What is Apex?

- ➔ Apex is a strongly typed, object-oriented programming language which is a proprietary language developed by salesforce.com to allow us to write the code that executes on the force.com platform.
- ➔ It is used for building SAAS applications on top of salesforce.com CRM functionality.
- ➔ Apex is saved, compiled and executed on the servers of the force.com platform.
- ➔ It enables the developer to add business logic to most of the system events including button on-clicks, related record updates and VF & lightning pages.

Features of Apex:

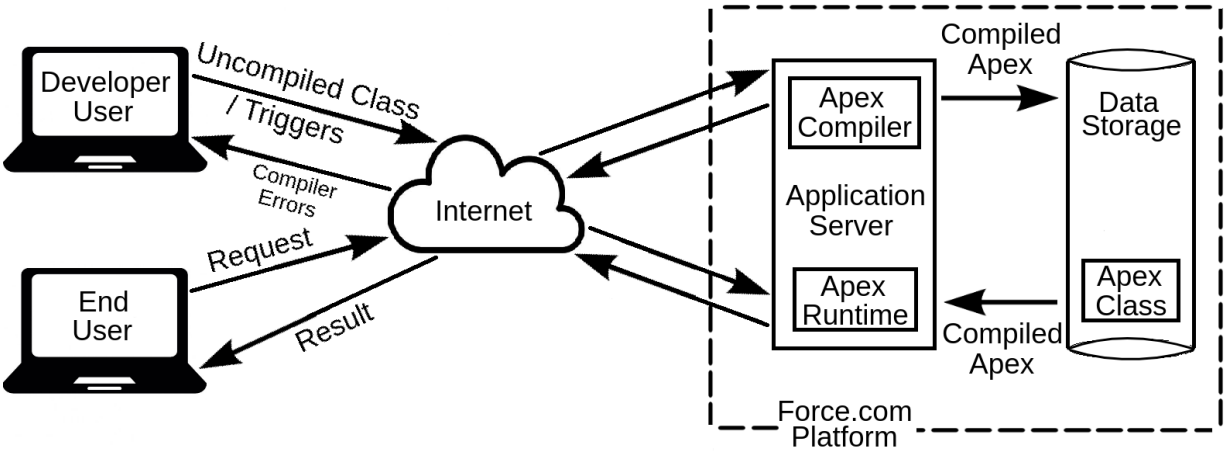
- ★ It upgrades automatically.
- ★ Integrated with the DB which means it can access and manipulate records without the need to establish the DB connection explicitly.
- ★ It has Java like syntax and it is easy to use.
- ★ It is easy to test as it provides built-in support for executing test cases.
- ★ Multi-Tenant Environment.
- ★ You can save your apex code against different versions of the force.com API.
- ★ Apex is a case-insensitive language.

When to use Apex?

1. To perform a couple of business processes which are not supported by workflows or processes or flows.
2. To perform complex validation over multiple objects.
3. To create web services and email services.
4. For transactions and rollbacks.

Flows of Action:

A. Developer Action: When the developer writes and saves the code to apex platform. The platform application server compiles the code into a set of instruction that can be understood by the apex-runtime, interprets and then saves those instructions as compiled apex.



B. End-User Action: When the end-user performs some action which involves the apex code or when the end-user triggers the execution of apex by clicking a button or accessing a VF page. The platform application server retrieves the compiled instructions from the meta-data and sends it through apex runtime which interprets before returning the result.

The end-user observes no differences in the execution time as compared to the standard application platform request.

Features that are not supported by Apex:

1. It cannot show the elements in UI other than error message.
2. It cannot change the standard SFDC functionality but can be used to stop its execution or to add a new functionality.
3. It cannot be used to create a temporary file.
4. It cannot create multiple threads.

Apex Environments:

There are several environments for developing apex code

- ➔ We can run apex in a developer org, production org and a sandbox.
- ➔ We cannot develop in our production org because live users are accessing the system so while we are developing it can destabilize our data and corrupt our application.
- ➔ Instead we do all our development work in sandbox or developer edition.

