "Query Editor" from Developer Console.
2. By using Apex Programming.

/* Prepare an SOSL Query, to search for the "test" keyword in Account and Lead Objects, by using Query Editor. */

```
find {test} IN ALL FIELDS RETURNING
        Account(id, name, rating, industry, type),
        Lead(id, firstname, lastname, email,company, status)
```

/*

Write an SOSL Query, to search for the keyword "apex" in Account, Lead, User and Position Object.

***/**

```
find {apex} IN ALL FIELDS RETURNING
        Account(id, name, rating, industry, type),
        Lead(id, firstname, lastname, email,company, status),
        User (id, firstname, lastname, username, email, title),
        Position__C(id, name, Location__C, position_status__C, close_date__c)
```

Note: Upon searching for the records inside the object by using "SOSL", the "Allow Search" checkbox should be selected in the Object level.

/*

Write an apex program to search for the specific content in the objects by using static SOSL Query.

***/**

Class Code:

```
public class GlobalSearchUtility
{
  Public static void SearchRecords()
  {
        List<List<SObject>> lstResults = [FIND 'test*' IN ALL FIELDS RETURNING
                Account(ID, name, rating, industry, annualrevenue, phone, fax, active__C),
                Contact(Id, firstname, lastname, email, title, phone, fax),
                Lead(id, firstname, lastname, title, status, company, phone, fax),
                Position__C(id, name, location__c, position_status__C,maximum_budget__C),
                Candidate__C(id, name, email_id__C, notice_period__C, contact_number__C)];

    if(! lstResults.isEmpty())
```

```
    {
        List<Account> lstAccounts = (List<Account>) lstREsults.get(0);
        List<Contact> lstContacts =         (List<Contact>) lstResults.Get(1);
        List<Lead> lstLeads = (List<Lead>) lstResults.get(2);
        List<Position__C> lstPos =          (List<Position__C>)        lstrEsults.Get(3);
        List<Candidate__C> lstCands = (List<Candidate__C>) lstResults.Get(4);

        system.debug('Accounts Records Collection Size..: '+ lstAccounts.size());
        for(Account  acc : lstAccounts)
         {
            system.debug('Account Record is...: '+ acc);
         }
        system.debug('----------------------------------------');

        system.debug('Contact Records Collection size....: '+ lstContacts.size());
        for(Contact con : lstContacts)
         {
            system.debug('Contact Record is...: ' + con);
         }
        system.debug('----------------------------------------');

        system.debug('Lead Records Collection Size is...: ' + lstLeads.size());
        for(Lead ld : lstLeads)
         {
            system.debug('Lead Record is...: '+ ld);
         }
        system.debug('----------------------------------------');

        system.debug('Position Records collection size is....: '+ lstPos.size());
        for(Position__C pos : lstPos)
         {
            system.debug('Position Record is...: '+ pos);
         }
        system.debug('----------------------------------------');

        system.debug('Candidate Record Collection Size is...: '+ lstCands.size());
        for(Candidate__C cnd : lstCands)
         {
            system.debug('Candidate Record is....: '+ cnd);
         }
    }
  }
}
```

Execution:

```
// Invoking the Procedure..
        GlobalSearchUtility.SearchRecords();
```

/*

Write an apex program to search for the Specific Search Text in multiple object by using Dynamic SOSL Queries. Pass the Search Text to the method at runtime as input.

***/**

Class Code:

```
public class GlobalSearchUtility
{
  // Dynamic SOSL Example..
   Public static void DynamicSearchByText(string searchText)
   {
      string searchQuery = 'FIND '+ '\'' + searchText + '*'+ '\''+ ' IN ALL FIELDS RETURNING '+
                                          'Account (Id, name, rating, industry, type, active__C),' +
                                          'Lead(id, firstname, lastname, email, company, title, status),'+
                                          'User(id, firstname, lastname, title, email, username, isactive),'+
                                          'Candidate__c(id, name, location__C, notice_period__C),'+
                                    'Position__C(id, name, location__C, close_date__C, position_status__c)';

      if(searchQuery != null)
      {
         List<List<SObject>> searchResults =        Search.query(searchQuery);

         if(! searchResults.isEmpty())
         {
            // Get Each Element from the Collection, by using index position.
            // The collection will return the results in the form of "List<SObject>"
            // We need to do the Explicit TypeCasting to convert the SObject into our own type.

            List<Account> lstAccounts = (List<Account>) searchResults.get(0);
            List<Lead> lstLeads = (List<Lead>) searchResults.get(1);
            List<User> lstUsers = (List<USer>) searchResults.get(2);
                  List<Candidate__C> lstCands = (List<Candidate__C>) searchResults.get(3);
            List<Position__c> lstPositions = (List<Position__c>) Searchresults.get(4);

            system.debug('Accounts Collection Size..: '+ lstAccounts.size());
            for(Account acc : lstAccounts)
            {
               system.debug('Account Record...: '+ acc);
            }
            system.debug('---------------------------------');
```

```
                system.debug('Lead Collection Size....: '+ lstLeads.size());
                for(Lead ld : lstLeads)
                {
                   system.debug('Lead Record is...: '+ ld);
                }
                system.debug('-----------------------------------');

                system.debug('User Records Collection Size...: '+ lstUSers.size());
                for(User u : lstUsers)
                {
                   system.debug('User Record is...: '+ u);
                }
                system.debug('-----------------------------------');

                system.debug('Position Records Collection Size...: '+ lstPositions.size());
                for(Position__C pos : lstPositions)
                {
                   system.debug('Position Record ...: '+ pos);
                }
                    system.debug('-----------------------------------');

                    system.debug('Candidate Results Count....: '+ lstCands.size());
                    for(Candidate__C cnd : lstCands)
                {
                   system.debug('Candidate Record...: '+ cnd);
                }
            }
        }
    }
}
```

Execution:

```
// Invoke the Method to search...
        GlobalSearchUtility.DynamicSearchByText('test');
```

EMAIL PROGRAMMING

There may be situations in which you would not want a workflow email alert, because either your criteria to send an email may be too complex or the person whom you want to send an email has to be determined dynamically.

In such situations you can choose to send an email through an Apex Class.

Salesforce allows us to send the email notifications to the users or external systems / to receive the emails from the users.

While sending/ Receiving the emails, we can add the Attachments also.

We have 2 Types of Email Services in Salesforce

1. **Inbound Email Messaging**
2. **Outbound Email Messaging**

Note: All Email Classes are available in the "Messaging" name space.

Governor Limits:

In s single transaction, we can have max. of 10 email invocations. i.e Maximum of 10 Messaging.SendEmail() methods per Transaction.

Inbound Emails:

This service is used to receive the Emails from the users or external systems. i.e Email is coming into our Salesforce instance.

It provides an interface called as "**Messaging.InboundEmailHandler**" in-order to receive the email from the user and store the email content in the associated object along with the attachment.

Outbound Emails:

Used to send the Emails to the users based on the ToAddress and with the required Subject and the Content.

We can able to send the Email Content in PlainText or HTML format.

We have 2 ways to send the Outbound Emails.

1. Messaging.SingleEmailMessage Class
2. Messaging.MassEmailMessage Class

Messaging.SingleEmailMessage Class:

By using this class, we can send the email notification to one or more recipients dynamically at runtime, with the required subject and content.

Note: Upon sending the email notification, we can add one or more attachments also.

It provides a set of "Instance Methods", which can be used to send the email notifications.

Syntax: Messaging.SingleEmailMessage emailObject = new Messaging.SingleEmailMessage();

Methods:

1. **SetToAddresses(<Collection>):**

This method is used to specify the Collection of email addresses, to send the email notification. We can pass the recipients email addresses as an "Array" or as a "Collection".

Note: We can add max.of 100 reciepient emailid's.

2. **SetCCAddresses(<Collection>):**

This method is used to specify the "CC Addresses", to whom we need to send the email copy.

Note: We can add max. of 25 CC Email Addresses.

3. **SetBCCAddresses(<Collection>):**

This method is used to specify the "BCC Addresses", to whome we need to send the email copy.

Note: We can add max. of 25 BCC Email Addresses.

4. **SetReplyTo(string replyToEmailId):**

By using this method, we can specify the replyTo email address, to whom the email reply can send.

5. **SetSenderDisplayName(string SenderName):**

This method is used to specify the Sender Display Name in the email notification.

Ex: Tesla Support Center, Tesla Marketing Team, Tesla Recovery Department, etc.

6. **SetSubject(string emailSubject):**

This method is used to specify the email subject.

7. **SetPlainTextBody(string emailContent):**

By using this method, we can send the PlainText Email content to the reciepients.

8. **SetHTMLBody(string htmlEmailContent):**

This method is used to send the HTML Email Content to the reciepients.

Sending Email Notification:

To send the email notification, salesforce provides a Method called as "Messaging.SendEmail()".

Ex: Messaging.SendEmail(new Messaging.SingleEmailMessage[]{emailObject});

Messaging.SendEmail(List<Messaging.SingleEmailMessage>);

Usecase: Write an apex program to send an email notification to the Candidate, upon creating a New Candidate Record.

Class Code:

```
public class CandidateHandler
{
   Public static void SendEmailToCandidates(List<Candidate__c> lstCandidates)
   {
      if(! lstCandidates.isEmpty())
      {
         List<Messaging.SingleEmailMessage> lstEmailsToCandidate = new List<Messaging.SingleEmailMessage>();

         for(Candidate__c cnd : lstCandidates)
         {
            // Prepare the Email Notification...
            Messaging.SingleEmailMessage email = new Messaging.SingleEmailMessage();

            string[] toEmailIDs = new string[]{cnd.Email_ID__c, 'ferozjani@gmail.com'};
            email.setToAddresses(toEmailIDs);

            /*
            string[] ccEmailIds = new string[]{cnd.Email_ID__c, 'feroz.crm@gmail.com'};
             email.SetCCAddresses(ccEmailIDs);

            string[] bccEmailIDs = new string[]{cnd.Email_ID__c, 'ferozjani@gmail.com'};
             email.SetBCCAddresses(bccEmailIDs);
            */

            email.SetSenderDisplayName('Candidate Record Creation Notification Alert');

            email.SetReplyTo('ferozjani@gmail.com');

            string emailSubject = 'Congratulations  '+ cnd.Name + '...!! Your Candidature has been successfully
submitted for the position.';
            email.SetSubject(emailSubject);

            string emailHTMLContent = 'Dear '+ cnd.Name + ', <br/> <br/> '+
                        'Thanks for showing interest in our organization positions. <br/><br/> '+
                        'We are pleased to inform you, that your candidature has been successfully submitted for
the
                        position.' +cnd.PositionName__c +
                                    '<br/><br/> Here are your Details... <br/><br/>' +
                                    'Candidature Reference ID .....: '+ cnd.Candidate_ID__c +
                                    '<br/> Candidature Status ....: '+ cnd.Candidature_Status__c +
                                    '<br/> Location .....: '+ cnd.Location__c +
                                    '<br/> Current CTC .....: '+ cnd.Current_CTC__c +
                                    '<br/> Expected CTC .....: '+ cnd.Expected_CTC__c +
                                    '<br/> Email Address .....: '+ cnd.Email_ID__c +
                                    '<br/> Contact Number .....: '+ cnd.Contact_Number__c+
                                    '<br/> <br/> Please contact on the below address if any changes or queries
<br/><br/>'+
                                    'Thanks & Regards <br/> Resource Management Team (RMG Team) <br/> Dell Inc.
                                    <br/> Hyderabad.' ;
            email.SetHTMLBody(emailHTMLContent);

            lstEmailsToCandidate.Add(email);
```

```
        }

        // send the email notifications...
        if(! lstEmailsToCandidate.isEmpty())
        {
           Messaging.sendEmail(lstEmailsToCandidate);
        }
      }
    }
}
```

Execution:

```
// Calling the method…

CandidateHandler. SendEmailToCandidates(<Candidates Collection>);
```

Messaging.MassEmailMessage Class:

By using this class, we can send the email notifications to one or more users by using the user defined subject and content or by using an email template.

We can send the Email notifications to the users by using the Record ID's.

Note: We can send the email notification to the Lead / Contact / User record id's (Who Id's).

It provides a set of instance methods as below.

Syntax: Messaging.MassEmailMessage massEmail = new Messaging.MassEmailMessage();

1. **SetTargetObjectIDs(List<ID>):**

This method is used to specify the List of "Who Id's", to whome we need to send the Email notification.

Note: We can add max. of 250 record id's inside this method at a time.

2. **SetTemplateId(EmailTemplateID):**

This method is used to specify the Email TemplateId, to be used to send email notifications.

Sending Email Notification:

Ex: Messaging.SendEmail(new Messaging.MassEmailMessage[] {<massEmailobjectName>});

/*

Write an apex program, to send the Email Notifications to All Contact People, whose records are not updated with their Account Details.

*/

Email Template:

Subject:

Reminder ...: Dear {!Contact.FirstName} {!Contact.LastName}..!! Update Your Account Details

Content:

Dear {!Contact.FirstName} {!Contact.LastName},

Thanks for purchasing the Products from our organization.

As we have observed in our Records, that your Contact details are not Updated with your Account Details.

Please contact on the below address, make sure your contact records should be associated with your respective Account Record.

Please contact on the below address.

Thanks & Regards,
{!User.FirstName} {!User.LastName},
Customer Support Team,
Microsoft Inc.,
Tolle Free : +1 (800) 898-4567
Email Id: customerSupport@Microsoft.com
Visit us : ww.Microsoft.com

Class Code:

```
public class MessagingUtility
{
   Public static void SendReminderNotificationToContacts()
  {
     Map<id,Contact> mapContacts = new Map<Id,Contact> ([select id, firstName, LastName, email, accountid
                                  from Contact  Where accountid = null and email != null Limit 10]);

     List<ID> contactIDs = new List<ID>();

     if(! mapContacts.isEmpty())
     {
        contactIds.AddAll(mapContacts.keySet());
     }


     if(! contactIds.isEmpty())
     {
        Messaging.MassEmailMessage massEmail = new Messaging.MassEmailMessage();

        EmailTemplate template = [Select id, name, isActive from EmailTemplate
                                  Where name = 'Update Account Details Notification Template'];
```

```
        if(template.Id != null && template.IsActive == true)
        {
          massEmail.SetTargetObjectIds(contactIds);
          massEmail.SetTemplateID(template.Id);

          Messaging.sendEmail(new Messaging.MassEmailMessage[]{massEmail});
        }
      }
    }
}
```

Execution:

// Invoke the Method...

MessagingUtility.SendReminderNotificationToContacts();

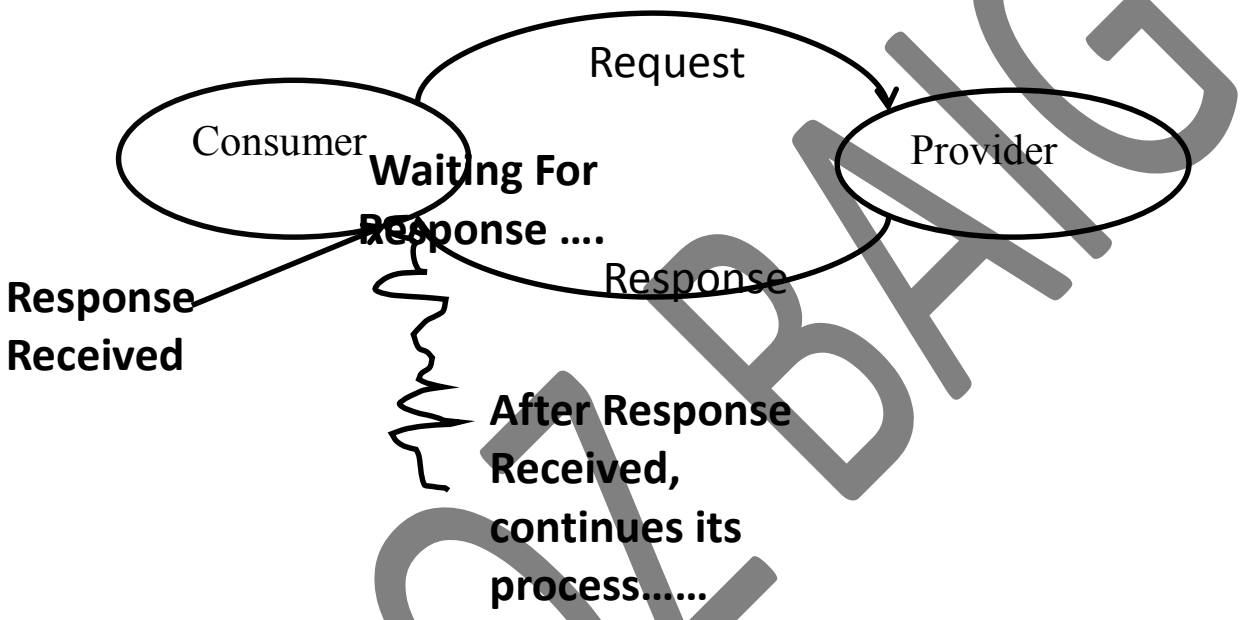
ASYNCHRONOUS PROGRAMMING

Synchronous Processing:

When you execute something synchronously, you wait for it to finish before moving on to another task.

Synchronous simply means that things follow in order. You do step 1, then you do step 2, then you do step 3. Step 3 **never** occurs before step 2 in this synchronous process. Step 2 must conclude before you start step 3.

Synchronous request-reply:
In this "Consumer" sends the request to "Provider", and "Consumer" blocks until response comes back from the "Provider", until response comes back "Consumer" can't proceed.



Asynchronous Proessing:

By using Asynchronous programming, we can execute multiple processes parallelly at the same time.

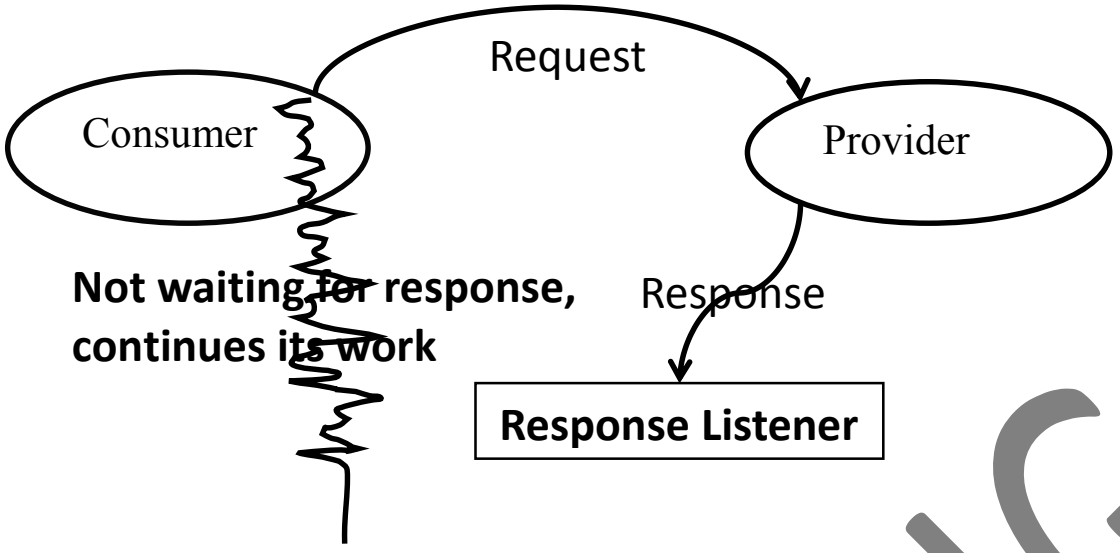
i.e. We no need to wait, till the first job done with it's operation.

When you execute something asynchronously, you can move on to another task before it finishes.

We need to use Asynchronous programming, upon performing the operations on the bulk records or upon using the Long-Running operations.

Asynchronous request-reply:

- ⇨ In this "Consumer" sends the request to "Provider" and "Consumer" will not wait until it gets the response.
- ⇨ After request sent to the "Provider" it continues its flow of execution
- ⇨ "Consumer" will have "**Response Listener**" which listens the response from the "Provider".



Salesforce provides the below features to implement the Asynchronous programming.

1. Batch Apex programming.
2. Schedule Apex Programming
3. Future Methods
4. Flex Queues
5. Queueable Apex.

BATCH APEX PROGRAMMING:

By using Batch apex, we can perform the DML operations on the bulk records at a time.

Batch Apex allows us to process maximum of 50 million records.

Batch apex jobs will executed outside of the Salesforce organization by using the Salesforce resources.

Note: All Batch Apex classes should be defined with the Access specifier "**Global**".

To implement the batch process, we need to use an interface called as "**Database.Batchable<SObject>**".

We need to inherit the interface into our Batch class as below.

Syntax:

```
Global Class <ClassName> implements Database.Batchable<SObject>
{
    // Business Logic...
    // Provide the Implementation for "Batchable" interface
    methods...
}
```

Batchable interface provides the below methods..

1. **Global Database.QueryLocator Start(Database.BachableContext bContext):**

This method is used to execute the SOQL query, to fetch the required records to be get processed.

In order to get more than 50,000 records, we need to execute the query by using "Database.getQueryLocator()" method.

Ex:

string selectLeads = 'select id, firstname, lastname, email, status, company from lead';

Database.getQueryLocator(selectLeads);

"**Database.getQueryLocator()**" will execute the query and store the resultset in Buffer memory and returns the buffer memory address to "Database.QueryLocator" class.

Database.QueryLocator class will get all the records and divide the records into the small manageable pieces / chunks called as "Batches".

Each batch can have max. of 1 - 2000 records. (Default Batch size is : 200)

Start() method will get each batch records from the buffer memory and pass the records to "Execute()" method for processing.

2. **Global void Execute(Database.BachableContext bContext ,List<SObject> recordsToProcess)**

This method will receive the records from "Start method" and perform the required business logic in the records.

We can implement, all the DML operations on the records.

Once the batch of records has been processed, then it will indicates to "Start()" method, to provide the next batch records.

This process will be continued till all the batches will get executes.

Note: Each batch records, will be treated as a discrete transaction. So that Failure of one batch will not affect the other batch records.

Apex doesn't remember the information of the previous batch records into the next batch.

3. **Global Void Finish(Database.BachableContext bContext)**

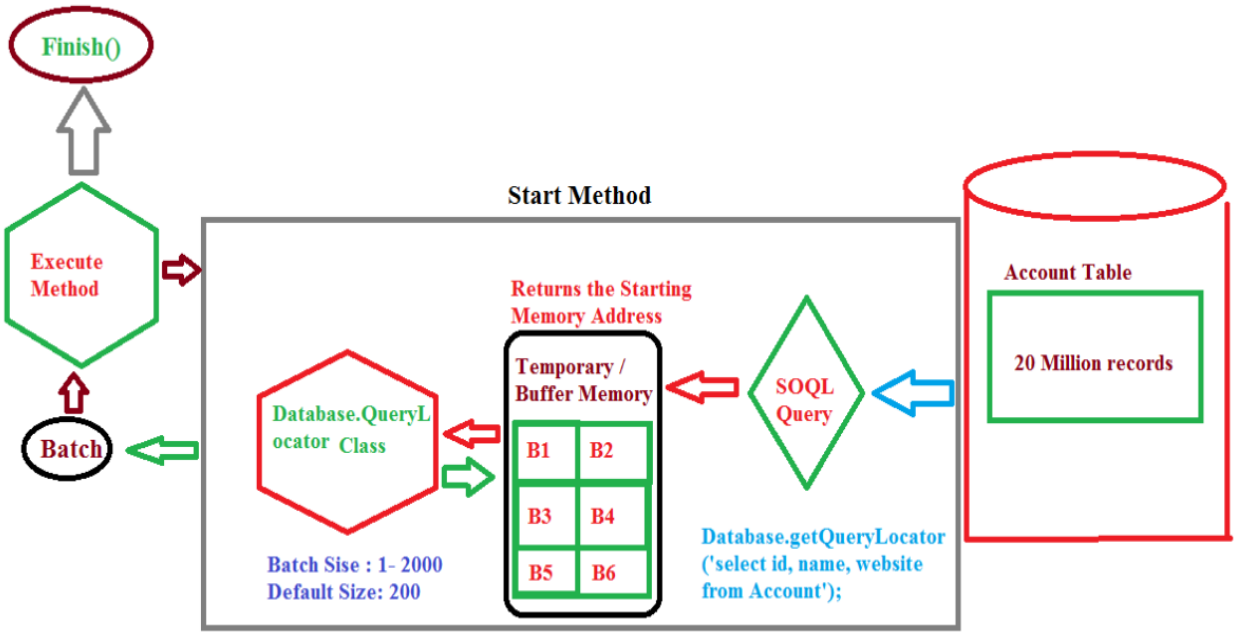
Once all the batches has been processed by "Execute()" method, then Finish method will get executes to perform the "Post-Execution Logic", like sending email notifications, etc.

Note: All the Batch Job details, will get tracked by using an object called as "AsyncApexJob" object.

We need to make a query to "AsyncApexJob" object to get the details for the required job by using "Job Id".

Ex: AsyncApexJob jobDetails = [select id, status, totaljobitems, jobitemsprocessed, NumberofErrors, createdby.email from AsyncApexJob where id = <Batch Job Id>];

Batch Apex Execution Process:



Invoking a Batch Job:

Batch Job can be invoked by using the below ways.

1. By using Triggers.
2. By using Execute Anonymous Window.
3. By using Scheduling
4. By using Another Batch Job.
5. By using Visual Force Pages.

Invoke BatchApex from Execute Anonymous Window:

To invoke a batch class manually, we need to use "Execute Anonymous Window".

1. Create the Object of the Batch Class.

    Ex: AccountOwnershipUpdateBatch accountBatch = new AccountOwnershipUpdateBatch();

2. Execute the Batch Class by using "Database.ExecuteBatch()" method.

Syntax: Id batchJobID = Database.ExecuteBatch(<BatchClassObjectName>);

Note: It will execute the BatchJob, by deviding the records into the various chunks of size 200 (Default batch size will be considered)

ID batchJobID = Database.ExecuteBatch(<BatchClassObjectName>, integer batchSize);

Note: We can indicate the required batch size by passing the integer value.

Usecase:

Write a Batch Apex Job, which can update All Account records website and Phone and Fax values

Batch Class Code:

```
Global class UpdateAccountsBatch implements Database.Batchable<SObject>
{
        Global Database.QueryLocator Start(Database.BatchableContext bContext)
        {
            string accountsQuery = 'select id, name, phone, fax, industry, website from Account';

             return Database.getQueryLocator(accountsQuery);
        }

        Global void Execute(Database.BatchableContext bContext, List<SObject> recordsToProcess)
        {
    if(! recordsToProcess.isEmpty())
    {
       List<Account> accountsToUpdate = new List<Account>();

       for(SObject sobjRecord : recordsToProcess)
       {
          Account accRecord = (Account) sobjRecord;

          // Assign the values...
          accRecord.Website = 'www.forsysinc.com';
          accRecord.Phone = '5555555555';
          accRecord.Fax = '8888888888';
          accRecord.Sic = 'Manufacturing';
          accRecord.Industry = 'Insurance';

          accountsToUpdate.add(accRecord);
       }

       if(! accountsToUpdate.isEmpty())
       {
          update accountsToUpdate;
       }
    }
  }

  Global void Finish(Database.BatchableContext bContext)
  {
     system.debug('Batch Job Id is.....: '+ bContext.getJobID());

     AsyncApexJob jobDetails = [Select id, status, TotalJobITems, jobItemsProcessed,
                                          NumberOfErrors, createdby.email
```

```
                    from AsyncApexJob
                        where id =: bContext.getJobID()];

// Sending the email Alerts...
      MessagingUtility.SendBatchJobStatusNotification(jobDetails, 'Update Accounts Batch');
   }
}
```

Messaging Utility Class:

```
public class MessagingUtility
{
   Public static void SendBatchJobStatusNotification(AsyncApexJob job, string jobName)
   {
      Messaging.SingleEmailMessage email = new Messaging.SingleEmailMessage();

      string[] toAddress = new string[] {job.Createdby.email, 'ferozjani@gmail.com'};
      email.SetToAddresses(toAddress);

      string emailSubject = 'Batch Job :  '+ jobName + ' - '+ job.Id + ' Execution Status Notification.';
      email.SetSubject(emailSubject);

      email.SetReplyTo('ferozjani@gmail.com');
      email.SetSenderDisplayName('Weekly Batch Job Status Notification Alert.');

      String emailContent = 'Dear Customer. <br/><br/> '+
                            'Here is the Status Notification of the Batch Job : '+ jobName +
                            '<br/><br/> Batch Job Id is ....: '+ job.Id +
                            '<br/> Batch Job Name is.....: '+ jobName +
                            '<br/> Batch Job Status is......: '+ job.Status +
                            '<br/> Total Number of Batches.......: '+ job.TotalJobItems +
                            '<br/> Number Of Batches Processed ......: '+        job.JobItemsProcessed +
                            '<br/> Number of Failed Batches ......: '+ job.NumberOfErrors +
                            '<br/> <br/> Thanks & Regards <br/> Dell Sales Supports. ';

      email.SetHTMLBody(emailContent);

      Messaging.sendEmail(new MEssaging.SingleEmailMessage[] {email});

   }
```

Execution:

```
// Create the object of the batch class..
UpdateAccountsBatch accountsBatch =  new UpdateAccountsBatch();

// Invoke the batch class...
  Database.executeBatch(accountsBatch, 20); --> Batch size : 20

//Database.executeBatch(accountsBatch); --> Batch size : 200
```

Note:
1. In Batch Apex, Each batch will be treated as a discrete transaction.
(i.e. Failure of one Batch will not be affected the other batches in the application).

