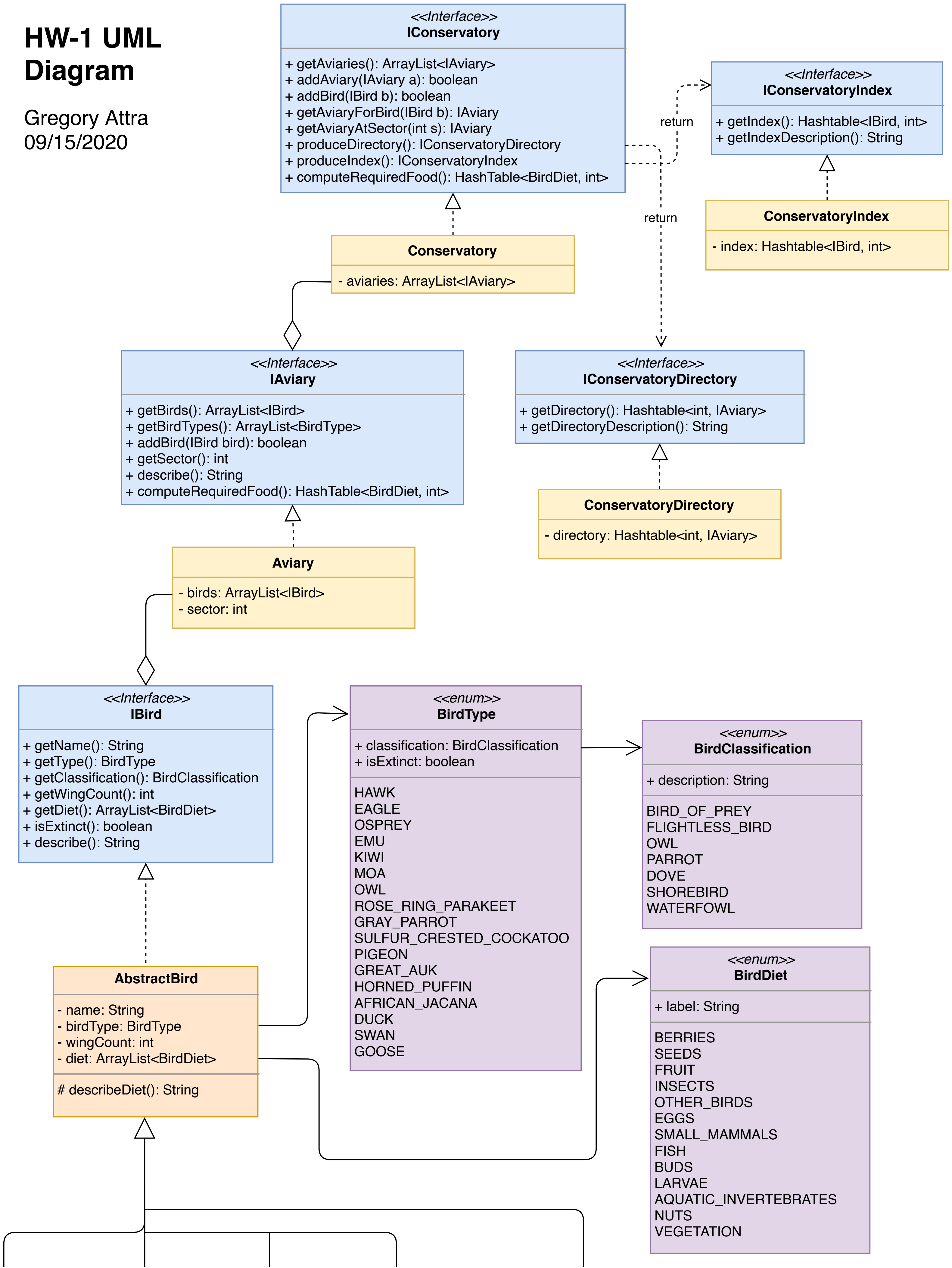
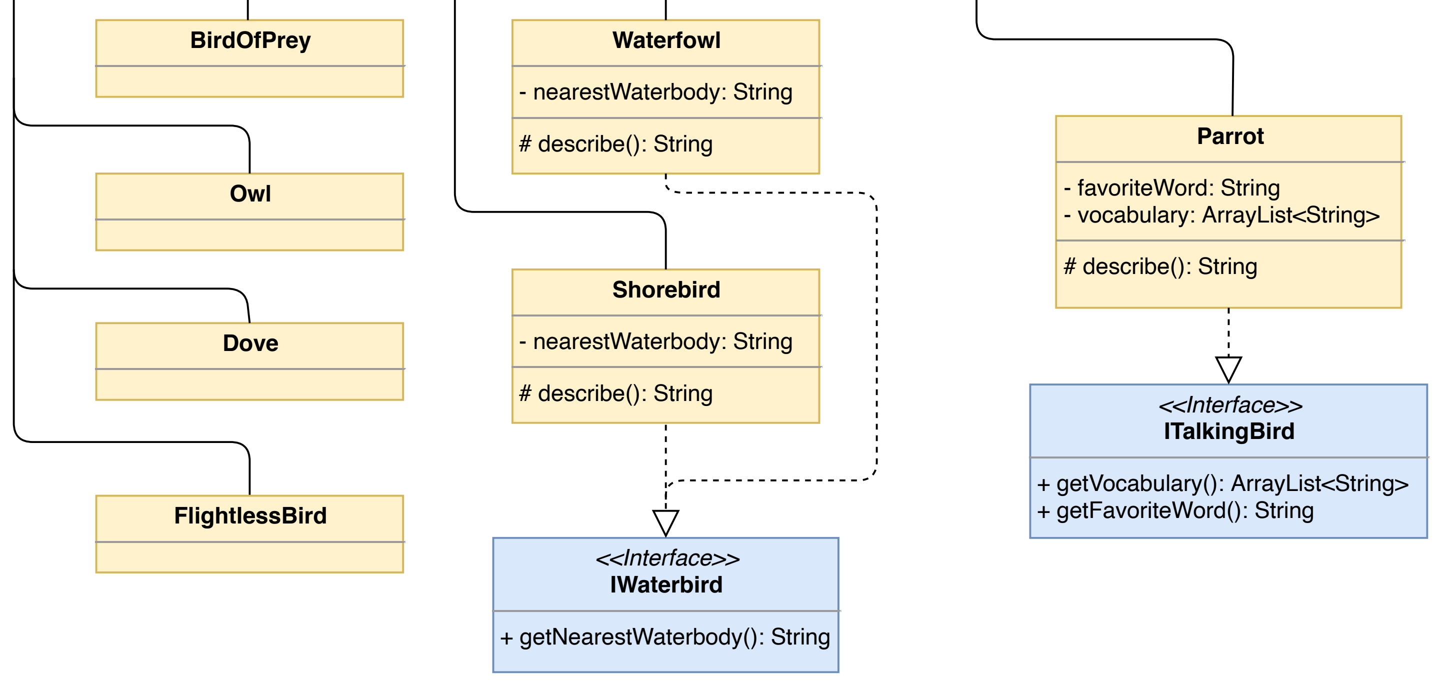
HW-1 UML Diagram