Different tools for writing Apex Code:

1. Force.com Developer Console: The developer console is an integrated developer environment and collection of tools that we can use to create, debug and test applications in our salesforce org.
2. Code Editor in Salesforce Interface: This code editor compiles all classes and triggers then they are some and flags the errors if there are any.

Note: The Code doesn't get saved until it compiles without errors. The force.com developer console allows you to write, test and debug our Apex code. On the other hand the code editor in the user interface enables only writing the code and does not support debugging or testing.

Apex Variables:

A variable is a named value holder in memory. In Apex, local variables are declared with Java-like syntax.

The name we choose for a variable is called an identifier. Identifier can be of any length but it must begin with an alphabet. The rest of the identifier can include digits also.

- ➔ Operators and spaces are not allowed.
- ➔ We can't use any of the apex reserved keywords when naming variables, methods or classes.
- ➔ Keywords are the words that are essential for Apex language.

```
Integer i_a;  ☐                    Integer i 2;   ✗
Integer _ia;  ✗                    Integer i$2;   ✗
Integer ia_;  ✗                    Integer $i2;   ✗
Integer i2;   ☐                    Integer $_i_2; ✗
```

Apex Constants:

Apex Constants are the variables whose values don't change after being initialized once.
Constants can be defined using the final keyword.
Constants can be assigned almost once either in the declaration itself or with a static initialization method, if the constant is defined in a class.

final integer a =5;

Note: We cannot use the final keyword in the declaration of a class or a method because in apex classes and methods are by-default final and cannot be overridden or inherited without using the virtual keyword.

Apex Literals:

```
Integer i = 123;
String str = 'abc';
```

Expression:

An Expression is a combination made up of variables, operators and method invocation that evaluates to a single value.

Datatypes:

Apex is a strongly typed language i.e we must declare the datatype of a variable when we first refer it.
All apex variables are initialized to null by-default.

Primitive Datatypes:

1. Integer Types:
   There are 2 datatypes that we can use to store values.
   a. Integer (4 bytes) → Range(-2147483647 to +2147483647)
   b. Long (8 bytes) → Range (-2^{63} to $+2^{63}-1$)

   Note: If the literal goes out of integer range then append l or L at the end of the literal because by-default a number is treated as integer. Eg. long L = 3434343434L;
   Note: If the value is not in the range of type defined then the compiles time error is generated.
2. Floating Point Datatypes:
   A floating point variable can represent a very wide range but with a fix number of digits in accuracy.
   a. Double (8 bytes)

b. Decimal: A number that includes a decimal part. Decimal is a arbitrary precision number.

Note: Currency fields are automatically define as type decimal.

3. Date, Time and DateTime:

   A value that indicates a particular date and time.

   Ex: `Date d = Date.newInstance(2016,6,15);`

   A value that indicates a particular time. Time values must always be created with a system static method. Time datatype stores time (hours, minutes, seconds, milliseconds).

   Ex: `Time t = Time.newInstance(12,5,2,7);`

   A value that indicates a particular date and time.

   Ex: `DateTime dt = Date.newInstance(1997,1,31,7,8,16);`

4. Boolean:

   This variable can either be true or false or null.

5. String:

   Any set of characters surrounded by single quotes. String can be null or empty and can include leading and trailing spaces.

   Ll

   Ex: `String s = 'a';`

6. ID:

   Any valid ID (18 character force.com identifier). If you set ID to a 15 character value then apex converts its value to its 18 character representation.

   Note: All invalid ID values are rejected with a runtime exception.A collection of binary data stored as a single object.

7. BLOB:

   Blob is typically used to store images, audio or other multimedia objects and sometimes binary executable code is also stored as a blob. We can convert this datatype to string or from string using the toString() and valueOf() methods.

   Ex:

```
String s = 'abc';
Blob b = Blob.valueOf(s);
String s1 = b.toString();
```

sObjects and Generic sObjects:

Unlike any other programming language like Java or C#, Apex is tightly integrated with the database.

Hence we do not have to create any database connection to access the records or insert new records.

Instead, in Apex, we have sObjects which represent a record in Salesforce.

