1. Create a custom Object “Student” with following fields: 1. Student Name (Default field and Type: Text). 2. Fee (field type: Currency) 3. Year (field type: Picklist, picklist values: First Year, Second Year, Third Year) Write down the steps to create an object and field wherever required.

Ans:

Creating “Student” object:

1. In your Salesforce org, click the gear icon, and select Setup.
2. Click the Object Manager tab
3. Click Create > Custom Object in the top-right corner.
4. In the Label section, enter “Student”. The Object Name and Record Name fields will auto-fill with the same name.
5. For Plural Label, enter “Students”.
6. Scroll to the bottom of the page, and select the checkbox Launch New Custom Tab Wizard after saving this custom object. Selecting this box will add your custom object as a tab in Salesforce.
7. Click Save.
8. On the New Custom Object Tab page, click the Tab Style field, and choose a style. The style sets the icon to display in the UI for the object.
9. Click Next, Next, and Save.

“Student” object is created successfully.

Creating “Student Name” field in “Student” object:

1. From Setup, go to Object Manager.
2. Select the “Student” Object.
3. In the sidebar, click Fields & Relationships.
4. Click New in the top right.
5. For data type, select “Text” data type.
6. Click Next.
7. Fill out the following:
    a. Field Label: Student Name
8. Check the Required box.
9. Click Next, Next again, and then Save.

“Student Name” field is created successfully in “Student” object.

Creating “Fee” field in “Student” object:

1. From Setup, go to Object Manager.
2. Select the “Student” Object.
3. In the sidebar, click Fields & Relationships.
4. Click New in the top right.
5. For data type, select “Currency” data type.
6. Click Next.
7. Fill out the following:
    a. Field Label: Fee
8. Check the Required box.(optional)

9. Click Next, Next again, and then Save.

“Fee” field is created successfully in “Student” object.

Creating “Year” field in “Student” object:

1. From Setup, go to Object Manager.
2. Select the “Student” Object.
3. In the sidebar, click Fields & Relationships.
4. Click New in the top right.
5. For data type, select “Picklist” data type.
6. Click Next.
7. Fill out the following:
    a. Field Label: Year
8. Select Enter values, with each value separated by a new line.
9. Enter the following values in the provided box:
    a. First Year
    b. Second Year
    c. Third Year
10. Check the Required box.(optional)
11. Click Next, Next again, and then Save.

“Year” field is created successfully in “Student” object.

2. Explain the following with an example in Salesforce :
    a) Objects b) Records c) Fields

Ans:

a) Objects:
Objects are the backbone of Salesforce's data model. They organize data into structured categories, similar to tables in a database. Objects allow you to store data relevant to your business processes. There are two main types:

Standard Objects: These are provided by Salesforce out-of-the-box to meet common business needs. Examples: `Account`, `Contact`, `Opportunity`, `Lead`, `Case`.

Custom Objects: These are created by Salesforce users to capture data that is unique to their specific business. You can create custom objects for things like "Project," "Event," or "Job Application" based on your business requirements.

Example:
The `Account` object stores data about businesses or companies you work with. Each business (like ABC Corp) would be one record inside this object.

b) Records:
A Record is an instance of an object, similar to a single row in a database table. Every record contains specific details about one item in the object. Think of records as the real-world data points you work with.

In the Account object, each company you interact with would have its own record. For instance:

Record 1:
Name: ABC Corp
Industry: Manufacturing
Revenue: $5M
Record 2:
Name: XYZ Inc
Industry: Healthcare
Revenue: $12M

Each record captures the details of a unique company (in the case of the Account object), and Salesforce organizes these records under the object.

c) Fields:
Fields define what kind of data you want to store for each record in an object. These are similar to columns in a database table, where each column holds a specific type of data.

Each object can have a variety of fields to capture different types of information, such as:

- Text Fields: Store text data like the company name or description.
- Number Fields: Store numeric data such as annual revenue or employee count.
- Picklist Fields: Allow users to select from a predefined list of values, such as industry type (Manufacturing, Healthcare, etc.).
- Date Fields: Capture dates such as the contract start date or last contact date.

