Understanding the Data Types

The Apex language is strongly typed so every variable in Apex will be declared with the specific data type. All apex variables are initialized to null initially. It is always recommended for a developer to make sure that proper values are assigned to the variables. Otherwise such variables when used, will throw null pointer exceptions or any unhandled exceptions.

Apex supports the following data types −

* Primitive (Integer, Double, Long, Date, Datetime, String, ID, or Boolean)
* Collections (Lists, Sets and Maps) (To be covered in Chapter 6)
* sObject
* Enums
* Classes, Objects and Interfaces

## Primitive Data Types

In this section, we will discuss the Primitive Data Types supported by Apex.

### Integer

A 32-bit number that does not include any decimal point. The value range for this starts from -2,147,483,648 and the maximum value is up to 2,147,483,647.

**Example**

We want to declare a variable which will store the quantity of barrels which need to be shipped to the buyer of the chemical processing plant.

Integer barrelNumbers = 1000;

system.debug(' value of barrelNumbers variable: '+barrelNumbers);

The **System.debug()** function prints the value of variable so that we can use this to debug or to get to know what value the variable holds currently.

Paste the above code to the Developer console and click on Execute. Once the logs are generated, then it will show the value of variable "barrelNumbers" as 1000.

### Boolean

This variable can either be true, false or null. Many times, this type of variable can be used as flag in programming to identify if the particular condition is set or not set.

**Example**

If the Boolean shipmentDispatched is to be set as true, then it can be declared as −

Boolean shipmentDispatched;

shipmentDispatched = true;

System.debug('Value of shipmentDispatched '+shipmentDispatched);

### 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**

Consider the following example to understand how the Date variable works.

//ShipmentDate can be stored when shipment is dispatched.

Date ShipmentDate = date.today();

System.debug('ShipmentDate '+ShipmentDate);

### 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**

If the company revenue is to be stored, then we will use the data type as Long.

Long companyRevenue = 21474838973344648L;

system.debug('companyRevenue'+companyRevenue);

### Object

We can refer this as any data type which is supported in Apex. For example, Class variable can be object of that class, and the sObject generic type is also an object and similarly specific object type like Account is also an Object.

**Example**

Consider the following example to understand how the object variable works.

Account objAccount = new Account (Name = 'Test Chemical');

system.debug('Account value'+objAccount);

**Note** − You can create an object of predefined class as well, as given below −

//Class Name: MyApexClass

MyApexClass classObj = new MyApexClass();

This is the class object which will be used as class variable.

### String

String is any set of characters within single quotes. It does not have any limit for the number of characters. Here, the heap size will be used to determine the number of characters. This puts a curb on the monopoly of resources by the Apex program and also ensures that it does not get too large.

**Example**

String companyName = 'Abc International';

System.debug('Value companyName variable'+companyName);

### Time

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

### Blob

The Blob is a collection of Binary data which is stored as object. This will be used when we want to store the attachment in salesforce into a variable. This data type converts the attachments into a single object. If the blob is to be converted into a string, then we can make use of the toString and the valueOf methods for the same.

### sObject

This is a special data type in Salesforce. It is similar to a table in SQL and contains fields which are similar to columns in SQL. There are two types of sObjects – Standard and Custom.

For example, Account is a standard sObject and any other user-defined object (like Customer object that we created) is a Custom sObject.

**Example**

//Declaring an sObject variable of type Account

Account objAccount = new Account();

//Assignment of values to fields of sObjects

objAccount.Name = 'ABC Customer';

objAccount.Description = 'Test Account';

System.debug('objAccount variable value'+objAccount);

//Declaring an sObject for custom object APEX\_Invoice\_c

APEX\_Customer\_c objCustomer = new APEX\_Customer\_c();

//Assigning value to fields

objCustomer.APEX\_Customer\_Decscription\_c = 'Test Customer';

System.debug('value objCustomer'+objCustomer);

### Enum

Enum is an abstract data type that stores one value of a finite set of specified identifiers. You can use the keyword Enum to define an Enum. Enum can be used as any other data type in Salesforce.

**Example**

You can declare the possible names of Chemical Compound by executing the following code −

//Declaring enum for Chemical Compounds

public enum Compounds {HCL, H2SO4, NACL, HG}

Compounds objC = Compounds.HCL;

System.debug('objC value: '+objC);

# Apex - Variables

Java and Apex are similar in a lot of ways. Variable declaration in Java and Apex is also quite the same. We will discuss a few examples to understand how to declare local variables.

String productName = 'HCL';

Integer i = 0;

Set<string> setOfProducts = new Set<string>();

Map<id, string> mapOfProductIdToName = new Map<id, string>();

Note that all the variables are assigned with the value null.

