

Phase 5: Apex Programming (Developer)

1.Classes & Objects

Classes in Apex

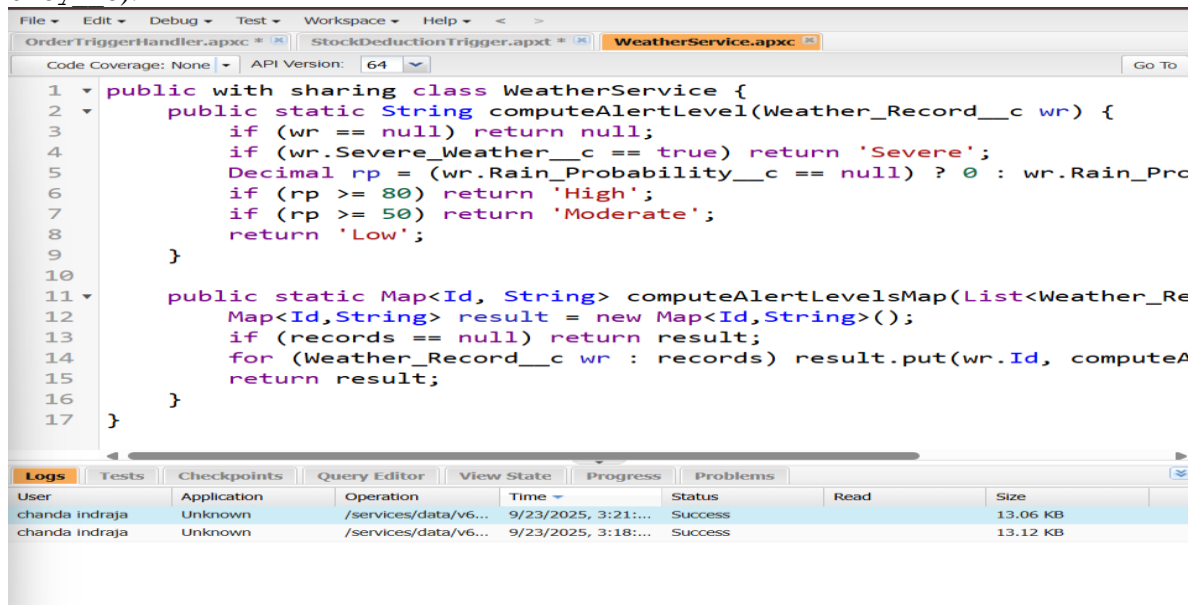
- **Definition:**
Classes are blueprints or templates used to create objects in Salesforce. They encapsulate **business logic**, **data structures**, and **methods** that define the behavior of an object.
- **Purpose:**
 - Centralize code for easier maintenance and reuse.
 - Separate complex logic from triggers to follow **best practices**.
 - Facilitate **unit testing** of business logic independently of UI or triggers.

Key Components of a Class:

- **Properties (Variables):** Hold data (e.g., temperature, humidity).
- **Methods (Functions):** Contain logic to manipulate or fetch data.
- **Constructors:** Special methods used to initialize objects.
- **Access Modifiers:** public, private, global – control visibility.
- **Example – Weather Project Objects:**

Types of Objects:

- **Standard Objects:** Predefined by Salesforce (e.g., Account, Contact).
- **Custom Objects:** Created specifically for your project (e.g., Weather_Record__c, City__c).



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

1. What is an Apex Trigger

- **Definition:**

An Apex Trigger is a piece of Apex code that executes **before or after a record is inserted, updated, deleted, or undeleted** in Salesforce.

- **Purpose:**

- Automate business processes.
- Perform operations that cannot be done with declarative tools like Workflow or Process Builder.
- Ensure data integrity and enforce complex logic.

