SALESFORCE

DEVELOPER I (450)

AlmaMate Info Tech Pvt. Ltd.

Spring Meadows Business Park
Plot no A-61, B/4, Sector 63,
Noida.

☎ 120) 4787878

✉ info@almamate.in

Table OfContent

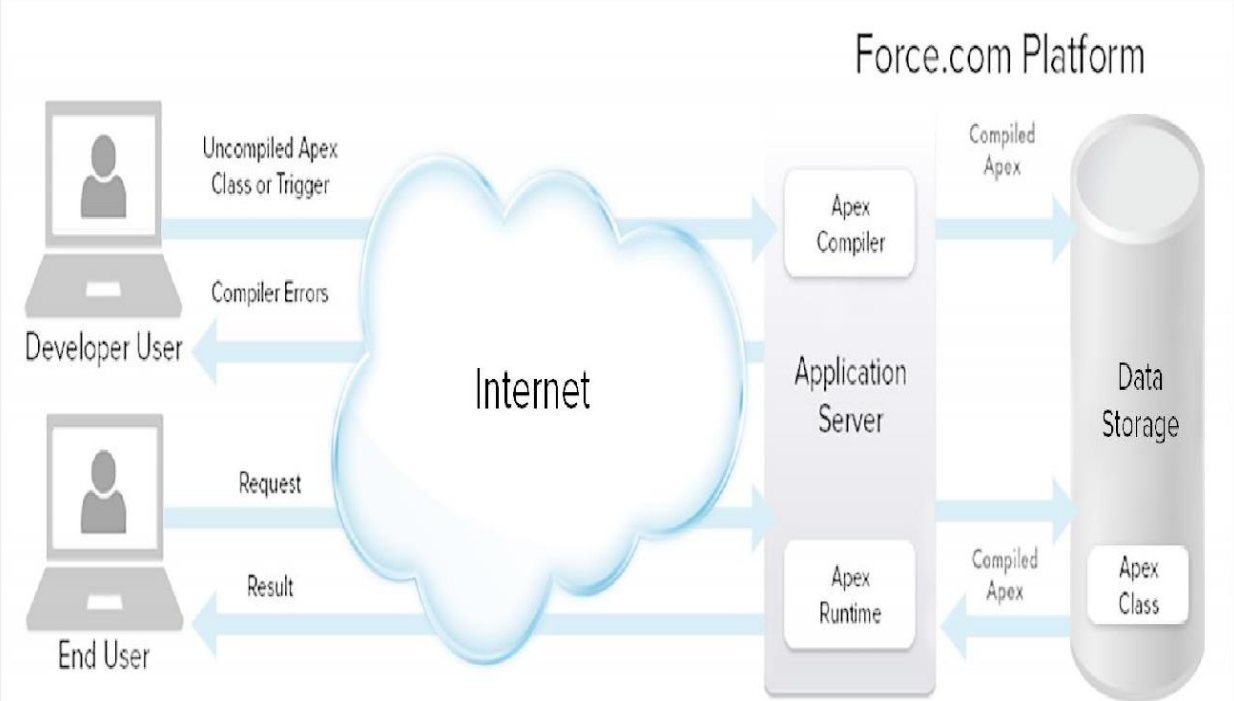
Apex Programming

Introduction to Apex Programming

Apex is a programming language that uses Java-like syntax and acts like database stored procedures. Apex enables developers to add business logic to system events, such as button clicks, updates of related records, and Visualforcepages.

As a language, Apex is:

- Hosted—Apex is saved, compiled, and executed on the server
- Automatically upgradeable—Because compiled code is stored as metadata in the platform, Apex is automatically upgraded as part of Salesforce releases.
- Object oriented—Apex supports classes, interfaces, and inheritance.
- Strongly typed—Apex validates references to objects at compile time.
- Multitenant aware—Because Apex runs in a multitenant platform
- Integrated with the database—It is straightforward to access and manipulate records.
- Data focused—Apex provides transactional access to the database, allowing you to roll back operations.
- Easy to use—Apex is based on familiar Java idioms.
- Easy to test—Apex provides built-in support for unit test creation, execution, and code coverage.
- Versioned—Custom Apex code can be saved against different versions of the API.



Apex Language Highlights

Like other object-oriented programming languages, these are some of the language constructs that Apex supports:

- Classes, interfaces, properties, and collections (including arrays).
- Object and array notation.
- Expressions, variables, and constants.
- Conditional statements (if-then-else) and control flow statements (for loops and while loops).

Unlike other object-oriented programming languages, Apex supports:

- Cloud development as Apex is stored, compiled, and executed in the cloud.
- Triggers, which are similar to triggers in database systems.
- Database statements that allow you to make direct database calls and query languages to query and search data.
- Transactions and rollbacks.

- The global access modifier, which is more permissive than the public modifier and allows access across namespaces and applications.
- Apex is a case-insensitive language.

Development Tools

You can write Apex and access debugging information directly in the browser by using the Salesforce user interface.

You can also write Apex on a client by using the Force.com IDE plugin for Eclipse.

Data Types Overview

Apex supports various data types, including a data type specific to Salesforce—the sObject data type.

- A primitive, such as an Integer, Double, Long, Date, Datetime, String, ID, Boolean, among others.
- An sObject, either as a generic sObject or as a specific sObject, such as an Account, Contact, or MyCustomObject__c (you'll learn more about sObjects in a later unit.)
- A collection, including:
  - A list (or array) of primitives, sObjects, user defined objects, objects created from Apex classes, or collections
  - A set of primitives
  - A map from a primitive to a primitive, sObject, or collection
- A typed list of values, also known as an *enum*
- User-defined Apex classes
- System-supplied Apex classes

PrimitiveDataTypesandVariables

Apexhasanumberofprimitivedatatypes.

- **String**:Stringsaresetofcharactersandareenclosed insinglequotes.Theystoretextvaluessuchasanameoranaddress.
- **Boolean**:Booleanvaluesholdtrueorfalsevaluesandyoucanusethemtotest whetheracertainconditionistrueorfalse.
- **Time,DateandDatetime**:Variablesdeclared withanyofthese datatypesholdtime,date,ortimeanddatevaluescombined.
- **Integer,Long,DoubleandDecimal**:Variablesdeclared withanyof these datatypesholdnumeric values.
- **Nullvariables**:Variablesthatyoudon'tassign valuesto.
- **Enum**:Anenumerationofcontantvalues.

String

UsetheStringdatatypewhenyouneedtostoretextvalues,suchasanameoranad dress.Stringsaresetsofcharactersenclosedin singlequotes.Forexample,'Iamastring'.Youcancreate astringandassignittoavariablesimplyby executingthefollowing:

```
String myVariable = 'I am a string.';
```

Thepreviousexamplecreatesaninstanceoftheclass,representedbythev ariable*myVariable*,andassignsitastring valuebetweensinglequotes.

Youcanalsocreate stringsfromthevaluesofothertypes,suchasdates,byusingtheStringstatic methodvalueOf().Executethe following:

```
Date myDate = Date.today();
String myString = String.valueOf(myDate); System.debug(myString);
```

The+operatoractsasaconcatenationoperator whenappliedtostrings.Thefollowingresultsinasinglestringbeingcreated:

```
System.debug( 'I am a string' + ' cheese');
```

The==and!=operatorsactasacaseinsensitivecomparisons.Executethefollowingtoconfirmthatboththecomparisonsbelowreturn true:

```
String x = 'I am a string';
String y = 'I AM A STRING';
String z = 'Hello!';
System.debug (x == y);
System.debug (x != z);
```

Let'stakealookatwhateachmethodof String does.

- TheendsWithmethodreturnstruebecausethestringendswiththesamestringasthatintheargument.
- Thelengthmethodreturnsthelengthofthestring.
- Thesubstringmethodproducesanewstringstarting fromthecharacterspecifiedinthefirstindexargument,countingfrom zero,throughthesecondargument.
- ThereplaceAllmethodreplaceseachsubstringthatmatchesaregularexpression withthespecifiedreplacement.

BooleanandConditional Statements

DeclareavariablewiththeBooleandatatypewhenitshouldhaveatrueorfalsevalue .Stringoperators

returnaBooleanvaluebasedontheresultofthestringcomparison.Youcanalsosimplecreate avariableandassign itavalue:

```
Boolean isLeapYear = true;
```

Thereare anumberofstandardoperatorsonBooleans.Forexample,allofthesestatements evaluateto false:

```
Boolean iAmFalse = !true;

Boolean iAmFalse2 = iAmFalse && true;
```

Boolean iAmFalse3 = iAmFalse || false;

Time,Date,andDatetime

Therearethreedatatypesassociated withdatesandtimes.TheTimedatatypestorestimes(hours,minutes,secondandmillis econds). TheDatedatatypestoresdates(year,month andday).TheDatetimedatatypestoresbothdatesandtimes.

Each of these classes has a newInstance method with which you can construct particular date and time values. For example, execute the following:

Date myDate = Date.newinstance(1960, 2, 17);

Time myTime = Time.newInstance(18, 30, 2, 20); System.debug(myDate);

System.debug(myTime);

Output is:

1960-02-17 00:00:00

18:30:02.020Z

The Date data type does hold a time, even though it's set to 0 by default. You can also create dates and times from the current clock:

Datetime myDateTime = Datetime.now();

Date today = Date.today();

The date and time classes also have instance methods for converting from one format to another. For example:

Time t = DateTime.now().time();

Datetime has the addHours, addMinutes, dayOfYear, timeGMT methods and many others. Execute the following:

Date myToday = Date.today();

Date myNext30 = myToday.addDays(30);

System.debug('myToday =' + myToday); System.debug('myNext30=' + myNext30);

Integer,Long,DoubleandDecimal

Tostorenumeric valuesinvariables,declareyourvariableswithoneofthenumericdatatypes:Integer,Long,DoubleandDecimal.

Integer: A32-bitnumberthatdoesn'tincludeadecimalpoint

Long: A64-bitnumberthatdoesn'tincludeadecimalpoint.

Double: A64-bitnumberthatincludesadecimalpoint.

Decimal:

Anumberthatincludesadecimalpoint.Decimalisanarbitraryprecisionnumber. Currencyfieldsareautomaticallyassignedthe typeDecimal.

Executethefollowingtocreatevariablesofeachnumerictype.

```
Integer i = 1;

Long l = 2147483648L;

Double d = 3.14159;

Decimal dec = 19.23;
```

YoucanusethevalueOfstaticmethodtocastastringtoanumerictype.Forexample,thefollowingcreatesanIntegerfromstring '10',andthenadds20toit.

```
Integer countMe = Integer.valueof('10') + 20;
```

NullVariables

Ifyoudeclare avariableanddon'tinitializeitwithavalue,it willbenull. Nullmeanstheabsenceofavalue.Youcan alsoassignnulltoanyvariabledeclaredwithaprimitivetype.Forexample,bothofthesestatements resultinavariablesettonull:

```
Boolean x = null;

Decimal d;
```

nums

Useenumerations(enums)tospecifyasetofconstants.Defineanewenumerationb yusingtheenumkeywordfollowedbythelistof identifiersbetween curlybraces.Eachvalueintheenumerationcorrespondstoan Integervalue,starting fromzeroandincrementing byonefrom lefttoright.Becauseeachvaluecorrespondstoaconstant,theidentifiersareinupper case.Forexample,thisexample definesanenumerationcalledSeasonthatcontainsthefourseasons:

```
public enum Season {WINTER, SPRING, SUMMER, FALL}
```

TheIntegervalueofWINTERis0,SPRING1,SUMMER2,FALL3.

Onceyoudefineyourenumeration,you canusethenewenumtypeasadatatypefordeclaringvariables.

```
public enum Season {WINTER, SPRING, SUMMER, FALL}

Season s = Season.SUMMER;

if (s == Season.SUMMER) {
    System.debug(s);
} else {
    System.debug('Not summer.');
}
```

Comments

Commentsarelinesoftextthatyouaddtoyourcodetodescribe whatitdoes.

Apexhastwoforms ofcomments.

Thefirstusesthe//tokentomarkeverythingonthesamelinetotherightofthetokenasacomment.

Thesecondencloses ablockoftext,possiblyacrossmultiplelines,betweenthe/*and*/tokens.

Loops

To repeatedlyexecute ablockofcodewhileagivenconditionholdstrue,usealoop.Apexsupportsdo-while,while,andforloops.

WhileLoops

Ado-whilelooprepeatedlyexecutesablockofcodeaslongasaBooleanconditionspecifiedinthewhilestatementremainstrue. Executethefollowingcode:

```
Integer count = 1;
do {
    System.debug(count);
    count++;
} while (count < 11);
```

ThewhileloprepeatedlyexecutesablockofcodeaslongasaBooleanconditionspecifiedatthebeginningremainstrue.

```
Integer count = 1;
while (count < 11) {
System.debug(count);
    count++;
}
```

ForLoops

Therearethreetypesofforloops.Thefirsttypeofforloopisatraditionalloopthatiterat esbysettingavariabletoavalue,checkinga condition,andperformingsomeactiononthevariable.

```
for (Integer i = 1; i <= 10; i++){
System.debug(i);}
```

Asecondtypeofforloopisavailableforiteratingoveralistoraset.

```
Integer[] myInts =new Integer[] {10,20,30,40,50,60,70,80,90,100};
for (Integer i: myInts) {
System.debug(i);
}
```

ThethirdtypeofforloopisSOQL for Loop.

Apex Collections

Lists hold an ordered collection of objects. Lists in Apex are synonymous with arrays and the two can be used interchangeably.

The following two declarations are equivalent.

The colors variable is declared using the List syntax.

```
List<String> colors
= new List<String>();
```

Alternatively, the colors variable can be declared as an array but assigned to a list rather than an array.

```
String[] colors = new List<String>();
```

You can add elements to a list when creating the list, or after creating the list by calling the add() method. The example shows you both ways of adding elements to a list.

```
List<String> colors = new List<String> { 'red', 'green', 'blue' };

List<String> moreColors = new
List<String>(); moreColors.add('orange');
moreColors.add('purple');
```

List elements can be read by specifying an index between square brackets, just like with array elements. Also, you can use the get() method to read a list element. The example also shows how to iterate over array elements.

```
String color1 = moreColors.get(0);
String color2 = moreColors[0];
// Iterate over a list to read
elements
for(Integer i=0;i<colors.size();i++) {
   // Write value to the debug
   log System.debug(colors[i]);
}
```

Apex supports two other collection types: Set and Map.

