Phase 5: Apex Programming (Developer)

This phase introduces the fundamentals of **Apex programming** in Salesforce. It includes object-oriented programming concepts, triggers, SOQL/SOSL, collections, control statements, and asynchronous processing. Below is documentation with **working code examples** and explanations for key concepts.

1□ Classes & Objects

Purpose: Classes are blueprints for creating objects. They encapsulate variables (fields), methods (functions), and logic. Objects are instances of classes.

Example: Attendee Handler Class

```
public class AttendeeHandler {

    // Method to assign default status to new Attendees
    public static void assignDefaultStatus(List<Attendee__c> attendees) {
        for (Attendee__c att : attendees) {
            if (String.isBlank(att.Status__c)) {
                att.Status__c = 'Registered';
            }
        }
    }

    // Method to link Feedback to Attendee
    public static void linkFeedback(List<Feedback__c> feedbackList) {
        for (Feedback__c fb : feedbackList) {
            if (fb.Attendee__c == null) {
                fb.Attendee__c = [SELECT Id FROM Attendee__c LIMIT 1].Id;
            }
        }
    }
}
```

2□ Apex Triggers (before/after insert/update/delete)

Purpose: Triggers allow developers to execute custom logic **before or after DML operations** (insert, update, delete, undelete) on Salesforce records.

EventFeedbackTrigger

```
trigger EventFeedbackTrigger on Feedback__c (after insert, after update,
after delete, after undelete) {
    Set<Id> eventIds = new Set<Id>();

    if(Trigger.isInsert || Trigger.isUndelete){
        for(Feedback__c f : Trigger.new) if(f.Event__c != null)
eventIds.add(f.Event__c);
    }

    if(Trigger.isDelete){
        for(Feedback__c f : Trigger.old) if(f.Event__c != null)
eventIds.add(f.Event__c);
    }

    if(Trigger.isUpdate){
        for(Feedback__c fNew : Trigger.new){
            Feedback__c fOld = Trigger.oldMap.get(fNew.Id);
            if(fNew.Event__c != null) eventIds.add(fNew.Event__c);
            if(fOld.Event__c != null) eventIds.add(fOld.Event__c);
        }
    }

    if(eventIds.isEmpty()) return;

    // Aggregate query for feedback counts
    Map<Id, Integer> eventToCount = new Map<Id, Integer>();
    for(AggregateResult ar : [
        SELECT Event__c e, COUNT(Id) cnt
        FROM Feedback__c
        WHERE Event__c IN :eventIds
        GROUP BY Event__c
    ]) {
        eventToCount.put((Id) ar.get('e'),
Integer.valueOf(String.valueOf(ar.get('cnt'))));
    }

    // Update Event records with new counts
    List<Event__c> eventsToUpdate = new List<Event__c>();
    for(Event__c ev : [SELECT Id, Feedback_Count__c FROM Event__c WHERE Id IN
:eventIds]) {
        Integer cnt = eventToCount.containsKey(ev.Id) ?
eventToCount.get(ev.Id) : 0;
        ev.Feedback_Count__c = cnt;
        eventsToUpdate.add(ev);
    }

    if(!eventsToUpdate.isEmpty()) update eventsToUpdate;
}
```

