APEX OOPS CONCEPT

1. What is Object-Oriented Programming (OOP), and how is it implemented in Apex?

Object-Oriented Programming (OOP) is a way of writing code that focuses on **objects**, which can contain data (fields) and code (methods). It emphasizes **encapsulation, inheritance, polymorphism**, and **abstraction** to create reusable and modular code.

In Apex, OOP is implemented through:

- **Classes**: Define templates for creating objects.
- **Objects**: Represent instances of classes.
- **Methods**: Encapsulate behavior.
- **Inheritance and Interfaces**: Enable code reuse and flexibility.

  Encapsulation: Keep the details inside the object. For example, a car has an engine, but you only use the steering wheel and pedals—you don't need to see how the engine works.

  Inheritance: Reuse existing code. If you already have a "Vehicle" class, you can make a "Car" class that takes all the vehicle features and adds its own extras.

  Polymorphism: Use one thing in different ways. For instance, a "drive" function can work for cars, bikes, or trucks but behave slightly differently for each.

  Abstraction: Focus on what's important and ignore the rest. For example, when using a TV remote, you care about pressing buttons, not how the signals are sent inside.

  Interface: Imagine an interface called **RemoteControl**. It defines the rules for any remote control: it must have buttons for **powerOn** and **powerOff**. The exact way these buttons work depends on the device (TV, AC, etc.).

2. What are classes and objects in Apex?

Class: A blueprint or template that defines the properties and methods for creating objects.

Object: An instance of a class that holds actual data and behavior defined by the class.

3. Explain the difference between a class and an object in Salesforce Apex.

Class	Object
A class is a blueprint or template for creating objects.	An object is an instance of a class that contains actual data.
Defines properties and methods that objects will use.	Represents a specific example of the class.
Example: Car class.	Example: myCar (an object of the Car class).

4. Explain the difference between global and public access modifiers in Apex.

Access Modifier	Description
Global	- Can be accessed from any class or trigger within the same org or across other orgs (if included in a managed package).
Public	- Can be accessed by any class or trigger in the same namespace but **not** outside of it.

When to use:

- **Global**: Use when you want to expose a class, method, or variable to other Salesforce orgs or in managed packages.
- **Public**: Use when you only need access within your Salesforce org or namespace.

5. What are primitive data types in Apex? Provide examples.

Primitive data types in Apex are basic data types that are predefined and simple to use. They represent single values (numbers, text, etc.).

Examples of primitive data types:

- **Integer**: Represents a whole number.
  - ☐ Example: Integer count = 10;
- **Double**: Represents a floating-point number.
  - ☐ Example: Double price = 99.99;
- **String**: Represents a sequence of characters (text).
  - ☐ Example: String name = 'John Doe';
- **Boolean**: Represents a true/false value.

  Example: Boolean isActive = true;

- **Date**: Represents a date without a time (e.g., YYYY-MM-DD).
  - ☐ Example: Date orderDate = Date.today();
- **Datetime**: Represents a date and time (e.g., YYYY-MM-DD HH:MM:SS).
  - ☐ Example: Datetime eventTime = Datetime.now();
- **ID**: Represents a Salesforce record identifier.
  - ☐ Example: ID accountId = '0012Y00000BHzPq';

---

6. What are non-primitive data types in Apex? Provide examples.

Non-primitive data types in Apex are more complex structures that allow you to store multiple values. These data types are **objects or collections** and can hold more than one value or a reference to another object.

Examples of non-primitive data types:

- **Lists**: A collection of elements, typically of the same type.
    - Example: List<Integer> scores = new List<Integer>{90, 85, 78};
- **Sets**: A collection of unique elements, where the order doesn't matter.
    - Example: Set<String> uniqueNames = new Set<String>{'Alice', 'Bob', 'Charlie'};
- **Maps**: A collection of key-value pairs.
    - Example: Map<String, Integer> cityPopulation = new Map<String, Integer>{'New York' => 8419600, 'Los Angeles' => 3980400};
- **Custom Objects**: These represent Salesforce records like Account, Contact, or custom objects.
    - Example: Account acc = new Account(Name = 'Acme Corporation');
- **sObjects**: A special non-primitive type that represents a Salesforce record.
    - Example: Account acc = [SELECT Name FROM Account LIMIT 1];

---

7. What are variables in Apex, and how are they declared?

Variables in Apex are used to store data values. They hold data that can be referenced and manipulated during code execution.

