salesforce
DEVELOPERS

UNIT-2

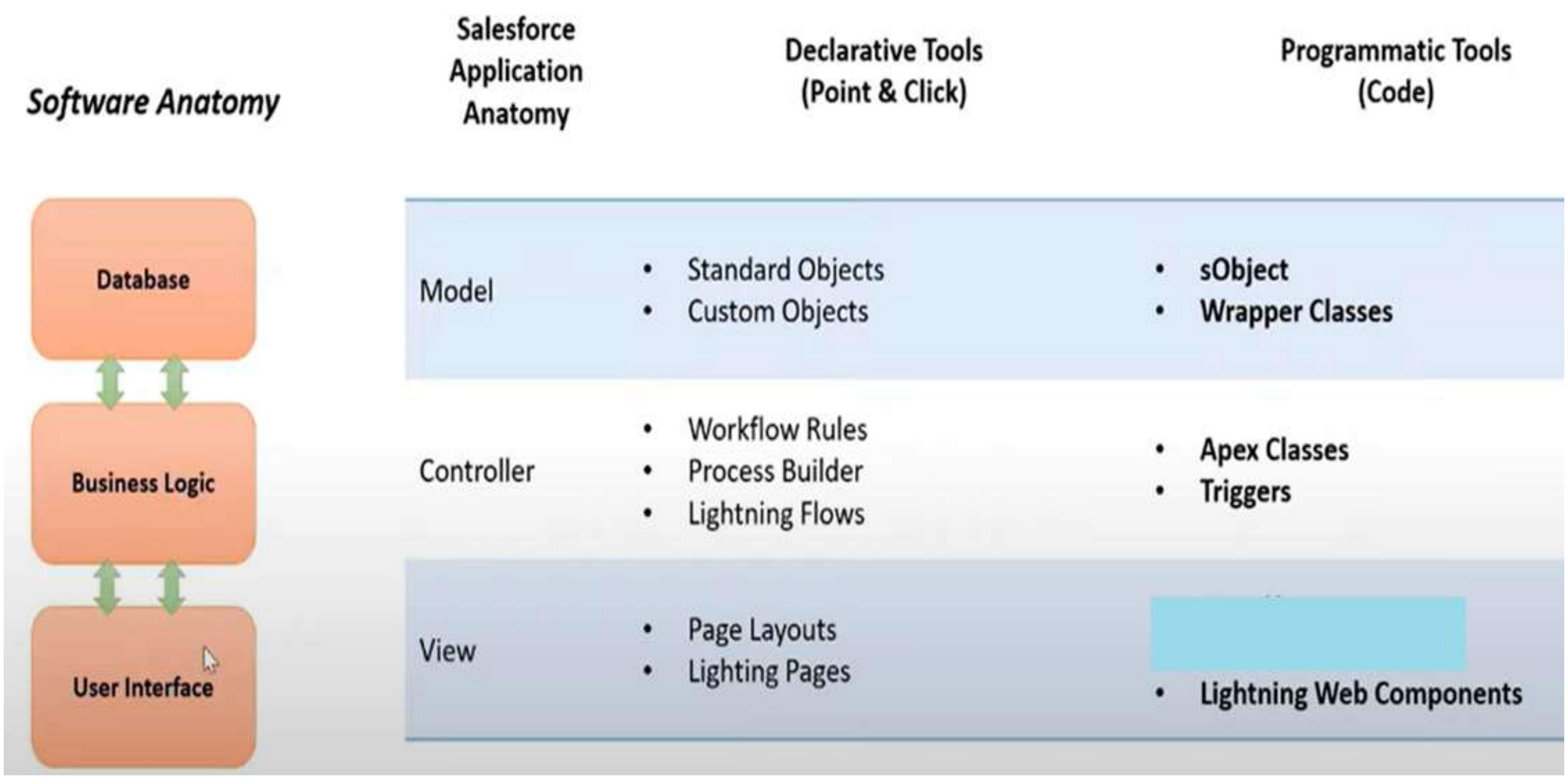
By
Mrs. P.SHYAMALA

UNIT–II

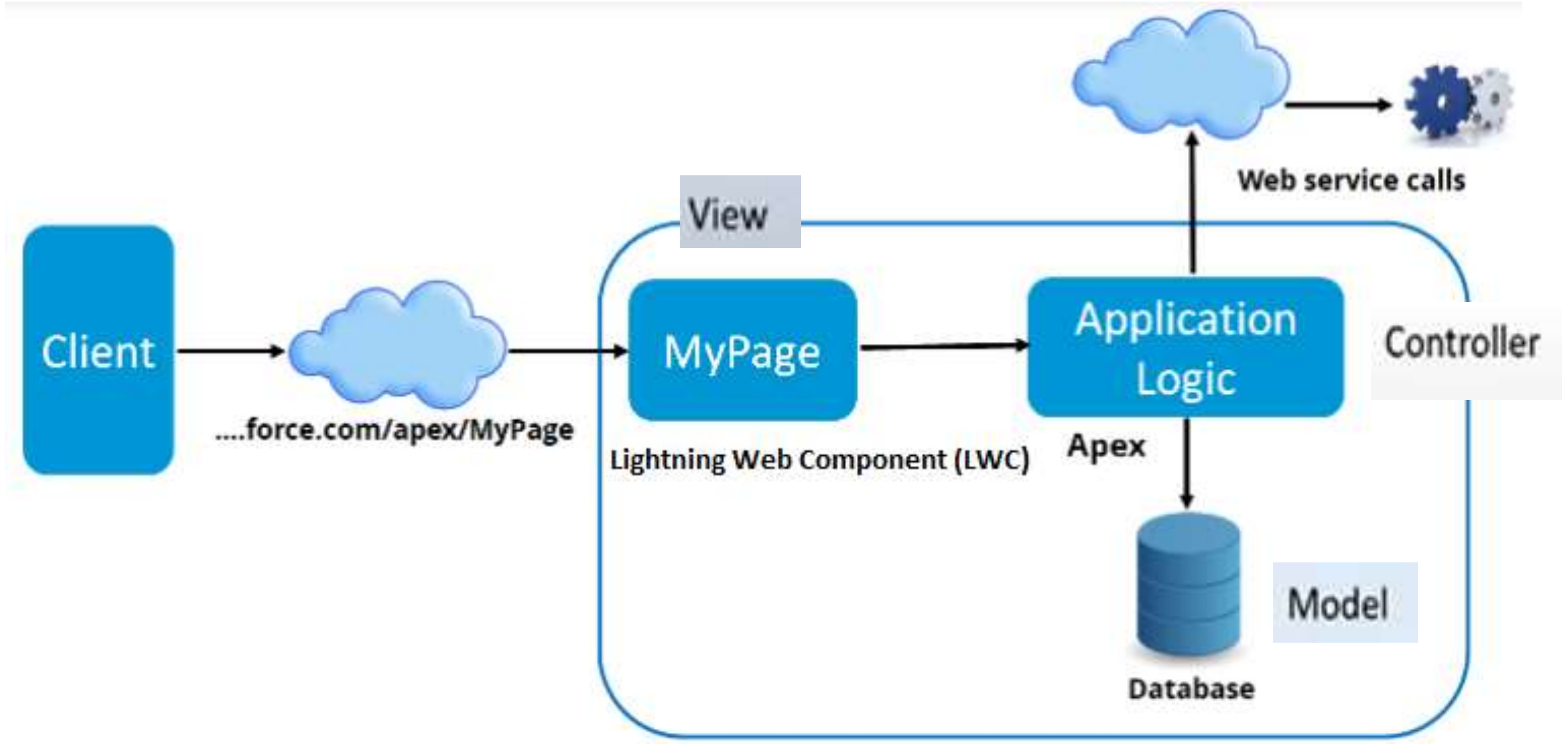
Apex Fundamentals: Introduction to Apex, Apex Classes, Database structure, Execute SOQL and SOSL Queries.

Apex Triggers: Introduction Apex Triggers, Bulk Apex Triggers.

Salesforce Application Anatomy (MVC Architecture)



Software Anatomy	Salesforce Application Anatomy	Declarative Tools (Point & Click)	Programmatic Tools (Code)
Database	Model	• Standard Objects • Custom Objects	• **sObject** • **Wrapper Classes**
Business Logic	Controller	• Workflow Rules • Process Builder • Lightning Flows	• **Apex Classes** • **Triggers**
User Interface	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.

APEX Basics

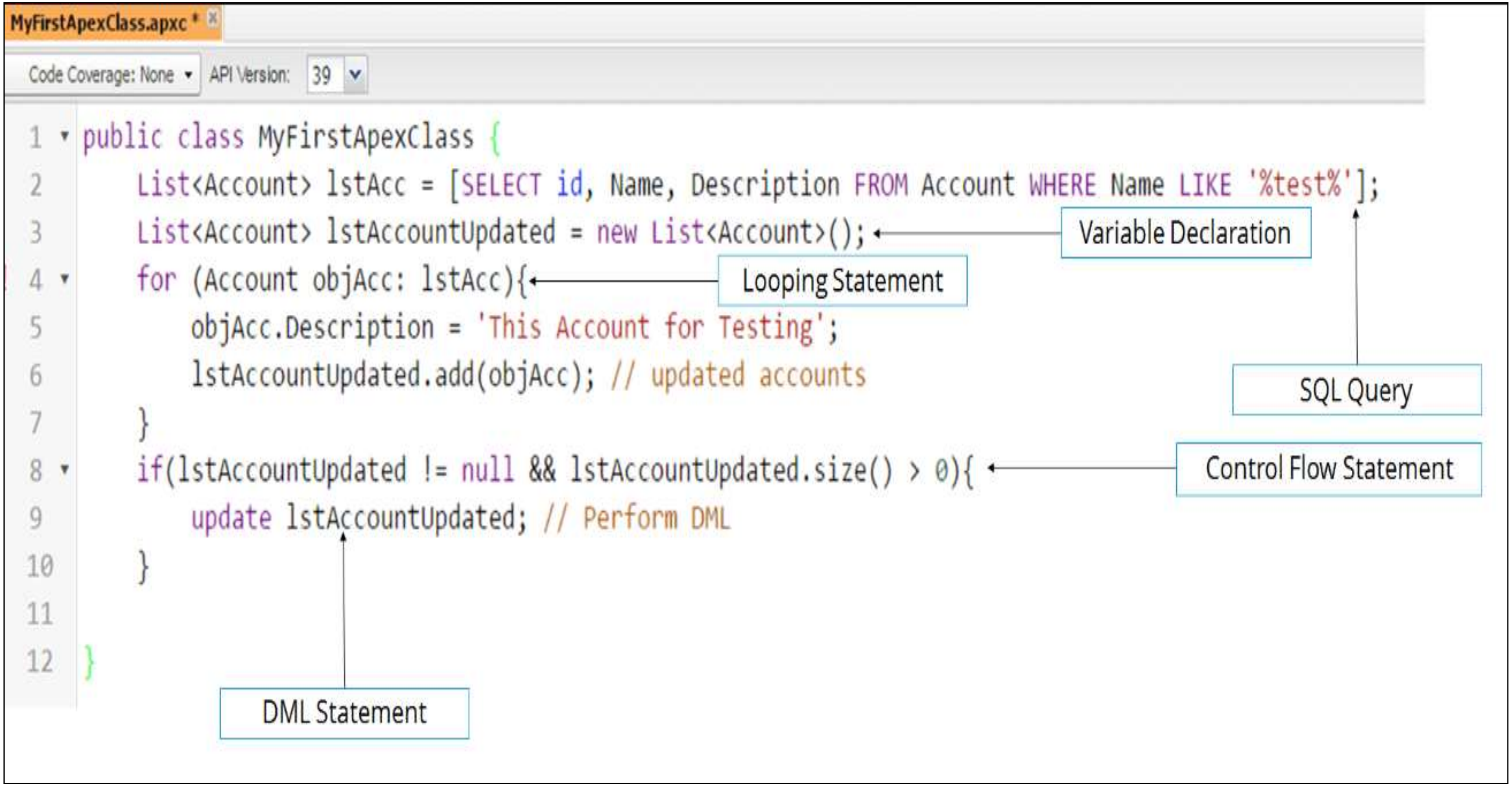
What is Apex?

- ❑ Apex is a **core Salesforce language**.
- ❑ 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 is **similar to Java and C# in terms of syntax and structure**, making it familiar to developers

Why to use Apex?

- ❑ Apex allows you to **execute flow and transaction control statements** on the Force.com platform.
- ❑ Apex is essential for Salesforce development because it allows for **deep customization and automation of business processes within the Salesforce platform**.
- ❑ It enables developers to **create custom triggers, batch processes, and scheduled jobs, ensuring efficient data management**.
- ❑ Apex's seamless **integration with Salesforce objects and its support for complex business logic** make it ideal for **enhancing CRM functionality**.