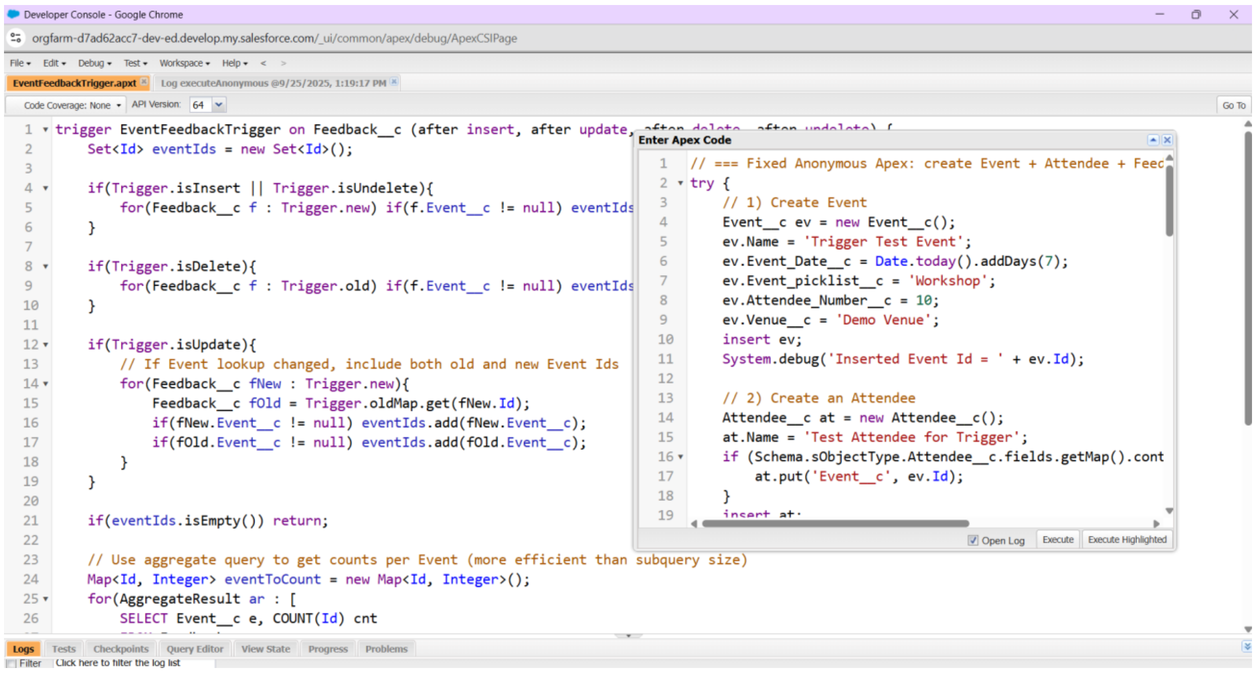
This trigger keeps the **Feedback_Count__c** field on the **Event__c** object up to date whenever feedback records are inserted, updated, deleted, or undeleted.

Anonymous Apex Test Script

```
try {
    // 1) Create Event
    Event__c ev = new Event__c(Name = 'Trigger Test Event', Event_Date__c =
Date.today().addDays(7), Event_picklist__c = 'Workshop', Attendee_Number__c =
10, Venue__c = 'Demo Venue');
    insert ev;

    // 2) Create an Attendee
    Attendee__c at = new Attendee__c(Name = 'Test Attendee for Trigger',
Event__c = ev.Id);
    insert at;

    // 3) Create Feedback records
    List<Feedback__c> fbs = new List<Feedback__c>();
    fbs.add(new Feedback__c(Name = 'Feedback 1', Event__c = ev.Id,
Attendee__c = at.Id, Comments__c = 'Great event'));
    fbs.add(new Feedback__c(Name = 'Feedback 2', Event__c = ev.Id,
Attendee__c = at.Id, Comments__c = 'Loved it'));
    insert fbs;

    // 4) Query Event to check Feedback_Count__c
    ev = [SELECT Id, Feedback_Count__c FROM Event__c WHERE Id = :ev.Id];
    System.debug('Feedback count after insert: ' + ev.Feedback_Count__c);

    // 5) Delete one feedback
    delete fbs[0];
    ev = [SELECT Id, Feedback_Count__c FROM Event__c WHERE Id = :ev.Id];
    System.debug('Feedback count after delete: ' + ev.Feedback_Count__c);

    // 6) Undelete feedback
    undelete fbs[0];
    ev = [SELECT Id, Feedback_Count__c FROM Event__c WHERE Id = :ev.Id];
    System.debug('Feedback count after undelete: ' + ev.Feedback_Count__c);
```

```
} catch (Exception ex) {
    System.debug('Exception: ' + ex.getMessage());
}
```

This is used in **Developer Console → Execute Anonymous** to test the **EventFeedbackTrigger** functionality.
It creates sample **Event, Attendee, Feedback** records and verifies that the Feedback_Count__c field on Event__c updates correctly during **insert, delete, and undelete** operations.



Developer Console - Google Chrome

orgfarm-d7ad62acc7-dev-ed.develop.my.salesforce.com/_ui/common/apex/debug/ApexCSIPage

EventFeedbackTrigger.apxt | Log executeAnonymous @9/25/2025, 1:19:17 PM

Code Coverage: None | API Version: 64

```
trigger EventFeedbackTrigger on Feedback__c (after insert, after update, after delete, after undelete) {
    Set<Id> eventIds = new Set<Id>();

    if(Trigger.isInsert || Trigger.isUndelete){
        for(Feedback__c f : Trigger.new) if(f.Event__c != null) eventIds
    }

    if(Trigger.isDelete){
        for(Feedback__c f : Trigger.old) if(f.Event__c != null) eventIds
    }

    if(Trigger.isUpdate){
        // If Event lookup changed, include both old and new Event Ids
        for(Feedback__c fNew : Trigger.new){
            Feedback__c fOld = Trigger.oldMap.get(fNew.Id);
            if(fNew.Event__c != null) eventIds.add(fNew.Event__c);
            if(fOld.Event__c != null) eventIds.add(fOld.Event__c);
        }
    }

    if(eventIds.isEmpty()) return;

    // Use aggregate query to get counts per Event (more efficient than subquery size)
    Map<Id, Integer> eventToCount = new Map<Id, Integer>();
    for(AggregateResult ar : [
        SELECT Event__c e, COUNT(Id) cnt
```

Enter Apex Code

```
// === Fixed Anonymous Apex: create Event + Attendee + Feec
try {
    // 1) Create Event
    Event__c ev = new Event__c();
    ev.Name = 'Trigger Test Event';
    ev.Event_Date__c = Date.today().addDays(7);
    ev.Event_picklist__c = 'Workshop';
    ev.Attendee_Number__c = 10;
    ev.Venue__c = 'Demo Venue';
    insert ev;
    System.debug('Inserted Event Id = ' + ev.Id);

    // 2) Create an Attendee
    Attendee__c at = new Attendee__c();
    at.Name = 'Test Attendee for Trigger';
    if (Schema.sObjectType.Attendee__c.fields.getMap().cont
        at.put('Event__c', ev.Id);
    }
    insert at;
```

3⃣ Trigger Design Pattern

- Keep triggers lean by delegating logic to **Handler Classes**.
- Ensure **one trigger per object**.
- Support bulk operations.
- Example: `AttendeeTrigger` calls `AttendeeHandler.assignDefaultStatus()`.

4⃣ SOQL & SOSL

- **SOQL**: Query records from a single object or related objects.

  ```
  List<Event__c> events = [SELECT Id, Name FROM Event__c WHERE Venue__c =
  'Demo Venue'];
  ```

- **SOSL**: Search across multiple objects.

  ```
  List<List<SObject>> results = [FIND 'Workshop' IN ALL FIELDS RETURNING
  Event__c(Id, Name), Attendee__c(Id, Name)];
  ```

5□ Collections: List, Set, Map

- **List**: Ordered collection.
- **Set**: Unique values.
- **Map**: Key-value pairs.

6□ Control Statements

- Use `if`, `for`, `while`, `switch` for logic control.
- Bulkify loops and avoid nested SOQL queries.

7□ Batch Apex

- Used for processing large data sets asynchronously in batches.
- Implement `Database.Batchable` interface.

8□ Queueable Apex

- Asynchronous processing with the ability to chain jobs.
- More flexible than future methods.

9□ Scheduled Apex

- Schedule Apex jobs to run at specific times.
- Implement `Schedulable` interface.

□ Future Methods

- Lightweight async execution for simple background tasks.
- Must be static and return void.

1□1□ Exception Handling

- Use `try-catch-finally` blocks.
- Catch `DmlException` and log errors properly.

1□2□ Test Classes

- Ensure at least **75% code coverage** for deployment.
- Test both positive and negative scenarios.
- Use `@isTest` annotation.

1□3□ Asynchronous Processing

- Includes Batch Apex, Queueable Apex, Scheduled Apex, and Future Methods.
- Improves scalability and avoids governor limit issues.

IN MY PROJECT I MAINLY USED APPEX TRIGGER (before/after insert/update/delete) AND OTHER THING AUTOMATICALLY GOT IMPLIMENTED