BEST PRACTICES OF APEX CLASS:-

1.BULKIFY APEX CODE

This principle is to handle multiple records at a time.

2.AVOID SOQL AND DML INSIDE THE FOR LOOP

Do not place (insert/update/delete/undelete) statement inside the loop to avoid iteration again and again .(?)

Do not place (insert/update/delete/undelete) statement inside the loop to avoid governor limit hit.

To avoid follow the below steps:-

- Move SOQL/DML out loops
- Query :-Get all the results using single query and iterate over the result
- DML :- Use List of sObject to perform DML operations.

3.QUERYING LARGE DATA SETS:-

This principle hits the governor limits.

Solution:- Use a SOQL query for Loop

```
for( List<Account> accList: [select id, name from Account where BillingCountry  LIKE '%United%']) {
        // add your logic here.
}
```

4.AVOID TIME OUT ISSUES:-

Do not Query the unnecessary fields due to it will take more time to retrieve the data.

Solution:- Query only the required fields and limited number of records. For that use filter conditions, limit, Order By, offset clauses etc.

5.USE OF MAP OF OBJECTS:-

This principle is used to avoid the SOQL inside the for loops.

This principle helps us to get the values of records from different SObjects Based on Looped Sobjects records.

Solution:-Use Collections.

6. USE OF THE LIMITS APEX METHOD:-

These methods are used to know how many resources like soql, dml used.

```
System.debug('Total Number of SOQL allowed in this Apex code: ' +
Limits.getLimitQueries());

Account acc=[select id, name from Account where BillingCity!=null  order by
createdDate desc limit 1];

System.debug('1. Number of Queries used in this Apex code so far: ' +
Limits.getQueries());
```

Now, using the above Limit methods we can check how many SOQL queries we can issue in the current Apex Context

```
If(Limits.getLimitQueries() - Limits.getQueries()>0) {
  // Execute SOQL Query here.
}
```

7. **AVOID HARD CODING :-**

Solution :- Use Schema Methods , Get the RecordTypeID of the Object.

```
Id accountRTId=
Schema.sobjectType.Account.getRecordTypeInfosByName().get('RTName').getRecordTypeId();
```

8. **USE DATABASE METHODS WHILE DOING DML OPERATIONS:-**

Using Database class Methods to specify whether to allow for partial records processing if errors are encountered.

Solution: -

```
Database.SaveResult[] accountResults=Database.Insert(accountsToBeInsert, false);
// Using above code, we can work on failure records
  For(Database.SaveResult sr:accountResults) {
    If(!sr.isSuccess()) {
    // add your logic here
    for(Database.Error err : sr.getErrors()) {
    }
  }
}
```

9.**EXCEPTION HANDLING IN APEX CODE:-**

Solution: Use try catch block for exception handling

```
try{
    // Apex Code
}Catch(Exception e){
}
```

10. **USE ASYNCHRONOUS APEX:-**

Using asynchronous apex to handle Long-Running Operations/ Bulk Data Processing, Scheduled Operations, to avoid mixed dml exception etc.

Generally asynchronous apex comes with higher governor limits.

11. **REMOVE DEBUG STATEMENTS.**

After use, debug statements should be removed to avoid 'cpu time limit exceeded' exception.

BEST PRACTICES OF APEX TRIGGERS:-

1. One Trigger per Object
    - Easy Maintainability
    - Control the Order of Execution

2.Logic Less Trigger

INSTEAD

```
trigger ContactTrigger on Contact (before insert) {
    for(Contact con: Trigger.New){
        con.Description = 'Login-Less Apex Triggers Rocks!';
    }
}
```

SOLUTION:- USE THIS

- Create an Apex class
- Create a method inside the apex class

- Call the Apex class method from trigger

```apex
public class ContactTriggerHandler {
    public static void handleBeforeInsert(List<Contact> contactList){
        for(Contact con: contactList){
            con.Description = 'Login-Less Apex Triggers Rocks!';
        }
    }
}
```

```apex
trigger ContactTrigger on Contact (before insert) {
    ContactTriggerHandler.handleBeforeInsert(Trigger.New);
}
```

3.**Avoid SOQL Inside the for Loop**

4.**Do Not Recode Standard Functionalities**:- Use FLS Instead

5.**Use The "With Sharing"or "Without Sharing" Keyword**

6.**Avoid HardCoding IDS**

7.**Handle Recursion**:- The Trigger calls itself or directly or indirectly.

Solution:-

Use Static Variables and Check for their value before executing the rest of the trigger **logic.**

8.Use Custom Metadata or Custom Settings for Configurations: - Avoid hard-coding and allow for easy updates to configuration values.

BEST PRACTICES OF APEX TEST CLASS:-

1. **USE TEST DATA FACTORY: -**

   - **Factory Classes:** Create utility classes or methods to generate test data. This helps in maintaining consistency and reduces redundancy.

2. **TEST DIFFERENT SCENARIOS**

   - POSITIVE: Ensure that your code works for expected valid value

- NEGATIVE:- Ensure it handles Exception
- BULK TESTING :-Test with multiple records.

3. **USE OF SYSTEM.RUNAS(System.runAs)**
    - To Test your code for different profiles and roles to verify security checks.

4. **USE ASSERTION:-**
    - Assert Statements Use System.assertEquals and System.assertNotEquals
    - System.assertEquals(expectedValue, actualValue, 'Optional error message');

5. **TEST ISOLATION:-**
    - Create each test data separately for each method

```
@isTest
static void testMethod1() {
   // Test code
}

@isTest
static void testMethod2() {
   // Test code
}
```

6. **USE TEST.STARTTEST AND TEST.STOPTEST(**Test.startTest and Test.stopTest)

```
Test.startTest();
// Code that you want to test
Test.stopTest();
```

7. **TEST COVERAGE:-** Aim for at least 75% code coverage, but strive for 100% to ensure all paths are tested.

8. **USE @TEST SETUP METHODS:**
    - **Reusable Test Data:** Use `@TestSetup` methods to create test data that can be shared across multiple test methods.

```
@TestSetup
static void setup() {
   // Create common test data
}
```

9. **AVOID SeeALLDATA=TRUE:-**

- **Data Isolation:** Avoid using `@isTest(SeeAllData=true)` as it makes your tests dependent on the existing data in the org, which can lead to inconsistent test results.

10. USE THE addError METHOD : - To validate the condition when necessary.