salesforce

MALLA REDDY UNIVERSITY

R-22

III YEAR B.TECH. (CSE) / I – SEM

MR22-1CS0155

Salesforce Platform Developer

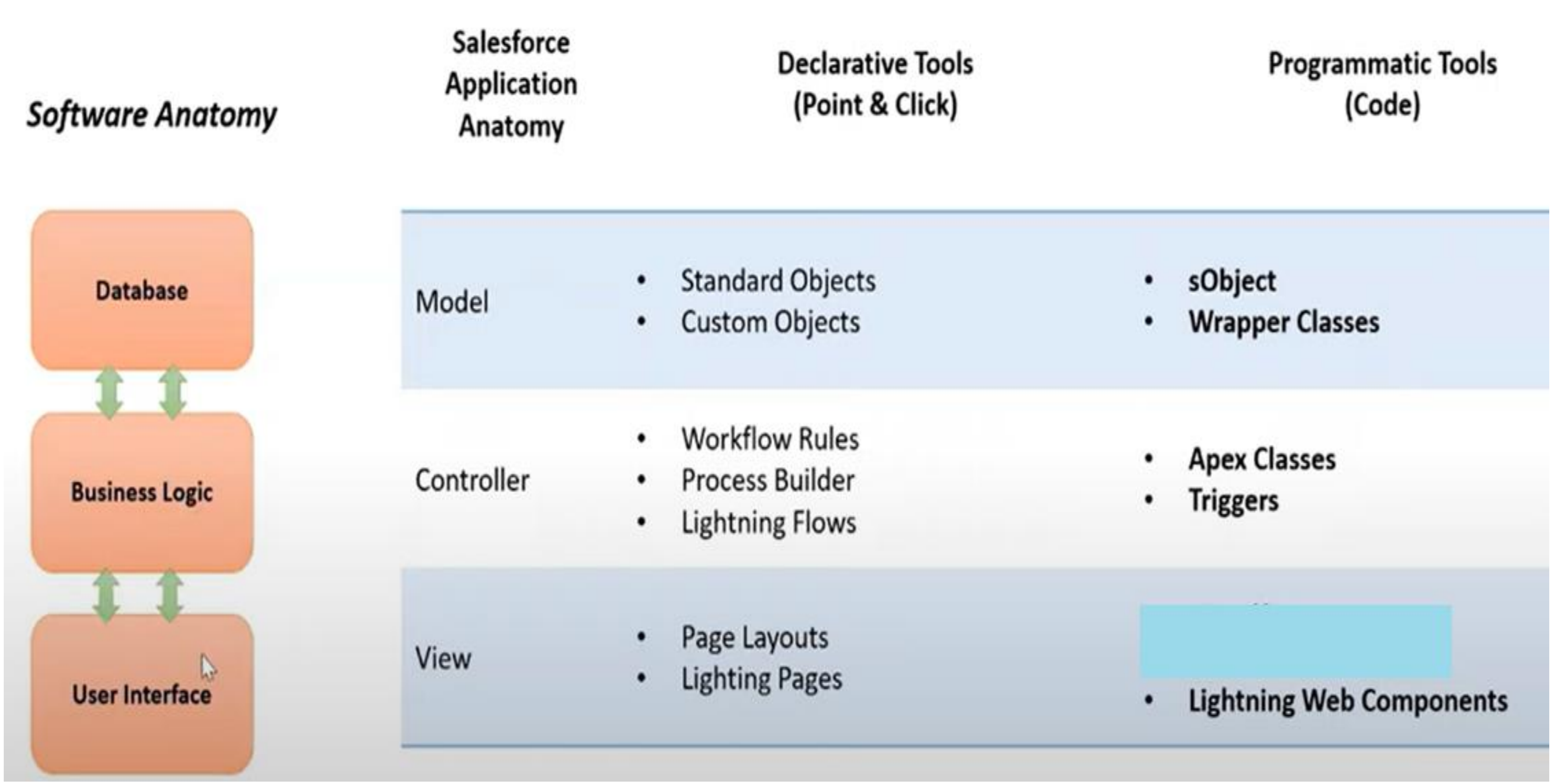
Dr. Shaik Hussain S I
Associate Prof/CSE
MRU

UNIT–II

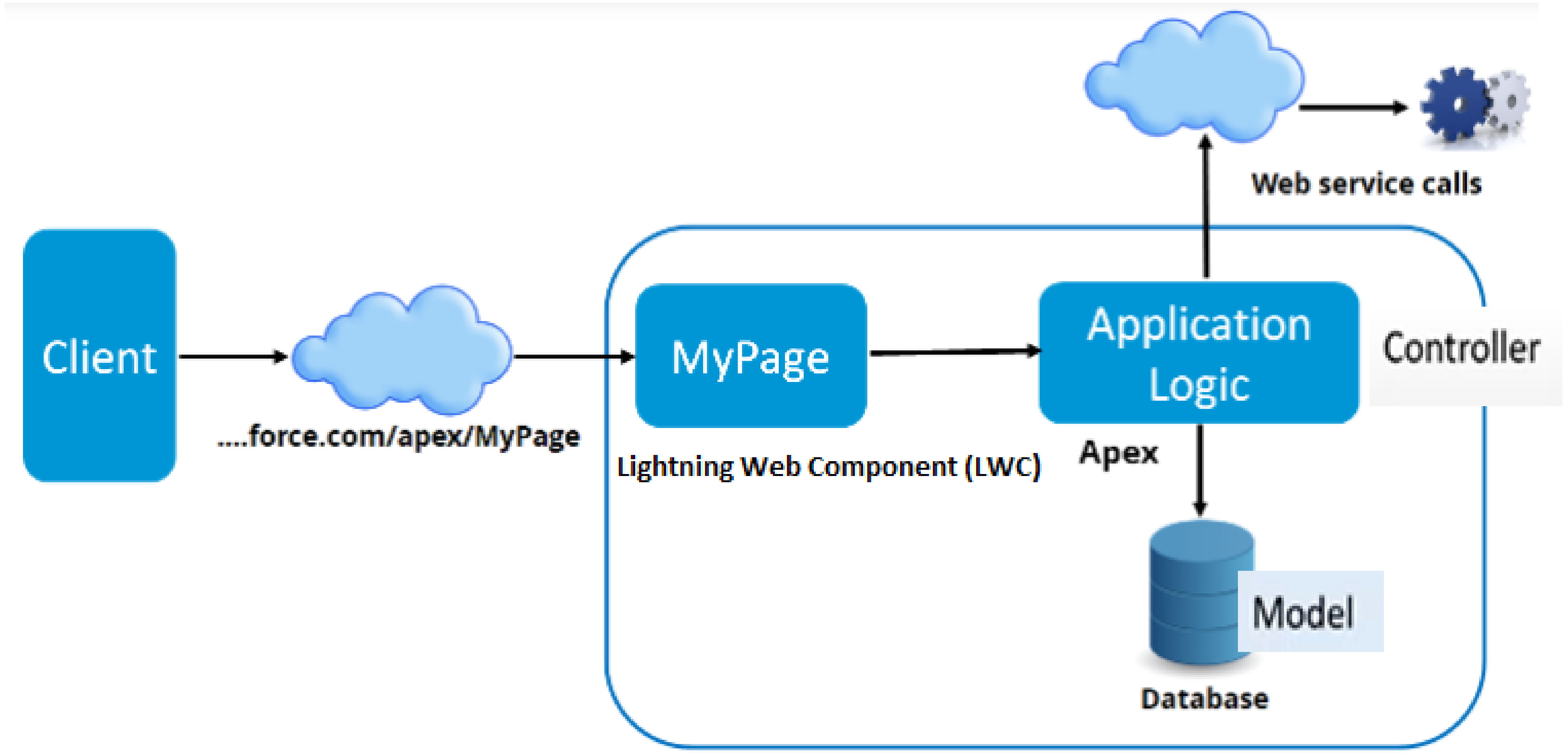
Apex Fundamentals: Introduction to Apex, Apex Classes, Database structure, Execute SOQL and SOSL Queries.

Apex Triggers: Introduction Apex Triggers, Bulk Apex Triggers.

Salesforce Application Anatomy (MVC Architecture)


Software Anatomy
Database
Business Logic
User Interface

Salesforce Application Anatomy	Declarative Tools (Point & Click)	Programmatic Tools (Code)
Model	• Standard Objects • Custom Objects	• **sObject** • **Wrapper Classes**
Controller	• Workflow Rules • Process Builder • Lightning Flows	• **Apex Classes** • **Triggers**
View	• Page Layouts • Lighting Pages	[redacted] • **Lightning Web Components**



The client or user either requests or provides information to the Salesforce application. This is generally done using **LWC (Lightning Web Component).**

This information is then passed on to the **application logic layer, written in Apex**.

Depending upon the information, data is either inserted or removed from the database.

Salesforce also provides you with the option of using web services to directly access the application logic.

Introduction to Apex

What is Apex?

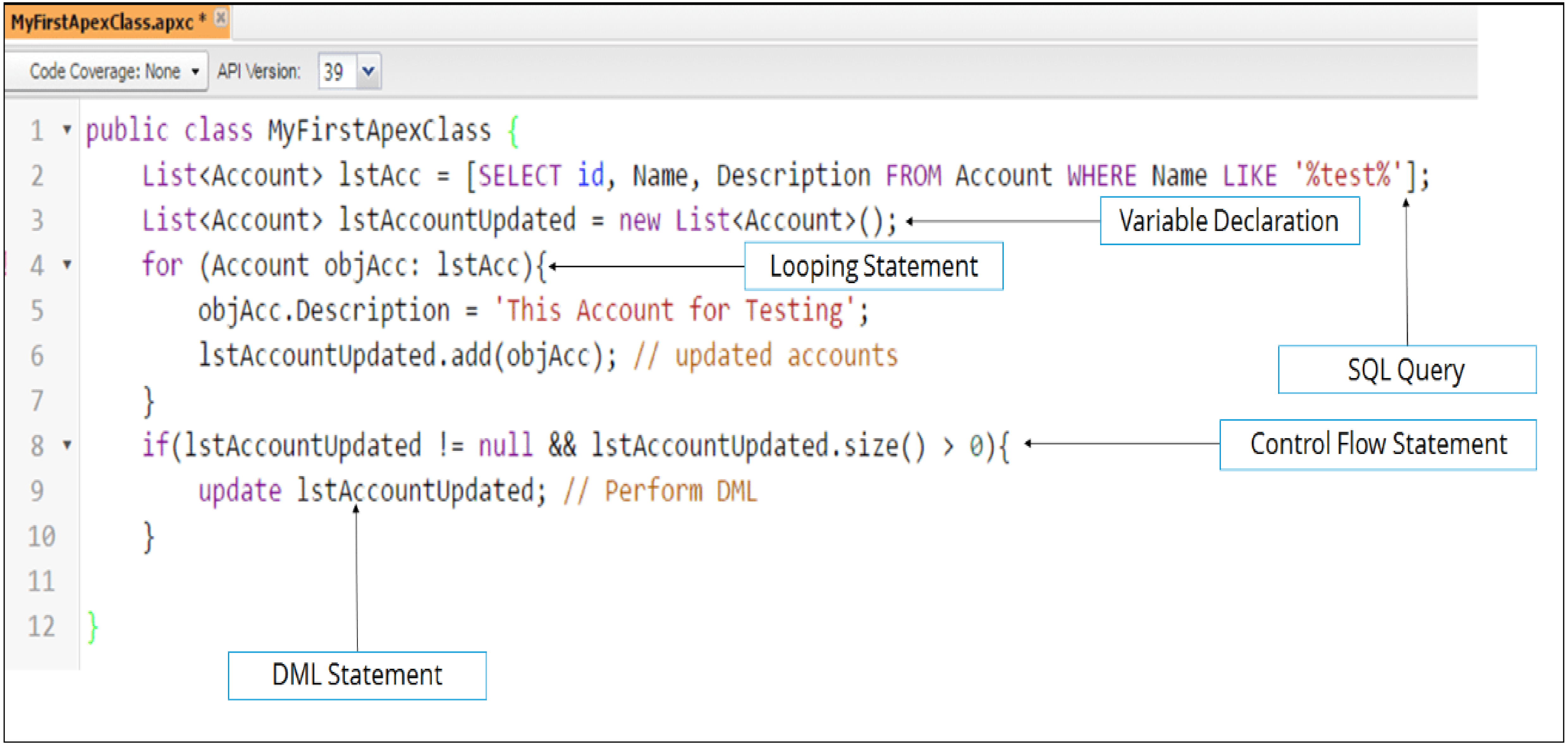
- Apex is a **strongly typed, object-oriented programming language developed by Salesforce**.