Example Fields in the `Account` object:

- Name: A text field storing the name of the company (e.g., ABC Corp).
- Industry: A picklist field allowing users to choose from options like Manufacturing, Healthcare, or Technology.
- Revenue: A currency field storing the company's revenue (e.g., $5M).

Full Example in Salesforce:

1. Object: `Account` (a Standard Object that holds company data).
2. Record: One specific company that you do business with, such as "ABC Corp."
3. Fields:
   - Name: ABC Corp
   - Industry: Manufacturing
   - Revenue: $5M
   - Phone: +1-555-1234
   - Website: www.abccorp.com

3. With a neat sketch, outline the various layers of Salesforce Architecture.

Ans:

https://www.javatpoint.com/salesforce-architecture

4. Create a test class for the following Apex class that converts a list of strings to uppercase:

```
public class StringUtils {
public List<String> convertToUpper(List<String> inputList) {
for (Integer i = 0; i < inputList.size(); i++)
{
inputList[i] = inputList[i].toUpperCase(); }
return inputList; } }
```

Write all tests that handle **empty lists and null values.**

Ans:

```
@isTest
public class StringUtilsTest {

  // Test case 1: Convert a valid list of strings to uppercase
  @isTest
  static void testConvertToUpper_ValidList() {
    // Arrange: create a list of strings
    List<String> inputList = new List<String>{'apple', 'banana', 'Cherry'};

    // Act: Call the convertToUpper method
    List<String> resultList = new StringUtils().convertToUpper(inputList);

    // Assert: Verify that the strings are converted to uppercase
    System.assertEquals('APPLE', resultList[0], 'First string should be in uppercase');
    System.assertEquals('BANANA', resultList[1], 'Second string should be in uppercase');
    System.assertEquals('CHERRY', resultList[2], 'Third string should be in uppercase');
  }

  // Test case 2: Handle an empty list
  @isTest
  static void testConvertToUpper_EmptyList() {
    // Arrange: create an empty list
```

```
    List<String> inputList = new List<String>();

    // Act: Call the convertToUpper method
    List<String> resultList = new StringUtils().convertToUpper(inputList);

    // Assert: Verify that the result list is still empty
    System.assertEquals(0, resultList.size(), 'The result should be an empty list');
}

// Test case 3: Handle a null list
@isTest
static void testConvertToUpper_NullList() {
    // Arrange: Set inputList to null
    List<String> inputList = null;

    try {
        // Act: Call the convertToUpper method
        List<String> resultList = new StringUtils().convertToUpper(inputList);
        // Assert: This line should not be executed, an exception is expected
        System.assert(false, 'Expected a NullPointerException to be thrown');
    } catch (NullPointerException e) {
        // Assert: Verify that a NullPointerException is thrown
        System.assert(true, 'NullPointerException was thrown as expected');
    }
}

// Test case 4: Handle a list with mixed-case strings
@isTest
static void testConvertToUpper_MixedCaseList() {
    // Arrange: create a list with mixed-case strings
    List<String> inputList = new List<String>{'aPpLe', 'BaNaNa', 'ChErRy'};

    // Act: Call the convertToUpper method
    List<String> resultList = new StringUtils().convertToUpper(inputList);
```

```
    // Assert: Verify that all strings are converted to uppercase
    System.assertEquals('APPLE', resultList[0], 'First string should be in uppercase');
    System.assertEquals('BANANA', resultList[1], 'Second string should be in uppercase');
    System.assertEquals('CHERRY', resultList[2], 'Third string should be in uppercase');
  }
}
```

5. Create a trigger “insertTrigger” on “Branch” Object: Insert a record on Branch Object and invoke a Trigger for the event “afterTrigger”.
   When a new record is created with BranchName=’NewBranch’, then a new record should be created on Teacher with TeacherName=’NewBranchTeacher’.

