Bird Conservatory Management System

Author: Gbadamassi Hanane
Class: CS5010 Lab 1

Welcome

Welcome to the **Bird Conservatory Management System**! This application is designed to help conservationists, zookeepers, and bird enthusiasts manage a complex sanctuary for our feathered friends. Whether you're rescuing a stray duck or finding a home for a majestic eagle, this system handles the logistics so you can focus on the birds.

The system is built to track various bird species, manage their dietary needs, and ensure they are housed safely in appropriate aviaries (we wouldn't want to put a Hawk in with a Dove!).

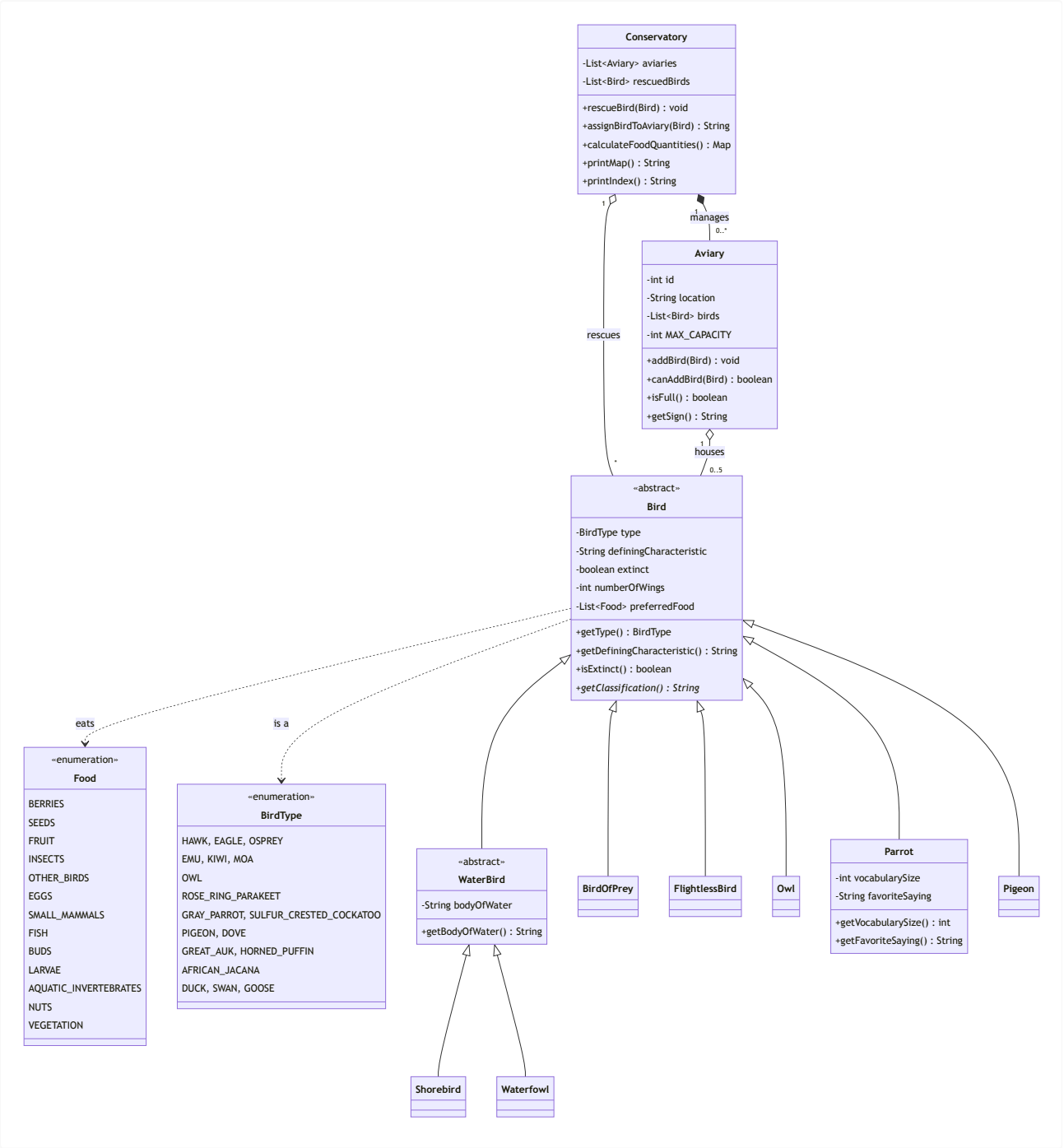
System Architecture

The Bird Conservatory System is designed around a modular object-oriented architecture that mimics the real-world hierarchy of a bird sanctuary.

