Session 3
Extending Class,Interface,Collection

Extending a class:

Inheritance in Apex is a mechanism by which one object acquires all the properties and behaviors of parent object.
The idea behind inheritance in Apex is that you can create new classes that are built upon existing classes. Whenever you inherit from an existing class, you can reuse methods and fields of parent / super class, as well as you can add new methods and fields also based on the requirements.

Terms used in Inheritance:

- **Class:** A class is a concept/prototype/template of common properties, from which objects are created.
- **Child Class/Sub Class:** Child class is a class which inherits the other class, known as child class. It is also called extended class, derived class, or sub class.
- **Parent Class/Super Class:** Parent class is the class from which a subclass inherits the features. It is also called a Super class or a base class.
- **Virtual:** The virtual modifier declares that this class allows for extension and overrides.
- **Override:** You cannot override a method with the override keyword in the child class unless the class and method have been defined as virtual.

```
Public virtual class parent{

//declaring private variables

public Integer var1=20,var2=40;

public void display(){

System.debug(‘display method of parent’);

}

public class child extends parent{

public void show(){

System.debug(‘value of var1=’+var1);

System.debug(‘value of var2=’+var2);

System.debug(‘this is from child method’);

}

execution:

child c=new child();

c.display();
```

```
c.show();
```

A class that extends another class inherits all the methods and properties of the extended class. In addition, the extending class can override the existing virtual methods by using the override keyword in the method definition.

Overriding a virtual method allows you to provide a different implementation for an existing method. This means that the behavior of a particular method is different based on the object you're calling it on. This is referred to as polymorphism.

A class extends another class using the extends keyword in the class definition. A class can only extend one other class, but it can implement more than one interface.

This example shows how the YellowMarker class extends the Marker class. To run the inheritance examples in this section, first create the Marker class.

```
public virtual class Marker {
   public virtual void write() {
     System.debug('Writing some text.');
  }
   public virtual Double discount() {
       return .05;
    }
}
```

Then create the YellowMarker class, which extends the Marker class.

```
// Extension for the Marker class

public class YellowMarker extends Marker {
    public override void write() {
        System.debug('Writing some text using the yellow marker.');
   }
 }
```

Execution:

```
Marker obj1, obj2;

obj1 = new Marker();

obj1.write(); // This outputs 'Writing some text.'

 obj2 = new YellowMarker();

obj2.write();   // This outputs 'Writing some text using the yellow marker.'

Double d = obj2.discount(); // We get the discount method for free// and can call it from the YellowMarker instance.
```

This code segment shows polymorphism. The example declares two objects of the same type (Marker). Even though both objects are markers, the second object is assigned to an instance of the YellowMarker class.

Hence, calling the write method on it yields a different result than calling this method on the first object, because this method has been overridden. However, you can call the discount method on the second object even though this method isn't part of the YellowMarker class definition. But it is part of the extended class, and hence, is available to the extending class, YellowMarker.

The extending class can have more method definitions that aren't common with the original extended class. For example, the RedMarker class below extends the Marker class and has one extra method, computePrice, that isn't available for the Marker class. To call the extra methods, the object type must be the extending class.

```
// Extension for the Marker class

public class RedMarker extends Marker {
public override void write() {
        System.debug('Writing some text in red.');
   }
   // Method only in this class
   public Double computePrice() {
 return 1.5;
  }
}
```

This snippet shows how to call the additional method on the RedMarker class.

```
RedMarker obj = new RedMarker();// Call method specific to RedMarker only
obj.write();
Double price = obj.computePrice();
```

Extensions also apply to interfaces—an interface can extend another interface. As with classes, when an interface extends another interface, all the methods and properties of the extended interface are available to the extending interface.

Method Overloading:

If a class have multiple methods with same name but different parameters (either different length of arguments or different data types of the arguments), is knwon as method overloading.

