We define a Bird class as the base class to hold common attributes for all bird classifications:

birdType: The type of bird (e.g., "Duck," "Hawk").

definingCharacteristic: A defining characteristic of the bird (e.g., "Webbed feet," "Sharp beak").
extinct: A boolean indicating whether the bird is extinct (true or false).
numWings: The number of wings (usually 2).
The preferredFood attribute is a list that stores the types of food the bird prefers to eat.
We provide methods (setPreferredFood and addPreferredFood) to set or add food items to this list.

We create specific subclasses for different bird classifications, such as BirdsOfPrey, FlightlessBird, Owls, Parrots, Pigeons, Shorebirds, and Waterfowl. Each subclass inherits from the Bird class and may have additional attributes that are specific to that classification.

How will your design encapsulate the different types of birds? What fields you expect to have in each class? What is their access and why?

In the design I proposed, encapsulation is achieved through the use of private fields and public methods in each class, ensuring that the internal data is hidden and can only be accessed or modified through well-defined interfaces. Here's how encapsulation is implemented in each class:

Bird Class:

Fields:

private String birdType: Stores the type of the bird.
private String definingCharacteristic: Stores the defining characteristic of the bird.
private boolean extinct: Indicates whether the bird is extinct.
private int numWings: Stores the number of wings.
private List<String> preferredFood: Stores the preferred food items.
Methods:

public Bird(birdType, definingCharacteristic, extinct, numWings, preferredFood): Public constructor to initialize the fields.
public void setPreferredFood(List<String> foodList): Public method to set the preferred food list.
public void addPreferredFood(String foodItem): Public method to add a food item to the preferred food list.
Access Control:

Fields are made private to encapsulate them. Public methods are provided to allow controlled access to and modification of the fields.

Subclasses (BirdsOfPrey, Owls, Parrots, etc.):

Inherited Fields from Bird Class:

The same fields as in the Bird class, such as birdType, definingCharacteristic, etc.
Additional Fields:

Specific fields related to the characteristics of each bird type, such as wingSpan for BirdsOfPrey, facialDisks for Owls, and vocabulary for Parrots.
Additional Methods:

Methods specific to each bird type, such as hunt() for BirdsOfPrey, rotateHead() for Owls, and speak() for Parrots.
Access Control:

The inherited fields from the Bird class are kept private to encapsulate them and ensure they can be accessed or modified only through public methods.
Specific fields related to each bird type may have different access modifiers depending on the specific requirements. For example, if a field should be read-only, you can make it private with a public getter method. If it needs to be set from outside, you can provide a setter method.
The encapsulation design ensures that the internal state of each bird object is protected, and external code can interact with the objects using well-defined and controlled methods. This helps maintain data integrity and allows for future modifications or enhancements to the class without affecting the external code that uses these classes.

Test Plan:

Test individual classes and methods to ensure they work correctly.
Tests:
For each class, we will write unit tests to check the working of constructors, setters, getters, and any other methods.
We can test edge cases, such as creating bird instances with extreme values or invalid input.

Test interactions between different classes.
Tests:
We will create test scenarios that involve multiple bird instances.
We can Test how subclasses interact with the base Bird class and ensure that inheritance and method overriding work as expected.

Test the overall functionality of our program.
Tests:
We will design test cases that cover typical use cases of our bird tracking program.
For example, create bird instances of various types, set their attributes, and perform operations like setting preferred food.

Examples:

Test Case 1: Create Bird Instances

Condition: Testing the creation of instances for different bird classifications.
Example Data:
Create an instance of BirdsOfPrey with "Hawk" as the bird type and other relevant attributes.
Create an instance of Owls with "Barn Owl" as the bird type and other relevant attributes.
Create an instance of Parrots with "African Grey Parrot" as the bird type and other relevant attributes.
Expected Outcome: Confirm that instances of different bird classifications can be created successfully.

Test Case 2: Set and Get Preferred Food

Condition: Testing the setting of preferred food for a bird.
Example Data:
Create a Parrots instance and set its preferred food to a list containing "Seeds," "Fruits," and "Nuts."
Expected Outcome: Verify that the preferred food list is correctly set.

Test Case 3: Test Extinct Birds

Condition: Testing whether the extinct attribute correctly represents whether a bird is extinct.
Example Data:
Create instances of extinct and non-extinct birds, such as a Pigeons instance (extinct) and a Parrots instance (non-extinct).
Expected Outcome: Verify that the extinct attribute reflects the correct status for each bird.

Test Case 4: Test Inheritance and Method Override

Condition: Testing that subclasses inherit attributes and methods from the Bird class and override them as needed.
Example Data:
Create a BirdsofPrey instance with attributes specific to birds of prey and use methods inherited from the Bird class.