2. Trigger Events

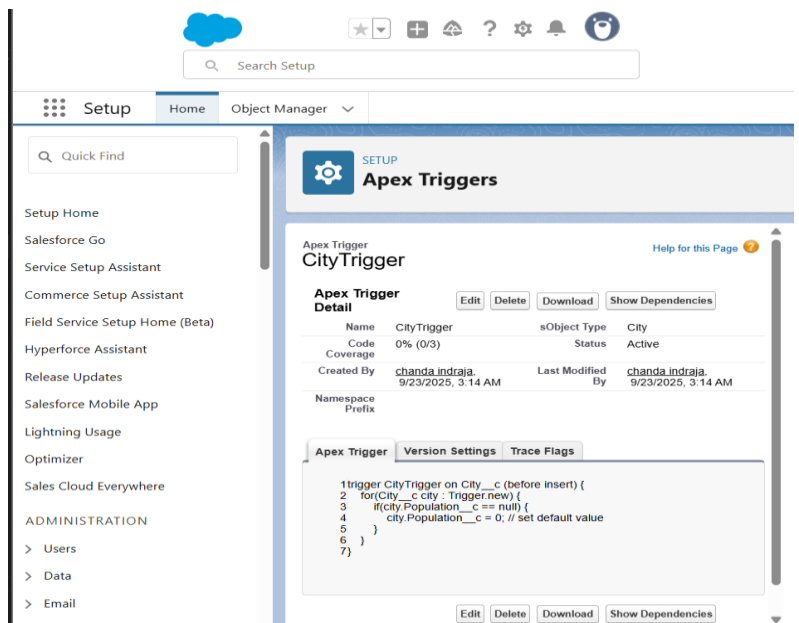
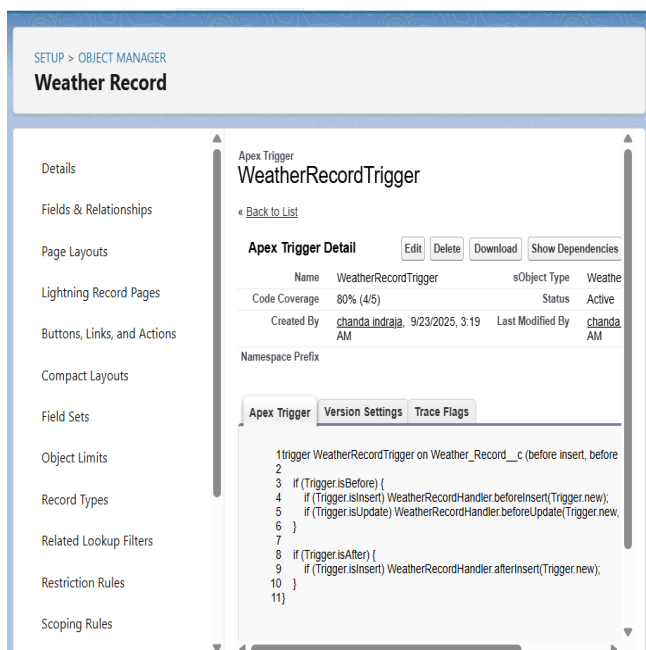
Triggers can be executed **before** or **after** DML (Data Manipulation Language) operations:

1. **Before Triggers:**

- Executed **before** a record is saved to the database.
- Used to **validate or modify field values** before insertion or update.
- Example: Automatically setting default temperature for new weather records.

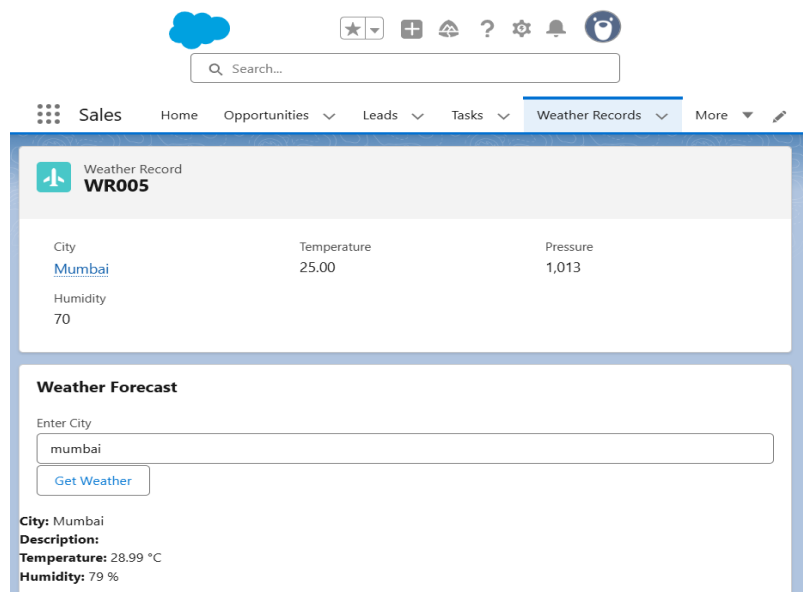
2. **After Triggers:**

- Executed **after** a record is saved to the database.
- Used for actions that require **record IDs** or interactions with other objects.
- Example: Creating related Weather_Record__c after a City__c is added.



```
File Edit Debug Test Workspace Help < >
WeatherRecord.apxt * WeatherRecordHandler.apxc * Log executeAnonymous @9/23/2025, 3:55:01 PM Log ex
Code Coverage: None API Version: 64 Go To

1 public with sharing class WeatherRecordHandler {
2
3 // BEFORE INSERT
4 public static void beforeInsert(List<Weather_Record__c> newList) {
5     for (Weather_Record__c wr : newList) {
6         if (wr.Temperature__c < -50 || wr.Temperature__c > 70) {
7             wr.addError('Temperature seems invalid. Please check');
8         }
9     }
10 }
11
12 // BEFORE UPDATE
13 public static void beforeUpdate(List<Weather_Record__c> newList,
14                                 List<Weather_Record__c> oldList) {
15     for (Weather_Record__c wr : newList) {
16         Weather_Record__c oldRec = oldList.get(wr.Id);
17
18         if (wr.Temperature__c != oldRec.Temperature__c &&
19             (wr.Temperature__c < -50 || wr.Temperature__c > 70)) {
20             wr.addError('Temperature seems invalid. Please check');
21         }
22     }
23 }
```



3.Trigger Design Pattern

1. What is a Trigger Design Pattern

- **Definition:**
A Trigger Design Pattern is a **structured way to write Apex triggers** that ensures clean, maintainable, and scalable code.
- **Purpose:**
 - Avoid messy triggers with duplicated logic.
 - Handle **bulk operations** efficiently.
 - Separate trigger logic from business logic (delegated to handler classes).