Example:

```
public class MethodOverloadingDemo {
   public void sum(Integer no1,Integer no2)
   {
      System.debug('Sum of two integers is='+(no1+no2));

   }
```

```
    public void sum(Decimal no1,Decimal no2){
        System.debug('Sum of two Decimal is='+(no1+no2));
    }
}
```

Execution:

```
MethodOverloading obj=new MethodOverloading();
obj.sum(10,20);
obj.sum(10.5,20.5);
```

Method overriding:

If a subclass (child class) has the same method as declared in the parent class, it is known as method overriding in Apex.

In other words, we can say If a subclass provides the specific implementation of the method that has been provided by one of its parent class, it is known as method overriding.

Note: Always make sure to achieve method overriding the scope must be of parent and child class relationship and you must have same name method needs to declare into child class with same signature of the method from Parent class to achieve method overriding in Apex.

Usage of Apex Method Overriding

Method overriding is used to provide specific implementation of a method that is already provided by its Parent/Super class.

Rules for Apex Method Overriding

- Method name must be same as in the parent class.
- Method name must have same parameter as in the parent class's method.
- Must be inheritance (IS-A relationship).

Example:

```
Pubic virtual class parent{
public virtual void display{
System.debug('this is parent');
}
}

public class child extends parent{
public override void display{
System.debug('this is child  overridden method');
}
```

Execution:

Child ch=new Child();

ch.display();

or

Parent ch=new Child();

ch.display();

Interfaces

An interface is like a class in which none of the methods have been implemented—the method signatures are there, but the body of each method is empty. To use an interface, another class must implement it by providing a body for all of the methods contained in the interface.

Interfaces can provide a layer of abstraction to your code. They separate the specific implementation of a method from the declaration for that method. This way you can have different implementations of a method based on your specific application.

Defining an interface is similar to defining a new class. For example, a company might have two types of purchase orders, ones that come from customers, and others that come from their employees. Both are a type of purchase order. Suppose you needed a method to provide a discount. The amount of the discount can depend on the type of purchase order.

You can model the general concept of a purchase order as an interface and have specific implementations for customers and employees. In the following example the focus is only on the discount aspect of a purchase order.

Here is the definition of the PurchaseOrder interface.

```
// An interface that defines what a purchase order looks like in general
public interface PurchaseOrder {
 Double discount();
}
```

```
public class CustomerPurchaseOrder implements PurchaseOrder {
public Double discount() {
       return .05;  // Flat 5% discount
 }
}
```

This class implements the PurchaseOrder interface for employee purchase orders.

```
// Another implementation of the interface for employees

public class EmployeePurchaseOrder implements PurchaseOrder {
    public Double discount() {
       return .10;  // It’s worth it being an employee! 10% discount      } }
```

Note the following about the above example:

- The interface PurchaseOrder is defined as a general prototype. Methods defined within an interface have no access modifiers and contain just their signature.
- The Customer PurchaseOrder class implements this interface; therefore, it must provide a definition for the discount method. Any class that implements an interface must define all the methods contained in the interface.

Example:

```
public interface Animal {
   void run();
   void eat();
   void sleep();

}

public class Cat implements Animal{
   public void eat()
   {
      System.debug('inside cat class eat method');
   }
   public void sleep()
   {
      System.debug('inside cat class sleep method');
   }
  public void run()
   {
      System.debug('inside cat class run method');
   }
  }

public class Tiger implements Animal{
   public void eat()
   {
      System.debug('inside tiger class eat method');
   }
   public void sleep()
   {
      System.debug('inside tiger class sleep method');
   }
  public void run()
   {
      System.debug('inside tiger class run method');
   }
   public void roar()
   {
      System.debug('inside roar method');
   }
   }
```

Execution:

```
Animal a=new Tiger();
```

```
a.eat();
a.sleep();
a.run();
a.roar(); //will throw error
```

Abstract Class:

To create abstract class, we need to use abstract definition modifier.

Allow to extend the child class.