2. Salesforce allows the "Partial Processing" between the batches.
(i.e. If one batch has been failed, then it will continue with the remaining batches as it is)

3. All Batch Jobs are Stateless. (i.e. We can't carry forward one batch results / information to the next batch).

Tracking the Batch Job Results:
We can track the batch job results as below.

Goto "Setup" menu and click on it.
1. Goto "Monitor" menu in left panel.
2. Click on "Jobs" and expand it.
3. Click on "Apex Jobs"
4. Select the Apex Job Name, and track the results.

STATE MANAGEMENT

By default Batch process will execute in "Stateless Mechanism".

i.e. One Batch information / results can't be remembered in the next batch.

To maintain the State between the batches, we need to implement our class with an interface "**Database.Stateful**".

Ex:
```
Global class <ClassName> impements Database.Batchable<SObject>,
                                                    Database.Stateful
{
    Start()

    Execute()

    Finish()
}
```

Usecase:

Write a Batch Apex Program, to Calculate the Total Annual revenue of all Account Records.

Batch Class:

```
Global class CalculateTotalRevenueBatch implements Database.Batchable<SObject>, Database.Stateful
{
```

```
   Global decimal totalAnnualRevenue = 0.0;

        Global Database.QueryLocator Start(Database.BatchableContext bContext)
   {
      string selectAccounts = 'select id, name, annualrevenue from Account where annualrevenue !=
null';

      return Database.getQueryLocator(selectAccounts);
   }

   Global void Execute(Database.BatchableContext bContext, List<SObject> recordsToProcess)
   {
      if(! recordsToProcess.isEmpty())
      {
         for(SObject objRecord : recordsToProcess)
         {
            Account acc = (Account) objRecord;

            totalAnnualRevenue += acc.AnnualRevenue;
         }
      }
   }

   Global void Finish(Database.BatchableContext bContext)
   {
      system.debug('Batch Job Id is.....: '+ bContext.getJobID());

      AsyncApexJob jobDetails = [Select id, status, TotalJobITems, jobItemsProcessed,
                                                NumberOfErrors, createdby.email
                        from AsyncApexJob
                                 where id =: bContext.getJobID()];
      // Sending the email Alerts...
      MessagingUtility.SendTotalAnnualRevenueNotification(jobDetails, 'CalculateTotalRevenueBatch',
totalAnnualrevenue);
   }
}
```

MessagingUtility Class:

```
public class MessagingUtility
{
   Public static void SendTotalAnnualRevenueNotification(AsyncApexJob job, string jobName, decimal
totalRevenue)
   {
      Messaging.SingleEmailMessage email = new Messaging.SingleEmailMessage();

      string[] toAddress = new string[] {job.Createdby.email, 'ferozjani@gmail.com'};
      email.SetToAddresses(toAddress);

      string emailSubject = 'Batch Job :  '+ jobName + ' - '+ job.Id + ' Execution Status Notification.';
```

```
        email.SetSubject(emailSubject);

        email.SetReplyTo('ferozjani@gmail.com');
        email.SetSenderDisplayName('Weekly Batch Job Status Notification Alert.');

        String emailContent = 'Dear Customer. <br/><br/> '+
                                    'Here is the Status Notification of the Batch Job : '+ jobName +
                                    '<br/><br/> Batch Job Id is ....: '+ job.Id +
                                    '<br/> Batch Job Name is.....: '+ jobName +
                                    '<br/> Batch Job Status is......: '+ job.Status +
                                    '<br/> Total Number of Batches.......: '+ job.TotalJobItems +
                                    '<br/> Number Of Batches Processed ......: '+ job.JobItemsProcessed +
                                    '<br/> Number of Failed Batches ......: '+ job.NumberOfErrors +
                                    '<br/> Total Annual Revenue is.....: '+ totalRevenue +
                                    '<br/> <br/> Thanks & Regards <br/> Dell Sales Supports. ';

        email.SetHTMLBody(emailContent);

        Messaging.sendEmail(new MEssaging.SingleEmailMessage[] {email});

    }
}
```

Execution:

```
// Create the object of the batch class..
CalculateTotalRevenueBatch annualrevenueBatch =  new CalculateTotalRevenueBatch();

// Invoke the batch class...
Database.executeBatch(annualrevenueBatch, 10);
```

Governor Limits for Batch Apex:

1. Up to five queued or active batch jobs are allowed for Apex.

2. A maximum of 50 million records can be returned in the Database.QueryLocator object. If more than 50 million records are returned, the batch job is immediately terminated and marked as Failed.

3. If the start method returns a QueryLocator, the optional scope parameter of Database.executeBatch can have a maximum value of 2,000.

4. If no size is specified with the optional scope parameter of Database.executeBatch, Salesforce chunks the records returned by the start method into batches of 200, and then passes each batch to the execute method.

5. Apex governor limits are reset for each execution of execute.

6. The start, execute, and finish methods can implement up to 10 callouts each.

7. The maximum number of batch executions is 250,000 per 24 hours.

Only one batch Apex job's start method can run at a time in an organization. Batch jobs that haven't started yet remain in the queue until they're started.

SCHEDULE APEX PROGRAMMNG

Instead of invoking the Batch Jobs manually, we can make it as scheduled. Which will run on the specified periodical intervals.

To make the Batch Class to be scheduled, we need to follow the below steps.

Step 1: We need to create a New Class, Which should implements an interface "**System.Schedulable**"

Step 2: Define the class with "Global Access Specifier".

Ex:

```
Global Class AnnualRevenueBatchScheduling implements System.Schedulable
{
    // Write the business logic...
}
```

Step 3: We need to provide the implementation for "Execute Method" of Schedulable interface.

Ex:

```
Global Class AnnualRevenueBatchScheduling implements System.Schedulable
{
    Global void Execute(Database.SchedulableContext sContext)
    {
        // Write the Business logic...
    }
}
```

Step 4: Write the Batch Apex Execution Code inside the "Execute" method.

Ex:

```
Global Class AnnualRevenueBatchScheduling implements System.Schedulable
{
    Global void Execute(Database.SchedulableContext sContext)
    {
        // Create the object of the batch Class..
        UpdateAccountsBatch accBatch = new UpdateAccountsBatch();

        // Execute Batch job...
        Database.ExecuteBatch(accBatch, 20);

    }
}
```

Step 5: Scheduling the Apex Class.

Click on "Setup" menu.

1. Goto "Build Menu" in left panel.
2. Click on "Develop" and expand it.
3. Click on "Apex Classes" link.
4. Goto the "Apex Classes" section, and Click on "Schedule Apex" button.
5. Enter the Scheduling Job Name.
6. Select the "Apex Class" to be get Scheduled.
   (Ex: AnnualRevenueBatchScheduling)
7. Click on "Next" button.
8. Select the Periodical intervals, to make the job to be get run.
   8.1. Select the Scheduling Period (From and To Dates)
   8.2. Select the Schedule (Daily / Weekly / Monthly)
   8.3. Select the Preferred Start time of the Job (12 AM, 1.00 AM, ...)
9. Click on "Save" button.

Ex: Class Code:

```
Global class TotalAnnualRevenueSchedulingJob implements System.Schedulable
{
        Global void Execute(System.SchedulableContext sContext)
  {
    // Create the object of the batch class..
    CalculateTotalRevenueBatch annualRevenueBatch = new CalculateTotalRevenueBatch();

    // Invoke the Batch Job..
    Database.executeBatch(annualRevenueBatch, 10);

  }
}
```

Tracking the Schedule Jobs Information:

We can track the Scheduled jobs details with the below navigation.

Goto "Setup" menu and click on it.

1. Goto "Monitor" menu in left panel.
2. Click on "Jobs" and expand it.
3. Click on "Scheduled Jobs" link.
4. Check the Scheduled Jobs Status.

FUTURE METHODS

There are two types of objects that you can interact with on the platform.

Setup Objects:

- A "setup" object is one that must be edited from the setup or builder area of the platform.

- Use setup objects to interact with metadata for declarative development.

For example, you may wish to create your own version of Setup, or restrict the amount of data you need to push to an app on a mobile phone.

Ex: User, UserRole, UserTerritory,Territory, GroupMember, RecordType, Profile, QueueObject, CustomField, CustomObject, CustomTab, EmailTemplate, FieldSet, RecentlyViewed,...etc.

Non-Setup Objects:

Non-Setup objects can be any one of standard objects like Account / Contact / Lead / Opportunity...etc or any custom object created by user / developer.

Ex: Position__C, Customer__C, Account, Contact, Opportunity , Product, PriceBook, etc...

UseCase:

When we try to perform the DML operations on both Setup Object and Non-Setup object in a Single Transaction, Salesforce will throws the "MIXED_DML_OPERATION Exception".

i.e. I am updating a User Record and Updating an Account record in the same ontext as below.

Example:

```
Public class MixedDMLExceptionTestClass
{
   Public static void DoDMLOperations()
   {
                    // Making User Record DeActive (Updating - DML)..

      User u = [select id, username, isActive from user where username = 'david.dodson@gmail.com' limit 1];

      if(u != null )
      {
         system.debug('User Record is...: '+ u);

         u.isActive = false;
         update u;

         System.debug('User Record Updated Successfully. : ' + u);
      }

                    // Udating Account Record's Annual Revenue value (DML)..

      Account acc = [select id, name, rating, annualrevenue from account where name ='Query String Test' limit 1];
      system.debug('Account Record is....: '+ acc);

      if(acc != null)
      {
         acc.annualrevenue = 60000;
```

```
        update acc;

        System.debug('Account Record Updated Successfully. : '+ acc);
      }

   }
}
```

Testing:

MixedDMLExceptionTestClass.DoDMLOperations();

Result:

You will be receiving the "MIXED_DML_OPERATION" Exception. As we can't perform DML operations on both Setup and Non-Setup objects in a single transaction (and vice versa).

Solution:
To avoid these issues, we need to use "Future Annotated Methods", defined with the help of "@Future()" annotation.

FUTURE METHODS:

The future annotation is a great feature on the Salesforce Platform to allow you to execute custom logic asynchronously.

Use the future annotation to identify methods that are executed asynchronously. When you specify future, the method executes when Salesforce has available resources.

For example, you can use the future annotation when making an asynchronous Web service callout to an external service.

A future method runs in the background, asynchronously.

You can call a future method for executing long-running operations, such as callouts to external Web services or any operation you'd like to run in its own thread, on its own time.

Note: You can also make use of future methods to isolate DML operations on different sObject types to prevent the "Mixed DML Error".

Each future method is queued and executes when system resources become available.

That way, the execution of your code doesn't have to wait for the completion of a long-running operation. A benefit of using future methods is that some governor limits are higher, such as SOQL query limits and heap size limits

- We need to use the **"@Future**" annotation to define Future Methods.
Syntax:

Global Class <ClassName>

```
{
        @future
        Public static void <FutureMethodName>(<Parameters List>)
        {
                // Todo..
        }
}
```

Rules / Considerations:

1. Future Annotated method should be always "Static".
2. Future methods return type should be "Void" always.
3. The specified parameters must be primitive data types, arrays of primitive data types, or collections of primitive data types.
Note: We can't pass SObjectTypes / Object Types as parameters to Future methods.
4. We can't call One Future method from another Future annotated method.
5. The maximum number of future method invocations per a 24-hour period is 250,000 or the number of user licenses in your organization multiplied by 200, whichever is greater
6. Methods with the future annotation cannot be used in Visualforce controllers in either getMethodName or setMethodName methods, nor in the constructor.
7. Asynchronous calls, such as @Future or ExecuteBatch, called in a startTest, stopTest block, do not count against your limits for the number of queued jobs

Solution:

```
Public class MixedDMLExceptionTestClass
{
   Public static void DoDMLOperations()
   {
      User u = [select id, username, isActive from user where username = 'david.dodson@gmail.com' limit 1];

      if(u != null )
      {
         system.debug('User Record is...: '+ u);

         u.isActive = false;
         update u;

         System.debug('User Record Updated Successfully. : ' + u);
      }

      UpdateAccount();

   }

   @future()
   Public static void UpdateAccount()
   {
```

```
    Account acc = [select id, name, rating, annualrevenue from account where name ='Query String
Test' limit 1];
    system.debug('Account Record is....: '+ acc);

    if(acc != null)
    {
      acc.annualrevenue = 60000;
      update acc;

      System.debug('Account Record Updated Successfully. : '+ acc);
    }

  }
}
```

Testing: (Execute Anonymous Window)
MixedDMLExceptionTestClass.DoDMLOperations();

FLEX QUEUES IN SALESFORCE

Previously, Salesforce allows us to submit only up to five batch jobs simultaneously by using Batch Apex.

The Apex flex queue enables you to submit up to 100 additional batch jobs for execution.

Note: From Spring ’15 onwards for each New Organization, Apex Flex Queue is enabled by default. To enable Apex Flex Queue in an existing organization, activate the Apex Flex Queue critical update in Setup

When you submit batch jobs by calling **Database.executeBatch()** method, the system places your batch job in "Holding" status before processing the job.

Any jobs that are submitted for execution are in "Holding" status and are placed in the Apex flex queue. Up to 100 batch jobs can be in the "Holding" status.

Note: This new status "**Holding**" indicates that the job is placed in the flex queue and is waiting to be processed when system resources become available.

When system resources become available, the system picks up jobs from the top of the Apex flex queue and moves them to the batch job queue.

The system can process up to five queued or active jobs simultaneously. The status of these moved jobs changes from Holding to Queued. Queued jobs get executed when the system is ready to process new jobs.

Administrators can modify the "Order" of jobs that are held in the Apex flex queue to control when they get processed by the system. We can list out the Flex Queue Jobs as below..

Click on "**Setup**" menu.

1. Goto "Monitor" menu in left panel.
2. Click on "Jobs" and expand it.
3. Click on "Apex Flex Queues".

QUEUEABLE APEX IN SALESFORCE

Salesforce has added the Queueable Interface in the Winter’15 release, which sounds like an alternative to the Future methods.

This interface enables you to add jobs to the queue and monitor them, which is an enhanced way of running your asynchronous Apex code compared to using future methods. (Because, Future method will not generate any ID of the Job, to track it)

Apex processes that run for a long time, such as complex database operations or external Web service callouts, you can run them asynchronously by implementing the Queueable interface and adding a job to the Apex job queue.

And we can track the Job Status by using "AsyncApexJob" object. In this way, your asynchronous Apex job runs in the background in its own thread and doesn’t delay the execution of your main Apex logic

Each queued job runs when system resources become available.

A benefit of using the Queueable interface methods is that some governor limits are higher than for synchronous Apex, such as heap size limits.

(Ex: Heap Size for Synchronous Apex: 6 MB. Heap Size for Asynchronous Apex: 12 MB) And (Number Of SOQL Queries to 200 per transaction)

Syntax:

```
Public Class <ClassName> implements Queueable
{
        Public void Execute(System.QueueableContext qContext)
        {
                // Write the Business Logic..
        }
}
```

Invoking Queueable Apex Class:

```
ID jobId = System.EnqueueJob(new <ClassName>());
```

Future Methods Vs. Queueable Interface:

Similarities:

1. Both Future and Queueable methods can be Queued.
2. Both the methods will get executed upon Salesforce Resources becomes available.

Differences:

1. We can make only One Future method in the class. But, we can make the Entire Class as the Queueable.

2. Future method will not return any JobID upon queued. So we don't have the tracking mechanism. But, Queueable interface will returns a JOB ID upon Queued. So we can Track the Status and monitor them easily.

3. Future Method will not allows us to pass the "SObject Types" as the Parameters. But, Your queueable class can contain member variables of non-primitive data types, such as sObjects or custom Apex types. Those objects can be accessed when the job executes.

4. **Chaining jobs**: You can chain one job to another job by starting a second job from a running job. Chaining jobs is useful if you need to do some processing that depends on another process to have run first.

Queueable Apex Limits:

1. You can add up to 50 jobs to the queue with System.enqueueJob in a single transaction.

2. No limit is enforced on the depth of chained jobs, which means that you can chain one job to another job and repeat this process with each new child job to link it to a new child job.

3. When chaining jobs, you can add only one job from an executing job with System.enqueueJob, which means that only one child job can exist for each parent queueable job. Starting multiple child jobs from the same queueable job isn't supported.

Usecase: Write an apex program to implement the Queueable Apex and implement the Chaining of jobs.

AccountsQueueable Code: (JOB 1) : (Creating Account Record)

```
public class AccountsQueueable implements Queueable
{
   public void Execute(QueueableContext qContext)
   {
      Account acc = new Account();

      acc.Name = 'Queueable Account';
      acc.Rating = 'Hot';
      acc.Industry = 'Finance';
      acc.Type = 'Prospect';
      acc.Ownership = 'Public';
      acc.AnnualRevenue = 5600000;
      acc.Phone = '9988776655';
      acc.Fax = '9988000000';
      acc.Website = 'www.gmail.com';
      acc.Active__c = 'Yes';
      acc.PAN_Number__c = 'ALPPB9439E';

      // Insert the Account...
      insert acc;

      // Chaining the Queueable Apex.. (Calling the Second Job)
      if(acc != null)
      {
          ID contactsJobID = system.enqueueJob(new ContactsQueueable(acc));
      }
   }
}
```

ContactsQueueable Code: (JOB 2): (Creating an Associated Contact)

```
public class ContactsQueueable implements System.Queueable
{
   Public Account accRecord;

   Public ContactsQueueable(Account acc)
   {
      accRecord = acc;
   }

   Public void Execute(system.QueueableContext qContext)
   {
      if(accRecord != null)
      {
         contact con = new Contact();
           con.FirstName = 'Queueable';
           con.LastName = 'Contact Record';
           con.Email = 'Queueable@gmail.com';
           con.Title = 'Queueable Test';
           con.SSN_Number__c = 'ALPPB9090R';

           con.AccountId = accRecord.id;
         insert con;
      }
   }
}
```

Execution:

```
id accountsJobID = system.enqueueJob(new AccountsQueueable());

AsyncApexJob jobDetails = [Select id, Status from AsyncApexJob where id =: accountsJobId ];

system.debug('Job Id is...: '+ accountsJobID);
```

APEX TRIGGERS

By using Workflow Rules, we can perform certain actions based on the event. But, we have the below limitations in workflow rules.

1. By using Workflow Rules, we can perform the below 4 actions. Like Sending Email Alerts, Assigning a Task to User, Updating a field value, and Send an outbound message to the external system.

2. Workflow Rules will fire always After Performing the operations.

3. By using Workflow Rules, we can't implement the Complex Validation Rules.

4. We can't perform DML operations through the workflow rules (Ex: Insert, Update, Delete, Upsert, Undelete)

5. By using workflow rules, we can update the Field in the same object, or in the Parent Object which is associated with "Master-Detail association". (i.e. We can update the field from the Child to Parent object) 5.1. If the association is Lookup, then workflow will not support.

5.2. If the user wants to update the fields from "Parent To Child", then workflow will not support.

6. We can't select multiple "Email Templates" to notify the people upon creating / updating the records inside the object with different subject and content.

To avoid the above limitations, we have to use "**Triggers**".

Trigger is a Custom Apex Code, which will get fired automatically upon performing the DML Events.

Trigger will get fired upon performing the DML operations (Insert, Update, Delete, Undelete) on the object records.

By using Triggers, we can implement the complex validation rules, complex Business Logics, and Complex Transactional flows inside the application.

Triggers can fire on both "Before" and "After" performing the operations.

By using Triggers, we can perform all the DML operations (like insert, update, delete, upsert, undelete) on the same object or on any other object records. Triggers will get fired "Asynchronously".

Each trigger should be get associated with an object. But, an object can have one or more associated triggers.

Syntax:

```
Trigger <TriggerName> ON <ObjectName>(<EventsList>)
{
        // Write the Business Logic..

}
```

Ex:

```
Trigger LeadTrigger ON Lead(<EventsList>)
{
        // Write the Business Logic...
}



Trigger PositionTrigger ON Position__C(<EventsList>)
{
        // Write the Business Logic..
}
```

Trigger Events:

Upon creating the Trigger, we need to specify, on what occasion to trigger should get fired.

Apex Provides the below 7 Events, can be used inside the trigger, to indicate when the trigger should get fired.

1. **Before Insert:** This event indicates the trigger should get fired "Before Inserting" the record into the object.

Ex: Validating the data, upon eliminating the Duplicate records, upon maintaining the One-one association, etc.

2. **Before Update:** This event will make the trigger to be get fired "Before Updating" a record into the object.

Ex: Validating the Data, eliminating the duplicate records, etc.

3. **Before Delete:** This event will make the trigger to be get fired "Before Deleting" the record from the associated object.

Ex: Prevent the Deletion of specific records, Verify the Access Permissions before deletion, Removing the Child records before removing the parent, etc.

4. **After Insert:** This event will make the trigger to be get fired "After Inserting" the record into the object.

Ex: Sending the Email Alerts, Synchronizing the Child records, performing post-execution logic, updating Rollup summary fields, etc.

5. **After Update:** This event will make the trigger should get fired, after updating the record.

Ex: Sending Email Alerts, Synchronize the records, etc.

6. **After Delete:** This event will make the trigger to be get fired after deleting the records from the object.

Ex: Updating Rollup Summary fields, Removing the associated Child records, etc.

7. **After Undelete:** This event will make the trigger should get fired after re-storing the deleted record back to the object.

Ex: Updating Rollup Summary field, etc.

Note: A Trigger can get associated with one or more events. We need to specify the multiple events by separating a comma.

Ex:

```
Trigger AccountsTrigger ON Account(Before Insert, Before Update)
{
        // Write the Business Logic for Before Insert..
                ....

        // Write the Business Logic for Before Update...
                ....
}
```

Ex:

```
Trigger CandidateTrigger ON Candidate__C(Before Insert, After Insert)
{
        // Write the Business Logic for Validation... (Before Insert)

        // Write the Business Logic for Replicate the Record (After Insert)

        // Write the Business Logic for Sending the Email Alert. (After Insert)
}
```

Note: Each trigger should be associated with an object. But, an object can have one or more Triggers associated with it.

Note:

1. As a best practice, it is always recommended to have only one trigger per an object.

If the object contains multiple triggers, which will fire on the same event, then we can't assure the order of execution. We don't have any options in salesforce to control the order of execution of triggers.

2. To avoid this, we have to write the code inside a single trigger in the required order to perform the actions.

Types of Triggers:

Apex provides 2 types of Triggers as below.

1. Before Triggers: These triggers will get fired "Before" performing the operations on the specified object.

Ex: By using Before Triggers we can achieve Complex Validation Rules, Eliminating Duplicate Records, Validating the Conditions before Updating / Deleting the records, etc.

2. After Triggers: These Triggers will get fired always "After" performing operations on the specified object.

Ex: By using After Triggers, we can implements the features to send Email Alerts, Update the Fields, Update Rollup Summary values, etc.

Trigger Context Variables:

Upon writing the Business Logic inside the Trigger, we need to segregate the code blocks associated with "Multiple Event" with the help of "Trigger Context Variables". These are used to get the Current Status of the Trigger.

Apex provides the below Trigger Context Variables

1. Boolean Trigger.isInsert: It returns the TRUE, if the trigger has been fired because of an insert operation. Else it returns FALSE.

2. Boolean Trigger.isUpdate: It returns TRUE, if the trigger has been fired because of an Update operation. Else it returns FALSE.

3. Boolean Trigger.isDelete: It returns TRUE, if the trigger has been fired, because of a Delete operation. Else it returns FALSE.

4. Boolean Trigger.isUndelete: It returns TRUE, if the trigger has been fired because of an "Undelete" operation. Else it returns FALSE.

5. Boolean Trigger.isBefore: It returns TRUE, if the Trigger is about to perform the operation. Else it returns FALSE.

6. Boolean Trigger.isAfter: It returns TRUE, if the trigger is already done with the operation. Else it returns FALSE.

Ex:

```
Trigger LeadTrigger ON Lead(Before Insert, After Insert, Before Delete)
{
    if(Trigger.isInsert && Trigger.isBefore)
    {
        // Write the Business logic to validate the record values.
    }

    if(Trigger.isInsert && Trigger.isAfter)
    {
        // Write the Business Logic, to replicate the Lead Record into Prospect
        Object.

        // Write the Business Logic, to Send the Email Alert.
    }

    if(Trigger.isDelete && Trigger.isBefore)
    {
        // Write the Business logic, to be fired during Before Deleting the record.
    }
}
```

7. Trigger.New:

Trigger.New is a List Collection, which contains the Current Context Records. i.e. It will hold the Newly iserting records into the Collection, for the processing.

Syntax: List<SObject> Trigger.New Note:

- Trigger.New will be available in both "Insert and Update" operations.
- Trigger.New will not be available in "Delete" operation

8. Trigger.Old:

It is a List Collection, which holds the collection of Previous Context records, which are holding the old values.

Syntax: List<SObject> Trigger.Old

Note:
- Trigger.Old Will not be available in "Insert" operation.
- It will be available in Update and Delete Operations.

9. Trigger.NewMap:

Trigger.NewMap is a Collection, which holds the Current Context records in "Map" format.

Syntax: Map<Id, SObject> Trigger.NewMap

Where

Key → Record ID will be the Key.

Value → Whole Record will be the value.

Note:

- Trigger.NewMap will not be available in "Before Insert" event.
- Trigger.NewMap will be available in "Update and After Insert" operations.

10. Trigger.OldMap:

Trigger.OldMap is a Map Collection, which holds the previous context records in "MAP" format.

Syntax: Map<Id, SObject> Trigger.OldMap

Where

Key --> Record ID will be the Key.

Value --> Whole Record will be the value.

Note: Trigger.OldMap will be available in "Update and Delete" operations

Trigger Bulkification / Bulkify Trigger:

Upon creating the Trigger, we have to write the Business in such a ways, that it can able to handle the processing of either One / more records at a time.

As a best practice, it is always recommended to make the Trigger "Bulkify".

We can achieve the Bulkify Trigger, by using "FOR" Loop. Which can iterate all the records and process all the records at a time. By using Bulkification, we can make the code "Generic".

Ex:

```
Trigger AccountsTrigger ON Account(Before Insert)
{
    if(Trigger.isInsert && Trigger.isBefore)
    {
        for(Account acc : Trigger.New) // Bulkify...
        {
            if(acc.Rating == 'Hot')
            {
```

```
                    }
                }
            }
        }
```

Ways to Create the Trigger:

Apex provides 3 Ways to Create the Triggers.