Gregory Attra
09/15/2020




BirdOfPrey
Owl
Dove
FlightlessBird
Waterfowl
- nearestWaterbody: String
# describe(): String
Shorebird
- nearestWaterbody: String
# describe(): String
<<Interface>>
IWaterbird
+ getNearestWaterbody(): String
Parrot
- favoriteWord: String
- vocabulary: ArrayList<String>
# describe(): String
<<Interface>>
ITalkingBird
+ getVocabulary(): ArrayList<String>
+ getFavoriteWord(): String

Greg Attra
09/15/2020

HW 1 - BIRDS

Testing Strategy
Each class will have its own test suite. The suite will be responsible for making sure constructors, getters and methods produce expected outputs. Tests will also be required to ensure that constraints are being properly enforced.

Conservatory
The Conservatory object is responsible for orchestrating the rescue of other birds, establishment of aviaries and communicating information about those aviaries, their location and the birds they house.

a. *constructor()*
    i. Assert that no conservatory is instantiated with more than 20 aviaries
    ii. Assert that each Aviary has a unique sector id ranging between 1 - 20
b. *getAviaries()* - returns ArrayList of aviary instances
    i. Assert aviary count matches expected
    ii. Assert each aviary attribute matches corresponding aviary attributes from list used in constructor
c. *addAviary(Aviary a)* - adds an aviary instance to the ArrayList attribute
    i. Assert happy-path results in successful addition of aviary instance to aviaries attribute
    ii. Assert aviary is not added when the conservatory already has 20 aviaries
    iii. Assert aviary is not added which does not have sector id specified
    iv. Assert aviary is not being added to sector already containing an aviary
d. *addBird(Bird b)* - adds a bird to an aviary in the conservatories ArrayList of aviaries
    i. Assert this results in successful addition of bird instance to a proper aviary when one exists
    ii. Assert that a new aviary is created if a proper aviary does not exist
    iii. Assert that no aviary is created and that the bird was not added if the conservatory already has 20 aviaries and none of them can house the bird
    iv. Assert that the bird is not added if it has a name matching the name of a bird already housed in the conservatory
e. *getAviaryForBird(Bird b)* - given a bird instance, find the aviary that houses it
    i. Assert that the correct aviary is returned in the case that conservatory instance has an aviary with that bird
    ii. Assert that no aviary is returned when conservatory is not housing that bird in one of its aviaries
f. *getAviaryByLocation(int sectorId)* - returns the aviary located at this sector
    i. Assert that the correct aviary is returned given a sector id for which the conservatory has an aviary