- It is used primarily for **building custom business logic within the Salesforce platform** (Force.com).

- Apex uses Java-like syntax and acts like database stored procedures.

- It enables developers to add business logic to system events, such as button clicks, updates of related records.

For Example



```
public class MyFirstApexClass {
    List<Account> lstAcc = [SELECT id, Name, Description FROM Account WHERE Name LIKE '%test%'];
    List<Account> lstAccountUpdated = new List<Account>();
    for (Account objAcc: lstAcc){
        objAcc.Description = 'This Account for Testing';
        lstAccountUpdated.add(objAcc); // updated accounts
    }
    if(lstAccountUpdated != null && lstAccountUpdated.size() > 0){
        update lstAccountUpdated; // Perform DML
    }

}
```

Introduction to Apex

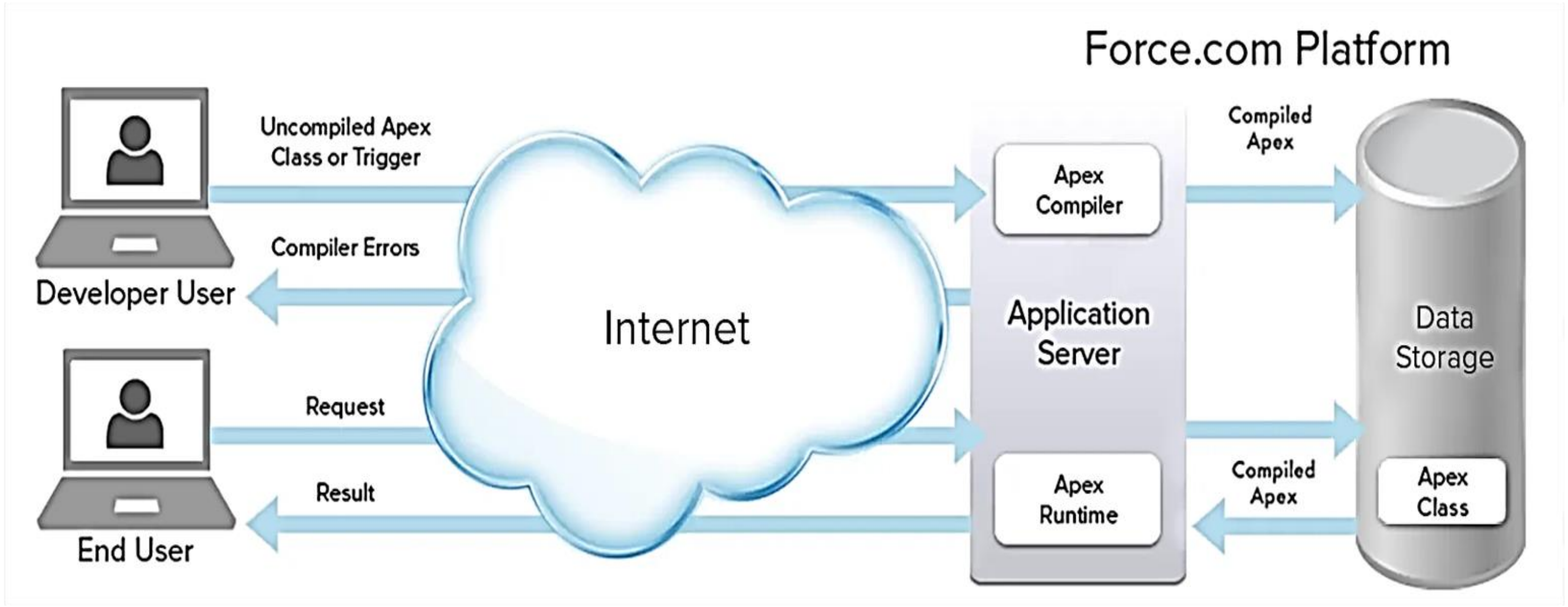
Features of Apex:

- **Hosted** - saved, compiled, and executed on the server.
- **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 provides direct access to records and their fields, and provides statements and query languages to manipulate those records.
- **Data focused** - Apex provides transactional access to the database, allowing you to roll back operations.

Introduction to Apex

- **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. Salesforce ensures that all custom Apex code works as expected by executing all unit tests prior to any platform upgrades.
- **Versioned** - Custom Apex code can be saved against different versions of the API.
- **Integrated with DML & APIs -** Apex has in-built DML activities like Insert, Delete, Update and DML exemption dealing with. It upholds loops that permit the handling of numerous records all at once.

Introduction to Apex



Introduction to Apex - Data Types

•The Apex language is strongly typed so every variable in Apex will be declared with the specific data type.

•All apex variables are initialized to null initially.

•It is always recommended for a developer to make sure that proper values are assigned to the variables.

•Otherwise such variables when used, will throw null pointer exceptions or any unhandled exceptions.

Types:

- **Primitive Data types**
- **sObject**
- **Collections**
- **Enum**

Introduction to Apex - Data Types

Primitive Data types:

- **Integer:**

A 32-bit number that does not include any decimal point.

Example:

```
Integer barrelNumbers = 1000;
System.debug(' value of barrelNumbers variable: '+barrelNumbers);
```

- **Long:**

This is a 64-bit number without a decimal point. This is used when we need a range of values wider than those provided by Integer.

Example:

```
Long companyRevenue = 21474838973344648L;
System.debug('companyRevenue'+companyRevenue);
```

Introduction to Apex - Data Types

- **Double:**

A 64-bit number that includes a decimal point. Doubles have a minimum value of -263 and a maximum value of 263-1.

Example:

```
Double pi = 3.14159;
Double e = 2.7182818284D;
```

- **Boolean:**

This variable can either be true, false or null.

Example:

```
Boolean shipmentDispatched;
shipmentDispatched = true;
System.debug('Value of shipmentDispatched '+shipmentDispatched);
```

Introduction to Apex - Data Types

- **Date:**

This variable type indicates a date. This can only store the date and not the time. For saving the date along with time, we will need to store it in variable of DateTime.

Example:

```
Date ShipmentDate = date.today();
System.debug('ShipmentDate '+ShipmentDate);
```

- **Datetime:**

A value that indicates a particular day and time, such as a timestamp. Always create datetime values with a system static method.

- **Time:**

This variable is used to store the particular time. This variable should always be declared with the system static method.

Introduction to Apex - Data Types

- **String:**

String is any set of characters within single quotes. It does not have any limit for the number of characters.

Example:

```
String companyName = 'Abc International';
System.debug('Value companyName variable'+companyName);
```

- **ID:**

When inserting records, the system assigns an ID for each record.

Any valid 18-character Lightning Platform record identifier.

If you set ID to a 15-character value, Apex converts the value to its 18-character representation.

All invalid ID values are rejected with a runtime exception.

Example:

```
ID id='00300000003T2PGAA0';
```

Introduction to Apex - Data Types

sObject:

- In Apex, the **sObject data type represents Salesforce objects**, both standard and custom.

- Each **Salesforce record** is represented as an **sObject** before it is inserted into Salesforce.

- It's a versatile data type used to **interact with any record** in Salesforce, encompassing fields, relationships, and metadata.

- sObjects are fundamental for **CRUD operations,** enabling robust **data management and manipulation.**

- This is a special data type in Apex. **sObject is a generic data type for representing an Object that exists in Force.com.**

Types:

- **Specific sObject**
- **Generic sObject**

Introduction to Apex - Data Types

- **Specific sObject:**

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

For Example, **Account** is a datatype for a standard object "Account" and **Student__c** for a custom object called "Student", when working with sObjects.

Example:

For Standard Object:

```
Account acc=new Account();
```

Here,
Account = sObject datatype
acc = sObject variable
new = Keyword to create new sObject Instance
Account() = Constructor which creates an sObject instance

For Custom Object:

```
Student__c st = new Student__c();
```

Introduction to Apex - Data Types

Accessing SObject fields:

- There are **two ways to add fields**: through the **constructor or** by using **dot notation**.
- SObject variable can be assigned a valid object reference with the **new** operator.

EXAMPLE:

(i) Adding values to Fields through constructor by specify them as **name-value pairs** inside the constructor

```
Account acct = new Account(Name='Ajay', Phone='9856745923', NumberOfEmployees=100);
System.debug(acct.Name);
```

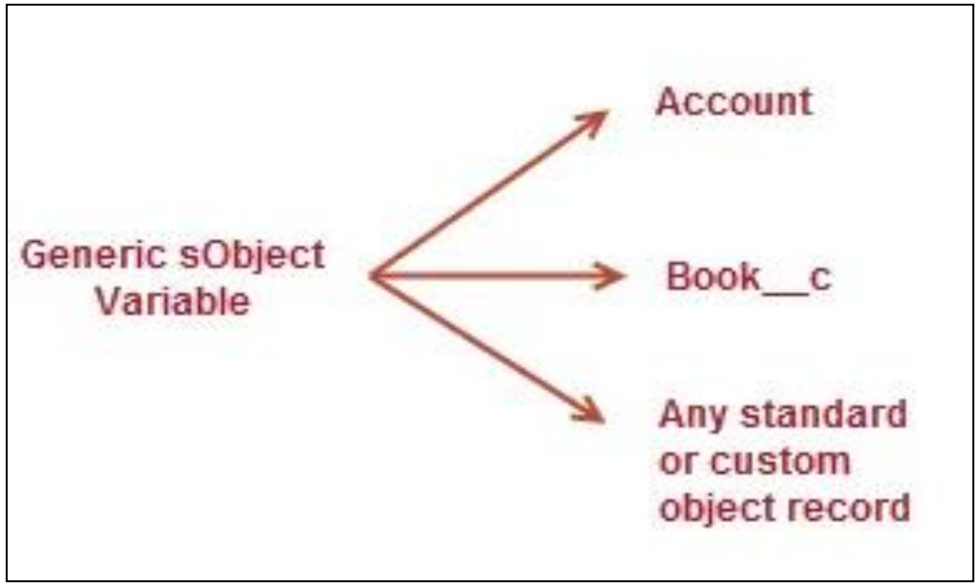
(ii) SObject fields can be accessed or changed with simple dot notation.

```
Account acct = new Account ();
 acct.Name = Acc1;
 acct.BillingCity = Washington;
```

Introduction to Apex - Data Types

Generic sObject:

- Generic sObject data type is used to declare the variables which **can store any type of sObject instance.**
- We use when we don't know the type of sObject our method is handling,
- Variables that are declared with the generic sObject data type can reference any Salesforce record, whether it is a standard or custom object record.



Introduction to Apex - Data Types

We cannot use dot operator to access the Generic sObject.