Sets

Asetis anunorderedcollectionofobjectsthatdoesn’tcontainanyduplicatevalues.Useaset whenyoudon’tneedtokeeptrackofthe orderoftheelementsinthecollection,andwhentheelementsareuniqueanddon’t havetobesorted.

Thefollowingexamplecreatesandinitializesanew set,addsanelement,andchecksifthesetcontainsthestring'b':

```
Set<String> s = new Set<String>{'a','b','c'};

// Because c is already a member, nothing will happen. s.add('c');

s.add('d');

if (s.contains('b')) {
    System.debug ('I contain b and have size ' + s.size());
}
```

Maps

Maps arecollectionsofkey-valuepairs,wherethekeysareofprimitivedatatypes.Useamapwhenyouwanttostorevaluesthatareto bereferencedthroughakey.Forexample,usingamapyoucanstorealistofaddresses thatcorrespondtoemployeeIDs.

```
Map<Integer,String> empAdd = new Map<Integer,String>();

empAdd.put (1, '123 San Francisco, CA');

empAdd.put (2, '456 Dark Drive, San Francisco, CA');
System.debug('Address for employeeID 2: ' +
empAdd.get(2));
```

Mapsalsosupportashortcutsyntaxforpopulating thecollectionwhencreatingit.

```
Map<String,String> myStrings = new Map<String,String>

{ 'a'=>'apple','b'=>'bee' };

System.debug(myStrings.get('a'));
```

Apex Classes

One of the benefits of Apex classes is code reuse. Class methods can be called by triggers and other classes.

Apexisanobject-orientedprogramming.Objectsarecreatedfromclasses—datastructuresthatcontainsclassmethods,instancemethods,anddata variables.Classes,inturn,canimplementaninterface,whichissimplyasetofmethods.

- **ClassesandObjects**:Classesaretemplatesfromwhichyoucan createobjects.
- **PrivateModifiers**:Theprivatemodifierrestrictsaccesstoaclass,oraclassmethodormembervariablecontainedinaclass,sothat theyaren'tavailabletootherclasses.
- **StaticVariables,ConstantsandMethods**:Staticvariables,constants,andmethods don'tdependonaninstanceofaclassandcan beaccessedwithout creatinganobjectfromofaclass.
- **Interfaces**:Interfacesarenamedsetsofmethod signaturesthatdon'tcontainanyimplementation.
- **Properties**:Propertiesallowcontrolledreadandwrite accesstoclassmembervariables.

DefiningClasses

Apexclassesaresimilarto Javaclasses. A*class*is atemplateorblueprintfromwhichobjectsarecreated.Anobjectisaninstanceofa class.

Forexample,aFridgeclassdescribesthestateofafridgeandeverythingyoucan dowithit.AninstanceoftheFridgeclassisa specificrefrigeratorthatcanbepurchasedorsold.

AnApexclasscancontainvariablesandmethods.

Variablesareusedtospecifythestateofanobject,suchastheobject'snameortype. Sincethesevariablesareassociatedwithaclassandaremembersofit,theyarereferred toasmembervariables.

Methodsareusedto controlbehavior,suchaspurchasingorsellinganitem.

Methodscanalsocontainlocalvariablesthataredeclaredinsidethemethodanduse donlybythemethod.Whereasclassmember variablesdefinetheattributesofanobject,suchasnameorheight,localvariablesinm ethodsareusedonlybythemethodanddon't describetheclass.

CreatingandInstantiating Classes

```
public class Fridge {

    public String modelNumber;

    public Integer numberInStock;

    public void updateStock(Integer justSold) {
        numberInStock = numberInStock - justSold;
    }
}
```

Theclasshastwomembervariables,modelNumberandnumberInStock,and onemethod,updateStock.Thevoidtypeindicatesthattheupdate Stockmetho ddoesn't returnavalue.

Youcannowcreate ObjectofthisnewclasstypeFridge,andmanipulatethem.

```
Fridge myFridge = new Fridge();

myFridge.modelNumber = 'MX-O';

myFridge.numberInStock = 100;

myFridge.updateStock(20);

Fridge myOtherFridge = new Fridge();

myOtherFridge.modelNumber = 'MX-Y';

myOtherFridge.numberInStock = 50;
```

```
System.debug('myFridge.numberInStock=' +
myFridge.numberInStock);
System.debug('myOtherFridge.numberInStock='
+ myOtherFridge.numberInStock);
```

PrivateModifiers

Bydeclaringthemembervariablesasprivate,youhavecontroloverwhichmemberva riablescanbereadorwritten,andhowthey're manipulatedbyotherclasses.Youcanprovidepublicmethodstogetandsettheval uesoftheseprivatevariables.Thesegetterand settermethodsarecalledproperties.Declaremethodsasprivate whenthesemethods areonlytobecalledwithinthedefiningclassandarehelpermethods.Helpermethods don't representthebehavioroftheclassbutare theretoservesomeutilitypurposes.

Bydefault,amethod orvariableisprivateandisvisibleonlytotheApexcodewithinthedefiningclass.

Let'smodifyourFridgeclasstouseprivatemodifiersforthemembervariables and include getter and setter methods.

Constructors

Apexprovidesadefaultconstructorforeachclassyoucreate.Forexample,youwereabl etocreateaninstanceoftheFridgeclass by callingnew Fridge(),eventhoughyoudidn'tdefinetheFridgeconstructor yourself.

However,theFridgeinstancein thiscasehasallitsmember variablessettonullbecause alluninitializedvariablesinApexarenull.

Sometimes youmightwantto providespecificinitialvalues,it'softenusefultohaveaconstructorthattakesparame terssoyoucaninitialize themembervariablesfromthepassedinargumentvalues.

Tryaddingtwoconstructors,onewithoutparametersandonewithparameters.

```
public Fridge() {
modelNumber = 'XX-XX';
numberInStock = 0;
}

public Fridge(String theModelNumber, Integer theNumberInStock) {
    modelNumber = theModelNumber;
    numberInStock = theNumberInStock;
}
```

Youcannowcreate aninstanceandsetthedefaultvaluesallatonceusingthesecondconstructor .

```
Fridge myFridge = new Fridge('MX-EO', 100); System.debug
(myFridge.getModelNumber());
```

Static Variables, Constants, and Methods

Eachindividualinstancehasitsowncopyofinstance variables,andtheinstancemethodscanaccessthesevariables.Staticvariables areassociatedwiththeclassandnottheinstanceandyoucanaccessthemwithouti nstantiatingtheclass.

Youcan alsodefinestaticmethodswhichareassociatedwiththeclass,nottheinstance.

```
public static Integer stockThreshold = 5;
```

To access this:

```
System.debug ( Fridge.stockThreshold );
```

Todeclareavariableasbeingaconstant—somethingthatwon't change. Youcanusethefinalkeyword todothisinApex;

```
public static final Integer STOCK_THRESHOLD = 5;

For static method we use:

public static void toDebug(Fridge aFridge) {

    System.debug ('ModelNumber = ' +
    aFridge.modelNumber);

    System.debug ('Number in Stock = ' +
    aFridge.numberInStock);

}
```

Property Syntax

InPrivateModifiers,youmodifiedthevariablestobeprivate,ensuringthattheycanonlybeaccessedthroughamethodandthereisashorthandsyntaxthatletsyoudefine avariableandcodethatshouldrunwhen thevariableisaccessedorretrieved.

Addanew property,`ecoRating`,tothe`Fridge`classbyaddingthefollowing:

```
public Integer ecoRating
{
   get { return ecoRating;
       }
   set {
   ecoRating = value; if (ecoRating < 0) ecoRating =0;
       }
}
```

Using sObjects

Because Apex is tightly integrated with the database, you can access Salesforce records and their fields directly from Apex. Every record in Salesforce is natively represented as an *sObject* in Apex.

Each Salesforce record is represented as an sObject before it is inserted into Salesforce. Likewise, when persisted records are retrieved from Salesforce, they're stored in an sObject variable.

Standard and custom object records in Salesforce map to their sObject types in Apex. Here are some common sObject type names in Apex used for standard objects.

- Account
- Contact
- Lead
- Opportunity

If you've added custom objects in your organization, use the API names of the custom objects in Apex. For example, a custom object called Merchandise corresponds to the Merchandise__c sObject in Apex.

Creating sObject Variables

To create an sObject, you need to declare a variable and assign it to an sObject instance. The data type of the variable is the sObject type.

The following example creates an sObject variable of type Account and assigns it to a new account with the name Acme.

```
Account acct = new Account(Name='Acme');
```

The names of sObjects correspond to the API names of the corresponding standard or custom objects. Similarly, the names of sObject fields correspond to the API names of the corresponding fields.

For custom objects and custom fields, the API name always ends with the __c suffix. For custom relationship fields, the API name ends with the __r suffix. For example:

- A custom object with a label of Merchandise has an API name of Merchandise__c.
- A custom field with a label of Description has an API name of Description__c.
- A custom relationship field with a label of Items has an API name of Items__r.

Creating sObjects and Adding Fields

Before you can insert a Salesforce record, you must create it in memory first as an sObject. Like with any other object, sObjects are created with the new operator:

```
Account acct = new Account();
```

The account referenced by the acct variable is empty because we haven't populated any of its fields yet. There are two ways to add fields: through the constructor or by using dot notation.

The fastest way to add fields is to specify them as name-value pairs inside the constructor. For example, this statement creates a new account sObject and populates its Name field with a string value.

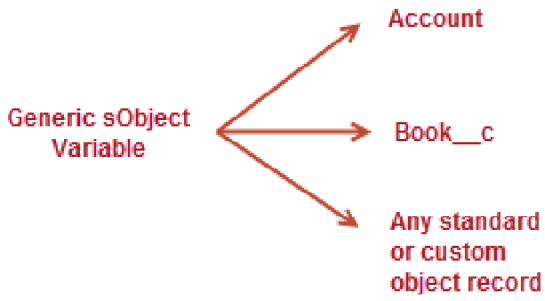
```
Account acct = new Account(Name='Acme');
```

Alternatively, you can use the dot notation to add fields to an sObject. The following is equivalent to the previous example, although it takes a few more lines of code.

```
Account acct = new Account();
acct.Name = 'Acme';
acct.Phone = '(415)555-1212';
acct.NumberOfEmployees = 100;
```

Working with the Generic sObject Data Type

Variables that are declared with the generic sObject data type can reference any Salesforce record, whether it is a standard or custom object record.



The example shows how the generic sObject variable can be assigned to any Salesforce object: an account and a custom object called Book__c.

```
sObject sobj1 = new Account(Name='Trailhead');
sObject sobj2 = new Book__c(Name='Workbook 1');
```

Casting Generic sObjects to Specific sObject Types

One of the benefits of doing so is to be able to access fields using dot notation, which is not available on the generic sObject. Since sObject is a parent type for all specific sObject types, you can cast a generic sObject to a specific sObject. This example shows how to cast a generic sObject to Account.

```
// Cast a generic sObject to an Account
Account acct = (Account)myGenericSObject;
// Now, you can use the dot notation to access fields on Account
String name = acct.Name;
```

Using the with sharing or without sharing Keywords

Use the with sharing or without sharing keywords on a class to specify whether or not to enforce sharing rules.

The with sharing keyword allows you to specify that the sharing rules for the current user be taken into account for a class. You have to explicitly set this keyword for the class because Apex code runs in 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.

Use the with sharing keywords when declaring a class to enforce the sharing rules that apply to the current user. For example:

```
public with sharing class sharingClass {
// Code here
}
```

Some things to note about sharing keywords:

- The sharing setting of the class where the method is defined is applied, not of the class where the method is called.
- If a class isn't declared as either with or without sharing, the current sharing rules remain in effect. For example, if the class is called by another class that has sharing enforced, then sharing is enforced for the called class.
- Both inner classes and outer classes can be declared as with sharing. The sharing setting applies to all code contained in the class, including initialization code, constructors, and methods.
- Inner classes do **not** inherit the sharing setting from their container class.
- Classes inherit this setting from a parent class when one class extends or implements another

Abstract class

It is a class which has both defined(with body) & non defined Methods. If a class is extending Abstract Class, then it needs(required) to implement all such methods of abstract Class , defined with abstract keyword

```
public abstract class AreaofShapes
{
public decimal rad = 0.8;
Public integer length ;
public decimal area{get;set;}

Public AreaofShapes(decimal length) {
// assume our default area calculation is length x length
area = length *length ;
}
//Note : if this class has parameterized constructor,
//then it must have 'no parameter constructor'
public AreaofShapes()
{ }
//Note : Abstract class can have Non abstract methods,
//same not true for interfaces ;
public Decimal circleArea(integer r)
{
area = (r)*(r)*(Math.acos(-1)); // Pi = Math.acos(-
1) return area;
}
// Note: Abstract methods must not define a body
//( not even curly braces), else error //will come up
  // These below methods must be overridden in
class //implementing this abstract class
public abstract decimal RectangleArea(decimal length, decimal width);
public abstract decimal hexagonArea(decimal length);
}

//Now Example of class extending abstract Class
public class calculateArea extends AreaofShapes
{
// Note :This class will have automatic access of variables
// of Abstract class
public calculateArea ()
{
//Calling constructor of parent abstract class using super keyword
super(5);
}
```

```
// Calling non abstract method of parent
public decimal Findcirclearea(integer r) {
return circleArea(r);
}
// This class must use override keyword for abstract methods.
// Note area variable not defined in this class and still
 used //as it is defined in //parent.
public override decimal RectangleArea(decimal length, decimal
width) area = length * width ;
return area;
}
public override decimal hexagonArea(decimal
length){ area = 6*length;
return area;
}
//this class can have its own methods also
public decimal calculateSquareArea() {
// I can't just call Constructor of Parent in child
//in any method ,/ super needs to be present in
constructor //of child
// area = super(length)// this line will give error,if uncommented
// area calculated by constructor, will be returned
// from below statement.
return area ;
}
}
```

Virtual Class

It also has defined and non defined methods. Here also, Other classes ***'extends'***virtual class but they '**don't** need to implement all methods of virtual class, defined with virtual keyword.