For Example

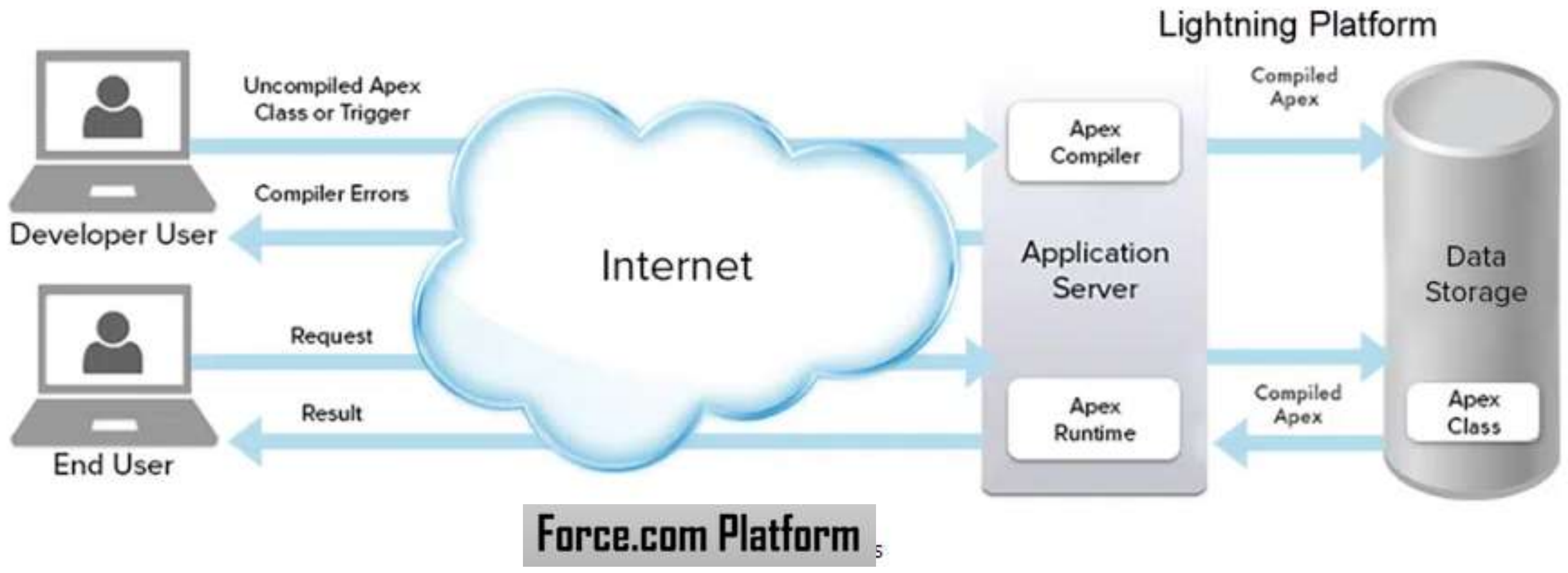


```
public class MyFirstApexClass {
    List<Account> lstAcc = [SELECT id, Name, Description FROM Account WHERE Name LIKE '%test%'];
    List<Account> lstAccountUpdated = new List<Account>();
    for (Account objAcc: lstAcc){
        objAcc.Description = 'This Account for Testing';
        lstAccountUpdated.add(objAcc);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

}
```

Introduction to Apex

APEX Basics

How Apex Works in Salesforce



When a developer writes and saves Apex code to the platform, the platform application server first **compiles the code into an abstract set of instructions that can be understood by the Apex runtime interpreter, and then saves those instructions as metadata.**

When an **end user triggers the execution of Apex**, perhaps by clicking a button or **accessing a page build from LWC,** the platform application server retrieves the compiled instructions from the metadata and sends them through the runtime interpreter before returning the result.

Features of Apex in Salesforce

1. 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.
- It has support for **SOQL (Salesforce Object Query Language) and SOSL (Salesforce Object Search Language) queries**, which return a bunch of subject records.

2. Easy to Use

The foundation of Apex includes **well-known Java idioms, including the notation for objects and arrays, variables, expressions, blocks, and conditional statements, as well as loops.**

When Apex adds new components, it does so with simple syntax and semantics that promote practical usage of the Lightning Platform. As a result, Apex generates code that is simple to write and concise.

Features of Apex in Salesforce

3. Data Focused

Multiple query and DML statements can be **threaded together into a single unit of work on the Salesforce server** using data-focused Apex. Similarly, developers can thread together several transaction statements on a database server using **database-stored procedures.**

4. Multitenant Aware

The Lightning Platform interprets, runs, and manages Hosted Apex in its entirety. Apex is **aware of multiple tenants and operates in a multitenant environment, just like the rest of the Lightning Platform.**

This is the reason behind the **Apex runtime engine's extreme caution when it comes to runaway code that could otherwise monopolize shared resources**. Code that goes outside its bounds fails and generates error messages that are easy to understand.

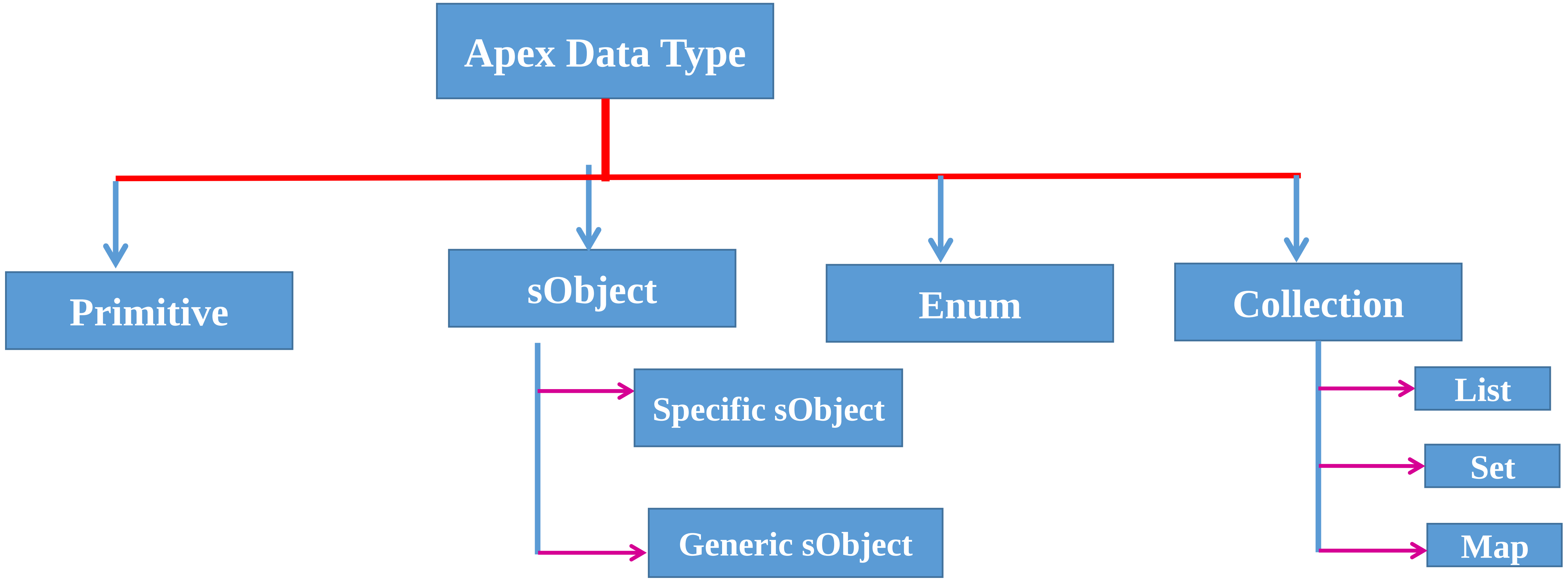
Introduction to Apex

Apex Data Types

Apex Datatypes is a data type that can **be assigned to a variable**. By defining the data type of a variable, you tell the programming language **what kind of information that variable can represent** and how it can be manipulated within your code. This helps prevent errors and ensures your program works as intended.

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

The data types in Apex can be broadly categorized into



Primitive Data Type in Apex

➢**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

Primitive Data Type in Apex

- **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

Primitive Data Type in Apex

- **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

Primitive Data Type in Apex

- **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:**

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';
select  Id , name from Contact
```

Introduction to Apex

sObject Data Type in Apex

What are sObjects in Salesforce?

- ❑ In Apex, the sObject data type represents **Salesforce objects, both standard and custom**.
- ❑ **sObjects in Apex serve as a representation of a Salesforce record.**
- ❑ It's a versatile data type used to **interact with any record** in Salesforce, **encompassing fields, relationships, and metadata**.

sObject Data Type in Apex

- ❑ 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**.

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 in the database they are stored in the sObject variable**.

Specific sObject Data Type in Apex

The API object name becomes the data type of the sObject variable in Apex.
In this example, **Account is the data type of the acct variable.**

For Example: **For Standard Object:**

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

Here,
Account = sObject datatype
acc = sObject variable
new = Keyword to create new sObject Instance
Account() = Constructor which creates an sObject instance
Name =‘Vikram’ = Initializes the value of the Name field in account sObject

- **For Custom Object:**

```
Student__c  std = new Student__c(Name = ‘Rishi’, Email__c = ‘rishi @gmail.com’);
```

Specific sObject Data Type in Apex

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);
```

(OR)

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

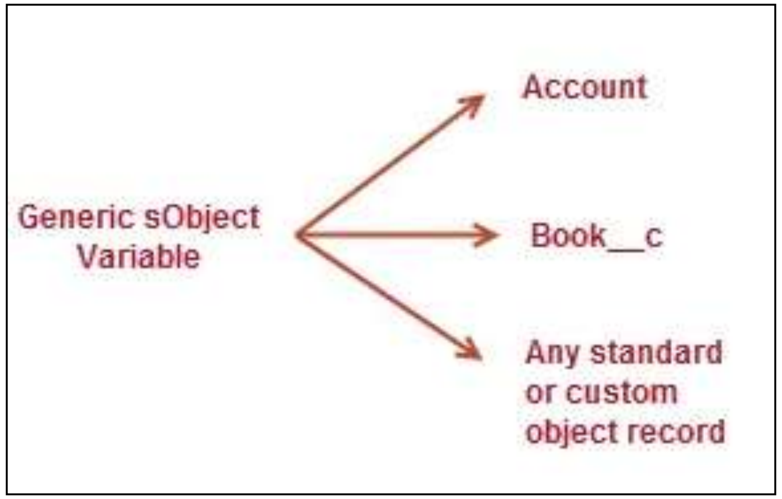
```
Account acct = new Account ();
      acct.Name = Acc1;
       acct.BillingCity = Washington;
```

Generic sObject Data Type in Apex

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.



This example shows how any Salesforce object, such as an account or a custom object called Book__c, can be assigned to a generic sObject variable.

Generic sObject Data Type in Apex

For example,

```
sObject s1;

s1=new Account();

s1.Name='Ajay'

system.debug(s1.Name);
```

// Will not Work

We cannot use dot operator for Generic sObject

Cast Generic sObjects to Specific sObject Types

Generic sObject Data Type in Apex

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. For example,

Account a = (Account)genericSObject;

We can cast a list of generic sObjects to a specific sObject type, like so:

List accountList = (List)genericSObjectList;

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

Generic sObject Data Type in Apex

Cast Generic sObjects to Specific sObject Types

```
sObject s1;

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

    s1 = new Branch__c();
    Branch__c Bran = (Branch__c) s1;
    Bran. BName__c = 'CSE';
    system.debug(Bran.BName__c );
```

Generic sObject Data Type in Apex

Setting and Accessing values from Generic sObjects:

Set a field value on an sObject using put() method

- 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', 'AAA');
```

Access a field on a sObjectusing get() method

- 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

Collection Data Type in Apex

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**

1. List collection

- A list is an **ordered collection** of elements **characterized by their indexes**.
- **"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.
- A list element **can be of any type**, including primitive types, collections, sObjects, user-defined types, and Apex types.
- They can also be **nested inside other lists**. It can have **seven levels of nested collections** inside of it.
- List in apex is **similar to the array in Java**.
- It can be used **to hold the results of SOQL queries**.
- **SOQL Queries:** Retrieve data from Salesforce objects and return it as a List.
- **DML Operations:** Use List to perform **bulk insert, update, or delete operations efficiently.**

Syntax for a list is as follows:

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

Here is an example:

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

You can use the array notation as well to declare the List,

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

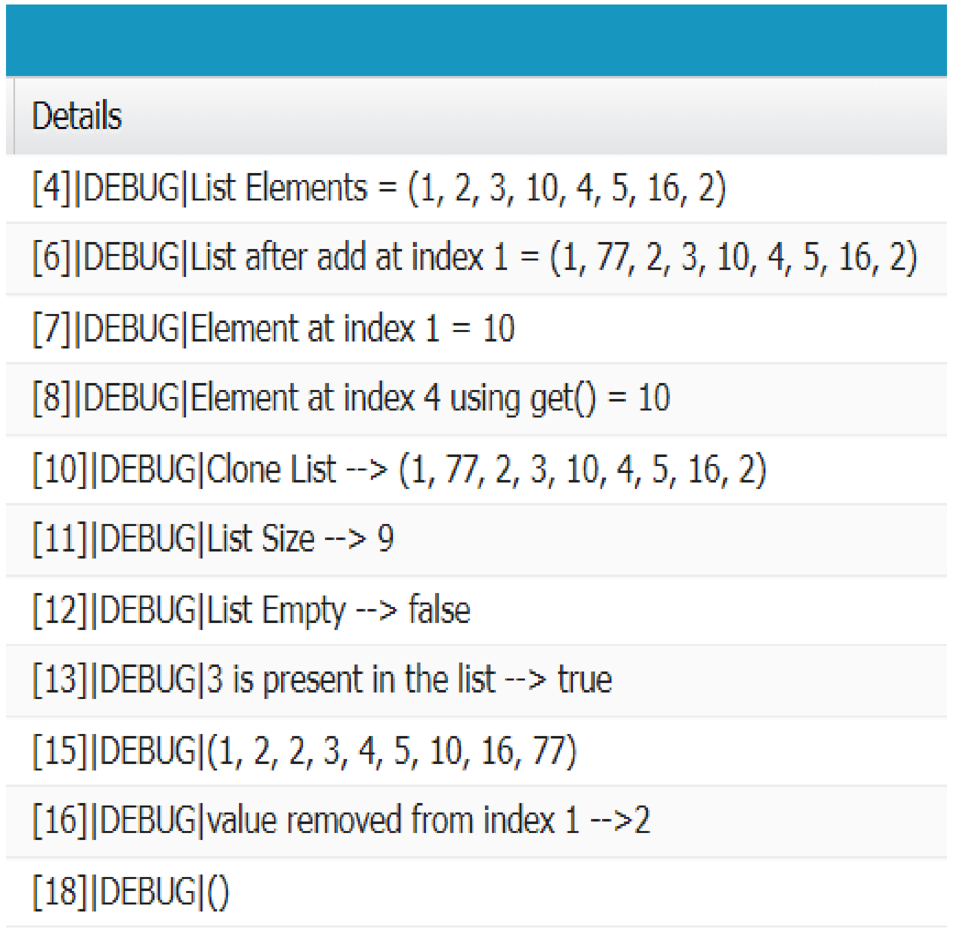
Methods supported by List.

1) **Add(Value) -** to add elements to the list.
2) **Add(index,value)** - to add the elements in the specific index to the list.
3) **get(index)** - to access elements from the list by specifying the index.
4) **Clone()** - This method is used to create a new instance of a collection that contains all the elements of the original collection. It is often used to create a **copy of a collection** .
5) **size() -** Returns the size of the list i.e. the **total number of elements** present in the list
6) **isEmpty()** - Returns true if the list is empty, otherwise returns false
7) **contains()** - Returns true if a **particular element is present** in the list, otherwise, returns false
8) **sort()** - Sort the elements in the list in ascending order
9) **Remove(index)** – to remove an element from the list from the specified index
10) **clear()** - The clear() method is used to remove all elements from a collection, effectively emptying it. It is commonly used to reset a collection to its initial empty state.

Methods supported by List - 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  = ' + numbers);
   System.debug('Element at index 4 = ' + 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

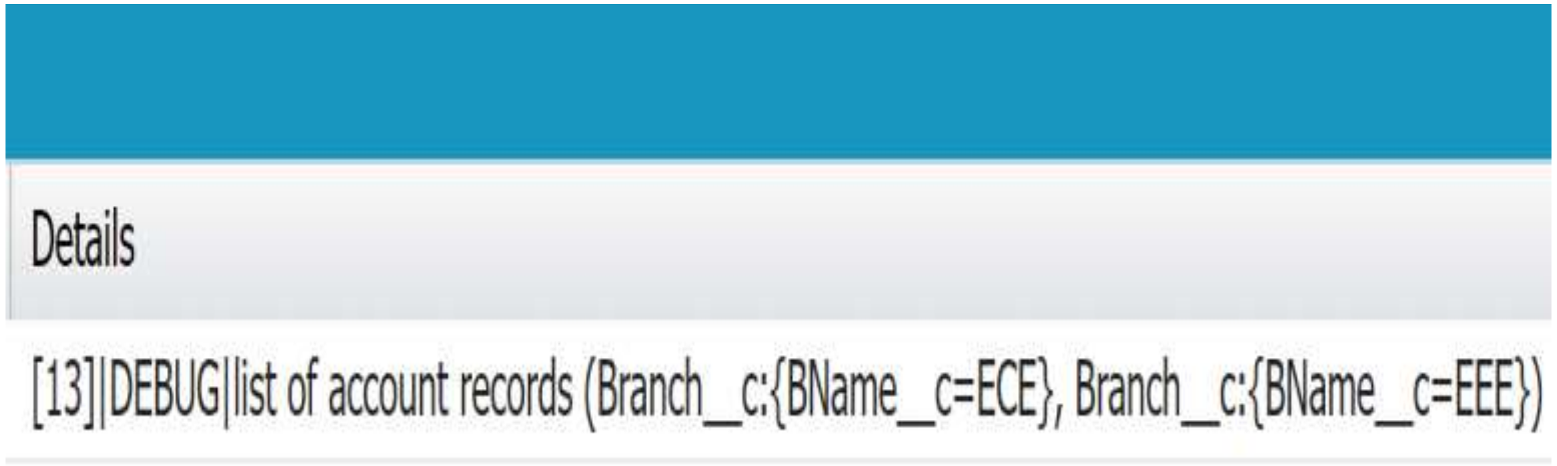


```
Details
[4]|DEBUG|List Elements = (1, 2, 3, 10, 4, 5, 16, 2)
[6]|DEBUG|List after add at index 1 = (1, 77, 2, 3, 10, 4, 5, 16, 2)
[7]|DEBUG|Element at index 1 = 10
[8]|DEBUG|Element at index 4 using get() = 10
[10]|DEBUG|Clone List --> (1, 77, 2, 3, 10, 4, 5, 16, 2)
[11]|DEBUG|List Size --> 9
[12]|DEBUG|List 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 ();    // sObject created
a1. BName__c = 'ECE';

accList.add(a1);

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

accList.add(a2);

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

OUTPUT

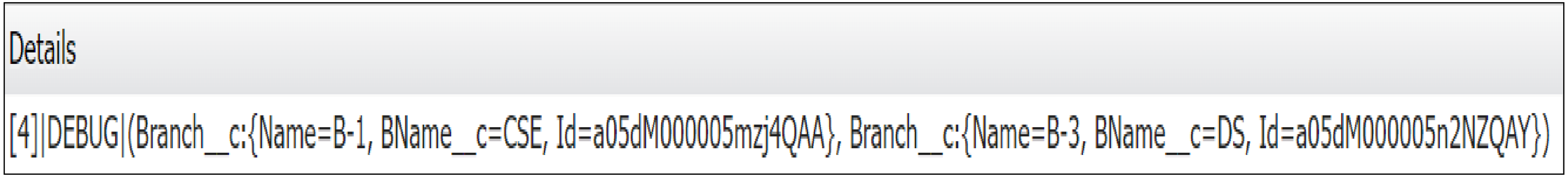


Details

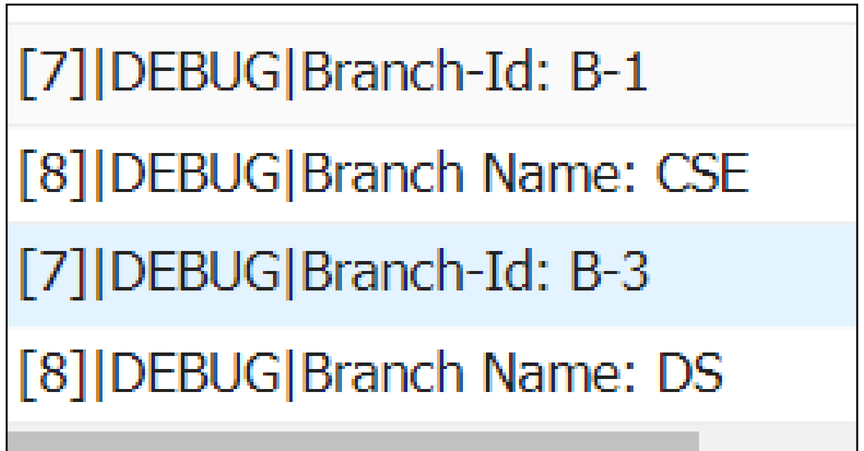
```
[13]|DEBUG|list of account records (Branch__c:{BName__c=ECE}, Branch__c:{BName__c=EEE})
```

SOQL used in List

```
List<Branch__c> customObjectRecords = [SELECT Name, BName__c FROM Branch__c];
system.debug(customObjectRecords);

for (Branch__c record : customObjectRecords) {
    System.debug('Branch-Id: ' + record.Name);
    System.debug('Branch Name: ' + record.BName__c);
}
```



```
Details
[4]|DEBUG|(Branch__c:{Name=B-1, BName__c=CSE, Id=a05dM000005mzj4QAA}, Branch__c:{Name=B-3, BName__c=DS, Id=a05dM000005n2NZQAY})
```



```
[7]|DEBUG|Branch-Id: B-1
[8]|DEBUG|Branch Name: CSE
[7]|DEBUG|Branch-Id: B-3
[8]|DEBUG|Branch Name: DS
```

Set Collection

- A set is an **unordered collection of elements**.
- **Insertion order is not preserved** in the set.
- Set elements can be of any data type—primitive types, collections, sObjects, user-defined types, and built-in Apex types
- Set **does not allow duplicate value.**
- **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.**
- They are typically used for **checking membership of data, not retrieving key/value pairs or values by index.**

Syntax for a Set is as follows:

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

Set Methods in Salesforce

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	Output

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');
		System.debug(l);
		s.addAll(l);
		System.debug(s);
			(abc,def)
			
			{ABC, XYZ, abc,def}

3	**clear()**
	Removes all the elements.
		s.clear();
		s.addAll(l);
		System.debug(s);
			{ abc,def}

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

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

Some common methods of set in Apex:

S.No	Function	Example	Output

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);
		s.retainAll(l);
		System.debug(s);
			{abc,def,ghi}
			
			{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(l);
		System.debug(s);
			{ghi}

Map Collection

- 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.**

  Example, ABC and abc will be considered as different keys and treated as unique.
- Maps are useful for working with large data sets, storing multiple values for a single key, and performing operations on groups of similar data

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  };
```

Some common methods of Map in Apex:

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

Some common methods of Map in Apex:

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)

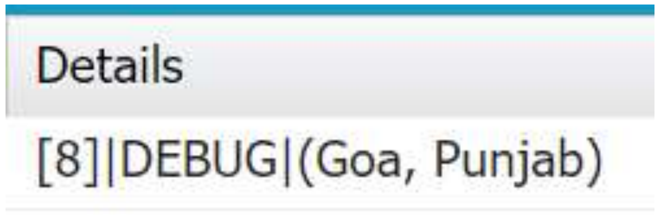
8	**isEmpty()**
	Returns true if the map has zero key-value pairs.
		Map<String, Integer> myMap = new Map<String, Integer>{'a' => 1, 'b' => 2};
		System.debug(myMap.isEmpty());
			False

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);
			{ }

Map with one key and multiple values

```
Map<String,List<String>> India = new  Map<String,List<String>>();

List<String> state = new List<String>();
state.add('Goa');
state.add('Punjab');

India.put('India-key',state);
System.debug(India.get('India-key'));
```

OUTPUT



```
Details
[8]|DEBUG|(Goa, Punjab)
```

Enum Data Type in Apex

An Enum, short for enumeration, is a data type used to define a set of named constants. Programmers use Enums to create a collection of related values that can be used to make code more readable and manageable.

By using an Enum, you can define a variable type that can only have one of a small set of predefined values.

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.

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

EXAMPLE

```
public enum OrderStatus
{
   NEW,
   PENDING,
   SHIPPED,
   DELIVERED,
   CANCELLED
}
```

In this example, the OrderStatus Enum denotes the possible states of an order in a sales process, improving the readability of the code that handles the various stages that an order can be in.

Methods of enum in Apex:

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

1. String Conversion:

An **Enum value can be easily converted to a string** by using the **name() method**, which returns the name of the Enum constant as a string.

Conversely, you can use the **valueOf()** static **method to convert a string to an Enum** constant, provided the string matches one of the Enum's constant names.

```
// From Enum to String
 String summerString = Season.SUMMER.name(); // Returns "SUMMER"
// From String to Enum
Season seasonEnum = Season.valueOf("SUMMER"); // Returns Season.SUMMER
```

2. ordinal()

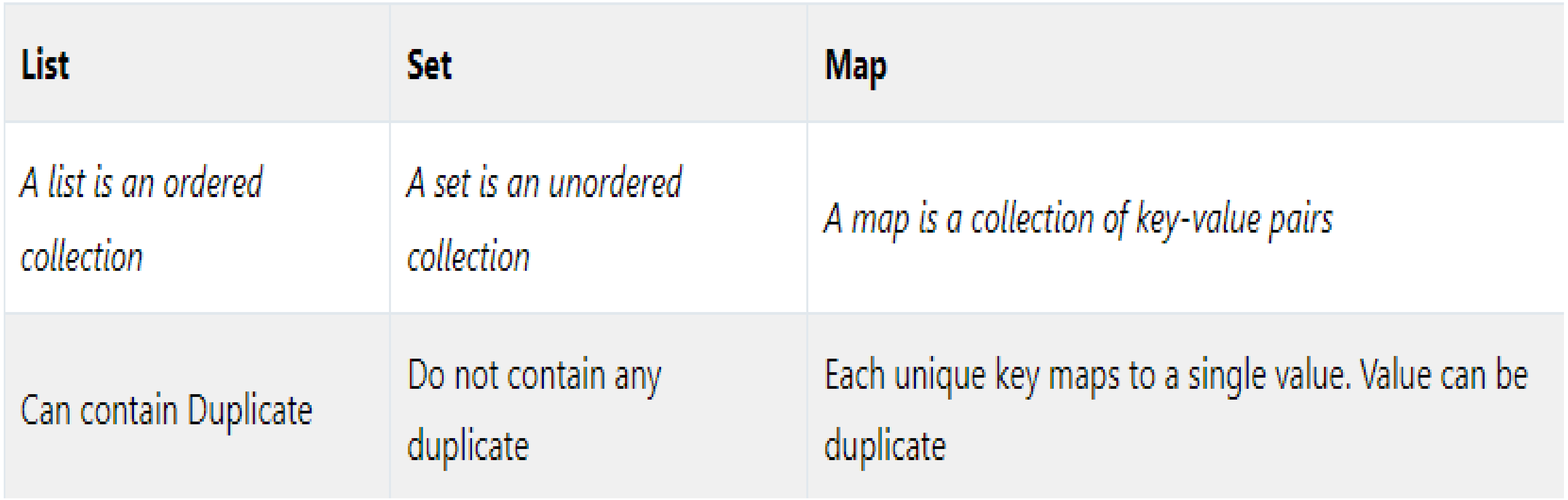
Returns the position of the item, as an Integer, in the list of Enum values starting with zero.