For example:

An account record named as Burlington Textiles in apex will be referred using an sObject, like this:

Account acc = new Account(Name='Disney');

The API object name becomes the data type of the sObject variable in Apex.

Here,

Account	=	sObject datatype
acc	=	sObject variable
new	=	keyword to create new sObject Instance
Account()	=	Constructor which creates an sObject instance
Name = 'Disney'	=	Initializes the value of the Name field in account sObject

Similarly if we want to create a contact record using Apex then we first need to create a sObject for it in Apex, like this:

Contact con = new Contact();

Now there are 2 ways to assign field values to the contact sObject:

1. Through Constructor:
   Contact con = new Contact(firstName = 'Shrey',lastName = 'Sharma');
2. Using dot notation:
   Contact con = new Contact();
   con.firstName = 'Shrey';
   con.lastName = 'Sharma';

SImilarly for Custom Objects:

Student__c st = new Student__c(Name = 'Arnold');

If we want to assign the field values for custom fields then also we have to write down their field API name, like:

<u>For standard object:</u>
Account acc = new Account(Name = 'Disney', NumberOfLocations__c = 56);
<u>For custom object:</u>

Student__c st = new Student__c(Name = 'Arnold', Email__c = 'arnold @gmail.com');

Generic sObject:

Generally while programming we use specific sObject type when we are sure of the instance of the sObject but whenever there comes a situation when we can get instance of any sObject type, we use generic sObject.

Generic sObject datatype is used to declare the variables which can store any type of sObject instance.

Ex

sObject s1 = new Account(Name = 'DIsney');

sObject s2 = new Contact(lastName = 'Sharma');

sObject s3 = new Student__c(Name = 'Arnold');

sObject Variable ---> Any sObject Datatype instance

Note: Every Salesforce record is represented as a sObject before it gets inserted into the database and also if you retrieve the records already present into the database they are stored in sObject variable.

Note: We can also cast the generic sObjects in specific sObjects like this:

Account acc = (Account) s1;
Contact con = (Contact) s1; *//Will throw a Runtime Exception: datatype mismatch*

1. Setting and Accessing values from Generic sObjects:
   Similar to the sObject, we can also set and access values from Generic sObject. However, the notation is a little different.
   Examples of setting the values and accessing them:
   a) Set a field value on an sObject
      sObject s = new Account();
      s.put('Name', 'Cyntexa Labs');
   b) Access a field on an sObject
      Object objValue = s.get('Name');

Enums:

An enum is a abstract datatype with values that each take on exactly one of the finite set of identifiers that you specify. Enums are typically used to define a set of possible values that don't otherwise have a numerical order such as suit of card or a particular season of the year.

Ex:
```
Public enum seasonwinter, summer, spring, fall);
Season s=season.winter; // define it into a new file to make it global
System.debug(s);
```

Rules of conversion:

In general, apex requires explicit conversion from one datatype to another however a few datatype can be implicitly converted like variables of lower numeric type to higher types.
Hierarchy of numbers: integer<long<double<decimal

Note: Once a value has been passed from a number of lower type to a number of higher type, the value is converted to the higher type of number.

- ➔ Ids can always be assigned to strings.
- ➔ String can be assigned to ids however at runtime the value is checked to ensure that it is a legitimate id, if it is not then a runtime error is thrown.
- ➔ If the numeric value of the right hand side exceeds the maximum value for an integer you get a compilation error. In this case, the solution is to append L or l to the numeric value so that it represents a long value which has a wider range.
- ➔ Arithmetic computation that produce values larger than the max values of the current type are set to overflow.
  Ex: Integer i= 2147483647 + 1; // overflow

Collections

Apex language provides developer with three classes (Set, List, Map) that makes it easier to handle collection of objects. In a sense, these collections works somewhat like arrays except their size can change dynamically and they have more advanced behaviours and easier access methods than arrays.

1) List :

Violet	Indigo	Blue	Green	Yellow	Orange	Red
0	1	2	3	4	5	6

Lists are used to store data in sequence. These are the widely used collections which can store primitives, user defined objects, sObjects, Apex Objects or other collection.