For example,

```
sObject s1;
s1=new Account();
s1.Name='Ajay'                    // Will not Work
System.debug(s1.Name);
```

Two ways to access the values of Generic sObject:

1. **Cast Generic sObjects to Specific sObject Types**
2. **Using put() and get() methods.**

Introduction to Apex - Data Types

1. Cast Generic sObjects to Specific sObject Types:

- In **Salesforce**, you can cast a generic sObject to a specific sObject type using the sObject's API name.
- 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.

Introduction to Apex - Data Types

Example:

```
sObject s1;

s1=new Account();
Account acc = (Account) s1;
acc.Name='Ajay';
system.debug(acc.Name);

s1=new Contact();
Contact con = (Contact) s1;
con.MobilePhone='98764597';
system.debug(con.MobilePhone);

sObject s2 = new Account();
Contact con = (Contact) s2;  //Will throw a Runtime Exception: datatype mismatch
```

Introduction to Apex - Data Types

2. Using put() and get() methods:

Set a field value on an sObject

- To set the value of a field on an SObject dynamically, you can use the ***put*** method on the SObject class.
- The ***put*** method takes two arguments: **the field name, and the value** you want to set.

```
sObject s = new Account();
s.put('Name', 'Cyntexa Labs');
```

Access a field value on a sObject

- To retrieve the value of a field on an SObject dynamically, you can use the get method on the SObject class.
- The get method takes one argument: **the field name**.

```
Object objValue = s.get('Name');
```

Introduction to Apex - Data Types

Collections:

- A collection is a type of variable in apex that can **store multiple items.**
- Collections have the ability to **dynamically rise and shrink** depending on the business needs.

Types of Collections

- **Salesforce has three types of collections:**

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

Introduction to Apex - Data Types

1. List:

- A list is an ordered collection of elements that are distinguished by their indices.
- **“List” is the keyword** to declare a list collection.
- Each list **index begins with 0.**
- The list can **store duplicate and null values.**
- The **list keyword** followed by the **data type** has to be used **within <> characters** to define a list collection.
- List elements can be of any data type—primitive types, collections, sObjects, user-defined types, and built-in Apex types.

Syntax for a list is as follows:

```
List<datatype> listName = new List<datatype>();
```

Example:

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

Declaring the initial values here,

```
List<String> colors = new List<String> {‘Blue', ‘Pink', ‘Green'};
```

Using the array notation as well to declare the List,

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

Introduction to Apex - Data Types

List of Accounts (sObject):

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

System.debug('Value AccountToDelete'+AccountToDelete);
```

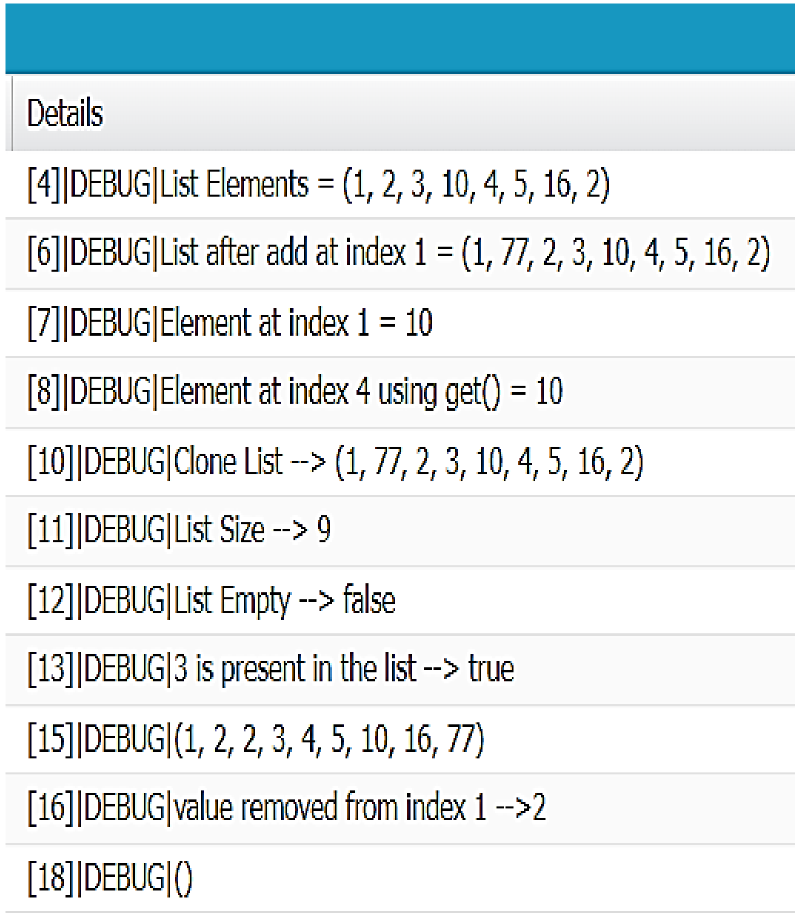
Methods for List:

add(Value) - to add elements to the list.
add(index,value) - adds elements to the list
size() - outputs the number of elements in list
get(index) - access the elements from the list at specified index position.
clear() - removes all elements from the list.
set(index,value) - overwrites element at existing index position.
isEmpty() - Returns true if the list is empty, otherwise returns false
contains() - Returns true if a particular element is present in the list, otherwise, false
Remove(index) - to remove an element from the list from the specified index

Example:

```
List<Integer> numbers = new List<Integer>{1,2,3,10,4,5};
numbers.add(16);
numbers.add(2);
System.debug( 'List Elements = ' + numbers);
numbers.add(1, 77);
System.debug('List after add at index 1 = ' + numbers);
System.debug('Element at index 1 = ' + numbers[4]);
System.debug('Element at index 4 using get() = ' + numbers.get(4));
List<Integer> num1 = numbers.clone();              // Clones the list
System.debug('Clone List --> ' + num1);
System.debug('List Size --> ' + numbers.size());
 System.debug('List Empty --> ' + numbers.isEmpty());
System.debug('3 is present in the list --> ' + numbers.contains(3));
numbers.sort();
System.debug(numbers);
System.debug('value removed from index 1 -->' + numbers.remove(1));
numbers.clear();
System.debug(numbers);
```

OUTPUT:



```
Details
[4]|DEBUG|List Elements = (1, 2, 3, 10, 4, 5, 16, 2)
[6]|DEBUG|List after add at index 1 = (1, 77, 2, 3, 10, 4, 5, 16, 2)
[7]|DEBUG|Element at index 1 = 10
[8]|DEBUG|Element at index 4 using get() = 10
[10]|DEBUG|Clone List --> (1, 77, 2, 3, 10, 4, 5, 16, 2)
[11]|DEBUG|List Size --> 9
[12]|DEBUG|List Empty --> false
[13]|DEBUG|3 is present in the list --> true
[15]|DEBUG|(1, 2, 2, 3, 4, 5, 10, 16, 77)
[16]|DEBUG|value removed from index 1 -->2
[18]|DEBUG|()
```

Declare list of sObjects- Example

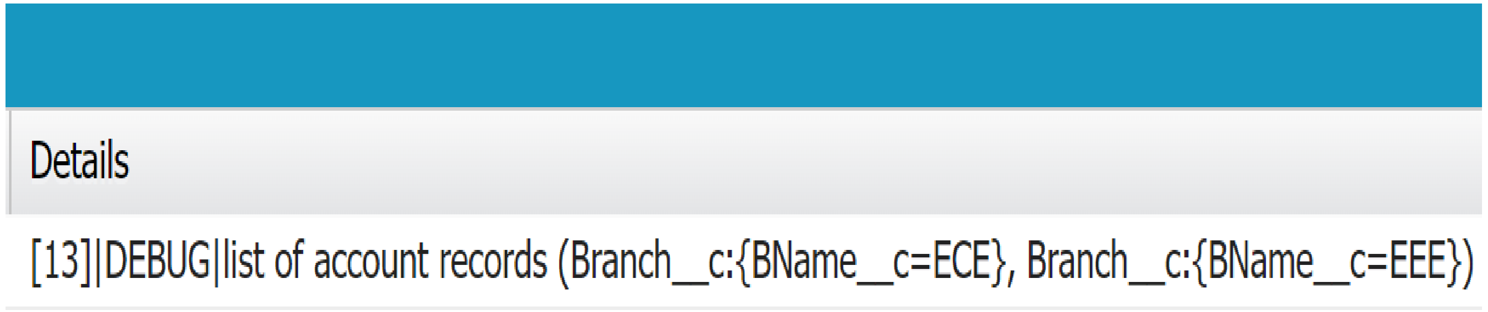
```
List<branch__c> accList = new List<branch__c>();

branch__c a1 = new branch__c ();
a1. BName__c = 'ECE';
accList.add(a1);

branch__c a2 = new branch__c ();
a1. BName__c = ‘EEE';
accList.add(a2);

system.debug('list of account records ' + accList);
```

OUTPUT



```
Details
[13]|DEBUG|list of account records (Branch__c:{BName__c=ECE}, Branch__c:{BName__c=EEE})
```

Introduction to Apex - Data Types

2. Set:

- A set is an **unordered collection of elements**.
- **Insertion order** is **not preserved** in the set.
- A Set **cannot have duplicate records**. Like Lists, Sets can be nested.
- You **cannot perform DML** with Set.
- You **can't index a Set** like you can a List/Array.
- You can perform typical set operations - such as check if the Set contains an element by value, or

remove an element by value.

Syntax for a Set is as follows:

```
Set<datatype> variablename = new Set<datatype>();
```

Example:

```
Set<string> ProductSet = new Set<string> {'Phenol', 'Benzene', 'H2SO4'};
System.debug('Value of ProductSet'+ProductSet);
```

Set Constructors:

Constructors for Set includes:

1. **Set()** - It helps in creating a new instance of the Set class. It can hold elements of any data type T.

```
Set<String> s = new Set<String>{‘XYZ’};
```

2. **Set(setToCopy)** - It helps in creating a new instance of the Set class by copying the elements of the defined Set. T is the data type of the element sets.

```
Set<String> s = new Set<String>{'XYZ'};
Set<String> s1 = new Set<String>(s);
System.debug(s1);
```

3. **Set(listToCopy)** - It helps in creating a new instance of the Set class by copying the elements in the list. The list can be any data type and the data type of element in the Set is T.

```
list<String> L = new list<String>{'XYZ'};
Set<String> s1 = new Set<String>(L);
System.debug(s1);
```

Some common methods of set in Apex:

S.No	Function	Example

1	**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>{‘XYZ’};
		s.add(‘abc’);
		s.add(‘ABC’);
		s.add(‘abc’);
		System.debug(s); //(ABC, XYZ, abc)

2	**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, XYZ, abc, def)

3	**clear()**
	Removes all the elements.
		s.clear();
		s.addAll(l);

4	**clone()**
	Makes duplicate of a set.
		set<String> s2 = s.clone();
		System.debug(s2); //(ABC, XYZ, abc, def)

5	**contains(elm)** Returns true if the set contains the specified element.	Boolean result = s.contains(‘abc’); System.debug(result); // true

S.No	Function	Example
6	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
7	size() Returns the size of set.	System.debug(s.size()); // 2

8	retainAll(list)
	Retains only the elements in this set that are contained in the specified list and removes all other elements. This method results in the intersection of the list and the set.
		s.add('ghi');
		System.debug(s); //(ABC,XYZ,abc,def,ghi)
		s.retainAll(l);
		System.debug(s); //('abc','def')

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

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

Introduction to Apex - Data Types

Example:

```
Set<String> ProductSet=new Set<String>();
ProductSet.add('HCL');
ProductSet.add('H2O');
ProductSet.add('H2SO4');
System.debug('Set with New Value: '+ProductSet);
ProductSet.remove('HCL');
System.debug('Set with removed value: '+ProductSet);
 System.debug (ProductSet.contains('HCL'));
System.debug (ProductSet. size());
System.debug('Value of Set with all values '+ProductSet);
```

Introduction to Apex - Data Types

3. Map:

- It is a **key value pair** which **contains the unique key** for each value.
- Both **key and value can be of any data type**.
- Map values may be **unordered** and hence we should not rely on the order in which the values are stored and try to **access the map always using keys**.
- Map value can be null.
- Map **keys when declared as String are case-sensitive.**

  i.e., ABC and abc will be considered as different keys and treated as unique.