```
Integer summerOrdinal = Season.SUMMER.ordinal(); // Returns 2
```

3. values()

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

```
List<Season> seasons = Season.values();
System.debug(seasons);     // Outputs: (WINTER, SPRING, SUMMER, FALL)
```



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

Apex Classes

Classes

A *class* is a **blueprint.**

Objects are based on classes.

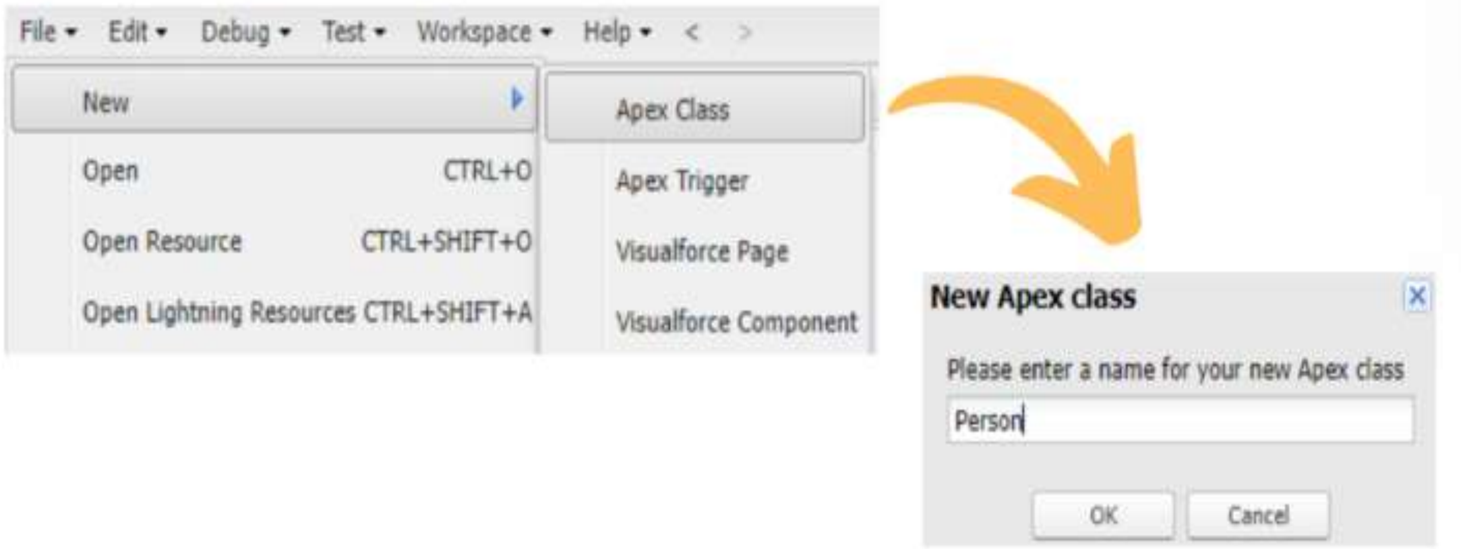
A class defines a set of **characteristics (Variables) and behaviors (Method)** that are common to all objects of that class.

Create a Class in Apex

To create a class,

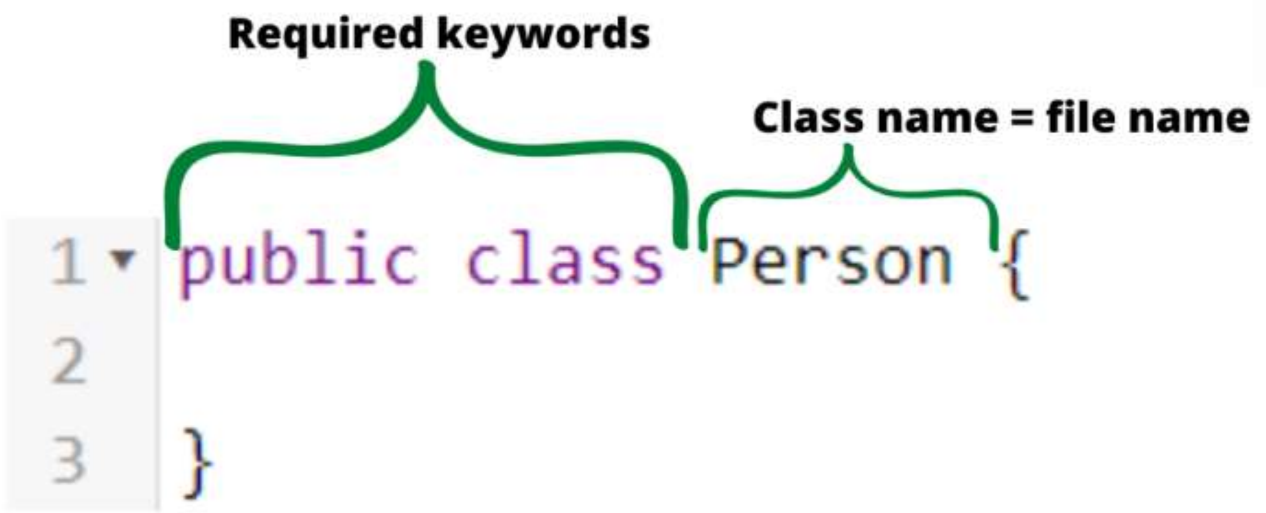
navigate to the Developer Console, click

File → New → Apex Class and type in the name of your class (e.g. "Person").



Apex Classes

```
Syntax:

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



```
public class Person {

}
```

Within the brackets, you can write your actual class logic.

- **Comments**
- **Variables**
- **Methods**
- **Constructors**
- **Other classes (called inner classes)**

Apex Classes

```
private | public | global [virtual | abstract | with sharing | without sharing] class ClassName
 [implements InterfaceNameList] [extends ClassName]
{
            // The body of the class
 }
```

To define a class, specify the following:

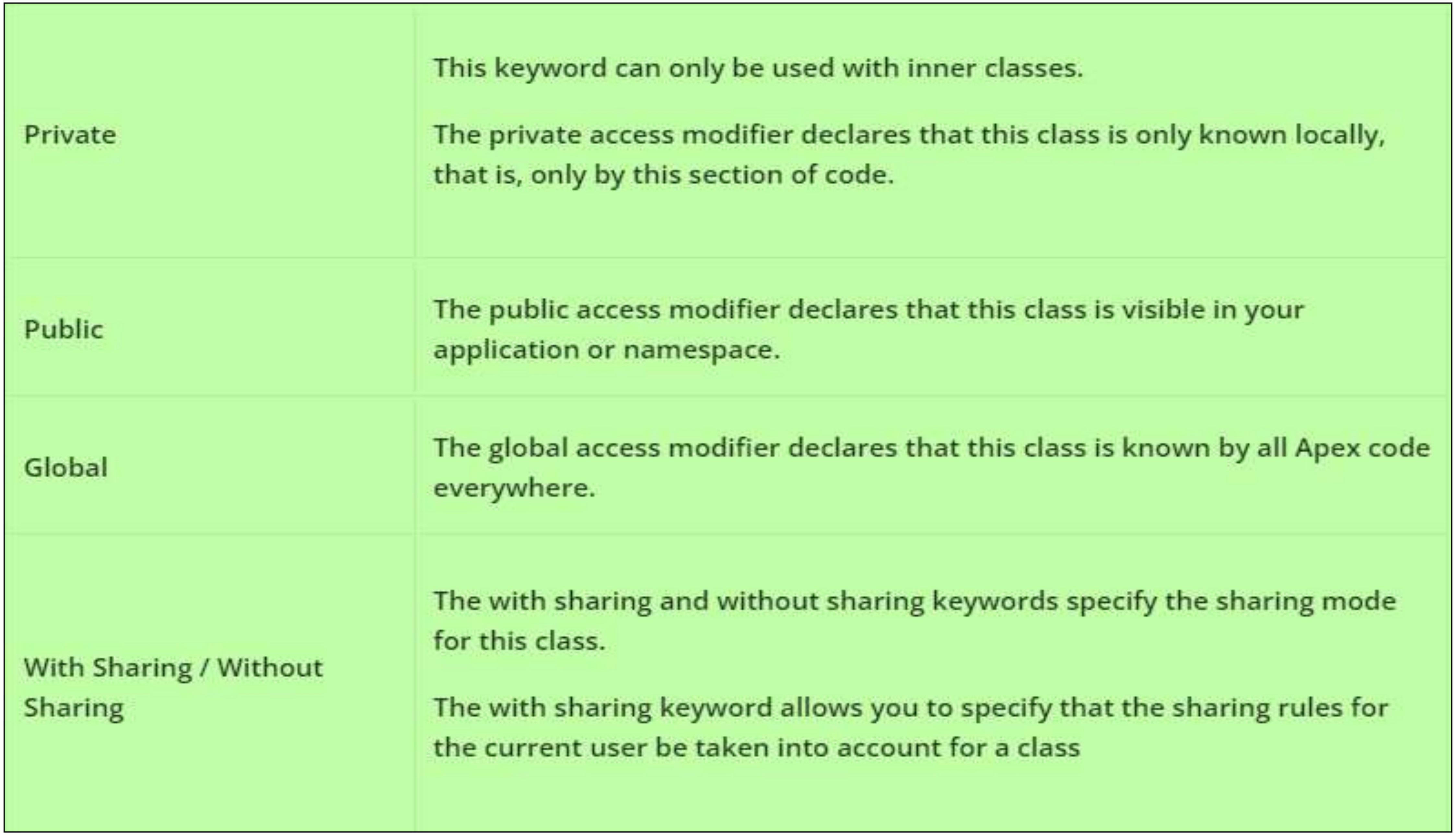
1. Access modifiers:

- You must use one of the access modifiers (such as public or global) in the declaration of a top-level class.
- You do not have to use an access modifier in the declaration of an inner class.

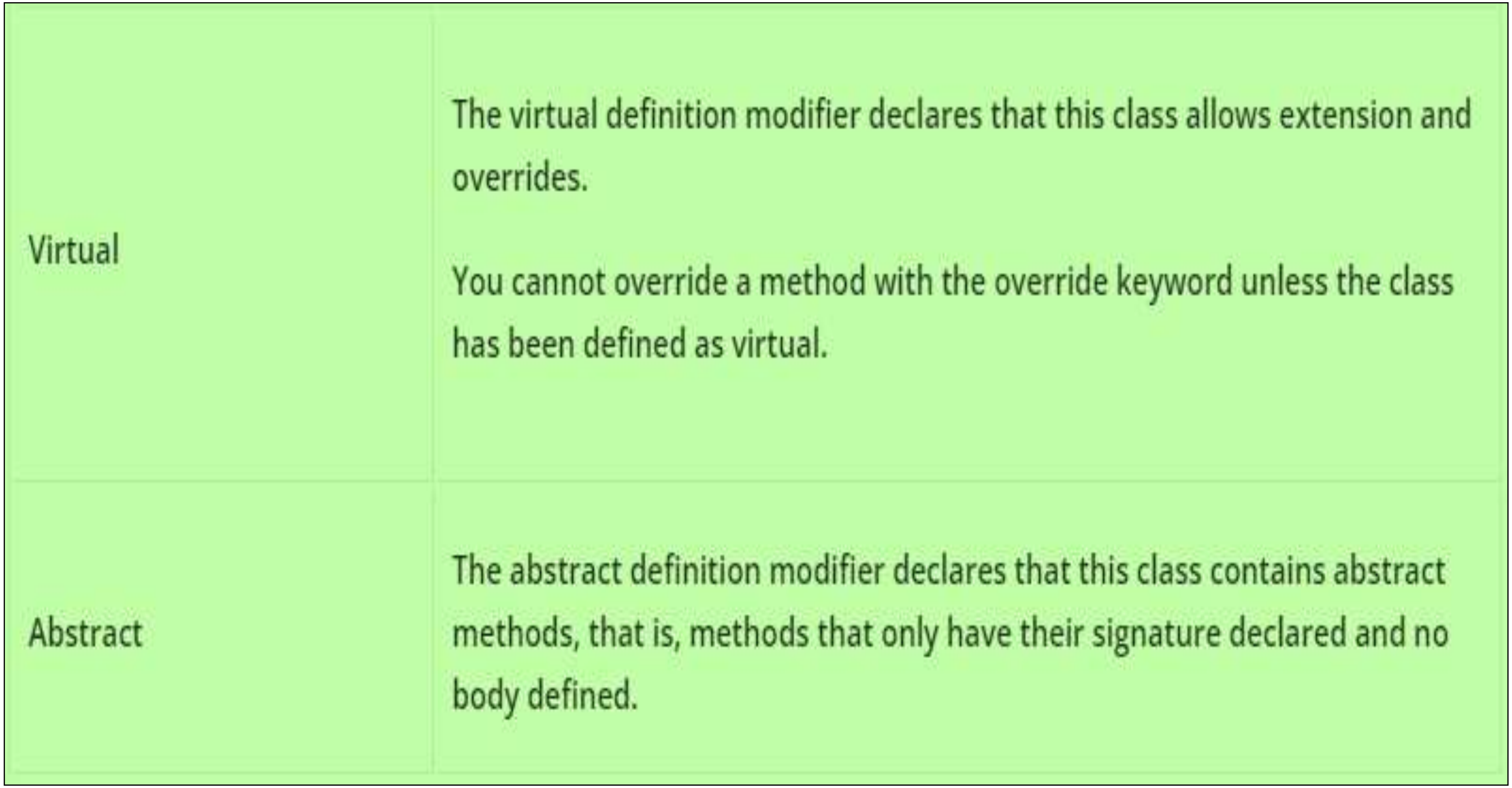
2. Optional definition modifiers (such as virtual, abstract, and so on)