There are 2 main properties of lists:

a. It stores data in sequential order.
b. The data which it stores is non-unique or can be duplicate.

Hence use a list when the sequence of elements is important and where uniqueness of the elements are not important.

List initialization

```
List<datatype> list1 = new List<datatype>();
Example:
      List<String> stList = new List<String>();
      stList.add('ABC');
      stList.add('DEF');
      stList.add('FGH');

OR
      List<String> stList = new List<String>{'ABC','DEF','FGH'};
```

List Array Notation

Developers comfortable with Array notation can also use it while using Lists. Array and List syntax can be used interchangeably. Array notation is a notation where a list's elements are referenced by enclosing the index number in square brackets after the list's name.

```
String[] nameList = new String[4];
//or
String[] nameList2 = new List<String>();
//or
List<String> nameList3 = new String[4];
// Set values for 0, 1, 2 indices
nameList[0] = 'Bhavna';
nameList[1] = 'Bhavya';
nameList[2] = 'Swati';
// Size of the list, which is always an integer value
Integer listSize = nameList.size();
// Accessing the 2nd element in the list, denoted by index 1
System.debug(nameList[1]); // 'Bhavna'
```

Some common methods of list:

add(element)	It adds an element into the list. it always adds at the last.	`List<String> l = new List<String>();`
		`l.add('abc');`
		`System.debug(l); // ('abc')`

size()	Returns the number of elements in the list.	`l.add('def');`
		`System.debug(l.size()); // 2`
		`System.debug(l); // ('abc','def')`

get(index)	Returns the element on the ith index.	`System.debug(l.get(0)); // 'abc'`

remove(index)	Removes the element on ith index.	`l.remove(1);` `System.debug(l); // ('abc')`
clone()	Makes a duplicate of a list.	`List<String> l2 = l.clone();`

set()	Sets the element on the ith position of the list. If there is already a value then value gets overridden.	`l.add('def');`
		`l.add('ghi');`
		`System.debug(l);//('abc','def','ghi')`
		`l.set(2,'aaa');`
		`System.debug(l);//('abc','def','aaa')`

sort()	Sorts the item in ascending order but works with primitive datatypes only.	`l.sort();` `System.debug(l);//('aaa','abc','def')`
isEmpty()	Returns true if the list is empty.	`System.debug(l.isEmpty()); // false`
clear()	Clears the list.	`l.clear()`

Lists can be multidimensional also:

a. 2-Dimensional:

```
List<List<Integer>> nestList = new List<List<Integer>>();
```

b. 3-Dimensional:

```
List<List<List<Integer>>> nestList = new List<List<List<Integer>>>();
```

Note: SOQL queries also returns list of records.

2) Set :

Value	'Lists'	'Sets'	'Maps'

A set is a collection of unique, unordered elements.

It can contain primitive datatypes (string, integer, date, etc.) , sObjects or user-defined objects.

The basic syntax of creating a set :

Set<datatype> set1 = new Set<datatype>; OR

Set<datatype> = new Set<datatype>{value1, value2,...};

a) The main characteristic of a set is the uniqueness of the elements. Yo can safely try to add the same element to a set more than once and it will disregard the duplicate without throwing an exception.

Set<String> collections = new Set<String>{'Lists', 'Sets', 'Maps'};
s1.add('lists');

b) However, there is a slight twist with sObject i.e. the uniqueness of the data respects Case Sensitivity.

c) Uniqueness of sObject is determined by comparing fields in their object.

d) The way you declare sets is similar to the way you declare lists. You can also have nested sets and sets of lists.