```
Map<string, string> map1 = new Map<string, string> {'1000'=>'HCL', '1001'=>'H2SO4'};
```

Map Initialization

1. Initialize a Map with **String Keys and Integer Values:**

```
Map<String, Integer> myMap = new Map<String, Integer> {
    ‘One’ => 1,
    ‘Two’ => 2,
    ‘Three’ => 3
};
```

2. Using Constructor You can initialize a map using the constructor and then add entries:

```
Map<String, Integer> myMap = new Map<String,
Integer>();
myMap.put('Key1', 1);
myMap.put('Key2', 2);
```

3. Initialize a Map with **String Keys and Custom Object Values:**

```
MyCustomObject__c obj1 = new MyCustomObject__c(Name = ‘Object1’);
MyCustomObject__c obj2 = new MyCustomObject__c(Name = ‘Object2’);
Map<String, MyCustomObject__c> myMap = new Map<String,
MyCustomObject__c>{‘Key1’ => obj1,  ‘Key2’ => obj2  };
```

S.No	Function	Example	Output
1	**put(‘key’, value)** **to add entries to the map**	Map<String, Integer> myMap = new Map<String, Integer>(); myMap.put(‘a', 1); myMap.put(‘b', 2);	**{a=1,b=2}**
2	**putAll(fromMap)** Copies all of the mappings from the specified map to the original map.	Map<String, Integer> map1 = new Map<String, Integer>{‘a’ => 1}; Map<String, Integer> map2 = new Map<String, Integer>{'a' => 1 ‘b’ => 2}; map1.putAll(map2);	**{a=1,b=2}**
3	**get(key)** method to retrieve a value by its key:	Integer value = myMap.get('a'); **System.debug(value);**	**1**
4	**Remove(key)** method to remove an entry by its key:	myMap.remove('Key2'); **System.debug(myMap);**	**{a=1}**
5	**containsKey (key)** method to check if a key exists in the map:	Boolean hasKey = myMap.containsKey('Key1'); **System.debug(haskey);**	**True**

S.No	Function	Example	Output

6	**keySet()**
	Returns a set that contains all of the keys in the map.
		Map<String, Integer> myMap = new Map<String, Integer>
		{'a' => 1, 'b' => 2};
		Set<String> keys = myMap.keySet();
		system.debug(keys);
			{a,b}

7	**values()**
	Returns a list that contains all the values in the map.
		Map<String, Integer> myMap = new Map<String, Integer>
		{'a' => 1, 'b' => 2};
		List<Integer> values = myMap.values();
		system.debug(values);
			(1,2)

8	**isEmpty()**
	Returns true if the map has zero key-value pairs.
		Map<String, Integer> myMap = new Map<String, Integer>
		{'a' => 1, 'b' => 2};
		System.debug(myMap.isEmpty());
			False

9	**clear()**
	Removes all of the key-value mappings from the map.
		Map<String, Integer> myMap = new Map<String, Integer>
		{'a' => 1, ‘b' => 2};
		myMap.clear()
		System.debug(myMap);
			{ }

Introduction to Apex - Data Types

Example 1:

```
Map<string, string> map1 = new Map<string, string> {'1000'=>'HCL', '1001'=>'H2SO4'};

System.debug('Value of Map1: '+map1);

system.debug(map1.get(‘1000’));

map1.put('1000','H20');

System.debug(map1.get('1000'));
```

Example 2:

```
Map<integer, string> map1 = new Map<integer, string> {1000=>'HCL', 1001=>'H2SO4'};

system.debug(map1.get(1000));
```

Iterating Through a Map using for loop

```
Map<String, Integer> myMap = new Map<String, Integer>{ 'a' => 1, 'b' => 2};
for (string key : myMap.keySet())
{
    integer value = myMap.get(key);
    System.debug('Key: ' + key + ', Value: ' + value);
}
```

Map with one key and multiple values

```
Map<String,List<String>> Countries = new  Map<String,List<String>>();
List<String> india = new List<String>();
india.add('Goa');
india.add('Punjab');
Countries.put('India-key',india);
System.debug(Countries.get('India-key'));
```

Introduction to Apex - Data Types

Example:

```
Map<string, string> map1 = new Map<string, string>();
Set<String> SetOfKeys;
map1.put('1002', 'Acetone');
map1.put('1003', 'Ketone');
System.debug('If output is true then Map contains the key and output is:’
                                                        +map1.containsKey('1002'));
String value = map1.get('1002');
System.debug('Value at the Specified key using get function: '+value);
SetOfKeys = map1.keySet();
System.debug('Value of Set with Keys '+SetOfKeys);
```

List	**Set**	**Map**
A list is an ordered collection	*A set is an unordered collection*	*A map is a collection of key-value pairs*
Can contain Duplicate	Do not contain any duplicate	Each unique key maps to a single value. Value can be duplicate

Introduction to Apex - Data Types

Enum:

- An Enum, short for enumeration, in Salesforce Apex is **a user defined data type to define a set of named constants**.
- Programmers use Enum to create a collection of related values that can be used to make code more readable and manageable.
- This ensures that the values are consistent across the application.
- For instance, you might use an Enum to represent the days of the week, months in a year, or the stages in a sales process.

Introduction to Apex - Data Types

To understand an Enum in Apex, consider these points:

•Enum contain fixed constant values, which makes code less prone to errors caused by using undefined values.

•These values are stored as named constants, making code easier to read and maintain.

•Enum provide a way to group and manage related sets of constants in an organized manner.

How to Define an Enum in Apex?

To define an enum, use the **enum** keyword in your declaration and use **curly braces** to demarcate the list of possible values.

Introduction to Apex - Data Types

The following code creates an enum called Season:

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

By creating the enum **Season**, you have also created a **new data type** called **Season**.

Enum Methods:

All Apex enums, whether user-defined enums or built-in enums, have the following common methods that takes no arguments.

values : This method returns the values of the Enum as a list of the same Enum type.

name : Returns the name of the Enum item as a String.

ordinal : Returns the position of the item, as an Integer, in the list of Enum values starting with zero.

valueOf() : Static method to convert a string to an Enum constant.

Note: Enum values **cannot have user-defined methods** added to them.

Introduction to Apex - Data Types

Example:

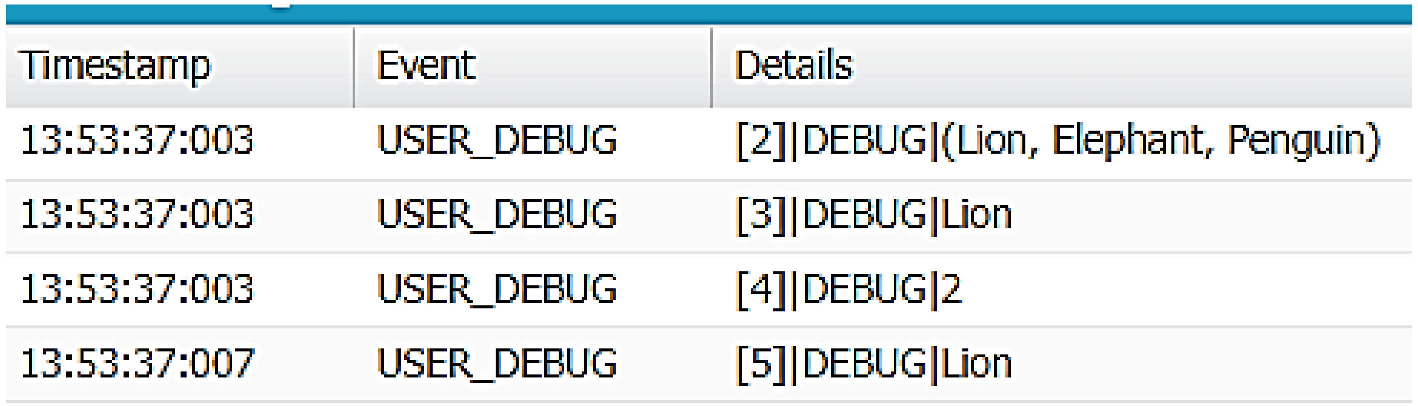
```
public enum Animal { Lion, Elephant, Penguin }

System.debug(Animal.values());

System.debug(Animal.Lion.name());

System.debug(Animal.Penguin.ordinal());

System.debug(Animal.valueOf('Lion'));
```



Timestamp	Event	Details
13:53:37:003	USER_DEBUG	[2]|DEBUG|(Lion, Elephant, Penguin)
13:53:37:003	USER_DEBUG	[3]|DEBUG|Lion
13:53:37:003	USER_DEBUG	[4]|DEBUG|2
13:53:37:007	USER_DEBUG	[5]|DEBUG|Lion

Apex Classes

Create an Apex Class:

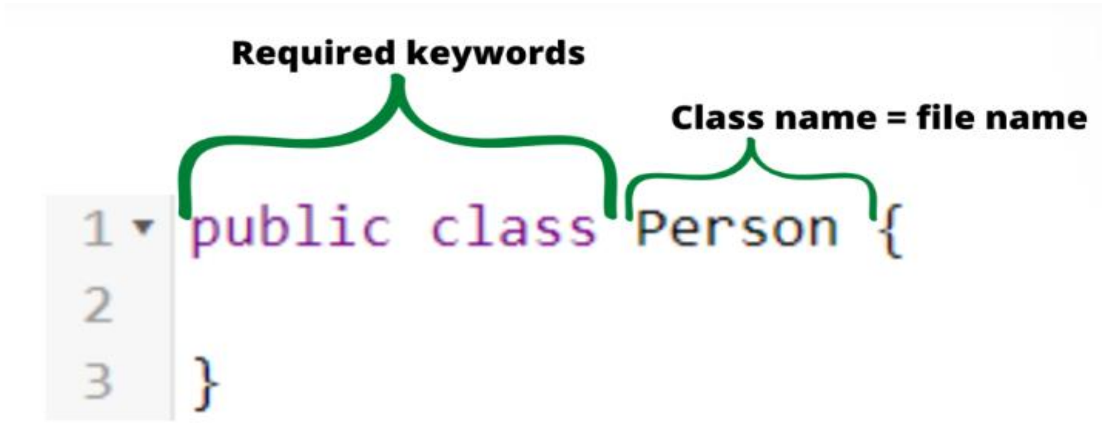
The Developer Console is where you write and test your code in Salesforce.

Steps to create an Apex class:

- Click the **gear** **icon** to access Setup in Lightning Experience and select **Developer Console**.
- From the **File menu**, select **New | Apex Class**.
- For the class name, enter **Person** and then click **OK**.

Syntax:

```
AccessModifier  class  ClassName {
data members;
methods;                    }
```



```
public class Person {

}
```

Apex Classes

Declaring a method

The syntax to declare an Apex method is:

```
public class MyClass {
public ReturnType methodName()
{
            return ReturnValue; }
}
```

Example