2. Key Components

1. **Trigger:**
 - The entry point that listens to events (insert, update, delete).
 - Should be **slim**; only calls the handler class.
2. **Handler Class:**
 - Contains the main business logic (create, update, delete related records).
 - Bulkified to handle multiple records at once.
3. **Utility or Service Classes (optional):**
 - For API calls, complex calculations, or reusable methods.

3. Benefits

- Easy to maintain and update.

- Single trigger per object.
- Supports bulk operations for multiple records.
- Reduces errors and improves readability.

The screenshot shows an IDE with two tabs: 'WeatherRecord.apxt' and 'WeatherRecordHandler.apxc'. The main editor displays the following Apex code:

```

1 trigger WeatherRecordTrigger on Weather_Record__c (before insert, be
2     if(Trigger.isBefore){
3         if(Trigger.isInsert) WeatherRecordHandler.beforeInsert(Trigg
4         if(Trigger.isUpdate) WeatherRecordHandler.beforeUpdate(Trigg
5
6
7
8     };
9     insert city;
10    System.debug('City created: ' + city.Id);
11
12    // Step 2: Insert a valid Weather_Record
13    Weather_Record__c wrValid = new Weather_Record__c(
14        Temperature__c = 25,
15        Pressure__c = 1013,
16        Humidity__c = 70,
17        City_Master_Detail__c = city.Id // | Corrected Master-Deta
18    );
19    insert wrValid;
20

```

An 'Enter Apex Code' modal window is open, showing the same code as the main editor. The bottom of the IDE shows a 'Logs' tab and a table with columns 'Name', 'Line', and 'Problem'.

4.SOQL & SOSL

1. SOQL (Salesforce Object Query Language)

- **Definition:**
SOQL is used to **query records from Salesforce objects**.
- **Purpose:**
 - Retrieve specific fields and records.
 - Filter and order data using WHERE, ORDER BY, LIMIT.
- **Example – Fetch Weather Records by City:**

```
List<Weather_Record__c> records = [
```

```
    SELECT Id, Temperature__c, Humidity__c
```

```
    FROM Weather_Record__c
```

```
    WHERE City__c = :cityId
```

```
];
```

2. SOSL (Salesforce Object Search Language)

- **Definition:**
SOSL is used to **search text, email, and phone fields** across multiple objects at once.
- **Purpose:**
 - Find records when you **don't know the exact field** where the data exists.
 - Useful for **search functionality** in Lightning Web Components (LWC) or Apex.
- **Example – Search Weather Records by Keyword:**

List<List<SObject>> searchResults = [FIND 'rain*' IN ALL FIELDS RETURNING Weather_Record__c(Name, Description__c)];

The screenshot shows the Salesforce Execution Log interface. The log table has columns for Timestamp, Event, and Details. A modal window titled "Log line #542 details" is open, displaying the following information:

```
16:43:39:024 USER_DEBUG [65]||DEBUG|SOSL Result: ((City__c:{Id=a08gL00000Ab2mgQAB, Name=Mumbai}, City__c:{Id=a08gL00000Ab1tpQAB, Name=Mumbai}, City__c:{Id=a08gL00000Aav24QAB, Name=Mumbai}), ())
```

The modal window has an "OK" button at the bottom.

The screenshot shows the Salesforce Execution Log interface. The log table has columns for Timestamp, Event, and Details. A modal window titled "Log line #499 details" is open, displaying the following information:

```
16:43:38:654 SOQL_EXECUTE_BEGIN [57]||Aggregations:0|SELECT Id, Last_Updated_Weather__c FROM City__c WHERE Id = :tmpVar1
```

The modal window has an "OK" button at the bottom.

5.Collections: List, Set, Map

Overview

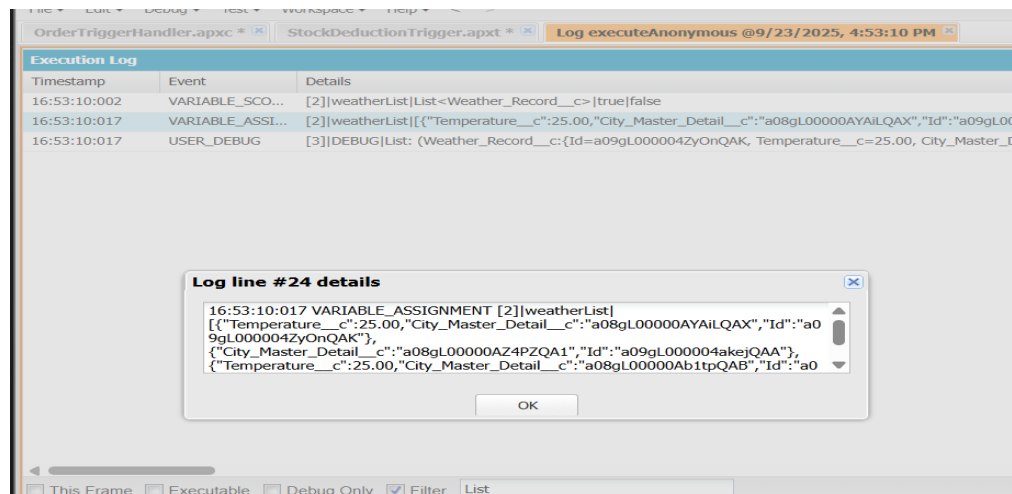
Collections in Apex are variables that can store multiple records or values together. They are widely used for **bulk data handling** in Salesforce.