Ans:

```
trigger insertTrigger on Branch (after insert) {
  // List to store Teacher records to insert
  List<Teacher__c> teachersToInsert = new List<Teacher__c>();

  // Loop through each inserted Branch record
  for (Branch__c branch : Trigger.New) {
    // Check if the BranchName is 'NewBranch'
    if (branch.BranchName__c == 'NewBranch') {
      // Create a new Teacher record
      Teacher__c newTeacher = new Teacher__c();
      newTeacher.TeacherName__c = 'NewBranchTeacher';

      // Add it to the list of teachers to be inserted
      teachersToInsert.add(newTeacher);
    }
  }
  // Insert all Teacher records after processing
  if (!teachersToInsert.isEmpty()) {
    insert teachersToInsert;
  }
}
```

6. Perform the following DML operations on Account Object:
   i) Insert ii) Update iii) Upsert iv) Delete v) Undelete

Ans:

i) Insert: Adding a new record to the Account object.

```
// Create a new Account record
Account newAccount = new Account();
newAccount.Name = 'Acme Corporation';
newAccount.Phone = '123-456-7890';

// Insert the Account record
insert newAccount;
```

ii) Update: Modifying an existing Account record.

```
// Fetch an existing Account record (assuming you know the ID or using a SOQL query)
Account existingAccount = [SELECT Id, Name, Phone FROM Account WHERE Name = 'Acme
Corporation' LIMIT 1];

// Modify the fields of the Account
existingAccount.Phone = '987-654-3210';

// Update the Account record
update existingAccount;
```

iii) Upsert: Insert the record if it doesn’t exist, or update it if it does exist.

```
// Create an Account record with a known external ID or use the record’s Salesforce ID for upsert
Account upsertAccount = new Account();
upsertAccount.Name = 'Acme Corporation';  // If 'Acme Corporation' doesn't exist, it will insert
the record
upsertAccount.Phone = '555-555-5555';    // If it exists, it will update the phone number

// Use upsert operation (using Name as an external ID or Salesforce ID)
upsert upsertAccount;
```

iv) Delete: Deleting an existing Account record.

```
// Fetch an existing Account record
Account accountToDelete = [SELECT Id FROM Account WHERE Name = 'Acme Corporation'
LIMIT 1];

// Delete the Account record
delete accountToDelete;
```

v) **Undelete: Restoring a soft-deleted Account record.**

```
// Query the Account from the Recycle Bin (where IsDeleted = true)
Account deletedAccount = [SELECT Id FROM Account WHERE Name = 'Acme Corporation' AND
IsDeleted = TRUE LIMIT 1];

// Undelete the Account record
undelete deletedAccount;
```

7. Explain the @isTest annotation in Apex with examples. Why is it used, and what are its key benefits?

Ans:

The @isTest annotation in Apex is used to define classes and methods that contain test code in Salesforce. It is an essential part of writing unit tests to ensure the quality and functionality of Apex code, like triggers, classes, and other components. Salesforce mandates that at least 75% of Apex code must be covered by unit tests before deployment to production.

Why is @isTest Used?

1. **Code Quality**: It helps ensure that the code behaves as expected and catches potential bugs before deployment.
2. **Code Coverage**: Salesforce requires 75% code coverage for deployment to production. Code coverage refers to how much of the code has been tested by unit tests.
3. **Isolation**: Test methods are executed in isolation and do not affect the actual data in the organization, making them safe for testing.
4. **Governance**: Salesforce uses test methods to enforce best practices like resource limits, governor limits, and ensuring the code runs efficiently.

Key Benefits of @isTest Annotation:

1. **Test Data Isolation**: Test data created in @isTest classes/methods is not saved in the database and is automatically rolled back after the test completes. This prevents data pollution.
2. **Parallel Testing**: Test methods can be run in parallel, which speeds up the execution of multiple tests.
3. **Improved Code Coverage**: Salesforce uses tests to measure code coverage, ensuring that all possible code paths are tested.
4. **Catch Errors Early**: Errors and issues in business logic can be detected early in the development process.
5. **Deployment to Production**: Salesforce enforces 75% code coverage as a requirement for deploying Apex code to production environments. This ensures the robustness of the code.