**Declaring Variables**

You can declare the variables in Apex like String and Integer as follows −

String strName = 'My String'; //String variable declaration

Integer myInteger = 1; //Integer variable declaration

Boolean mtBoolean = true; //Boolean variable declaration

**Apex variables are Case-Insensitive**

This means that the code given below will throw an error since the variable 'm' has been declared two times and both will be treated as the same.

Integer m = 100;

for (Integer i = 0; i<10; i++) {

integer m = 1; //This statement will throw an error as m is being declared

again

System.debug('This code will throw error');

}

**Scope of Variables**

An Apex variable is valid from the point where it is declared in code. So it is not allowed to redefine the same variable again and in code block. Also, if you declare any variable in a method, then that variable scope will be limited to that particular method only. However, class variables can be accessed throughout the class.

**Example**

//Declare variable Products

List<string> Products = new List<strings>();

Products.add('HCL');

//You cannot declare this variable in this code clock or sub code block again

//If you do so then it will throw the error as the previous variable in scope

//Below statement will throw error if declared in same code block

List<string> Products = new List<strings>();

# Apex - Constants

In Apex, Constants are used when we want to define variables which should have constant value throughout the program execution. Apex constants are declared with the keyword 'final'.

## Example

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);

# Apex - Decision Making

Decision-making structures require that 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.

In this chapter, we will be studying the basic and advanced structure of decision-making and conditional statements in Apex. Decision-making is necessary to control the flow of execution when certain condition is met or not. Following is the general form of a typical decision-making structure found in most of the programming languages

# Apex - if statement

An **if** statement consists of a Boolean expression followed by one or more statements.

## Syntax

if boolean\_expression {

/\* statement(s) will execute if the boolean expression is true \*/

}

If the Boolean expression evaluates to true, then the block of code inside the if statement will be executed. If the Boolean expression evaluates to false, then the first set of code after the end of the if statement(after the closing curly brace) will be executed.

# Apex - if else statement

An **if** statement can be followed by an optional **else** statement, which executes when the Boolean expression is false.

## Syntax

if boolean\_expression {

/\* statement(s) will execute if the boolean expression is true \*/

} else {

/\* statement(s) will execute if the boolean expression is false \*/

}

If the Boolean expression evaluates to true, then the **if block of code** will be executed, otherwise else block of code will be executed.

# Apex - if elseif else statement

An **if** statement can be followed by an optional **else if...else** statement, which is very useful to test various conditions using single **if...else if** statement.

## Syntax

The syntax of an **if...else if...else** statement is as follows −

if boolean\_expression\_1 {

/\* Executes when the boolean expression 1 is true \*/

} else if boolean\_expression\_2 {

/\* Executes when the boolean expression 2 is true \*/

} else if boolean\_expression\_3 {

/\* Executes when the boolean expression 3 is true \*/

} else {

/\* Executes when the none of the above condition is true \*/

}

# Apex - Loops

Loops are used when a particular piece of code should be repeated with the desired number of iteration. Apex supports the standard traditional for loop as well as other advanced types of Loops. In this chapter, we will discuss in detail about the Loops in Apex.

A loop statement allows us to execute a statement or group of statements multiple times and following is the general from of a loop statement in most of the programming languages

# Apex - For Loop

A **for** loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times. Consider a business case wherein, we are required to process or update the 100 records in one go. This is where the Loop syntax helps and makes work easier.

## Syntax

for (variable : list\_or\_set) { code\_block }

## Example

Consider that we have an Invoice object which stores information of the daily invoices like CreatedDate, Status, etc. In this example, we will be fetching the invoices created today and have the status as Paid.

**Note** − Before executing this example, create at least one record in Invoice Object.

// Initializing the custom object records list to store the Invoice Records created today

List<apex\_invoice\_\_c> PaidInvoiceNumberList = new List<apex\_invoice\_\_c>();

// SOQL query which will fetch the invoice records which has been created today

PaidInvoiceNumberList = [SELECT Id,Name, APEX\_Status\_\_c FROM APEX\_Invoice\_\_c WHERE

CreatedDate = today];

// List to store the Invoice Number of Paid invoices

List<string> InvoiceNumberList = new List<string>();

// This loop will iterate on the List PaidInvoiceNumberList and will process each record

for (APEX\_Invoice\_\_c objInvoice: PaidInvoiceNumberList) {

// Condition to check the current record in context values

if (objInvoice.APEX\_Status\_\_c == 'Paid') {

// current record on which loop is iterating

System.debug('Value of Current Record on which Loop is iterating is'+objInvoice);

// if Status value is paid then it will the invoice number into List of String

InvoiceNumberList.add(objInvoice.Name);

}

}

System.debug('Value of InvoiceNumberList '+InvoiceNumberList);

# Apex - SOQL For Loop

This type of **for** loop is used when we do not want to create the List and directly iterate over the returned set of records of the SOQL query. We will study more about the SOQL query in subsequent chapters. For now, just remember that it returns the list of records and field as given in the query.