```
public class Person
 {
     public String name;
     public Integer age;
   public void sayName()
   {          System.debug('My name is ' + name); }
public void sayAge()
 {            System.debug('I am ' + age);          }    }
```

Here,

- **public**: This means that you can use your method in any application within your org.
- **methodName**: An identifier that you will use to call your method and execute logic within the method.
- **ReturnType**: Any data type that your method returns. See the return statement section.
- **ReturnValue**: Value that is returned from the method back to the calling function.

Apex Classes

Run the Code:

- An **anonymous block** is **Apex code** that **does not get stored**, but can be **compiled and executed** on demand right from the Developer Console.
- This is a great way to **test your Apex Classes or run sample code**.

1. In the **Developer Console,** select **Debug | Open Execute Anonymous Window.**

2. In the **Enter Apex Code window,** enter the code for execution

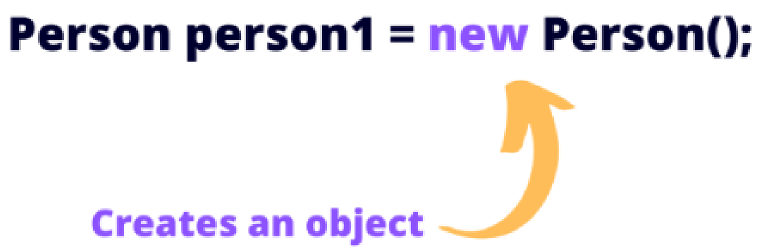
3. At the bottom right**,** click **Execute.**

Creating Object for a class and accessing variables and method inside the class

Calling methods

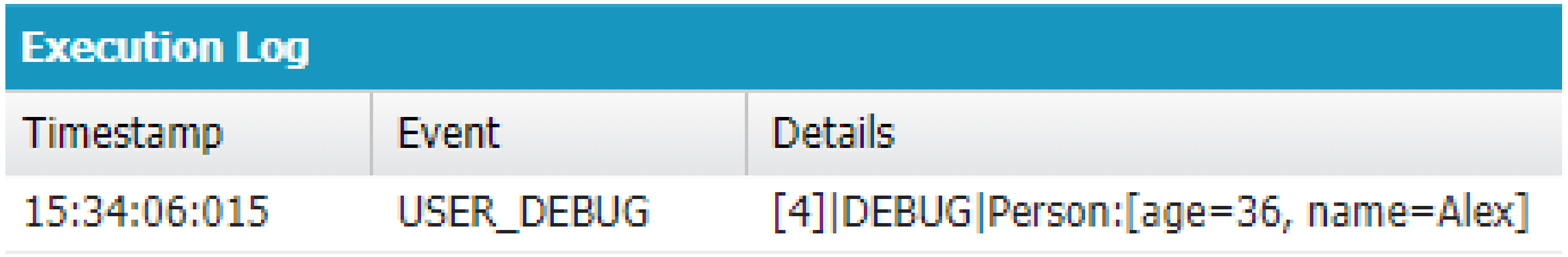
Navigate to Developer Console → Debug → Execute Anonymous Window

```
Person person1 = new Person();
person1.name = 'Alex';
person1.age = 36;
person1.sayName();
person1.sayAge();
```



With The "new" keyword the Apex engine will **create an Apex object**. Apex will then immediately allocate an actual physical space somewhere on the server for the object

OUTPUT



Execution Log

Timestamp	Event	Details
15:34:06:015	USER_DEBUG	[4]|DEBUG|Person:[age=36, name=Alex]

Apex Classes

Constructor in Apex:

- The constructor is a special method .
- The **Method name will be the same as a class**.
- Access specifier will be public.
- This method **will be invoked only once that is at the time of creating an object.**
- This is used to **instantiate the data** members of the class.
- Constructor **will not have a return type**.

There are 3 types of constructors:

1. **Default Constructor**

2. Non-parameterized Constructor

3. Parameterized Constructor

Apex Classes

1. Default Constructor
If an Apex Class doesn't contain any constructor then Apex compiler by default creates a dummy constructor on the name of the class when we create an object for the class.
The default constructor literally does nothing!

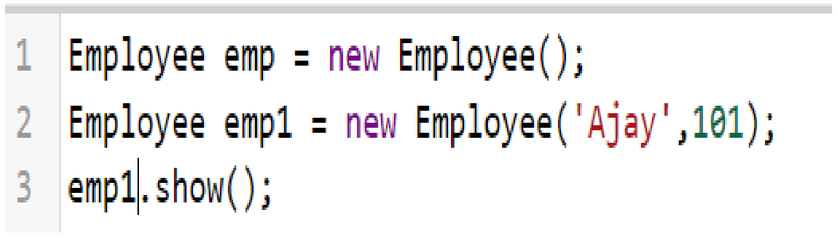
2. Non-parameterized Constructor
It is a constructor that doesn't have any parameters is called Non-parameterized constructor.
Parameters are nothing but the values we are passing inside a constructor.

3. Parameterized Constructor
It is a constructor that has parameters.
That means here we will take input from the user and then map it with our variable.

Apex Classes

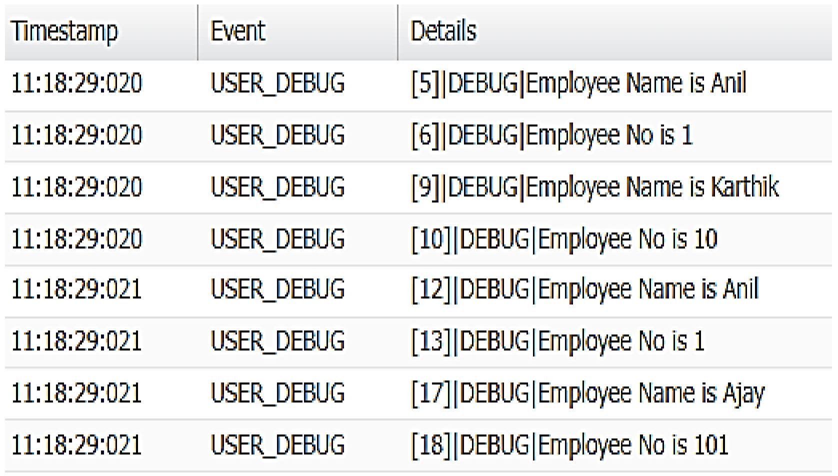
```
public class Employee                                   //Class
 {   String EmployeeName='Anil';
     Integer EmployeeNo=01;
public Employee( )                          //Non-Parameterized Constructor
  {   System.debug('Employee Name is '+ EmployeeName);
      System.debug('Employee No is '+ EmployeeNo);
      EmployeeName = 'Karthik';
      EmployeeNo = 10;
      System.debug('Employee Name is '+ EmployeeName);
     System.debug('Employee No is '+ EmployeeNo);   }
public Employee(string Name, integer num)     //Parameterized Constructor
  {   System.debug('Employee Name is '+ EmployeeName);
      System.debug('Employee No is '+ EmployeeNo);
      EmployeeName = Name;
      EmployeeNo = num; }
public void show()                                      //Method
 {   System.debug('Employee Name is '+ EmployeeName);
     System.debug('Employee No is '+ EmployeeNo);  }     }
```

Execute Anonymous Window



```
Employee emp = new Employee();
Employee emp1 = new Employee('Ajay',101);
emp1.show();
```

OUTPUT

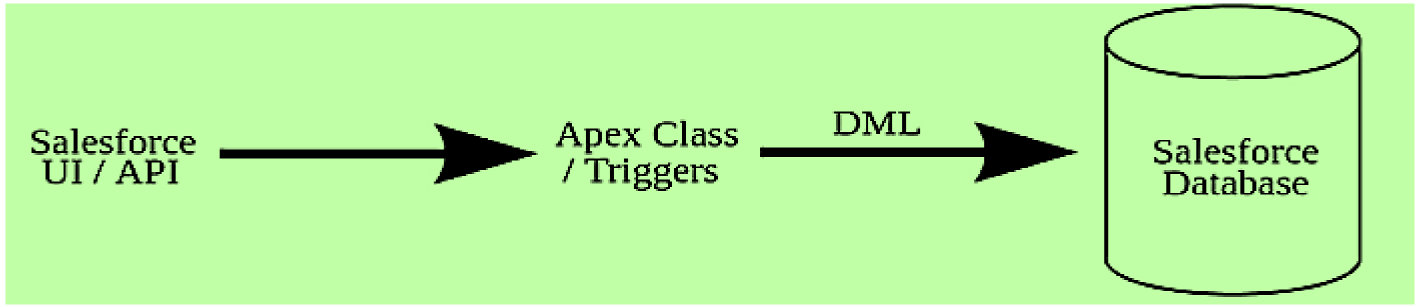


Timestamp	Event	Details
11:18:29:020	USER_DEBUG	[5]|DEBUG|Employee Name is Anil
11:18:29:020	USER_DEBUG	[6]|DEBUG|Employee No is 1
11:18:29:020	USER_DEBUG	[9]|DEBUG|Employee Name is Karthik
11:18:29:020	USER_DEBUG	[10]|DEBUG|Employee No is 10
11:18:29:021	USER_DEBUG	[12]|DEBUG|Employee Name is Anil
11:18:29:021	USER_DEBUG	[13]|DEBUG|Employee No is 1
11:18:29:021	USER_DEBUG	[17]|DEBUG|Employee Name is Ajay
11:18:29:021	USER_DEBUG	[18]|DEBUG|Employee No is 101

Database structure

Manipulate Records with DML:

- Because Apex is a **data-focused language** and is saved on the Lightning Platform, **it has direct access to your data in Salesforce**.
- By **calling DML statements**, you can quickly **perform operations on your Salesforce records**.
- DML allows you to **perform operations on a single sObject record or in bulk on a list of sObjects records** at once.
- 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.

Database structure

DML Statements:

- Insert
- Delete
- Update
- Undelete
- Upsert
- Merge

The upsert and merge statements are particular to Salesforce

Database structure

Insert Branch and Student Record:

Insert operation is used to create new records in Database. You can create records of any Standard or Custom object using the Insert DML statement.

Adding New Branch:

```
    Branch__c br = new Branch__c(Branch_Name__c='Data Science');
    insert br;                                    // Insert the branch by using DML

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

Database structure

Adding New Branch and assigning Student:

```
Branch__c br = new Branch__c(Branch_Name__c='Data Science Department');
insert br;                                        // Insert the branch by using DML
Student__c st = new Student__c(Student_Name__c = 'Student', Branch__c=br.Id);
 Insert st;
```

Adding existing Branch to the Student Record :

```
List<Branch__c> l1=[Select Id, Branch_Name__C from Branch__c where Branch_Name__C='CSE'];
Student__c st = new Student__c(Student_Name__c = 'StudentXY', Branch__c=l1[0].Id);
Insert st;
```

Database structure

Update Branch record:

The update DML operation modifies one or more existing sObject records. update is analogous to the UPDATE statement in SQL.

```
Branch__c branch=[Select Branch_Name__c from Branch__c where
                                        Branch_Name__c='Information Technology'];
branch.Branch_Name__c='IT';
update branch;
```

Delete Branch record:

The delete DML operation deletes one or more existing sObject records.

```
Branch__c branch=[Select Id, Branch_Name__c from Branch__c where Branch_Name__c='CS'];
delete branch;
```

Database structure

Undelete Branch record:

You can undelete the record which has been deleted and is present in Recycle bin. All the relationships which the deleted record has, will also be restored.