Ex:
```
// Empty set initialized
Set<String> strSet = new Set<String>();
// Set of List of Strings
Set<List<String>> set2 = new Set<List<String>>();
```

e) Some common methods of set:

add(element)	Adds an element to set and only takes the argument of special datatype while declaring the set.	`Set<String> s = new Set<String>();`
		`s.add('abc');`
		`s.add('ABC');`
		`s.add('abc');`
		`System.debug(s);//('abc','ABC')`

addAll(list/set)	Adds all of the elements in the specified list/set to the set if they are not already present.	`List<String> l = new List<String>();`
		`l.add('abc');`
		`l.add('def');`
		`s.addAll(l);`
		`System.debug(s);//('abc','ABC','def')`

clear()	Removes all the elements.	`s.clear();` `s.addAll(l);`
clone()	Makes duplicate of a set.	`List<String> s2 = s.clone();` `System.debug(s);//('abc','def')`
contains(elm)	Returns true if the set contains the specified element.	`Boolean result = s.contains('abc');` `System.debug(result); // true`
containsAll(list)	Returns true if the set contains all of the elements in the specified list. The list must be of the same type as the set that calls the method.	`Boolean result = s.containsAll(l);` `System.debug(result); // true`
size()	Returns the size of set.	`System.debug(s.size()); // 2`
retainAll(list)	Retains only the elements in this set	`s.add('ghi');`

	that are contained in the specified list and removes all other elements.	`System.debug(s);//('abc','def','ghi')`
		`s.retainAll(l);`
		`System.debug(s);//('abc','def')`

remove(elm)	Removes the specified element from the set if it is present.	`s.add('ghi');`
		`s.remove('ghi');`
		`System.debug(s);//('abc','def')`

removeAll(list)	Removes the elements in the specified list from the set if they are present.	`s.add('ghi');`
		`s.removeAll(l);`
		`System.debug(s);//('ghi')`

3) Map :

130	131	132	133	134	135
'Bhavya'	'Divya'	'Bhawna'	'Sapna'	'Pushpa'	'Harsha'

a) Map is collection of key, value pairs. Keys can be any primitive datatype while values can include primitives, apex objects, sObjects and other collections.

b) Use maps when want to quickly find out something with help of a key, each key must be unique but you can have duplicate values in your map.

c) Syntax for declaration:

```
Map<Datatype_Key, Datatype_value> m = new Map<Datatype_Key, Datatype_value>();
```

d) To create a map of all the account records in your org, the Salesforce ID should be the key, which is the unique identifier for all the values of the account record.

Ex:
```
// Nested map
Map<ID, Set<String>> m = new Map<ID, Set<String>>();
```

e) Initialisation of Map:

```
Map<Integer, String> m = new Map<Integer, String>{5=>'Kalpana'};
m.put(1, 'Bhavna');
m.put(2, 'Sapna');
m.put(3, 'Divya');
m.put(4, 'Bhavya');
m.put(2, 'Divya'); // this will override previous value
```

f) Map of List:

Map<Integer, List<Integer>> m = new Map<Integer, List<Integer>>();
m.put(1, new List());

g) Methods of Map:

put(key,value)	Associates the specified value	`Map<Integer,String> m = new Map<Integer,String>();` `m.put(1, 'Bhavna');`

	with the specified key in the map.	`m.put(2, 'Sapna');` `System.debug(s);//(1='Bhavna',2='Sapna')`

putAll(map)	Copies all of the mappings from the specified map to the original map.	`Map<Integer,String> n = new Map<Integer,String>();`
		`n.put(3, 'Bhavya');`
		`n.put(4, 'Divya');`
		`m.putAll(n);`
		`System.debug(m);// (1='Bhavna',2='Sapna' 3='Bhavya',4='Divya')`

get(key)	Returns the value to which the specified key is mapped, or null if the map contains no value for this key.	`System.debug(m.get(1)); // 'Bhavna'`
		`System.debug(m.get(2)); // 'Sapna'`
		`System.debug(m.get(3)); // 'Bhavya'`
		`System.debug(m.get(4)); // 'Divya'`

values()	Returns a list that contains all the values in the map.	`List<String> l = new List<String>();`
		`l = m.values();`
		`System.debug(l);`
		`// ('Bhavna','Sapna','Bhavya','Divya')`

clone()	Makes a duplicate copy of the map.	`Map<Integer,String> o = new Map<Integer,String>();`
		`o = m.clone();`
		`System.debug(o);// (1='Bhavna',2='Sapna' 3='Bhavya',4='Divya')`

keySet()	Returns a set that contains all of the keys in the map.	`System.debug(m.keySet());` `// (1,2,3,4)`

containsKey (key)	Returns true if the map contains a mapping for the specified key.	`Boolean result1 = m.containsKey(3);`
		`System.debug(result1); // true`
		
		`Boolean result2 = m.containsKey(6);`
		`System.debug(result2); // false`

isEmpty()	Returns true if map has 0 key.	`Boolean result = m.isEmpty();` `System.debug(result); // false`
size()	Returns the number of key-value pairs	`System.debug(m.size()); // 4`

	in the map.	
remove(key)	Removes the mapping for the specified key from the map, if present, and returns the corresponding value.	`m.remove(4);` `System.debug(m);//(1='Bhavna',2='Sapna'3='Bhavya')`
clear()	Removes all of the key-value mappings from the map.	m.clear();

Note: Maps are used frequently to store records that you want to process or as containers for lookup data. It is very common to query for records and store them in a map so that you can do something with them.

Operators:

1. Add/Sub:

   This operator adds or subtracts the value of Y from the value of X according to the following rules.

   - ➔ If X and Y are integers or doubles then adds or subtracts the value of Y from X and if any X or Y is double then the result is double.
   - ➔ If X is a date and Y is an integer then it returns a new date that is incremented or decremented by the specified number of days.
   - ➔ If X s a DateTime and Y is an integer or double then it returns a new DateTime which is incremented or decremented by specified number of days with a fractional portion corresponding to a portion of the day.

   ```
   Ex: main() {
           Double d = 5.2;
           DateTime dt = new DateTime.now();
           DateTime finalDate = dt -d ;
           System.debug(dt);
           System.debug(finalDate);
   }
   ```

   Note: Only in case of '+' if X is a string and Y is a string or any other type of non-null argument then it concatenates Y to the end of X.

2. Shorthand Operator:

   a. |= (OR assignment operator):

   If X(boolean) and Y(boolean) are both false then X remains false otherwise X is assigned true.

   b. &= (AND assignment operator):

If X(boolean) and Y(boolean) are both true then X remains true otherwise X is assigned false.

3. Equality Operator(==):

If a value of X equals Y, the expression evaluates to true otherwise the expression evaluates to false.

Note: Unlike java(==), in apex it compares object value equality not reference equality except for user-defined types..

Note: String comparison using(==) is case insensitive.

Note: ID comparison using == is case sensitive.

- ➔ User-defined types are compared by reference which means that the 2 objects are equal if they reference the same location in the memory.
- ➔ You can override this default comparison behavior by equals() and hashcode() methods in your class to compare object values instead.

```
main()
{
        Integer a[] = new Integer[5];
        Integer b[] = new Integer[5];
        for(Integer i=0; i<5; i++)
        a[i]=i+1;
        System.debug(a);
        System.debug(b);
        System.debug(a==b);
}
```

Note: Arrays(==) performs a deep check of all the values before returning its result. Likewise for collection and built-in apex objects.

Note: The comparison of any two values can never result in null though X and Y can be literal null.

Note: SOQL and SOSL use '=' for their equality operator not '=='.

4. Exact Equality Operator(===):

If X and Y reference the exact same location in memory, the expression evaluates to true otherwise false.

```
main()
{
        FirstClass fc = new FirstClass();
        FirstClass fc2 = new FirstClass();
        System.debug(if(fc===fc2));
}
```

5. Exact Inequality Operator(!==):

If X and Y do not reference the exact same location in emory, the expression evaluates to true otherwise false.

6. Relational Operators(<,>,<=,>=):

a. If X or Y equals null and are integers, doubles, dates or DateTimes then it will return false.
b. A non-null string or id value is always greater than a null value.
c. If X and Y are ids then, they must reference the same value otherwise a runtime error occurs.
d. If X or Y is an id and other value is a sting then the string is validated and treated as an id.
e. X and Y can not be boolean.

```
Id i1 = "------";
Id i2 = "------";
System.debug(i1>i2);
```

7. Operator Precedence:
    a. () {} ++ --
    b. -x ! +x
    c. */
    d. +-
    e. < > <= >= instanceOf
    f. == !=
    g. &&
    h. ||
    i. += == *=