1. **By using Standard Navigation**:
Click on "Setup" menu.
1. Goto "Build" menu in Left Panel.
2. Click on "Develop" and expand it.
3. Click on "Apex Triggers" link.
4. Click on "New" button, to Create a New Trigger.
5. Write the Trigger Code inside the Editor.
6. Click on "Save" button.

Note: Once the user click on "Save" button, it will send the code the "Force.com" platform. It will compile the code and save the compiled code in "Metadata Repository".

2. By using Developer Console

Goto your Name and Expand it.

1. Click on "Developer Console" link.
2. Goto the "File Menu" and expand it.
3. Click on "New" menu item.
4. Click on "Apex Trigger" from Submenu.
5. Enter the Trigger Name inside the Textbox, to be get created.
6. Select the ObjectName from the Picklist.
7. Click on "Ok" button.
8. Write the Trigger Business logic inside the Editor.
9. Click on "CTRL + S" to save the Trigger code.

3. By using Eclipse IDE.

De-Activating the Trigger:

In few cases we no need to fire the triggers for some time. So that, we can make the trigger de-activated as below.

Click on Setup menu.

1. Goto "Build" menu in left panel.
2. Click on "Develop" and expand it.

3. Click on "Apex Triggers".
4. Select the Required trigger to de-activate
5. Click on "Edit" button.
6. Unselect the checkbox "isActive"
7. Click on "Save" button.

/*

Create a Trigger on Contact Object, to make the Email and Phone Fields should be Manadatory upon creating or updating a Contact Record.

***/**

Object Name: Contact
Event Name : Before Insert, Before Update

```
trigger ContactDetailsValidationTrigger on Contact (before insert, before update)
{
   // Before Insert Or Before Update
    if( (Trigger.isInsert || Trigger.isUpdate ) && Trigger.isBefore)
   {
      for(Contact con : Trigger.New)
      {
         if(con.Email == null || con.Email =='')
         {
            con.Email.AddError('Please Enter the Email Id value.');
         }
         else if(con.Phone == null || con.Phone == '')
         {
            con.Phone.AddError('Please Enter the Contact Number.');
         }
      }
   }
}
```

/* **Create a Trigger on Account Object, To assign the Annualrevenue value as below..**

Industry Name		**Annualrevenue**
Banking	**---->**	**5000000**
Finance	**---->**	**4000000**
Insurance	**---->**	**3500000**
Healthcare	**---->**	**2500000**
Else	**---->**	**500000**

***/**

ObjectName : Account
Event Name : Before Insert , Before Update

```
trigger AutoPopulateAnnualRevenue on Account (before insert, before update)
{
   if( (Trigger.isInsert || Trigger.isUpdate) && Trigger.isBefore)
   {
      for(Account acc : Trigger.New)
      {
         if(acc.industry == 'Banking')
            acc.Annualrevenue = 5000000;
         else if (acc.industry =='Finance')
            acc.Annualrevenue = 4000000;
         else if(acc.industry == 'Insurance')
            acc.annualrevenue = 3500000;
         else if(acc.industry == 'Healthcare')
            acc.annualrevenue = 2500000;
         else
            acc.annualrevenue = 500000;
      }
   }
}
```

/*

Create a Trigger, to make sure Each Contact Should be get associated with an Account.

*/

ObjectName : Contact
Event Name : Before Insert , Before Update

Note:

Between Account and the Contact, we have Lookup Association.

Account --> Parent

Contact --> Child.

Relationship Fields ---> Contact : AccountID

```
trigger MakeContactAssociated on Contact (before insert, Before Update)
{
   if((Trigger.isInsert || Trigger.isUpdate) && Trigger.isBefore)
   {
      for(Contact con : Trigger.New)
      {
```

```
        if(con.AccountId == null )
        {
           con.AddError('Please Make the Contact Should be associated with an Account.');
        }
      }
   }
}

/*
        Create a Trigger To Prevent the deletion of an Active Account Record.
*/

ObjectName : Account
Event Name : Before Delete

trigger PreventDeletionOfActiveAccount on Account (Before Delete)
{
   if(Trigger.isBefore && Trigger.isDelete)
   {
      for(Account acc : Trigger.Old)
      {
         if(acc.Active__c == 'Yes')
         {
            acc.AddError('Your Are Not Authorized to Delete an Active Account.');
         }
      }
   }
}

/*
        Write an apex Trigger, to prevent the Duplicate Candidate Records inside the object.
*/
ObjectName : Candidate__C
Event Name : Before Insert, Before Update

trigger PreventDuplicateCandidates on Candidate__c (before insert, before update)
{
   if( (Trigger.isInsert || Trigger.isupdate) && Trigger.isBefore)
   {
      for(Candidate__c cnd : Trigger.New)
      {
         integer recordsCount =    [select count() from Candidate__C where name =: cnd.name];
```

```
        if(recordsCount > 0)
        {
          cnd.AddError('We have a Candidate Record with this Name. Duplicate Candidates cannot be accepted.');
        }
      }
   }
}
```

**/*
Write a Trigger, to Prevent the Deletion of the Associated contacts upon removing the Account record.
*/**

ObjectName : Account
Event Name : Before Delete

```
trigger PreventContactsDeletion on Account (before Delete)
{
  f(Trigger.isBefore && Trigger.isDelete)
   {
                List<Contact> lstContacts = [Select id, firstname, lastname, accountid from Contact where accountid IN : Trigger.OldMap.KeySet()];

      if(! lstContacts.isEmpty())
      {
        for(Contact con : lstContacts)
        {
          con.Accountid = null;
        }

        update lstContacts;
      }

   }
}
```

**/*
Write an Apex Trigger, to synchronize the Contact Record's Phone and Fax value with the associated Account Record's Phone and Fax Numbers.
*/**

```
trigger PreventContactsDeletion on Account (before Delete, After Update)
{
   if(Trigger.isAfter && Trigger.isUpdate)
   {
      List<Account> lstAccounts = [select id, name, phone, fax,
                                   (Select id, firstname, lastname, phone, fax from Contacts)
                                   from Account where ID in : Trigger.NewMap.keyset()];

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

      if(! lstAccounts.isEmpty())
      {
         for(Account acc : lstAccounts)
         {
            if(! acc.contacts.isEmpty())
            {
               for(Contact con : acc.Contacts)
               {
                  con.Phone = acc.Phone;
                  con.Fax = acc.Fax;

                  contactsToUpdate.Add(con);
               }
            }
         }
      }

      if( ! contactsToUpdate.isEmpty())
      {
         update contactsToUpdate;
      }
   }
}
```

Apex Sharing

When the Object's OWD has been assigned as "Private", then the Record will be visible to the "Record Owner and his Manager".

Other users inside the organization will not have the access on the record.

To make the record visible to the other users inside the Organization, we have to use the below features.

1. Manual Sharing (Achieved by using "Sharing" Button)

2. Automated Sharing (Sharing Rules)
   We can create our own automated rules, to share the records to the required users based on the specified conditions with the required "Level Of Access".

Drawbacks:

1. We need to define the conditions, upon creating the rule. (i.e. We can't add the conditions at runtime)
2. We need to select the users / Public Group to share the record, upon creating the Rule.
3. We need to hardcode the "Level Of Access" upon configuring the Rule.

To avoid these problems, we can use "Apex Sharing", through which we can share the record to a User / Group of Users (Public Group) based on the need with the required "Access Level".

On a Single Record, Multiple users can have Different Access Level.

```
                                                --> User1 (Read Only Access)
                                                |
Account Record --- -> User2 (Owner : Full Access)
                                                |
                                                --> User3 (Read / Write Access)
```

All the Sharing information will get resides inside the "Share" object provided by the salesforce.

For each Standard / Custom object, salesforce will provides a Share Object as below.

Object Name	Share Object Name
Account	AccountShare
Contact	ContactShare
Opportunity	OpportunityShare
Lead	LeadShare
Case	CaseShare
...	
...	
Position__c	Position__Share
Candidate__C	Candidate__Share
Hiring_Manager__c	Hiring_Manager__Share

Note:

Share Objects can be accessible, if the associated Parent object's OWD has been assigned as either "Private / Public ReadOnly".

Upon sharing the record through Programatically, we have to use the "associated Share Classes" provided by the salesforce.

Each Share Class contains the below properties..

Ex:

Position__Share posShare = new Position__Share();

Properties:

1. ParentID = "Position Record ID"

This property is used to specify the "Record ID", which needs to be get shared to the users.

Ex:

posShare.ParentId = "Position Record Id";

2. AccessLevel ="EDIT / READ".

This property is used to specify the Level Of Access to be granted to the users on the record.

EDIT --> Grant Read and Write Permissions.
READ --> Grant Only View Permission on the Record.

Ex:

posShare.AccessLevel = 'EDIT';

3. UserOrGroupID = "User / Group ID"

This property is used to specify the "User ID / Public Group ID", to whome the record should get shared.

Ex:

posShare.UserOrGroupID = 'User ID (005)';
posShare.UserOrGroupID = 'Public Group ID (00G)';

4. RowCause='Sharing Reason Name'

This property is used to specify the Sharing Reason, because of which the record is visible to the users.

Ex:

posShare.RowCause = Schema.Position__Share.RowCause.Manual;
(i.e. Manual Sharing)

The Developer / Administrator can create their own sharing reasons based on the need.

Click on the "Required Object's Tab". (Ex: Positions)

1. Expand the "Force.com Quick Access Menu".
2. Click on "View Object" link.
3. Goto the "Sharing Reason Section".

4. Click on "New" button, to create a New Sharing Reason.
5. Enter the Sharing Reason Name and the Label.
(Ex: Label : Requested By Client
Name : Requested_By_Client__c)
6. Click on "Save" button.

Ex: posShare.RowCause = Schema.Position__Share.RowCause.Requested_By_Client__c;

/*

Configure an Apex Sharing Process, to Share the Hiring Manager Record to the Selected User with "Read / Write" access.

Pre-Requisite:

1. Create a "Lookup Field" inside the Hiring Manager Object with the Parent "USer".

Note:

1. **Upon creating a New Hiring Manager, the record should be visible to the Selected User" with the help of "Lookup".**
2. **In future, if the "User" in lookup field has been changed, then the "New User" should get the Access on the Record. (i.e. Old User Should not get the Access)**
3. **Once the New USer has got the Access on the Record, then the New User and Old User Should be Notified by sending the email notification to his email id automatically.**

***/**

Trigger Code:

```
trigger HiringManagerSharingTrigger on Hiring_Manager__c (after insert, after update)
{
   if(Trigger.isAfter && Trigger.isInsert)
   {
      List<Hiring_Manager__Share> lstHrsToShare = new List<Hiring_Manager__Share>();

      for(Hiring_Manager__C hr : Trigger.New)
      {
         if(hr.Share_Record_To__c != null)
         {
            Hiring_Manager__Share hrShare = new Hiring_Manager__Share();

            hrShare.ParentId = hr.Id;
            hrShare.UserOrGroupId = hr.Share_Record_To__c;
            hrShare.AccessLevel = 'EDIT';
            hrShare.RowCause = Schema.Hiring_Manager__Share.RowCause.Requested_By_Client__c;

             lstHrsToShare.Add(hrShare);
         }
      }
```

```
    if(! lstHrsToShare.isEmpty())
    {
      Insert lstHrsToShare;
    }
  }

  if(Trigger.isAfter && Trigger.isUpdate)
  {
    Set<ID> removeAccessTo = new Set<ID>();
    List<Hiring_Manager__Share> grantAccessTo = new List<Hiring_Manager__Share>();

   for(Hiring_Manager__c hr : Trigger.New)
    {
          // Verify, the user field value has been changed or not..

      if(hr.Share_Record_To__c != null && (Trigger.OldMap.Get(hr.id).Share_Record_To__C !=
Trigger.NewMap.get(hr.id).Share_Record_To__c))
      {
        // Get the Old User Id, to remove the Access..
       removeAccessTo.Add(Trigger.OldMap.Get(hr.id).Share_Record_To__c);

        Hiring_Manager__Share hrShare = new Hiring_Manager__Share();

         hrShare.ParentId = hr.Id;
        hrShare.UserOrGroupId = hr.Share_Record_To__c;
        hrShare.AccessLevel = 'EDIT';
        hrShare.RowCause = Schema.Hiring_Manager__Share.RowCause.Requested_By_Client__c;

       grantAccessTo.Add(hrShare);
      }
    }

   // Remove the Access to Old Users.
    if(! removeAccessTo.isEmpty())
    {
          Delete [select id, userOrGroupId from Hiring_Manager__Share
                  where UserOrgroupId IN : removeAccessTo and
                  parentid IN : Trigger.NewMap.KeySet()];
    }

  // Grant the Access to Newly selected user...
    if(! grantAccessTo.isEmpty())
    {
      Insert grantAccessTo;
    }
  }
}
```

CUSTOM SETTINGS IN APEX

Custom Settings are also like as Custom Objects, Where we can store the information by creating the fields.

1. Salesforce.com introduced Custom Setting, which allows you to store custom data sets and associate them on an organization-wide, profile or user basis.

2. Custom Settings are essentially custom objects that are exposed in the applications cache and are accessible via their own API.

3. You can certainly build your own custom objects to store settings but using custom settings is much quicker (again they are stored in the application cache) and do not count against SOQL limits when fetched.

4. You can also use custom settings in formula fields, Apex code, Visualforce Pages, and the Web Services API.

Custom settings come in two flavors:

1. List Custom Settings.

1.1. List custom settings allow you to store organization-wide static data that is frequently used.
1.2. List Custom Settings can't be accessible in Formula Fields.

2. Hierarchy Custom Settings:

2.1. Hierarchy Custom Settings allows you to store Profile / User Specific data.
2.2. These can be accesible from Formula Fields using
"$Setup.<CustomSettingName>.<FieldName>"

Note: When you create a new custom settings the platform creates a custom object in the background for you (notice the API Name field). You then add additional fields to the custom setting in the same manner that you do for custom objects

Always it is recommended to use the Custom Settings while storing the Static information. Instead of wasting one Custom Object.

In Custom object, we have to retrieve the data by writing the SOQL Queries.

Note:

1. Whereas in Custom Settings, We can retrieve the Data without writing any SOQL queries.

2. Custom Settings Data will be storing into the Application Cache. So that we can efficiently use the records.

3. We can store max of 10 MB of data in Custom Settings Data in an organization. Which is purely based on the number of Full Licenses (Salesforce Licenses) available in the organization.

Ex:

1 Full License --> 1 MB of Storage
2 Full Licenses --> 2 MB of Storage
5 Full Licenses --> 5 MB of Storage
10 Full Licenses --> 10 MB of Storage
15 Full Licenses --> 10 MB of Storage

4. We can access the Custom Settings data in Formula Fields, Apex Code and Visual Force pages and Web services and API's.

Creating Custom Settings:

We can create the custom setting as below.

Goto Setup --> Goto Develop
--> Click on Custom Settings
--> Click on "New" Button
- Enter the Custom Setting Label and Unique Name
- Select the Type as "List"
- Select the Accessibility as "Public"
- Click on "Save"

Create the Fields in the Custom Settings:

We can able to create the new fields in the custom setting in-order to store the data.

Note: We can create up to 300 fields in Custom Settings

Creating Field :

- Goto the Custom Field Section

- Click on New, Select the Field Type as "Text" Enter the Name : "Region Manager Name"

- Click on Save

Storing the Data in Custom Settings:

1. Goto the Custom Setting Page --> Click on "Manage"

2. Click on "New" to create a new record

3. Enter the Details for the Region Name and Region Manager Name fields.

Ex: Asia --> Yashwanth Kumar

Americas --> David Dodson

Europe --> Rajesh Kumar.

....

Accessing the Custom Setting Data:

Methods in Custom Settings:

1. GetAll()

2. GetInstance(CustomSettingRecordName)

3. GetValues(CustomSettingKeyName)

- To Access the Custom Setting Data, we have to use "getAll()" function.

- The Return type of the custom setting is MAP.

Key: Region Name is the Key

Value : The Whole Record of that Region

Syntax:

Map<string,Custom_Setting_Name> <objName> = <Custom_Setting_Name>.getAll();

Ex:

Map<string, QEdgeRegionalInformation__c> = RegionDetails__c.getAll();

RegionDetails__c reg = RegionDetails__c.getInstance('Americas');

RegionDetails__c reg = RegionDetails__c.getValues('Americas');

Accessing Custom Settings Data:

// Getting the Records from Custom Settings...

```
Map<string, RegionDetails__c> mapRegions = RegionDetails__C.GetAll();
system.debug('Collection Size is...: '+ mapRegions.size());
    if(! mapRegions.isEmpty())
    {
      for(RegionDetails__c reg : mapRegions.Values())
      {
        system.debug('Record is...: '+ reg);
      }
    }
```

// Get the specified Region Details...

```
system.debug('Americas Region Details....: '+ RegionDetails__c.GetInstance('Americas'));
```

// Get the Region Manager Name of "Middle East"

```
system.debug('Region Manager Name is...: '+ RegionDetails__c.GetValues('Middle East').Region_Manager_Name__c);
```

Trigger Code:

```
trigger HiringManagerTrigger on Hiring_Manager__c (before insert, before update)
{
        if(Trigger.isBefore && (Trigger.isInsert || Trigger.isUpdate))
  {
    for(Hiring_Manager__C hManager : Trigger.New)
    {
      if(hManager.Region__c != null && hManager.Region__c != '')
      {
        hManager.Region_Manager__c =
RegionDetails__C.GetValues(hManager.Region__c).Region_Manager_Name__c;
      }
    }
  }
}
```

BULK LEAD CONVERSION

Lead is nothing but a Person / an Organization / a Business, who showed interest in our organization products / services.

Leads are also called as "Prospect / Potential Customer / Potential Client".

Upon converting the Lead as the Customer, we have to use "Convert" button, which presents on the Lead Record's Detailed Page. i.e. We need to Convert each Lead record manually by using Convert button, one by one.

We don't have any such features to convert the Bulk Lead Records as the customer at a time.

If the user converts the Lead Records one by one in sequential manner, will consume more time.

To avoid this problem, We have the below options.

1. We can Use the "AppExchange" Products.

2. We can use our own Custom Apex Code, to Convert the Bulk Lead Records at a time.

To convert the Lead Records as the Customer through programmatically, we have to use a Class called as "**Database.LeadConvert**" class.

This class is used to Hold the Lead Record, which needs to be get Converted. To achieve this, we need to create the Object of the "LeadConvert" class.

Ex:

```
Database.LeadConvert lConvert = new Database.LeadConvert();
```

To hold the multiple lead records to be get converted at a time, we need to create a "List" Collection as below.

Ex:

```
List<Database.LeadConvert> lstLeadConverts = new List<Database.LeadConvert>();
```

Methods:

LeadConvert class provides a set of ready-made methods to be used to specify the Lead Settings to be used upon conversion.

1. SetLeadID(<LeadRecordID>):

This method is used to specify the "Lead Record ID", which needs to be get converted.

Ex: lConvert.SetLeadID('00Q7F000003TPVJ');

2. SetSendNotificationEmail(Boolean):

This method is used indicate, that an email notification to be routed to the Lead Owner regarding the Conversion or not.

TRUE --> Send the Notification Email

FALSE --> Won't Send the Notification

Ex: lConvert.SetSendNotificationEmail(true);

3. SetDoNotCreateOpportunity(<Boolean>):

This method is used indicate, the Opportunity record should be created upon conversion or not.

TRUE --> Won't Create the Opportunity

FALSE --> Will Create the Opportunity

Ex: lConvert.SetDoNotCreateOpportunity(false);

4. SetConvertedStatus(<String Status>):

This method is used to assign the Status for the Lead Record upon conversion.

Ex:

We need to collect the LeadStatus value from the "LeadStatus" object dynamically at runtime.

```
LeadStatus lStatus = [select id, MasterLabel, isConverted from LeadStatus
                                        Where isConverted = true Limit 1];
        if(lStatus != null)
        {
                lConvert.SetConvertedStatus(lStatus.MasterLabel);
        }
```

Database.ConvertLead(List<Database.LeadConvert>):

This method is used to Convert All the Specified Lead Records as the customers at a time.

We can track the Result of Each Lead Record Conversion process with the help of "**Database.LeadConvertResult**" class.

Ex: Database.LeadConvertResult[] results = Database.convertLead(lstLeadConverts);

/*

Create a Trigger, to AutoConvert the Lead Record upon changing the Lead Status to "Closed - Converted".

*/

Object Name : Lead

Event Name : After Update

```
trigger AutoLeadConversion on Lead (After Update)
{
        if(Trigger.isAfter && Trigger.isUpdate)
  {
    // Get the Lead Record Status...
    LeadStatus lStatus = [Select id, MasterLabel, isConverted
                          from LeadStatus
```

```
                                        Where isConverted = true
            Limit 1];

                        // List to hold the Lead Records to Convert..
        List<Database.LeadConvert> lstLeadsToConvert = new List<Database.LeadConvert>();

        for(Lead ld : Trigger.New)
        {
          if(ld.Status == 'Closed - Converted' && ld.IsConverted == false)
          {
            // Store the Lead Details into the LeadConvert Class..
            Database.LeadConvert lConvert = new Database.LeadConvert();

            // Specify the Lead Id...
            lConvert.SetLeadID(ld.Id);

            // Specify the Email Alert..
            lConvert.SetSendNotificationEmail(true);

            // Specify the option to generate Opportunity..
            lConvert.SetDoNotCreateOpportunity(ld.Do_Not_Create_Opportunity__c);

            // Specify the Lead Status...
            lConvert.SetConvertedStatus(lStatus.MasterLabel);

           // Add the Record to Collection to Convert Bulk Records..
           lstLeadsToConvert.Add(lConvert);
          }
        }

        // Convert the Lead Records...
        if(! lstLeadsToConvert.isEmpty())
        {
          Database.LeadConvertResult[] results = Database.convertLead(lstLeadsToConvert);
        }
      }
    }
```


De-Activating the Triggers:

Triggers are the Pluggable components, which can be make it as "ON / OFF".

We can De-activate the Trigger as below.

Click on "Setup" menu.

1. Goto "Build Menu" in left panel.

2. click on "Develop and Expand it".

3. Click on "Apex Triggers" link.

4. Select the Required Trigger to Deactivate.

5. Click on "Edit" link beside the Trigger Name.

6. Unselect the checkbox "IsActive"

7. Click on "Save" button.

Note: Once the Trigger has been De-Activated, then it won't fire upon performing the DML operations on the object.

TRIGGER HANDLER FACTORY

Once the Trigger has been deployed to the Production Instance. If there is any issues, which required to modify the Trigger code, Then Production instance doesn't allow you to modify the code of the Trigger.

We need to get the Code from the Production to the Sandbox. And update the code in the Sandbox. And re-Deploy to the Production instance.

To avoid this issue, We need to Write the Trigger Business Logic inside an Handler Class, and we can call the Handler Class Methods from the Trigger with the Help of Trigger Handler Factory.

If the Business Logic is exist inside the "Class", then we can re-use the Class Code in multiple places in the application.

Note: We can't remove the Trigger from the Production instance directly.

Step 1: We need to Get the Trigger Code from Production to "Sandbox".

Step 2: De-Active the Trigger in the "Sandbox". And move the De-Activated Trigger to the "Production".

Step 3: We can remove the De-Activated Trigger from the Production by using "Delete" button.

Handler Class Code:

```
/*
 * @ Author : Feroz Baig
 * @ Description : This Class Contains the Handler Methods / Business Logic,
 *                 to be referenced inside the Lead Triggers.
 * @ Version : 1.0
 * @ Created Date : 10/10/2019
 * @ Reference : AutoLeadConversion
 * @ Modified By :
 *      Modified By          Modified Date          Reason
 *      ------------------------------------------------------------------
```

```
 *  Pawan Kumar                 18/02/2018              Added the Code for Defect#1234
 */
public class LeadHandler
{
   /*
   *       @ Method Name : AfterUpdate
   *       @ Description : This method is used to convert the Bulk Lead Records     as Customers.
   *       @ Parameters:
   *                  @Param Name : lstLead                    @Type :  List<Lead>
   *       @ Returns : N/A
   *       @ Throws : DML Exceptions
   *       @ References : AutoLeadConversion
   *       @ Version : 1.0
   */

Public static void AfterUpdate(List<Lead> lstLeads)

  {
      // Get the Lead Record Status...
      LeadStatus lStatus = [Select id, MasterLabel, isConverted
                                    from LeadStatus
                                          Where isConverted = true
                                                Limit 1];

      // List to hold the Lead Records to Convert..
      List<Database.LeadConvert> lstLeadsToConvert = new List<Database.LeadConvert>();

      for(Lead ld : lstLeads)
      {
          if(ld.Status == 'Closed - Converted' && ld.IsConverted == false)
          {
              // Store the Lead Details into the LeadConvert Class..
              Database.LeadConvert lConvert = new Database.LeadConvert();

              // Specify the Lead Id...
              lConvert.SetLeadID(ld.Id);

              // Specify the Email Alert..
              lConvert.SetSendNotificationEmail(true);

              // Specify the option to generate Opportunity..
              lConvert.SetDoNotCreateOpportunity(ld.Do_Not_Create_Opportunity__c);

              // Specify the Lead Status...
              lConvert.SetConvertedStatus(lStatus.MasterLabel);
```

```
                // Add the Record to Collection to Convert Bulk Records..
                lstLeadsToConvert.Add(lConvert);
              }
            }

            // Convert the Lead Records...
            if(! lstLeadsToConvert.isEmpty())
            {
               Database.LeadConvertResult[] results = Database.convertLead(lstLeadsToConvert);
            }
          }
        }
```

Trigger Code:

```
        trigger AutoLeadConversion on Lead (After Update)
        {
          if(Trigger.isAfter && Trigger.isUpdate)
           {
             LeadHandler.AfterUpdate(Trigger.New);
           }
        }
```

/*
Write an Apex Trigger, to Synchronize the HiringManager Records into the corresponding Recruiter records and vice-versa.
***/**

CheckRecursive Class:

```
public class CheckRecursive
{
   private static boolean run = true;

   public static boolean runOnce()
   {
      if(run)
      {
                run=false;
                return true;
      }
    else
      {
          return run;
      }
   }
}
```

Hiring Manager Object Trigger:

```
trigger HiringManagerObjectTrigger on Hiring_Manager__c (After insert, after update)
{
   if(Trigger.isInsert && Trigger.isAfter)
   {
      HiringManagerHandler.AfterInsert(Trigger.New);
   }

   if(Trigger.isAfter && Trigger.isUpdate)
   {
      if(CheckRecursive.Runonce())
      {
          HiringManagerHandler.AfterUpdate(Trigger.NewMap);
      }
   }
}
```

Hiring Manager Handler Class:

```
/*
 * @ Author : Feroz Baig
 * @ Description : This Class is used to Synchronize the HiringManager Records into Recruiter Object.
 * @ Version : 1.0
 * @ Created Date : October 12, 2019
 * @ Reference : HiringManagerObjectTrigger
 * @ Modified By :
 *        Modified By              Modified Date              Reason
 *          ---------------------------------------------------------------
*
 */
public class HiringManagerHandler
{
   /*
   *      @ Method Name : AfterInsert
   *      @ Description : This Method is used to Create the Recruiter Records based
   *                                          on the Newly created Hiring Manager Records.
   *      @ Parameters:
   *              @Param Name : lstHrs                    @Type :  List<Hiring_Manager__c>
   *      @ Returns : N/A
   *      @ Throws : DML Exceptions
   *      @ References : HiringManagerObjectTrigger
   *      @ Version : 1.0
   */
```

```
Public static void AfterInsert(List<Hiring_Manager__C> lstHrs)
{
   if(! lsthrs.isEmpty())
   {
      List<Recruiter__C> lstRec = new List<Recruiter__C>();

      for(Hiring_Manager__c hr : lstHrs)
      {
         Recruiter__C rec = new Recruiter__C();

         rec.Name = hr.name;
         rec.Email_Id__c = hr.Email_Id__c;
         rec.Contact_Number__c = hr.Contact_Number__c;
         rec.Location__c = hr.Location__c;

         rec.HiringManager__c = hr.id;

         lstRec.add(rec);
      }

      if(!lstRec.isEmpty())
      {
         insert lstRec;
      }
   }
}

Public static void AfterUpdate(Map<Id,Hiring_Manager__c> hrsToUpdate)
{
   if(!hrsToUpdate.isEmpty())
   {
            List<Recruiter__c> lstRecruiters = [select id, name, email_id__C,
                     contact_number__C, location__C, HiringManager__C
                              from Recruiter__C
                              Where HiringManager__c IN : hrsToUpdate.keySet()];

      if(! lstRecruiters.isEmpty())
      {
         List<Recruiter__c> recruitersToUpdate = new List<Recruiter__C>();

         for(Recruiter__C rec : lstRecruiters)
         {
            Hiring_Manager__c hr = hrsToUpdate.Get(rec.HiringManager__c);

            rec.Name = hr.Name;
```

```
            rec.Location__c = hr.Location__c;
            rec.Email_Id__c = hr.Email_Id__c;
            rec.Contact_Number__c = hr.Contact_Number__c;

            recruitersToUpdate.Add(rec);
          }

          if(! recruitersToUpdate.isEmpty())
          {
            update recruitersToUpdate;
          }
        }
      }
    }
}
```

Recruiter Object:

Trigger:

```
--------
trigger RecruiterTrigger on Recruiter__c (after Update)
{
        if(Trigger.isAfter && Trigger.isUpdate)
  {
    if(CheckRecursive.runOnce())
    {
        RecruiterHandler.AfterUpdate(Trigger.New);
    }
  }
}
```

Handler Class:

```
public class RecruiterHandler
{
  Public static void AfterUpdate(List<Recruiter__C> lstRecruiters)
  {
    if(!lstRecruiters.isEmpty())
    {
      // Get the HiringManager Record Id's..
      List<ID> hrIds = new List<ID>();
      for(Recruiter__C rec : lstRecruiters)
      {
        hrIDs.add(rec.HiringManager__c);
      }
```

```
        Map<id, Hiring_Manager__C> mapHrs = new Map<ID,Hiring_MAnager__C>([select id, name,
location__C, email_id__c,Contact_Number__C  from Hiring_Manager__C
                                                    where id in : hrIds]);
        if(! mapHrs.isEmpty())
        {
           List<Hiring_Manager__c> hrsToUpdate = new List<Hiring_Manager__c>();

           for(Recruiter__C rec : lstRecruiters)
           {
             Hiring_Manager__c hr =  mapHrs.Get(rec.HiringManager__c);

              hr.Name = rec.Name;
              hr.Email_Id__c = rec.Email_Id__c;
              hr.Location__c = rec.Location__c;
              hr.Contact_Number__c = rec.Contact_Number__c;

              hrsToUpdate.add(hr);
           }

           if(! hrsToUpdate.isEmpty())
           {
              update hrsToUpdate;
           }
        }

     }
   }
}
```

TRIGGER BEST PRACTICES

1. Trigger should support bulk records as well - Always use "For Loop" to process records inside the trigger

2. Always check for null pointer exceptions.

3. Always use try and catch block in the trigger to handle Exceptions.

4. Differentiate events with separate blocks to avoid usage of trigger.new and trigger.old

5. Do not use DML operations inside the for loop. Add them to the list and update/insert/delete the list out side the for loop.