Example of @isTest Annotation in Apex:

```
@isTest
public class AccountTest {

  @isTest
  static void testAccountInsertion() {
    // Arrange: Set up the test data
    Account acc = new Account();
    acc.Name = 'Test Account';

    // Act: Perform the DML operation
    insert acc;

    // Assert: Verify that the account was inserted
    Account insertedAcc = [SELECT Id, Name FROM Account WHERE Name = 'Test Account' LIMIT
1];
    System.assertEquals('Test Account', insertedAcc.Name);
  }
}
```

8. Perform the following DML operations on the Salesforce Object using Apex class. Insert a record on 'Student' object with Student Name = 'CSE_Student', in which the BranchName is a Master-Detail field. So, assign the record Id of the Branch = 'CSE' for the field BranchName in 'Student' object

Ans:

```
public class StudentRecordCreator {

  // Method to insert a student record with a branch 'CSE'
  public void insertStudentWithBranch() {
    // Step 1: Query the Branch record where Branch Name is 'CSE'
    Branch__c cseBranch = [SELECT Id FROM Branch__c WHERE Name = 'CSE' LIMIT 1];

    // Step 2: Create a new Student record and assign the Branch Id to the BranchName field
    Student__c newStudent = new Student__c();
```

```
    newStudent.Name = 'CSE_Student';  // Setting the Student Name
    newStudent.BranchName__c = cseBranch.Id;  // Assigning the BranchId to the Master-Detail field

    // Step 3: Insert the Student record into the database
    insert newStudent;

    // Optionally, output the inserted record details for confirmation (only for logs or debug)
    System.debug('New student record inserted: ' + newStudent.Id);
  }
}
```

9. Explain why formula fields are useful.Create a simple formula to display an account field on the contact detail page.

Ans:

Why Formula Fields Are Useful

Formula fields in Salesforce are custom fields that automatically calculate and display values based on predefined logic. They are extremely powerful because:

1. **Real-Time Calculations**: Formula fields are automatically calculated whenever the record is accessed or updated. No manual intervention is required.
2. **Cross-Object Data Access**: Formula fields can reference fields from related objects, allowing data from parent records to be displayed or used in child records.
3. **Simplification**: They reduce the need for custom code or workflows to perform calculations or display related data.
4. **Data Consistency**: Formula fields ensure that data is always accurate and consistent since they are derived directly from the source fields.
5. **No DML Needed**: Unlike traditional fields, formula fields don't require DML (Data Manipulation Language) operations to update values, which helps reduce system resource usage.
6. **Read-Only Fields**: Formula fields are always read-only, ensuring that users or automation processes cannot modify their calculated values.

Example: Formula to Display an Account Field on the Contact Detail Page

If you want to display a field from the related Account object on the Contact detail page, you can create a formula field on the Contact object.

Scenario:

You want to display the Account Name on the Contact record without needing to manually query or navigate to the Account record.

Steps to Create a Formula Field:

1. **Navigate to the Object Manager**:
    - Go to Setup → Object Manager → Contact.
2. **Create a New Field**:
    - Under the **Fields & Relationships** section, click **New**.
    - Choose **Formula** as the field type, and click **Next**.
3. **Configure Formula Field**:
    - Enter the **Field Label** (e.g., "Account Name on Contact").
    - The **Field Name** will be auto-populated, but you can modify it.
    - Choose **Text** as the formula return type, and click **Next**.
4. **Build the Formula**:
    - In the formula editor, click **Insert Field**.
    - Navigate to **Contact → Account → Name**, and insert the field.
    - Your formula should look like this:

      Account.Name
5. **Verify and Save**:
    - Click **Next**, and configure the field-level security.
    - Choose which page layouts will display this field, and click **Save**.