3. Required: The **keyword class** followed by the name of the class

4. Optional extensions and/or implementations



Private	This keyword can only be used with inner classes. The private access modifier declares that this class is only known locally, that is, only by this section of code.
Public	The public access modifier declares that this class is visible in your application or namespace.
Global	The global access modifier declares that this class is known by all Apex code everywhere.
With Sharing / Without Sharing	The with sharing and without sharing keywords specify the sharing mode for this class. The with sharing keyword allows you to specify that the sharing rules for the current user be taken into account for a class



Virtual	The virtual definition modifier declares that this class allows extension and overrides. You cannot override a method with the override keyword unless the class has been defined as virtual.
Abstract	The abstract definition modifier declares that this class contains abstract methods, that is, methods that only have their signature declared and no body defined.

Apex Classes

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**.

The code in your editor looks like this:

```
public class Person
{                    }
```

Apex Classes
Methods inside classes in Apex

Declaring a method

The syntax to declare an Apex method is:

```
public class MyClass {
public ReturnType methodName()
{           // here you can write your code
            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

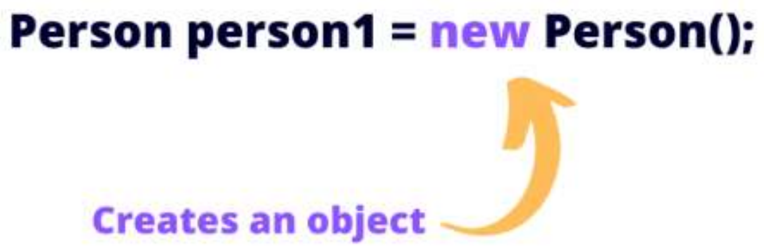
3. At the bottom right**,** click **Execute.**

To Run the Code -Creating Object for a class and accessing variables and method inside the class

Calling methods

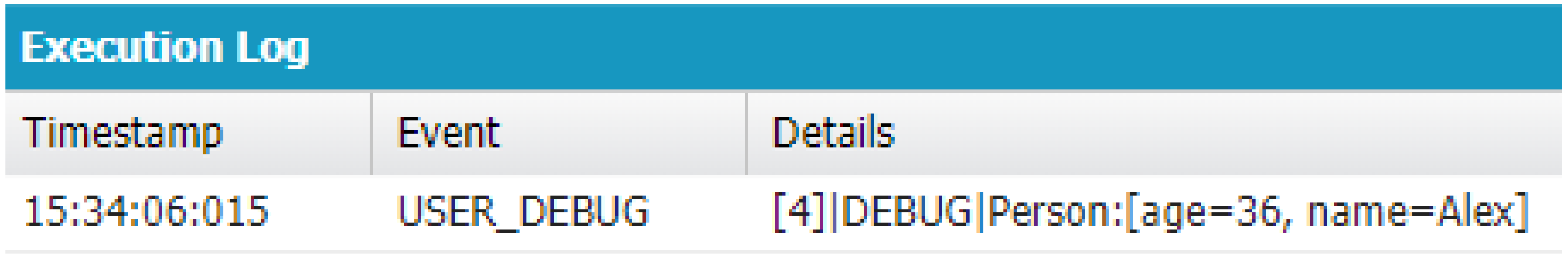
Navigate to Developer Console → Debug → Execute Anonymous Window

```
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]

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

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!

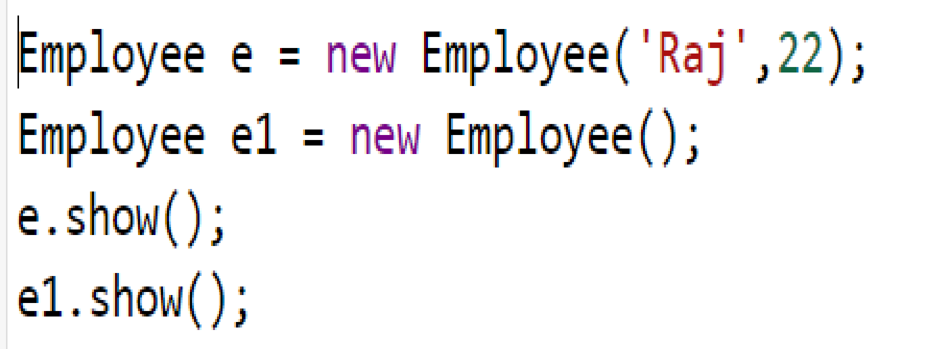
2. Non-parameterized Constructor
It is a constructor that doesn't have any parameters is called Non-parameterized constructor.
Parameters are nothing but the values we are passing inside a constructor.

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

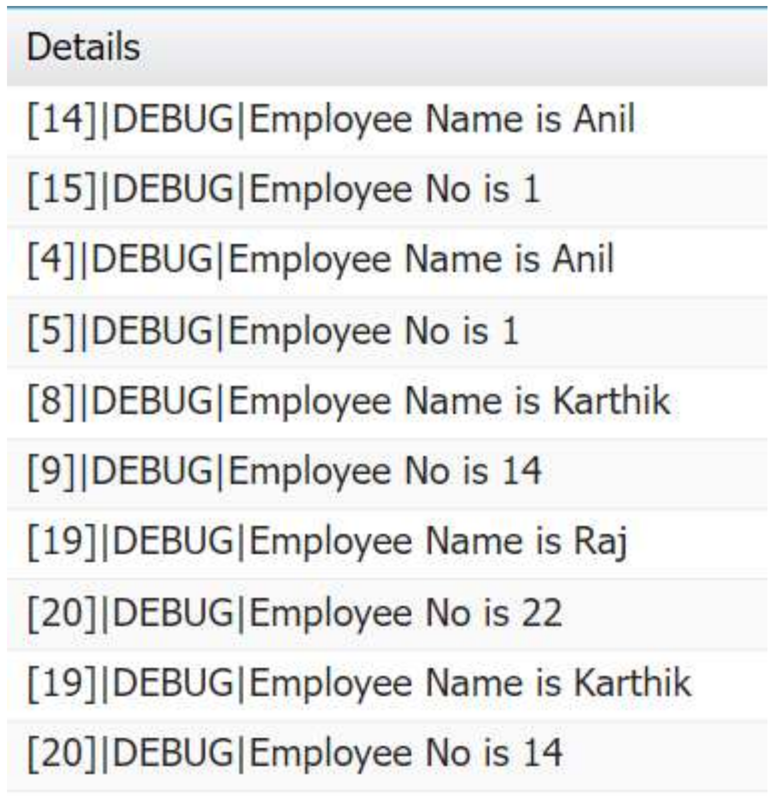
```
public class Employee                                   //Class
 {
public Employee( )                             //Non-Parameterized Constructor
 {   System.debug('Employee Name is '+ EmployeeName);
     System.debug('Employee No is '+ EmployeeNo);
     EmployeeName = 'Karthik';
     EmployeeNo = 14;
     System.debug('Employee Name is '+ EmployeeName);
    System.debug('Employee No is '+ EmployeeNo);
 }
 String EmployeeName='Anil';
    Integer EmployeeNo=01;

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



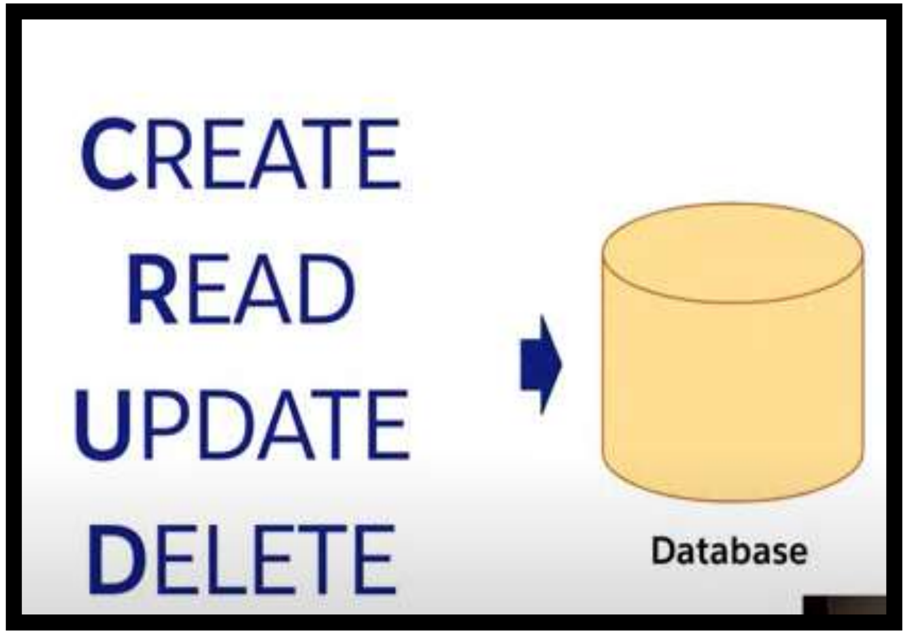
```
Employee e = new Employee('Raj',22);
Employee e1 = new Employee();
e.show();
e1.show();
```

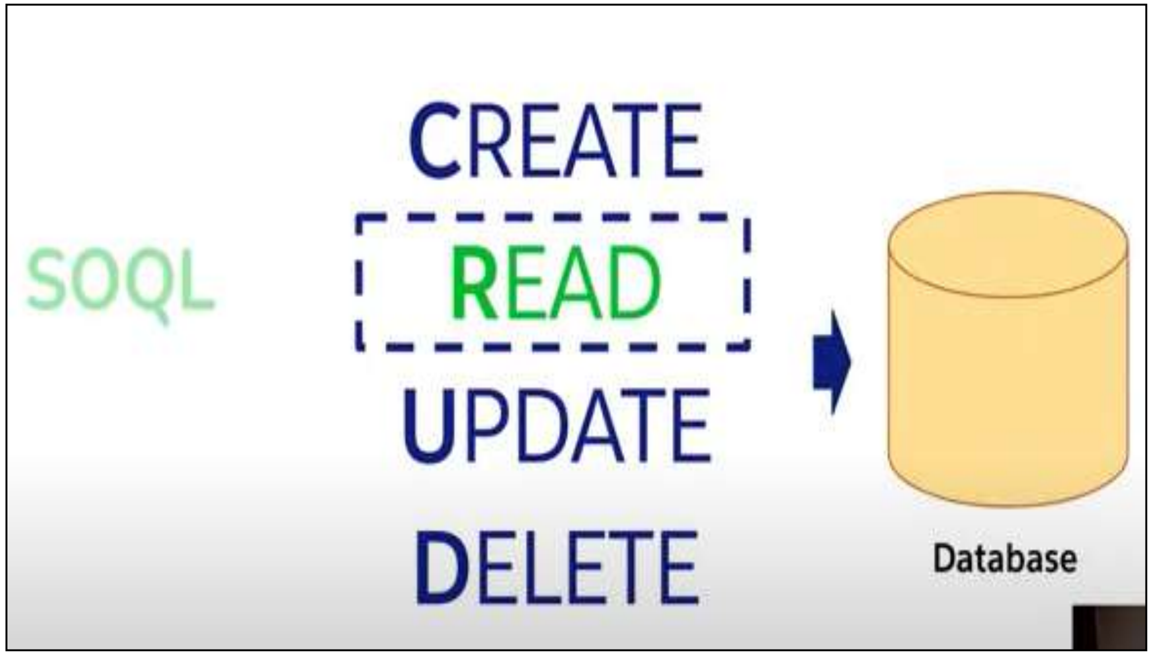
OUTPUT

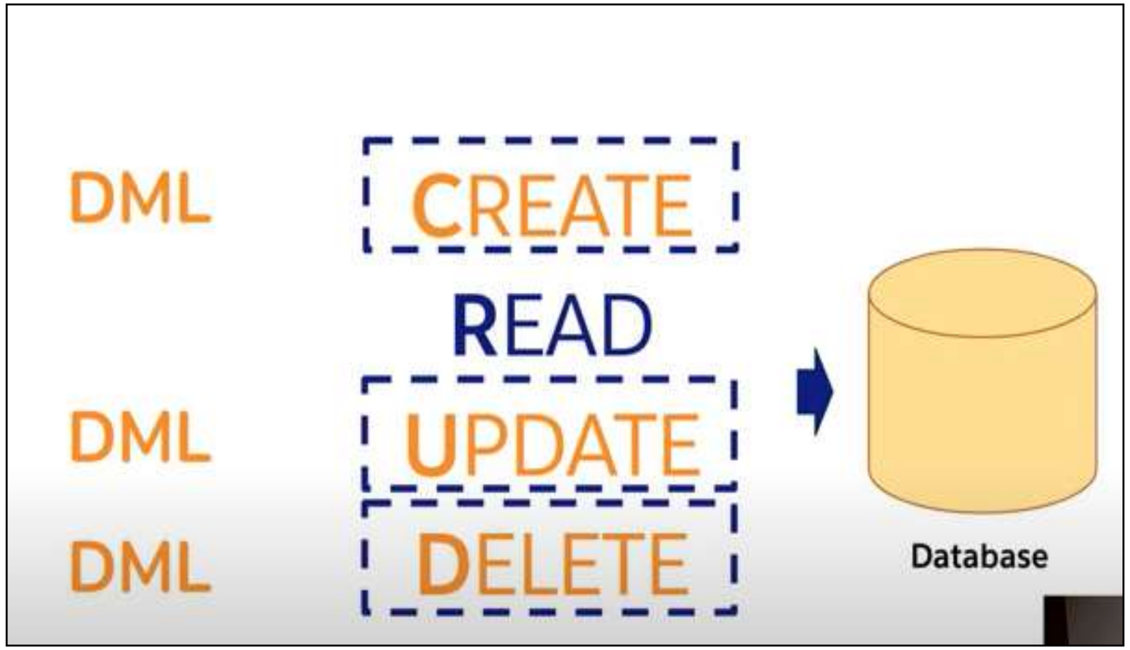