## Syntax

for (variable : [soql\_query]) { code\_block }

or

for (variable\_list : [soql\_query]) { code\_block }

One thing to note here is that the **variable\_list** or variable should always be of the same type as the records returned by the Query. In our example, it is of the same type as APEX\_Invoice\_c.

## Example

Consider the following **for loop** example using SOQL **for** loop.

// The same previous example using For SOQL Loop

List<apex\_invoice\_\_c> PaidInvoiceNumberList = new

List<apex\_invoice\_\_c>(); // initializing the custom object records list to store

// the Invoice Records

List<string> InvoiceNumberList = new List<string>();

// List to store the Invoice Number of Paid invoices

for (APEX\_Invoice\_\_c objInvoice: [SELECT Id,Name, APEX\_Status\_\_c FROM

APEX\_Invoice\_\_c WHERE CreatedDate = today]) {

// this loop will iterate and will process the each record returned by the Query

if (objInvoice.APEX\_Status\_\_c == 'Paid') {

// Condition to check the current record in context values

System.debug('Value of Current Record on which Loop is iterating is '+objInvoice);

//current record on which loop is iterating

InvoiceNumberList.add(objInvoice.Name);

// if Status value is paid then it will the invoice number into List of String

}

}

System.debug('Value of InvoiceNumberList with Invoice Name:'+InvoiceNumberList);

# Apex - Strings

String in Apex, as in any other programming language, is any set of characters with no character limit.

**Example**

String companyName = 'Abc International';

System.debug('Value companyName variable'+companyName);

## String Methods

String class in Salesforce has many methods. We will take a look at some of the most important and frequently used string methods in this chapter.

### contains

This method will return true if the given string contains the substring mentioned.

**Syntax**

public Boolean contains(String substring)

**Example**

String myProductName1 = 'HCL';

String myProductName2 = 'NAHCL';

Boolean result = myProductName2.contains(myProductName1);

System.debug('O/p will be true as it contains the String and Output is:'+result);

### equals

This method will return true if the given string and the string passed in the method have the same binary sequence of characters and they are not null. You can compare the SFDC record id as well using this method. This method is case-sensitive.

**Syntax**

public Boolean equals(Object string)

**Example**

String myString1 = 'MyString';

String myString2 = 'MyString';

Boolean result = myString2.equals(myString1);

System.debug('Value of Result will be true as they are same and Result is:'+result);

### equalsIgnoreCase

This method will return true if stringtoCompare has the same sequence of characters as the given string. However, this method is not case-sensitive.

**Syntax**

public Boolean equalsIgnoreCase(String stringtoCompare)

**Example**

The following code will return true as string characters and sequence are same, ignoring the case sensitivity.

String myString1 = 'MySTRING';

String myString2 = 'MyString';

Boolean result = myString2.equalsIgnoreCase(myString1);

System.debug('Value of Result will be true as they are same and Result is:'+result);

### remove

This method removes the string provided in stringToRemove from the given string. This is useful when you want to remove some specific characters from string and are not aware of the exact index of the characters to remove. This method is case sensitive and will not work if the same character sequence occurs but case is different.

**Syntax**

public String remove(String stringToRemove)

**Example**

String myString1 = 'This Is MyString Example';

String stringToRemove = 'MyString';

String result = myString1.remove(stringToRemove);

System.debug('Value of Result will be 'This Is Example' as we have removed the MyString

and Result is :'+result);

### removeEndIgnoreCase

This method removes the string provided in stringToRemove from the given string but only if it occurs at the end. This method is not case-sensitive.

**Syntax**

public String removeEndIgnoreCase(String stringToRemove)

**Example**

String myString1 = 'This Is MyString EXAMPLE';

String stringToRemove = 'Example';

String result = myString1.removeEndIgnoreCase(stringToRemove);

System.debug('Value of Result will be 'This Is MyString' as we have removed the 'Example'

and Result is :'+result);

### startsWith

This method will return true if the given string starts with the prefix provided in the method.

**Syntax**

public Boolean startsWith(String prefix)

**Example**

String myString1 = 'This Is MyString EXAMPLE';

String prefix = 'This';

Boolean result = myString1.startsWith(prefix);

System.debug(' This will return true as our String starts with string 'This' and the

Result is :'+result);