Medicare Cloud — Phase 5: Apex Programming

Classes & Objects

Purpose: Encapsulate behavior and data in reusable, testable units. Apex is an object-oriented language.

Key points:

- Classes contain fields (variables), methods (functions), constructors, and inner classes.
- Access modifiers: public, private, global, protected.
- Use static for class-level helpers or constants.
- Keep logic in service classes (not UI or triggers) for reusability and testability.

Example:

```
public class CarePlanService {
    public static Care_Plan__c createStandardPlan(Id patientId) {
        Care_Plan__c plan = new Care_Plan__c(Patient__c = patientId,
Plan_Type__c = 'Standard');
        insert plan;
        return plan;
    }
}
```

Apex Triggers

Purpose: Execute logic automatically in response to DML on records.

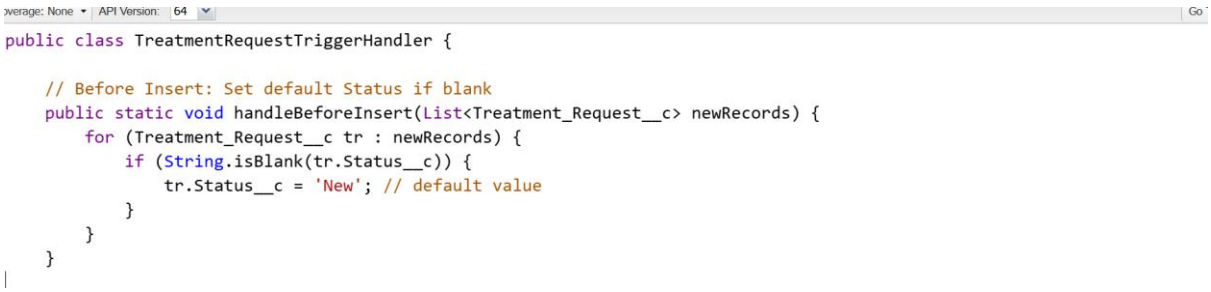
Key points:

- Trigger events: before insert, after insert, before update, after update, before delete, after delete, after undelete.

- Context variables: Trigger.new, Trigger.old, Trigger.isInsert, Trigger.isUpdate, Trigger.isDelete, Trigger.newMap, Trigger.oldMap.
- Use **before** triggers to modify fields before save (no DML). Use **after** triggers for related-record DML or callouts (requires separate async callout).

Example skeleton:

```
trigger TreatmentRequestTrigger on Treatment_Request__c (before insert, after update) {
    if (Trigger.isBefore && Trigger.isInsert) {
        TreatmentRequestTriggerHandler.handleBeforeInsert(Trigger.new);
    }
    if (Trigger.isAfter && Trigger.isUpdate) {
        TreatmentRequestTriggerHandler.handleAfterUpdate(Trigger.new, Trigger.oldMap);
    }
}
```



Trigger Design Pattern

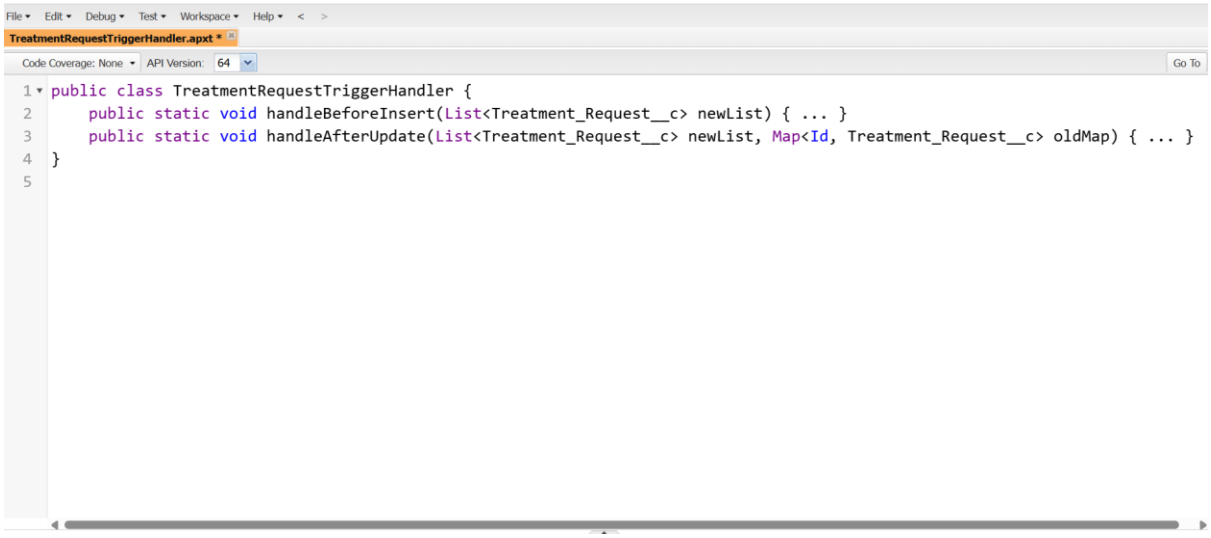
Purpose: Structure triggers to be maintainable, testable and to prevent recursion and governor-limit issues.