```
Details
[14]|DEBUG|Employee Name is Anil
[15]|DEBUG|Employee No is 1
[4]|DEBUG|Employee Name is Anil
[5]|DEBUG|Employee No is 1
[8]|DEBUG|Employee Name is Karthik
[9]|DEBUG|Employee No is 14
[19]|DEBUG|Employee Name is Raj
[20]|DEBUG|Employee No is 22
[19]|DEBUG|Employee Name is Karthik
[20]|DEBUG|Employee No is 14
```

Execute SOQL and SOSL Queries

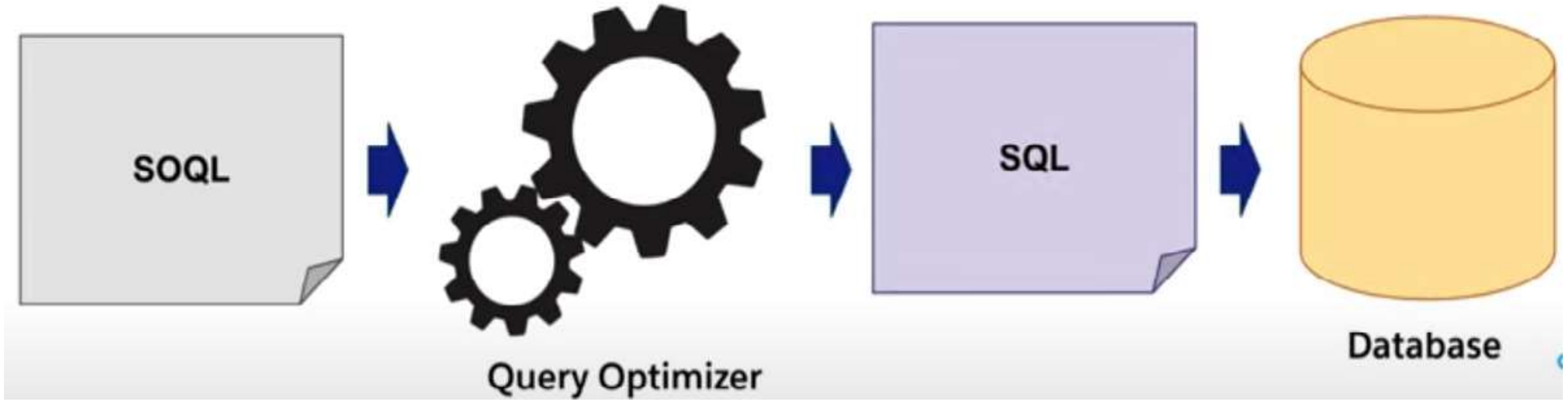






Execute SOQL Queries

- The **Salesforce Object Query Language(SOQL)** designed to work with **SFDC Database.**
- It is **used to retrieve data** from the Salesforce database according to the specified conditions and objects.
- It can search a record on a given criterion only in **single sObject.**
- It **cannot search across multiple objects but it does support nested queries.**
- 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 Queries

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

Syntax

```
SELECT fields FROM ObjectName WHERE <Condition>
```

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

Example

```
Account[] accts = [SELECT Name,Phone FROM Account];     //array of sObject
                              (OR)
List<Account> accts = [SELECT Name, Phone FROM Account];   // List of sObject
```

Execute SOQL Queries

The query has two parts:

1. SELECT Name, Phone: 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 (e.g. SELECT Phone).

Like SQL languages, **you can’t specify * for all fields**. You must specify every field explicitly to get it.

2. FROM Account: This part specifies the standard or custom object that you want to retrieve. In this example, it’s Account. For a custom object called Branch__C .

Execute SOQL 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 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 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 three account that is returned.

```
List<Account> acc = [SELECT Name, Phone FROM Account LIMIT 3];
```

Execute SOQL Queries

Combine All option clause 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);                    // Write all account array info
```

Execute SOQL Queries

Using SOQL Aggregates to Summarize Data

Aggregate functions allow us to **summarize data and calculate metrics** like count, sum, average

- COUNT(): Returns the total number of records.
- SUM(): Calculates the sum of values for a numeric field.
- AVG(): Returns the average value of a numeric field.
- MIN(): Returns the minimum value for a field.
- MAX(): Returns the maximum value for a field.

```
List<AggregateResult>  results = [SELECT COUNT(Name) RecordCount FROM branch__c];
 Integer totalCount = (Integer)results[0].get(' RecordCount ');
  system.debug(totalCount);
```

- In Salesforce Apex, the **AggregateResult class** is used to handle the results of SOQL queries that **include aggregate functions, such as COUNT(), SUM(), MAX(), etc.**
- **RecordCount** is an **alias** given to the result of COUNT(Name) for easier reference in the Apex code.
- The result is **cast to Integer** because COUNT(Name) returns a numeric value representing the number of records.

Execute SOQL 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 **givenYear** variable in the WHERE clause.

```
String givenYear = 'I';
student__C[] stud = [SELECT Name, SName__c FROM student__C WHERE Year__c =: givenYear];
system.debug(stud);
```

Execute SOSL Queries

Salesforce Object Search Language (SOSL)

- It 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 a search language in salesforce and the important feature is , we can search in **multiple objects at same time**.
- In **SOQL, we can query only one object at a time** but in SOSL, We can **search for some specified string like 'testString' in multiple objects at the same time.**
- We can **mention in which fields of all the sObjects, we want to search** for the string specified.
- The SOSL query start with the keyword **'FIND'.**

Execute SOSL Queries

- You can specify, **which fields to return for each object mentioned** in SOSL query. Suppose you have performed search on three objects Account & Contact. Then you can mention like, for list returned with **Account results only (Name, Industry) fields** should come, and for **Contacts results (firstName, lastName) fields**
- The **result of SOSL is a list of lists of sObjects “List<List<sObject>>”.**
- The returned result contains the list of sObjects in the **same order as order mentioned in SOSL query.**
- If a SOSL query does not return any records for a specified sObject type, then search results include an empty list for that sObject.
- The **search string should be at least two characters long**.
- **Cannot be used on triggers**
- **DML operations cannot be performed**

Execute SOSL Queries

Syntax of a basic SOSL query

```
FIND ‘search_query’ [IN searchGroup] [RETURNING list of object and Fields]
```

- **search_query**: The term you want to search for.
- **searchGroup**: Specifies the context of the search. It is optional.
  If not specified, the **default search group is all fields**
  - ALL FIELDS
  - NAME FIELDS
  - EMAIL FIELDS
  - PHONE FIELDS
  - SIDEBAR FIELDS
- **object_list**: The objects and fields you want to return.

Example

```
FIND ‘John’ IN Name Fields RETURNING Contact(Id, Name, Email), Account(Id, Name)
```

This query searches for the term “John” in the Name fields of both Contact and Account objects, and it retrieves the Id, Name, and Email of contacts and Id and Name of accounts.

Execute SOSL Queries

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

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

SearchGroup 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.

Example

```
String soslQuery = 'FIND {Acme* OR *Corp?} IN ALL FIELDS RETURNING Account(Id, Name), Contact(Id, Name)';

 // Execute the SOSL query
List<List<SObject>> searchResults = [soslQuery];
```

- **Acme*:** This searches for any terms that start with "Acme" and can have any characters following it.
  For example, it will match "Acme Corporation" and "Acme Inc."
- ***Corp?:** This searches for terms that end with "Corp" followed by exactly one more character.
  For example, it will match "CorpX" but not "Corporation".
- **OR**: Used to combine multiple search terms. In this query, Acme* OR *Corp? means the search will return results
  that match either "Acme*" or "*Corp?".

Execute SOSL Queries

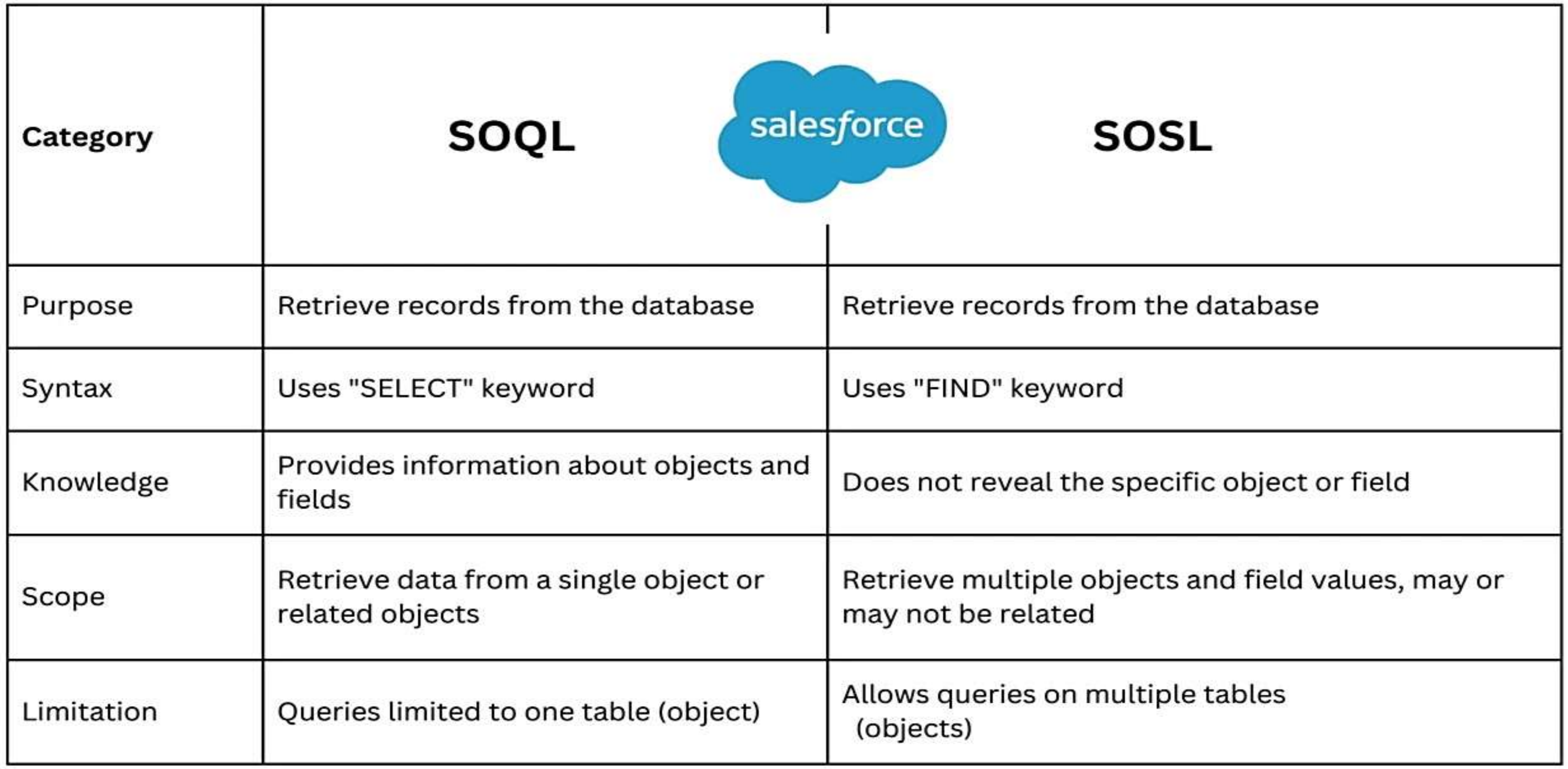
Execute SOSL in Apex

```
// Execute the SOSL query
List<List<SObject>> searchResults = [FIND ‘John’ IN ALL FIELDS RETURNING Account(Id, Name), Contact(Id, Name)];

// Process the results
List<Account> accounts = (List<Account>) searchResults[0];
List<Contact> contacts = (List<Contact>) searchResults[1];

// Example of processing results
for (Account acc : accounts) {
   System.debug('Found Account: ' + acc.Name);
}

for (Contact con : contacts) {
   System.debug('Found Contact: ' + con.Name);
}
```

•**Execute the SOSL query**: Use the sosl keyword directly in square brackets to execute the query.
•**Process the results**: The result of a SOSL query is a **list of lists**, where each **sublist contains records of a specific object type**. You need to **cast the results** appropriately.