The **ALL ROWS** keyword queries all rows for both top level and aggregate relationships, including deleted records and archived activities.

```
    Branch__c branch=[Select Id, Branch_Name__c from Branch__c where Branch_Name__c='CS'
                                                                                ALL ROWS];
    undelete branch;
```

Database structure

Upsert:

- 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 and can save you time as you don’t have to determine which records exist first.
- 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.
    - If the key isn’t matched, then a new object record is created.
    - If the key is matched once, then the existing object record is updated.
    - If the key is matched multiple times, then an error is generated and the object record is not inserted or updated.

Database structure

Upsert Syntax:

- This example shows how upsert **inserts a new branch and updates an existing branch name in branch record in one call.**
- This upsert call inserts a new Branch ‘Data Science’ and updates the existing Branch Name ‘Information Technology to ‘IT’.

```
Branch__c br1 = new Branch__c (Branch_Name__c ='Data Science');          //new branch created
Branch__c br2=[Select Branch_Name__c from Branch__c where Branch_Name__c ='Information
                                                                              Technology'];
br2. Branch_Name__c ='IT';                                                //change branch name
List<Branch__c> bran = new List<Branch__c> { br1, br2 };          // List to hold the new contacts to upsert
upsert bran;
```

Database structure

Merge:

- When you have duplicate lead, contact, case, or account records in the database, cleaning up your data and consolidating the records might be a good idea.
- The merge operation merges up to three records into one of the records, deletes the others, and reparents any related records.

```
List < Account > accList = [SELECT Name FROM Account LIMIT 3];
Account a = accList[0];
Account b = accList[1];
Account c = accList[2];
List < Account > mergeList = new List < Account > ();
mergeList.add(accList[1]);
mergeList.add(accList[2]);
merge a b;                              // for 2 records
merge a mergeList;                      // for 3 records
```

Note:

- Merge only works with Accounts, Leads and Contacts.
- You can merge 3 records at a time not more than that.

Database structure

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.

This example produces a DmlException because it attempts to insert an account without the required Name field. The exception is caught in the catch block.

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

Database structure

Bulk DML:

- You can perform DML operations either on a single sObject, or in bulk on a list of sObjects.
- DML limit of 150 statements per Apex transaction.
- This limit is in place to ensure fair access to shared resources in the Lightning Platform.
- Performing a DML operation on a list of sObjects counts as one DML statement, not as one statement for each sObject.

Database structure

This example inserts contacts in bulk by inserting a list of contacts in one call. The sample then updates those contacts in bulk too.

```
 List<Contact> conList = new List<Contact> {
   new Contact(FirstName='Joe',LastName='Smith',Department='Finance'),
       new Contact(FirstName='Kathy',LastName='Smith',Department='Technology'),
       new Contact(FirstName='Caroline',LastName='Roth',Department='Finance'),
       new Contact(FirstName='Kim',LastName='Shain',Department='Education')};
 insert conList;                                    // Bulk insert all contacts with one DML call
 List<Contact> listToUpdate = new List<Contact>();  // List to hold the new contacts to update
 for(Contact con : conList) {                       // Iterate through the list and add a title only
   if (con.Department == 'Finance') {               //  if the department is Finance
       con.Title = 'Financial analyst';
       listToUpdate.add(con);                       // Add updated contact sObject to the list.
   } }
update listToUpdate;                                // Bulk update all contacts with one DML call
```

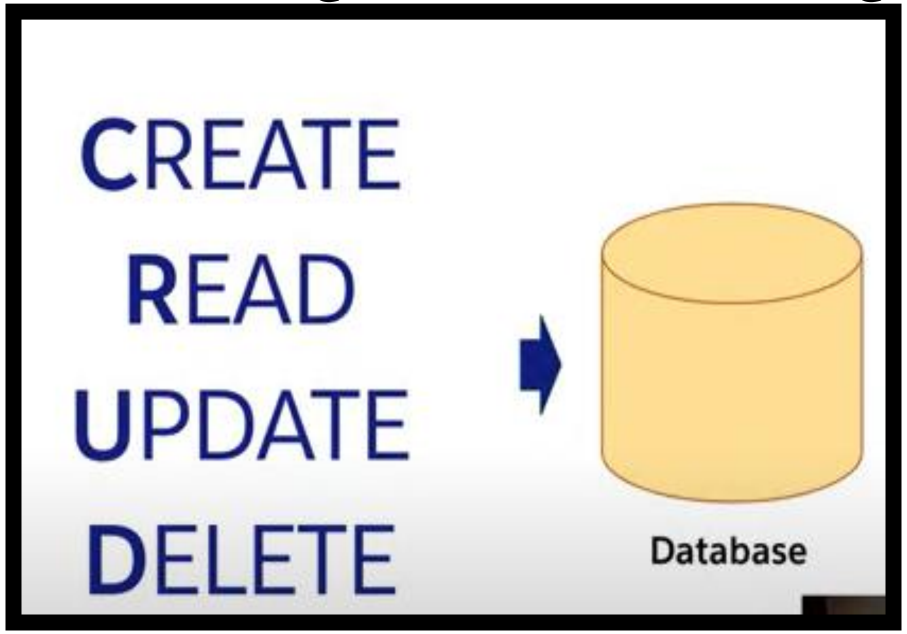
Database structure

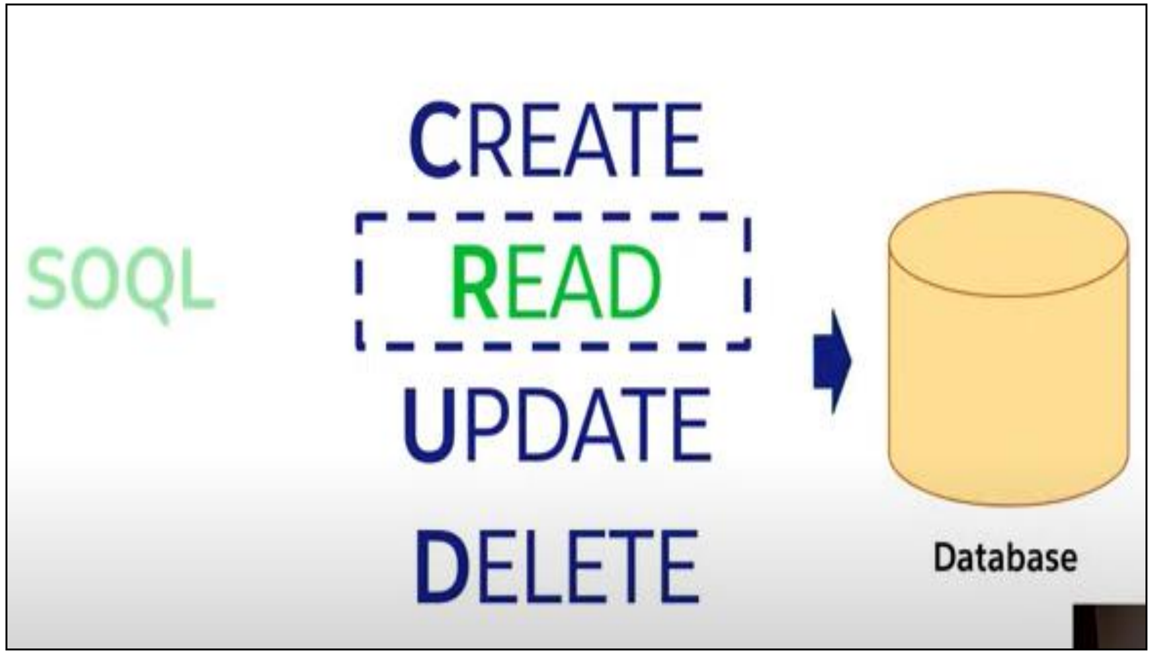
Database Methods

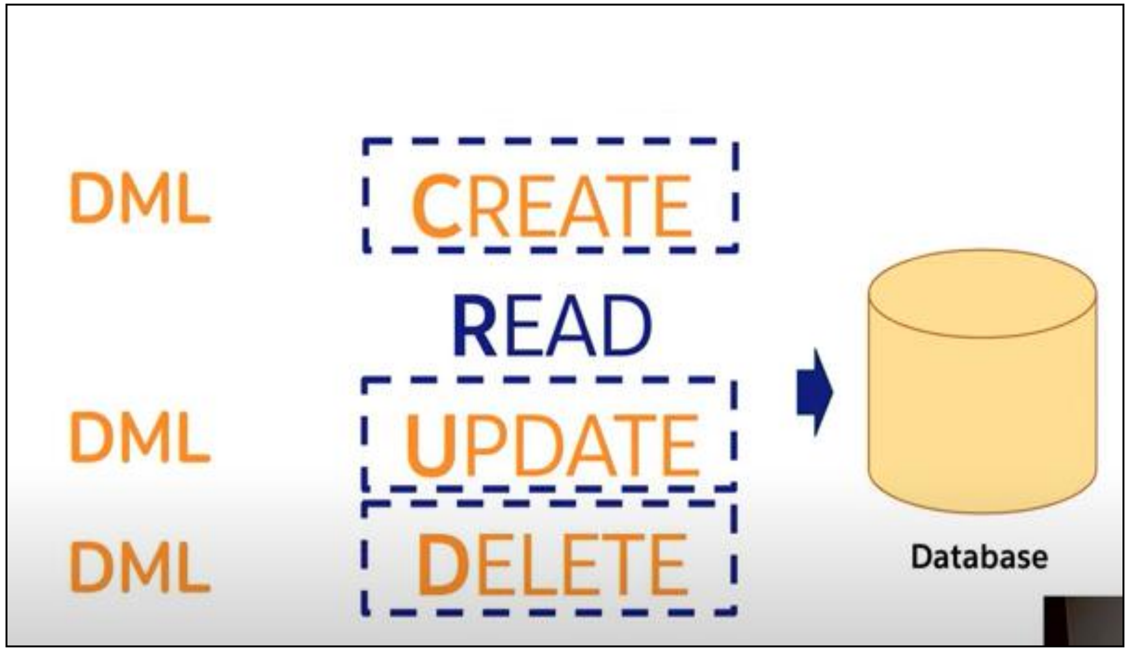
Apex contains the built-in Database class, which provides methods that perform DML operations and mirror the DML statement counterparts.
These Database methods are static and are called on the class name.

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

Execute SOQL and SOSL Queries







Execute SOQL and SOSL Queries

Write SOQL Queries:

- To read a record from Salesforce, you must write a query.
- Salesforce provides the Salesforce Object Query Language, or SOQL in short, that you can **use to read** saved records.
- SOQL is similar to the standard SQL language but is customized for the Lightning Platform.
- Because Apex has direct access to Salesforce records that are stored in the database, you can embed SOQL queries in your Apex code and get results in a straightforward fashion.
- When SOQL is embedded in Apex, it is referred to as **inline SOQL.**

Execute SOQL and SOSL Queries

Use 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 and to **test your SOQL queries** before adding them to your Apex code.
- When you use the Query Editor, you must **supply only the SOQL statement without the Apex code** that surrounds it.

Execute SOQL and SOSL Queries

Basic SOQL Syntax

This is the syntax of a basic SOQL query:

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

The **WHERE** clause is **optional**.

The query has two parts:

- **SELECT fields :** This part lists the fields that you would like to retrieve. The fields are specified after the SELECT keyword in a comma-delimited list. Or you can specify only one field, in which case no comma is necessary.