Example :

```
public virtual class Player {

String name;

//Note parametrized constructor is present
public Player(String name) {
this.name = name;
}
```

```
// If parametrized constructor is present, you need to define
// non parametrized one also.
public Player ()
{ }
// well defined method without virtual keyword
Public String getName(){
return this.Name;
}
// Method with virtual Keyword. Note, it also has body unlike
// abstract class or interface
public virtual String getGreeting(){
return 'Are you ready to play, ' + this.getName() + '?';
}
}
```

Now giving example of class extending virtual class

```
public class footballPlayer extends Player
{
// Note even if this class body is empty, this can be saved,
// so it means class extending virtual class NEEDN'T implements
// all methods of parent defined with virtual Keyword
//But for abstract class, if method is defined as abstract,
//it NEEDS to be implemented

public footballPlayer (String name){
super(name);
}
// its my choice, to override a virtual method of parent
class. //its not necessary.See use of override keyword.
public override String getGreeting(){
return 'It\'s a great day for football, ' + this.getName() + '!';
}
}
```

Interfaces

An *interface* is like a class in which none of the methods have been implemented—the method signatures are there, but the body of each method is empty. To use an interface, another class must implement it by providing a body for all of the methods contained in the interface.

Defining an interface is similar to defining a new class. For example, a company might have two types of purchase orders, ones that come from customers, and others that come from their employees. Both are a type of purchase order.

Suppose you needed a method to provide a discount. The amount of the discount can depend on the type of purchase order.

You can model the general concept of a purchase order as an interface and have specific implementations for customers and employees. In the following example the focus is only on the discount aspect of a purchase order.

This is the definition of the PurchaseOrder interface.

```
// An interface that defines what a purchase order looks like in general
public interface PurchaseOrder {

   // All other functionality
   excluded Double discount();

}
```

This class implements the PurchaseOrder interface for customer purchase orders.

```
// One implementation of the interface for customers
public class CustomerPurchaseOrder implements PurchaseOrder {
   public Double discount() {
      return .05; // Flat 5% discount
   }
}
```

This class implements the PurchaseOrder interface for employee purchase orders.

```
// Another implementation of the interface for employees
public class EmployeePurchaseOrder implements PurchaseOrder {
    public Double discount() {
      return .10; // It's worth it being an employee! 10% discount
    }
 }
```

Note:(Abstract class, Virtual Class & Interface)

1. Abstract class has variables as well properties; these are not present in interface.
2. Abstract class can have well defined methods , which can't be present in Interfaces.
3. If method is defined as abstract, it can't have body— same as interface but different from virtual class. In virtual class, method defined as virtual, can have body .
4. If method is defined as abstract, **it needs(required)** to be implemented, same as interface, but different from virtual class. In virtual class, you may or may not implement virtual methods.
5. In class extending Abstract Class, you use 'override keyword(same way as in virtual class) to override method defined as 'Abstract' in parent Abstract Class.
6. Virtual class can have variables unlike interfaces.
7. virtual class can be empty without any methods.
8. Virtual class can have well defined methods including constructors.
9. Classes extending virtual class are not required to implement methods defined wi th virtual keyword in Parent.
10. We use override keyword to override any method defined in parent virtual class.
11. It can have both defined and non defined methods but generally has well defined methods. These methods can exist with or without virtual keyword.
12. Interfaces doesn't have methods with body
13. Interfaces doesn't have variables
14. Methods of interfaces doesn't have access modifiers(Public private etc).

DML

Manipulating Records with DML

Create and modify records in Salesforce by using the Data Manipulation Language, abbreviated as DML. DML provides a way to manage records by providing simple statements to insert, update, merge, delete, and restore records.

This example adds the Acme account to Salesforce. An account sObject is created first and then passed as an argument to the insert statement, which persists the record in Salesforce.

```
Account acct = new Account(Name='Acme', Phone='(415)555-1212',
NumberOfEmployees=100);

Insert acct;
```

DML Statements

The following DML statements are available.

- insert
- update
- upsert
- delete
- undelete
- merge

Each DML statement accepts either a single sObject or a list (or array) of sObjects. Operating on a list of sObjects is a more efficient way for processing records.

ID Field Auto-Assigned to New Records

When inserting records, the system assigns an ID for each record. In addition to persisting the ID value in the database, the ID value is also autopopulated on the sObject variable that you used as an argument in the DML call.

```
Account acct = new Account(Name='Acme', Phone='(415)555-1212',
NumberOfEmployees=100);
insert acct;

// Get the new ID on the inserted sObject argument
ID acctID = acct.Id;
// Display this ID in the debug log
System.debug('ID = ' + acctID);
```

You can reuse this sObject variable to perform further DML operations, such as updates, as the system will be able to map the sObject variable to its corresponding record by matching the ID.

Bulk DML

You can perform DML operations either on a single sObject, or in bulk on a list of sObjects. Performing bulk DML operations is the recommended way because it helps avoid hitting governor limits, such as the DML limit of 150 statements per Apex transaction.

Upserting Records

If you have a list containing a mix of new and existing records, you can process insertions and updates to all records in the list by using the upsert statement. Upsert helps avoid the creation of duplicate records

The upsert statement matches the sObjects with existing records by comparing values of one field. If you don't specify a field when calling this statement, the upsert statement uses the sObject's ID to match the sObject with existing records in Salesforce. Alternatively, you can specify a field to use for matching. For custom objects, specify a custom field marked as external ID. For standard objects, you can specify any field that has the idLookup property set to true.

Upsert Syntax

upsert *sObject | sObject[]*

upsert *sObject | sObject[] field*

- If the key is not matched, a new object record is created.
- If the key is matched once, the existing object record is updated.
- If the key is matched multiple times, an error is generated and the object record is neither inserted or updated.

Deleting Records

You can delete persisted records using the delete statement. Deleted records aren't deleted permanently fromForce.com, but they're placed in the Recycle Bin for 15 days from where they can be restored.

```
Contact[] contactsDel = [SELECT Id FROM Contact WHERE LastName='Smith'];
delete contactsDel;
```

DML Statement Exceptions

If a DML operation fails, it returns an exception of type DmlException. You can catch exceptions in your code to handle error conditions.

```
try {
   // This causes an exception because
   //   the required Name field is not
   provided. Account acct = new Account();
   // Insert the account
   insert acct;
} catch (DmlException e) {
   System.debug('A DML exception has occurred: '
           + e.getMessage());
}
```

Database Methods

Apex contains the built-in Database class, which provides methods that perform DML operations and mirror the DML statement counterparts.

These Database methods are static and are called on the class name.

- Database.insert()
- Database.update()
- Database.upsert()
- Database.delete()
- Database.undelete()
- Database.merge()

Unlike DML statements, Database methods have an

optional allOrNone booleanparameter that allows you to specify whether the operation should partially succeed. When this parameter is set to false, if errors occur on a partial set of records, the successful records will be committed and errors will be returned for the failed records. Also, no exceptions are thrown with the partial success option.

```
Database.insert(recordList, false);
```

The Database methods return result objects containing success or failure information for each record. For example, insert and update operations each return an array of Database.SaveResult objects.

```
Database.SaveResult[] results = Database.insert(recordList, false);
```

Upsert returns Database.UpsertResult objects, and
delete returns Database.DeleteResult objects.

By default, the allOrNone parameter is true, which means that the Database method behaves like its DML statement counterpart and will throw an exception if a failure is encountered.

```
Database.insert(recordList);
```

And:

```
Database.insert(recordList, true);
```

Example: Inserting Records with Partial Success

```
List<Contact> conList = new List<Contact> {

    new Contact(FirstName='Joe',LastName='Smith',Department='Finance'),
```

```
        new Contact(FirstName='Kathy',LastName='Smith',Department='Technol
ogy'),

        new Contact(FirstName='Caroline',LastName='Roth',Department='Financ
e'),

        new Contact()};


Database.SaveResult[] srList = Database.insert(conList, false);

for (Database.SaveResult sr : srList) {

   if (sr.isSuccess()) {

      System.debug('Successfully inserted contact. Contact ID: ' +

   sr.getId()); } else {

      for(Database.Error err : sr.getErrors()) { System.debug('The

         following error has occurred.');

         System.debug(err.getStatusCode() + ': ' + err.getMessage());

         System.debug('Contact fields that affected this error: ' +
err.getFields());

    }

   }

}
```

Should You Use DML Statements or Database Methods?

- Use DML statements if you want any error that occurs during bulk DML processing to be thrown as an Apexexception that immediately interrupts control flow (by using try. . .catch blocks)

Working with Related Records

Create and manage records that are related to each other through relationships.

Inserting Related Records

You can insert records related to existing records if a relationship has already been defined between the two objects, such as a lookup or master-detail relationship. A record is associated with a related record through a foreign key ID. For example, if inserting a new contact, you can specify the contact's related account record by setting the value of theAccountId field.

```
Account acct = new Account(Name='SFDC Account');

insert acct;

ID acctID = acct.ID;

Contact mario = new Contact(

  FirstName='Mario',

  LastName='Ruiz',

  Phone='415.555.1212',

  AccountId=acctID);
```

```
insert mario;
```

Updating Related Records

Fields on related records can't be updated with the same call to the DML operation and require a separate DML call. For example, if inserting a new contact, you can specify the contact's related account record by setting the value of theAccountId field. However, you can't change the account's name without updating the account itself with a separate DML call. Similarly, when updating a contact, if you also want to update the contact's related account, you must make two DML calls.

```
// Query for the contact, which has been associated with an account.
Contact queriedContact = [SELECT Account.Name
                          FROM Contact
                           WHERE FirstName = 'Mario' AND LastName='Ruiz'
                           LIMIT 1];


queriedContact..Phone = '(415)555-1213';

queriedContact..Account.Industry = 'Technology';

update queriedContact;

update queriedContact.Account;
```

Deleting Related Records

The delete operation supports cascading deletions. If you delete a parent object, you delete its children automatically, as long as each child record can be deleted.

```
Account[] queriedAccounts = [SELECT Id FROM Account WHERE Name='SFDC Account'];

delete queriedAccounts;
```

AboutTransactions

DML operations execute within a transaction. All DML operations in a transaction either complete successfully, or if an error occurs in one operation, the entire transaction is rolled back and no data is committed to the database. The boundary of a transaction can be a trigger, a class method, an anonymous block of code, an Apex page, or a custom Web service method.

SOQL(Salesforce Object Query Language)

To read a record from Salesforce, you need to write a query. Salesforce provides the Salesforce Object Query Language, or SOQL in short, that you can use to read saved records.

When SOQL is embedded in Apex, it is referred to as *inline SOQL*.

To include SOQL queries within your Apex code, wrap the SOQL statement within square brackets and assign the return value to an array of sObjects.

```
Account[] accts = [SELECT Name,Phone FROM Account];
```

Prerequisites

Some queries in this unit expect the org to have accounts and contacts. Before you run the queries, create some sample data.

```
// Add account and related contact
Account acct = new Account(
   Name='SFDC Computing',
   Phone='(415)555-1212',
   NumberOfEmployees=50,
   BillingCity='San Francisco');
insert acct;

// Once the account is inserted, the sObject will be
// populated with an ID.
// Get this ID.
ID acctID = acct.ID;

// Add a contact to this account.
```

```
Contact con = new Contact(
   FirstName='Carol',
   LastName='Ruiz',
   Phone='(415)555-1212',
   Department='Wingo',
   AccountId=acctID);
insert con;


// Add account with no contact
Account acct2 = new Account(
   Name='The SFDC Query Man',
   Phone='(310)555-1213',
   NumberOfEmployees=50
,
   BillingCity='Los Angeles', Description='Expert
   in wing technologies.');
insert acct2;
```

To run SOQL query Using the Query Editor

The Developer Console provides the Query Editor console, which enables you to run your SOQL queries and view results. The Query Editor provides a quick way to inspect the database. It is a good way to test your SOQL queries before adding them to your Apex code.

```
SELECT Name,Phone FROM Account
```

Basic SOQL Syntax

This is the syntax of a basic SOQL query:

```
SELECT fields FROM ObjectName [WHERE Condition]
```

The WHERE clause is optional.

For example, the following query retrieves accounts and gets two fields for each account: the ID and the Phone.

```
SELECT Phone FROM Account
```

Unlike other SQL languages, you can't specify * for all fields. You must specify every field you want to get explicitly. If you try to access a field you haven't specified in the SELECT clause, you'll get an error because the field hasn't been retrieved.

You don't need to specify the Id field in the query as it is always returned in Apex queries, whether it is specified in the query or not.

For example:

```
SELECT Id,Phone FROM Account
```

and

```
SELECT Phone FROM Account
```

are equivalent statements.

The only time you may want to specify the Id field if it is the only field you're retrieving because you have to list at least one field:

```
SELECT Id FROM Account
```

You may want to specify the Id field also when running a query in the Query Editor as the ID field won't be displayed unless specified.

Filtering Query Results with Conditions

If you have more than one account in the org, they will all be returned. If you want to limit the accounts returned to accounts that fulfill a certain condition, you can add this condition inside the WHERE clause.

```
SELECT Name,Phone FROM Account WHERE Name='SFDC Computing'

SELECT Name,Phone FROM Account WHERE (Name='SFDC Computing' AND
NumberOfEmployees>25)

SELECT Name,Phone FROM Account WHERE (Name='SFDC Computing' OR
(NumberOfEmployees>25 AND BillingCity='Los Angeles'))
```

For example, you can retrieve all accounts whose names start with SFDC by using this condition: WHERE Name LIKE 'SFDC%'. The % wildcard character matches any or no character. The _ character in contrast can be used to match just one character.

Ordering Query Results

When a query executes, it returns records from Salesforce in no particular order

```
SELECT Name,Phone FROM Account ORDER BY Name
```

The previous statement is equivalent to:

```
SELECT Name,Phone FROM Account ORDER BY Name ASC
```

To reverse the order, use the DESC keyword for descending order:

```
SELECT Name,Phone FROM Account ORDER BY Name DESC
```

You can sort on most fields, including numeric and text fields. You can't sort on fields like rich text and multi-select picklists.

Limiting the Number of Records Returned

You can limit the number of records returned to an arbitrary number by adding the LIMIT n clause where n is the number of records you want returned.

```
Account oneAccountOnly = [SELECT Name,Phone FROM Account LIMIT 1];
```

You can't use a LIMIT clause in a query that uses an aggregate function, but does not use a GROUP BY clause. For example, the following query is invalid:

SELECT MAX(CreatedDate) FROM Account LIMIT 1

OFFSET is intended to be used in a top-level query, and is not allowed in most sub-queries, so the following query is invalid and will return a MALFORMED_QUERY error:

SELECT Name, Id FROM Merchandise__c WHERE Id IN(SELECT Id FROM Discontinued_Merchandise__c

LIMIT 100 OFFSET 20) ORDER BY Name

A sub-query can use OFFSET only if the parent query has a LIMIT 1 clause. The following query is a valid use of OFFSET in

SELECT Name, Id(SELECT Name FROM Opportunity LIMIT 10 OFFSET 2) FROM Account ORDER BY Name LIMIT 1

OFFSET cannot be used as a sub-query in the WHERE clause, even if the parent query uses LIMIT 1.