Execute SOQL and SOSL Queries

Differences and Similarities Between SOQL and SOSL

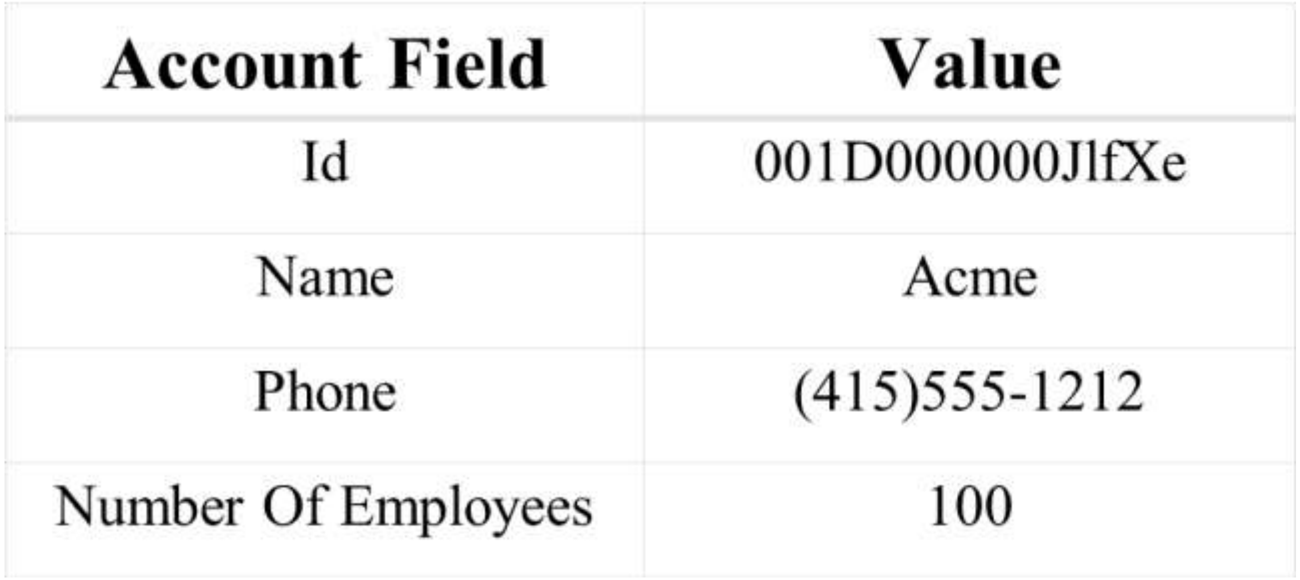


Category	SOQL	SOSL
Purpose	Retrieve records from the database	Retrieve records from the database
Syntax	Uses "SELECT" keyword	Uses "FIND" keyword
Knowledge	Provides information about objects and fields	Does not reveal the specific object or field
Scope	Retrieve data from a single object or related objects	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)

Database structure

Use sObjects:

- Apex is tightly integrated with the database, so you can access Salesforce records and their fields directly from Apex.
- **Every record** in Salesforce is natively represented as an **sObject** in Apex.
- **For example:** the **Acme account record** corresponds to an **Account sObject** in Apex.



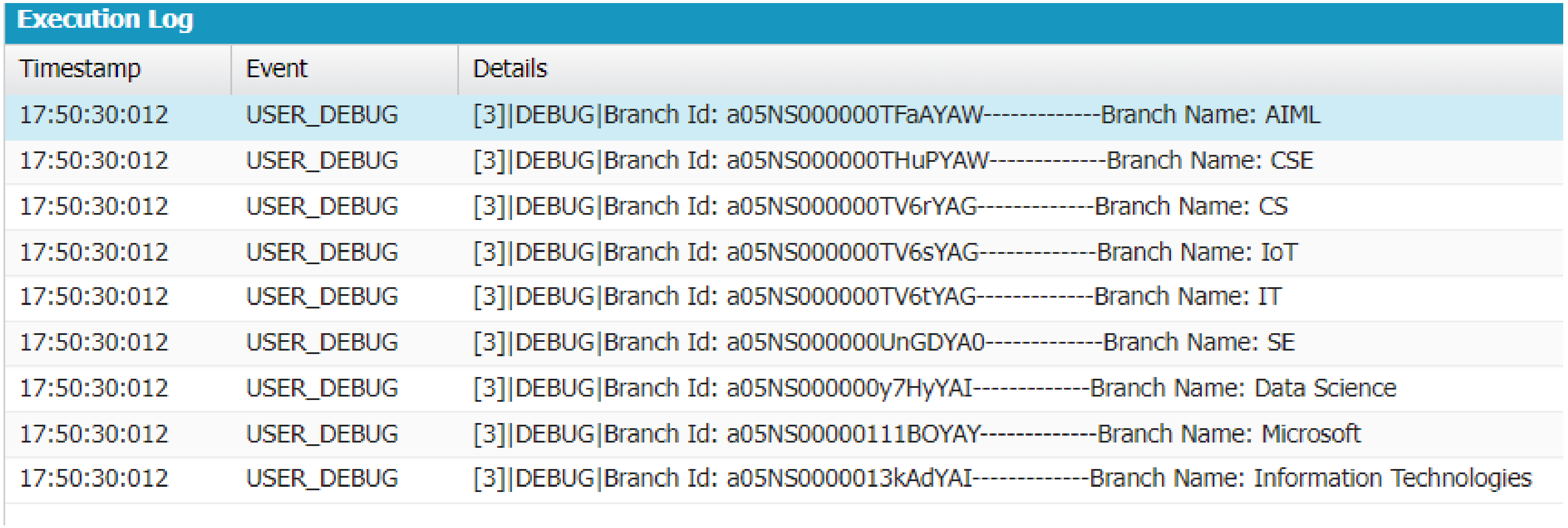
Account Field	Value
Id	001D000000JlfXe
Name	Acme
Phone	(415)555-1212
Number Of Employees	100

Database structure

```
public class Utility {
 public static void display() {
          List<branch__c> accList = new List<branch__c>();
          branch__c a1 = new branch__c ();
          a1. BName__c = 'ECE';
          accList.add(a1);
          }
          }
```

Database structure

```
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);
         }
   }
}
```



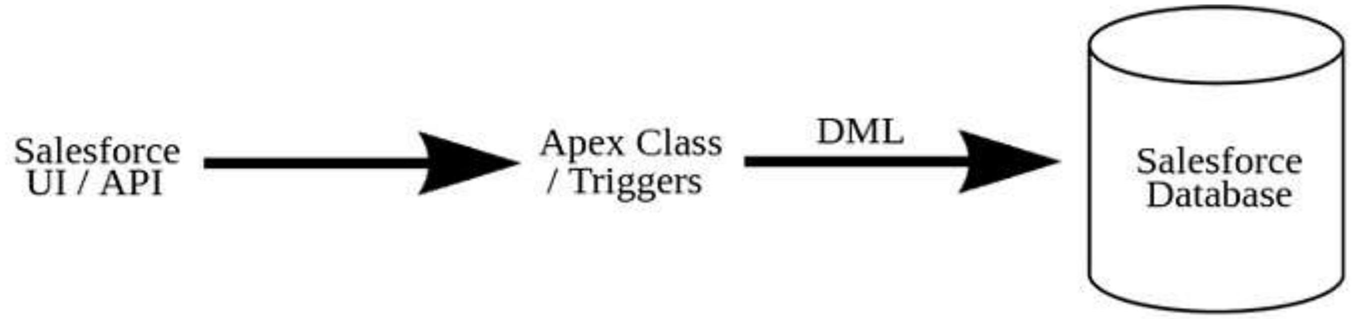
Execution Log

```
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

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
- Update
- Upsert
- Delete
- Undelete
- Merge

The upsert and merge statements are particular to Salesforce

Database structure

DML Statements:

- Insert - The insert operation inserts new records into the database. Using the Insert DML statement, you can create records of any Standard or Custom object.

- Update- Update operations are used to update existing records stored in the database. You have to query the existing records from the database to perform an update operation on them.

- Upsert - Upsert statements allow you to insert new records or update existing records in a single call. Upsert statements determine whether a record already exists or not by using the record’s ID.

- Delete- Delete operations are performed on the existing records in the database. Deleted records are not deleted permanently from salesforce; they are stored for 15 days in the recycle bin; from there they can be restored within 15 days.

- Undelete- When you delete the records, they are stored in the recycle bin for 15 days and after 15 days they are deleted permanently from the salesforce. The undelete statement is used to restore records that have been deleted but are still in the Recycle Bin.

- Merge- The merge statement merges up to three records of the same sObject type into one of the records, deleting the others, and re-parenting any related records.

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

Adding New Branch:

```
Branch__c br = new Branch__c (BName__c ='Data Science');
 insert br;                                    // Insert the branch by using DML statement
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 for the new Branch:

```
Branch__c br = new Branch__c ( BName__c ='Data Science');
    insert br;                                   // Insert the branch by using DML
Student__c st = new Student__c (SName__c = ‘NEW Student', RelBId__c=br.Id);
    Insert st;
```

Adding Student Record for the existing Branch :

```
List<Branch__c> BL=[Select Id, BName__c from Branch__c where BName__c ='CSE'];
ID x=BL[0].Id;
Student__c st = new Student__c(SName__c = 'StudentXY', RelBId__c=x);
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 BName__c from Branch__c where BName__c ='Information Technology'];

 branch.BName__c ='IT';

 update branch;                                //Update the branch by using DML
```

Delete Branch record:

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

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

delete branch;              //deleted the branch by using DML
```

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, BName__c  from Branch__c where BName__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:

This example shows how upsert **inserts a new branch and updates an existing branch name in branch record in one call.**

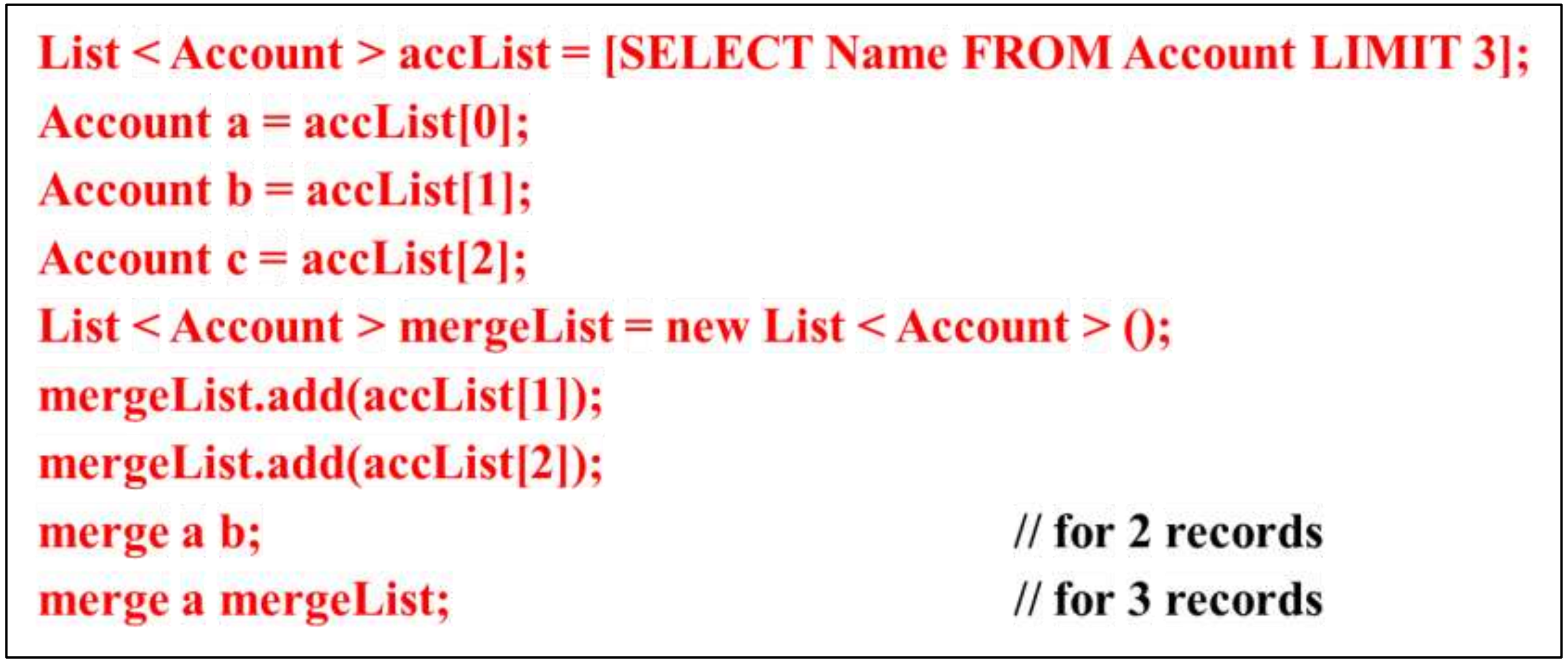
This upsert call **inserts a new Branch 'Data Science'** and **updates the existing Branch Name 'Information Technology to 'IT' .**

```
Branch__c br1 = new Branch__c (BName__c ='Data Science');      //new branch created

Branch__c br2=[Select BName__c from Branch__c where BName__c ='Information Technology'];

br2.BName__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.**

Apex Triggers - Introduction

What is Apex Triggers?