6. Do not use SOQL and SOSL statements inside for loop.

7. Write the Proper Test Classes for the Trigger. And maintain minimum 1% of code coverage for each trigger. And Overall organization wide, we have to maintain 75% of code coverage while moving the code from sandbox to production environment.

8. Avoid recursiveness in triggers by putting the proper conditions.

Note: Recursive Triggers can be avoided by using the "Static Boolean" variables.

Do not execute the trigger on all the update. Only execute the trigger on specific update.

Here the trigger will get executed on update of stagename field only and will not trigger on other field's update

To avoid recursive use the below code:

```
if(Trigger.isUpdate && opp.stagename == 'Qualification'  && opp.stagename != Trigger.oldMap.get(opp.id).stagename)

  opptyIds.add(opp.Id);
```

Error Code:

```
if(Trigger.isUpdate && opp.stagename == 'Qualification')

    opptyIds.add(opp.Id);
```

9. As a Best Practice, it is Recommended to maintain only One Trigger per an Object. Because, We dont have the option in salesforce to control the execution order of triggers.

10. It is recommended to maintain Trigger Handler Factory.(i.e instead of writing the Trigger code inside the Trigger, we can write the code in a Class and call the class from the trigger.)

TRIGGER ORDER OF EXECUTION

1. The original record is loaded from the database (or initialized for an insert statement)

2. The new record field values are loaded from the request and overwrite the old values

3. All before triggers execute (TRIGGERS)

4. System validation occurs, such as verifying that all required fields have a non-null value, and running any user-defined validation rules (VALIDATIONS)

5. The record is saved to the database, but not yet committed

6. All after triggers execute

7. Assignment rules execute

8. Auto-response rules execute

9. Workflow rules executed (WORKFLOW)

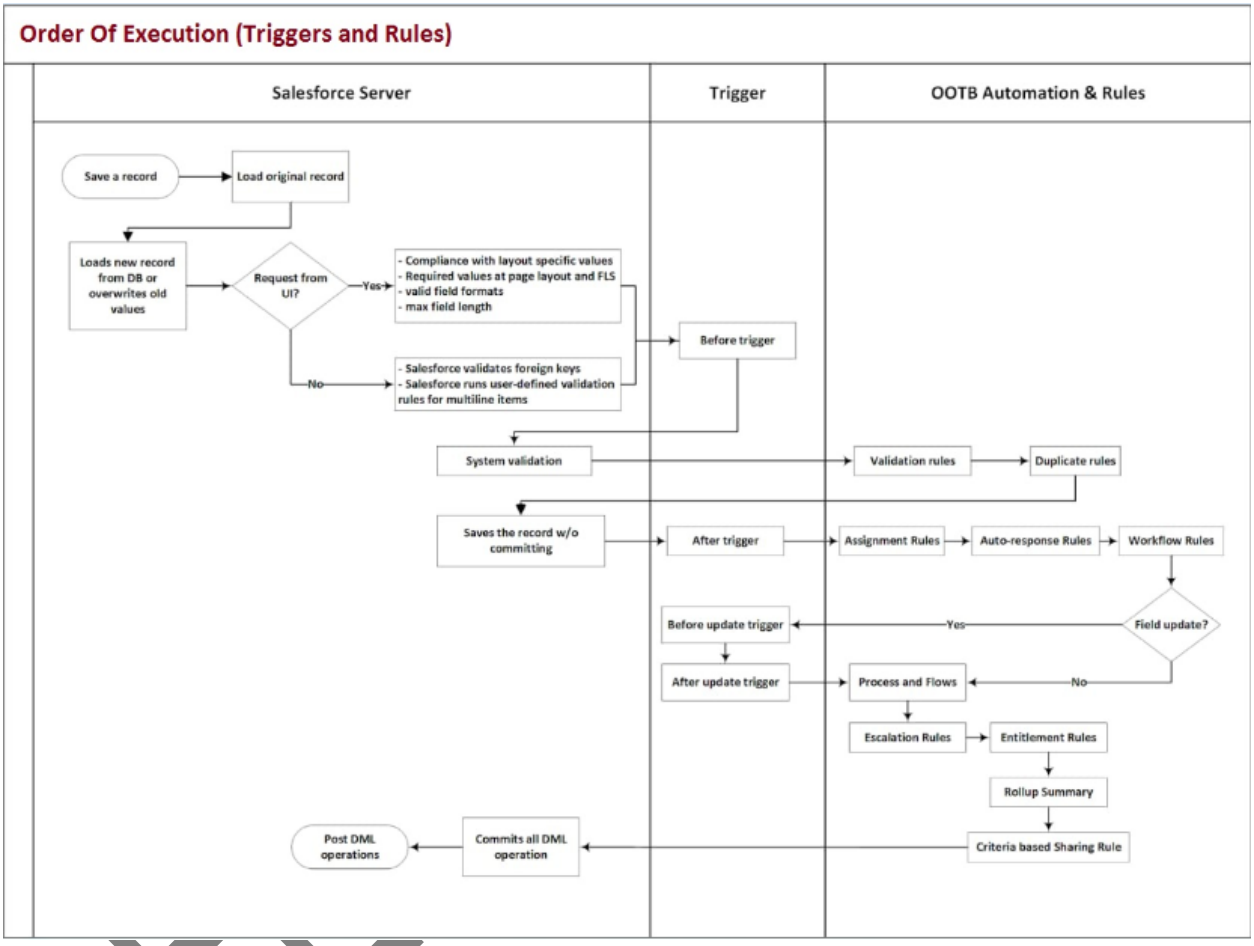
10. If there are workflow field updates, the record is updated again.

11. If the record was updated with workflow field updates, before and after triggers fire one more time (and only one more time) along with Standard Validation Rules. (Note: In this case, Custom validation rules will not fire.)

12. Escalation rules execute.

13. All DML operations are committed to the database

14. Post-commit logic executes, such as sending email.



EXCEPTION HANDLING

During the Application Development, each application should get compiled by using the Apex Compiler, And then it should get executed by using Apex Runtime.

Applications can cause the errors in below 2 cases.

1. Compile Time Errors:

These will get occurred upon compiling the source code by the Apex Compiler. Compilation errors will get occurred due to the below reasons.

1. Due to the syntactical mistakes inside the code.
2. Due to lack of system resources.

Compile time errors are Not Dangerous. As these will not cause any issues in the application. We need to resolve these compile time errors, by writing the code as per the syntactical rules given by the programming language.

2. Runtime Errors (Exceptions):

Upon executing the application, sometimes it will causes the errors called as "Runtime Errors". Which are called as "**Exceptions**".

Runtime Errors will get occurred because of the below reasons.

1. Due to Wrong implementation of Logic.
2. Due to wrong inputs supplied by the user.
3. Due to missing of values for the required fields.

An Exception is nothing but an undesired behavior of an application at run time. It is an unwanted event or situation, which disrupts the normal flow of execution.

When an exception occurred, then Apex stops the application execution. i.e. abnormal termination of the application will happened. It will stops the execution of the code immediately. And will not execute the remaining statements inside the program. i.e. it will halts all the DML processes and will not commit the results into the database.

In Apex programming, if an exception occurred, then salesforce will log the exception details into the "**Debug Log**" file, and it will send an email notification to the Currently running user regarding the Exception details along with the "Organization Id and User Id" .

We can track the exception details by using the Debug Log file.

Goto Setup menu → Goto "**Monitor**" menu in the Left panel.

1. Click on "**Logs**" and expand it.

2. Click on "Debug Logs" link.

We need to enable the "Debug Logs", by adding the required user details. And apply the required filters to track the exception messages.

Note: We can fetch the debug logs for the specified time period, which should be max of 24 hours.

Ex:

StartTime:
End Time:

The Time duration between the "Start Time" and "End Time" should be max. of 24 hours.

Types of Exceptions in Apex Programming:

Apex programming provides 2 types of Exceptions as below.

1. System Defined Exceptions / Built-in Exceptions.
2. User Defined Exceptions / Customized Exceptions.

System Defined Exceptions:

Apex provides a number of built-in exception types that the runtime engine throws if errors are encountered during execution.

Ex:

ListException
LimitException
DMLException
ArrayIndexOutOfBoundsException
MathException (i.e. DividedByZeroException)
QueryException
CalloutException
FileNotFoundException
NullPointerException

... etc.

// Below code causes the "DML Exception"

```
Account acc = new Account(); insert
acc;
```

// Below code causes the "Query Exception"

```
Account acc = [select id, name, rating from account];
```

// Below code causes the "NullPointer Exception"

```
string s; system.debug('Search for the Characters ...: '+
s.contains('ab'));
```

// Below code causes the "Math Exception"

```
system.debug('Division Result is...: '+ (234/0));
```

// Below code causes the "List Exception"

```
List<string> lstCountries = new List<string>{'India','USA'};
system.debug('Country Names are...: ');
system.debug(lstCountries[0]);  system.debug(lstCountries[1]);
system.debug(lstCountries[2]);
```

The Developer can handle the Runtime Exceptions caused by the application with the help of Exception Handling Mechanism. This can be achieved by using "Try, Catch, and Finally Blocks"

Try Block:

This block is used to specify the apex code, which may cause the Exceptions.

Ex: Place all the DML Operations Code, SOQL Query Code inside the TRY Block.

Catch Block:

This block is used to Catch the exceptions raised by the application in-order to make the graceful termination of the application.

Finally Block:

This block is used to specify the cleanup code. i.e. We can release the resources which has been used in the application. The finally block follows a try block or a catch block. A finally block of code always executes, irrespective of occurrence of an Exception.

Note: For All System defined Exceptions, the Base Class is "**Exception**"

Syntax:

```
Try
{
        // Write the Code, from which you are expecting  the exceptions
}
Catch(<ExceptionClass Name> <refName>)
{
        // Write the Code to Handle the Exception.
}
Finally
{
        // Write the code to Release the resources / Cleanup code.
}
```

Note: When an exception is occurred, then the corresponding method will creates an "Exception Object", and will throws the exception to the Apex Runtime. Then Apex runtime will search for the corresponding handling code and will execute it to make the graceful termination of the program.
We can have one or more "Catch Blocks" for a Try Block

Ex:

```
Try
{
        // Write the Code, from which you are expecting the exceptions
}
Catch(DMLException ex)
{
        // Handle the DMLException.
}
Catch(QueryException e)
{
        // Handle the QueryExceptions..
}
Catch(NullPointerException nEx)
{
        // Handle the NullPointerException..
}
Finally
{
        // Release the resources / Cleanup code..
}
```

Example: (Class Code):

```
public class ExceptionTestClass
{
   Public void DoMoreStuff()
   {
      Account acc;
      Try
      {
         acc = new Account();
            acc.Name = 'Sample';
         insert acc;

         system.debug('Division Results...: '+ (890/0));
      }
      Catch(DMLException ex)
      {
         System.debug('DML Exception occurred' );
      }
      Catch(QueryException e)
      {
         System.debug('Query Exception occured' );
      }
      Catch(NullPointerException nEx)
      {
         System.debug('NullPointerException occured' );
      }
      Catch(MathException exe)
      {
         system.debug('Something went wrong the application due to Math Operations.');
      }
      Finally
      {
         acc= null;
      }
    }
}
```

Execution:

ExceptionTestClass ex = new ExceptionTestClass(); ex.DoMoreStuff();

Exception Class Methods:

This class can handle any type of exceptions raised by the apex program.

1. **String GetMessage():** This method will return the exception message raised by the Apex program.
2. **String getStackTraceString():** This method is used to get the complete flow of the exception.
3. **Integer getLineNumber():** This method return the Line Number of the code, which causes the exception.
4. **String getCause():** This method returns the Exception Reason.
5. **String GetTypeName():** This method returns the Type of the exception.

Example Class Code:

```
public class ExceptionHandlerClass
{
   public static void HandleExceptions()
   {
      Account acc = new Account();

try
     {
          acc = [select id, name, rating, annualrevenue from Account];
system.debug('Account Records are....: '+ acc);
     }
     Catch(System.QueryException qEx)
     {
                system.debug('Query Exception has been occurred');
          system.debug('Exception Type is....: '+ qEx.getTypeName());
          system.debug('Exception Message is...: '+ qEx.getMessage());
          system.debug('Error Line Number is....: '+ qEx.getLineNumber());
          system.debug('Error Stack Trace String is....: '+ qEx.getStackTraceString());
          system.debug('Get Error Reason ....: '+ qEx.getCause());
     }
     Catch(Exception ex)
     {
        system.debug('General Exception Error Occurred.');
     }
finally
     {
```

```
            acc = null;
        }
    }
}
```

Execution Process:

```
ExceptionHandlerClass.HandleExceptions();
```

User Defined Exceptions:

Upon exception has been occurred, the Build-in exceptions will display the error message in the Technical Terms, Which can't be understandable by the user.

The developer can display the Custom Error Messages to the end user, upon raising an exception with the help of by creating the "Custom Exception Classes".

(Because, the user can't throw the Built-in exceptions. But he can Catch the Built-in exceptions)

By using Custom Error Messages, you can also specify detailed error messages and have more custom error handling in your catch blocks.

Note: To create your custom exception class, extend the built-in Exception class and make sure your class name ends with the word Exception.

Syntax:

```
Public Class <ClassName> extends Exception
{
    // Write the code..
}
```

Ex:

```
Public Class MyException extends Exception
{
}
```

Note: Upon creating the Custom Exceptions, always the class name should be ends with "Exception". Else Apex compiler will raises a compilation error.

Custom Exception Class:

```
public class AccountException extends Exception
{
}
```

Class Code:

```
public class ExceptionHandlerClass
{
   Public static void InsertAccount()
   {
        Account acc;
             Try
             {
                  acc = new Account();
                  insert acc;

             }
             Catch(DMLException ex)
             {
                  throw new AccountException('Exception Occured Upon Inserting Account Record');
             }
             Finally
             {
                  acc = null;
             }
        }
}
```

Execution:

```
ExceptionHandlerClass.InsertAccount();
```

SANDBOX IN SALESFORCE

During a development lifecycle, multiple tasks are carried out by separate teams. From developing an application to testing and releasing, tasks must follow a specific release process.

Some development and testing tasks are done in parallel, while other tasks depend on other deployments to be completed first.

Separating environments for these tasks (Development, Testing, Integration, and Staging) makes team development possible and helps ensure a high quality for the release.

You can obtain separate development and testing environments for the development lifecycle by creating sandbox organizations. Sandboxes contain a copy of your organization's metadata and sometimes its data too.

Note: Sandboxes are isolated from your production org, so operations that you perform in your sandboxes don't affect your production org.

- Sandbox provides a development environment for Salesforce applications.
- Allows us to development and Custom App building.
- Customizing the Business Process and workflows.
- Testing integrations and AppExchange Apps.
- Performance Testing / Functional Testing End user training.

We can create sandbox only from Production. Sandbox option available under Data Management in Production.

The number of sandbox's we can create depends on the license which we have.

There are 4 types of Sandboxes in Salesforce.

1. **Developer Sandbox.**
2. **Configuration only Sandbox (Developer Pro Sandbox)**
3. **Partial Copy Sandbox (Available since Winter'14)**
4. **Full Sandbox (which is exact copy of Production including data).**

Developer Sandbox:

- It is copy of production without data.
- Developer sandboxes are intended for developing code and testing.
- Multiple users can log into a single Developer sandbox.
- You can refresh a Developer sandbox once per day.
- Max. Storage Space is 200 MB.

Developer Pro / Configuration Only Sandbox:

- The main difference between this and Developer is the amount of data that can be stored. It also grabs some product data from production. If those two things are important, use this one.

- Developer Pro sandboxes copy all of your production organization's reports, dashboards, price books, products, apps, and customizations under Setup, but exclude all of your organization's standard and custom object records, documents, and attachments.
- Creating a Developer Pro sandbox can decrease the time it takes to create or refresh a sandbox from several hours to just a few minutes.
- You can refresh a Developer Pro sandbox once per day. Max. Storage space is 1 GB

Partial Copy Sandbox:

- Partial Data sandboxes include all of your organization's metadata and add a selected amount of your production organization's data that you define using a sandbox template.
- A Partial Data sandbox is a Developer sandbox plus the data you define in a sandbox template.
- Contains replica of Production and along with sample production data.
- We can create Templates to Load the Data from Production.
- We can refresh this sandbox for every 5 days.
- Storage space is up to 5 GB.

Full Sandbox:

- Full sandboxes copy your entire production organization and all its data, including standard and custom object records, documents, and attachments. You can refresh a Full sandbox every 29 days.
- While creating sandbox with data user have a choice of limiting data with Sandbox Templates, those sandbox's are called Partial data Sandbox's.
- You can refresh a Full Sandbox every 29 days.
- Size of this Sandbox is equals to Production Sandbox. Sandbox templates are only available for Partial Data or Full sandboxes.

Sandbox Limitations:

1. Unlimited Edition: Provides Licenses for

1 Full Sandbox

5 Developer Pro Sandbox

100 Developer Sandbox

1 Partial Copy Sandbox

2. Enterprise Edition: Provides Licenses for

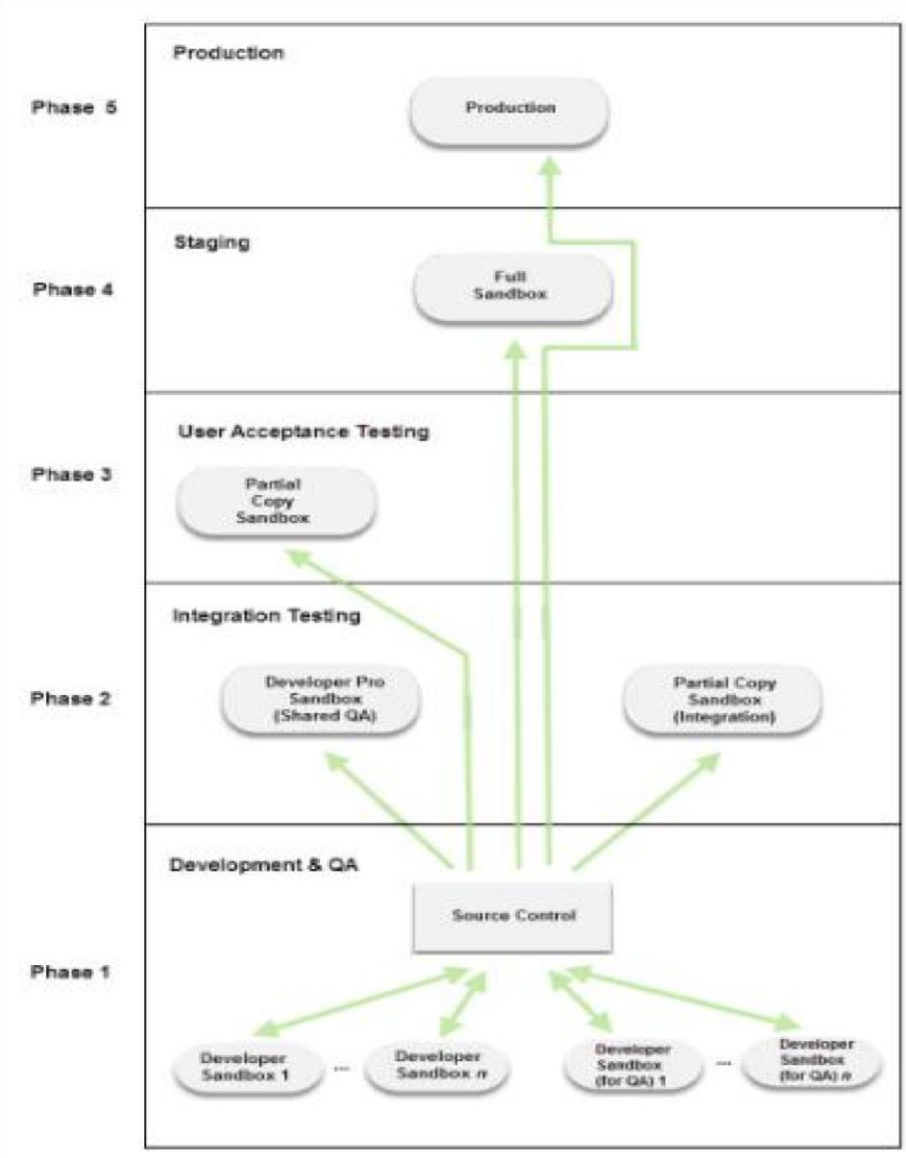
Provides 25 Developer Sandboxes

1 Partial Copy Sandbox

3. Professional Edition: Provides Licenses for

Provides 10 Developer Sandboxes (Changeset's are Not Available)

APPLICATION LIFECYCLE DIAGRAM



TEST CLASSES FOR UNIT TESTING

Upon implementing the application funcationalities, the developer will prepare a collection of Classes, Triggers, Batch / Schedule Classes, WebServices, API's, and VF Controller Classes to achieve the customer requirements.

Once the features has been implemented, we need to test the features to make sure, it should match the customer requirements.

It's a developer responsibility to perform the "Unit Testing" of the business logic.

Upon deploying the code components from Sandbox to Production, we have to maintain min. 75% of the quality code, to make sure it should not break any of the functionality in the cloud.

To achieve this, we need to prepare the "Automated Test Scripts" called as "Test Classes".

By using "Test Classes", we can test the application features automatically before moving to the production to maintain the "Quality Code" in the application.

Note: Upon moving the code from Sandbox to Sandbox, Code coverage is not mandatory.

But, upon moving the code from Sandbox to Production, we have to maintain an overall of 75% of the code coverage for the entire Organization.

Note: If the code coverage is less than 75%, then Production instance will not allowed to deploy the code.

Note:

Test Classes should be prepared for the below components.

1. Apex Classes
2. VF Controller Classes
3. Triggers
4. Batch and Schedule Classes
5. WebServices & API's.

We can verify the Organization's Code Coverage with the below navigation.

Click on "Setup" menu.

1. Goto "Build" menu in left panel.
2. Click on "Develop" and expand it.
3. Click on "Apex Classes" link.
4. Click on the hyperlink "Estimate your Organization Code Coverage".

Note:

We can verify the code coverage of each class / trigger/ etc, by opening the class code from the Standard Navigation.

Ex:

Build --> Develop --> Apex Classes

1. Open any of the Apex Class (Ex: CommonUtility)
2. Verify the Field : Code Coverage (Ex: 0%)

Note:

For each class / Trigger, we need to prepare a separate Test class.

Upon preparing the test class, we have to follow the below notations.

Ex:

Class Name: CommonUtility
Test Class Name: CommonUtilityTest

Class Name: AccountManagement
Test Class Name: AccountManagement__Test

Class Name : MathOperations
Test Class Name: Test__MathOperations

Test Class should be prepared by using the annotation **"@isTest**".
i.e. @isTest annotation, will indicates to the compiler that it is a Test Class, to test the application functionality.

Note:

Test class should be prepared with the Access Specifier either "Private / Public".

Syntax:

```
@isTest
<Private/Public> Class <TestClassName>
{
        // Write the Test Methods..
}
```

Ex:

```
@isTest
Public Class AccountManagementTest
{
        // Write the Testing Code..
}

@isTest
Private Class AccountTriggerTest
{
        // Write the Trigger Testing Logic..
}
```

Note:

We need to prepare a testmethod inside the "TestClass". Which will indicates the Starting point of the Application Testing.

Rules:

1. Test Method should be always "Static".
2. Test Method should not return any value to the calling environment.
   (i.e. Return Type should be always "Void")

3. Test method should be defined with the keyword "testmethod".
4. We can call / invoke the normal mathods from the "Test Method".

Syntax:

```
static testmethod void <MethodName>()
{
    // Write the Testing Code..
}
```

Ex:

```
@isTest
Public Class AccountManagementTest
{
    static testmethod void AccountsTest()
    {
        // Write Testing Code for "AccountManagement" Class
    }
}
```

Note: Test Class Code will not be counted as part of the "Apex Code Characters Limit".

Upon deploying the code from Sandbox to Production, along with the Clas Code, we need to deploy the associated "Test Class" also.

Note: We can run the test class by using "Run Test" button, available on the Test Class's Detailed Page.