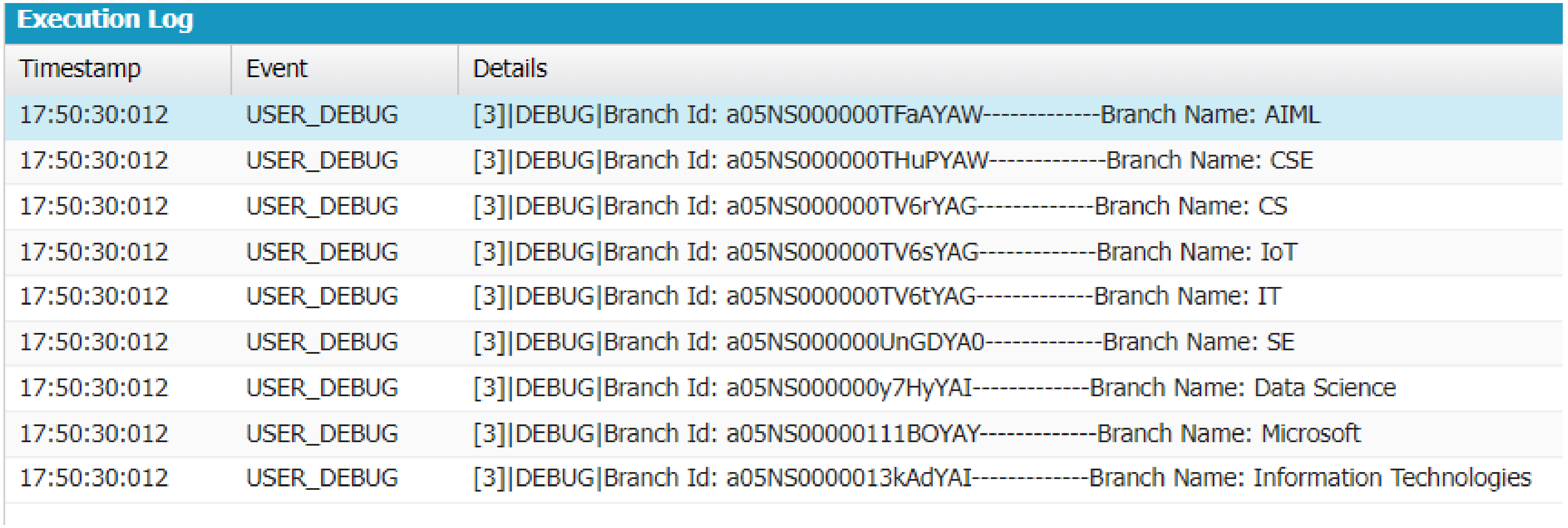
- **FROM ObjectName :** This part specifies the standard or custom object that you want to retrieve. In this example, it's **Account**. For a **custom object** called **Student**, it is **Student__c**.

For example, the following retrieves all account records with two fields, Name and Phone, and returns an array of Account sObjects.

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

Execute SOQL and SOSL Queries

```
public class Utility {
 public static void display() {
     List<Branch__c> acc=[SELECT Id, Branch_Name__c FROM Branch__c];
      for(Branch__c br : acc)
     {   System.debug('Branch Id:'+br.Id+'----------'+'Branch Name: '+ br.Branch_Name__c);
         }
   }
}
```

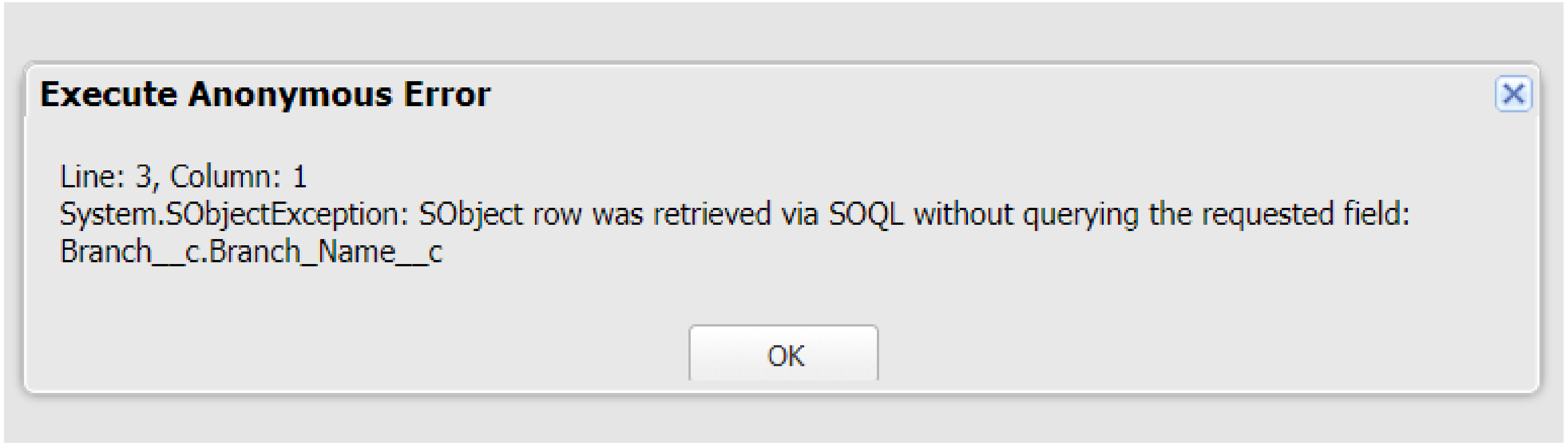


Execution Log

Timestamp	Event	Details
17:50:30:012	USER_DEBUG	[3]|DEBUG|Branch Id: a05NS000000TFaAYAW------------Branch Name: AIML
17:50:30:012	USER_DEBUG	[3]|DEBUG|Branch Id: a05NS000000THuPYAW------------Branch Name: CSE
17:50:30:012	USER_DEBUG	[3]|DEBUG|Branch Id: a05NS000000TV6rYAG------------Branch Name: CS
17:50:30:012	USER_DEBUG	[3]|DEBUG|Branch Id: a05NS000000TV6sYAG------------Branch Name: IoT
17:50:30:012	USER_DEBUG	[3]|DEBUG|Branch Id: a05NS000000TV6tYAG------------Branch Name: IT
17:50:30:012	USER_DEBUG	[3]|DEBUG|Branch Id: a05NS000000UnGDYA0------------Branch Name: SE
17:50:30:012	USER_DEBUG	[3]|DEBUG|Branch Id: a05NS000000y7HyYAI------------Branch Name: Data Science
17:50:30:012	USER_DEBUG	[3]|DEBUG|Branch Id: a05NS00000111BOYAY------------Branch Name: Microsoft
17:50:30:012	USER_DEBUG	[3]|DEBUG|Branch Id: a05NS0000013kAdYAI------------Branch Name: Information Technologies

Database structure

```
public class Utility {
 public static void display() {
      List<Branch__c> acc=[SELECT Id FROM Branch__c];
       for(Branch__c br : acc)
      {   System.debug('Branch Id:'+br.Id+'-----------'+'Branch Name: '+ br.Branch_Name__c);
           }
   }
}
```


Execute Anonymous Error

Line: 3, Column: 1
System.SObjectException: SObject row was retrieved via SOQL without querying the requested field: Branch__c.Branch_Name__c

Execute SOQL and SOSL Queries

WHERE - Filter 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.
- The following example retrieves only the accounts whose names are SFDC Computing.
- Note that comparisons on strings are case-insensitive.

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

The **WHERE** clause can contain multiple conditions that are grouped by using logical operators (AND, OR) and parentheses.
For example, this query returns all accounts whose name is SFDC Computing that have more than 25 employees:

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

Execute SOQL and SOSL Queries

ORDER BY - Order Query Results

- When a query executes, it returns records from Salesforce in no particular order, so you can't rely on the order of records in the returned array to be the same each time the query is run.
- You can however choose to sort the returned record set by adding an ORDER BY clause and specifying the field by which the record set should be sorted.

This example sorts all retrieved accounts based on the Name field.

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

The default sort order is in alphabetical order, specified as ASC. 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
```

NOTE:

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

Execute SOQL and SOSL Queries

LIMIT n - Limit 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 to be returned.

Limiting the result set is handy when you don’t care which records are returned, but you just want to work with a subset of records.

For example, this query retrieves the first account that is returned.

Notice that the returned value is one account and not an array when using **LIMIT 1**.

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

Execute SOQL and SOSL Queries

Combine 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.');
System.debug(accts);                                    // Write all account array info
```

Execute SOQL and SOSL Queries

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

Execute SOQL and SOSL Queries

Query 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 use a different method for retrieving records—records are retrieved using efficient chunking with calls to the query and queryMore methods of the SOAP API.
- By using SOQL for loops, you can avoid hitting the heap size limit.
- SOQL for loops iterate over all of the sObject records returned by a SOQL query. The syntax of a SOQL forloop 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. It is preferable to use the sObject list format of the SOQL for loop as the loop executes once for each batch of 200 sObjects.

Execute SOQL and SOSL Queries

Example:

```
insert new Account[]{new Account(Name = 'for loop 1'),
new Account(Name = 'for loop 2'),
new Account(Name = 'for loop 3')};
Integer i=0;
for (Account[] tmp : [SELECT Id FROM Account WHERE Name LIKE 'for loop '])
{
        System.debug(tmp[i]);
         i++;
}
```

Execute SOQL and SOSL Queries

Write SOSL Queries:

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. SOSL is similar to Apache Lucene.

Adding SOSL queries to Apex is simple—you can embed SOSL queries directly in your Apex code. When SOSL is embedded in Apex, it is referred to as ***inline SOSL.***

Basic SOSL Syntax:

SOSL allows you to specify the following search criteria:

- Text expression (single word or a phrase) to search for
- Scope of fields to search
- List of objects and fields to retrieve
- Conditions for selecting rows in the source objects

Execute SOQL and SOSL Queries

This is the syntax of a basic SOSL query in Apex:

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

Remember that in the Query Editor and API, the syntax is slightly different:

```
FIND {SearchQuery} [IN SearchGroup] [RETURNING ObjectsAndFields]
```