        ii. Assert that no aviary is returned given a sector id for which the conservatory does not have an aviary
        iii. Assert throws when given invalid sector id (< 0 or > 20)
    g. *produceDirectory()* - generates, populates and returns ConservatoryDirectory instance
        i. Setup a few aviaries with some birds and instantiate a conservatory using them
        ii. Validate the ConservatoryDirectory is properly set up:
            1. Assert that the the `directory` attribute keys (which represents the aviary's sector id) match the corresponding Aviary's sector attribute
            2. Assert that the `directory` attribute has the same number of aviaries as the conservatory
    h. *produceIndex()* - generates, populates and returns ConservatoryIndex instance
        i. Set up a few aviaries with some birds and instantiate a conservatory using them
        ii. Validate the ConservatoryIndex is properly set up:
            1. Assert that the `index` attribute has the correct length, which should match the total number of birds housed in the conservatory
            2. Assert that for each value in the `index` list the value (sector id) matches the sector id of the aviary housing the corresponding bird (key value). A useful function here is the conservatory's "findAviaryForBird(Bird b)" method
    i. *computeRequiredFood()* - generates a hashtable counting the required quantity for each BirdDiet given the birds housed in the conservatory
        i. Assumptions:
            1. A bird only needs 1 of its BirdDiet per day
            2. This method is computing the required quantity for a single day
        ii. Instantiate a conservatory with dummy data and hard-code a HashTable with the expected food requirements
        iii. Assert that the method returns a matching HashTable

ConservatoryIndex

The conservatory index is mostly packaged functionality. It is populated with a HashTable<IBird, int> where the `int` value represents the sector where the aviary housing the corresponding bird (the key) is located in the conservatory. There are two main functions:

- getIndex() returns the raw `index` attribute (HashTable)
- getIndexDescription() converts the HashTable into a human-readable block of well-formatted text

The constructor for this class takes simply a HashTable<IBird, int> so there is nothing the handle or test for that the compiler wouldn't already catch.

a. *getIndex()* - returns the raw `index` attribute
    i. Instantiate a ConservatoryIndex object with some dummy data
    ii. Assert this method returns the `index` HashTable. Compare the length of the table and check each index of the HashTable, comparing it to the corresponding index of the object passed into the constructor.
b. *getIndexDescription()* - returns a formatted string representation of the index

        i. Instantiate a ConservatoryIndex object with some dummy data. Hard code the expected string
        ii. Assert that the method output matches the expected string

ConservatoryDirectory

The conservatory directory is similar to the index in that it is mostly packaged functionality. It is populated with a HashTable<int, IAviary> where the `int` value represents the sector where the aviary (the key) is located in the conservatory. There are two main functions:

- getDirectory() returns the raw `directory` attribute (HashTable)
- getDirectoryDescription() converts the HashTable into a human-readable block of well-formatted text

The constructor for this class takes simply a Hastable<int, IAviary> so there is nothing the handle or test for that the compiler wouldn't already catch.

c. *getDirectory()* - returns the raw `directory` attribute
    i. Instantiate a ConservatoryDirectory object with some dummy data
    ii. Assert this method returns the `directory` HashTable. Compare the length of the table and check each index of the HashTable, comparing it to the corresponding index of the object passed into the constructor.
d. *getIndexDescription()* - returns a formatted string representation of the index
    i. Instantiate a ConservatoryIndex object with some dummy data. Hard code the expected string
    ii. Assert that the method output matches the expected string

Aviary

The Aviary class is responsible for housing rescued birds. A Conservatory has an aggregation of Aviaries. An Aviary can only house up to 5 birds and cannot mix certain types of birds: birds of prey, flightless birds, and waterfowl must remain isolated. Extinct birds cannot be added to the aviary.

a. *constructor()*
    i. Assert that the `birds` ArrayList passed to the constructor does not:
        1. Contain extinct birds
        2. Contain birds of types that cannot be mixed
        3. Does not exceed a length of 5
    ii. Assert that a non-negative int ranging between 1 - 20 was used for the `sector` attribute. 0 or null values are not permitted.
b. *getBirds()* - returns the raw ArrayList<IBird> of each bird housed in the Aviary
    i. Instantiate Aviary with dummy data
    ii. Validate that the returned bird list has the same length and that each bird in the list matches the corresponding bird in the list used at instantiation
c. *getBirdTypes()* - returns an ArrayList<BirdType> of the types of birds housed in the Aviary
d. *addBird(IBird bird)* - adds a bird to the aviary if possible
    i. Assert that the bird was successfully inserted in the case that the aviary has space and the bird type is permitted
        1. Do this for empty aviary
        2. Do this for aviary with bird types that can be mixed together