To declare a variable in Apex, you define the data type followed by the variable name. You can also initialize the variable with a value when declaring it.

8. What are the different types of operators in Apex? Provide examples.

Apex supports several types of operators for performing operations on variables and values. They can be categorized into **Arithmetic**, **Comparison**, **Logical**, **Assignment**, **Bitwise**, and **Ternary** operators.

9. How do logical operators (AND, OR, NOT) work in Apex?

Logical operators in Apex are used to combine multiple boolean expressions to evaluate more complex conditions.

- **AND** (&&): Returns true if both conditions are true.

- **OR** (||): Returns true if at least one of the conditions is true.
- **NOT** (!): Reverses the boolean value (true becomes false and vice versa).

10. What is a function in Apex, and how is it defined?

A function (or method) in Apex is a block of code that performs a specific task. Functions can take parameters and return values. They help in organizing code into reusable and modular units.

Defining a Function:

```
public returnType functionName(parameterType parameterName)
{
    // Function logic
    return returnValue;
}
```

Example:

```
apex
Copy code
public Integer addNumbers(Integer a, Integer b) {
    return a + b;  // Function that returns the sum of two
numbers
}
```

11. What is the difference between a static function and an instance function in Apex?

- **Static Function**: A function defined with the static keyword. It belongs to the class rather than an instance of the class. It can be called without creating an object of the class.

Example:

```
public class Mylass {
    public static Integer multiply(Integer a, Integer b) {
        return a * b;
    }
}
```

- **Instance Function**: A function that is tied to an object instance. You must create an instance of the class to call the instance function.

Example:

```
public class MyClass {
    public Integer subtract(Integer a, Integer b) {
        return a - b;
    }
}
```

12. Explain function overloading with an example in Apex.

Function overloading is the ability to define multiple functions with the same name but different parameter lists (either in number, type, or both). This allows for different behaviors based on the arguments passed to the function.

Example:

```
public class MathOperations {
    public Integer add(Integer a, Integer b) {
        return a + b;
    }

    public Double add(Double a, Double b) {
```

```
        return a + b;
    }

    public Integer add(Integer a, Integer b, Integer c) {
        return a + b + c;
    }
}
```

In this example, the add function is overloaded to handle different types and numbers of parameters.

13. What is the purpose of function parameters in Apex?

Function parameters in Apex allow you to pass values into a function. They are used to provide input data to the function so that it can perform specific operations based on those inputs.

Example:

```
public Integer multiply(Integer a, Integer b) {
    return a * b;
}
```

14. What is the difference between a void function and a function that returns a value?

- **Void Function**: A function that does not return any value. It performs some task but doesn't send any data back to the caller.

Example:

```
public void printMessage(String message) {
    System.debug(message);
```

```
}
```

- **Function that Returns a Value**: A function that returns a specific value after performing some logic.

Example:

```
public Integer addNumbers(Integer a, Integer b) {
    return a + b;
}
```

15. What are constructors in Apex, and how do you define them in a class?

A **constructor** is a special method in a class that is automatically invoked when an object is created. It is used to initialize the object's properties.

Defining a Constructor in Apex:

- Constructor name must match the class name.
- Can accept parameters to initialize properties.

16. What is the purpose of the this keyword in Apex?

The this keyword in Apex refers to the **current instance** of the class. It is primarily used to:

1. Differentiate between instance variables and method parameters when they have the same name.
2. Access instance methods or properties within the class.

17. What is Encapsulation, and how is it implemented in Apex?

Encapsulation is an OOP principle that restricts direct access to the internal details of a class and exposes functionality through controlled methods. This ensures that the class data is protected and accessed only in a controlled way.

In Apex, encapsulation is implemented by:

1. Declaring variables as private.
2. Using **getter** and **setter** methods to control access to these variables.

18.Why is it important to use access modifiers like private, public, protected, and global in Apex?

Access modifiers control the visibility of classes, methods, and variables. They are important because:

1. **Protect Data**: Prevent unauthorized or accidental modification of sensitive data.
2. **Encapsulation**: Expose only what is necessary for external use.
3. **Reusability**: Promote modularity and reuse without exposing implementation details.

Access Modifiers in Apex:

Modifier	Visibility
private	Accessible only within the same class.
public	Accessible anywhere within the same namespace.
protected	Accessible within the class and its subclasses.
global	Accessible across namespaces (e.g., for managed packages).