Types of Collections

- **List**
 - Ordered collection.
 - Can contain duplicate values.
 - Index starts from 0.
 - Example use: Store multiple city names.

```
List<String> cities = new List<String>{'Delhi', 'Mumbai', 'Chennai'};
```

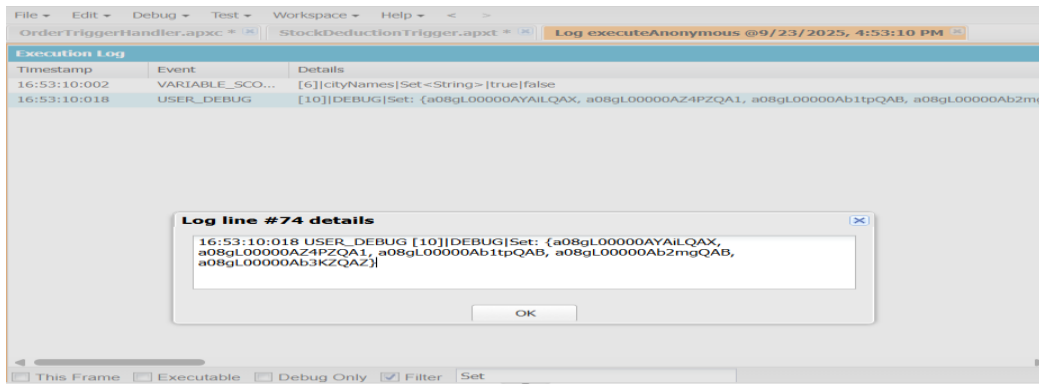
```
System.debug(cities[0]); // Delhi
```



- **Set**
 - Unordered collection.
 - Does not allow duplicate values.
 - Example use: Store unique city names to avoid duplicates.

```
Set<String> cities = new Set<String>{'Delhi', 'Mumbai', 'Delhi'};
```

```
System.debug(cities.size());
```



- **Map**

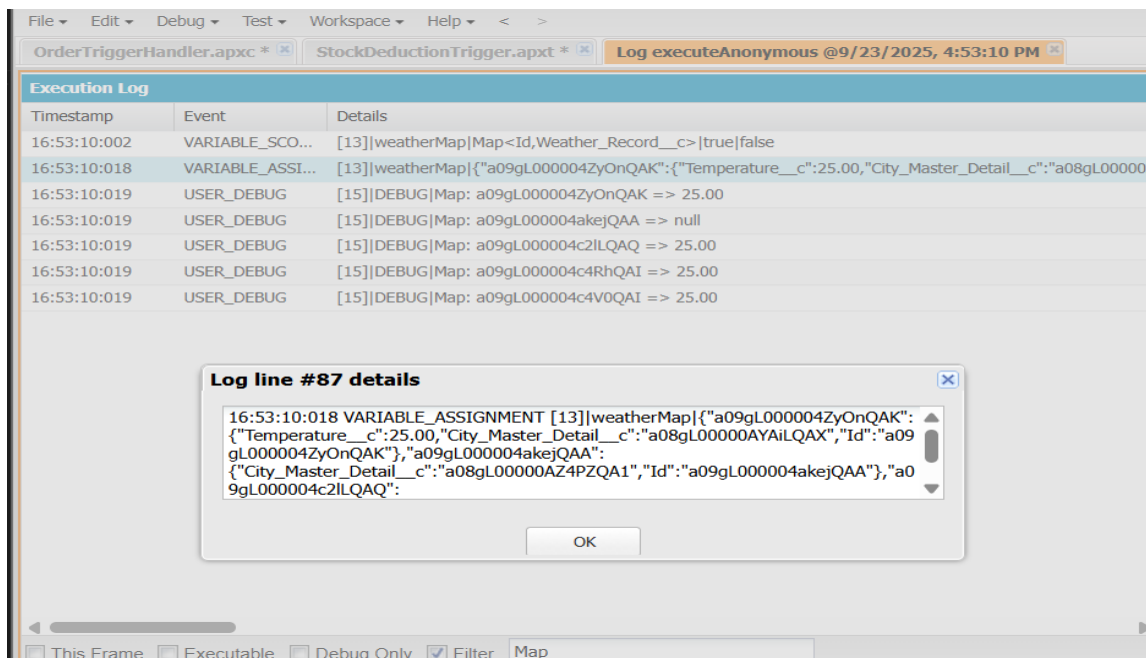
- Collection of key–value pairs.
- Each key is unique, but values can repeat.
- Example use: Map city names to their temperature values.

```
Map<String, Integer> cityTemp = new Map<String, Integer>();
```

```
cityTemp.put('Delhi', 32);
```

```
cityTemp.put('Mumbai', 29);
```

```
System.debug(cityTemp.get('Delhi')); // 32
```