Core Components

1. **Bird Hierarchy**: At the foundation is the abstract `Bird` class, which defines the common contractual obligations for all species (eating, describing themselves, etc.). Specific families like `BirdOfPrey` or `Waterfowl` extend this to add specialized behaviors (like hunting or swimming) and data (like water sources).
2. **Aviary Management**: The `Aviary` class acts as a container with strict admission policies. It encapsulates the logic for "who can live with whom," preventing invalid state transitions (like adding a predator to a room of prey).
3. **Conservatory Coordinator**: The `Conservatory` class serves as the facade for the entire system. It orchestrates the interaction between birds and aviaries, handling the high-level logic of "rescue," "assign," and "feed" so the user doesn't have to micromanage individual objects.

To help you visualize how these pieces fit together, here is a diagram of the system's "DNA":



Meet the Birds

Our system isn't just a database; it understands that every bird is unique.

The `Bird` Family

At the heart of the system is the **Bird** class. Every bird has some basic traits:

- **Type**: What species is it? (e.g., Hawk, Duck)
- **Characteristics**: What makes it special? (e.g., "Sharp talons")
- **Diet**: What does it eat? (e.g., Seeds, Fish)
- **Wings**: Because while most have 2, accidents happen, and we need to track that.
- **Extinct Status**: Sadly, we sometimes track birds that are no longer with us, like the Great Auk.

We then have tailored families:

- **Birds of Prey**: Hunters like Hawks and Eagles. They contain special logic to ensure they aren't housed with potential "snacks".
- **Water Birds**: Includes **Waterfowl** (Ducks/Swans) and **Shorebirds** (Puffins), which also track which body of water they love.
- **Parrots**: Our chatty friends! We specifically track how many words they know and their favorite saying.
- **Flightless Birds**: Emus, Kiwis, and Moas. They live on the ground.
- **Owls** and **Pigeons**: Independent groups with their own specific behaviors.

The Conservatory: A Simulation

The **Conservatory** acts as the central hub. Think of it as the "Brain" of the operation.

1. Rescuing Birds

When a bird is brought in, we first "rescue" it. This registers the bird in our system without immediately putting it in a cage. This allows us to assess the bird before finding it a home.

2. Housing (The Aviary Logic)

The most complex part of our job is housing. We can't just throw everyone together! The **Aviary** class handles these strict rules:

- **Capacity**: Maximum of 5 birds per aviary. No overcrowding!
- **Safety First**: **Birds of Prey** and **Waterfowl** are territorial. They cannot be mixed with other bird types.
- **Extinct Birds**: We can track extinct birds in our records, but we obviously can't put them in physical aviaries (they would be ghosts!).

3. Kitchen Duty

A hungry bird is an unhappy bird. The Conservatory can automatically generate a **Food Order List**. It looks at every bird in every aviary and calculates exactly how much of each food type (Seeds, Fish, Insects, etc.) needs to be ordered.

Testing Plan

To ensure the system works as expected, we have defined the following test cases. Each case targets a specific condition with example input and expected output. We didn't just guess that this would work; we proved it. We used **JUnit** to create a suite of strict tests:

- **Boundary Testing**: We tried to add a 6th bird to an aviary to make sure the system refused it.
- **Safety Testing**: We tried to put a Hawk in with a Dove. The system correctly blocked the move.
- **Logic Testing**: We calculated food orders for complex mixes of birds to ensure the math was perfect.
- **Search Testing**: We verified that if you stick a Parrot in "Aviary 4", the system can find it instantly.

1. Bird Creation Tests

Test ID	Condition	Example Data	Expected Result
TC-01	Create valid BirdOfPrey	Hawk, 2 wings, [FISH, SMALL_MAMMALS]	Success (Object created)
TC-02	BirdOfPrey with <2 foods	Eagle with [FISH] only	`IllegalArgumentException`
TC-03	BirdOfPrey with >4 foods	Osprey with 5 foods	`IllegalArgumentException`
TC-04	Create extinct FlightlessBird	Moa, extinct=true	`isExtinct()` returns true

TC-05	Parrot with valid vocabulary	Gray Parrot, vocab=50	Success
TC-06	Parrot with vocab >100	Parakeet, vocab=150	`IllegalArgumentException`
TC-07	Shorebird with water source	Puffin, "Pacific Ocean"	`getBodyOfWater()` returns "Pacific Ocean"
TC-08	Null bird type	null type	`IllegalArgumentException`
TC-09	Negative vocabulary	Parrot, vocab=-1	`IllegalArgumentException`
TC-10	Empty body of water	Shorebird, ""	`IllegalArgumentException`