MathOperations Class:

==========

Class Code:

```
public class MathOperations
{
    Public void Addition(integer x, integer y)
   {
     system.debug('Addition Result is....: '+ (x+y));
   }

   Public void Subtraction(integer x, integer y)
   {
     system.debug('Subtraction Result is....: '+ (x-y));
   }

   Public void Multiplication(integer x, integer y, integer z)
   {
     system.debug('Multiplication Result is....: '+ (x * y * z));
   }
}
```

Test Class:

```
@isTest
```

```
public class MathOperationsTest
{
     static testmethod void MathTest()
   {
      MAthOperations m = new MathOperations();

      m.Addition(200,450);
      m.Subtraction(4500,2300);
      m.Multiplication(100,23,45);
   }
}
```

DatabaseHelper Class:

==========

Class Code:

```
public class DatabaseHelper
{
   Public static void SearchContacts(string searchText)
   {
      if(searchText != null && searchText != '')
      {
         string contactsQuery = 'select id, firstname, lastname, title, email, phone, fax from Contact ';

       if(Pattern.matches('[a-zA-Z0-9.+_-]+@[a-zA-Z.-]+.[a-zA-Z]{2,4}', searchText))
        {
           system.debug('This is in Email Id format');
           contactsQuery += 'Where email =: searchText';
        }
        else
        {
           system.debug('This is a String value.');
           contactsQuery += 'where lastname =: searchText';
        }

        List<Contact> lstContacts =  Database.query(contactsQuery);
        system.debug('Collection size is...: '+ lstContacts.size());

        if(! lstContacts.isEmpty())
        {
           for(Contact con : lstContacts)
           {
              system.debug('Contact Record is..: '+ con);
           }
        }
      }
   }

   Public static void GetPositionsByLocation(string cityName)
   {
```

```
    if(cityName != null && cityName != '')
    {
       string posQuery = 'select id, name, location__C, close_date__C, maximum_budget__C from Position__C where
location__c =: cityName';

       List<Position__c> lstPositions = Database.query(posquery);

       if( !lstPositions.isEmpty())
       {
          for(Position__C pos : lstPositions)
          {
             system.debug('Position Record is...: '+ pos);
          }
       }
    }
  }

  Public static void GetAllPositionsByOffset()
  {
    List<Position__C> lstPositions = [select id, name, location__C,
                                close_date__C, maximum_budget__C,
                                 createddate
                                from Position__c
                                order by createddate desc
                                OFFSET 5];

    if(! lstPositions.isEmpty())
    {
       for(Position__C pos : lstPositions)
       {
          system.debug('Record is...: '+ pos);
       }
    }
  }

  Public static void GetAnOpportunityRecord()
  {
    Opportunity oppty =  [select id, name, amount, stagename, closedate
                  from Opportunity
              Limit 1];
    if(oppty != null)
    {
       system.debug('Opportunity Record is...: '+ oppty);
    }
  }
}
```

Test Class:

```
@isTest
```

```
public class DatabaseHelperTest
{
     static testmethod void TestDatabaseOperations()
   {
      Hiring_Manager__C hr = new Hiring_Manager__c(name = 'Sample HR', location__c = 'Hyderabad',email_id__C =
'samplehr@gmail.com', contact_number__c = '9988990000');
      insert hr;

      Position__C pos = new Position__C();
                  pos.Name = 'Sample Position';
                  pos.Location__c = 'Hyderabad';
                  pos.Close_Date__c = system.today().AddDays(30);
                  pos.Open_Date__c = system.today();
                  pos.Position_Status__c = 'New Position';
                  pos.Minimum_Pay__c = 1000000;
                  pos.Maximum_Budget__c = 1400000;
                  pos.Position_Description__c = 'Sample Position';
                  pos.Skills_Required__c = 'Testing, Apex, Salesforce CRM';
                  pos.Contact_Number__c = '9988998899';
                  pos.HiringManagerID__c = hr.id;
                  pos.Number_of_Positions__c = 3;
                  pos.RecordTypeId = Schema.SObjectType.Position__c.GetRecordTypeInfosByName().Get('IT Position
Record Type').GetRecordTypeID();
                  pos.Email_ID__c = 'sample@gmail.com';

                  insert pos;

      DatabaseHelper.GetPositionsByLocation('Hyderabad');
      DatabaseHelper.GetAllPositionsByOffset();

      // Testing the ContactSearch Method...
      Contact con = new Contact(firstname = 'Sample',lastname = 'Contact1', email = 'sample1@gmail.com',
phone='8899889999', title = 'Manager', pan_number__c='ALPPB9439E', passport_number__c = 'ALPPB8989E');
      insert con;

      DatabaseHelper.SearchContacts('Contact1');
      DatabaseHelper.SearchContacts('sample1@gmail.com');

      // Testing GetOpportunities() method..
      Opportunity oppty = new Opportunity(name='Sample Oppty', closeDate = system.today(), amount = 50000,
stagename='Prospecting');
      insert oppty;

      DatabaseHelper.GetAnOpportunityRecord();
   }
}
```

AutoLeadConversion Trigger:

==========

Trigger Code:

```
trigger AutoLeadConversion on Lead (After Update)
{
 if(Trigger.isAfter && Trigger.isUpdate)
  {
    LeadHandler.AfterUpdate(Trigger.New);
  }
}
```

Handler Class:

```
public class LeadHandler
{
  /*
  *  @ Method Name : AfterUpdate
  *  @ Description : This method is used to convert the Bulk Lead Records
       as Customers.
  *   @ Parameters:
  *    @Param Name : lstLead      @Type :  List<Lead>
  *   @ Returns : N/A
  *   @ Throws : DML Exceptions
  *   @ References : AutoLeadConversion
  *   @ Version : 1.0
  */
 Public static void AfterUpdate(List<Lead> lstLeads)
  {
    // Get the Lead Record Status...
    LeadStatus lStatus = [Select id, MasterLabel, isConverted
                  from LeadStatus
                    Where isConverted = true
                     Limit 1];

  // List to hold the Lead Records to Convert..
    List<Database.LeadConvert> lstLeadsToConvert = new List<Database.LeadConvert>();

    for(Lead ld : lstLeads)
    {
      if(ld.Status == 'Closed - Converted' && ld.IsConverted == false)
      {
        // Store the Lead Details into the LeadConvert Class..
        Database.LeadConvert lConvert = new Database.LeadConvert();

        // Specify the Lead Id...
        lConvert.SetLeadID(ld.Id);

        // Specify the Email Alert..
        lConvert.SetSendNotificationEmail(true);
```

```
            // Specify the option to generate Opportunity..
            lConvert.SetDoNotCreateOpportunity(ld.Do_Not_Create_Opportunity__c);

            // Specify the Lead Status...
            lConvert.SetConvertedStatus(lStatus.MasterLabel);

           // Add the Record to Collection to Convert Bulk Records..
           lstLeadsToConvert.Add(lConvert);
         }
      }

      // Convert the Lead Records...
      if(! lstLeadsToConvert.isEmpty())
      {
         Database.LeadConvertResult[] results = Database.convertLead(lstLeadsToConvert);
      }
   }
}
```

Test Class:

```
@isTest
public class AutoLeadConversionTest
{
     static testmethod void TestLeadConververionProcess()
   {
      List<Lead> lstLeads = new List<Lead>();

      for(integer counter = 1; counter <= 10; counter++)
      {
         Lead ld = new Lead(firstname = 'Sample',lastname = 'Lead -'+ counter ,company = 'Sample' ,email =
'sample@gmail.com', title = 'Manager', industry = 'Banking', phone = '9900990000', fax = '9900990000', status = 'Open
- Not Contacted');
         lstLeads.Add(ld);
      }

                  insert lstLeads;

      for(Lead ld : lstLeads)
      {
         ld.Status = 'Closed - Converted';
      }

      update lstLeads;
   }
}
```

Batch Class:

==========

Class Code:

```
Global class UpdateContactDetailsBatch implements Database.Batchable<SObject>
{
  Global Database.QueryLocator Start(Database.BatchableContext bContext)
   {
      string contactsQuery = 'select id, lastname, phone, mobilePhone, title, fax, pan_number__C, passport_number__c
from Contact';

      return Database.getQueryLocator(contactsQuery);
   }

   Global void Execute(Database.BatchableContext bContext, List<SObject> recordsToProcess)
   {
      if( ! recordsToProcess.isEmpty())
      {
         List<Contact> contactsToUpdate = new List<Contact>();

         for(SObject rec : recordsToProcess)
         {
            Contact con =(Contact) rec;

            con.Title = 'Marketing Head';
            con.Phone = '9988998899';
            con.MobilePhone = '9900990000';
            con.Fax = '9900000000';
            con.PAN_Number__c = 'ALPPB9090E';
            con.Passport_Number__c = 'ALHUJ3456E';

            contactsToUpdate.Add(con);
         }

         if(! contactsToUpdate.isEmpty())
         {
            //update contactsToUpdate;
            Database.update(contactsToUpdate, false);
         }
      }
   }

   Global void Finish(Database.BatchableContext bContext)
   {
     system.debug('Batch Job Id is....: '+ bContext.getJobID());

      AsyncApexJob jobDetails = [select id, status, totaljobitems,
                                 jobitemsprocessed, numberOferrors,
                                 createdby.email
                                from AsyncApexJob
```

```
                        where id =: bContext.GetJobId()];

      MEssagingHelper.SendBatchJobStatusNotification(jobDetails, 'UpdateContactDetailsBatch',0.0);
   }
}
```

Test Class Code:

```
----------
@isTest
public class UpdateContactDetailsBatchTest
{
     static testmethod void TestContactsBatch()
   {
      List<Contact> lstContacts = new List<Contact>();

      for(integer counter = 1; counter <= 100; counter++)
      {
         Contact con = new Contact();
                  con.FirstName = 'Test';
                  con.LastName = 'Contact - '+ counter;
                  con.Email = 'unittest@gmail.com';
                  con.Title = 'Manager';
                  con.Department = 'Sales';
                  con.Phone = '9900990000';
                  con.Fax = '9988999999';
                  con.PAN_Number__c = 'ALPPB9329E';
                  con.Passport_Number__c = 'ALYUIE456S';

         lstcontacts.Add(con);
      }

      insert lstContacts;

      // These methods are used to bypass the governor limits..
      Test.startTest();

                  UpdateContactDetailsBatch myBatch = new UpdateContactDetailsBatch ();
                  Database.executeBatch(mybatch);

      Test.stopTest();
   }
}
```

Future Method Class:

==========

Future Class:

```
-------------
Global class FutureMethodClass
{
   Public static void InsertHiringManager(string username)
```

```
    {
        Hiring_Manager__c hr = new Hiring_Manager__C();

        hr.Name = 'Raj Kumar';
        hr.Location__c = 'Hyderabad';
        hr.Contact_Number__c = '9900990000';
        hr.Email_ID__c = 'rajk@gmail.com';

        insert hr;

        if(hr.id != null)
        {
            system.debug('Hiring Manager has been Created. Record Id is...: '+ hr.id);

            DeActivateUSerRecord(username);


        }
    }

    @future()
    Global static void DeActivateUSerRecord(string userNameValue)
    {
        // User --> Setup Object.
        User userToDeActivate = [select id, username, isActive
                                 from User
                                 Where username =: userNameValue and
                                    isActive = true
                                 Limit 1 ];
        if(userToDeactivate != null)
        {
            userToDeactivate.IsActive = false;

            update userToDeActivate;
        }
    }
}
```

Test Class:

```
----------
@isTest
public class FutureMethodClassTest
{
     static testmethod void TestFutureMethods()
    {
        User userRecord = [Select id, profile.id from User where username = 'training.batch51@cis.com' limit 1];

        /*
         * This method is used to execute the Test class based on the specified
         *  user's profile level permissions.
```

```
     *
     * By using this method, we can avoid the MIXED_DML_OPERATION Exception
     * in the Test Class.
     */
     system.Runas(userREcord)
     {
        // Get the Profile..
        Profile p = [select id from Profile where name = 'CIS Platform Profile' LIMIT 1];

        // Create a User Record..
        User u = new User(Alias = 'standt', Email='standarduser@test.com',
           EmailEncodingKey='UTF-8', LastName='Unit Testing', LanguageLocaleKey='en_US',
           LocaleSidKey='en_US', ProfileId = p.Id,
           TimeZoneSidKey='America/Los_Angeles', UserName='st2018user@test.com');
        insert u;

        FutureMethodClass.InsertHiringManager('st2018user@test.com');
     }
   }
}
```

Schedule Class:

==========

Seconds Minutes Hours Day_of_month Month Day_of_week Optional

```
//For every 5 minutes
    String sch = '0 5 * * * ? '

System.Schedule('Schedule Name', <BatchClassObjectName>,  <CronExpression>);
```

CODE MIGRATION (DEPLOYMENT)

Code Migration:

The code components implemented by the developers / Administrators, as part of the application requirement to be get moved / deployed to the other instances of the landspace.

Ex:

Developer Sandbox ---> QA Sandbox --> UAT Sandbox --> Stage --> Production.

Code Components: Apps, Tabs, Objects, Fields, Validation Rules, Workflow Rules, Apex classes, Visualforce Pages, Triggers, etc.

Note: Upon deploying the code from sanbox to production, we should have min 75% of code coverage.

To deploy the code from one instance to another, we have the below features.

1. Change Sets (Salesforce)
2. Eclipse IDE
3. Force.com Migration Tool (ANT Migration - ANT Scripts)
4. Version Controlling Tools
    - 4.1. TFS (Team Foundation Server) - Microsoft
    - 4.2. SVN (Sub Version)
    - 4.3. GitHub
    - 4.4. Source Tree
    - 4.5. Perforce
    - 4.6. CVS
5. Release Management Tools.
    - 5.1. AutoRabit's Release Management Tool
    - 5.2. Bit Bucket

ChangeSets:

It is a feature in salesforce, used to do the Code Migration. By using ChangeSets, we can migrate the code from one instance to another instance.

ChangeSets feature will be available in only Licensed Editions. (Not in Free Developer Edition)

Note: We need to login into sandboxes by using "https://test.salesforce.com".

Note: All sandboxes are available in "CS" server (CS - Customer Sandbox)

We can able to add a collection of code components to the "ChangeSets", that can be deployed to the other instances (Sandbox / Production).

Note: This option is available under "Deploy" menu in Left panel.

We have 2 types of changesets.

1. Outbound ChangeSet: Used to move the changes to target organizations.
2. Inbound ChangeSet: changes are coming into the organizationss

Note: Before migrating the code from one instance to another, the organizations should be connected organizations.

Creating Outbound ChangeSet:

Login into "Developer Sanbox"

1. Click on "Setup" menu.
2. Goto "Deploy" menu in left panel.
3. Click on "Outbound ChangeSet" link.
4. Click on "New" button, to create a New Outbound ChangeSet
5. Enter the ChangeSet Name, and the Description.

   Ex:

   Name: Requirement1405ToQA_V1.0

   Description: This changeset includes the code components has been implementd as part of the requirement "Requirement ID : 1405".
6. Click on "Create" button.
7. Add all the required components to be migrated to the target instance.
8. Click on "Upload" button.
9. Select the "Target Organization / environment (QA)" by using "Radio Button".

VISUAL FORCE PAGES

Limitations in Configuration:

1. By using Page Layout Customization, We can arrange the fields in either One / Two Columns inside the section.

We can't place the fields in more than 2 columns. To avoid this problem, we have to use "Visualforce Programming".

2. We can't customize the Tab's Home Page. To customize the Tab's Home Page, based on the customer needs we need to design the Visualforce page and to be visible upon click on the "Tab".

3. We can't Change the Background Color / Style /Text in the Page Layouts. To make the the user interface look like attractive, we have to go with "Visualforce Programming".

To overcome the above limitations, we have to use "**VisualForce Programming**".

VisualForce is a Web Development framework, used to design dynamic and attractive user interfaces. (Because, We can't customize all the features exist in the standard page layout.)

By using VisualForce, we can design our own customized user interfaces.

VisualForce supports all the Standard Web Technologies exist in the market like HTML, DHTML, CSS, Java Script, VB Script, JScript, JQuery, Extension JS, Knockout JS, Backbobe JS, Angular JS, Flash Scripting, etc.

Salesforce provides a set of readymade tags, to design the custom webpages. VisualForce Tags look like as HTML Tags.

VisualForce contains all the tags as "Dependent Tags". i.e. Each tag should have its corresponding ending tag.

Visual Force Supports XML Style of Syntax. Visual Force supports MVC Development Style.

Note: VisualForce Tags are not case sensitive.

Visualforce supports the Code-Behind Technique, in which the USer Interface Part code will be separated into the different file. And Business Logic Code will be separated into the different file.

By using this Approach, we can reduce the Development Time and the Cost. And we can provide the Code Security.

(User Interface Code (Visualforce Code) --> .vfp)
(Business Logic Code (Apex Code) --> .apxc)

VisualForce provides the Real-Time Development style. Which allows us to Write, Compile, Execute and View the results in the same window.

MVC Architecture:

MVC is popular as it isolates the application logic from the user interface layer and supports separation of concerns.

The Model-View-Controller (MVC) is an architectural pattern that separates an application into three main logical components: the model, view, and the controller.

Each of these components are built to handle specific development aspects of an application.

MVC is one of the most frequently used industry-standard web development framework to create scalable and extensible projects.

Visual Force supports MVC Architecture for application development.

Model : It describes, how the data has been organized in the backend. Model represents the Data Structure with the help of Database Tables.

Ex: Standard Objects, Custom Objects, Fields and Relationships

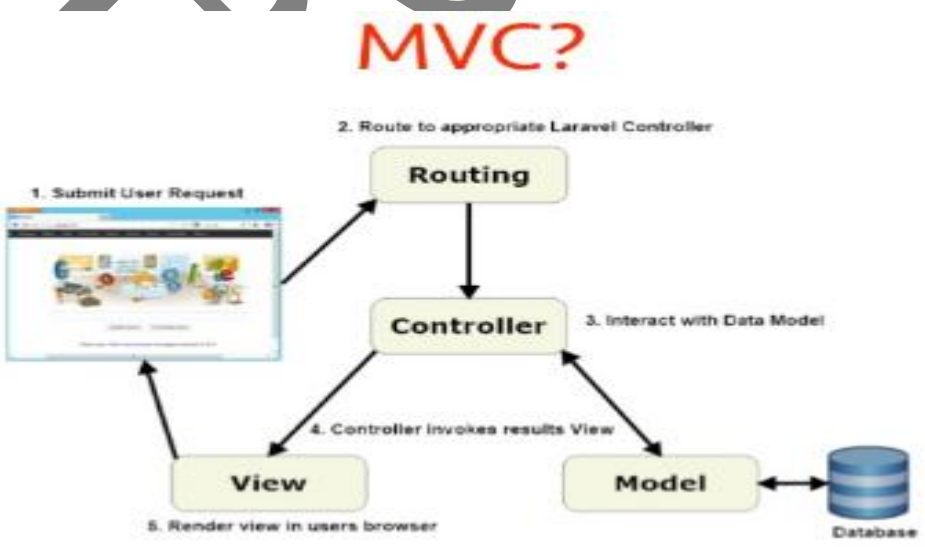
View: It represents the Presentation / Front End / View part of the application. It describes, how the Data is represented in the User Interface.

Ex: Page Layouts, Tabs --> Standard User Interfaces. VF Pages --> Custom User Interfaces

Controller: Controller is used to perform the actions on the objects and represent the data and results in the user interface. This is also referred as "Business Logic"

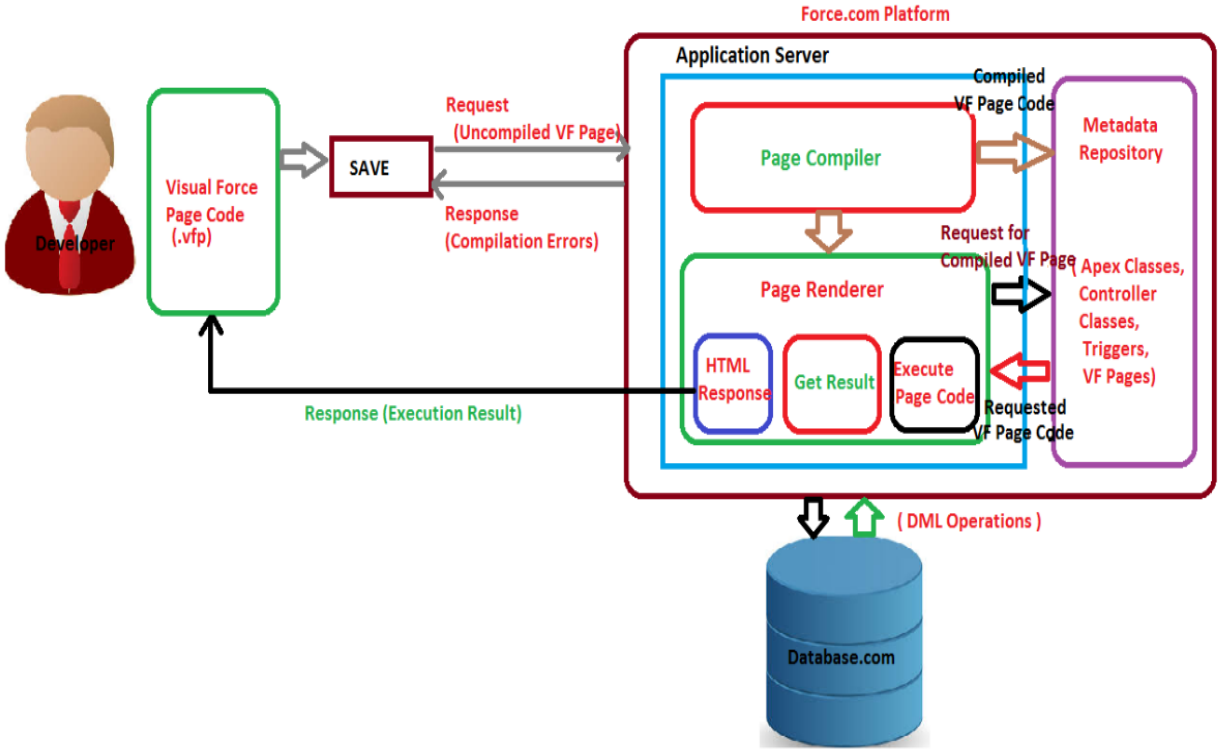
Ex: Standard Business Logic: Validation Rules, Workflow Rules, Approval processes, etc.

Custom Business Logic: Apex Classes, Trigger, SOQL, SOSL, etc.



Visual Force Page Execution Life Cycle:

Each Visualforce PAge code has to be get Compiled by using "**Page Compiler**" and it should get executed with the help of "**Page Renderer**" inside the Application Server.



VisualForce Components:

Each visualforce page contains two components.

1. **VisualForce Markup:**

- It contains a set of VisualForce Tags, has been used to design the user interface.
- Each VisualForce Tag is a Dependent Tag, i.e. each starting tag should requires an ending tag.
- VisualForce Tags are not case sensitive.
- Each VisualForce Page Tag should be prefixed with the word "apex".

Ex: <apex:page>
....
</apex:page>

Each Tag contains one or more attributes, which are used to supply the input values.

2. VisualForce Controller:

- It contains the corresponding business logic, using by the visualforce page controles.
- It can be implemented by using "Apex, SOQL, SOSL, DML" statements.

Note: A visualforce page can reference one or more controller classes.

Visual Force Page Controls Hierarchy:

```
Page (<apex:page>)
    |
    |---> SectionHeader (<apex:SectionHeader>)
    |
     ---> Form Control(<apex:form>)
            |
             --> PageBlock (<apex:PageBlock>)
                    |
                    | --> PageBlock Buttons(<apex:pageblockButtons>)
                    |            |
                    |             ---> Command Buttons (<apex:CommandButton>)
                    |
                     --> PageBlockSection (<apex:pageblocksection>)
                           |
                           | ---> Input Controls, Output Controls
                           |
                           | ---> PageBlockSectionItem (<apex:pageblocksectionitem>)
                                    |
                                    | ---> Input Controls
                                    |
                                     ---> Output Controls
```


Ways to Create a Visualforce Page:

Salesforce provides the below 4 ways to design the Visualforce Page.

1. **Standard Navigation**:

   Click on "Setup" menu.

   1. Goto "Build" menu in left panel.
   2. Click on "Develop" and expand it.
   3. Click on "Visualforce Pages" link.
   4. Click on "New" button, to create a New Page.
   5. Enter the Page Name inside the "Textbox"
   6. Write the code in the Editor.
   7. Click on "Save / QuickSave" button, to save the code.

Note: When the user click on "Save" button, to VF page code will send to the Force.com platform and will save the compiled code into the "Metadata Repository".

8. Click on "Preview" button, to run the code and to see the results.

2. **Developer Console:**

Click on your Name and expand it.

1. Click on "Developer Console" link.
2. Click on "File" menu.
3. Click on "New" menu item.
4. Click on "Visualforce Page" sub menu item.
5. Enter the Visualforce Page Name, to be get Created.
6. Click on "Ok" button. It will creates a New Page.
7. Write the Code inside the Page provided by the Salesforce.
8. Use "CTRL + S" to save the code.

Note: It provides an autosave option, which will send the VF page code to the force.com platform for each 10 fraction of seconds.

9. Click on "Preview" button, to Execute the Code.

Ex:

```
<apex:page >
    Welcome to Visualforce Programming. This is my First Visualforce Page.
    <br/><br/>
    <b> I have designed this Page by using Developer Console.</b>
</apex:page>
```

3. **URL Format:**

By using URL Format, we can create the Visualforce page as below.

Goto the Address Bar, and Prepare the Page as below.

Syntax: https://<instancenumber>.salesforce.com/apex/<pageName>

Ex: https://ap5.salesforce.com/apex/SamplePage2

Note: Upon creating the Page by using "URL Format", we need to enable the "**Development Mode**" checkbox.

Click on your name and Expand it.

1. Click on "My Settings" link.
2. Goto "Personal" option and expand it.
3. Click on "Advanced User Details" link.
4. Click on "Edit" button.
5. Select the Checkbox "Development Mode".
6. Click on "Save" button.

Ex:

```
<apex:page >

        Welcome to Visualforce Programming. <br/> <br/>

        <b> This is my Second Visualforce Page, designed by using URL Format.
        </b>
</apex:page>
```

4**. Eclipse IDE / Visual Studio Code Editor.**

VISUALFORCE CONTROLLERS

Controller defines the Business Logic Part, Which will act as the mediator between View (User Interface) and Model (Database). Each Visualforce page will reference a Controller, to invoke the actions.

Controller will receive the request from the User Interface, and will perform the necessary actions on the Database objects and will represent the data and results in the user interface.

VisualForce provides, 3 types of controllers. Which are used to implement the business logic. And can be invoked by using visualforce controls (Ex: Buttons, Hyperlinks, Picklists, etc).

Salesforce provides a readymade class for each object (Standard / Custom) with the same name as the class name as below.

OBJECT NAME	READYMADE CLASS NAME
Account	Account
Contact	Contact
Opportunity	Opportunity
Lead	Lead
Case	Case
Position__c	Position__c
Customer__C	Customer__C
Candidate__C	Candidate__C
…….	……

Note: In each class, salesforce provides a set of readymade methods can be invoked by using the Button on the USer Interface (PageLayout / Visualforce Page).

Calling the Method: Each variable and methods exist inside the class will be referenced as a "Merge Field"

Syntax:

{!<VariableName>} → Ex: {! AccountName}, {! Industry}
{!<MethodName>} → Ex: {! InsertAccount}, {! SearchAccount}
{!<objectName>} → Ex: {! lstAccounts} , {! lstContacts}

Readymade Methods:

1. Save(): It will save the record into the associated object, and will re-direct the user to the detailed page of the record.

Ex: Save() method → {!Save}

2. QuickSave(): This method is used to save the record inside the associated object.

Ex: {!QuickSave}

3. **Cancel():** This method is used to cancel the previous operation.

   Ex: {!Cancel}

4. **Delete():** This method is used to delete the selected record from the object.

   Ex: {!Delete}

Pagination Methods:

5. **First():** This method will return the Set of records to be displayed upon click on "First" button during the pagination.

   Ex: {!First}

6. **Next():** This method is used to display the next set of records, upon click on "Next" button.

   Ex: {!Next}

7. **Previous():** This method is used to display the previous set of records.

   Ex: {!Previous}

8. **Last():** This method is used to display the Last Page record, upon click on "Last" button.

   Ex: {!Last}

9. **HasPrevious():** This method returns a boolean indicator, which is used to navigate the user to previous page records.

   Ex: {! hasPrevious}

10. **hasNext():** This method returns a Boolean indicator, which is used to navigate the user to next page records.

    Ex: {!hasNext}

Upon designing the Visualforce page, we need to reference either a Standard class provides by salesforce or we can use our own custom class also.

To indicate the Controller class to be referenced in the visualforce page, Salesforce provides 3 types of controllers.