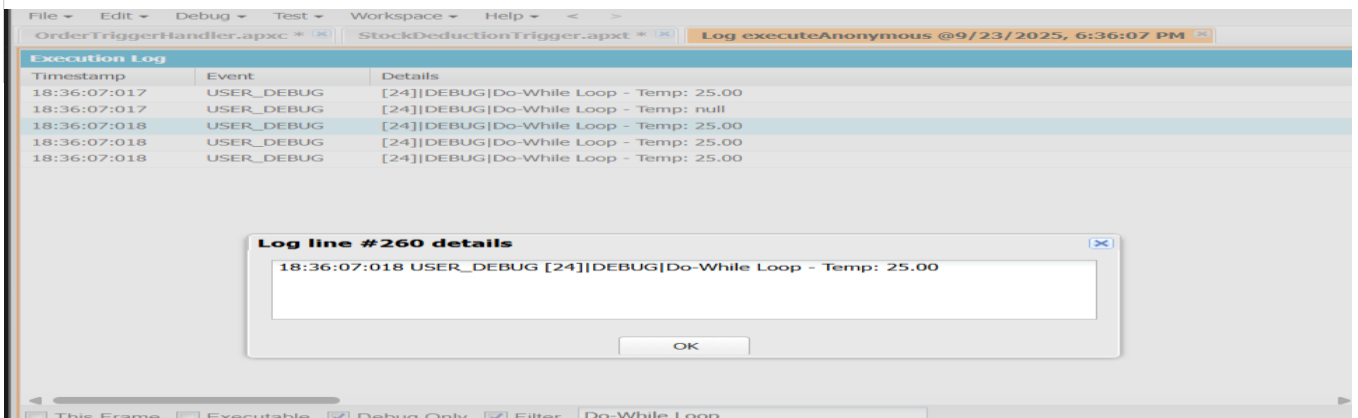
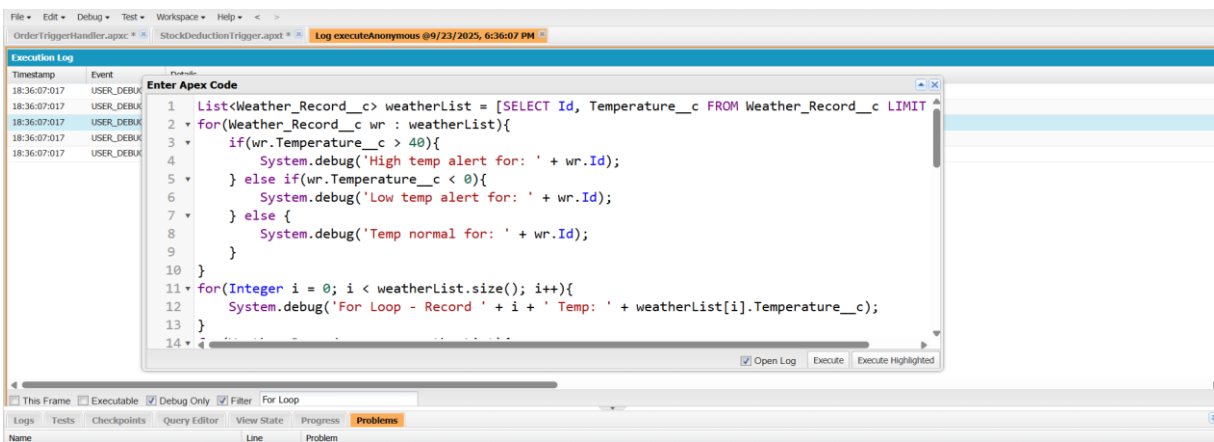
6. Control Statements

Overview

Control statements define the **flow of logic** in Apex programs.

Types

- **Conditional Statements**
 - if-else → Executes code based on conditions.
 - switch → Cleaner alternative to multiple if-else statements.
- **Looping Statements**
 - for → Iterates through records for a fixed number of times.
 - for-each → Simplified loop for collections.
 - while → Repeats as long as condition is true.
- **Break & Continue**
 - break → Immediately exits a loop.
 - continue → Skips current iteration and moves to next



7. Batch Apex

Overview

Batch Apex is used to process **large volumes of data** in manageable chunks, without hitting governor limits.

Why It's Needed

- Salesforce has strict **limits** on processing records.
- Batch Apex splits large jobs into smaller sets and processes them asynchronously.