Combining All Pieces Together

You can combine the optional clauses in one query, in the following order:

```
SELECT Name,Phone FROM Account
        WHERE (Name = 'SFDC Computing' AND NumberOfEmployees>25)
        ORDER BY Name
        LIMIT 10
```

Execute the following SOQL query in Apex by using the Execute Anonymous window in the Developer Console. Then inspect the debug statements in the debug log. One sample account should be returned.

```
Account[] accts = [SELECT Name,Phone FROM Account
                   WHERE (Name='SFDC Computing' AND NumberOfEmployees>25)
                   ORDER BY Name
                   LIMIT 10];
System.debug(accts.size() + ' account(s) returned.');
// Write all account array info
```

SubQueries

The subquery can filter by ID (primary key) or reference (foreign key) fields. A semi-join is a subquery on another object in an IN clause to restrict the records returned. An anti-join is a subquery on another object in a NOT IN clause to restrict the records returned.

Sample uses of semi-joins and anti-joins include:

• Get all contacts for accounts that have an opportunity with a particular record type.

• Get all open opportunities for accounts that have active contracts.

Get all open cases for contacts that are the decision maker on an opportunity.

• Get all accounts that do not have any open opportunities.

If you filter by an ID field, you can create parent-to-child semi- or anti-joins, such as Account to Contact. If you filter by a reference field, you can also create child-to-child semi- or anti-joins, such as Contact to Opportunity, or

child-to-parent semi- or anti-joins, such as Opportunity to Account.

The following restrictions apply to the main WHERE clause of a semi-join or anti-join query:

The left operand must query a single ID (primary key) or reference (foreign key) field. The selected field in a subquery can be a reference field.

For example:

SELECT Id FROM Idea WHERE (Id IN (SELECT ParentId FROM Vote WHERE CreatedDate > LAST_WEEK AND Parent.Type='Idea'))

– The left operand can't use relationships. For example, the following semi-join query is invalid due to the Account.Id relationship field:

SELECT Id FROM Contact WHERE Account.Id IN(SELECT ...)

No more than two IN or NOT IN statements per WHERE clause.

A subquery must query a field referencing the same object type as the main query.

The selected column in a subquery must be a foreign key field, and cannot traverse relationships. This limit means that you cannot use dot notation in a selected field of a subquery. For example, the following query is valid:

SELECT Id, Name FROM Account WHERE Id IN (SELECT AccountId FROM Contact WHERE LastName LIKE 'Brown_%')

You cannot use subqueries with OR.

COUNT, FOR UPDATE, ORDER BY, and LIMIT are not supported in subqueries.

Group By

You can use the GROUP BY option in a SOQL query to avoid iterating through individual query results. That is, you specify a group of

records instead of processing many individual records.

The syntax is:

[GROUP BY ***fieldGroupByList***]

*fieldGroupByList*specifies a list of one or more fields, separated by commas, that you want to group by. If the list of fields in a SELECT clause includes an aggregate function, you must include all non-aggregated fields in the GROUP BY clause.

For example, to determine how many leads are associated with each LeadSource value use GROUP BY.

SELECT LeadSource, COUNT(Name) FROM Lead GROUP BY LeadSource

You must use a GROUP BY clause if your query uses a LIMIT clause and an aggregated function. For example, the following query is valid:

SELECT Name, Max(CreatedDate) FROM Account GROUP BY Name LIMIT 5

The following query is invalid as there is no GROUP BY clause:

SELECT MAX(CreatedDate) FROM Account LIMIT 1

You can't use child relationship expressions that use the __r syntax in a query that uses a GROUP BY clause

You can use an alias for any field or aggregated field in a SELECT statement in a SOQL query. Use a field alias to identify the field when you're processing the query results in your code.

For example:

SELECT Name n, MAX(Amount) max FROM Opportunity GROUP BY Name

Any aggregated field in a SELECT list that does not have an alias automatically gets an implied alias with a format expr***i***, where *I* denotes the order of the aggregated fields with no explicit aliases. The

value of *i* starts at 0 and increments for every aggregated field with no explicit alias.

In the following example, MAX(Amount) has an implied alias of expr0, and MIN(Amount) has an implied alias of expr1.

SELECT Name, MAX(Amount), MIN(Amount) FROM Opportunity GROUP BY Name

In the next query, MIN(Amount) has an explicit alias of min. MAX(Amount) has an implied alias of expr0, and SUM(Amount) has an implied alias of expr1.

SELECT Name, MAX(Amount), MIN(Amount) min, SUM(Amount) FROM Opportunity GROUP BY Name

Use the GROUP BY ROLLUP optional clause in a SOQL query to add subtotals for aggregated data in query results. You can include up to three fields in a comma-separated list in a GROUP BY ROLLUP clause.
This simple example rolls the results up by one field:

SELECT LeadSource, COUNT(Name) cnt FROM Lead GROUP BY ROLLUP(LeadSource)

This example rolls the results up by two fields:

GROUPING(***fieldName***) function identifies whether the row is a subtotal for a field.
SELECT Status, LeadSource, COUNT(Name) cnt FROM Lead GROUP BY ROLLUP(Status, LeadSource)

GROUPING(***fieldName***) returns 1 if the row is a subtotal for the field, and 0 otherwise.

SELECT LeadSource, Rating, GROUPING(LeadSource) grpLS, GROUPING(Rating) grpRating, COUNT(Name) cnt

FROM Lead GROUP BY ROLLUP(LeadSource, Rating)

HAVING is an optional clause that can be used in a SOQL query to filter results that aggregate functions return.

For example:

SELECT LeadSource, COUNT(Name) FROM Lead GROUP BY LeadSource HAVING COUNT(Name) > 100

The next query returns accounts with duplicate names:

SELECT Name, Count(Id) FROM Account GROUP BY Name HAVING Count(Id) >

The following query is invalid as City is not included in the GROUP BY clause:

SELECT LeadSource, COUNT(Name) FROM Lead GROUP BY LeadSource HAVING COUNT(Name) > 100 and City LIKE 'San%'

A HAVING clause can't contain any semi- or anti-joins.

Use aggregate functions in a GROUP BY clause in SOQL queries to generate reports for analysis. Aggregate functions include AVG(), COUNT(), MIN(), MAX(), SUM(), and more.

You can also use aggregate functions *without* using a GROUP BY clause. For example, you could use the AVG() aggregate function to find the average Amount for all your opportunities.

SELECT AVG(Amount) FROM Opportunity

To find the average Amount for all your opportunities by campaign.

SELECT CampaignId, AVG(Amount) FROM Opportunity GROUP BY CampaignId

Note the following when using COUNT():

- COUNT() must be the only element in the SELECT list.
- You can use COUNT() with a LIMIT clause.
- You can't use COUNT() with an ORDER BY clause. Use COUNT(***fieldName***) instead.
- You can't use COUNT() with a GROUP BY clause

Accessing Variables in SOQL Queries

SOQL statements in Apex can reference Apex code variables and expressions if they are preceded by a colon (:). The use of a local variable within a SOQL statement is called a *bind*.

This example shows how to use the targetDepartment variable in the WHERE clause.

```
String targetDepartment = 'Sales';
Contact[] techContacts = [SELECT FirstName,LastName
                          FROM Contact WHERE Department=:targetDepartment];
```

Querying Related Records

Relationship queries are similar to SQL joins. However, you cannot perform arbitrary SQL joins. The relationship queries in SOQL must traverse a valid relationship path.There must be a parent-to-child or child-to-parent relationship connecting the objects.

To get child records related to a parent record, add an inner query for the child records. The FROM clause of the inner query runs against the relationship name, rather than a Salesforce object name.

This example contains an inner query to get all contacts that are associated with each returned account.

SELECT Name, (SELECT LastName FROM Contacts) FROM Account WHERE Name = 'SFDC Computing'

This next example embeds the example SOQL query in Apex and shows how to get the child records from the SOQL result by using the Contacts relationship name on the sObject.

```
List<Account> listAcc = [SELECT Name, (SELECT FirstName,LastName FROM Contacts)FROM Account
WHERE Name = 'SFDC Computing'];
for(Account a1:listAcc){
system.debug('account '+a1.name);
List<Contact> cts = a1.Contacts;
        integer i=1;
for(contact c1:cts)
System.debug(' Name of associated contact '+ (i++) +' : ' + c1.FirstName + ', ' + c1.LastName);
}
```

You can traverse a relationship from a child object (contact) to a field on its parent (Account.Name) by using dot notation. For example, the following Apex snippet queries contact records whose first name is Carol and is able to retrieve the name of Carol's associated account by traversing the relationship between accounts and contacts.

```
Contact[] cts = [SELECT Account.Name FROM Contact
            WHERE FirstName = 'Carol' AND LastName='Ruiz'];
Contact carol = cts[0];
String acctName = carol.Account.Name;
System.debug('Carol\'s account name is ' + acctName);
```

When you design SOQL relationship queries, there are several limitations to consider.

- Relationship queries are not the same as SQL joins. You must have a relationship between objects to create a join in SOQL.
- No more than 35 child-to-parent relationships can be specified in a query. A custom object allows up to 25 relationships, so can reference all the child-to-parent relationships for a custom object in one query.
- No more than 20 parent-to-child relationships can be specified in a query.

- In each specified relationship, no more than five levels can be specified in a child-to-parent relationship. For example, Contact.Account.Owner.FirstName (three levels).

- In each specified relationship, only one level of parent-to-child relationship can be specified in a query.

Querying Record in Batches By Using SOQL For Loops

With a SOQL for loop, you can include a SOQL query within a for loop. The results of a SOQL query can be iterated over within the loop.

SOQL for loops iterate over all of the sObject records returned by a SOQL query. The syntax of a SOQL for loop is either:

```
for (variable : [soql_query]) {

    code_block

}
```

or

```
for (variable_list : [soql_query]) {

    code_block

}
```

Both *variable* and *variable_list* must be of the same type as the sObjects that are returned by the *soql_query*.

SOSL (Salesforce Object Search Language)

Salesforce Object Search Language (SOSL) is a Salesforce search language that is used to perform text searches in records. Use SOSL to search fields across multiple standard and custom object records in Salesforce.

Using the Query Editor

The Developer Console provides the Query Editor console, which enables you to run SOSL queries and view results.

Basic SOSL Syntax

This is the syntax of a basic SOSL query:

```
FIND 'SearchQuery' [IN SearchGroup] [RETURNING ObjectsAndFields]
```

SearchQuery is the text to search for (a single word or a phrase). Search terms can be grouped with logical operators (AND, OR) and parentheses. Also, search terms can include wildcard characters (*, ?). The * wildcard matches zero or more characters at the middle or end of the search term. The ?wildcard matches only one character at the middle or end of the search term.

SearchGroup is optional. It is the scope of the fields to search. If not specified, the default search scope is all fields.*SearchGroup* can take one of the following values.

ALL FIELDS

NAME FIELDS

EMAIL FIELDS

PHONE FIELDS

SIDEBAR FIELDS

ObjectsAndFields is optional. It is the information to return in the search result—a list of one or more sObjects and, within each sObject, list of one or more fields,

with optional values to filter against. If not specified, the search results contain the IDs of all objects found.

Single Words and Phrases

A *SearchQuery* contains two types of text:

Single Word— single word, such as test or hello. Words in the SearchQuery are delimited by spaces, punctuation, and changes from letters to digits (and vice-versa). Words are always case insensitive.

Phrase— collection of words and spaces surrounded by double quotes such as "john smith". Multiple words can be combined together with logic and grouping operators to form a more complex query.

SOSL Apex Example

```
FIND {Wingo} IN ALL FIELDS RETURNING Account(Name),
Contact(FirstName,LastName,Department)
```

The SOSL query returns records that have fields whose values matches Wingo. Based on our sample data, only the contact has a field with the value Wingo, so this contact is returned..

The search query in the Query Editor and the API must be enclosed within curly brackets ({Wingo}). In contrast, in Apex the search query is enclosed within single quotes ('Wingo').

This is an example of a SOSL query that searches for accounts and contacts that have any fields with the word 'SFDC'.

```
List<List<Sobject>> lobs=[find 'abc@abc.com' in all fields
returning Account(Name), Lead(name,leadsource), contact(firstname,lastname,phone)];


List<Account> lacc=(List<Account>)lobs[0];
List<Lead> llead=(List<Lead>)lobs[1];
List<Contact> lcnt=(List<Contact>)lobs[2];
system.debug('Account detail.....');
for(Account a1:lacc)
```

```
system.debug(a1.Name);
system.debug('Lead detail.....');
for(Lead l1:llead)
system.debug(l1.name+'--'+l1.leadsource);
system.debug('Contact detail.....');
for(Contact c1:lcnt)
system.debug(c1.firstname + ' '+ c1.lastname+'----'+c1.phone);
```

Differences and Similarities Between SOQL and SOSL

Use SOQL when:
• You want to search against the org's database. Results from a database search include matches for the exact search string.
• You know in which objects or fields the data resides.
• You want to:
– Retrieve data from a single object or from multiple objects that are related to one another
– Count the number of records that meet specified criteria
– Sort results as part of the query
– Retrieve data from number, date, or checkbox fields

Use SOSL when:
• You don't know in which object or field the data resides, and you want to find it in the most efficient and fastest way possible.
• SOSL searches also take advantage of the advanced features of the search index, such as out-of-order matching, synonyms, lemmatization, and spell check.
– Retrieve multiple objects and fields efficiently, and the objects might or might not be related to one another.

If your searches are too general, they are slow and return too many results. Use the following clauses to write more targeted and useful searches.
• IN: Limits the types of fields to search, including email, name, or phone.
• LIMIT: Specifies the maximum number of rows to return.
• OFFSET: Displays the search results on multiple pages.
• RETURNING: Limits the objects and fields to return

Exercises

1. Find out all name from contact where name start with letter 'A' and mailingstate equals 'california'.
2. Find all account where created date is after 26th april 2011.
3. Find amount for all opportunity which has created in year 2011;
4. Fnd detail of all cases which has contacts lastname as null

5. Find account IDs of all events with a non-null activity date (use Event object).
6. Find out all name from account where billingState is from California or new York.