1. Standard Controller:

- It describes the visualforce page referencing the Standard class methods provided by salesforce by default.
- We can refer either a "Standard Object class or a Custom Object class".
- We can invoke all the methods exist inside the class from the Visualforce Page.
- We have to indicate the Controller class name to be referenced by the Visualforce page, we have to use an attribute called as "StandardController" in the Page Tag.

Syntax:

```
<apex:Page StandardController = "<StandardClassName>" >
    ....
    ....
</apex:page>
```

Ex:

```
<apex:page StandardController="Account" >
    ....
</apex:page>

<apex:page StandardController="Position__C" >
    ...
</apex:page>
```

Note: In a Visualforce page, we can reference only one "Standard Controller class".

2. Extension Controller:

- Upon designing the Visualforce page, we can use standard methods.
- If the existing standard functionalities are not sufficient, then we can extend the business logic, by referencing our own class methods inside the same page.

  Ex: Standard Save() method, will not verify the duplication check. So, instead of using standard Save() method, I want to use my own "CustomSave()" method to save the record along with the "Standard methods".
- We have to specify the Extension class with the attribute "Extensions".

Syntax:

```
<apex:page standardController= "StandardClassName" Extensions="Custom Class Name" >
    .....
</apex:page>
```

Note: A Visualforce page can reference one or more extension controller classes.

Ex:

```
<apex:page standardController= "StandardClassName"  Extensions="CustomClass1, CustomClass2,..." >
    .....
</apex:page>
```

Note: Extension Controller should be always used along with either "Standard Controller or custom controller".

3. Custom Controller:

This attribute is used to specify the "Custom Controller" class, which is referencing inside the visualforce page.

Syntax:

<apex:page controller="<Custom Class Name>">

...

</apex:page>

Ex:

```
<apex:page controller="CustomAccountPageController">
    ...
</apex:page>
```

VISUALFORCE PAGE TAGS

VisualForce provides a set of readymade tags, to be used upon designing the visualforce pages based on the customer / business needs.

Page Tag : <apex:Page> :

This tag is used to design a visualforce Page, which can be used to place one or more visual appearance controls.

Note: This tag should be the "Root" tag in the VisualForce page. A visualforce page can have only one "<apex:page>" tag.

Attributes:

1. **Setup : Boolean** → This attribute is used to indicate, whether the visualforce page can be displayed along with the Setup" options in the left panel or not. Syntax:

   Setup = "true" --> Setup Menu will be displayed

   Setup = "false" --> Setup menu will not be displayed

   Ex:

   ```
   <apex:page setup="true">
       ...
   </apex:page>
   ```

2. **SideBar : Boolean** → This attribute is used to indicate, whether the visualforce page can be displayed along with the "Sidebar" options in the left panel or not.

   Syntax:

   SideBar = "true" --> SideBar Menu will be displayed

   SideBar = "false" --> SideBar menu will not be displayed

   Ex:

   ```
   <apex:page sidebar="false">
       ...
   </apex:page>
   ```

3. **ShowHeader : Boolean** → This attribute is used to indicate the Salesforce header options to be visible on the page or not.

Syntax:

```
<apex:page ShowHeader="true / False" >
    ....
</apex:page>
```

4. **TabStyle = "ObjectName"** → This attribute is used to specify the object name, whose tabstyle needs to be applied to the page.

Syntax:

```
<apex:page TabStyle="Account / Contact / Position__c" >
    ....

</apex:page>
```

5. **StandardController = "<ClassName>"** → This attribute is used to indicate the Standard controller class to be referenced by the visualforce page.

Syntax:

```
StandardController = "Account"
StandardController = "Position__c"
```

Ex:

```
<apex:page standardcontroller="Account" >
    ...
</apex:page>
```

Note: This VF page can invoke all the methods exist in the "Account" class provided by salesforce.

6. **Controller = "Custom Class Name"** → This attribute is used to indicate the custom / user defined class name, to be referenced by the visualforce page.

Ex:

```
<apex:page controller="AccountManagementController" >
    ...

</apex:page>
```

7. **Extensions = "Extension Controller Class"** → This attribute can be used along with either "StandardController / Custom Controller".

This attribute is used to indicate the extension class to be referenced by the visualforce page along with Standard classes.

Ex:

```
<apex:page StandardController="Account" Extensions="<AccountExtensionClass>">
                    ...
</apex:page>
```

8. **RenderAs="pdf"** → This attribute is used to display the VF page content in a "pdf" file format.

```
Ex:   <apex:page renderAs="pdf">
            ...
      </apex:page>
```

9. **Action="<MethodName>"** → This attribute is used to specify the method name to be invoked upon loading the VF page.

```
Ex:   <apex:page action ="{!CreateNewAccount}">
            ...

      </apex:page>
```

10. **Readonly = "Boolean"** → This attribute is used to display more than 1,000 records in the VF page.

Note:

- IF the Readonly="True", then we can display max. of 10,000 record on the VF page.
- If the Readonly = "False", we can display max. of 1,000 records on the page.

```
Ex:   <apex:page readonly="true">
            ...

      <apex:page>
```

11. **ContentType** → This attribute is used to display the VF page content as a word document.

Syntax: contentType="application/msword#<FileName>.doc"

Ex: contentType="application/msword#MyContent.doc"

SectionHeader Tag: (<apex:SectionHeader>):

This tag is used to define the SectionHeader for the VF page. By using this tag, we can specify the Title and Subtitle for the page.

Note: This tag has be placed inside the <apex:page> tag.

Attributes:

Title = "TitleName"

Subtitle = "SubTitleName"

Syntax: <apex:sectionHeader title="Account Edit" subtitle="New Account"/>

Form Tag (<apex:form>):

This tag is used to place all the input or output control's inside the VF page. It is a container control, Which contains the other controls inside that.

Note: A VF Page can have one or more "FORM" tags.

Syntax:

```
<apex:form>
    ... // All Input Controls...
</apex:form>
```

PageBlock Tag (<apex:PageBlock):

This tag is used to divide the VF page into multiple blocks. In each block we can display the different content.
Note: A VF Page can have one or more PageBlocks.

Attributes:

1. **Title = "string"** → This attribute is used to specify the title for the page block.
   Ex:
   title ="Account Records" title ="Contacts"
2. **Rendered = "Boolean"** → This attribute is used to indicate, whether the pageblock to be displayed on the VF page or not.
   Ex:
   rendered = "true" → Display the pageBlock
   rendered = "fale" → Will not display.

Ex:

```
<apex:pageBlock title="PageBlock Title" rendered ="true">
    ...
</apex:pageblock>
```

PageBlockButtons: (<apex:pageBlockButtons>):

This tag provides a placeholder to place the required buttons on the VF page, to invoke the action methods.

Attributes:

1. **Title**: Title attribute is used to specify the title information for the buttons section.

2. **Location**: (TOP / Bottom / Both): This attribute is used to indicate the location of the button controls.

Note: Default location value is "Both"

Note: This tag should be placed inside the "PageBlock" tag.

Syntax:

```
<apex:pageblockButtons title="My Buttons" location="top">
.... // Define the Button Controls...
</apex:pageblockButtons>
```

CommandButton: (<apex:commandButton>):

This tag is used to place the commandButtons on the visualforce page. A VF Page can have one or more commandButtons.

Attributes:

1. **Value**:This attribute is used to specify the caption for the button.

2. **Action**: This attribute is used to specify the method name, which needs to be invoke upon click on the button.

3. **Rerender**: This attribute is used to indicate the component name, to be refreshed after performing the action.

Syntax:

```
<apex:commandButton value="Save" action="{!Save}" rerender="<ComponentID>"/>

<apex:commandButton value="Cancel" action="{!Cancel}"/>

<apex:commandButton value="Search" action="{!DoSearch}"/>
```

PageBlockSection: (<apex:pageblocksection>):

This tag is used to divide the PageBlock into multiple sections. And we can place a set of controles inside each section.

A PageBlock can have one or more PageBlockSections. PageBlockSection tag has to be placed inside the "PageBlock" tag.

Attributes:

1. Title: String → This attribute is used to specify the title for the "PageBlockSection".

2. Columns: Integer → This attribute is used to specify an integer value, which indicates the number of columns to be used to place the fields.

3. Collapsible: Boolean → This attribute is used to indicate that the pageblocksection to be displayed with "Collapsible" option or not.

Syntax:

```
<apex:pageblocksection title="Account Information" columns="3" Collapsible="true">

    .... Place the fields...

</apex:pageblocksection>
```

PageBlockSectionItem (<apex:PageBlockSectionItem>):

This tag is used to divide the PageBlockSection into multiple sub sections. In Each PageBlockSectionItem, we can place max. of 2 controls. This tag has to be placed inside the "PageBlockSection" tag.

Syntax:

```
<apex:pageblocksectionitem>
    ...
</apex:pageblocksectionitem>
```

Q: Rendered Vs. ReRender Vs. RenderAs

Rendered:

- This Attribute is used to indicate, whether the object/ page/component should be display or not.
- This is a Boolean Flag, Which contains wither TRUE / FALSE.
  - TRUE - Indicates the display the page/component
  - FALSE - Indicates Don't display the page/component.

RenderAs:

- This attribute is available in Page Tag.
- This is used to display the Visual Force Page as a "pdf"

```
Ex:  <apex:page renderAs="pdf">
        .....
     </apex:page>
```

ReRender:

- This attribute is available for the object/command buttons
- Used to Refresh the specified objects, instead of the whole page.

```
Ex: <apex:commandButton rerender ="<componentname1>,< componentname2>">
```

.....

</apex:commandButton>

Input Text : (<apex:inputText>):

This tag is used to provide an input textbox on the user interface. The user can provide the input value to the application by using this Tag. This tag can be used to bind the salesforce object fields also.

Properties:

1. Label="To Specify the Label to be visible"
2. html-Placeholder = "To Specify the Watermark Text"
3. Rendered = "Boolean"
4. Value ="{! <VariableName>}"

This property will bind the textbox value to the specified variable.

Syntax: <apex:inputText Label=”Label Text” value="<Field Name to Bind>"/>

Ex:

```
<apex:inputtext value="{!Position__c.Name}"/>
<apex:inputtext value="{!Account.Name}"/>

string customerName = 'RameshKumar';
<apex:inputtext value="{!customerName}"/>
```

Input Field : (<apex:inputfield>):

This tag is used to bind the field of an object. It provides the visualapperance on the Page, based on the Binded field lable, Style, Field Type and the values.

i.e this tag will get the field properties along with the values(If it is a Picklist / Date type). **Syntax:**

```
<apex:inputfield value="{!<ObjectName.FieldAPIName>}"/>
```

Properties:

1. Value = "{! <ObjectName>.<FieldName>}"
2. Rendered = "Boolean"
3. Required = "Boolean" --> To make the field Mandatory.

Ex:

```
<apex:inputfield value="{!Account.Rating}"/>
<apex:inputfield value="{!Account.Name}"/>
<apex:inputfield value="{!Position__c.Name}"/>
<apex:inputfield value="{!Position__c.Location__C}" required="true"/>
```

Usecase:

Create a VisualForce Page to Perform the Operations on Position object.

1. Upon Click on the "Save" button, record has to be get inserted into the object.

2. Upon Click on "Cancel" button, it should cancel the current operation.

```
<apex:page standardController="Position__c" showHeader="true" sidebar="false" setup="false"
tabStyle="Hiring_Manager__c" >

  <apex:sectionHeader title="Position Edit" subtitle="New {!$ObjectType.Position__c.Label}"/>

      <apex:form >
        <apex:pageblock title="Position Page Block">
          <apex:pageblockButtons title="Buttons" location="both">
            <apex:commandButton value="Save" action="{!Save}" />
            <apex:commandButton value="Save & New" action="{!Save}"/>
            <apex:commandButton value="Cancel" action="{!Cancel}"/>
            <apex:commandButton Value="Delete" action="{!Delete}"/>
          </apex:pageblockButtons>

          <apex:pageblockSection title="Position Information" collapsible="true" columns="2">
      <!-- Commented Code...
            <apex:inputtext value="{!Position__c.Name}"/>
            <apex:inputtext value="{!Position__c.Location__c}"/>
           -->

            <apex:inputfield value="{!Position__c.Name}"/>
            <apex:inputfield value="{!Position__c.Location__c}" required="true"/>
            <apex:inputfield value="{!Position__c.Position_Status__c}"/>
            <apex:inputfield value="{!Position__c.Number_Of_Positions__c}"/>
            <apex:outputText ></apex:outputText>
            <apex:outputText ></apex:outputText>
            <apex:inputfield value="{!Position__c.HiringManagerId__c}"/>
            <apex:inputfield value="{!Position__c.Share_Record_To__c}"/>
          </apex:pageblockSection>

          <apex:pageblockSection title="Compensation Details" collapsible="true" columns="2">
            <apex:inputfield value="{!Position__c.Minimum_Pay__c}"/>
            <apex:inputfield value="{!Position__c.Maximum_Pay__c}"/>
```

```
    </apex:pageblockSection>

    <apex:pageblockSection title="MileStone Details" columns="2">
      <apex:inputfield value="{!Position__c.Open_Date__c}"/>
      <apex:inputfield value="{!Position__c.Close_Date__c}"/>
    </apex:pageblockSection>

     <apex:pageblockSection title="Additional Information" columns="2">
      <apex:inputfield value="{!Position__c.Passport_Required__c}"/>
      <apex:inputfield value="{!Position__c.Travel_Required__c}"/>
      <apex:inputfield value="{!Position__c.Website_del__c}"/>
      <apex:inputfield value="{!Position__c.Email_Id__c}"/>
</apex:pageblockSection>

    <apex:pageBlockSection title="Position Description" columns="1">
      <apex:inputfield value="{!Position__c.Position_Description__c}"/>
      <apex:inputfield value="{!Position__c.Skills_Required__c}"/>
    </apex:pageBlockSection>

  </apex:pageblock>
</apex:form>
</apex:page>
```

Testing the Page:

1. Go to the VisualForce Page Enter the details in the field and click on "Save" button.

Observation: It should save the record into the object anb will re-direct the user to "Detailed" page of the record.

2. **Use the "Page Parameters":**

We can use the same VisualForce page, to load an existing record as below.

Syntax:

https://c.ap2.visual.force.com/apex/CustomPositionPage?id=<recordID>

Ex: https://c.ap2.visual.force.com/apex/CustomPositionPage?id=a012800000ahCv8

Observation: It should load the details of the specified record id.

Overring the Standard Buttons:

Salesforce provides a set of button controls for each object by default like New, Edit, Save, Delete, Cancel, Accept, etc.

For each button, salesforce assigned a standard page layout. Which will be displayed upon click on the button.

We can override the standard buttons by using "Custom VisualForce pages" as below.

UseCase: Create a Visualforce Page and override the "New" button of "Position" object.

Goto Setup → Click on "Positions" tab.

1. Expand the "Force.com Quick Access Menu".
2. Click on "Vew Object" link.
3. Goto "Buttons, Links, And Actions" section.
4. Goto "New" button.
5. Click on "Edit" link for New Button.
6. Select the radio button "VisualForce Page"
7. Select the Required VisualForce Page from the Picklist
8. Select the "Checkbox"- Skip RecordType Selection (IF needed)
9. Click on "Save" button.

Testing the Feature:

Goto **Positions** Tabs

1. Click on "New" button.
2. It should display your Custom VisualForce Page.

Assignments:

1. **Design a Visualforce Page with the required sections, to Insert the Lead Records in the Object.**
2. **Design a Visualforce page with the required sections and controls, to Manage the Account Records in the object.**

Detail Tag (<apex:detail>):

This tag is used to design the detailed page of an object. Which can be used to display the record details in the required format.

Properties:

1. title = "Boolean"
   This Property is used to enable / disable the Title of the record.

2. InlineEdit = "Boolean"
   This property is used allow the InlineEditing feature on the detailed page or not.

3. RelatedList = "Boolean"

This property is used to display the RelatedList on the detailed page or not.

4. ShowChatter = "Boolean"

This property is used to decide whether the Chatter Feed should be get visible on the Detailed page or not.

Ex:

```
<apex:detail inlineedit = "true" title="true"
                    ShowChatter="false" RelatedList="false"/>
```

Note: The "Detail" tag should be get placed outside of the "Form" tag.

Usecase:

Design the Detailed Page for the Candidate Object with the required sections.

VF Page Code:

```
<apex:page standardController="Candidate__c" setup="false" sidebar="false" showheader="true"
tabStyle="Candidate__c" >
    <chatter:feedWithFollowers entityId="{!Candidate__c.id}"/>
    <br/><br/>
  <apex:detail inlineEdit="true" relatedList="true" title="false" showChatter="false"/>
</apex:page>
```

Testing the Page:

Syntax:

Goto the URL and change the URL as below.

https://ap5.salesforce.com/apex/<PageName>?id=<recordid>

Ex: https://ap5.salesforce.com/apex/CandidateEditPage?id=a037F00000CpM6R

RelatedList (<apex:RelatedList>):

In the detailed page, if the relatedList's are not enabled. Then we can display the required associated child records with the help of "RelatedList" tag.

By using RelatedList tag, we can display only one Child object records at a time.

Properties:

1. List = "RelatedListName"

We need to specify the associated related list name.

For each child object, salesforce provides a seperate RelatedList name by default.

Child Object Name	RelatedList Name
Contact	Contacts
Opportunity	Opportunities
Case	Cases
OpenActivity	OpenActivities
Activity History	ActivityHistories
Notes & Attachements	CombinedAttachments
Approval Steps	ProcessSteps
...	
Position__c	Positions__r
Customer__c	Customers__r

2. PageSize = "Integer"

This propertys used to indicate, how many number of records to be get visible per page inside the relatedList.

Syntax:

```
<apex:RelatedList list="<RelatedListName>" pagesize="Number"/>
```

Ex:

```
<apex:RelatedList list="Opportunities" pagesize="10"/>
<apex:RelatedList list="Candidates__r" pagesize="5"/>
```

Usecase:

Design a Visualforce page to display the Detailed information of the Account Record with the associated RelatedLists.

VF Page Code:

```
<apex:page standardController="Account" setup="false" sidebar="false" showheader="true" tabStyle="Solution" >
    <apex:detail inlineEdit="true" relatedList="false" title="true" showchatter="true"/>

    <apex:relatedList list="Cases" pageSize="5"/>
    <apex:relatedList list="Contacts" pageSize="2"/>
    <apex:relatedList list="CombinedAttachments" pageSize="5"/>
    <apex:relatedList list="ProcessSteps" pageSize="5"/>
    <apex:relatedList list="OpenActivities" pageSize="4"/>
    <apex:relatedList list="Hiring_Managers__r" pageSize="4"/>
</apex:page>
```

Testing the Page:

https://c.ap5.visual.force.com/apex/AccountsDetailedPage?id=0017F00000Kr8iL

Tab Control (<apex:TabPanel>):

This tag is used to design the Tab Panel control on the visualforce page.

Syntax:

```
<apex:TabPanel>
        ...
        ...
        ...
</apex:TabPanel>
```

Tab (<apex:Tab>):

This tag is used to define an indepent tab, to be get placed inside the Tab Panel.

Syntax:

```
<apex:Tab Label="Name" Name = "Tab Name">
        ....
        .... Desing the user interface
        ....
</apex:Tab>
```

Note: Inside a TabPanel, we can place one or more Tabs. Each tab can have its own user interface, to fire some actions.

Usecase:

========

Design a visualforce page, to display the Detailed Page Of an Account Record.
All the RelatedLists should be visible in the form of "Tabs". And Override the Standard view button with the New VF Page.

VF Page Code:

```
<apex:page standardController="Account" setup="false" sidebar="false" showHeader="true" tabStyle="Position__c"
>

  <apex:tabpanel switchtype="ajax" >
    <apex:Tab label="Account Info" name="accDetails">
      <apex:detail inlineEdit="true" relatedList="false" title="true" showChatter="true"/>
    </apex:Tab>

    <apex:Tab label="Contacts" name="accContacts">
      <apex:relatedList list="Contacts" pageSize="5"/>
    </apex:Tab>

    <apex:Tab label="Opportunities" name="accOppty">
      <apex:relatedList list="Opportunities" pageSize="10"/>
    </apex:Tab>

    <apex:Tab label="Cases" name="accCases">
      <apex:relatedList list="Cases" pageSize="5"/>
```

```
    </apex:Tab>

    <apex:Tab label="Activities" name="accOpenActivity">
      <apex:relatedList list="OpenActivities" pageSize="5"/>
    </apex:Tab>

    <apex:Tab label="Notes & Attachments" name="accAttach">
      <apex:relatedList list="CombinedAttachments" pageSize="5"/>
    </apex:Tab>
  </apex:tabpanel>
</apex:page>
```

Note: Override the Standard "View" button of Account object with the "Newly Created Visualforce Page".

Custom Controller:

Upon using the standard business logic inside the visualforce page, in few cases it won't suits to our requirement.

To achieve this, we need to create our own classes to implement the corresponding business logic based on the need.

And can be referenced from the visualforce page with the "Controller" attribute. We have to use the Custom controller, upon referencing the "Custom Apex Class" inside the VF page.

Syntax:

```
<apex:Page Controller="<Custom ClassName>" >
  ...
  ...
</apex:Page>
```

To achieve this, We need to follow the below steps.

Step 1: Create your own Controller Class based on the need.

Ex:

```
Public Class AuthenticateUser
{

}
```

Step 2: Define the required variables inside the class.
Note: The variables, Collections, and objects which can be accessible from the "Visualforce" page, should be defined with "GET & SET" accessors.

GET --> Used to access / Read the value of the variable from the VF page.

SET --> Used to Set / Assign / Write the value to the variable from the VF Page.

Note: GET & SET Accessors are comes under the "Properties".

Ex:

```
Public Class AuthenticateUser
{
        Public string userName {get;set;}
        Public string passWord {get;set;}
        Public List<Account> lstAccounts {get;set;}
        Public Customer__C cust {get;set;}

        Public void ValidateUserCredentials()
        {
                // Write the Business Logic, to Authenticate the User.
        }
}
```

Step 3: Reference the Controller Class inside the VF Page by using "Controller Attribute".

Ex:

```
<apex:Page Controller="AuthenticateUser">
        ...
        ...
</apex:Page>
```

Step 4: Invoke the required methods by using the Button Controls as below.

Ex:

```
<apex:Page Controller="AuthenticateUser">

        <apex:CommandButton value="Login" action="ValidateUserCredentials"/>

</apex:Page>
```

/*

Create a Visualforce Page, to Show the Student Details by Creating a Custom "Student Class".

***/**

VF Page Code:

```
<apex:page controller="StudentsController" >

  Student Code is ....: {!studentId} <br/>
  Student Name is.....: {!studentName} <br/>
  Fee Amount is.......: {!feeAmount} <br/>
  Student Address is..: {!address}

</apex:page>
```

Controller Class:

```
public class StudentsController
{
   Public string studentID{get;set;}
   Public string studentName{get;set;}
   public string address {get;set;}
   Public decimal feeAmount {get;set;}

   // Defining a Default Constructor...
   Public StudentsController()
   {
      studentId = '10001';
      studentName = 'Ramesh Kumar';
      feeAmount = 6000;
      address = 'Road #5, Banjara Hills, Hyderabad.';
   }
}
```

Note: Upon Loading a Visualforce page, it will creates the object for the corresponding referenced "Custom Controller Class" by default. So, the Default constructor of the class will get fired automatically.

To assign the pre-defined values to the class variable, we have to write the code inside the "Default Constructor".

**/*
Design a Visualforce Page. Upon Loading the Page, It should create a Hiring Manager Record automatically.
*/**

VF Page Code:

```
<apex:page controller="CreateHiringManagerController" action="{!CreateHRRecord}" >
   {!result}
</apex:page>
```

Controller Class:

```
public class CreateHiringManagerController
{
   Public string result{get;set;}

   Public void CreateHRRecord()
   {
      // Create a Hiring Manager Record..
      Hiring_Manager__C hr = new Hiring_Manager__C();

      hr.Name = 'Suresh Sahoo';
      hr.Location__C = 'Hyderabad';
      hr.Email_id__C = 'suresh@gmail.com';
      hr.Contact_Number__c = '9900990000';

      insert hr;
```

```
        if(hr.id != null)
        {
            result = 'Hiring Manager Record Inserted with the ID ...: '+ hr.id;
        }
    }
}
```

Note: Visualforce doesn't allows us to perform the "DML Operations" inside the Constructor of the Class. But the Constructor allows us to write the SOQL Queries. To perform the DML operations, we have to use the user defined methods.

**/*
Design a visualforce page to implement a Calculator.
*/**

VF Page Code:

```
<apex:page controller="BasicCalculatorController" setup="false" sidebar="false" showheader="true"
tabStyle="Position__c" >
<apex:sectionHeader title="Calculator" subtitle="Basic Calculator"/>
<apex:form >
   <apex:Pageblock title="Sample Calculator">

      <apex:pageblockButtons location="bottom" >
         <apex:commandButton value="Addition" action="{!Addition}" rerender="res"/>
         <apex:commandButton value="Subtraction" action="{!Subtraction}" rerender="res"/>
         <apex:commandButton value="Multiply" action="{!Multiplication}" rerender="res"/>
         <apex:commandButton value="Division" action="{!Division}" rerender="res"/>
      </apex:pageblockButtons>

      <apex:PageblockSection title="Calculator" columns="1" collapsible="true">
         <apex:inputtext label="Enter First Number" html-placeholder="First Number" value="{!firstNumber}"/>
         <apex:inputtext label="Enter Second Number" html-placeholder="Second Number"
value="{!secondNumber}"/>
         <apex:outputLabel id="res" >The Result is.....: {!result} </apex:outputLabel>
      </apex:PageblockSection>
   </apex:Pageblock>
</apex:form>
</apex:page>
```

Controller Class:

```
public class BasicCalculatorController
{
   Public integer firstNumber {get;set;}
   Public integer secondNumber {get;set;}
   Public integer result {get;set;}
```

```
    Public void Addition()
    {
        result = firstNumber + secondNumber;
    }

    Public void Subtraction()
    {
        result = firstNumber - secondNumber;
    }

    Public void Multiplication()
    {
        result = firstNumber * secondNumber;
    }

    Public void Division()
    {
        if(secondnumber > 0)
        {
            result = firstNumber / secondNumber;
        }
    }
}
```

Error Messages:

Upon performing the operations through the visualforce page, if any errors occurred. Then it will display the Error Message in the Technical Terms, which can't be understandable by the End user.

We can display our own customized user friendly error message on the visualforce page. To achieve this we have to follow the below Steps.

Step 1: Make the VF Page capable to Show Error Messages.

By default, VF Page is not capable to display the error messages on the Page.

We can give the ability to the Page to show the error messages, by adding "**PageMessages**" tag.

Syntax:

```
<apex:PageMessages id="refName"> </apex:PageMessages>
```

Step 2: Display the Custom Error Messages through Apex.

Based on the need, we can prepare an Error Message throgh Apex Programming. And we will display the message on the page as below.

Syntax:

```
ApexPages.AddMessage(new ApexPages.Message( <Error Message Icon>, 'Error Message String'));
```

Visualforce Provides the below 5 Error Icons.