19. What is inheritance, and how does it work in Apex?

Inheritance is an OOP concept where one class (child or subclass) can inherit properties and methods from another class (parent or superclass). It allows for **code reuse** and promotes modularity.

How it works in Apex:

- The child class uses the extends keyword to inherit from the parent class.
- The child class can use the parent class's methods and properties or override them.
- The extends keyword is used to indicate that a class (child) inherits from another class (parent).

20. What is the use of the super keyword in Apex? Provide an example of calling a parent class method.

The **super keyword** is used to call a method or constructor of the parent class from a child class. It helps to extend or reuse the functionality of the parent class.

21. How do you override methods in Apex? Provide an example.

Method overriding in Apex allows a child class to provide a specific implementation of a method that is already defined in its parent class. To override a method, you use the **@Override** annotation before the method in the child class.

Example: Overriding a Method

```
// Parent class
public class Animal {
    public virtual void speak() {
        System.debug('The animal makes a sound.');
    }
}

// Child class
public class Cat extends Animal {
    @Override
    public void speak() {
        System.debug('The cat meows.');
    }
}
```

```
// Usage
Cat myCat = new Cat();
myCat.speak(); // Outputs: The cat meows.
```

Key Points for Overriding:

- The parent method must be marked as virtual or abstract.
- The overriding method must use the @Override annotation.

22. What is polymorphism, and how is it implemented in Apex?

Polymorphism is an OOP concept where an object or method can take multiple forms. It allows a single interface to represent different types of behavior.

In Apex, polymorphism is implemented through:

1. **Method Overloading**: Same method name with different parameters in the same class.
2. **Method Overriding**: A child class provides a specific implementation of a method defined in the parent class.

23. How do method overloading and method overriding differ in Apex?

Aspect	Method Overloading	Method Overriding
Definition	Same method name, different parameters.	Same method name, same parameters, but in a child class.
Implemented in	Same class.	Parent and child classes.
Annotation	No annotation needed.	Requires @Override annotation.
Purpose	Provides multiple versions of a method.	Modifies or extends the behavior of a parent method.

24. Can static methods be overridden in Apex? Why or why not?

No, **static methods cannot be overridden** in Apex because:

1. Static methods belong to the **class itself** and not to any specific object.
2. Overriding is an instance-based concept where the child class modifies the behavior of the parent class method.

However, static methods can be **redeclared** in the child class, but this is not true overriding.

Example:

```
public class ParentClass {
    public static void display() {
        System.debug('Parent Class Display Method');
    }
}

public class ChildClass extends ParentClass {
    public static void display() {
        System.debug('Child Class Display Method');
    }
}

// Usage
ParentClass.display(); // Outputs: Parent Class Display
Method
ChildClass.display();  // Outputs: Child Class Display
Method
```

25. What is abstraction, and how is it applied in Apex?

Abstraction is the process of hiding implementation details and showing only the essential features of an object or concept. It allows you to focus on what an object does rather than how it does it.

In Apex, abstraction is implemented using:

- **Abstract Classes**: Define common behavior but defer implementation to child classes.
- **Interfaces**: Define a contract (method signatures) without implementation.

---

26. What are abstract classes in Apex, and how are they used?

An **abstract class** is a class that cannot be instantiated on its own. It provides a base for other classes to build upon and may include:

- Abstract methods (methods without a body).
- Concrete methods (methods with a body).

Usage:

- To enforce a consistent structure in child classes.
- To define shared behavior and properties.

Example:

```
// Abstract class
public abstract class Animal {
    public String name;

    // Abstract method (no body)
    public abstract void speak();

    // Concrete method
    public void displayName() {
```

```
        System.debug('Animal Name: ' + name);
    }
}

// Child class
public class Dog extends Animal {
    @Override
    public void speak() {
        System.debug('The dog barks.');
    }
}

// Usage
Dog myDog = new Dog();
myDog.name = 'Buddy';
myDog.displayName(); // Outputs: Animal Name: Buddy
myDog.speak();       // Outputs: The dog barks.
```

Key Points:

- The speak() method is defined in the child class since it was abstract in the parent class.
- The displayName() method is inherited directly without modification.

---

27. How do you implement interfaces in Apex? Provide an example.

To implement an interface in Apex, a class must define all the methods declared in the interface.