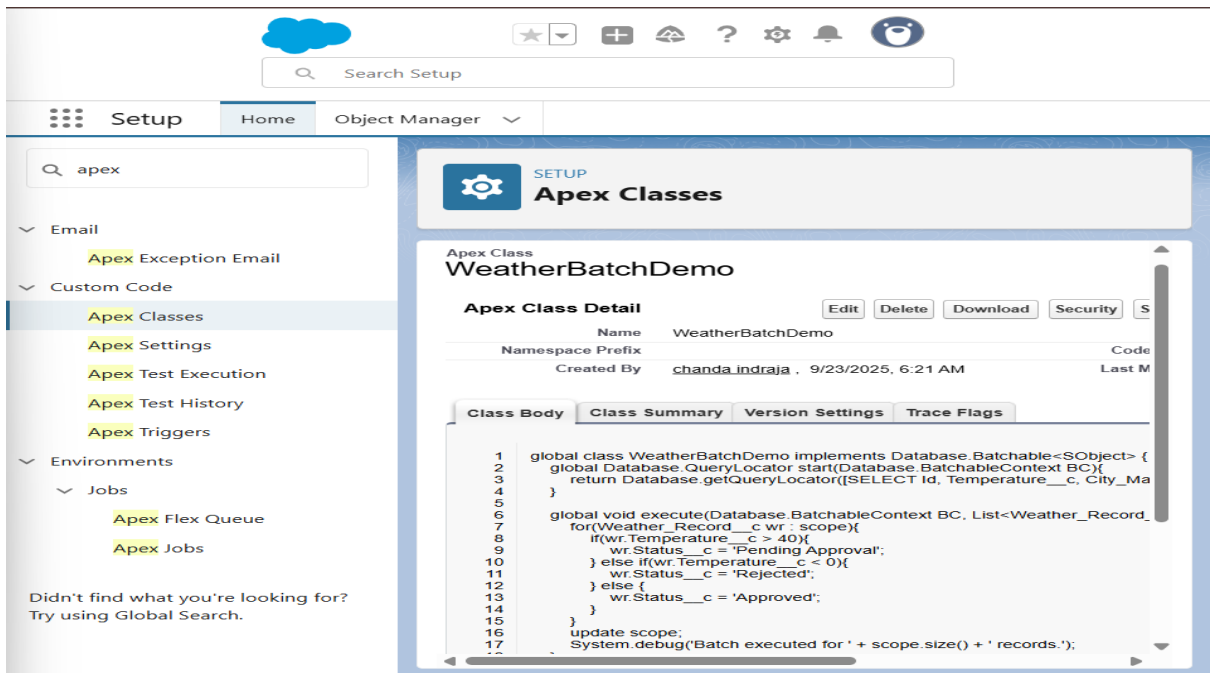
Structure

A Batch Apex class has three main methods:

1. **Start** → Collects the records to be processed.
2. **Execute** → Runs on each batch of records.
3. **Finish** → Performs final tasks once all batches are complete.

Benefits

- Handles millions of records safely.
- Runs in the background (asynchronous).
- Useful for scheduled jobs, data cleanup, and API-based data processing.



The screenshot shows the Salesforce Setup interface. The left sidebar contains a search bar with 'apex' and a list of categories: Email, Custom Code, Environments, and Jobs. Under Custom Code, 'Apex Classes' is selected. The main content area is titled 'Apex Classes' and shows the details for the 'WeatherBatchDemo' class. The class details include the name, namespace prefix, and the class body code. The class body code is as follows:

```
1 global class WeatherBatchDemo implements Database.Batchable<SObject> {
2     global Database.QueryLocator start(Database.BatchableContext BC){
3         return Database.getQueryLocator([SELECT Id, Temperature__c, City_Ma
4     })
5     }
6     global void execute(Database.BatchableContext BC, List<Weather_Record_
7         for(Weather_Record__c wr : scope){
8             if(wr.Temperature__c > 40){
9                 wr.Status__c = 'Pending Approval';
10            } else if(wr.Temperature__c < 0){
11                wr.Status__c = 'Rejected';
12            } else {
13                wr.Status__c = 'Approved';
14            }
15        }
16        update scope;
17        System.debug('Batch executed for ' + scope.size() + ' records.');
```

8.Queueable Apex

Overview

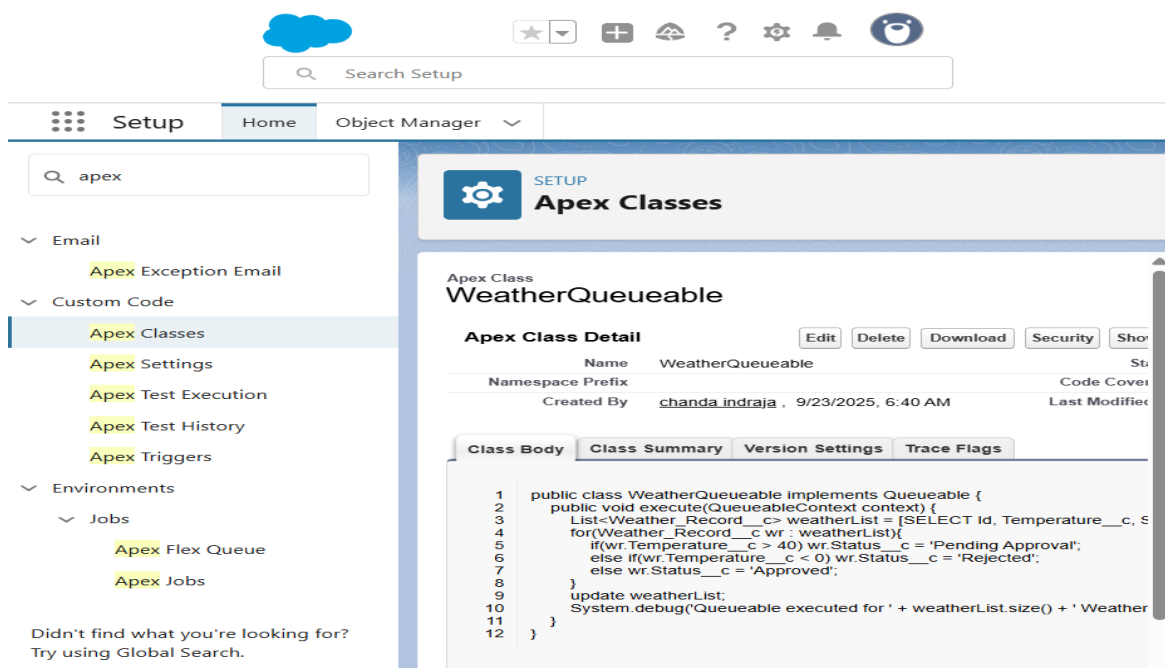
Queueable Apex is used to run **asynchronous jobs** in Salesforce. It is similar to `Future` methods but more powerful.