➢ A trigger is a **set of Apex code** that runs **before or after Data manipulation language (DML) events**.

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

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

➢ Just like database systems support triggers, Apex provides **trigger support for managing records.**

➢ Triggers in Apex are like **automated reactions** that happen in response to changes in your Salesforce data.

➢ Apex Triggers are specifically **designed to perform a complex series of actions** in Salesforce, **after and before some events**

➢ Triggers are a powerful tool for implementing complex business logic and **ensuring data integrity within your Salesforce organization.**

Apex Triggers - Introduction

When to use salesforce triggers

We should 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 workflow rules instead.**

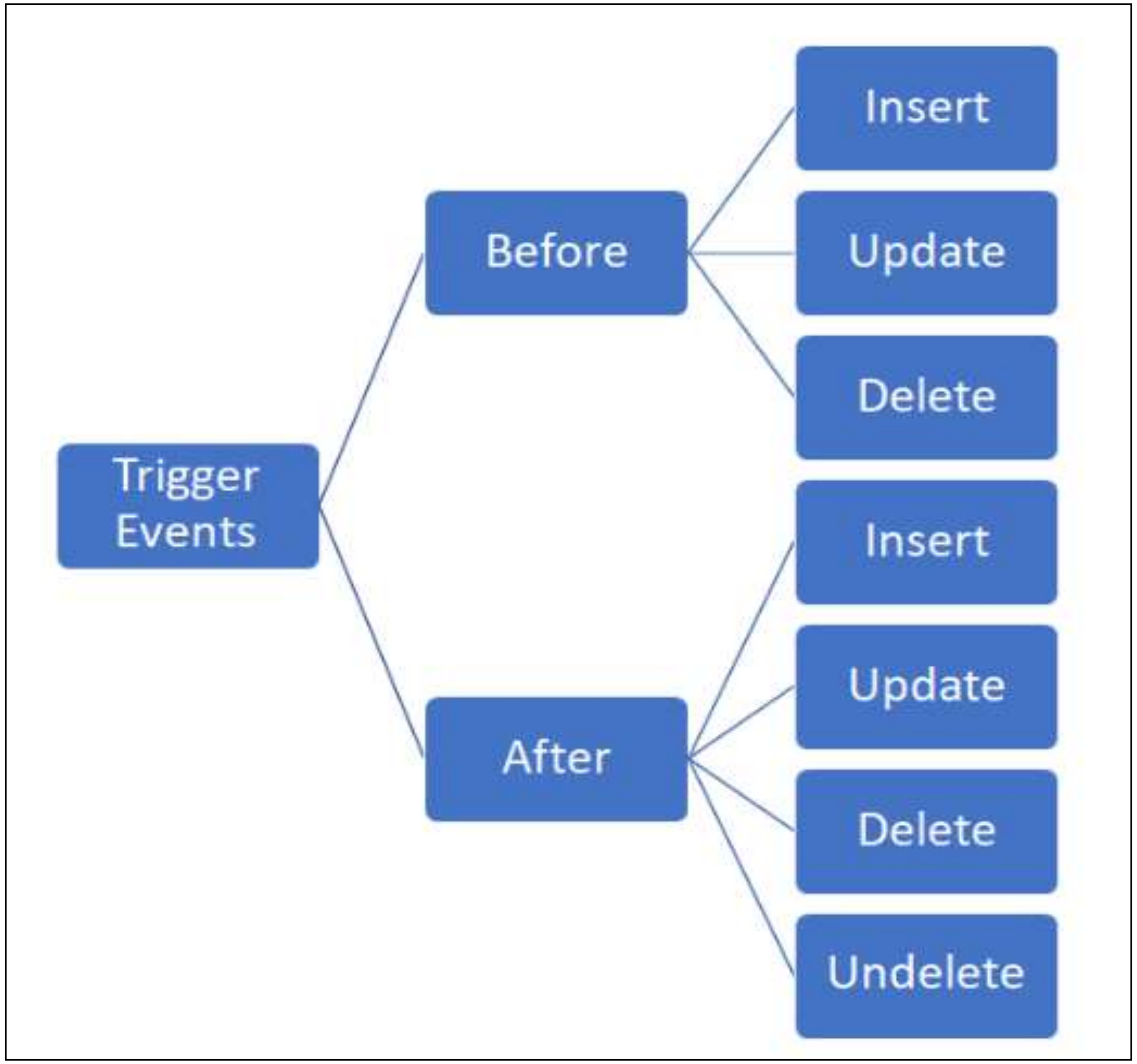
Trigger Syntax

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

A trigger is a set of statement which can be **executed based on the trigger_events.**

Apex Triggers - Introduction

Here is a **list of trigger events** in salesforce. We can **specify multiple trigger events** in a comma-separated list.



Apex Triggers - Introduction

Different type of Triggers

There are **two types** of triggers:

1. **Before triggers**

They are used to perform a task **before a record is inserted or updated or deleted**. These are used to **update or validate record values before they are saved to the database.**

2. **After triggers**

They are used if we **want to use the information-set by Salesforce system and to make changes in the other records.**

They 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.**

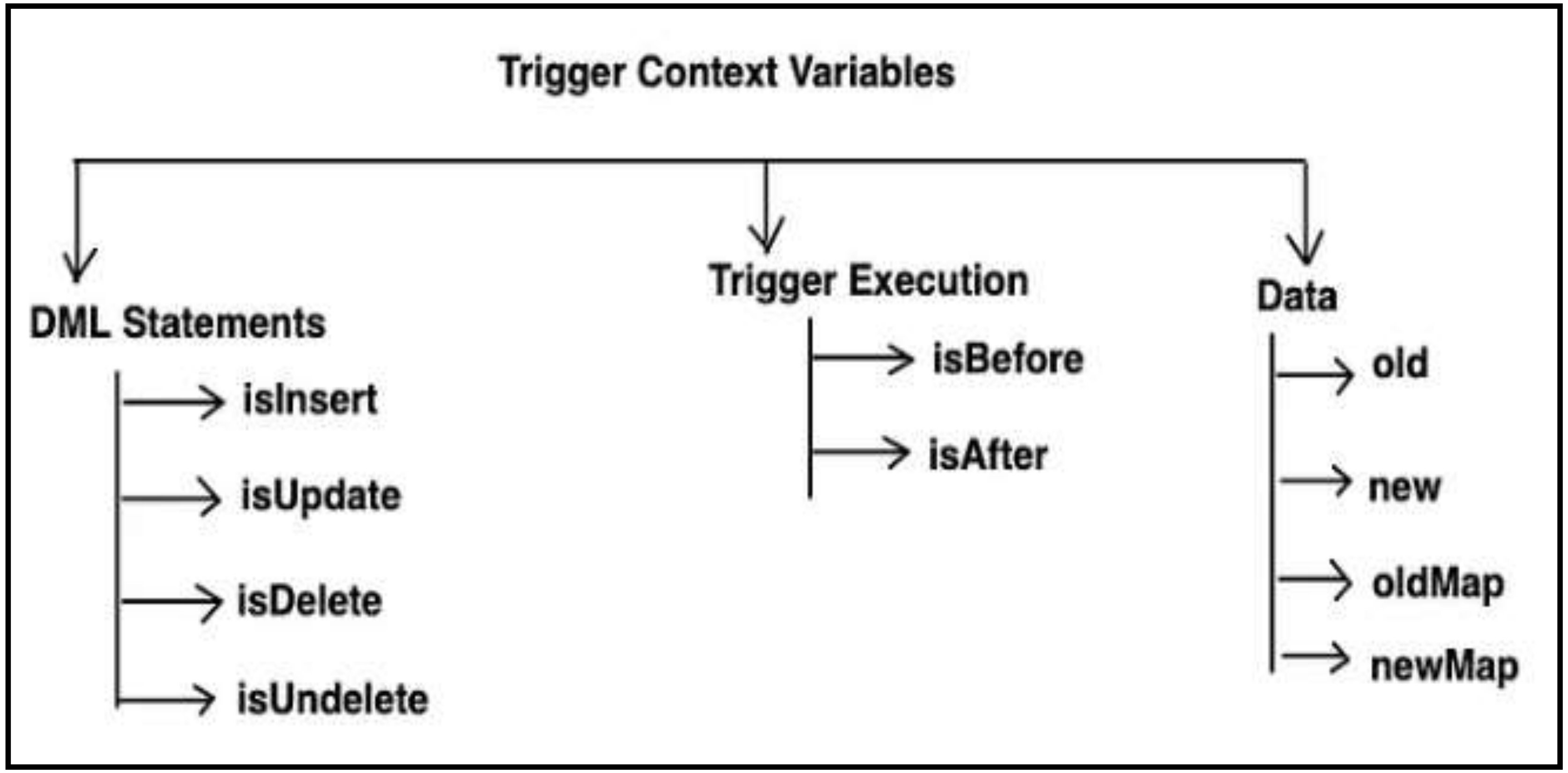
The records that fire the **after trigger are read-only.**

Apex Trigger Context Variable

- When ever a **trigger is invoked** it has **a group of supportive variables**. Which hold runtime information during the current trigger execution and these variables are known as Trigger Context Variables (TCV).

- All triggers define implicit variables that **allow developers to access run-time context**.

- These variables are **contained in the System.Trigger class.**



Apex Trigger Context Variable

Context Variable	Usage
isExecuting	Returns true if the current context for the Apex code is a trigger, not a Visualforce page, a Web service, or an executeanonymous() API call.
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**.
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.

Apex Trigger Context Variable

Context Variable	Usage
isUndelete	Returns **true** if this trigger was fired after a **record is recovered from the Recycle Bin**. This recovery can occur after an undelete operation from the Salesforce user interface, Apex, or the API.
new	Returns **a list of the new versions of the sObject records**. This sObject list is only available in **insert, update, and undelete triggers**, and the records can only be modified in before triggers.
old	Returns **a list of the old versions of the sObject records**. This sObject list is only available in **update and delete triggers**.
size	The **total number of records** in a trigger invocation, both old and new.
oldMap	A map of IDs to the old versions of the sObject records. This map is only available in update and delete triggers.
newMap	A map of IDs to the new versions of the sObject records. This map is only available in before update, after insert, after update, and after undelete triggers.

Apex Triggers - Introduction

EXAMPLE

trigger Trigger1 on Account (before insert, after insert, after update, before update, after Delete, before Delete, After undelete)

```
{
   if(trigger. isBefore)
   {
        if(trigger.isInsert){
               system.debug('<--- BEFORE INSERT --->');
     }
      if(trigger.isupdate)
   {
              system.debug('Before Update-->');
              system.debug('Before Record-->'+trigger.old);
    }
      if(trigger.isDelete)
    {
              system.debug('Before DELETE-->');
    }
 }
if(trigger.isAfter)
{
       if(trigger.isInsert){
            system.debug('<--- AFTER INSERT --->');
        }
        if(trigger.isupdate){
              system.debug('After Update-->');
              system.debug('After Record-->'+trigger.new);
       }
      if(trigger.isDelete){
              system.debug('AFTER DELETE-->');
       }
      if(trigger.isunDelete){
               system.debug('Restored Record-->');
               system.debug('Restored Record-->'+trigger.new);
     }
}
    }
```

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.

Apex Trigger – AccountTrigger.apxt

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

Apex Class – AccountClass.apxc

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

Apex Triggers - Introduction

Calling a Class Method from a Trigger:

Example

Before Insert & After Insert Trigger in custom object:

Apex Trigger – TeacherTigger.apxt

```
trigger TeacherTrigger on Teacher__c (Before insert,After insert) {
 if (Trigger.isInsert) {
       if (Trigger.isBefore) {
                   TeacherClass.beforeInsert(Trigger.new); }
       if (Trigger.isAfter) {
                   TeacherClass.afterInsert(Trigger.new); }
 }
}
```

Apex Triggers - Introduction

Calling a Class Method from a Trigger:

Apex Class – TeacherClass.apxc

```
public class TeacherClass {
 public static void beforeInsert(List<Teacher__c> newList)  {
   for(Teacher__c a: newList) {
            if(a.TName__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(BName__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.

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.**

This example **(MyTriggerBulk)**.

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:

- The following example **(SoqlTriggerBulk)** 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
   }   }
```