Semi join and anti join

Semi join with ID

7. Find out all account where which has opportunity as close lost

Anti join with ID

8. Find all account that do no have any open opportunity.

Anti join with reference field

9. Find out opportunity detail for all contact whose leadsource is not web(use leadsource field of contact.

Multiple semi join & anti join

10. Find out all account that have open opportunity and their last name should be same as last name of account 'Apple'.
11. Find out opportunity Id and their related line Items (use Opportunity and OpportunityLineItems object) for all opportunity whose line items total value is more than 10000.
12. Find Account records in alphabetical order by first name, sorted in descendingorder, with accounts that have null names appearing last:
13. Find out 125 name of account where industry is media, sorted in ascending order with null coming last.
14. Find out top 10 account name on the basis of annualrevenue.
15. Find out 5-10 account name on the basis of annualrevenue. Skip first four record. (Use Offset)
16. Find Average amount for all opportunity .
17. Find average amount for all opportunity based on campaign.
18. find the sum of the Amount values for all your opportunities for each calendar year.

Apex Trigger

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

Typically, you use triggers to perform operations based on specific conditions, to modify related records or restrict certain operations from happening.

Triggers can be defined for top-level standard objects, such as Account or Contact, custom objects, and some standard child objects. Triggers are active by default when created.

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

before insert

before update

before delete

after insert

after update

after delete

after undelete

Types of Triggers

There are two types of triggers.
Before triggers are used to update or validate record values before they're saved to the database.

After triggers are used to access field values that are set by the system (such as a record's Id or LastModifiedDatefield), and to effect changes in other records**.**

Trigger Syntax

```
trigger TriggerName on ObjectName (trigger_events) {

   code_block

}
```

Ex.

```
Trigger Sample on Account(before insert,after delete)

{

Code block

}
```

In trigger you do not have to commit data manually, it automatically saves into database.

Using Context Variables

To access the records that caused the trigger to fire, use context variables. For example, Trigger.New contains all the records that were inserted in insert or update triggers. Trigger.Old provides the old version of sObjects before they were updated in update triggers, or a list of deleted sObjects in delete triggers.

Triggers can fire when one record is inserted, or when many records are inserted in bulk via the API or Apex. Therefore, context variables, such asTrigger.New, can contain only one record or multiple records. You can iterate over Trigger.New to get each individual sObject.

Trigger.NEW

It is a context variable which contains list of records which has caused the trigger to fire.

Trigger.NEW can be used in following trigger events.

Before insert

Before update

After insert

After update

After undelete

Trigger.NEW can't be used in delete operations.

Trigger.NEW in Before insert

Suppose we have an Customer object with three records:

Cid	Name	Age	Phone
111	aaa	23	1234
222	bbb	34	3455
333	ccc	45	9876

And we are trying to insert new records into customer object as:

444	ddd	34	1234
555	eee	23	3456

Then the new record that we are trying to insert are stored in Trigger.New in before insert event. I.e.

If we write

List<Customer__c> ust=Trigger.New;

And if we display the value of cust List, it will show,

444	ddd	34	1234
555	eee	23	3456

We can directly perform changes in records stored in Trigger.NEW in before insert event.

For eg:

```
Trigger Sample1 on Customer__c(before insert)

{

For(Customer__c c:Trigger.NEW)

{

c.Age=30;}}
```

Before insert event occurs before new records are inserted into the database, so we can not retrieve the new record using dml operations in before trigger.

For eg:

When we have customer object with following records:

Cid	Name	Age	Phone
111	aaa	23	1234
222	bbb	34	3455
333	ccc	45	9876

And if we are trying to insert two new records as:

444	ddd	34	1234
555	eee	23	3456

If there is any before insert trigger where we have written any dml statement as:

Trigger example on customer__C(before insert)

{

List<Customer__c> my=[select cid__c,name,Age__c,phone__c from customer__c];

// this query will fetch only three records as remaining two records are not yet inserted.

}

Before Insert

These triggers fires when we try to insert a new record into an object.

All the statement written in trigger block are executed before new records are saved in database.

In these event Trigger.NEW stores the new record that we are trying to insert.

Eg:

1. when inserting new Account record if acount with same name exist, we should prevent the insert operation.

```
triggerAccountInsert on Account (before insert) {

    List<String> names=new List<String>();

for(Account a1:Trigger.new)

names.add(a1.name);

   List<Account> la=[select name from account where name in :names];

for(Account a2:Trigger.new)
```

```
  {   for(Account a3:la)
      {if(a3.name==a2.name)
a2.name.addError('duplicate accounts....');
a2.addError('duplicate accounts....');
      }
  }}
```

TestCases:

```
@isTest
Public class AcountInsert
{
Public static testMethod void testinsert()
{
String addError;
String myname='madhuri';
Account a1=new Account(name=myname);
List<Account> x=[select name from Account where name=:myname];
If(x.size()<1)
{
System.assertEquals(0,x.size());
Insert a1;
}
Else
```

```
{
addError='Existing';
}
System.assertEquals('Existing',addError);
}
}}
```

After Insert

This triggers are fired when new records are successfully saved in the database.

Trigger.new is used to refer to the list of new records which were inserted.

On the new list of records we can perform DML operations.

For eg:

Suppose we have a Customer object with three records:

Cid	Name	Age	Phone
111	aaa	23	1234
222	bbb	34	3455
333	ccc	45	9876

And when we insert new records into customer object as:

444	Ddd	32	3456
555	Eee	56	7655

Opertaions written in trigger body are executed after the records are saved into database as

Cid	Name	Age	Phone

111	aaa	23	1234
222	bbb	34	3455
333	ccc	45	9876
444	ddd	32	3456
555	eee	56	7655

Ex:

When a new contact is created for any account, update the account phone with contact phone

```
triggerUpdatePhone on Contact (before insert,before update)
   { List<String> s1=new list<string>();
for(contact c1:trigger.new)
   {
s1.add(c1.accountid);
   }
   List<Account> acc=[select id from account where id in:s1];
  List<Account> ins=new List<account>();
for (account x:acc)
   {
for(contact c:trigger.new)
   {
if(x.id==c.accountid)
      {
         x.phone=c.phone;
```

```
        ins.add(x);
      }
   }
   }
update ins;
}
```

Test case:

@isTest

```
@isTest
Public class TestPhone{
Static testmethod void updatePhone(){
Account a=new Account(name='sam',phone='103');
Insert a;
Contact c=new Contact(lastname='kumar',accountId=a.id,phone='456');
Insert c;
Account acc=[select id,phone from Account where id=:c.accountId limit 1];
System.assertEquals(acc.phone,c.phone); }}
```

Update events in trigger

There are two update events.

Before update

After update

Trigger.old and trigger.new can be used in update event.

For eg:

Suppose we have an Customer object with three records:

Cid	Name	Age	Phone
111	aaa	23	3455
222	bbb	34	2344
333	ccc	27	9876
444	ddd	56	2346

And when we update the records into customer object from

333	Ccc	27	9876
444	Ddd	56	2345

To:

333	Ccc	29	7654
444	Ddd	35	1234

Then trigger.new will store

333	Ccc	29	7654
444	Ddd	35	1234

Whereas trigger.old will have previous value as:

333	Ccc	27	9876
444	Ddd	56	2345

Before update event

When we modify the value of a record and click on update, the before trigger is called on object and its code block is executed.

Records are updated with new values in database.

To perform any changes in the values of new record, we can do it directly using trigger.new in before trigger.

We can't use DML operation to perform any change in the record that are in Trigger.new as they are not committed in before update event.

After Update event

The operation written in after update triggers fires when the changes done on record are saved to database.

In after update trigger operation we can only read the data from trigger.

To perform any changes on the records in after update trigger we have to write DML statements.

Example: - write a trigger that will prevent a user from creating the lead that already exists as a contact. We will use lead/contact email address to prevent duplicates.

Lead has an email address.

Try to find matching contact based on email address(Using SOQL!).

If match found give the user an error.

If match not found do nothing.

triggerfindDuplicates on Lead (before insert) {

```
List<Contact>c1=[select email from contact];

//List<lead> lx=new list<lead>();

for (lead l1:trigger.new)

{

for(contact c:c1)

{

if((l1.email).equals(c.Email))

{

l1.addError('matching fields in leads and contacts');

}


}


}}
```

Delete events in trigger

There are two types of delete events

Before delete

After delete

Trigger.Old can be used in before or after delete to store the list of records which we are trying to delete.

On these records we can only perform read only operation.

For eg:

Suppose we have a Customer object with three records:

Cid	Name	Age	Phone

111	aaa	23	3455
222	bbb	34	2344
333	ccc	27	9876
444	ddd	56	2346

And when we deletethe records from customer object as:

333	ccc	27	9876
444	ddd	56	2345

Then the trigger.Old will contain the following record.

333	ccc	27	9876
444	ddd	56	2345

Example : -

When we are trying to delete customer record delete all Corresponding child record from test object.

```
Trigger customerdel on Customer__c (After Delete)

{

   List<Test__c> t = [Select id from test__c where cDetails in : Trigger.old];

   Delete test;

}
```

Event after undelete

When we undelete the record from recycle bin, then the operation written in after undelete trigger is executed.

Trigger.new holds the record which we have undeleted.

Trigger Context Variables

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

oldMap	A map of IDs to the old versions of the sObject records. Note that this map is only available in update and delete triggers.
Size	The total number of records in a trigger invocation, both old and new.

example:

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

        if (Trigger.isInsert)

      {

system.debug('5');

        if (Trigger.isBefore) {

        System.debug('1');

              } else if (Trigger.isAfter)

        { System.debug('2');

              }

           }

        else if (Trigger.isDelete)

        { System.debug('3');

           }

}
```

Calling a Class Method from a Trigger

You can call public utility methods from a trigger. Calling methods of other classes enables code reuse, reduces the size of your triggers, and improves maintenance of your Apex code.

Example:

```
trigger ApexClassCallingTrigger on Contact (before insert,after delete) {
if (Trigger.isInsert) {
            Integer recordCount = Trigger.New.size();
system.debug(recordCount);
EmailManager.sendMail('atul.rai@almamate.in', 'Test mail',
recordCount + ' contact(s) were inserted.');
   }
else if (Trigger.isDelete) {
            // Process after delete
          }
}
```

Apex class EmailManager:

```
public class EmailManager {
public static void sendMail(String address, String subject, String body) {
     Messaging.SingleEmailMessage mail = new Messaging.SingleEmailMessage();
String[] toAddresses = new String[] {address};
mail.setToAddresses(toAddresses);
```

```
mail.setSubject(subject);
mail.setPlainTextBody(body);
Messaging.SendEmailResult[] results = Messaging.sendEmail(
new Messaging.SingleEmailMessage[] { mail });
inspectResults(results);
   }
private static Boolean inspectResults(Messaging.SendEmailResult[] results) {
      Boolean sendResult = true;
for (Messaging.SendEmailResult res : results) {
if (res.isSuccess()) {
System.debug('Email sent successfully');
         }
else {
sendResult = false;
System.debug('The following errors occurred: ' + res.getErrors());
         }
      }
return sendResult;
   }
}
```

To invoke this trigger:

```
Contact c = new Contact(LastName='Test Contact');

insert c;
```

Adding Related Records

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

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

Example

```
trigger AddRelatedRecord on Account(after insert, after update) {

List<Opportunity> oppList = new List<Opportunity>();

Map<Id,Account> acctsWithOpps = new Map<Id,Account>(

   [SELECT Id,(SELECT Id FROM Opportunities) FROM Account WHERE Id
IN :Trigger.New]);

for(Account a : Trigger.New)

 {

if (acctsWithOpps.get(a.Id).Opportunities.size() == 0)

  {

oppList.add(new Opportunity(Name=a.Name + '
Opportunity',StageName='Prospecting',

                    CloseDate=System.today().addMonths(1),AccountId=a.Id));
```

```
 }

}

        if (oppList.size() > 0) {

        insert oppList;

}}
```

To test the trigger, create an account in the Salesforce user interface and name it Apples & Oranges.

Using Trigger Exceptions

To prevent saving records in a trigger, call the addError() method on the sObject in question. The addError() method throws a fatal error inside a trigger. The error message is displayed in the user interface and is logged.

The following trigger prevents the deletion of an account if it has related opportunities.

Example:

```
trigger AccountDeletion on Account (before delete) {


for (Account a : [SELECT Id FROM Account

                WHERE Id IN (SELECT AccountId FROM Opportunity) AND

               Id IN :Trigger.old])

{

Trigger.oldMap.get(a.Id).addError('Cannot delete account with
related opportunities.');

 }
```

```
}
```

Bulk Trigger Design Patterns

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

The following sections demonstrate the main ways of bulkifying your Apex code in triggers: operating on all records in the trigger, and performing SOQL and DML on collections of sObjects instead of single sObjects at a time.

Operating on Record Sets

Bulkified triggers operate on all sObjects in the trigger context.

Typically, triggers operate on one record if the action that fired the trigger originates from the user interface. But if the origin of the action was bulk DML or the API, the trigger operates on a record set rather than one record.

Example Non Bulk----

```
trigger MyTriggerNonBulk on Account(before insert) {
        Account a = Trigger.New[0];
        a.Description = 'New description';
    }
```

Example Bulk------

```
trigger MyTriggerBulk on Account(before insert) {
  for(Account a : Trigger.New) {
    a.Description = 'New description';
  }
```

```
}
```

Performing Bulk SOQL

By using SOQL features, you can write less code and make fewer queries to the database. Making fewer database queries helps you avoid hitting query limits, which are 100 SOQL queries for synchronous Apex or 200 for asynchronousApex.

The example makes a SOQL query inside a for loop to get the related opportunities for each account, which runs once for each Account sObject in Trigger.New. If you have a large list of accounts, a SOQL query inside a for loop could result in too many SOQL queries.

Example SOQL Non bulk:----

```
trigger SoqlTriggerNotBulk on Account(after update) {
    for(Account a : Trigger.New) {
        // Get child records for each account
        // Inefficient SOQL query as it runs once for each account!
        Opportunity[] opps = [SELECT Id,Name,CloseDate
                              FROM Opportunity WHERE AccountId=:a.Id];
        // Do some other processing
    }
}
```

Example Soql Trigger Bulk :

The SOQL query uses an inner query—(SELECT Id FROM Opportunities)—to get related opportunities of accounts.

The SOQL query is connected to the trigger context records by using the IN clause and binding the Trigger.Newvariable in the WHERE clause—WHERE Id IN :Trigger.New. This WHERE condition filters the accounts to only those records that fired this trigger.

Because the related records are already obtained, no further queries are needed within the loop to get those records.