Abstract class can contain methods signed as abstract, to clarify, it is a method that has only a signature (body is not defined).

Child class **must** implement all methods declared as abstract!
Abstract class can also include other methods, which have the logic. To allow child class access those methods use protected or public keyword.

Cannot be initialize directly: new TestAbstractClass();
Abstract class can contain both virtual and abstract methods.

Abstract class is some kind of “partial” class. Therefore, some methods are implemented, some needs to be implemented by child class.

virtual methods can be override, but this is not mandatory.

```
    ublic abstract class AbstMyClass {
  public void method1(){              // Defined and Implemented
    System.debug('I am in abstract class m1');
    }
    public void method2(){                // Defined and Implemented
   System.debug('I am in method2');
   }
public abstract void method3();   //declared
}
public class AbstractChild extends AbstMyClass{
  public override void method3(){
    System.debug('i am in abstract child method3');
  }
}
```

Collection:

- Collection supports dynamic memory allocation, which provides efficient memory utilization.
- Collection can grow or shrink at runtime, depends upon number of elements added or removed from collection.
- Collection provides utility methods to manage and manipulate elements stored in collection.

Apex provides three collection classes:

1. List
2. Set
3. Map

List:

- List is an ordered collection. ie, it preserves insertion order.
- List can store primitive type, SObject type, collection type and user defined type elements.
- List allows to store duplicate elements.
- Homogeneous and heterogeneous elements can be stored in List.
- The elements stored in List, can be accessed by index position, which starts from zero.
- Syntax: List<datatype> var = new List<datatype>();

Example:

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

// Holds a Collection of integer elements.

 List<string> countryNames = new List<String>();
                              |
  //Holds a Collection of String elements

       List<ID> recordIds = new List<ID>();
                          |
   //Holds a collection of record id's.

       List<Account>  lstAccounts = new List<Account>();
                                    |
    //Holds a Collection of Account Records.

       List<Case> lstCases = new List<Case>();
                             |
      // Holds a Collection of Case Records.

       List<Position__C> lstPositions = new List<Position__C>();
                                        |
       //Holds a Collection of Position Records
```

Declare and initialize list:

```
List<String> groceries=new List<String>{‘Tea’};

List<String> groceries=new List<String>;
```

groceries.add(‘Coffee’);

```
List<String> groceries = new List<String>();
//The output for this statement is null
System.debug('Groceries: ' + groceries);
//Use the add method to add an element to the list
        groceries.add('Tea');
        groceries.add('Sugar');
        groceries.add('Honey');
        groceries.add(2, 'Milk');

        //The output for this statement is Tea, Sugar, Milk, Honey
        System.debug('Groceries: ' + groceries);
```

Example 2:

```
String[] groceries = new String[4];
System.debug('Groceries: ' + groceries);
groceries.add('Tea');
groceries.add('Sugar');
 System.debug('Added 2 items: ' + groceries);
 groceries.add(1, 'Honey');
System.debug('Added Honey in index 1: ' + groceries);
System.debug('Item 0: ' + groceries[0]);
```

Methods of List:

1. add(<element>):Used to add element in List collection. Element is inserted at the end of List collection.

```
List<String> lstElements = new List<String>();
lstElements.add(‘Apex’);
```

2. add(<int indexPosition>, <element>):Used to insert element at specified index position in List collection.

```
LstElements.add(1,’India’);
```

3. Integer size():It returns size of List collection.

```
system.debug('Collection Size is...: '+ lstElements.Size());
```

4. Boolean isEmpty():Used to check wheather List collection is empty or not.

Ex:

```
                if(lstElements.IsEmpty())
                system.debug('Collection is Empty.');
                        else
                system.debug('Collection is Not Empty.');
```

5. addAll(<collection>):Used to add one List collection of elements in current List collection.