Key points / pattern:

- Single trigger per SObject that calls a handler class.
- Handler class has methods by context (beforeInsert, afterUpdate, etc.).
- Use a static Boolean or a `Set<Id>` in a utility class to prevent recursion.
- Keep logic small in triggers; complex logic belongs in service classes.

Example handler outline:

```
public class TreatmentRequestTriggerHandler {
   public static void handleBeforeInsert(List<Treatment_Request__c> newList)
{ ... }
   public static void handleAfterUpdate(List<Treatment_Request__c> newList,
Map<Id, Treatment_Request__c> oldMap) { ... }
}
```



SOQL & SOSL

Purpose: Query records. SOQL (Salesforce Object Query Language) retrieves rows from a single object (with joins). SOSL (Salesforce Object Search Language) performs text searches across multiple objects.

Key points:

- Use selective SOQL (filter on indexed fields, avoid LIKE '%...%' when possible).
- Avoid queries inside loops; bulkify queries.
- Use relationship queries (SELECT Id, Account.Name FROM Contact) and subqueries.
- SOSL is good for global text searches across objects and fields.

Example SOQL:

```
List<Patient__c> pts = [SELECT Id, Name, Diagnosis__c FROM Patient__c
WHERE Diagnosis__c = :diagnosisLimit];
```

Example SOSL:

List<List<sObject>> results = [FIND :searchText IN ALL FIELDS RETURNING Patient__c(Id, Name), Treatment_Request__c(Id, Name)];

Collections: List, Set, Map

Purpose: Efficiently manage groups of elements and perform bulk operations.

Key points:

- List<T> preserves order and allows duplicates.
- Set<T> stores unique values (fast membership checks).
- Map<Key, Value> stores key-value pairs (fast lookups).
- Common pattern: query into a Map<Id, SObject> for fast reference during processing.

Example:

```
Map<Id, Patient__c> patientMap = new Map<Id, Patient__c>([SELECT Id, Name FROM Patient__c WHERE Id IN :patientIds]);
Set<String> policySet = new Set<String>{'P1','P2'};
List<Care_Plan__c> plansToInsert = new List<Care_Plan__c>();
```

Control Statements

Purpose: Standard flow control constructs to implement conditional logic and loops.

Key points:

- if, else if, else, switch (Apex switch supports types), loops (for, for-each, while).
- Use for (SObject s : records) for iteration over collections.
- Use continue / break where appropriate.
- Use try/catch/finally for exception handling.

Example:

```
for (Treatment_Request__c tr : Trigger.new) {
   if (tr.Amount__c > 200000) {
      // tier-2 approval
```

```
    } else {
        // normal path
    }
}
```

Batch Apex

Purpose: Process large volumes of data asynchronously in manageable chunks.

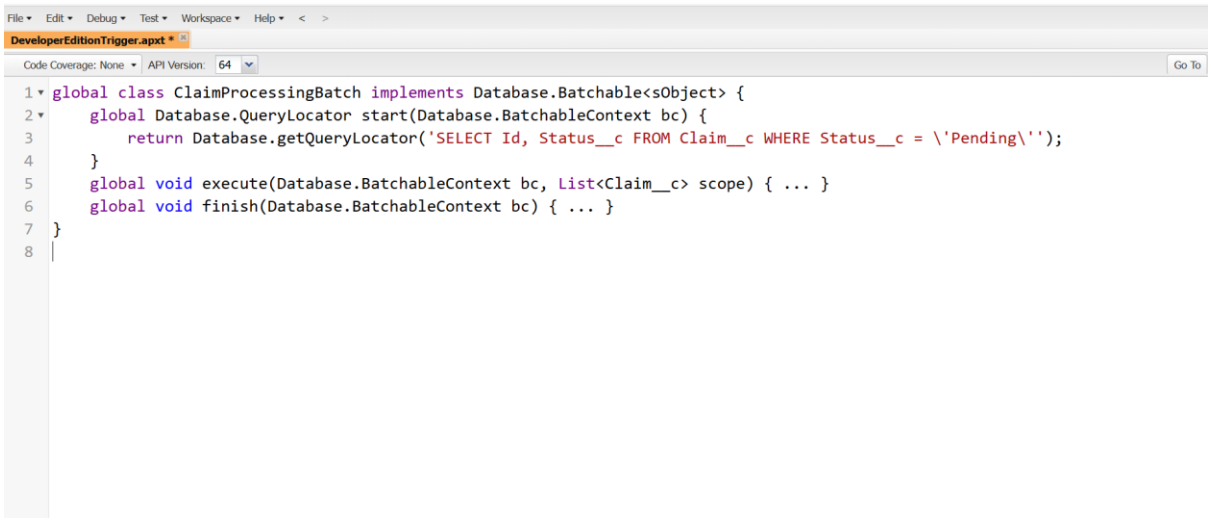
Key points:

- Implement Database.Batchable<SObject> with start, execute, and finish methods.
- start returns a QueryLocator or Iterable.
- execute runs in batches (default batch size configurable).
- Use Database.Stateful if you need to maintain state across batches.
- Ideal for long-running data processing (millions of rows), mass updates, or reprocessing.

Skeleton:

```
global class ClaimProcessingBatch implements Database.Batchable<sObject> {
    global Database.QueryLocator start(Database.BatchableContext bc) {
        return Database.getQueryLocator('SELECT Id, Status__c FROM
Claim__c WHERE Status__c = \'Pending\');
    }
```

```
    global void execute(Database.BatchableContext bc, List<Claim__c> scope) {
... }
    global void finish(Database.BatchableContext bc) { ... }
}
```



Queueable Apex

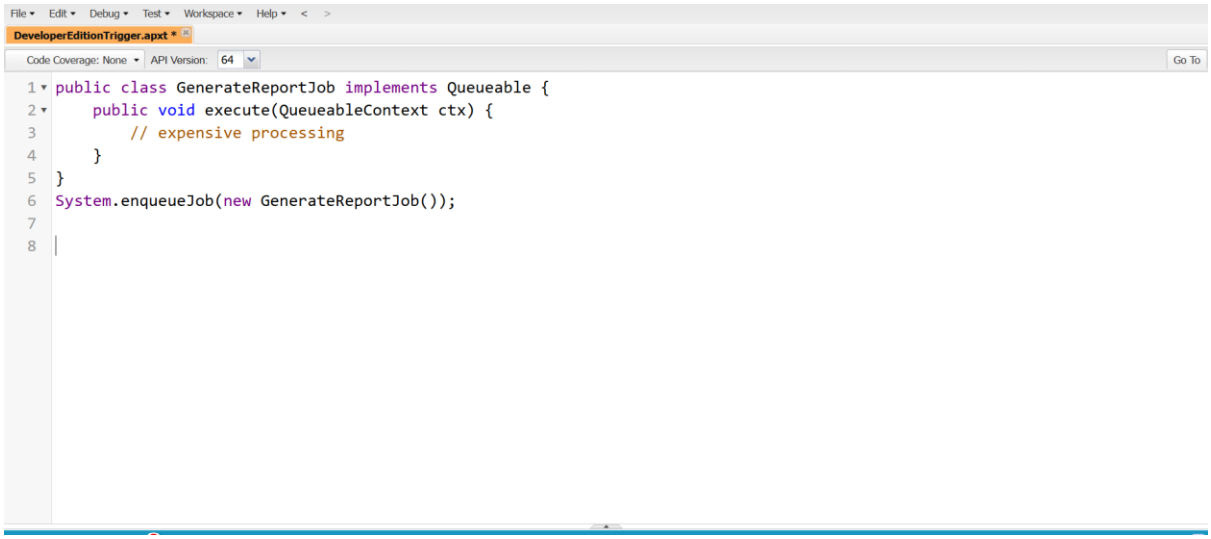
Purpose: Flexible async processing with richer types and chaining capability (preferred over @future for complex tasks).

Key points:

- Implement Queueable and call with System.enqueueJob(new MyJob(args)).
- Supports non-primitive arguments (sObjects, collections) up to limits.
- Jobs can be chained (enqueue another job from execute).
- Traceable via AsyncApexJob.

Example:

```
public class GenerateReportJob implements Queueable {
    public void execute(QueueableContext ctx) {
        // expensive processing
    }
}
System.enqueueJob(new GenerateReportJob());
```



Scheduled Apex

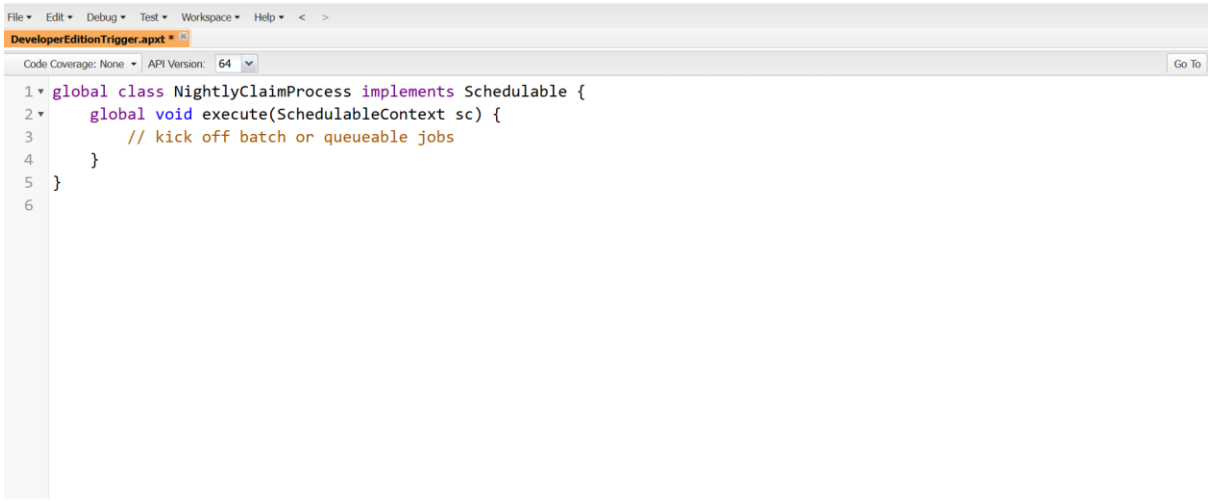
Purpose: Run Apex classes at scheduled intervals or cron expressions.

Key points:

- Implement Schedulable with execute(SchedulableContext sc).
- Schedule via System.schedule('Job Label', cronExpression, new MySchedulable()) or via UI.
- Good for periodic maintenance, nightly jobs, or recurring reports.

Example:

```
global class NightlyClaimProcess implements Schedulable {
   global void execute(SchedulableContext sc) {
      // kick off batch or queueable jobs
   }
}
System.schedule('Nightly Claim', '0 0 2 * * ?', new NightlyClaimProcess()); // runs 2:00 AM daily
```



Future Methods

Purpose: Fire-and-forget async methods for simple asynchronous tasks and callouts.

Key points:

- Annotate with @future (optionally @future(callout=true)).
- Methods must be static, return void, and accept only primitive or collections of primitive types.
- Limited concurrency and invocation counts per transaction.

Example:

```
@future(callout=true)
public static void callExternalService(String endpoint) {
   // perform HTTP callout
}
```

Exception Handling

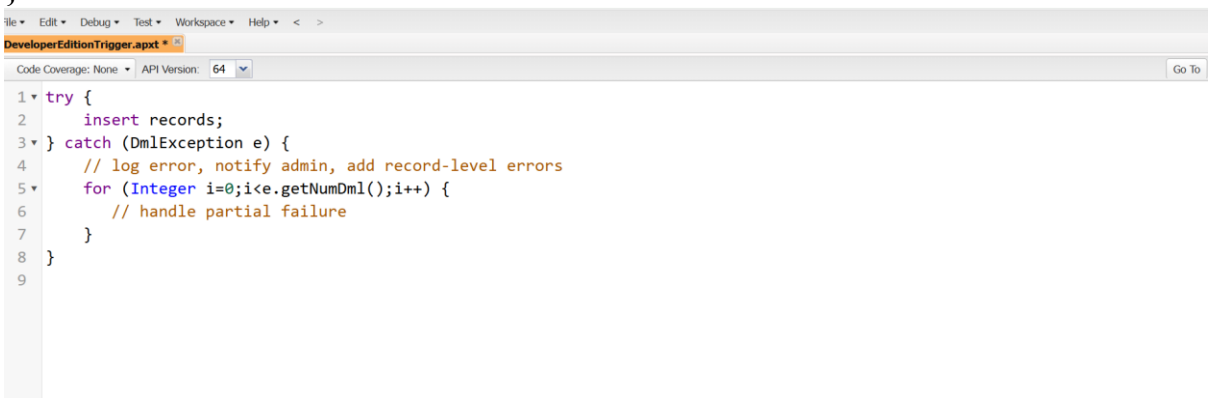
Purpose: Handle runtime errors gracefully and surface meaningful messages.

Key points:

- Use try { ... } catch (Exception e) { ... } finally { ... }.
- Create custom exceptions via public class MyException extends Exception {}.
- Use sObject.addError('msg') to report validation errors on records in triggers.
- Avoid swallowing exceptions silently; log or rethrow as appropriate.

Example:

```
try {
   insert records;
} catch (DmlException e) {
   // log error, notify admin, add record-level errors
   for (Integer i=0;i<e.getNumDml();i++) {
     // handle partial failure
   }
}
```



Test Classes

Purpose: Validate code correctness and meet Salesforce deployment coverage requirements.

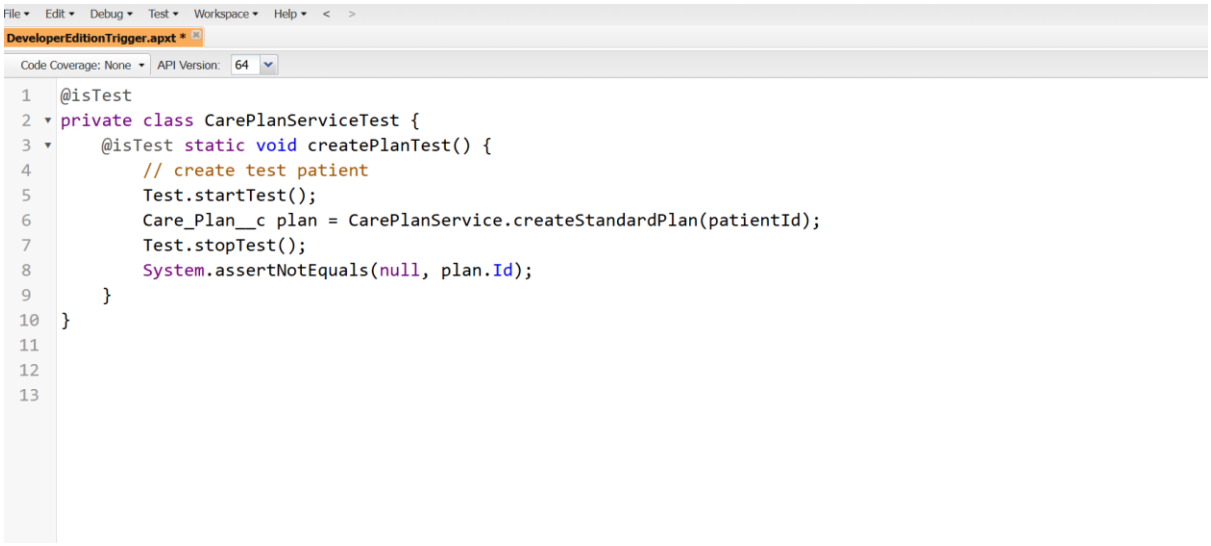
Key points:

- Annotate tests with @isTest (or use testMethod older style).
- Use SeeAllData=false (create test data inside tests) to ensure isolation.
- Use Test.startTest() / Test.stopTest() to test async behavior.
- Assert expected outcomes using System.assertEquals / System.assert.

Example structure:

```
@isTest
```

```
private class CarePlanServiceTest {
    @isTest static void createPlanTest() {
        // create test patient
        Test.startTest();
        Care_Plan__c plan = CarePlanService.createStandardPlan(patientId);
        Test.stopTest();
        System.assertNotEquals(null, plan.Id);
    }
}
```



Asynchronous Processing (Summary)

Purpose: Offload long-running or large-volume work from synchronous transactions.

Key mechanisms:

- **Future methods:** simple async, limited arguments.
- **Queueable Apex:** richer types, chaining, monitoring.
- **Batch Apex:** large-scale processing in chunks.
- **Scheduled Apex:** periodic execution of logic.

When to use which:

- Use **Batch** for millions of rows or large reprocessing jobs.
- Use **Queueable** for complex single-run async jobs and when you need to pass sObjects/collections.
- Use **Future** only for simple fire-and-forget or legacy needs.

- Use **Scheduled** to regularly trigger batch/queueable jobs.