Topic	Apex
Created By	Siddharth Pandit (Salesforce Instructor)
Contact No.	+91 879 333 5440 / 988 12 988 13
Email	Siddharthpandit.salesforce@gmail.com

- **Apex supports a number of different data types:**
  I. primitive data types such as Integer and Date
  II. sObject types that represent persistent objects
  III. collections and enumerations

- **Primitive Data Types:**
  I. Blob - for storing binary data
  II. Boolean
  III. Date, Time and Datetime
  IV. Decimal - for representing arbitrary precession numbers, including currency
  V. ID - the Force.com database record identifier type
  VI. Integer, Long, Double and String

Here are some variable definitions for primitives that should give you a feeling for using these data types:

```
1 DateTime dt = System.now() + 1;
2 Boolean mustI = true;
3 String s = 'abc'.toUpperCase();
4 Decimal d = Decimal.valueOf('123');
```

- **Collections:**
  I. Apex supports Sets, Lists and Maps, as well as enumerations
  II. These are pretty straightforward, so we'll just provide a few examples here
  III. Note that the first element of a collection is at index position zero

a. **Sets**:

A set is an unordered collection of primitives that does not contain any duplicate elements:

```
Set<String> s = new Set<String>{'a','b', 'c'};
s.add('c');
System.assert(s.contains('b'));
System.assert(s.size() == 3);
```

b. **Lists**:

A list is a collection of elements. Use a list when the sequence is important. You can have duplicates in a list:

```
List<Integer> myList = new List<Integer>();
myList.add(47);
myList.get(0);
```

You can also use the array syntax for lists. For example:

```
String [] colors = new List<String> ();
colors[3] = 'Green';
```

c. **Maps:**

Maps are collections of key-value pairs, and support a shortcut syntax for populating the collection:

```
Map<String,String> myStrings = new Map<String,String>{'a'=>'b','c'=>'d'.toUpperCase()};
Map<ID,Contact> m = new Map<ID, Contact>([select id, lastname from contact]);
```

The last statement above contains a sneak peek of the database integration, populating the map with retrieved contact objects, which in turn only have their `id` and `lastname` fields populated.

d. ENUM:

Apex also supports enumerations. The following code creates a new data type called Season, then declares a variable of that data type, and assigns it a value.

```
public enum Season {WINTER, SPRING, SUMMER, AUTUMN}
Season s = Season.AUTUMN;
```

❖ **Statements and Expressions:**

I. Below example is just to give you a test

```
Integer count = 1;
while (count < 11) {
  System.debug(count);
  count++;
}
for (Integer i = 0, j = 0; i < 10;i++){
  System.debug(i+1);
}
Integer[] myInts = new Integer[]{1,2,3,4,5,6,7,8,9,10};
for (Integer i : myInts) {
  System.debug(i);
}
```

II. The last for loop shows how to iterate over a collection. Apex also supports a special loop for iterating over persisted sObject records returned by a query:

```
String s = 'Acme';
for (Account a : [select id, name from account where name like :(s+'%')]){
  //Your code
}
```

Note how the query is embedded in square brackets, and how it makes a reference to the variable s.

❖ **Database Integration:**

I. In this section we'll show how the language can be used to create, persist and update database objects (called sObjects), as well as query the database and iterate over results

II. It is also used for writing database triggers

III. sObject refers to any object that can be stored in the Force.com platform database

IV. These are not objects in the sense of instances of Apex classes; rather, they are representations of data that has, or will be, persisted

V. sObject is also the name of the generic abstract type that can be used to represent any persisted object type. Assuming the database has an Account sObject, with fields for name and billingcity, then either of the following two lines will create an sObject:

```
1 sObject s = new Account();
2 Account a = new Account( name='Acme', billingCity='Edinburgh');
```

❖ **Queries and Embedded Queries:**

The Force.com platform supports two query languages:

I. Salesforce Object Query Language (SOQL) is a query-only language. While similar to SQL in some ways, it's an object query language that uses relationships, not joins, for a more intuitive navigation of data. You'll see an example in a minute.