Key Features

- Allows **chaining** of jobs (one queueable can enqueue another).
- Supports **complex data types** like `sObjects` and custom classes as parameters.
- More flexible than `Future` methods.

Benefits

- Handles long-running processes without blocking users.
- Can process large datasets in the background.
- Easy to monitor in the **Apex Jobs** page.
- **Interface Used:** `System.Queueable`.
- Can process **more records than Future methods**, but still within governor limits.
- Can be **monitored, paused, and retried** from the **Apex Jobs** page.
- Supports **complex data types** as parameters (Lists, `sObjects`, Maps).
- Jobs can be **chained** → one job automatically starts another.
- Easier to test compared to `Future` methods.



The screenshot displays the Salesforce Setup interface. On the left, the 'Setup' menu is visible with a search bar containing 'apex'. The 'Custom Code' section is expanded, showing 'Apex Classes' as the selected option. The main content area is titled 'Apex Classes' and shows details for the 'WeatherQueueable' class. The 'Apex Class Detail' section includes buttons for 'Edit', 'Delete', 'Download', 'Security', and 'Show'. Below this, a table lists the class's metadata: Name (WeatherQueueable), Namespace Prefix, Created By (chanda.indraja), and Created Date (9/23/2025, 6:40 AM). The 'Class Body' tab is active, displaying the following Apex code:

```
1 public class WeatherQueueable implements Queueable {
2     public void execute(QueueableContext context) {
3         List<Weather_Record__c> weatherList = [SELECT Id, Temperature__c, S
4         for (Weather_Record__c wr : weatherList){
5             if (wr.Temperature__c > 40) wr.Status__c = 'Pending Approval';
6             else if (wr.Temperature__c < 0) wr.Status__c = 'Rejected';
7             else wr.Status__c = 'Approved';
8         }
9         update weatherList;
10        System.debug('Queueable executed for ' + weatherList.size() + ' Weather
11    }
12 }
```

9.Scheduled Apex

1. Purpose

- Automates tasks at **specific times or intervals**.
- Removes the need for manual execution of repetitive jobs.
- Useful for **maintenance, integration, and data refreshes**.

2. Features

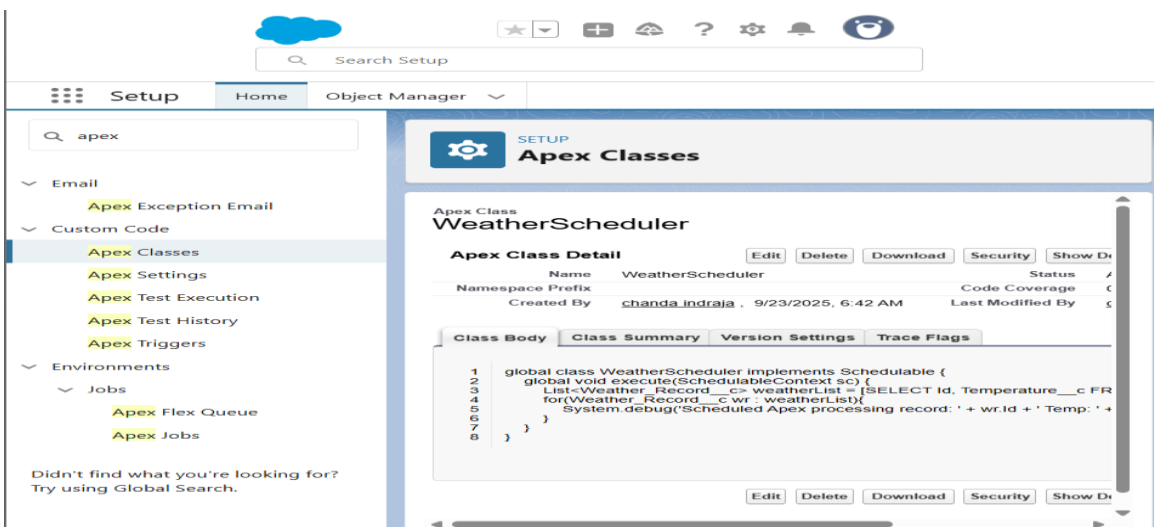
- Runs in the background (asynchronous execution).
- Can be scheduled via **Setup UI** or **System.schedule() method**.
- Uses **cron expressions** to define schedule (similar to UNIX cron jobs).
- Can call **Batch Apex** or **Queueable Apex** inside scheduled jobs.

3. Use Cases

- Daily or nightly updates (e.g., refresh weather data for all cities).
- Weekly or monthly reports (e.g., send summary of weather trends).
- Data cleanup jobs (e.g., remove old or duplicate weather records).
- Scheduled API callouts (e.g., fetch data from external weather API).

