Phase 5: Apex Programming (Developer)

1. Apex Classes & Objects:

(ResidentConnect): A class to calculate total flats in a society.

```
public class SocietyHelper {
   public static Integer getTotalFlats(Id accountId) {
      List<Flat__c> flats = [SELECT Id FROM Flat__c WHERE
Account__c = :accountId];
      return flats.size();
   }
}
```

steps: Setup → Apex Classes → New → Paste code → Save.

2. Apex Triggers:

```
trigger UpdateResidentCount on Contact (after insert, after delete) {
   Set<Id> accIds = new Set<Id>();
   if(Trigger.isInsert){
      for(Contact c : Trigger.new){
         accIds.add(c.AccountId);
      }
   }
   if(Trigger.isDelete){
      for(Contact c : Trigger.old){
         accIds.add(c.AccountId);
      }
   }
   List<Account> accList = [SELECT Id, Total_Residents__c,
                           (SELECT Id FROM Contacts) FROM Account
                           WHERE Id IN :accIds];
   for(Account a : accList){
      a.Total_Residents__c = a.Contacts.size();
   }
   update accList;
}
```

3. Trigger Design Pattern:

```
trigger ResidentTrigger on Contact (after insert, after delete) {
   ResidentTriggerHandler.updateResidentCount(Trigger.new, Trigger.old,
Trigger.isInsert, Trigger.isDelete);
}

public class ResidentTriggerHandler {
   public static void updateResidentCount(List<Contact> newList,
List<Contact> oldList, Boolean isInsert, Boolean isDelete){
      // logic here
   }
}
```

4. SOQL & SOSL:

```
List<Case> cases = [SELECT Id, Status FROM Case WHERE Status = 'Emergency'];

List<List<SObject>> results = [FIND 'John' IN ALL FIELDS RETURNING Contact(Name,
Email)];
```

5. Collections (List, Set, Map):

List: Ordered collection → `List<String> names = new List<String>();`

Set: Unique values → `Set<String> emails = new Set<String>();`

Map: Key-value pair → `Map<Id, Contact> contactMap = new Map<Id, Contact>([SELECT Id, Name FROM Contact]);`

6. Control Statements:

```
for(Contact c : [SELECT Name, Email FROM Contact]){
   if(c.Email == null){
      System.debug('Resident has no email');
   }
}
```

7.Batch Apex (for large data):

global class RecalculateResidentsBatch implements

```
Database.Batchable<SObject> {
    global Database.QueryLocator start(Database.BatchableContext bc){
        return Database.getQueryLocator('SELECT Id FROM Account');
    }
    global void execute(Database.BatchableContext bc, List<Account>
accList){
        for(Account a : accList){
            a.Total_Residents__c = [SELECT COUNT() FROM Contact
WHERE AccountId = :a.Id];
        }
        update accList;
    }
    global void finish(Database.BatchableContext bc){}
}
```

8. Queueable Apex (asynchronous, lightweight):

```
public class EmergencyNotifier implements Queueable {
    public void execute(QueueableContext context){
        // Send custom notifications
    }
}
```

9. Scheduled Apex (run at specific time):

```
global class EventReminderScheduler implements Schedulable {
    global void execute(SchedulableContext sc){
        // Send event reminders
    }
}
```

$$\begin{tikzcd}
\text{[Resident Inserted]} \arrow[d] \\
\text{Trigger} \rightarrow \text{Handler Class} \arrow[d] \\
\text{Updates Account.ResidentCount} \arrow[d] \\
\text{Async Processing (Future/Queueable/Batch)}
\end{tikzcd}$$

$\downarrow$

Notifications / Reports / Updates