Example:

```
// Interface
```

```
public interface Vehicle {
    void start();
    void stop();
}

// Implementing the interface
public class Car implements Vehicle {
    public void start() {
        System.debug('The car starts.');
    }
    public void stop() {
        System.debug('The car stops.');
    }
}

// Usage
Car myCar = new Car();
myCar.start(); // Outputs: The car starts.
myCar.stop();  // Outputs: The car stops.
```

Key Points:

- The implements keyword is used to implement an interface.
- All methods in the interface must be implemented in the class.

28. What is an interface in Apex, and how is it different from a class?

An **interface** in Apex is a blueprint that defines a contract of methods that a class must implement. Unlike a class, an interface:

- Cannot have concrete (implemented) methods.
- Does not contain variables, constructors, or instance methods.

Key Differences:

Aspect	Interface	Class
Definition	Contains only method signatures (no implementation).	Can contain implemented methods and variables.
Inheritance	A class can implement multiple interfaces.	A class can extend only one class.
Purpose	Used to define a contract for unrelated classes.	Used to define common functionality.

29. What is the difference between static and instance variables/methods in Apex?

Aspect	Static	Instance
Definition	Belongs to the class and shared across all instances.	Belongs to a specific object instance.
Memory	Allocated once for the class.	Each object gets its own copy.
Access	Can be accessed without creating an instance of the class.	Requires an object instance to access.
Use Case	Used for shared data or utilities.	Used for data and behavior specific to an object.

30. What is exception handling in Apex?

Exception handling in Apex is the process of managing errors or unexpected situations that occur during the execution of a program. Apex provides a

mechanism to handle runtime errors, allowing the program to continue or gracefully handle the issue without crashing.

In Apex, exception handling is done using try, catch, finally, and throw statements.

31. How do you handle Apex Governor Limits in OOP-based classes?

Governor Limits in Apex are restrictions set by Salesforce to ensure the efficient use of resources. When using OOP, managing these limits requires careful planning. Some techniques include:

1. **Bulkify Code**: Ensure your classes and methods are designed to handle bulk operations (e.g., handling multiple records at once) to minimize the risk of exceeding limits.
    - Example: Use collections (Lists, Maps, Sets) to process records in bulk.
2. **Use Efficient Queries**: Limit the number of SOQL queries and DML operations by querying only necessary fields and using FOR loops to process records efficiently.
3. **Utilize the @future Annotation**: Offload long-running tasks or complex logic to future methods or asynchronous processes to avoid hitting limits in the main execution context.
4. **Limit Loops and SOQL Queries**: Minimize nested loops and ensure queries are done in bulk (outside of loops).

32. Order of Execution

1. **Loads Initial record.**
2. If the request came from a standard UI edit page, Salesforce runs **system validation** to check the record for page layout-specific rules, field definition, and Maximum field length.
3. Executes **before record-triggered flows**.
4. Executes all before triggers.
5. Runs most **Custom validation.**
6. Executes **duplicate rules**.
7. **Saves the record to the database, but doesn’t commit yet.**

8. Executes **all after triggers**.
9. **Assignment rules.**
10. **Executes auto-response rules.**
11. Executes **workflow rules**. If there are **workflow field updates**,
    - **updates the record again.**
    - Due to Workflow field updates introducing new duplicate field values, executes **duplicate rules again.**
    - If the record was updated with workflow field updates, f**ires before update triggers and after update trigger one more time (and only one more time),** in addition to **standard validations**. **Custom validation rules are not run again.**
12. **Escalation rules.**
13. Executes these **Salesforce Flow automations**
    - Processes
    - Flows launched by processes
    - Flows launched by workflow rules (flow trigger workflow actions pilot)
14. Executes **record-triggered flows** that are configured to run **after the record is saved.**
15. Executes **entitlement rules**.
16. If the record contains a **roll-up summary field** or is part of a cross-object workflow, **performs calculations and updates the roll-up summary field** in the parent record. **Parent record goes through save procedure.**
17. If the parent record is updated, and a grandparent record contains a roll-up summary field or is part of a cross-object workflow, performs calculations and updates the roll-up summary field in the grandparent record. **Grandparent record goes through save procedure.**
18. Executes **Criteria Based Sharing evaluation**.
19. **Commits all DML operations** to the database.
20. Executes **all after-commit logic.**
    - sending email.
    - asynchronous Apex jobs
    - Asynchronous paths in record-triggered flows