```
string[] countryNames = new string[]{'USA','Australia','Japan','China','UK'};
                        lstElements.AddAll(countryNames);
```

6. get(<Integer index>):Used to return element stored at specified index position in List collection.
Ex:

```
system.debug('Element exist at the 2nd position is..: '+ lstElements.Get(2));
```

7. remove(<Integer indexPosition>): Used to remove element exist at specified index position from List collection.
Ex:

```
        lstElements.remove(2);
```

8. clear():Used to remove all elements from List collection.

```
LstElements.clear();
```

9. sort():Used to sort elements stored in List collection in ascending order.

```
LstElements.sort();
```

10. set(<Integer indexPosition>, <element>):Used to override element stored at specified index position by new element in List collection.

11. Integer indexOf(elementName): it returns the index position of specified elements in list.
Ex:

12: clone: it will create duplicate copy of collection

Ex:

```
        List<String> backupCopy = customerNames.Clone();
```

Set:

- Set is an unordered collection. ie, it doesn't preserve insertion order.
- Set can store primitive type, SObject type, collection type and user defined type elements.
- Set doesn't allow to store duplicate elements.
- Set maintains uniqueness of elements by using "Binary Comparison".
- Homogeneous and heterogeneous elements can be stored in Set.
- The elements stored in Set, cannot be accessed by index position as insertion order is not preserved.

- Syntax: Set<datatype> var = new Set<datatype>();

To declare a set, use the `Set` keyword followed by the primitive data type name within <> characters.
For example:

```
Set<String> myStringSet = new Set<String>();
```

The following example shows how to create a set with two hardcoded string values.

```
// Defines a new set with two elements
Set<String> set1 = new Set<String>{'New York', 'Paris'};
```

To access elements in a set, use the system methods provided by Apex. For example:

```
// Define a new set
Set<Integer> mySet = new Set<Integer>();
// Add two elements to the set
mySet.add(1);
mySet.add(3);
mySet.remove(1);
```

Methods of Set:

1. add(<element>):Used to add element in Set collection. Upon adding the element, Salesforce arranges elements in sorting order by default.

2. addAll(<collection>): Used to add one Set collection of elements in current Set collection.

3. Integer size(): It returns size of Set collection.

4. Boolean isEmpty(): Used to check whether Set collection is empty or not.

5. Boolean contains(<element>):Used to check wheather specified element is exist in Set collection or not.

6. remove(<element>):Used to remove specified element from Set collection.

7. removeAll(<Collection>): Used to specified Set collection of elements from current Set collection.

8. clear(): Used to remove all elements from Set collection.

Map:

- It is a collection where element is stored in key-value pair.
- Key must be unique, whereas value can be unique or duplicate.
- Map is unordered collection of elements. It doesn't preserve insertion order.
- Key and value can be of primitive type, SObject type, collection type and user defined type.

Example:

Country (Key)	'United States'	'Japan'	'France'	'England'	'India'
Currency (Value)	'Dollar'	'Yen'	'Euro'	'Pound'	'Rupee'

- Syntax:

Map<keyDataType, valueDataType> var = new Map<keyDataType, valueDataType>();

- Example: Map<ID, Account> accounts = new Map<ID, Account>();

Map<String, String> country_currencies = new Map<String, String>();

Map<ID, Set<String>> m = new Map<ID, Set<String>>();

Methods of Map:

1. put(<key>, <value>): Used to add new element in Map collection.

2. putAll(<MapCollection>): Used to add one Map collection in current Map collection.

3. Integer size(): It returns size of Map collection.

4. Boolean isEmpty():Used to check wheather Map collection is empty or not.

5. remove(<key>): Used to remove key-value pair element from Map collection, whose key is specified.

6. containsKey(<key>):Used to check wheather specified key is exist in Map collection or not.

7. get(<key>): Used to get the value which is associated with specified key.

8. Set<datatype> keySet(): It returns all keys exist in Map collection in the form of Set collection.

9. List<datatype> values(): It returns all values exist in Map collection in the form of List collection.

10. clear(): Used to remove all elements from Map collection.