1. ApexPages.Severity.ERROR
2. ApexPages.Severity.INFO
3. ApexPages.Severity.CONFIRM
4. ApexPages.Severity.WARNING
5. ApexPages.Severity.FATAL

Ex:

ApexPages.AddMessage(new ApexPages.Message(ApexPages.Severity.Confirm, 'operation performed Successfully.'));

/*
Design a visualforce page to implement a Calculator, By adding the Custom Error Messages.
***/**

VF Page Code:

```
<apex:page controller="BasicCalculatorController" setup="false" sidebar="false" showheader="true"
tabStyle="Position__c" >
<apex:sectionHeader title="Calculator" subtitle="Basic Calculator"/>
<apex:form >

   <apex:pagemessages id="errors"></apex:pagemessages>

   <apex:Pageblock title="Sample Calculator">

      <apex:pageblockButtons location="bottom" >
         <apex:commandButton value="Addition" action="{!Addition}" rerender="res, errors"/>
         <apex:commandButton value="Multiply" action="{!Multiplication}" rerender="res, errors"/>
         <apex:commandButton value="Division" action="{!Division}" rerender="res, errors"/>
      </apex:pageblockButtons>

      <apex:PageblockSection title="Calculator" columns="1" collapsible="true">
         <apex:inputtext label="Enter First Number" html-placeholder="First Number" value="{!firstNumber}"/>
         <apex:inputtext label="Enter Second Number" html-placeholder="Second Number"
value="{!secondNumber}"/>
         <apex:outputLabel id="res" >The Result is.....: {!result} </apex:outputLabel>
      </apex:PageblockSection>
   </apex:Pageblock>
</apex:form>
</apex:page>
```

Controller class:

```
public class BasicCalculatorController
{
   Public integer firstNumber {get;set;}
   Public integer secondNumber {get;set;}
   Public integer result {get;set;}

   Public void Addition()
   {
      result = firstNumber + secondNumber;
```

```
    }

    Public void Multiplication()
    {
        result = firstNumber * secondNumber;
    }

    Public void Division()
    {
        if(secondnumber > 0)
        {
            result = firstNumber / secondNumber;

            ApexPages.AddMessage(new ApexPages.Message(ApexPages.Severity.Confirm, 'Operation Completed
Successfully.'));
        }
        else
        {
            ApexPages.AddMessage(new ApexPages.Message(ApexPages.Severity.Warning, 'Second Number Should be
always greaterthan Zero.'));
        }
    }
}
```

PageBlockTable (<apex:PageBlockTable>):

This tag is used to represent the records information in the Tabular format.

This tag contains the default Table format and will iterate the specified collection from Starting to Ending automatically.

Note: It is a Child Component to the "PageBlock" tag.

Properties:

1. Value="{!<collectionName / DataSource>}"
   This property is used to specify the collection name / datasource, where the actual records get resides.

2. Var="referenceName"
   This property is used to specify the referencename, in which each record should get resides.

Ex:

```
List<Account> lstAccounts = [select id, name, rating, industry from Account];

<apex:PageBlockTable value="{!lstAccounts}" var="acc">
        ...
        ...
</apex:PageBlockTable>
```

Column Tag (<apex:Column>):

Upon representing the records in the Tabular format, we need to define one or more columns inside the Table.

To define a column in the Table, we have to use "Column" tag. Which should be a Child component to the "PageBlockTable".

Properties:

1. HeaderValue="Caption for the Column"

This property is used to specify the column Header. This is an optional property.

If the user didn't specify this property, then the binded field's "Field Lable" will be considered as the "Column Header".

Ex:

HeaderValue = "Account Rating"

HeaderValue = "Action"

2. Value = "{! <refName.FieldName>}"

This property is used to bind a specified field to the column. So that it will display the field values inside the column.

Ex:

value="{!acc.Rating}"

Value="{!acc.Pan_Number__C}"

Ex:

```
<apex:PageBlockTable value="{!lstAccounts}" var="acc">
  <apex:column headerValue = "Account ID" value="{!acc.id}"/>
  <apex:column value="{!acc.Name}"/>
  <apex:column value="{!acc.Rating}"/>
  <apex:column value="{!acc.industry}"/>
</apex:PageBlockTable>
```

/*

Design a visualforce page, to represent all the Position Records in the Tabular format upon loading the Page.

***/**

VF Page Code:

```
<apex:page controller="ShowPositionsController" setup="false" sidebar="false" showHeader="true"
tabStyle="Solution" >
<apex:sectionHeader title="Positions" subtitle="All Positions"/>
<apex:form >
   <apex:pageblock title="All Positions ( {!lstPositions.size} )">

      <apex:PageblockTable value="{!lstPositions}" var="pos">
         <apex:column value="{!pos.Name}"/>
         <apex:column value="{!pos.Location__c}"/>
         <apex:column value="{!pos.Position_Status__c}"/>
```

```
        <apex:column value="{!pos.Maximum_Budget__c}"/>
        <apex:column value="{!pos.Close_Date__c}"/>
        <apex:column value="{!pos.Email_ID__c}"/>
      </apex:PageblockTable>

   </apex:pageblock>
</apex:form>

</apex:page>
```

Controller Class:

```
public class ShowPositionsController
{
   Public List<Position__c> lstPositions {get;set;}

   // Defining the Default Constructor..
   Public ShowPositionsController()
   {
      lstPositions = new List<Position__C>();

      lstPositions = [select id, name, location__c, close_date__C, position_status__c,
         Maximum_budget__c, email_id__c
            from Position__c];
   }
}
```

/*
Design a visualforce page, to Search for the Account Records based on the specified search text.
***/**

VF Page Code:

```
<apex:page controller="SearchAccountsController" sidebar="false" setup="false" showheader="true"
tabStyle="Hiring_Manager__c" >
<apex:sectionHeader title="Accounts" subtitle="Search Accounts"/>
<apex:form >
   <apex:Pageblock title="Search Accounts Block">
      <apex:pageBlockSection title="Search Accounts" columns="2" collapsible="false">
         <apex:inputtext label="Enter Search Text" html-placeholder="Account Name" value="{!searchText}" />
         <apex:commandButton value="Search Accounts" action="{!SearchAccounts}" rerender="pgBlock"/>
      </apex:pageBlockSection>
   </apex:Pageblock>

   <apex:pageblock title="Search Results ( {!lstAccounts.size} )" id="pgBlock">
      <apex:pageblockTable value="{!lstAccounts}" var="acc">
         <apex:column value="{!acc.Name}"/>
         <apex:column value="{!acc.Rating}"/>
         <apex:column value="{!acc.Industry}"/>
         <apex:column value="{!acc.Type}"/>
```

```
        <apex:column value="{!acc.Annualrevenue}"/>
        <apex:column value="{!acc.Active__c}"/>
      </apex:pageblockTable>
   </apex:pageblock>
</apex:form>
</apex:page>
```

Controller Class:

```
public class SearchAccountsController
{
   Public string searchText {get;set;}
   Public List<Account> lstAccounts {get;set;}

   Public void SearchAccounts()
   {
      lstAccounts = new List<Account>();

      if(searchText != null && searchText != '')
      {
         searchText += '%';

         lstAccounts = [Select id, name, rating, industry, annualrevenue,
                        type, active__C
                       from Account
                        Where name like : searchText];
      }
   }
}
```

OutputLink Tag (<apex:outputlink>):

This tag is used to prepare a Hyperlink inside the visualforce Page. And when the user click on the Hyperlink, we can re-direct the user to the specified URL.

Properties:

1. Value="Target Page URL"
   To specify the URL of the Page, to which the user should get re-direct.

   Ex: value="www.salesforce.com"

2. Target="_blank / _self / _New / _Search / _Parent"
   This attribute is used to specify the location, where the specified URL should get opened.
   (i.e. in a New Tab / in a New Window / in the Same Window /etc.)

Syntax:

```
<apex:outputlink value="www.salesforce.com" target="_blank">
        Click Here to Goto Salesforce.com
</apex:outputlink>
```

VF Page Code:

```
<apex:page sidebar="false" setup="false" showheader="true" tabStyle="Hiring_Manager__c" >

<apex:sectionHeader title="Accounts" subtitle="Search Accounts"/>
  <apex:form >
          <apex:pageblock title="Hyperlink Demo">
                  <apex:outputLink value="http://www.salesforce.com" target="_blank" >Salesforce.com
</apex:outputLink>
          </apex:pageblock>
  </apex:form>
</apex:page>
```

Usecase:

Design a Visualforce Page, to search for the Account Records and display the Account Name as the Hyperlink.

When the user click on the Account Name, then re-direct user to the Detailed Page of the Selected Account Record.

VF Page Code:

```
<apex:page controller="SearchAccountsController" sidebar="false" setup="false" showheader="true"
tabStyle="Hiring_Manager__c" >
<apex:sectionHeader title="Accounts" subtitle="Search Accounts"/>
<apex:form >
  <apex:Pageblock title="Search Accounts Block">
    <apex:pageBlockSection title="Search Accounts" columns="2" collapsible="false">
      <apex:inputtext label="Enter Search Text" html-placeholder="Account Name" value="{!searchText}" />
      <apex:commandButton value="Search Accounts" action="{!SearchAccounts}" rerender="pgBlock"/>
    </apex:pageBlockSection>
  </apex:Pageblock>

  <apex:pageblock title="Search Results ( {!lstAccounts.size} )" id="pgBlock">
    <apex:pageblockTable value="{!lstAccounts}" var="acc">
      <apex:column headerValue="Account Name" >
        <apex:outputLink value="/{!acc.id}" target="_blank" >{!acc.Name}</apex:outputLink>
      </apex:column>
      <!--  <apex:column value="{!acc.Name}"/> -->
      <apex:column value="{!acc.Rating}"/>
      <apex:column value="{!acc.Industry}"/>
      <apex:column value="{!acc.Type}"/>
      <apex:column value="{!acc.Annualrevenue}"/>
      <apex:column value="{!acc.Active__c}"/>
    </apex:pageblockTable>
  </apex:pageblock>
```

```
</apex:form>
</apex:page>
```

Controller Class:

```
public class SearchAccountsController
{
   Public string searchText {get;set;}
   Public List<Account> lstAccounts {get;set;}

   Public void SearchAccounts()
   {
      lstAccounts = new List<Account>();

      if(searchText != null && searchText != '')
      {
         searchText += '%';

         lstAccounts = [Select id, name, rating, industry, annualrevenue,
                        type, active__C
                       from Account
                        Where name like : searchText];
      }
   }
}
```

Extension Controller:

Upon designing the visualforce page, in few cases we need to use both "Standard Class Methods" and "Custom class Methods" also inside a single visualforce page.

To achieve this, we have to use "Extension Controller" inside the visualforce page.

By using Extension Controller, we can use all the "Standard Methods", and along with "Our Own Custom functionalities" inside the page.

To add our custom class inside the Page, we have to use "Extensions" property inside the Page tag.

A VisualForce page can reference one or more Extension Controller classes along with a "StandardController".

Syntax:

```
  Public Class AccountsExtensions
  {
          Public void ValidateAccounts()
          {
                  // Write the Business Logic..
          }
  }
```

```
<apex:Page StandardController="Account" Extensions="AccountExtension1, AccountExtension2,….">
        ...
        ...
</apex:Page>
```

/*

Design a VisualForce Page to Create the New Account Records.

1. **Needs to verify the Duplicate Account Records based on the Name.**
2. **Record should be created, if No Duplicates Found in the object.**
3. **Display the Error Message, if any Duplicate Records Found.**

***/**

VF Page Code:

```
<apex:page StandardController="Account" extensions="AccountsExtensionController" setup="false" sidebar="false"
showheader="true" tabStyle="Hiring_Manager__c" >
<apex:sectionHeader title="Account" subtitle="New Account"/>
<apex:form >
  <apex:pageblock title="Account Edit">

    <apex:pagemessages id="errors"> </apex:pagemessages>

    <apex:pageblockButtons location="both">
      <apex:commandButton value="Save" action="{!ValidateDuplicateAccounts}" rerender="errors"/>
      <apex:commandButton value="Cancel" action="{!Cancel}"/>
    </apex:pageblockButtons>

    <apex:pageblockSection title="Account Details" columns="2" collapsible="true">
      <apex:inputfield value="{!Account.Name}"/>
      <apex:inputfield value="{!Account.AccountNumber}"/>
      <apex:inputfield value="{!Account.Site}"/>
      <apex:inputfield value="{!Account.Rating}" required="true"/>
      <apex:inputfield value="{!Account.Industry}"/>
      <apex:inputfield value="{!Account.AnnualRevenue}"/>
      <apex:inputfield value="{!Account.Phone}"/>
      <apex:inputfield value="{!Account.Fax}"/>
      <apex:inputfield value="{!Account.Active__c}"/>
      <apex:inputfield value="{!Account.Type}"/>
    </apex:pageblockSection>
  </apex:pageblock>
</apex:form>
</apex:page>
```

Controller Class:

```
public class AccountsExtensionController
{
  Public Account acc{get;set;}

  public AccountsExtensionController(ApexPages.StandardController controller)
```

```
  {
      acc= (Account) controller.GetRecord();
  }

  Public void ValidateDuplicateAccounts()
  {
     integer recordsCount = [Select count() from account where name =: acc.name];

     if(recordsCount > 0)
     {
        ApexPages.AddMessage(new ApexPages.Message(ApexPages.Severity.Warning, 'Duplicate Records Found with
the Same Name.'));
     }
     else
     {
        insert acc;
        ApexPages.AddMessage(new ApexPages.Message(ApexPages.Severity.Confirm, 'Account Record Inserted
Successfully.'));
     }
  }
}
```

Pagination / Paging:

By default VisualForce page allows us to show max. of 1,000 records at a time.

When the user tries to display more than 1,000 records it will display the Error Message.

To show more than 1,000 records inside the VF page, we need to set the "ReadOnly property as TRUE", inside the Page Tag.

Ex:

```
  <apex:Page ReadOnly = "true" ...>
          ...
             ...
  </apex:page>
```

Now, this page will allows us to show max. of 10,000 records on the Page.

But, ReadOnly property will make the Page as "ReadOnly State", which prevents the DML operations on the page. (i.e. The user can't perform any DML operations on the records)

To avoid this problem, we need use "Pagination". By which we can divide all the result records into the various smaller peices called as "Pages".

Each Page contains a set of records to be get visible on the Page.

Each Page will be identified by using an index position starting from "1".

Note:
By using Pagination, we can avoid the Performance issues, and ViewState exceptions also.

By using Pagination, we can show more than 10,000 records on the page by dividing into various smaller peices.

To achieve the Pagination, Apex provides a class called as "StandardSetController".

This class provides a set of readymade methods, to be used to achieve the Pagination.

Syntax:
StandardSetController sController = new StandardSetController(Database.getQueryLocator([SOQL Query]));

Database.getQueryLocator():
This method is used to fetch more than 50,000 records by executing the SOQL Query.

This method will return upto 50 million records, and the resultant records will get placed inside the "Temporary Memory" inside the application server.

And it will return the starting memory address to the StandardSetController class.

StandardSetController Class Methods:

1. SetPageSize(<Integer>):
   This method is used to indicate the number of records to be get displayed per page.

   Ex:
   ```
   SetPageSize(5);
   SetPageSize(25);
   ```

2. GetRecords():
   This method will get the current page records and will display on the VF page.

   Ex:
   ```
   sController.getRecords();
   ```

3. ResultSize():
   This method will returns an integer, which indicates the number of records has been returned by the SOQL Query.

4. First():
   This method will get the First Page records and display inside the PageBlockTable.

5. Next():
   This method will returns the Next Page Records.

6. Previous():
   This method will returns the Previous Page records.

7. Last():
   This method will returns the Last Page Records.

8. HasPrevious():
It returns TRUE, if we have the previous page records. Else it returns FALSE.

9. HasNext():
It returns TRUE, if we have the Next page records. Else it returns FALSE.

10. PageNumber():
This method will returns the Current Page Index Number.

/*
Design a VisualForce Page, to display the Account Records by using Pagination.
***/**

VF Page Code:

```
<apex:page controller="AccountsPaginationController" setup="false" sidebar="false" showheader="true"
tabStyle="Hiring_Manager__c" >
<apex:sectionHeader title="Accounts" subtitle="Accounts Pagination"/>
<apex:form >
  <apex:pageblock title="Accounts PageBlock" id="pgBlock">

    <apex:pageBlockTable value="{!Accounts}" var="acc">
      <apex:column headerValue="Account Name">
        <apex:outputLink value="/{!acc.id}" target="_blank">{!acc.name}</apex:outputLink>
      </apex:column>
      <apex:column value="{!acc.Rating}"/>
      <apex:column value="{!acc.industry}"/>
      <apex:column value="{!acc.AnnualRevenue}"/>
      <apex:column value="{!acc.Type}"/>
      <apex:column value="{!acc.Phone}"/>
      <apex:column value="{!acc.Fax}"/>
      <apex:column value="{!acc.Active__c}"/>
    </apex:pageBlockTable>

    <center>
    <apex:outputtext > Of {!controller.resultsize} </apex:outputtext>   

    <apex:commandLink value="First" action="{!controller.First}" rerender="pgBlock" rendered="{!
controller.hasPrevious}"/>   
    <apex:commandlink value="Previous" action="{!controller.Previous}" rerender="pgBlock"
rendered="{!controller.hasPrevious}"/>   
    <apex:commandLink value="Next" action="{!controller.Next}" rerender="pgBlock"
rendered="{!controller.hasNext}"/>   
    <apex:commandLink value="Last" action="{!controller.last}" rerender="pgBlock"
rendered="{!controller.hasNext}"/>
    </center>
  </apex:pageblock>
</apex:form>
</apex:page>
```

Controller Class:

```
public class AccountsPaginationController
{
   Public ApexPages.StandardSetController controller
   {
      Get
      {
        if(controller == null)
        {
           controller = new ApexPages.StandardSetController(Database.getQueryLocator([Select id, name, rating, industry, annualrevenue,   type, phone, fax, active__c from Account  Order by name]));

        }
        return controller;
      }

      Set;
   }

   Public List<Account> getAccounts()
   {
      controller.SetPageSize(5);
      return (List<Account>) controller.GetRecords();
   }
}
```

Page Redirection: (Page Navigation)

By using this feature, we can re-direct the user from one VF page to another.

To navigate the user from one page to another page, we have to to use "PageReference" class. Which holds the address of the Target Page to be get re-directed.

Syntax:

```
  PageReference <objectName> = new PageReference('/apex/<PageName>');
  <objectName>.SetRedirect(true);
```

Ex:

```
  PageReference pgRef = new PageReference('/apex/Page2');
  pgRef.SetRedirect(true);
```

```
/*
  Design a visualforce page, to re-direct the user from one page to another.
*/
```

Source Page:

VF Page Code:

```
<apex:page controller="SourcePageController" setup="false" sidebar="false" showheader="true"
tabStyle="Hiring_Manager__c" >
<apex:sectionHeader title="Source Page" subtitle="Page Navigation"/>
<apex:form >
  <apex:pageblock title="Source Page">
    <apex:commandButton value="Goto Target Page" action="{!GotoTargetPage}"/>
  </apex:pageblock>
</apex:form>
</apex:page>
```

Controller Class:

```
public class SourcePageController
{
  Public PageReference GotoTargetPage()
  {
    PageReference pgRef = new PageReference('/apex/TargetPage');
    pgRef.SetRedirect(true);

    return pgRef;
  }
}
```

Target Page:

VF Page Code:

```
<apex:page controller="TargetPageController" setup="false" sidebar="false" showheader="true"
tabStyle="Candidate__c" >
<apex:sectionHeader title="Target Page" subtitle="Page Navigation"/>
<apex:form >
  <apex:pageblock title="Target Page">
    <apex:commandButton value="Goto Source Page" action="{!GotoSourcePage}" />
  </apex:pageblock>
</apex:form>
</apex:page>
```

Controller Class:

```
public class TargetPageController
{
  Public PageReference GotoSourcePage()
  {
    PageReference pgRef = new PageReference('/apex/SourcePage');
    pgRef.SetRedirect(true);

    return pgRef;
  }
```

}

QueryString:

QueryString is used to pass One or more parameter values from One VisualForce page to another.

Syntax:

```
PageReference pgRef = new PageReference('/apex/<PageName>?<parmName>='+ <value>);
```

Ex:

```
string accountName = 'Merck Inc.';

PageReference pgRef = new PageReference('/apex/Page2?name='+accountName);
```

Note: We can pass multiple parameters to the target page as below.

Ex:

```
string accountName = 'Merck Inc.';
string industry = 'Banking';

PageReference pgRef = new PageReference('/apex/Page2?name='+accountName+ '&ind='+industry);
```

Note: In target page, we need to capture the parameter values we below.

Syntax:

```
ApexPages.CurrentPage().GetParameters().Get('<ParamName>');
```

Ex:

```
string accName = ApexPages.CurrentPage().GetParameters().Get('name');

string industryName =ApexPages.CurrentPage().GetParameters().Get('ind');
```

/*

Design a VisualForce Page, to search for the Account Records based on the inputText.

And display the Search Results in a seperate VisualForce Page.

***/**

Source Page:

==========

VF Page Code:

```
<apex:page controller="SourcePageController" setup="false" sidebar="false" showheader="true"
tabStyle="Hiring_Manager__c" >
<apex:sectionHeader title="Source Page" subtitle="Page Navigation"/>
<apex:form >
  <apex:pageblock title="Source Page">

    <apex:PageblockSection title="Search Accounts" collapsible="true" columns="2">
```

```
        <apex:inputtext label="Enter Account Name :" html-placeholder="Enter Account Name" value="{!searchText
}"/>
        <apex:commandButton value="Search Accounts" action="{!SearchAccounts}"/>
      </apex:PageblockSection>

   </apex:pageblock>

</apex:form>
</apex:page>
```

Controller Class:

```
-----------------
public class SourcePageController
{
   Public string searchText {get;set;}

   Public PageReference SearchAccounts()
   {
      PageReference pgRef = new PageReference('/apex/TargetPage?name='+searchText);
      pgRef.SetRedirect(true);

      return pgRef;
   }
}
```

Target Page:

```
============
VF Page Code:
-------------
<apex:page controller="TargetPageController" setup="false" sidebar="false" showheader="true"
tabStyle="Candidate__c" >
<apex:sectionHeader title="Target Page" subtitle="Page Navigation"/>
<apex:form >
   <apex:pageblock title="Target Page">
     <apex:commandButton value="Goto Source Page" action="{!GotoSourcePage}" />

     <apex:pageblockSection title="Search Results: ( {!lstAccounts.size} )" collapsible="false" columns="1">
       <apex:pageblockTable value="{!lstAccounts}" var="acc">
         <apex:column value="{!acc.Name}"/>
         <apex:column value="{!acc.Rating}"/>
         <apex:column value="{!acc.Industry}"/>
         <apex:column value="{!acc.Phone}"/>
         <apex:column value="{!acc.Fax}"/>
         <apex:column value="{!acc.Annualrevenue}"/>
         <apex:column value="{!acc.Active__c}"/>
       </apex:pageblockTable>
     </apex:pageblockSection>
   </apex:pageblock>
</apex:form>
</apex:page>
```

Controller Class:

```
public class TargetPageController
{
   Public List<Account> lstAccounts {get;set;}

   Public TargetPageController()
   {
      lstAccounts = new List<Account>();

      string accountName = ApexPages.CurrentPage().GetParameters().Get('name');

      if(accountName != null && accountName !='')
      {
         accountName += '%';

         lstAccounts = [select id, name, rating, industry, annualrevenue, active__c, phone,
               fax from Account
               Where name like : accountName];
      }
   }

   Public PageReference GotoSourcePage()
   {
      PageReference pgRef = new PageReference('/apex/SourcePage');
      pgRef.SetRedirect(true);

      return pgRef;
   }
}
```

Wrapper Class:

==========

Wrapper Class is nothing but a container, which can hold multiple object records inside a single collection.

By using Wrapper class, we can merge multiple collections data into a single collection.

This can be achieved by creating our own custom datatype or class with the required members.

We need to create our own Custom Class with the reuired variables, methods, constructors, properties, etc.

Ex:

```
  Public Class Employee
  {
            public integer empId;
            public string empName, address;
            public decimal salary;
            public Date joiningDate;
```

```
}

Employee emp = new Employee();
        emp.empId = 10001;
        emp.empName = 'Pawan Kumar';
        emp.salary = 20000;
        emp.address = 'Bangalore';
        emp.joiningDate = Date.newinstance(2015, 10,10);

List<Employee> lstEmployees = new List<Employee>();
        lstEmployees.Add(emp);
```

/*

Design a visualforce page, to display Lead, Contact records into a single Table.

1. Display each record with a "Checkbox".

2. Display the Record Name with "Hyperlink".

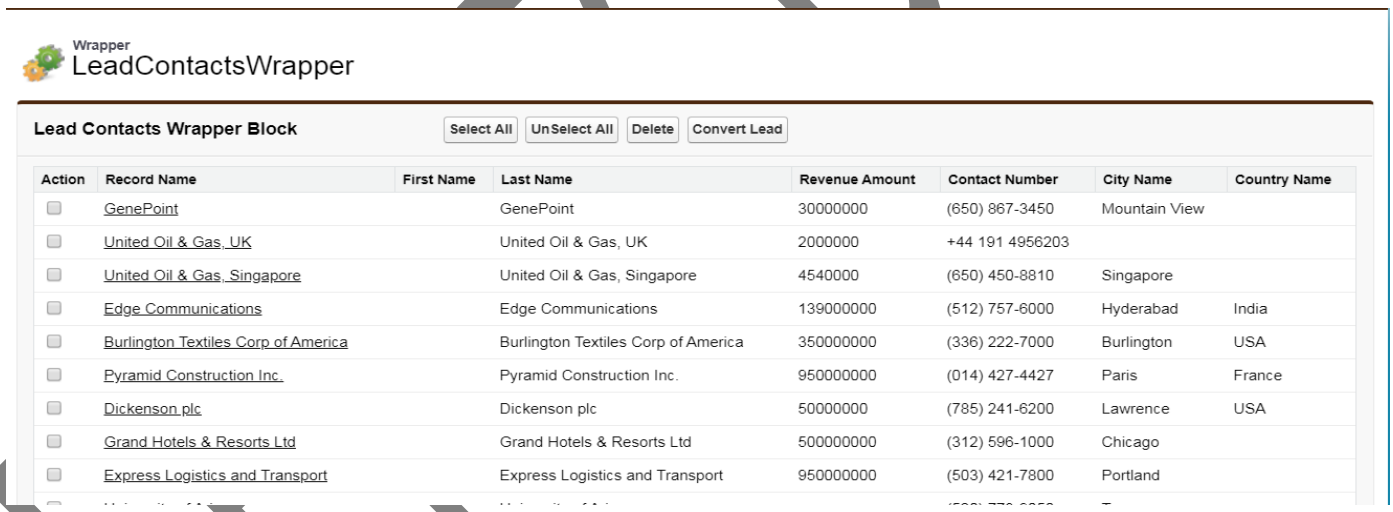
3. When the user click on "Hyperlink", then it should re-direct the user to the selected record.