Result:

Now, when viewing a Contact record, you'll see a formula field that displays the related Account's Name directly on the Contact detail page.

Key Benefits of This Formula:

- **Real-Time Data**: The Account's name will always be displayed and updated in real-time, reflecting any changes made to the Account record.
- **Cross-Object Reference**: You can pull data from the parent Account object without duplicating information, ensuring that updates to the Account are reflected automatically in the Contact record.
- **No Code**: This functionality is achieved without writing any code, making it easy to implement and maintain.

Formula fields like this enhance data visibility, streamline business processes, and reduce the need for custom development.

10. Describe what a roll up summary field is.Create a roll up summary field where the total price of all products related to an opportunity is displayed.

Ans:

What is a Roll-Up Summary Field?

A **Roll-Up Summary Field** in Salesforce is a special type of field available on **Master-Detail** relationships. It allows users to automatically calculate and display aggregated data from a child object on the parent object. The roll-up summary field can perform calculations such as:

1. **COUNT**: Count the number of child records.
2. **SUM**: Calculate the total value of a numeric field from child records.
3. **MIN**: Display the minimum value of a numeric field from child records.
4. **MAX**: Display the maximum value of a numeric field from child records.

Roll-up summary fields are only available on **Master-Detail relationships**, meaning the parent (master) object must have a Master-Detail relationship with the child (detail) object. The most common use case is calculating values like total sales, total quantity, or total price from related records.

Roll-Up Summary Field Example: Total Price of All Products Related to an Opportunity

In this example, we will create a roll-up summary field that calculates the **Total Price of all Products** related to an **Opportunity**. This is useful in scenarios where each Opportunity has multiple Products, and you want to sum the price of those products at the Opportunity level.

Objects Involved:

- **Opportunity**: The parent object.
- **OpportunityLineItem (Opportunity Product)**: The child object, representing the products related to an opportunity.

Steps to Create a Roll-Up Summary Field:

1. **Navigate to the Opportunity Object**:
    - Go to **Setup** → **Object Manager** → **Opportunity**.
2. **Create a New Field**:
    - Under the **Fields & Relationships** section, click **New**.
    - Choose **Roll-Up Summary** as the field type, and click **Next**.
3. **Define the Roll-Up Summary Field**:
    - **Field Label**: Enter a label like "Total Product Price".
    - **Field Name**: This will be auto-generated, but you can modify it.
    - **Summarized Object**: Choose OpportunityLineItem (the child object representing the products related to the opportunity).
    - **Roll-Up Type**: Select **SUM** (since we want to sum up the total price).
4. **Select the Field to Aggregate**:

- After selecting the summarized object (OpportunityLineItem), choose the field you want to summarize.
- Choose **TotalPrice** as the field to sum up (this represents the total price of each product).
- Click **Next**.

5. **Set Field-Level Security and Add to Layout**:
   - Choose which profiles can see this field.
   - Select the page layouts where you want to display this field.
6. **Save the Field**:
   - Click **Save** to create the roll-up summary field.

Result:

Now, on each **Opportunity** record, there will be a **Total Product Price** field that displays the sum of all the related **Opportunity Product (OpportunityLineItem)** prices. Whenever you add or remove products from an opportunity, the roll-up summary field will automatically recalculate the total.

Key Benefits of Roll-Up Summary Fields:

1. **Automated Calculations**: Roll-up summary fields automate the process of calculating aggregated data from related records, eliminating the need for custom code.
2. **Real-Time Updates**: The field is updated automatically in real time when related records are added, removed, or modified.
3. **Simplified Reporting**: Roll-up summary fields can be used in reports, dashboards, and workflows to enhance decision-making and provide better insights into sales, inventory, or other important data points.
4. **No Need for Triggers**: Without roll-up summary fields, you would need to use Apex triggers to manually calculate aggregate values, which can be more complex and require additional maintenance.

