Session 2

Conditional Statements,Looping Statements,
Arrays in Apex,Class,Objects,Methods

Conditional Statements in Apex:

The programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

The Conditional Statements in Apex works similar to java.

1. if Statement : if the condition is true then subsequent code will get execute otherwise not.

Syntax:

```
if(condition){
//if condition true this block will execute
}
```

Example:

```
Integer i=5;
if(i==5)
{
   System.debug('value of i=5');
}
```

2. if else statement : if the condition is true then if block will execute otherwise it will jump to else block.

Syntax:

```
if(condition){
//if condition is true this block will execute
}
else{
//if condition false then else block will execute
}
```

Example:

```
Integer i=7;
if(i==5)
{
   System.debug('value of i=5');
}
else{
   System.debug('value of i is not 5');
}
```

3. else if statement: multiple if conditions are involved depending upon that if condition block will execute otherwise else block will execute.

```
Integer place=2;
String medal_color;
if (place == 1) {
    medal_color = 'gold';
    } else if (place == 2) {
    medal_color = 'silver';
    } else if (place == 3) {
    medal_color = 'bronze';
    } else {
    medal_color = null;
    }
System.debug(medal_color);
```

4)Switch Statements

Apex provides a switch statement that tests whether an expression matches one of several values and branches accordingly.

The syntax is:

```
switch on expression {
    when value1 {         // when block 1
     // code block 1
   }
    when value2 {              // when block 2
      // code block 2
   }
 when value3 {            // when block 3
       // code block 3
   }    when else {               // default block, optional
  // code block 4
 }
}
```

Example: null value example

```
Integer i;
switch on i {
   when 2 {
      System.debug('when block 2');
   }
   when null {
      System.debug('bad integer');
   }
   when else {
      System.debug('default ' + i);
   }
```

```
}
```

Example : when i variable initializes to a value.

```
Integer i=2;
switch on i {
   when 2 {
      System.debug('when block 2');
   }
   when null {
      System.debug('bad integer');
   }
   when else {
      System.debug('default ' + i);
   }
}
```

Multiple Values Examples

The Apex `switch` statement doesn't fall-through, but a `when` clause can include multiple literal values to match against. You can also nest Apex `switch` statements to provide multiple execution paths within a `when` clause.

```
switch on i {
when 2, 3, 4 {
      System.debug('when block 2 and 3 and 4');
  }
   when 5, 6 {
      System.debug('when block 5 and 6');
 }   when 7 {
     System.debug('when block 7');
  }
   when else {
     System.debug('default');   }
}
```

Note: To club multiple conditions together, logical operators (&&, ||, !) operators are used.

Iterative / Looping Statements:

1. while :

A while loop repeatedly executes a block of code as long as a Boolean condition specified in the while statement remains true.

Syntax:

```
        while(condition)
        //block of while
        }
```

Example:

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

2. do while : The Apex do-while loop repeatedly executes block of code as long as particular boolean condition remains true.

It syntax is:

```
do
{
//code block
}while(condition);
```

curley braces are always required around a code block.

Example:

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

3. for loop:

There are three types of for loops. The first type of for loop is a traditional loop that iterates by setting a variable to a value, checking a condition, and performing some action on the variable.

The traditional for loop in Apex corresponds to the traditional syntax used in Java and other languages. Its syntax is:

```
for (init_stmt; exit_condition; increment_stmt) {
//    code_block
}
```

When executing this type of for loop, the Apex runtime engine performs the following steps, in order:

1. Execute the *init_stmt* component of the loop. Note that multiple variables can be declared and/or initialized in this statement.
2. Perform the *exit_condition* check. If true, the loop continues. If false, the loop exits.
3. Execute the *code_block*.
4. Execute the *increment_stmt* statement.
5. Return to Step 2.

As an example, the following code outputs the numbers 1 - 10 into the debug log. Note that an additional initialization variable, `j`, is included to demonstrate the syntax:

```
for (Integer i = 0, j = 0; i < 10; i++) {
  System.debug(i+1);
}
```

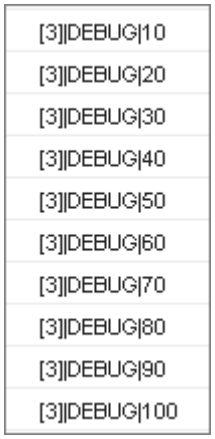
for-each (Enhanced for loop)

A second type of for loop is available for iterating over a list or a set.

Execute the following code:

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

The previous example iterates over every integer in the list and writes it to the output.



The list or set iteration for loop:

```
for (variable : list_or_set) {
//  code_block
}
```

where variable must be of the same primitive or sObject type as *list_or_set*.

The SOQL for loop:

```
for (variable : [soql_query]) {
// code_block
}
```

or

```
for (variable_list : [soql_query])
{
 //  code_block
}
```

Constants:

Apex constants are variables whose values don’t change after being initialized once. Constants can be defined using the final keyword

The final keyword means that the variable can be assigned at most once, either in the declaration itself, or with a static initializer method if the constant is defined in a class.

Consider a **CustomerOperationClass** class and a constant variable **regularCustomerDiscount** inside it −

```
public class CustomerOperationClass {
   static final Double regularCustomerDiscount = 0.1;
   static Double finalPrice = 0;

   public static Double provideDiscount (Integer price) {
      //calculate the discount
      finalPrice = price - price * regularCustomerDiscount;
      return finalPrice;
   }
}
```

To see the Output of the above class, you have to execute the following code in the Developer Console Anonymous Window −

```
Double finalPrice = CustomerOperationClass.provideDiscount(100);
System.debug('finalPrice '+finalPrice);
```

Static variables and Methods:

Static variable are global variable which are not related to particular object.

- Static variable are initialized only once.
- You can call a static variable directly by class name, similar goes for static methods.
- Memory is allocated to static variable only once whereas for non static variable memory is allocated every time an instance is created for class.
- Static variable do not occupy memory in view state.
- We can use static variables and methods only with outer class they are not allowed with inner class.

Example:

```
public class TestStaticClass {
    public static string testVariable='First';
public static string testVariable1='Second';

public static void method1()
{
system.debug('In method 1');
// Using testVariable in method1
testVariable1=TestStaticClass.testVariable;

}

public static void method2()
{
system.debug('In method 2');
// Calling method1 in method2
TestStaticClass.method1();
    System.Debug('test variable1='+testVariable);
   System.Debug('test variable1='+testVariable1);

}
}
```

Execution:

```
TestStaticClass.method1();  //valid
TestStaticClass.method2();  //valid
TestStaticClass obj=new TestStaticClass();
obj.testVariable1;  //invalid
obj.method1();   //invalid
obj.method2();  //invalid
```

Difference between static and non static:

	Static (variables,methods)	Instance (variables,methods)
Association	Class based	Object based
Replication	Once	One copy per instance
Initialization	Class is loaded	Instance in created
Syntax	Myclass.staticmembername();	Myclass m1=new Myclass();

Array:

Array can hold collection of similar elements. Each element in the array is located by index. Index value starts with zero. So the first element occupies the index zero.

- All elements are stored in contigueous memory location.
- Array is created at runtime.
- Array can be of Primitive datatype or sObject datatype.

Syntax:

datatype[] arrayName; // the square brackets [] is the notation used to declare an array

Declaration and initialization of Array:

```
String[] countries = new String[]{'India', 'USA', 'Russia'};
```

Example:

```
Integer[] element = new Integer[5];
Account[] acc = new Account[5];
```

- In case of primitive type, fixed size of memory is allocated as primitive type has fixed memory size.
- In case of sObject type, fixed size of memory is not allocated as sObject type has no fixed memory size. It depends upon number of fields taken and assigned value to it.
- Array cannot be created if size is unknown.
- Array has a method, "size()" which returns size of an array.

Examples:

Integer[] ages; // 30, 40, 50..

String[] names; // ‘suresh’, ‘ramesh’

Account[] accs;

Position__c[] pos; // position__c is a custom object for an example

The above statements will define the array with null values. The memory has to be allocated using the keyword “new” as below.

The size of array can be set using the below syntax.

```
Integer[] ages = new Integer[4];  // 4 denotes the size of your array
String[]  names;
names = new String[10];      // 10 elements can be stored in this array
```

Array Example;

```
Integer[] a=new Integer[]{30,40,25,50,32};

   for(Integer i=0;i<a.size();i++){

   System.debug('array elements='+a[i]);

}
```

Hence, in memory the elements are placed in the below format and each element can be located by index.

```
_____________________
0   |  1   | 2    |  3  | 4  |
_____________________
30 | 40 |  25 | 50 | 32 |
_____________________
```

Display each element using “for loop”:

```
for(Integer i = 0; i < 5; i++){
system.debug(‘i = ‘ + ages[i]);
}
```

Array can be initialized while declaring itself as below

```
//Initializing Array
Integer[] RollNos = new Integer[]{1001, 3003, 2002, 4004};
for(Integer i = 0; i < 4; i++){
system.debug(‘RollNos = ‘ + RollNos[i]);
}
```

creating array of sObjects.

```
Account a1 = new Account(Name=’Test Account1′, Industry=’Education’);
Account a2 = new Account(Name=’Test Account2′, Industry=’Training’);
Account[] accArray = new Account[]{a1, a2 };

system.debug(accArray);
```

forEach loop in apex to iterate collections.

syntax:

```
for(dataType variable: array){

}
```

```
//Example 1
Integer[] arr=new Integer[]{10,20,30,40,50};

  for(Integer a:arr)

{

   System.debug('arr='+a);

}//size method to get the size of an array
for(Integer i = 0; i < ages.size(); i++){  //size() – to get the size of array
system.debug(‘i = ‘ + ages[i]);
}
//Example 2
String[] names = new String[]{‘Suresh,’Ramesh’,’Rajesh’};
for(String s:names){
system.debug(s);
}
```

Note: Arrays of arrays are not allowed. So array can only be an one-dimensional. In Apex, the array is a notation used to declare single dimensional list. List is a type of collection type in apex.

Access Modifiers in Apex:

Apex allows you to use the private,public,protected and global access modifiers when defining methods and variables.

By default, a method or variable is visible only to the Apex code *within the defining class*. You must explicitly specify a method or variable as public in order for it to be available to other classes in the same application namespace

- Apex provides below access specifiers:

1. private

- private members can be accessed within class only.
- We cannot access private members from outside the class.
- private access specifier can be applied for both, class and class members.
- It is default access specifier in Apex.

2. public

- public members can be accessed within the class, as well as outside the class within Salesforce org.

3. protected

- protected access specifier is used in "Inheritance".
- protected members are accessible within the class and its subclass.
- Other classes which are not subclasses of class, cannot access protected members.

4. global

- global members are accessible within the Salesforce org and outside the Salesforce org also.
- Batch Apex classes, Schedule Apex classes, Webservices and API's to be defined with global access specifier.

> Note:it is recommend using the `global` access modifier rarely, if at all. Cross-application dependencies are difficult to maintain.
>
> Syntax:
> [(none)|private|protected|public|global] declaration

For example:

```
// private variable s1
private string s1 = 'Test String';
// public method get()
public string get() {    ... }
```

Classes:

A class is a template or blueprint from which objects are created. An object is an instance of a class.

Class has logical existance, object has physical existance. We cant make class as static.

All objects have state and behavior, that is, things that an object knows about itself, and things that an object can do.

A class can contain variables and methods. Variables are used to specify the state of an object, such as the object's `Name` or `Type`. Since these variables are associated with a class and are members of it, they are commonly referred to as member variables. Methods are used to control behavior.

A class can contain other classes, exception types, and initialization code.

Syntax:

private | public | global

[virtual | abstract | with sharing | without sharing]

class ClassName [implements InterfaceNameList] [extends ClassName]

{

// The body of the class

}

Example:

```
public class MyClass{
//MyClass code goes here
}
```

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

The `private` access modifier declares that this class is only known locally, that is, only by this section of code. This is the default access for inner classes—that is, if you don't specify an access modifier for an inner class, it is considered `private`. This keyword can only be used with inner classes (or with top level test classes marked with the `@isTest` annotation).

- The `public` access modifier declares that this class is visible in your application or namespace.
- The `global` access modifier declares that this class is known by all Apex code everywhere. All classes that contain methods defined with the `webservice` keyword must be declared as `global`. If a method or inner class is declared as `global`, the outer, top-level class must also be defined as `global`.
- 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`.
- The `abstract` definition modifier declares that this class contains abstract methods, that is, methods that only have their signature declared and no body defined.

A class can implement multiple interfaces, but only extend one existing class. This restriction means that Apex does not support multiple inheritance. The interface names in the list are separated by commas.

Class Variables:

To declare a variable, specify the following:

- Optional: Modifiers, such as `public` or `final`, as well as `static`.
- Required: The data type of the variable, such as String or Boolean.
- Required: The name of the variable.
- Optional: The value of the variable.

Use the following syntax when defining a variable:

[public | private | protected | global] [final] [static] data_type variable_name [= value]
For example:

private static final Integer MY_INT; private final Integer i = 1;

Class Methods:

To define a method, specify the following:

- Optional: Modifiers, such as `public` or `protected`.
- Required: The data type of the value returned by the method, such as String or Integer. Use `void` if the method does not return a value.
- Required: A list of input parameters for the method, separated by commas, each preceded by its data type, and enclosed in parentheses `()`. If there are no parameters, use a set of empty parentheses. A method can only have 32 input parameters.
- Required: The body of the method, enclosed in braces `{}`. All the code for the method, including any local variable declarations, is contained here.

Use the following syntax when defining a method:

[public | private | protected | global] [override] [static] data_type method_name (input parameters)
{// The body of the method}

For Example:

```
 public static Integer getInt() {
    return MY_INT;
}
```

Constructors:

A constructor is code that is invoked when an object is created from the class blueprint. You do not need to write a constructor for every class. If a class does not have a user-defined constructor, a default, no-argument, public constructor is used.

Constructor define size of the class.

Constructor can not be static.

The syntax for a constructor is similar to a method, but it differs from a method definition in that it never has an explicit return type and it is not inherited by the object created from it.

After you write the constructor for a class, you must use the new keyword in order to instantiate an object from that class, using that constructor.

For example, using the following class:

```
public class TestObject {    // The no argument constructor
   public TestObject() {
// more code here  }}
```

A new object of this type can be instantiated with the following code:

```
TestObject myTest = new TestObject();
```

If you write a constructor that takes arguments, you can then use that constructor to create an object using those arguments.

If you create a constructor that takes arguments, and you still want to use a no-argument constructor, you must create your own no-argument constructor in your code. Once you create a constructor for a class, you no longer have access to the default, no-argument public constructor.

In Apex, a constructor can be overloaded, that is, there can be more than one constructor for a class, each having different parameters.

```
public class ConstructorDemo {
    public ConstructorDemo(){

        System.debug('you are in default constructor');
    }
    public ConstructorDemo(Integer i)
    {
        System.debug('value of i='+i);
    }
}
```

Execution:

Constructordemo obj=new ConstructorDemo();

ConstructorDemo obj2=new ConstructorDemo(10);

The following example illustrates a class with two constructors: one with no arguments and one that takes a simple Integer argument.

It also illustrates how one constructor calls another constructor using the this(...) syntax, also know as constructor chaining.

```
public class TestObject2 {
private static final Integer DEFAULT_SIZE = 10;
Integer size;
 public TestObject2() {
       this(DEFAULT_SIZE);
     }
 public TestObject2(Integer ObjectSize) {
     size = ObjectSize;
System.debug('size='+size);
     }
}
```

New objects of this type can be instantiated with the following code:

```
 TestObject2 myObject1 = new TestObject2(42);
TestObject2 myObject2 = new TestObject2();
```