```
trigger SoqlTriggerBulk on Account(after update) {

    // Perform SOQL query once.

    // Get the accounts and their related

    opportunities. List<Account> acctsWithOpps =

        [SELECT Id,(SELECT Id,Name,CloseDate FROM Opportunities)

         FROM Account WHERE Id IN :Trigger.New];

    // Iterate over the returned accounts

    for(Account a : acctsWithOpps) {

        Opportunity[] relatedOpps = a.Opportunities;

        // Do some other processing

    }

}
```

Alternatively, if you don't need the account parent records, you can retrieve only the opportunities that are related to the accounts within this trigger context. This list is specified in the WHERE clause by matching the AccountId field of the opportunity to the ID of accounts in Trigger.New: WHERE AccountId IN :Trigger.New. The returned opportunities are for all accounts in this trigger

context and not for a specific account. This next example shows the query used to get all related opportunities.

```
trigger SoqlTriggerBulk on Account(after update) {

    // Perform SOQL query once.

    // Get the related opportunities for the accounts in this trigger.

    List<Opportunity> relatedOpps =
[SELECT Id,Name,CloseDate FROM Opportunity

        WHERE AccountId IN :Trigger.New];

    // Iterate over the related opportunities

    for(Opportunity opp : relatedOpps) {

        // Do some other processing

    }

}
```

You can reduce the previous example in size by combining the SOQL query with the for loop in one statement: the SOQL for loop. Here is another version of this bulk trigger using a SOQL for loop.

```
trigger SoqlTriggerBulk on Account(after update) {

    // Perform SOQL query once.

    // Get the related opportunities for the accounts in this trigger,

    // and iterate over those records.

    for(Opportunity opp :
```

```
[SELECT Id,Name,CloseDate FROM Opportunity
    WHERE AccountId IN :Trigger.New]) {
    // Do some other processing
  }
}
```

Performing Bulk DML

When performing DML calls in a trigger or in a class, perform DML calls on a collection of sObjects when possible. Performing DML on each sObject individually uses resources inefficiently. The Apex runtime allows up to 150 DML calls in one transaction.

If a bulk account update operation fired the trigger, there can be many accounts. If each account has one or two opportunities, we can easily end up with over 150 opportunities.

Example Dml Trigger Non Bulk :---

```
trigger DmlTriggerNotBulk on Account(after update) {
  // Get the related opportunities for the accounts in this trigger.
  List<Opportunity> relatedOpps =
[SELECT Id,Name,Probability FROM Opportunity
    WHERE AccountId IN :Trigger.New];
  // Iterate over the related opportunities
  for(Opportunity opp : relatedOpps) {
    // Update the description when probability is greater
```

```
        // than 50% but less than 100%
        if ((opp.Probability >= 50) && (opp.Probability < 100)) {
            opp.Description = 'New description for opportunity.';
            // Update once for each opportunity -- not efficient!
            update opp;
        }
    }
}
```

This next example shows how to perform DML in bulk. The example adds the Opportunity sObject to update to a list of opportunities (oppsToUpdate) in the loop. Next, the trigger performs the DML call outside the loop on this list after all opportunities have been added to the list. This pattern uses only one DML call regardless of the number of sObjects being updated.

```
trigger DmlTriggerBulk on Account(after update) {
    // Get the related opportunities for the accounts in this trigger.
    List<Opportunity> relatedOpps =
[SELECT Id,Name,Probability FROM Opportunity
        WHERE AccountId IN :Trigger.New];
    List<Opportunity> oppsToUpdate = new List<Opportunity>();
// Iterate over the related opportunities
    for(Opportunity opp : relatedOpps) {
```

```
        // Update the description when probability is greater
        // than 50% but less than 100%
        if ((opp.Probability >= 50) && (opp.Probability < 100)) {
            opp.Description = 'New description for opportunity.';
            oppsToUpdate.add(opp);
        }
    }
    // Perform DML on a collection
    update oppsToUpdate
;
}
```

Bulk Design Pattern in Action: Trigger Example for Getting Related Records

Let's apply the design patterns you've learned by writing a trigger that accesses accounts' related opportunities. Modify the trigger example from the previous unit for the AddRelatedRecord trigger. The AddRelatedRecord trigger operates in bulk, but is not as efficient as it could be because it iterates over all Trigger.New sObject records. This next example modifies the SOQL query to get only the records of interest and then iterate over those records.

```
trigger AddRelatedRecord on Account(after insert, after update) {
    List<Opportunity> oppList = new List<Opportunity>();
    // Add an opportunity for each account if it doesn't already have one.
```

```
    // Iterate over accounts that are in this trigger but that don't
  have opportunities.

    for (Account a : [SELECT Id,Name FROM Account

                WHERE Id IN :Trigger.New AND

                Id NOT IN (SELECT AccountId FROM Opportunity)]) {

       // Add a default opportunity for this account oppList.add(new

       Opportunity(Name=a.Name + ' Opportunity',

                        StageName='Prospecting',

                        CloseDate=System.today().addMonths(1),

                        AccountId=a.Id));

    }

    if (oppList.size() > 0) {

       insert oppList;

    }

  }
```

Triggers and Order of Execution

When you save a record with an insert, update, or upsert statement, Salesforce performs the following events in order.

Loads the original record from the database or initializes the record for an upsert statement.

Loads the new record field values from the request and overwrites the old values.

If the request came from a standard UI edit page, Salesforce runs system validation to check the record for:

Compliance with layout-specific rules

Required values at the layout level and field-definition level

Valid field formats

Maximum field length

When the request comes from other sources, such as an Apex application or a SOAP API call, Salesforce validates only the foreign keys.

Executes all before triggers.

Runs most system validation steps again. The only system validation that Salesforce doesn't run a second time (when the request comes from a standard UI edit page) is the enforcement of layout-specific rules.

Executes duplicate rules. If the duplicate rule identifies the record as a duplicate and uses the block action, the record is not saved and no further steps, such as after triggers and workflow rules, are taken.

Saves the record to the database, but doesn't commit yet.

Executes all after triggers.

Executes assignment rules.

Executes auto-response rules.

Executes workflow rules.

If there are workflow field updates, updates the record again.

If the record was updated with workflow field updates,

fires before update triggers and after update triggers one more time (and only one more time), in addition to standard validations. Custom validation rules, duplicate rules, and escalation rules are not run again.

Executes processes.

If there are workflow flow triggers, executes the flows.

Executes escalation rules.

Executes entitlement rules.

If the record contains a roll-up summary field or is part of a cross-object workflow, performs calculations and updates the roll-up summary field in the parent record. Parent record goes through save procedure.

If the parent record is updated, and a grandparent record contains a roll-up summary field or is part of a cross-object workflow, performs calculations and updates the roll-up summary field in the grandparent record. Grandparent record goes through save procedure.

Executes Criteria Based Sharing evaluation.

Commits all DML operations to the database.

Executes post-commit logic, such as sending email.

During a recursive save, Salesforce skips steps 8 (assignment rules) through 17 (roll-up summary field in the grandparent record).

VisualForce with Controller

VisualForce Recap

Visualforce is a web development framework that enables developers to build sophisticated, custom user interfaces for mobile and desktop apps that can be hosted on the Force.com platform.

Visualforce enables developers to extend Salesforce's built-in features, replace them with new functionality, and build completely new apps.

Visualforce can integrate with any standard web technology or JavaScript framework to allow for a more animated and rich user interface.

Where We Can Use Visualforce

Salesforce provides a range of ways that you can use Visualforce within your organization. You can extend Salesforce's built-in features, replace them with new functionality, and build completely new apps.

The following are some of the ways you can add Visualforce to your organization.

- Display a Visualforce Page from a Tab
- Display a Visualforce Page in Salesforce1
- Display a Visualforce Page within a Standard Page Layout

- Display a Visualforce Page by Overriding Standard Buttons or Links
- Display a Visualforce Page Using Custom Buttons or Links
- Link Directly to a Visualforce Page

You can view, create, and edit Visualforce pages several different ways in Salesforce.

```
<apex:page sidebar="false">

  <h1>Hello World</h1>


  <apex:pageBlock title="A Block Title">
    <apex:pageBlockSection title="A Section Title">

      I'm three components deep!
    </apex:pageBlockSection>

  </apex:pageBlock>

</apex:page>
```

Visualforce pages can display data retrieved from the database or web services, data that changes depending on who is logged on and viewing the page, and so on. This dynamic data is accessed in markup through the use of global variables, calculations, and properties made available by the page's controller. Together these are described generally as *Visualforce expressions*. Use expressions for dynamic output or passing values into components by assigning them to attributes.

The expression syntax in Visualforce is: {! *expression* }

Global Variables

Use global variables to access and display system values and resources in your Visualforce markup.

```
<apex:page sidebar="false">
  <apex:pageBlock title="User Status">
    <apex:pageBlockSection columns="1">

    </apex:pageBlockSection>
  </apex:pageBlock>

</apex:page>
```

1. This markup creates a box with platform styling, ready for you to add some useful information.
2. Add the following markup between the <apex:pageBlockSection> tags.

```
{! $User.FirstName }
```

3. Add two more expressions that use the $User global variable to the markup for the User Status panel so that the page looks like the following.

```
<apex:page sidebar="false">
  <apex:pageBlock title="User Status">
    <apex:pageBlockSection columns="1">
      {! $User.FirstName } {! $User.LastName }
      ({! $User.Username })
    </apex:pageBlockSection>
  </apex:pageBlock>
</apex:page>
```

Visualforce expressions are case-insensitive, and spaces within the {! ... } are ignored. So these expressions all produce the same value:

- {! $User.FirstName}
- {!$USER.FIRSTNAME}
- {! $user.firstname }

Visualforce lets you use more than just global variables in the expression language. It also supports formulas that let you manipulate values.

```
{! $User.FirstName & ' ' & $User.LastName }
<p> Today's Date is {! TODAY() } </p>
<p> Next week it will be {! TODAY() + 7 } </p>
```

Example: Add the following to the page markup, below the date expressions.

```
<p>The year today is {! YEAR(TODAY()) }</p>
<p>Tomorrow will be day number {! DAY(TODAY() + 1) }</p>
```

```
<p>Let's find a maximum: {! MAX(1,2,3,4,5,6,5,4,3,2,1) } </p>
<p>The square root of 49 is {! SQRT(49) }</p>
<p>Is it true? {! CONTAINS('salesforce.com', 'force.com') }</p>
```

Standard Controller

Visualforce uses the traditional model–view–controller (MVC) paradigm, and includes sophisticated built-in controllers to handle standard actions and data access, providing simple and tight integration with the Force.com database. These built-in controllers are referred to generally as *standard controllers*, or even ***the*** standard controller.

Most standard and all custom objects have standard controllers that can be used to interact with the data associated with the object, so you don't need to write the code for the controller yourself. You can extend the standard controllers to add new functionality, or create custom controllers from scratch.

If you want to use the standard controller to reference a specific record, it needs to know the record identifier, or ID, of the record to work with. It uses the ID to retrieve the data, and to save it back to the database when the record's data is changed.

```
<apex:page sidebar="false" standardController="Account">
  <apex:pageBlock title="Account
    Summary"> <apex:pageBlockSection>
```

```
            Name: {! Account.Name } <br/>
            Phone: {! Account.Phone } <br/>
            Industry: {! Account.Industry } <br/>
            Revenue: {! Account.AnnualRevenue } <br/>
        </apex:pageBlockSection>
    </apex:pageBlock>
</apex:page>
```

Display Fields from Related Records

Use dot notation to display data from related records.

```
Account owner: {! Account.Owner.Name } <br/>
```

Output Components

Visualforce includes nearly 150 built-in components that you can use on your pages. Components are rendered into HTML, CSS, and JavaScript when a page is requested.

Create a Visualforce Page with a Standard Controller

Use output components with a standard controller to make it easy to access and display record details.

<apex:detail> is a coarse-grained output component that adds many fields, sections, buttons, and other user interface elements to the page in just one line of markup.

Display Related Lists

Use <apex:relatedList> to display lists of records related to the current record.

```
<apex:relatedList list="Contacts"/>
<apex:relatedList list="Opportunities" pageSize="5"/>
```

Display Individual Fields

Use <apex:outputField> to display individual fields from a record.

```
<apex:outputField value="{!
Account.Name }"/>
<apex:outputField value="{!
Account.Phone }"/>
<apex:outputField value="{! Account.Industry }"/>
<apex:outputField value="{! Account.AnnualRevenue }"/>
```

The field values are displayed all on one line, without labels, and without other formatting but when you wrap it

in <apex:pageBlock> and <apex:pageBlockSection> components, its behavior changes quite a bit.

```
<apex:pageBlock title="Account Details">
  <apex:pageBlockSection>
    <apex:outputField value="{! Account.Name }"/>
    <apex:outputField value="{! Account.Phone }"/>
    <apex:outputField value="{! Account.Industry }"/>
    <apex:outputField value="{! Account.AnnualRevenue }"/>
  </apex:pageBlockSection>
</apex:pageBlock>
```

Display A Table

Use <apex:pageBlockTable> to add a table of data to a page.

What exactly is a related list? What does <apex:relatedList> do when you add it to a page?

- It grabs a list of similar data elements. For example, a list of contacts for the account.
- It sets up a table with columns for each field, headers atop each column, and so on.
- For each item in the list—for each related contact—it adds a row to the table, and fills in each column with the appropriate field from that record.

You can do the same thing in your own Visualforce markup using *iteration components*. Replace the two <apex:relatedList/> lines with the following markup.

```
<apex:pageBlock title="Contacts">
  <apex:pageBlockTable value="{!Account.contacts}" var="contact">
    <apex:column
value="{!contact.Name}"/>
    <apex:column value="{!contact.Title}"/>
    <apex:column value="{!contact.Phone}"/>
  </apex:pageBlockTable>
</apex:pageBlock>
```

Introduction to Visualforce Forms

Creating and editing data is a fundamental aspect of any app. Visualforce provides everything you need to easily create pages that can create new records, or retrieve a record, edit its values, and save the changes back to the database.

Create a Basic Form

Use <apex:form> and <apex:inputField> to create a page to edit data. Combine <apex:commandButton> with the saveaction built into the standard controller to create a new record, or save changes to an existing one.