Limitations:

- Roll-up summary fields only work on **Master-Detail relationships**.
- They cannot be used on **Lookup relationships** unless you convert the Lookup relationship to a Master-Detail relationship.

11. What are the steps to create an Apex class with methods and test your Apex Classes with an example?

Ans:

1. Create an Apex Class with Methods

An Apex class is a blueprint for creating objects with properties and methods that define the behavior. Here's a simple example of an Apex class with a couple of methods.

Example: Apex Class

```
public class Calculator {
```

```
    // Method to add two numbers
    public Integer addNumbers(Integer num1, Integer num2) {
        return num1 + num2;
    }

    // Method to subtract two numbers
    public Integer subtractNumbers(Integer num1, Integer num2) {
        return num1 - num2;
    }

    // Method to multiply two numbers
    public Integer multiplyNumbers(Integer num1, Integer num2) {
        return num1 * num2;
    }
}
```

2. **Test Your Apex Class**

Testing is a critical part of the Salesforce development lifecycle. Salesforce enforces a rule that 75% of Apex code must be covered by tests before deploying to production. For that, you need to write a test class.

Steps to Create a Test Class:

1. **Create a Test Class**: A test class is an Apex class annotated with @isTest. Each method in the test class that tests logic is annotated with @isTest.
2. **Write Test Methods**: Write methods to test the behavior of the original class. Each method should test a specific functionality and use System.assert() to check the correctness of the results.
3. **Use Test.startTest() and Test.stopTest()**: These methods are used when testing governor limits or asynchronous operations, but for simple operations, they are not mandatory.

Example: Test Class for the Calculator Class

```
@isTest
public class CalculatorTest {

    // Test method for addition
    @isTest
    static void testAddNumbers() {
        // Arrange: Create an instance of Calculator class
```

```
        Calculator calc = new Calculator();

        // Act: Call the method and store the result
        Integer result = calc.addNumbers(10, 5);

        // Assert: Check if the result is as expected
        System.assertEquals(15, result, 'The addNumbers method should return 15');
    }

    // Test method for subtraction
    @isTest
    static void testSubtractNumbers() {
        Calculator calc = new Calculator();
        Integer result = calc.subtractNumbers(10, 5);
        System.assertEquals(5, result, 'The subtractNumbers method should return 5');
    }

    // Test method for multiplication
    @isTest
    static void testMultiplyNumbers() {
        Calculator calc = new Calculator();
        Integer result = calc.multiplyNumbers(10, 5);
        System.assertEquals(50, result, 'The multiplyNumbers method should return 50');
    }
}
```

3. Run the Test Class

Once you've written your test class, you need to run it to ensure everything works as expected.

Steps to Run Tests:

1. **Using the Developer Console**:
    - Open the **Developer Console** in Salesforce.
    - Navigate to **Test** → **New Run**.
    - Select your test class (CalculatorTest) and click **Run**.

- You can view the results in the **Tests** tab, including whether the test passed or failed, and any debug logs.

2. **Using Setup**:
    - Go to **Setup** → **Apex Test Execution**.
    - Click **Select Tests** and choose the test class you want to run.
    - Click **Run**.

4. Check Test Coverage

After running the tests, you should check the **test coverage** to ensure that your code meets the minimum requirement of 75% coverage.

- **In Developer Console**:
    - After running the test, go to the **Code Coverage** tab. You will see the percentage of the code that was covered by the test.
    - You can also view which lines were tested and which lines were not.

Benefits of Testing Apex Classes:

1. **Ensures Code Quality**: Testing helps to validate that your business logic works as expected.
2. **Prevents Bugs**: Automated tests catch issues early in the development cycle.
3. **Enforced by Salesforce**: Salesforce requires a minimum of 75% code coverage for deployment to production.
4. **Avoids Future Failures**: When you add new functionality, tests help ensure that new changes don't break existing functionality.

12. State sObject datatype. Describe in detail about its type with suitable example.

Ans: