Module: Core Java

Session 9: Inheritance and Polymorphism practice

* This is practice a session; you will work on Inheritance and Polymorphism assignments.
* You can discuss your doubts with the trainer

**Assignments:**

**Assignment 1 - Inheritance Concepts Assignment**

Develop classes to generate paychecks for the all employees in the company. paychecks are generated once every month.

There are two types of employees Permanent and Temporary

Every employee has atttributes-name, hireDate, wage and a method generatePayCheck to calculate the salary for the pay period and the vacation balance aacured(if applicable) Permanent Employee has attribute - vacationBalance(in Hours) and a method recordVacation. This method will be used to update the balance when employee takes vacation. Every pay period, employee accures 5 hours vacation. Salary for a permanent employee is annual salary divided by number of months.

Temporary Employee has attribut Number of hours worked. Salary for temporary employee is wage(hourly rate) multiplied by the number of hours worked. There is no vacation for temporary employees.

Employe working in Sales department are Permanent Employees. The salary for Sales Employee also include a sales commission component, which is certain percentage of sales made by him/her during that pay period. You can implement this by adding two attributes sales and commission(percentage).

Please take appropriate access specifier into consideration when designing the classes. Use data types that you think are appropriate.

Polymorphism/Dynamic Binding

Develop a test driver program( with main method) to verify the code written for above employee classes. Assume that there are two Permanent Employees, Two Temporary Employees and two Sales Employees working in the company. Create these objects with your own choice of values and then invoke the method generatePayCheck for all the six employees(this method is implemented using polymorphism/dynamic binding).

Add some print statements (to standard output) in your code to illustrate that your program calculates earnings for all the employees correctly.

**Assignment 2 – Inheritance Assignment**

## Objectives

* Be able to derive a class from an existing class
* Be able to define a class hierarchy in which methods are overridden and fields are hidden
* Be able to use derived-class objects
* Implement a copy constructor

## Introduction

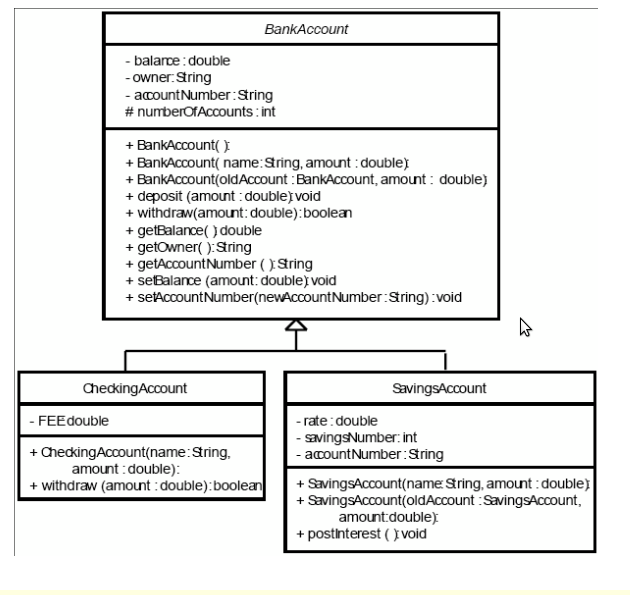
In this lab, you will be creating new classes that are derived from a class called BankAccount. A checking account **is a** bank account and a savings account is a bank account as well. This sets up a relationship called inheritance, where BankAccount is the superclass and CheckingAccount and SavingsAccount are subclasses.

This relationship allows CheckingAccount to inherit attributes from BankAccount (like owner, balance, and accountNumber, but it can have new attributes that are specific to a checking account, like a fee for clearing a check. It also allows CheckingAccount to inherit methods from BankAccount, like deposit, that are universal for all bank accounts.

You will write a withdraw method in CheckingAccount that overrides the withdraw method in BankAccount, in order to do something slightly different than the original withdraw method.

You will use an instance variable called accountNumber in SavingsAccount to hide the accountNumber variable inherited from BankAccount.

The UML diagram for the inheritance relationship is as follows:



## Task 1: Extending BankAccount

* Create AccountDriver.java and BankAccount.java in a directory or create a new project in Eclipse IDE and create these two classes.
* Source code for these two classes as follow:
* BankAccount.java is complete and will not need to be modified.
* Create a new class called CheckingAccount that extends BankAccount.
* It should contain a static constant FEE that represents the cost of clearing one check. Set it equal to 15 cents.
* Write a constructor that takes a name and an initial amount as parameters. It should **call the constructor for the superclass**. It should initialise accountNumber to be the current value in accountNumber concatenated with “-10”. All checking accounts at this bank are identified by the extension “-10”. There can be only one checking account for each account number. Remember since accountNumber is a private member in BankAccount, it must be changed through a mutator method.
* Write a new instance method, withdraw, that **overrides** the withdraw method in the superclass. This method should take the amount to withdraw, add to it the fee for check clearing, and call the withdraw method from the superclass. Remember that to override the method, it must have the same method heading. Notice that the withdraw method from the superclass returns true or false depending if it was able to complete the withdrawal or not. The method that overrides it must also return the same true or false that was returned from the call to the withdraw method from the superclass.
* Compile and debug this class.

