# 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.

## 1️⃣ 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;  
 }  
 }  
 }  
}

## 

## 2️⃣ 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

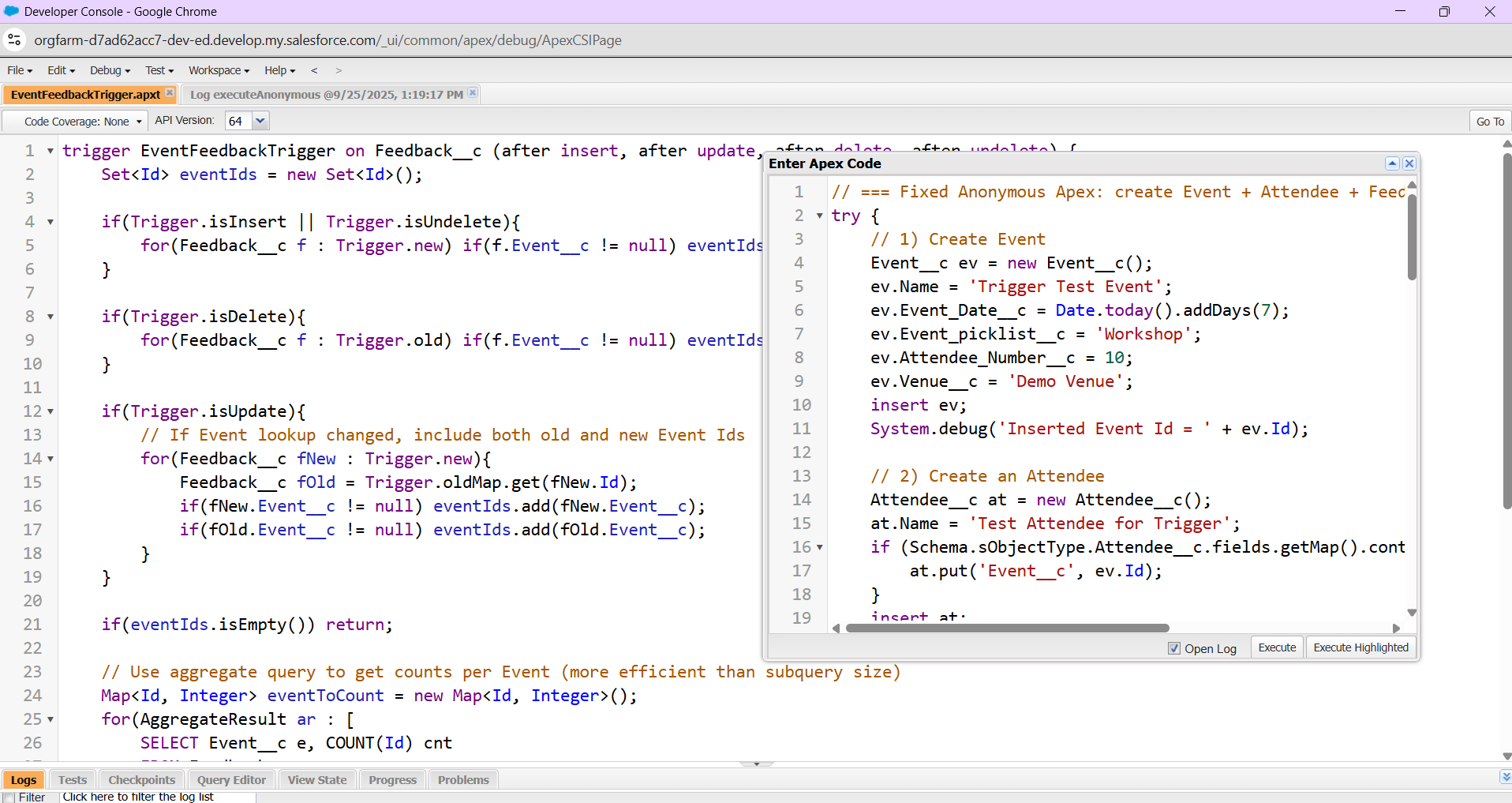
trigger EventFeedbackTrigger on Feedback\_\_c (after insert, after update, after delete, after undelete) {  
 Set<Id> eventIds = new Set<Id>();  
  
 if(Trigger.isInsert || Trigger.isUndelete){  
 for(Feedback\_\_c f : Trigger.new) if(f.Event\_\_c != null) eventIds.add(f.Event\_\_c);  
 }  
  
 if(Trigger.isDelete){  
 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.



## 3️⃣ 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().

## 4️⃣ 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)];

## 5️⃣ Collections: List, Set, Map

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

## 6️⃣ Control Statements

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

## 7️⃣ Batch Apex

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

## 8️⃣ Queueable Apex

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

## 9️⃣ 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.

## 1️⃣1️⃣ Exception Handling

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

## 1️⃣2️⃣ Test Classes

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

## 1️⃣3️⃣ 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**