            3. Do this for aviary with bird type that can't be mixed, but the bird being added matches that bird type
        ii. Assert that the bird was not added when the aviary is full
        iii. Assert that the bird was not added when the aviary doesn't permit the type of bird being added - i.e. Aviary has Birds of Prey and the bird being added is a Waterfowl
        iv. Assert that, when a bird of a new type not yet housed in the aviary is successfully added, the `birdTypes` array list is updated to account for that new type

- *e.* *getSector()* - returns the int ID of the sector where the aviary is located within the conservatory
    - i. Instantiate an Aviary with a specific Sector ID
    - ii. Assert that this method returns that sector id
- *f.* *describe()* - returns a string describing the types of birds housed in the aviary and the count of each type.
    - i. Instantiate a few Aviaries with dummy data
        1. One aviary houses waterfowl
        2. Another houses talking birds like Parrots
        3. Another houses owls and doves
    - ii. Hard code the expected string response for each aviary
        1. Aviary one description should contain a sentence about each bird's nearest waterbody
        2. Aviary two description should contain the number of words each bird knows and their favorite words
        3. Aviary three should not contain any information about waterbodies or vocabulary or favorite words, as these birds aren't housed in that aviary
- *g.* *computeRequiredFood()* - generates a hashtable counting the required quantity for each BirdDiet given the birds housed in the aviary
    - i. Assumptions:
        1. A bird only needs 1 of its BirdDiet per day
        2. This method is computing the required quantity for a single day
    - ii. Instantiate an aviary with dummy data and hard-code a HashTable with the expected food requirements
    - iii. Assert that the method returns a matching HashTable

AbstractBird

The AbstractBird class represents the individual birds housed in the conservatory. How their properties may be set is determined by their type and the constructor used.

Below is the testing strategy for classes that extend AbstractBird:

(Strategy for all extending classes) - BirdOfPrey, Dove, Owl, Waterfowl, Shorebird, Dove, Parrot, FlightlessBird

- *a.* *constructor()*
    - i. Assert that a name is specified
    - ii. Assert that wing count is not less than 0, and not greater than 2

        iii. Assert that diet length is >= 2 and <= 4
            1. These constraints are handled by the constructors which throw exceptions upon their violation. The tests should assert these exceptions.
    b. *getName(), getType(), getClassification(), getWingCount(), getDiet(), isExtinct()*
        i. Tests for these getter methods will assert that the values used in the constructor match what is returned by their respective getter method
    c. *describe()* - returns a string describing the classification of the bird and unique features about the specific bird instance
        i. Instantiate a bird with dummy data and assert that:
            1. The description contains that bird's classification description (found in BirdClassification enum)
            2. The description contains that bird's diet description
    d. *describeDiet()* - a private helper method that converts the `diet` array list attribute into a human-readable string
        i. Instantiate a bird with dummy data and hard code the expected string
        ii. Assert that the method returns that expected string

Waterfowl, Shorebird

    a. *constructor()*
        i. Assert that waterfowl and shorebirds are instantiated with a `nearestWaterbody` string
            1. These constraints are handled by the constructors which throw exceptions upon their violation. The tests should assert these exceptions.
    b. *describe()* - returns a string describing the classification of the bird and unique features about the specific bird instance
        i. Assert that for waterfowl and shorebirds the description contains the bird instance's `nearestWaterbody`
    c. *getNearestWaterbody()* - returns the `nearestWaterbody` attribute value
        i. Assert that value returned matches value passed into constructor

Parrot

    a. *constructor()*
        i. Assert that talking birds are instantiated with a vocabulary list and a favorite word. Assert that the vocab list is greater than 0 and <= 100
            1. These constraints are handled by the constructors which throw exceptions upon their violation. The tests should assert these exceptions.
    b. *describe()* - returns a string describing the classification of the bird and unique features about the specific bird instance
        i. For talking birds, i.e. Parrot, assert that the description contains the bird's favorite word and vocabulary count
        ii. Assert that for waterfowl and shorebirds the description contains the bird instance's `nearestWaterbody`
    c. *describeVocabulary()* - a private helper method that converts the `vocabulary` array list attribute into a human-readable string
        i. Instantiate a talking bird with dummy data and hard-code the expected string
        ii. Assert that the method returns the expected string

        iii. Assert that a non-talking bird instance returns “This bird does not talk and has no vocabulary.”
    d. *getVocabulary()* - returns ArrayList<String> of known words
        i. Assert that length of returned list matches length of list passed to constructor
        ii. Assert that each word in returned list matches corresponding word in list passed to constructor
    e. *getFavoriteWord()* - returns the bird’s favorite word (String)
        i. Assert that the returned String matches the String passed to constructor

FlightlessBird

a. *constructor()*
    i. Assert that `wingCount` == 0 for flightless birds