```
<apex:page standardController="Account">
```

```
    <h1>Edit Account</h1>

    <apex:form>

        <apex:inputField value="{! Account.Name }"/>

        <apex:commandButton action="{! save }" value="Save" />

    </apex:form>

</apex:page>
```

Add Field Labels and Platform Styling

Place form elements

within <apex:pageBlock> and <apex:pageBlockSection> tags to organize and group them, and to have the form adopt the platform visual style.

```
<apex:pageBlockSection columns="1"> <apex:inputField
    value="{! Account.Name }"/> <apex:inputField value="{!
    Account.Phone }"/> <apex:inputField value="{!
    Account.Industry }"/> <apex:inputField value="{!
    Account.AnnualRevenue }"/>
</apex:pageBlockSection>
```

Display Form Errors and Messages

Use <apex:pageMessages> to display any form handling errors or messages.

Under the <apex:pageBlock> tag, add the following line.

```
<apex:pageMessages/>
```

Edit Related Records

Make it easy for users to edit related information by providing links to related records. Below the existing closing </apex:pageBlock> tag, add the following markup.

```
<apex:pageBlock title="Contacts">
  <apex:pageBlockTable value="{!Account.contacts}"
    var="contact"> <apex:column>
      <apex:outputLink
        value="{! URLFOR($Action.Contact.Edit, contact.Id) }">
        Edit
      </apex:outputLink>
       
      <apex:outputLink
        value="{! URLFOR($Action.Contact.Delete, contact.Id) }">
        Del
      </apex:outputLink>
    </apex:column>
    <apex:column value="{!contact.Name}"/>
    <apex:column value="{!contact.Title}"/>
```

```
        <apex:column value="{!contact.Phone}"/>
    </apex:pageBlockTable>
</apex:pageBlock>
```

Introduction to the Standard List Controller

The standard list controller allows you to create Visualforce pages that can display or act on a set of records.

The standard list controller provides many powerful, automatic behaviors such as querying for records of a specific object and making the records available in a collection variable, as well as filtering of and pagination through the results. Adding the standard list controller to a page is very similar to adding the standard (record) controller, but with the intent of working with many records at once, instead of one record at a time.

Display a List of Records

Use the standard list controller and an iteration component, such as <apex:pageBlockTable>, to display a list of records.

The standard (record) controller makes it easy to get a single record loaded into a variable you can use on a Visualforcepage. The standard *list* controller is similar, except instead of a single record, it loads a list, or *collection*, of records into the variable.

```
<apex:page standardController="Contact" recordSetVar="contacts">
  <apex:pageBlock title="Contacts List">

    <!-- Contacts List -->
    <apex:pageBlockTable value="{! contacts }" var="ct">
      <apex:column value="{! ct.FirstName }"/>
      <apex:column value="{! ct.LastName }"/>
      <apex:column value="{! ct.Email }"/>
      <apex:column value="{! ct.Account.Name }"/>
    </apex:pageBlockTable>

  </apex:pageBlock>
</apex:page>
```

Using a standard list controller is very similar to using a standard controller. First you set the standardController attribute on the <apex:page> component, then you set the recordSetVar attribute on the same component. The standardController attribute sets the object to work with, just like with the standard controller. The recordSetVar sets the name of the variable to be created with the collection of records, here, {! contacts }. By convention, this variable is usually named the plural of the object name.

Add List View Filtering to the List

Use {! listViewOptions } to get a list of list view filters available for an object. Use {! filterId } to set the list view filter to use for a standard list controller's results.

```
<apex:page standardController="Contact" recordSetVar="contacts">
  <apex:form>
    <apex:pageBlock title="Contacts List" id="contacts_list">
      Filter:
      <apex:selectList value="{! filterId }" size="1"> <apex:selectOptions
        value="{! listViewOptions }"/> <apex:actionSupport
        event="onchange" reRender="contacts_list"/>
      </apex:selectList>
      <apex:pageBlockTable value="{! contacts }" var="ct">
        <apex:column value="{! ct.FirstName }"/>
        <apex:column value="{! ct.LastName }"/>
        <apex:column value="{! ct.Email }"/>
        <apex:column value="{! ct.Account.Name }"/>
      </apex:pageBlockTable>
    </apex:pageBlock>
  </apex:form>
</apex:page>
```

Introduction to Custom Controllers

A custom controller is an Apex class that uses the default, no-argument constructor for the outer, top-level class. You cannot create a custom controller constructor that includes parameters.Custom controllers contain custom logic and data manipulation that can be used by a Visualforce page.

For example, a custom controller can retrieve a list of items to be displayed, make a callout to an external web service, validate and insert data, and more—and all of these operations will be available to the Visualforce page that uses it as a controller.

A Visualforce Page that Uses a Custom Controller

```
<apex:page controller="myController"
tabStyle="Account"> <apex:form>
<apex:pageBlock title="Congratulations {!$User.FirstName}">
        You belong to Account Name: <apex:inputField
value="{!account.name}"/>
<apex:commandButton action="{!save}" value="save"/>
</apex:pageBlock>
</apex:form>
</apex:page>
```

A custom controller that is an Apex class.

```
public class MyController {
private final Account account;
public MyController() {
account = [SELECT Id, Name, Site FROM Account
            WHERE Id = :ApexPages.currentPage().getParameters().get('id')];
  }
public Account getAccount() {
return account;
  }
public PageReference save() {
update account;
return null;
  }
```

```
}
```

{! *someExpression* } in Visualforce markup automatically connects to a method named *getSomeExpression*() in your controller.

Defining Getter Methods

One of the primary tasks for a Visualforce controller class is to give developers a way of displaying database or computed values in page markup. Methods that enable this type of functionality are called *getter methods*, and are typically named get*Identifier*, where *Identifier* is the name for the records or primitive values returned by the method.

For example, the following controller has a getter method for returning the name of the controller as a string:

```
public class MyController {

public String getName() {
return 'MyController';
   }

}
```

To display the result of the getName method in page markup, use {!name}:

```
<apex:page controller="MyController">
   <apex:pageBlock title="Hello {!$User.FirstName}!">
      This is your new page for the {!name} controller.
   </apex:pageBlock>
</apex:page>
```

Defining Action Methods

Action methods perform logic or navigation when a page event occurs, such as when a user clicks a button, or hovers over an area of the page.

Action methods can be called from page markup by using {! } notation in the action parameter of one of the following tags:

- <apex:commandButton> creates a button that calls an action
- <apex:commandLink> creates a link that calls an action
- <apex:actionPoller> periodically calls an action
- <apex:actionSupport> makes an event (such as “onclick”, “onmouseover”, and so on) on another, named component, call an action
- <apex:actionFunction> defines a new JavaScript function that calls an action
- <apex:page> calls an action when the page is loaded

In the below example a command button is bound to the save method in the Account standard controller.

```
<apex:page controller="MyController" tabStyle="Account">
<apex:form>
<apex:pageBlock title="Hello {!$User.FirstName}!">
        You are viewing the {!account.name} account.
        <p/> Change Account Name: <p/>
<apex:inputField value="{!account.name}"/><p/>
<apex:commandButton action="{!save}" value="Save New Account
Name"/> </apex:pageBlock>
```

```
</apex:form>
</apex:page>
```

Defining Navigation Methods

Custom controller action methods can navigate users to a different page by returning a PageReference object.

A PageReference is a reference to an instantiation of a page. Among other attributes, PageReferences consist of a URL and a set of query parameter names and values.

In a custom controller or controller extension, you can refer to or instantiate a PageReference in one of the following ways:

. Page.existingPageName

- Refers to a PageReference for a Visualforce page that has already been saved in your organization.

```
PageReference pageRef
= new PageReference('***partialURL***');
```

- Creates a PageReference to any page that is hosted on the Force.com platform. For example, setting *'partialURL'* to'/apex/HelloWorld' refers to the Visualforce page located at http://*mySalesforceInstance*/apex/HelloWorld. Likewise, setting '*partialURL*' to '/' + '*recordID*' refers to the detail page for the specified record.

PageReference pageRef
= new PageReference('*fullURL***');**

- Creates a PageReference for an external URL. For example:

  PageReference pageRef = new PageReference('http://www.google.com');

More Examples:

Example 1:
Visualforce Page 1:

```
<apex:page controller="NewAndExistingController"
tabstyle="Account"> <apex:form >
<apex:pageBlock mode="edit" >
<apex:pageMessages />
<apex:pageBlockSection >
<apex:inputField value="{!Account.name}"/>
<apex:inputField value="{!Account.phone}"/>
<apex:inputField value="{!Account.industry}"/>
</apex:pageBlockSection>
<apex:pageBlockButtons location="bottom" >
<apex:commandButton value="Save" action="{!save}" />
</apex:pageBlockButtons>
</apex:pageBlock>
</apex:form>
</apex:page>
```

custom controller 1:

```
public class NewAndExistingController
{
public Account account { get; private set;
} public NewAndExistingController()
  {
     Id id = ApexPages.currentPage().getParameters().get('id');
account =(id==null)? new Account():[SELECT Name, Phone, Industry
FROM Account WHERE Id = :id];
  }
public PageReference save()
 {
try {
upsert(account);
   }
catch(System.DMLException e) {
ApexPages.addMessages(e);
return null;
   }
  //PageReference redirectSuccess = new
ApexPages.StandardController(Account).view();
  PageReference redirectSuccess = Page.success;
    //PageReference redirectSuccess=ApexPages.currentPage();
   // PageReference redirectSuccess=Page.success;
   // PageReference redirectSuccess=new
PageReference('/0019000001aRxEy'); return (redirectSuccess);
 }
}
```

Example 2:
Visualforce Page 2:

```
<apex:page controller="Example">
<apex:form >
<apex:outputLabel id="one">enter u'r name
</apex:outputLabel> <apex:inputText value="{!name}" />
```

```
<apex:commandButton value="click me" reRender="two" action="{!show}"
/> <apex:outputLabel id="two">{!name}</apex:outputLabel> </apex:form>

</apex:page>
```

custom controller 2:

```
public class Example
{
String name;
public String getName()
{
return name;
}
public void setName(String na)
   {
name=na;
   }
public PageReference show()
   {
name=name+' is my name:';
return null;
   }
}
```

Example 3:
Visualforce Page 3:

```
<apex:page controller="WendCAccController"
tabStyle="Account"> <apex:form >
<apex:pageBlock title="Congratulations {!$User.FirstName}">
        You belong to Account Name: <apex:inputField
value="{!account.name}"/>
<apex:commandButton action="{!save}" value="save"/>
</apex:pageBlock>
</apex:form>
```

```
</apex:page>
```

custom controller 3:

```
public class WendCAccController {
private final Account account;
public WendCAccController() {
account = [SELECT Id, Name, Site FROM Account
            WHERE Id = :ApexPages.currentPage().getParameters().get('id')];
   }
public Account getAccount() {
return account;
   }
public PageReference save() {
update account;
return null;
   }
}
```

Example 4:
Visualforce Page 4:

```
<apex:page controller="addMultipleAcc">
<apex:form >
<apex:pageBlock >
<apex:pageBlockTable value="{!listAccount}"
var="acc"> <apex:column headerValue="Account
Name"> <apex:inputField value="{!acc.Name}" />
</apex:column>
<apex:column headerValue="Account Number">
<apex:inputField value="{!acc.accountNumber}" />
</apex:column>
<apex:column headerValue="Account Type
"> <apex:inputField value="{!acc.type}" />
</apex:column>
<apex:column headerValue="Industry ">
<apex:inputField value="{!acc.industry}" />
```

```
</apex:column>
</apex:pageBlockTable>
<apex:pageBlockButtons >
<apex:commandButton value="add one more account"
action="{!addAccount}" />
<apex:commandButton value="save accounts" action="{!saveAccount}" />
</apex:pageBlockButtons>
</apex:pageBlock>
</apex:form>
</apex:page>
```

custom controller 4:

```
 public class addMultipleAcc {
   Account ac=new Account();
public List<account> listAccount{
get;set;} public addMultipleAcc()
   {
listAccount=new List<Account>();
listAccount.add(ac);
   }
public void addAccount()
   {
      Account ac1=new Account();
listAccount.add(ac1);
   }
public pageReference saveAccount()
   {
insert listAccount;
return Page.success;
   }
   }
```

Building a Controller Extension

A controller extension is any Apex class containing a constructor that takes a single argument of type ApexPages.StandardController or *CustomControllerName*, where *CustomControllerName* is the name of a custom controller you want to extend.

The following class is a simple example of a controller extension:

```
public class myControllerExtension {
private final Account acct;
      // The extension constructor initializes the private member
   // variable acct by using the getRecord method from the standard controller.
public myControllerExtension(ApexPages.StandardController stdController) {
      this.acct = (Account)stdController.getRecord();
   }
public String getGreeting() {
return 'Hello ' + acct.name + ' (' + acct.id + ')';
   }
}
```

The following Visualforce markup shows how the controller extension from above can be used in a page:

```
<apex:page standardController="Account"
extensions="myControllerExtension">
{!greeting} <p/>
```

```
<apex:form>
<apex:inputField value="{!account.name}"/><p/>
<apex:commandButton value="Save" action="{!save}"/>
</apex:form>
</apex:page>
```

Multiple controller extensions can be defined for a single page through a comma-separated list. This allows for overrides of methods with the same name. For example, if the following page exists:

```
<apex:page standardController="Account"
extensions="ExtOne,ExtTwo"
showHeader="false"> <apex:outputText
value="{!foo}" /> </apex:page>
```

with the following extensions:

```
public class ExtOne {
public ExtOne(ApexPages.StandardController acon) { }
public String getFoo() {
return 'foo-One';
   }
}

public class ExtTwo {
public ExtTwo(ApexPages.StandardController acon) { }
public String getFoo() {
return 'foo-Two';
   }
}
```

The value of the <apex:outputText> component renders as foo-One. Overrides are defined by whichever methods are defined as the first member in the comma-separated list.

More Examples:
Example 1:

VisualforcePage 1:

```
<apex:page standardController="Account" extensions="WendCEAcc">

{!greeting} <p/>

<apex:form >
<apex:inputField value="{!account.name}"/><p/>
<apex:commandButton value="Save" action="{!save}"/>
</apex:form>
</apex:page>
```

Controller extension:

```
public class WendCEAcc {
private final Account acct;

public WendCEAcc(ApexPages.StandardController stdController) {
      this.acct = (Account)stdController.getRecord();
   }
public String getGreeting() {
return 'Hello ' + acct.name + ' (' + acct.id + ')';
   }
}
```

Apex Unit Tests

The Apex testing framework enables you to write and execute tests for your Apex classes and triggers on the Force.com platform. Apex unit tests ensure high quality for your Apex code and let you meet requirements for deploying Apex.

Apex code can only be written in a sandbox environment or a Developer org, not in production. Apex code can be deployed to a production org from a sandbox. Before you can deploy your code or package it for the Force.com AppExchange, at least 75% of Apex code must be covered by tests, and all those tests must pass.

The following are the benefits of Apex unit tests.

- Ensuring that your Apex classes and triggers work as expected
- Meeting the code coverage requirements for deploying Apex to production or distributing Apex to customers via packages

What to Test in Apex

Salesforce recommends that you write tests for the following:

Single action

Test to verify that a single record produces the correct, expected result.

Bulk actions

Any Apex code, whether a trigger, a class or an extension, may be invoked for 1 to 200 records. You must test not only the single record case, but the bulk cases as well.

Positive behavior

Test to verify that the expected behavior occurs through every expected permutation, that is, that the user filled out everything correctly and did not go past the limits.

Negative behavior

There are likely limits to your applications, such as not being able to add a future date, not being able to specify a negative amount, and so on.

Restricted user

Test whether a user with restricted access to the sObjects used in your code sees the expected behavior. That is, whether they can run the code or receive error messages.

Test Method Syntax

Test methods take no arguments and have the following syntax:

```
@isTest static void testName() {
   // code_block
```

```
}
```

Alternatively, a test method can have this syntax:

```
static testMethod void testName() {
   // code_block
 }
```

Using the isTest annotation instead of

the testMethod keyword is more flexible as you can specify parameters in the annotation

The visibility of a test method doesn't matter, so declaring a test method as public or private doesn't make a difference as the testing framework is always able to access test methods. For this reason, the access modifiers are omitted in the syntax.

Test methods must be defined in test classes, which are classes annotated with isTest.

This sample class shows a definition of a test class with one test method.