II. Salesforce Object Search Language (SOSL) is a simple language for searching across all persisted objects.

III. These languages can be embedded in your Apex code, making data retrieval easy. This code retrieves an sObject (a record from the database) that has the name field assigned to Acme:

```
1 sObject s = [select id, name from account where name='Acme'];
```

IV. This code retrieves all matching accounts (assuming that there'll be zero or more), assigning them to a list:

```
1 String myName = 'Acme';
2 Account [] accts = [select ID from Account where name=:myName];
```

V. The following code retrieves the first matching account and assigns the annualRevenue field value to a variable:

```
1 Double rev= [select annualRevenue from Account where name = 'Acme'][0].annualRevenue;
```

❖ **Triggers:**

I. Triggers are written in Apex, and execute before or after an insert, update, delete or undelete event occurs on an sObject

II. The syntax that introduces a trigger definition will look a little familiar, and begins with the `trigger` keyword:

```
01 trigger myAccountTrigger on Account (before insert, before update) {
02   if (Trigger.isInsert) {
03    //
04   }
05   if (Trigger.isUpdate) {
06     for(Account a: Trigger.new)
07       if (a.name == 'bad')
08         a.name.addError('Bad name'); // prevent update
09      }
10 }
```

This example fires before any `Account` sObject is inserted or updated.
The `Trigger.new` context variable provides access to the list of accounts being inserted or updated. Update and delete triggers can use `Trigger.old` to refer to the old versions of the objects being updated or deleted.

III. Triggers make full use of Apex, allowing you to continue using a familiar language for data manipulation

IV. There are a few restrictions - for example it doesn't make sense to allow call outs to web services from within a trigger as that would unduly lengthen the transaction

❖ **Data Manipulation Language:**

I. Apex code can also contain data manipulation language (DML) operations to retrieve, insert, delete and update data in the database

II. You can also create and manipulate save points.

Here is an examples:

```
1 Account[] accounts = new account[] {new Account(name='foo'), new Account(name='bar')};
2 insert accounts;
```

❖ **Testing:**

I. The Force.com platform requires that at least 75% of your Apex classes are covered by testing before code can be deployed to a production system

II. Ideally, you should strive for 100% coverage. As a result, Apex development usually goes hand in hand with unit test development

III. This restriction isn't in place in the development edition organization

IV. In Apex, test methods are denoted with the `testMethod` keyword

V. Here's an example of a class that provides a test method:

```
01 public class myClass {
02   static testMethod void myTest() {
03     Account a = new Account(name='foo');
04     insert a;
05     System.assertEquals('foo', [select name from Account where id=:a.id].name);
06     System.assertEquals(1, [select count() from Account where id=:a.id]);
07     try {
08       delete a;
09       Foo.myMethod(a); //call some method
10     } catch (DmlException e) {
11       System.assert(false);  // assert that we should never get here
12     }
13   }
14 }
```

These tests typically use the `System.assert()` series of methods to flag code behavior and catch all exceptions

❖ **Apex as a Controller Language:**

I. Visualforce is the user interface layer on the Force.com platform

II. It provides a model-view-controller (MVC) paradigm to creating user interfaces, where incoming web requests (from a browser, say) can be routed to a controller that can perform certain actions, and then display a result.

III. Visualforce controllers and extensions must be written in Apex

IV. These controllers typically look no different to the Apex code appearing in this article, though they do have access to additional data types, classes, methods and properties

V. For example, in Visualforce controllers you have access to page parameters, and Visualforce pages themselves are first-class citizens within Apex, allowing you to redirect to various pages, render the pages as PDF and email as a blob, and so on.

Here's a simple controller:

```
01 public class MyController {
02   PageReference where;
03   public PageReference dynamicJump() {
04     if (ApexPages.currentPage().getParameters().get('p') != null) {
05       where = Page.foo;
06       where.setRedirect(true);
07     } else {
08       where = Page.bar;
09       where.getParameters().put('p', 'nowHasParam');
10     }
11     return where;
12 }
```

PageReference is the data type of a Visualforce page, and this code examines the parameters of the current page and redirects to one of two other pages depending on what it finds. Such controllers can also execute DML, query the database and return results for example.