4. When click on "Select All" button, it should select all the checkboxes.

5. When i click on "Unselect All" button, it should unselect all the checkboxes

6. When i click on "Delete" button, by selecting the checkboxes, it should remove the selected records from the object.

***/**



Action	Record Name	First Name	Last Name	Revenue Amount	Contact Number	City Name	Country Name
☐	GenePoint		GenePoint	30000000	(650) 867-3450	Mountain View	
☐	United Oil & Gas, UK		United Oil & Gas, UK	2000000	+44 191 4956203		
☐	United Oil & Gas, Singapore		United Oil & Gas, Singapore	4540000	(650) 450-8810	Singapore	
☐	Edge Communications		Edge Communications	139000000	(512) 757-6000	Hyderabad	India
☐	Burlington Textiles Corp of America		Burlington Textiles Corp of America	350000000	(336) 222-7000	Burlington	USA
☐	Pyramid Construction Inc.		Pyramid Construction Inc.	950000000	(014) 427-4427	Paris	France
☐	Dickenson plc		Dickenson plc	50000000	(785) 241-6200	Lawrence	USA
☐	Grand Hotels & Resorts Ltd		Grand Hotels & Resorts Ltd	500000000	(312) 596-1000	Chicago	
☐	Express Logistics and Transport		Express Logistics and Transport	950000000	(503) 421-7800	Portland	

Wrapper Class Code:

```
public class LeadContactWrapper
{
   Public id recordId{get;set;}
   public string fName {get;set;}
   public string lName {get;set;}
   Public string emailId {get;set;}
   Public string contactNumber {get;set;}
   Public string faxNumber {get;set;}
   Public boolean isSelected {get;set;}
```

```
    // Create a Constructor, which will convert the Lead record
    // into our own custom datatype
    Public LeadContactWrapper(Lead ldRecord, boolean isChecked)
    {
       recordId = ldRecord.Id;
       fName = ldrecord.FirstName;
       lName = ldRecord.LastName;
       contactNumber = ldRecord.Phone;
       emailId = ldrecord.Email;
       faxNumber = ldrecord.Fax;
       isSelected = isChecked;
    }

    // Create a Constructor, which will convert the Contact record
    // into our own custom datatype
    Public LeadContactWrapper(Contact con, Boolean isChecked)
    {
            recordId = con.id;
       fName = con.FirstName;
       lName = con.LastName;
       emailId = con.Email;
       faxNumber = con.Fax;
       contactNumber = con.Phone;
       isSelected = isChecked;
    }
}
```

VF Page Code:

```
<apex:page controller="CustomWrapperController" action="{!GetData}" setup="false" sidebar="false"
showheader="true" tabStyle="Customer__c" >
<apex:sectionHeader title="Wrapper" subtitle="Lead Contact Wrapper"/>
<apex:form >
   <apex:pageblock title="Lead Contacts Wrapper" id="pgBlock">
      <apex:pageblockButtons location="top">
         <apex:commandButton value="Select All" action="{!SelectAllRecords}" rerender="pgBlock"/>
         <apex:commandButton value="Unselect All" action="{!UnSelectAll}" rerender="pgBlock"/>
         <apex:commandButton value="Delete" action="{!DeleteSelectedRecords}" rerender="pgBlock"/>
      </apex:pageblockButtons>

      <apex:pageblockTable value="{!lstWrapper}" var="wr">
         <apex:column headerValue="Action">
            <apex:inputcheckbox value="{!wr.isSelected}"/>
         </apex:column>
         <apex:column headerValue="Record Name">
            <apex:outputLink value="/{!wr.recordid}" target="_blank" > {!wr.fName} {!wr.lName} </apex:outputLink>
         </apex:column>
      <!--    <apex:column value="{!wr.recordID}" headerValue="Record ID"/> -->
         <apex:column value="{!wr.FName}" headerValue="First Name"/>
```

```
        <apex:column value="{!wr.lName}" headerValue="Last Name"/>
        <apex:column value="{!wr.EmailId}" headerValue="Email ID"/>
        <apex:column value="{!wr.ContactNumber}" headerValue="Contact Number"/>
        <apex:column value="{!wr.faxNumber}" headerValue="Fax Number"/>
      </apex:pageblockTable>
   </apex:pageblock>
</apex:form>/>
</apex:page>
```

Controller Class:

```
public class CustomWrapperController
{
   Public List<LeadContactWrapper> lstWrapper {get;set;}

   // This method is used to get the Lead and Contact records
   // and convert them into custom wrapper type.
   Public void GetData()
   {
      lstWrapper = new List<LeadContactwrapper>();

      List<Lead> lstLeads = [select id, firstname, lastname, email,
                        phone, fax
                       from Lead
                          Where isConverted = false
                            Limit 10];

      if(! lstLeads.isEmpty())
      {
         for(Lead ld : lstLeads)
         {
            lstWrapper.Add(new LeadContactWrapper(ld, false));
            /*
            LeadContactWrapper wrapper = new LeadContactWrapper(ld);
            lstWrapper.Add(wrapper);
            */
         }
      }

      List<Contact> lstContacts = [select id, firstname, lastname, email,
                              phone, fax
                             from Contact
                              Limit 10];

      if(! lstContacts.isEmpty())
      {
         for(Contact con : lstContacts)
         {
            lstWrapper.Add(new LeadContactWrapper(con, false));
         }
```

```
    }
  }

  Public void SelectAllRecords()
  {
    if(!lstWrapper.isEmpty())
    {
       for(LeadContactWrapper wr : lstWrapper)
       {
          wr.isSelected = true;
       }
    }
  }

  Public void UnSelectAll()
  {
     if(! lstWrapper.isEmpty())
     {
        for(LeadContactWrapper wr : lstWrapper)
        {
           wr.isSelected = false;
        }
     }
  }

  Public void DeleteSelectedRecords()
  {
     if(! lstWrapper.isEmpty())
     {
        List<id> recordIdsToDelete = new List<ID>();

        for(LeadContactWrapper wr: lstWrapper)
        {
           if(wr.isSelected == true)
           {
              recordIdsToDelete.Add(wr.recordId);
           }
        }

        if(! recordIdsToDelete.isEmpty())
        {
           Database.Delete(recordIDsToDelete);

           GetData();
        }
     }
  }
}
```

AJAX REQUESTS IN VISUALFORCE PAGES

AJAX --> Asynchronous Java Script And XML.

To perform the operations the use will click on the required button, to invoke the request. Or the User has to refresh the page to invoke the request.

In this Synchronous process, the user has to perform the operations one by one in sequential manner. Which can be invoked by click on the "Button Control" or by click on "Refresh Button" in the browser.

The user has to wait till the completion of the first request, in-order to invoke the second request.

To avoid this problem, we can invoke the requests Asynchronously. Which will invoke the request automatically based on some event (Ex: Changing the Picklist value selection, Click on Enter Button, Typing some characters in text box, etc);

By using Asynchronous request, we can improve the performance of the application.

To invoke the Asynchronous request, VisualForce provides the below Tags.

1. **Action Poller (<apex:ActionPoller>)**
2. **Action Support (<apex:ActionSupport>)**
3. **Action Function (<apex:ActionFunction>)**
4. **Action Status (<apex:ActionStatus>)**
5. **ActionRegion (<apex:ActionRegion>)**

ActionPoller (<apex:ActionPoller>):

This tag is used to invoke the request for each specified time interval automatically.

Once the Specified time is over, it will invoke the request and perform the specified opteration.

Once the operation has been done, we can perform partial page refreshment.

Properties:

1. Interval ="Time in Seconds"
2. Action = "{! <MethodName> }"
   We need to specify the method name, which should be executed upon the specified time period.
3. ReRender = "Component Id to Refresh"

Ex:

```
<apex:actionPoller interval="10"
                   Action="{!GetAllAccounts}"
                   ReRender="pgBlock"/>
```

/*
Design a VisualForce Page, to refresh the Lead Records Table for each 10 seconds automatically.
***/**

VF Page Code:

```
<apex:page controller="LeadsActionPollerController" action="{!GetAllUnconvertedLeadRecords}" setup="false"
sidebar="false" tabStyle="Candidate__c" >
<apex:sectionHeader title="Lead Records" subtitle="All Lead Records"/>
<apex:form >

   <apex:actionPoller interval="10" action="{!GetAllUnconvertedLeadRecords}" rerender="pgBlock"/>

   <apex:pageBlock title="All Lead Records : {!lstLeads.size} " id="pgBlock">

      <apex:pageBlockTable value="{!lstLeads}" var="ld" >
         <apex:column value="{!ld.FirstName}"/>
         <apex:column value="{!ld.LastName}"/>
         <apex:column value="{!ld.Title}"/>
         <apex:column value="{!ld.Phone}"/>
         <apex:column value="{!ld.Fax}"/>
         <apex:column value="{!ld.Company}"/>
         <apex:column value="{!ld.Status}"/>
      </apex:pageBlockTable>

   </apex:pageBlock>
</apex:form>
</apex:page>
```

Controller Class:

```
public class LeadsActionPollerController
{
   Public List<Lead> lstLeads {get;set;}

   Public void GetAllUnconvertedLeadRecords()
   {
      lstLeads= new List<Lead>();

      lstLeads = [Select id, firstname, lastname, email, company, title,
                  rating, industry, status, phone, fax
                  from Lead
                     Where isConverted = false];
   }
}
```

Execute Anonymous Window(Create Leads in Background)

```
-----------------------------------------------------
  // Creating the Lead Records..
      List<Lead> lstLeads = new List<Lead>();

      for(Integer counter = 1; counter <= 5; counter++)
      {
```

```
        Lead ld = new Lead(firstname = 'Sample',status = 'Open - Not Contacted', lastname = 'Unittesting -'+ counter,
email='sample@gmail.com', title = 'Manager', company = 'IBM', phone = '999999999', fax = '9900990000', industry =
'Banking', ssn_number__c='ALPPB9329E', pan_number__c = 'ALPPB8989E');
            lstLeads.Add(ld);
      }
      insert lstLeads;
```

Action Support (<apex:ActionSupport>):

By using Action Support control, we can provide the additional ability to the existing controls. So that it can execute the controller action methods.

Ex: Upon changing the Picklist value selection, we can invoke the request to get the associated Contacts, etc.

Note: By using Action Support, we can achieve the Field Dependeency between the Picklist controls, through programming.

Properties:

1. Action = "{!<Method Name>}"
2. Event = "Specify the Event Name (Onchange, onblur, etc)"
3. ReRender = "Component ID to Refresh"

Ex:

```
  <apex:ActionSupport Action="{!LoadAssociatedContacts}"
                                          Event = "onchange"
                                          ReRender = "pgBlock"/>
```

/*
Design a visualforce page, to load the associated contacts based on the selected account from the picklist.
***/**

VF Page Code:

```
<apex:page controller="ShowAllAccountsPicklistController" tabStyle="Candidate__c" >
<apex:sectionHeader title="Account Records" subtitle="All Accounts"/>
<apex:form >
   <apex:pageblock title="Account Records">
     <apex:pageblockSection title="All Accounts" collapsible="true" columns="1">

        <apex:selectList value="{!selectedAccountID}" label="Select the Account Name :" size="1" multiselect="false">

           <apex:actionsupport action="{!GetRelatedContacts}" event="onchange"
rerender="pgBlock"></apex:actionSupport>

           <apex:selectoptions value="{!lstOptions }"></apex:selectoptions>
        </apex:selectList>

     </apex:pageblockSection>
   </apex:pageblock>
```

```
   <apex:pageblock title="Related Contacts : (  {!lstContacts.size}  ) " id="pgBlock">
      <apex:pageblockTable value="{!lstContacts}" var="con">
         <apex:column value="{!con.FirstName}"/>
         <apex:column value="{!con.lastname}"/>
         <apex:column value="{!con.Title}"/>
         <apex:column value="{!con.Email}"/>
         <apex:column value="{!con.Phone}"/>
         <apex:column value="{!con.fax}"/>
      </apex:pageblockTable>
   </apex:pageblock>

</apex:form>
</apex:page>

Controller Class:
---------------------
public class ShowAllAccountsPicklistController
{
   Public List<SelectOption> lstOptions {get;set;}
   Public List<Contact> lstContacts {get;set;}

   Public string selectedAccountID {get;set;}

   // Default Constructor...
   Public ShowAllAccountsPicklistController()
   {
      List<Account> lstAccounts= [Select id, name from Account
                                  order by name];

      lstOptions = new List<SelectOption>();

      if(! lstAccounts.isEmpty())
      {
         lstOptions.Add(new SelectOption('', '---- None ----'));

         for(Account acc : lstAccounts)
         {
            //SelectOption option = new SelectOption(acc.id, acc.name);
            lstOptions.Add(new SelectOption(acc.id, acc.name));
         }
      }
   }

   Public void GetRelatedContacts()
   {
      lstContacts = new List<Contact>();

      if(selectedAccountId != null && selectedAccountID != '')
      {
```

```
        lstContacts = [Select id, firstname, lastname, email, phone,fax, title, accountid
                       from Contact
                          Where accountid =: selectedAccountId ];
      }
    }
}
```

Action Function (<apex:ActionFunction>):

By using Action Function tag, we can invoke the Controller action methods through JavaScript.

We can make the ActionFunction tag as "Global". i.e. We can define an ActionFunction tag inside the "Form" tag, and we can reference the ActionFunction from any place inside the page.

By using ActionFunction, we can display the Progress / Status of the operation / request.

Once the operation has been done, we can perform the partial page refreshment based on the need.

Properties:

1. Name = "Action Function Name"
2. Action="{! <MethodName>}"
3. ReRender = "ComponentID To Refresh"
4. Status = "Action Status ID"
   --> To Show the Progress / Status of the operation.

Ex:

```
<apex:actionfunction name="searchLeads"
                                                action = "{!SearchLeadRecords}"
                                                ReRender="pgBlock"
                                                status ="result"/>
```

ActionStatus (<apex:ActionStatus>):

This tag is used to represent the Progress / Status of the current operation / request invoked by the user.

Properties:

ID = "actionStatusID"
startText = "Message To Display Upon Progress"
stopText = "Message to display after completion of request"

Ex:

```
<apex:actionstatus
                        id="accStatus"
                        startText = "Please Wait. Loading Data ...!! "
                        stopText ="Contacts Ready"/>
```

/*

Design a visualforce page, to search for the Lead Records based on the Last Name entered by the user in the search textbox.

*/
VF Page Code:

```
<apex:page controller="SearchLeadsFunctionController" setup="false" sidebar="false" showheader="true"
tabStyle="Candidate__c" >
<apex:sectionHeader title="Search" subtitle="Search Leads"/>
<apex:form >

  <apex:actionFunction name="SearchLeads" action="{!SearchLeadRecordsByLastName}" rerender="pgBlock"
status="accStatus"/>

  <apex:pageblock title="Search Lead Records">
    <apex:pageblockSection title="Search Leads" collapsible="false" columns="2">

      <apex:inputtext label="Enter Lead Name" html-placeholder="Enter Lead Last Name" value="{!searchText }"
onkeydown="! if(event.keyCode==13) {onblur(); SearchLeadRecordsByLastName;}"/>
      <apex:commandButton value="Search Leads" onclick="SearchLeads(); return false; "/>

    </apex:pageblockSection>
  </apex:pageblock>

  <apex:pageBlock title="Search Results" id="pgBlock">

    <apex:pageblockTable value="{!lstLeads}" var="ld">
      <apex:column value="{!ld.FirstName}"/>
      <apex:column value="{!ld.lastname}"/>
      <apex:column value="{!ld.Title}"/>
      <apex:column value="{!ld.Email}"/>
      <apex:column value="{!ld.Status}"/>
      <apex:column value="{!ld.Fax}"/>
    </apex:pageblockTable>

     <apex:actionStatus id="accStatus" startText="Please Wait. Loading Data..!" stoptext="Leads Ready">

    </apex:actionStatus>

    <!--
    <apex:actionStatus id="accStatus">
      <apex:facet name="status">
        <center>
          <image src="/img/loading.gif"/>
        </center>
      </apex:facet>
    </apex:actionStatus>
    -->
  </apex:pageBlock>
</apex:form>
</apex:page>
```

Controller Class:

```
public class SearchLeadsFunctionController
{
   Public string searchText {get;set;}

   Public List<Lead> lstLeads {get;set;}

   Public void SearchLeadRecordsByLastName()
   {
      if(searchText != null && searchText != '')
      {
         searchText += '%';

         lstLeads = new List<Lead>();

          lstLeads = [select id, firstname, lastname, email, company, title,
                        industry, phone, fax, status
                          from Lead
                            Where lastname like : searchText];
      }
   }
}
```

ActionRegion (<apex:ActionRegion>):

This tag is used to indicate what controls on the page to be get processed by the salesforce server. Instead of processing the whole page controls.

So that we can improve the performance of the page.

Only the Controls has been placed inside the "ActionRegion" tag, will be processed by the salesforce server.

Syntax:

```
  <apex:actionregion id="accRegion">
          ...
          ... Controls to be processed by the Server.
          ...
  </apex:actionregion>
```

**/*
  Design a visualforce page, to enable / disable the "Phone Number" field on the Page, based on the "Customer Priority" value.
          Customer Priority == "High" --> Display Phone Field.
          Customer Priority == "Low / Medium" --> Do Not Display.
*/**

VF Page Code:

```
<apex:page controller="withActionregionController" tabStyle="Account">

   <apex:form id="myform">

      <apex:pageBlock id="pageId">

         <apex:pageBlockSection title="If you will select High Customer Priority then phone textbox will be shown"
columns="1" id="out" collapsible="false">

            <apex:pageBlockSectionItem >

               <apex:outputLabel value="{!$ObjectType.Account.fields.CustomerPriority__c.label}" for="priority"/>

               <apex:actionRegion >

               <apex:inputField value="{!acc.CustomerPriority__c}" id="priority" >

                  <apex:actionSupport action="{!priorityChanged}" reRender="pageId" event="onchange"/>

               </apex:inputField>

               </apex:actionRegion>

            </apex:pageBlockSectionItem>

            <apex:inputField value="{!acc.Phone}" rendered="{!showPhone}"/>

         </apex:pageBlockSection>

         <apex:pageBlockSection title="Accounts Information" columns="2" collapsible="false">
            <apex:inputfield value="{!acc.Name}"/>
            <apex:inputfield value="{!acc.Rating}"/>
            <apex:inputField value="{!acc.Industry}"/>
         </apex:pageBlockSection>

      </apex:pageBlock>

   </apex:form>
</apex:page>
```

Controller Class:

```
----------------
public class withActionregionController
{
   public Account acc{get;set;}
   public Boolean showPhone{get;set;}

   public withActionregionController()
   {
      acc = new Account();
```

```
        showPhone = false;

    }

    public PageReference priorityChanged()
    {
        if(acc.CustomerPriority__c == 'High')
        {
            showPhone = true;
        }
        else
        {
            showPhone = false;
        }
        return null;
    }
}
```

VIEW STATE IN VISUALFORCE

View State holds state of the Visualforce page. To maintain state in a Visualforce page, the Force.com platform includes the state of components, field values, and controller state in a hidden form element.
View state is an object, which persists the Form data during the post-back implementations.

It preserves the form control's data temporarily in our local system.

View state data will be stored in the form of "hidden" fields.

View state will convert the form data in "Base64" enco ding format, in-order to provide the security. View State is encrypted and transferred to client with each response.

We can view the view state information in "Page View Source" option from the menu, which displays when we release the right mouse button on the form.

Note: All variable, and collections and objects defined inside the controller class will be persisted in "ViewState" object.

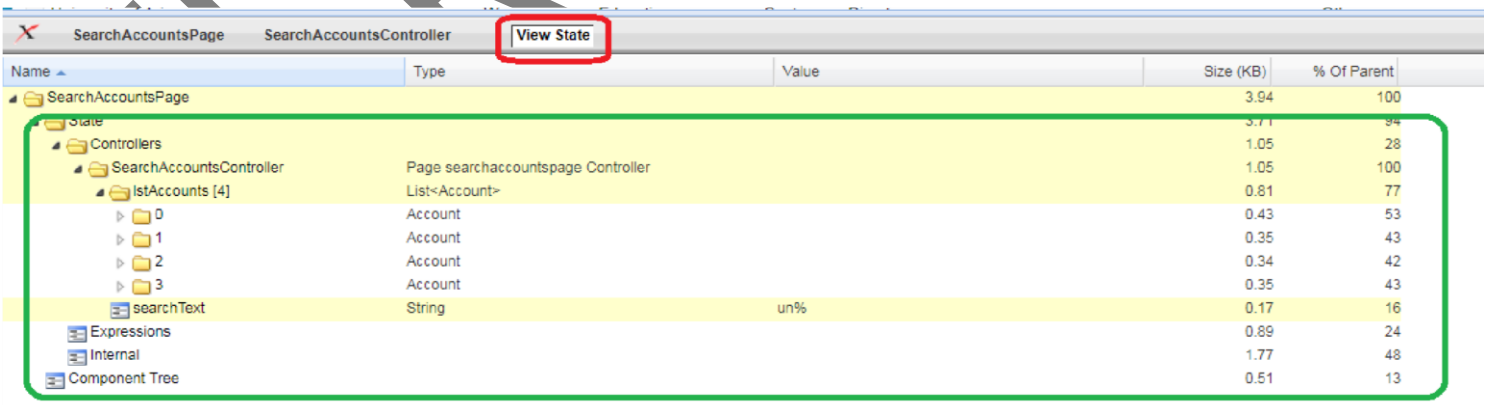
Ex: <input type="hidden" name="at" value="AF6bupNdff7BqLXCDzEkOHF8B0cotGDGtw">

View State information can be viewed by using **"View State Inspector"** to check the View State Data.

Note: View State Inspector will be only visible when developer mode is ON in User profile.

Goto your Name and expand it.

1. Click on "My Settings" link.
2. Click on "Personal" in the left panel and expand it.
3. Click on "Advanced User Details" link.
4. Click on "Edit" button.
5. Select the checkbox "**Enable ViewState in Development Mode**".
6. Click on "Save" button.



Note: We can store max. of 135 KB of data in View State. If the View state max. size exceeds, then it will raise the exception "View State Limit Exceeds".

To avoid the View State Limit Exceptions, we need to use the keyword "Transient". i.e. We have to define the objects or the variable with the Transient keyword, for which the data need not to be persist into viewstate. **Syntax**: public Transient <Variable Name / ObjectName>;

Ex:

```
public Transient integer accountNumber;
public Transient string customerName;
public Transient List<Account> lstAccounts;
```

The above variable and object's data will not persist in Viewstate.

Best Practices (To avoid ViewState Exceptions):

1. Use "Transient" keyword, to indicate the object information need not be stored inside the view state.

Ex:

```
Public transient string customerName {get;set;}
Public Transient List<Account> lstAccounts{get;set;} Public
Transient Map<Id, Account> mapAccounts{get;set;}
```

2. Declare variable as **Static**, as it is not saved in View State.

3. Avoid the usage of same data with multiple copies. (i.e. maintain only one copy of the records)

4. Query only the required columns information by using SOQL, to be get used inside the visual force page.

5. Avoid the large number of SOQL queries, by using "Custom Settings" which can be access without using SOQL queries and stores the data in "Application cache". So that we can make our application lightweight.

6. Use the HTML tags, instead of using VisualForce page tags.

7. Use filters and pagination to reduce data requiring state.

With Sharing (User Mode) Vs. Without Sharing (System Mode)

Use the with sharing or without sharing keywords on a class to specify whether or not to enforce sharing rules.

With Sharing → User Mode

Without Sharing → System Mode.

Note: With Sharing or Without Sharing keywords can be applicable only on Class level.

With Sharing:

The “With Sharing” keyword allows you to specify that the sharing rules for the current user be taken into account for a class.

When the class has been defined with the keyword "With Sharing", then the class will be executed based on the currently loggedin user's profile level permissions & Sharing Rules will get enabled.

(i.e. upon making the records to be visible, it will consider the profile level permissions of the user)
i.e. Each user will get a different view.(i.e. records count will be different based on the login in user)

Ex:

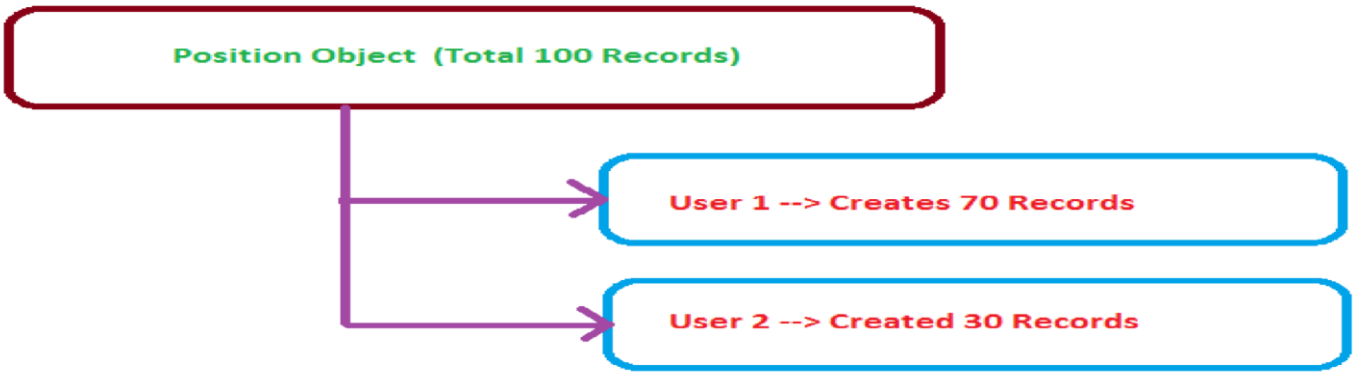
```
Public with sharing class SampleSharingClass
{
        // Write the Business Logic here...
}
```

Without Sharing:

Use the without sharing keywords when declaring a class to ensure that the sharing rules for the current user are **not**enforced.

By default apex code is run under default system context. In system context, Apex code has access to all objects and fields - object permissions, field-level security, sharing rules aren’t applied for the current user.

UseCase:



Without Sharing: (System Mode)

While creating the class, if the user didn't mention the "With Sharing" keyword, then the class methods will be executed in system mode.

It will execute the class irrespective of user profile level permission. i.e every user will get the same view.

Without sharing is also called as "System Mode". Which is Default mode for the classes.

Ex:

```
Public class PositionDetails
{
    public integer getPositions()
    {
        // Code to Fetch Position Records.
    }
}
```

Observation:

- USer1 → Executed the Method (i.e. getPositions()) --> It will display all the 100 record to the users. (Note: Irrespective of the owner.)
- USer2 → Executed the Method (i.e. getPositions()) --> It will display all the 100 record to the users. (Note: Irrespective of the owner.) **With**

Sharing: (USer Mode)

If the user has defined the class with the keyword "With Sharing", then the class methods will be executed in "USer Mode".

It will execute the class methods based on the user profile level permission. i.e each user will able to see the records, on which he has the access. With sharing is also called as "User Mode".

Ex:

```
Public with sharing class PositionDetails
{
    public integer getPositions()
    {
        // Code to Fetch Position Records.
    }
}
```

Observations:

- USer1 → Executed the Method (i.e. getPositions()) --> It will display 70 record to the users. (Note: Associated to the user1.)
- USer2 → Executed the Method (i.e. getPositions()) --> It will display 30 record to the users. (Note: Associated to the user2.)

Granting the Access on the VisualForce Page to User:

To grant the access on the visualforce page to one or more users, we need to use "PermissionSets"

1. Create a Permission set, by adding all the required visualforce pages.
2. Assign the PermissionSet to Required Users.

Granting the Access on the VisualForce Page to Profile:

To grant the access on a visualforce page to the profile users, we have to use the below navigation.

Goto Setup --> Goto Administer options.

1. Click on "Profiles" link.
2. Select the Required Profile from the List(Based on the User).
3. Goto the Profile.
4. Goto "Enabled VisualForce Pages" Related List.
5. Click on "New" button.
6. Select the VisualForce Pages from the picklist, on which we need to grant the access.
7. Click on "Save" button.

Note:

- If we grant the access on the VisualForce page through the profile, then all the Users associated with the profile will get the access on the page.
- If we grant the access on the VisualForce through the profile, then All the Referenced Controller classes also will be granted to the profile users.

Static Resources:

Static Resource is a location, which provides some amount of space to upload all the static resources referenced inside the visualforce page. i.e. We can upload the images, pictures, Documents, PDF Files, .Zip Files, .Jar Files, etc.

i.e. All the Scripting Files, Libraries, CSS Stylesheets should be uploaded into the Static Resource.

Note:

1. Each Static Resource size should not be exceeds 5 MB.

2. In an organization, we can upload the static resources upto max. of 250 MB.

Note: Upon uploading the Static Resource, we can enable the "Cache Control", to improve the application performance.

Navigation:

Click on "Setup" menu

1. Goto "Build" menu in left panel.
2. Click on "Develop" and expand it.
3. Click on "Static Resource" link.
4. Click on "New" button, to upload a New Static Resource
5. Enter the Static Resource Name, Description.
6. Select the File / Image to be uploaded by using "Browse" button.
7. Select the Cache Control (Private / Public) 8. Click on "Save" button.

Note: All the Static Resources can be accessed by using the Global variable "$Resource".

Syntax: {!$Resource.<StsaticResourceName>}

Note: We can query all the Static Resources by using the below SOQL Query.

Ex: List<StaticResource> lstResources = [select id, name, body from staticresource];

/* **Design a visualforce page, to display a Static Resource(PDF File) content in the VF page.** */

VF Page Code:

```
<apex:page setup="false" sidebar="false" showHeader="true" >
    <apex:sectionHeader title="Static Resources" subtitle="Show PDF File"/>
    <apex:form >
        <object data="{!$Resource.AsynchronousProcessing}"
                Width="100%"
               type="application/pdf"
    style="height :500px; border : 1px solid grey;"
                />
    </apex:form>
</apex:page>
```