```
@isTest
private class MyTestClass {
   @isTest static void myTest() {
      // code_block
```

```
    }
}
```

Test classes can be either private or public. If you're using a test class for unit testing only, declare it as private. Public test classes are typically used for test data factory classes.

Example:

Apex class:

```
public class TemperatureConverter {
    // Takes a Fahrenheit temperature and returns the Celsius
    equivalent. public static Decimal FahrenheitToCelsius(Decimal fh) {
        Decimal cs = (fh - 32) *
        5/9; return cs.setScale(2);
    }
}
```

TestCase:

```
@isTest
 private class TemperatureConverterTest {

    @isTest static void testWarmTemp() {
        Decimal celsius = TemperatureConverter.FahrenheitToCelsius(70);
        System.assertEquals(21.11,celsius);
```

```
    }

    @isTest static void testFreezingPoint() {
        Decimal celsius = TemperatureConverter.FahrenheitToCelsius(32);
        System.assertEquals(0,celsius);
    }

    @isTest static void testBoilingPoint() {
        Decimal celsius = TemperatureConverter.FahrenheitToCelsius(212);
        System.assertEquals(100,celsius,'Boiling point temperature is
not expected.');
    }

    @isTest static void testNegativeTemp() {
        Decimal celsius = TemperatureConverter.FahrenheitToCelsius(-
        10); System.assertEquals(-23.33,celsius);
    }
}
```

Each test method verifies one type of input: a warm temperature, the freezing point temperature, the boiling point temperature, and a negative temperature. The verifications are done by calling theSystem.assertEquals() method, which takes two parameters: the first is the expected value, and the second is the actual value.

There is another version of this method that takes a third parameter—a string that describes the comparison being

done, which is used in testBoilingPoint(). This optional string is logged if the assertion fails.

To run the methods in test class.

In the Developer Console, click Test | New Run.

Under Test Classes, select the test class.

To add all the test methods in the selected class to the test run, click Add Selected.
Click Run.

In the Tests tab, you see the status of your tests as they're running. Expand the test run, and expand again until you see the list of individual tests that were run. They all have green checkmarks.

After you run tests, code coverage is automatically generated for the Apex classes and triggers in the org.

Whenever you modify your Apex code, rerun your tests to refresh code coverage results.

To update your code coverage results, use **Test** | **Run All** rather than **Test** | **New Run**.

Unit Test Considerations

- Starting with Salesforce API 28.0, test methods can no longer reside in non-test classes and must be part of classes annotated with isTest.
- Test methods can't be used to test Web service callouts. Instead, use mock callouts
- You can't send email messages from a test method.
- Since test methods don't commit data created in the test, you don't have to delete test data upon completion.
- If a test class contains a static member variable, and the variable's value is changed in a testSetup or test method, the new value isn't preserved. Other test methods in this class get the original value of the static member variable.
- Field history tracking records (such as AccountHistory) can't be created in test methods because they require other sObject records to be committed first (for example, Account).

Example TvRemoteControl:

Apex class:

```
public class TVRemoteControl {
   Integer volume;
   static final Integer MAX_VOLUME=50;

   public TVRemoteControl(Integer v)
   {
      volume=v;
   }

   public Integer increaseVolume(Integer v)
   {
```

```
        volume+=v;
        if(volume>MAX_VOLUME)
        {
           volume=MAX_VOLUME;
        }
        return volume;
    }
public Integer decreaseVolume(Integer v)
    {
    volume -=v;
    if(volume<0)
      volume=0;

        return volume;
    }
    public static String getMenuOptions()
    {
        return 'Audio setting-Video-setting';
    }

}
```

TestCase:

```
@isTest
public class TVRemoteControlTest {

    @isTest static void testVolumeIncrease()
    {
        TvRemoteControl tv=new TVRemoteControl(10);
        Integer newVol=tv.increaseVolume(15);
        System.assertEquals(25,newVol);
    }


    @isTest static void testVolumeDecrease()
    {
```

```
    TvRemoteControl tv=new TVRemoteControl(20);
    Integer newVol=tv.decreaseVolume(15);
    System.assertEquals(5,newVol);
  }

   @isTest static void testVolumeDecreaseUnderMin()
  {
    TvRemoteControl tv=new TVRemoteControl(10);
    Integer newVol=tv.decreaseVolume(100);
    System.assertEquals(0,newVol);
  }
      @isTest static void testVolumeIncreaseOverMax()
  {
    TvRemoteControl tv=new TVRemoteControl(10);
    Integer newVol=tv.IncreaseVolume(100);
    System.assertEquals(50,newVol);
  }

  @isTest static void testGetMenuOptions()
  {
    String str=TVRemoteControl.getMenuOptions();
    system.assertNotEquals(null,str);
    system.assertNotEquals('',str);
  }
}
```

Testing Best Practices

Cover as many lines of code as possible. Before you can deploy Apex or package it for the Force.com AppExchange, the following must be true.

- If code uses conditional logic (including ternary operators), execute each branch.
- Make calls to methods using both valid and invalid inputs.

- Complete successfully without throwing any exceptions, unless those errors are expected and caught in a try…catch block.
- Always handle all exceptions that are caught, instead of merely catching the exceptions.
- Use System.assert methods to prove that code behaves properly.
- Use the runAs method to test your application in different user contexts.
- Exercise bulk trigger functionality—use at least 20 records in your tests.
- Use the ORDER BY keywords to ensure that the records are returned in the expected order.
- Not assume that record IDs are in sequential order.
- Set up test data:
- Create the necessary data in test classes, so the tests do not have to rely on data in a particular organization.
- Create all test data before calling the Test.startTest method.
- Since tests don't commit, you won't need to delete any data.
- Write comments stating not only what is supposed to be tested, but the assumptions the tester made about the data, the expected outcome, and so on.
- Test the classes in your application individually. Never test your entire application in a single test.

Test Apex Triggers

Before deploying a trigger, write unit tests to perform the actions that fire the trigger and verify expected results.

Let's test a trigger that we worked with earlier in the Writing Apex Triggers unit. If an account record has related opportunities, the AccountDeletion trigger prevents the record's deletion.

Trigger: AccountDeletion

```
trigger AccountDeletion on Account (before delete) {
    // Prevent the deletion of accounts if they have related
    contacts. for (Account a : [SELECT Id FROM Account
                WHERE Id IN (SELECT AccountId FROM Opportunity) AND
                Id IN :Trigger.old]) {
       Trigger.oldMap.get(a.Id).addError(
          'Cannot delete account with related opportunities.');
    }
}
```

TestCase:

```
@isTest
 private class TestAccountDeletion {
    @isTest static void TestDeleteAccountWithOneOpportunity() {
       // Test data setup
       // Create an account with an opportunity, and then try to delete it
```

```
    Account acct = new Account(Name='Test Account');
    insert acct;
    Opportunity opp = new Opportunity(Name=acct.Name + ' Opportunity',
                        StageName='Prospecting',
                        CloseDate=System.today().addMonths(1),
                        AccountId=acct.Id);
    insert opp;
    // Perform   test
    Test.startTest();
    Database.DeleteResult result = Database.delete(acct, false);
    Test.stopTest();
    // Verify
    // In this case the deletion should have been stopped by the trigger,
    // so verify that we got back an error. System.assert(!result.isSuccess());
    System.assert(result.getErrors().size() > 0); System.assertEquals('Cannot
    delete account with related opportunities.',

                    result.getErrors()[0].getMessage());
  }
}
```

Deploying Apex

You can deploy Apex using:

Change Sets

the Force.com IDE

the Force.com Migration Tool

SOAP API

Any deployment of Apex is limited to 5,000 code units of classes and triggers.

Using Change Sets To Deploy Apex

You can deploy Apex classes and triggers between connected organizations, for example, from a sandbox organization to your production organization. You can create an outbound change set in the Salesforce user interface and add the Apex components that you would like to upload and deploy to the target organization.

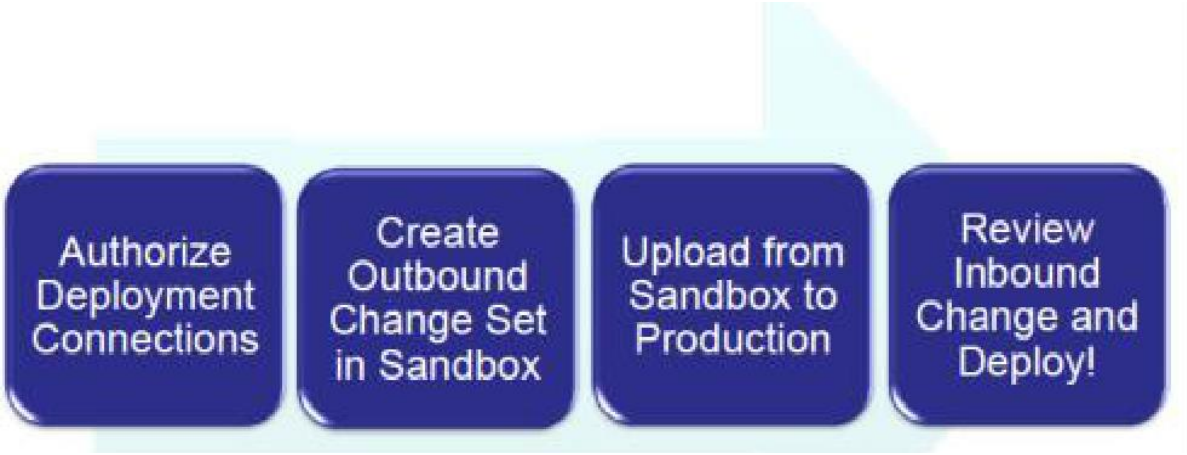
Change sets represent sets of customizations in your org (or metadata components) that you can deploy to a connected org. Use change sets as a point-and-click tool to migrate your customizations.

There's no need to download files to a local file system. Other deployment methods require you to work with local files.

The change set tool helps you discover and include dependent components. For example, a new custom field can't be migrated if the custom object it belongs to doesn't exist in the target org.

You define the set of components once. You can reuse the same set of components for another deployment by cloning the change set. Cloning change sets is helpful during the iterative phases of a project.

Deploying with change sets involves the following steps.



- The first step is needed only once per org.
- You can upload a change set from one sandbox to another sandbox, or from sandbox to production, and vice versa.
- You can validate the change set as part of your review in the final step.

Authorize a Deployment Connection

Before you can receive change sets from a sandbox or other organization, authorize a deployment connection in the organization that receives the changes.

Log into the organization that'll receive inbound change sets. Usually this is the production organization associated with your sandbox.

- From Setup, enter Deployment in the Quick Find box, then select **Deployment Settings**.
- Click **Edit** next to the organization from which you want to receive outbound change sets. Usually this is your sandbox.
- Select **Allow Inbound Changes** and click **Save**.

Create and Upload an Outbound Change Set

Typically, you create an outbound change set in a sandbox organization and deploy it to production. But depending on your development lifecycle, you might choose to migrate changes in either direction between related organizations.

- From Setup, enter Outbound Change Sets in the Quick Find box, then select **Outbound Change Sets**.
- Click **New**.
- Enter a name for your change set and click **Save**.
- In the Change Set Components section, click **Add**.

- Choose the type of component (for example, Custom Object or Custom Field), the components you want to add, and click **Add To Change Set**.
- Click **View/Add Dependencies** to see whether the components you've added to the change set are dependent on other customizations.
- Select the dependent components you want to add and click **Add To Change Set**.
- Click **Upload** and choose your target organization.

The outbound change set detail page displays a message and you get an email notification when the upload is complete.

Now log into the target organization, where you can see the inbound change set.

Validate an Inbound Change Set

Validating a change set allows you to see the success or failure messages without committing the changes.

- From Setup, enter Inbound Change Sets in the Quick Find box, then select **Inbound Change Sets**.
- Click the name of a change set.
- Click **Validate**.
- The validation process locks the resources that are being deployed. Changes you make to locked resources or items

related to those resources while the validation is in progress can result in errors.

- After the validation completes, click **View Results**.
- If you receive any error messages, resolve them before you deploy. The most common causes of errors are dependent components that aren't included in the change set and Apex test failures.

Deploy an Inbound Change Set

Deploying a change set commits the changes it contains to the target organization.

- From Setup, enter Inbound Change Sets in the Quick Find box, then select **Inbound Change Sets**.
- In the Change Sets Awaiting Deployment list, click the name of the change set you want to deploy.
- Click **Deploy**.

A change set is deployed in a single transaction. If the deployment is unable to complete for any reason, the entire transaction is rolled back. After a deployment completes successfully, all changes are committed to your org and the deployment can't be rolled back.