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>> searchList = [FIND 'SFDC' IN ALL FIELDS RETURNING Account(Name),
Contact(FirstName,LastName)];
```

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.

Execute SOQL and SOSL Queries

Text searches are case-insensitive. For example, searching for Customer, customer, or CUSTOMER all return the same results.

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.

Execute SOQL and SOSL Queries

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.

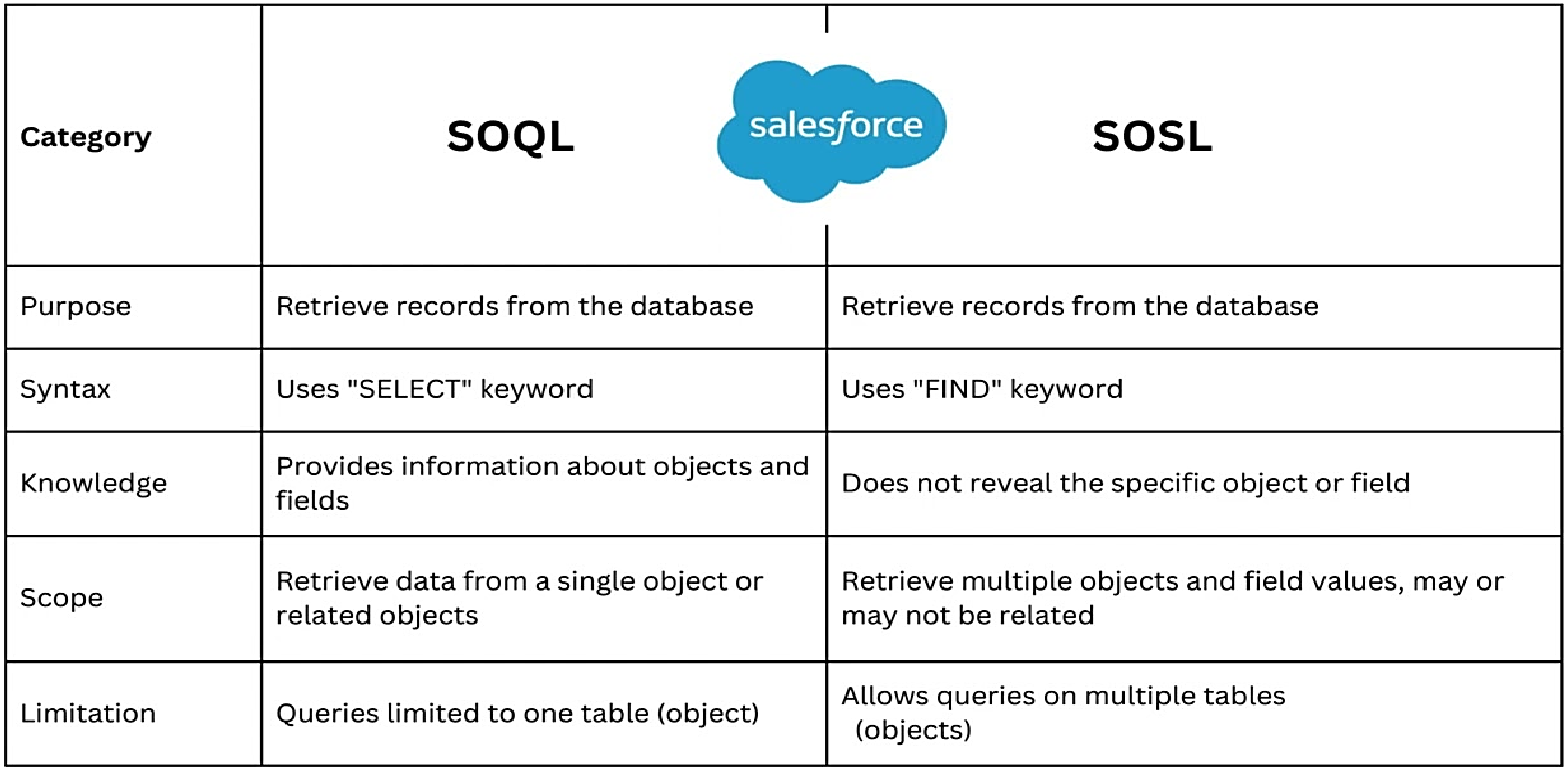
SOSL Apex Example:

The SOSL query references this local variable **soslFindClause** by preceding it with a colon, also called ***binding*.**

```
String soslFindClause = 'Wingo OR SFDC';
List<List<sObject>> searchList = [FIND :soslFindClause IN ALL FIELDS RETURNING Account(Name),
                                        Contact(FirstName,LastName,Department)];
Account[] searchAccounts = (Account[])searchList[0];
Contact[] searchContacts = (Contact[])searchList[1];
System.debug('Found the following accounts.');
for (Account a : searchAccounts) {    System.debug(a.Name);    }
System.debug('Found the following contacts.');
for (Contact c : searchContacts) {    System.debug(c.LastName + ', ' + c.FirstName);   }
```

Execute SOQL and SOSL Queries

Differences and Similarities Between SOQL and SOSL



Category	SOQL	SOSL
Purpose	Retrieve records from the database	Retrieve records from the database
Syntax	Uses "SELECT" keyword	Uses "FIND" keyword
Knowledge	Provides information about objects and fields	Does not reveal the specific object or field
Scope	Retrieve data from a single object or related objects	Retrieve multiple objects and field values, may or may not be related
Limitation	Queries limited to one table (object)	Allows queries on multiple tables (objects)

Apex Triggers - Introduction

- Apex triggers enable you to **perform custom actions before or after events** to records in Salesforce, such as **insertions, updates, or deletions.**
- Just like database systems support triggers, Apex provides **trigger support for managing records**.
- Typically, you use triggers to **perform operations based on specific conditions**, to **modify related records** or **restrict certain operations** from happening.
- You can use triggers to **execute SOQL and DML** or calling custom Apex methods.
- Use triggers to perform tasks that can't be done by using the point-and-click tools in the Salesforce user interface.
- **For example**, if validating a field value or updating a field on a record, use validation rules and flows. Use Apex triggers if performance and scale is important, if your logic is too complex for the point-and-click tools, or if you're executing CPU-intensive operations.
- Triggers are **active by default** when created. Salesforce **automatically fires active triggers** when the specified database events occur.

Apex Triggers - Introduction

Trigger Syntax:

A trigger definition starts with the trigger keyword. It is then followed by the name of the trigger, the Salesforce object that the trigger is associated with, and the conditions under which it fires.

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

Types of Triggers:

There are two types of triggers,

1. Before triggers are used to update or validate record values before they're saved to the database.

2. After triggers are used to access field values that are set by the system (such as a record's Id or LastModifiedDate field), and to affect changes in other records. The records that fire the after trigger are read-only.

Apex Triggers - Introduction

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

Apex Triggers - Introduction

Trigger Example:

This simple trigger fires before you insert an account and writes a message to the debug log.

1. In the Developer Console, click File | New | Apex Trigger.
2. Enter HelloWorldTrigger for the trigger name, and then select Account for the sObject. Click Submit.
3. Replace the default code with the following.

```
trigger HelloWorldTrigger on Account (before insert) {
    System.debug('Hello World!');     }
```

4. To save, press Ctrl+S.
5. To test the trigger, create an account.

    a. Click Debug | Open Execute Anonymous Window.

    b. In the new window, add the following and then click Execute.

```
Account a = new Account(Name='Test Trigger');
insert a;
```

6. In the debug log, find the Hello World! statement. The log also shows that the trigger has been executed.

Apex Triggers - Context Variables

isExecuting	Returns true if the current context for the Apex code is a trigger.
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.
new	Returns a list of the new versions of the sObject records available in insert, update, and undelete triggers, and the records can only be modified in before triggers.
newMap	A map of IDs to the new versions of the sObject records available in before update, after insert, after update, and after undelete triggers.
old	Returns a list of the old versions of the sObject records available only in update and delete triggers.
oldMap	A map of IDs to the old versions of the sObject records available only in update and delete triggers.
operationType	Returns an enum of type System.TriggerOperation corresponding to the current operation. Possible values of the System.TriggerOperation enum are: BEFORE_INSERT, BEFORE_UPDATE, BEFORE_DELETE, AFTER_INSERT, AFTER_UPDATE, AFTER_DELETE, and AFTER_UNDELETE.
size	The total number of records in a trigger invocation, both old and new.

Apex Triggers - Introduction

Using Context Variables:

- To access the records that caused the trigger to fire, use context variables.

For example,

- Triggers can fire when one record is inserted, or when many records are inserted in bulk via the API or Apex.
- You can iterate over Trigger.new to get each individual sObject.

This example iterates over each account in a for loop and updates the Description field for each.

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

Apex Triggers - Introduction

Some other **context variables return a Boolean value** to indicate whether the trigger was fired due to an update or some other event. These variables are useful when a trigger combines multiple events.

For example:

```
trigger ContextExampleTrigger on Account (before insert, after insert, after delete) {
  if (Trigger.isInsert) {
    if (Trigger.isBefore) {
                                                        // Process before insert
    } else if (Trigger.isAfter) {
                                                         // Process after insert
    }   }
  else if (Trigger.isDelete) {
                                                        // Process after delete
  }    }
```

Apex Triggers - Introduction

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.
- It also allows you to use object-oriented programming.

```
trigger ClassmethodTrigger on Account (before insert) {
AccountTrigger.beforeInsert(Trigger.new);  }


public class AccountTrigger{
public void beforeInsert(List<Account> newList)  {
    for(Account a : Trigger.new) {
        a.Description = 'New description';    }    }        }
```

Apex Triggers - Introduction

Before Insert: **Apex Trigger**

```
trigger TeacherRecord on Teacher__c (before insert) {
  if (Trigger.isInsert) {
    if (Trigger.isBefore) {
      TeacherTrigger.beforeInsert(Trigger.new);     }   }   }
```

Class for Trigger:

```
public class TeacherTrigger {
public static void beforeInsert(List<Teacher__c> newList)  {
   for(Teacher__c a: newList) {
if(a.Teacher_Name__c=='Teacher5')
      {          a.Experience__c = 5;   }
else    {   a.Experience__c = 0; }    }   } }
```

Apex Triggers - Introduction

After Insert: **Apex Trigger**

```
        trigger TeacherRecord on Teacher__c (after insert) {
                if (Trigger.isInsert) {
                    if (Trigger.isAfter) {
                        TeacherTrigger.afterInsert(Trigger.new);    }   }  }
```

Class for Trigger:

```
            public class TeacherTrigger {
            public static void afterInsert(List<Teacher__c> newList)  {
            Branch__c br = new Branch__c(Branch_Name__c='New Branch');
                insert br;  }     }
```

Apex Triggers - Introduction

Before Insert vs After Insert: Apex Trigger

```
trigger TeacherRecord on Teacher__c (before insert, after insert) {
  if (Trigger.isInsert) {
    if (Trigger.isBefore) {
        TeacherTrigger.beforeInsert(Trigger.new);
}
else if (Trigger.isAfter) {
              TeacherTrigger.afterInsert(Trigger.new);
 }   }  }
```

Apex Triggers - Introduction

Before Insert vs After Insert: Class for Trigger

```
public class TeacherTrigger {
public static void beforeInsert(List<Teacher__c> newList)  {
    for(Teacher__c a: newList) {
if(a.Teacher_Name__c=='Teacher5')
        {
            a.Experience__c = 5;   }
else
    {        a.Experience__c = 0; }    }   }
 public static void afterInsert(List<Teacher__c> newList)  {
Branch__c br = new Branch__c(Branch_Name__c='New Branch');
              insert br;  }      }
```

Apex Triggers - Bulk Apex Triggers

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 benefit of bulkifying your code is that bulkified code can process large numbers of records efficiently and run within governor limits on the Lightning Platform.
- The following sections demonstrate the main ways of bulkifying your Apex code in triggers:

  1. operating on all records in the trigger, and
  2. performing SOQL and DML on collections of sObjects instead of single sObjects at a time.

- The SOQL and DML bulk best practices apply to any Apex code, including SOQL and DML in classes.
- The examples given are based on triggers and use the Trigger.new context variable.

Apex Triggers - Bulk Apex Triggers

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.

- The following trigger **(MyTriggerNotBulk)** assumes that only one record caused the trigger to fire.
- This trigger doesn't work on a full record set when multiple records are inserted in the same transaction.

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

Apex Triggers - Bulk Apex Triggers

This example **(MyTriggerBulk)** is a modified version of MyTriggerNotBulk.

It uses a for loop to iterate over all available sObjects.

This loop works if Trigger.new contains one sObject or many sObjects.

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

Apex Triggers - Bulk Apex Triggers

Performing Bulk SOQL:

- SOQL queries can be powerful. You can retrieve related records and check a combination of multiple conditions in one query.
- 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 asynchronous Apex.
- The following trigger (SoqlTriggerNotBulk) shows a SOQL query pattern to avoid.
- 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.

Apex Triggers - Bulk Apex Triggers

```
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
    }
}
```

Apex Triggers - Bulk Apex Triggers

- The following example **(SoqlTriggerBulk)** is a modified version of the previous one and shows a best practice for running SOQL queries.
- The SOQL query does the heavy lifting and is called once outside the main loop.
    - 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.new variable in the WHERE clause—WHERE Id IN :Trigger.new. This WHERE condition filters the accounts to only those records that fired this trigger.
- Combining the two parts in the query results in the records we want in one call: the accounts in this trigger with the related opportunities of each account.

Apex Triggers - Bulk Apex Triggers

```
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
    }   }
```