2. Aviary Tests

Test ID	Condition	Example Data	Expected Result
TC-11	Add to empty aviary	Empty aviary + Duck	`canAddBird()` returns true
TC-12	Add extinct bird	Moa to any aviary	`canAddBird()` returns false
TC-13	Aviary capacity limit	Add 6th bird	`canAddBird()` returns false
TC-14	Mix BirdOfPrey with Pigeon	Hawk aviary + Dove	`canAddBird()` returns false
TC-15	Mix FlightlessBird with Owl	Emu aviary + Owl	`canAddBird()` returns false
TC-16	Mix Waterfowl with Parrot	Duck aviary + Parrot	`canAddBird()` returns false
TC-17	Mix compatible birds	Owl aviary + Pigeon	`canAddBird()` returns true
TC-18	Mix Shorebird with Pigeon	Puffin + Dove	`canAddBird()` returns true
TC-19	BirdsOfPrey together	Hawk + Eagle	`canAddBird()` returns true
TC-20	Waterfowl together	Duck + Swan	`canAddBird()` returns true

3. Conservatory Tests

Test ID	Condition	Example Data	Expected Result
TC-21	Rescue new bird	Conservatory + Duck	Bird added to `rescuedBirds`
TC-22	Rescue null bird	null	`IllegalArgumentException`
TC-23	Rescue same bird twice	Duck twice	`IllegalStateException`
TC-24	Assign to new aviary	Empty conservatory + Duck	New aviary created, bird assigned
TC-25	Assign compatible birds	Duck then Swan	Assigned to same aviary
TC-26	Assign incompatible	Hawk then Dove	Assigned to separate aviaries
TC-27	Assign extinct bird	Moa	`IllegalStateException`
TC-28	Maximum 20 aviaries	21st aviary needed	`IllegalStateException`
TC-29	Calculate food - single	Duck in aviary	Returns map with correct quantities
TC-30	Calculate food - overlap	Multiple birds, shared foods	Returns map with summed quantities

TC-31	Lookup bird in aviary	Duck in Aviary 1	Returns "Aviary 1" message
TC-32	Lookup bird not found	Non-existent bird	Returns "not found" message
TC-33	Invalid aviary sign	`getAviarySign(999)`	`IllegalArgumentException`
TC-34	Print empty map	Empty conservatory	Returns "No aviaries"
TC-35	Print map with aviaries	2 aviaries	Returns string containing both
TC-36	Print index alphabetically	Hawk, Duck, Eagle	Returns list ordered: Duck < Eagle < Hawk

---

Why We Built It This Way

Why inheritance?

We used a hierarchy (e.g., `Duck` extends `Waterfowl` extends `WaterBird` extends `Bird`) because it mirrors biology. It allows us to write code once (like "all birds eat") and reuse it, while still allowing specific overrides (like "Parrots speak").

Why is everything private?

You'll see variables like `- type` or `- birds` are locked away (`private`). This is **Encapsulation**. It prevents accidental tampering. You can't just reach in and change a bird's wingspan; you have to ask the bird nicely (use a method).

Why the "rescue" step?

We separated `rescueBird()` from `assignBirdToAviary()` to mimic real life. Sometimes you receive a bird but don't have a cage ready yet. This two-step process offers flexibility.

File Structure

```
lab-5010/
├── src/
│   ├── birds/
│   │   ├── Bird.java
│   │   ├── BirdOfPrey.java
│   │   ├── FlightlessBird.java
│   │   ├── Owl.java
│   │   ├── Parrot.java
│   │   ├── Pigeon.java
│   │   ├── WaterBird.java
│   │   ├── Shorebird.java
│   │   ├── Waterfowl.java
│   │   ├── BirdType.java
│   │   └── Food.java
│   └── conservatory/
│       ├── Aviary.java
│       └── Conservatory.java
├── test/
│   ├── birds/
│   │   └── BirdTest.java
│   └── conservatory/
│       ├── AviaryTest.java
│       └── ConservatoryTest.java
├── res/
```

```
│   └── design_document.md
└── pom.xml
```

Conclusion

The Bird Conservatory Management System provides a robust, object-oriented solution for managing the complexities of a bird sanctuary. By leveraging encapsulation, inheritance, and clear separation of concerns (Aviary vs. Conservatory), the design ensures that business rules—like not mixing prey with predators—are enforced at the structural level rather than relying on manual oversight.

The system is extensible; adding new bird types or food requirements involves creating new classes or enum values rather than rewriting core logic. The comprehensive test suite guarantees reliability, making this a safe and scalable tool for conservation efforts.