4. Cron Expression Format

- Structure: **Seconds Minutes Hours Day_of_Month Month Day_of_Week Year (optional)**
- Example:
 - "0 0 0 * * ?" → Runs daily at midnight.
 - "0 0 12 ? * MON" → Runs every Monday at 12 PM.



The screenshot displays the Salesforce Setup interface. On the left, the 'Setup' menu is visible with 'Apex Classes' selected under 'Custom Code'. The main content area shows the 'Apex Classes' page for a class named 'WeatherScheduler'. The page includes a search bar at the top, a 'Search Setup' field, and a 'Setup' tab. The 'Apex Class Detail' section shows the class name 'WeatherScheduler', its namespace prefix, and creation/modification details. Below this, the 'Class Body' tab is active, displaying the following Apex code:

```
1 global class WeatherScheduler implements Schedulable {
2     global void execute(SchedulableContext sc) {
3         List<Weather_Record__c> weatherList = [SELECT Id, Temperature__c FR
4         for(Weather_Record__c wr : weatherList){
5             System.debug("Scheduled Apex processing record: " + wr.Id + " Temp: " +
6         }
7     }
8 }
```

The code is displayed in a text editor with line numbers. At the bottom of the page, there are buttons for 'Edit', 'Delete', 'Download', 'Security', and 'Show D'.

10.Future Methods

Overview

Future methods are used to run processes **asynchronously** in the background, especially for **callouts to external systems**.

Purpose

- Run operations **asynchronously** (in the background).
- Ideal for long-running tasks or external callouts.

Features

- Annotated with `@future`.
- Executes in a **separate thread** from the main transaction.
- Can be used for **callouts, email sending, and data updates**.

Use Cases

- Making **HTTP callouts** to external systems.
- Performing **resource-intensive calculations**.
- Allows integration with external APIs without blocking user actions.
- Helps avoid governor limits during synchronous execution.

The screenshot displays the Salesforce Setup interface. On the left, a navigation menu shows 'Setup' with a search bar containing 'apex'. Under 'Custom Code', 'Apex Classes' is selected. The main content area is titled 'Apex Classes' and shows details for the 'WeatherFuture' class. The 'Apex Class Detail' section includes fields for Name (WeatherFuture), Namespace Prefix, Created By (chanda.indraja), and Created Date (9/23/2025, 6:43 AM). Below this, the 'Class Body' tab is active, displaying the following code:

```
1 public class WeatherFuture {
2     @future
3     public static void updateWeatherStatus() {
4         List<Weather_Record__c> weatherList = [SELECT Id, Temperature__c, S
5         for(Weather_Record__c wr : weatherList){
6             if(wr.Temperature__c > 40) wr.Status__c = 'Pending Approval';
7             else if(wr.Temperature__c < 0) wr.Status__c = 'Rejected';
8             else wr.Status__c = 'Approved';
9         }
10        update weatherList;
11        System.debug('Future method executed for ' + weatherList.size() + ' Wea
12    }
13 }
```

11. Exception Handling

Overview

Exception handling in Apex is used to **catch errors gracefully** and prevent program crashes.

Types of Exceptions

- **System exceptions:** Thrown by Salesforce platform (e.g., `NullPointerException`, `DmlException`).
- **Custom exceptions:** Defined by developers for specific business scenarios.

Mechanism

- Uses `try`, `catch`, and `finally` blocks.
 - **try:** Code that may cause exception.
 - **catch:** Handles the exception.
 - **finally:** Always executes (cleanup).

Benefits

- Prevents application failures.
- Provides user-friendly error messages.
- Ensures smooth transaction rollbacks.

12. Test Classes

1. Purpose

- Validate Apex code functionality.
- Ensure code is **deployable to production** (minimum 75% coverage required).
- Prevent bugs and unintended behavior in real-time usage.

2. Features

- Written using `@isTest` annotation.
- Run in a **test context** (no impact on org data).
- Support **positive, negative, and bulk testing**.

3. Use Cases

- Verifying **trigger logic** when records are inserted or updated.
- Testing **Batch, Queueable, and Future methods**.
- Ensuring **custom validations and exceptions** work as expected.

The screenshot displays the Salesforce Apex Test Execution interface. On the left, a sidebar contains navigation links for various setup and development tools. The main panel, titled 'Apex Test Execution', provides a summary of test runs and a detailed table of results. The table includes columns for Status, Class, Method, Pass/Fail, Error Message, and Stack Trace. A specific test run for 'WeatherRecordHandlerTest' is highlighted, showing a successful outcome with a duration of 0:00. The error message section provides further details about the test execution, including the class and line number where an exception occurred.

13. Asynchronous Processing

1. Purpose

- Run long-running or resource-intensive tasks **outside the main transaction**.
- Improves system performance and **avoids hitting governor limits**.

2. Types

- **Future Methods** → Lightweight background tasks.
- **Queueable Apex** → Supports job chaining and monitoring.
- **Batch Apex** → Processes millions of records in chunks.
- **Scheduled Apex** → Runs jobs at defined times.

3. Use Cases

- Making **API callouts** to weather services.
- Performing **large data updates** (e.g., refreshing weather history).
- Running **nightly or weekly scheduled jobs**.
- Sending **bulk notifications or emails**
- Helps handle **large data volumes efficiently**
- Provides **flexibility** in executing jobs.