## Task 2: Creating a Second Subclass

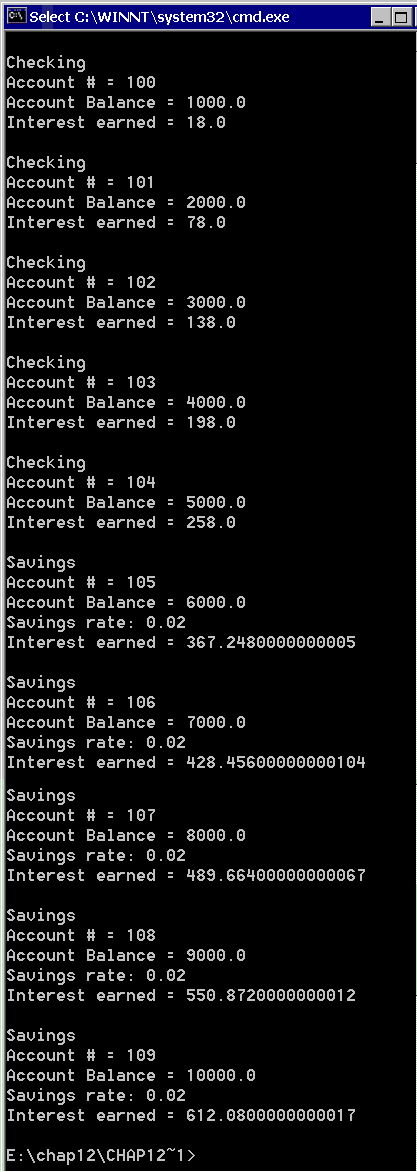
1. Create a new class called SavingsAccount that extends BankAccount.
2. It should contain an instance variable called rate that represents the annual interest rate. Set it equal to 2.5%.
3. It should also have an instance variable called savingsNumber, initialised to 0. In this bank, you have one account number, but can have several savings accounts with that same number. Each individual savings account is identified by the number following a dash. For example, 100001-0 is the first savings account you open, 100001-1 would be another savings account that is still part of your same account. This is so that you can keep some funds separate from the others, like a Christmas club account.
4. An instance variable called accountNumber that will hide the accountNumber from the superclass, should also be in this class.
5. Write a constructor that takes a name and an initial balance as parameters and calls the constructor for the superclass. It should initialise accountNumber to be the current value in the superclass accountNumber (the hidden instance variable) concatenated with a hyphen and then the savingsNumber.
6. Write a method called postInterest that has no parameters and returns no value. This method will calculate one month’s worth of interest on the balance and deposit it into the account.
7. Write a method that overrides the getAccountNumber method in the superclass.
8. Write a copy constructor that creates another savings account for the same person. It should take the original savings account and an initial balance as parameters. It should call the copy constructor of the superclass, assign the savingsNumber to be one more than the savingsNumber of the original savings account. It should assign the accountNumber to be the accountNumber of the superclass concatenated with the hyphen and the savingsNumber of the new account.
9. Compile and debug this class.
10. Use the AccountDriver class to test out your classes. If you named and created your classes and methods correctly, it should not have any difficulties. If you have errors, do not edit the AccountDriver class. You must make your classes work with this program.
11. Running the program should give the following output:

Account Number 100001-10 belonging to Benjamin Franklin Initial balance = $1000.00 After deposit of $500.00, balance = $1500.00 After withdrawal of $1000.00, balance = $499.85 Account Number 100002-0 belonging to William Shakespeare Initial balance = $400.00 After deposit of $500.00, balance = $900.00 Insufficient funds to withdraw $1000.00, balance = $900.00 After monthly interest has been posted, balance = $901.88 Account Number 100002-1 belonging to William Shakespeare Initial balance = $5.00 After deposit of $500.00, balance = $505.00 Insufficient funds to withdraw $1000.00, balance = $505.00 Account Number 100003-10 belonging to Isaac Newton

**Assignment 3 -Advance Inheritance Concepts Assignment**

The purpose of this exercise is to use inheritance, an abstract class, abstract methods, and override the toString() method. All the requirements are listed below. Save the classes you create as Account.java, Checking.java, Savings.java, and AccountArray.java.

1. For the Account class, create:
   1. two protected variables for the account number and the account balance.
   2. two get methods - one for the account number and one for the account balance
   3. two set methods - one for the account number and one for the balance.
   4. a constructor that requires an account number and sets the balance to 0.
   5. a method to override the toString() method which returns a String containing: Account # = < account number> Account Balance = <account balance> where <account number> is the object's account number and <account balance> is the object's account balance.
   6. an abstract compute Interest() method that takes one integer argument and returns a double.
   7. Do not allow either the account number or the balance to be set to a negative number.
2. The Checking class will be a subclass of Account. For the Checking class, create:
   1. a method to override the to String() method which returns a String containing: "Checking" , the account number, and the account balance. Do not print the interest earned as part of to String().
   2. a constructor that takes an account number as an argument and calls the super class constructor passing this account number.
   3. a method to implement the abstract compute Interest() method of the super class. This method takes one argument for the interest period and returns the interest earned. The interest earned is 2 % of the balance over $700 times the interest period. The interest period is passed as an argument to the compute Interest() method. For example, the interest on $3000 at 2% for three years is (3000 - 700) times .02 times 3 or 138.
3. The Savings class will be a subclass of Account. For the Savings class, create:
   1. an additional private variable to hold the interest rate.
   2. a get and set method for the interest rate. Do not allow the interest rate to be set to a negative number.
   3. a method named to String() that overrides the to String() method of the Object class. This method should return "Savings", the account number, the account balance and the interest rate. Do not print the interest earned as part of to String().
   4. a constructor with two arguments - the account number and the interest rate. This constructor calls the super class constructor passing the account number.
   5. a method to implement the abstract computeInterest() method of the superclass. This method takes one argument for the interest period and returns the interest earned. The interest is calculated as (1 + interest rate)period times the balance minus the balance. The interest period is passed as an argument to the computeInterest() method. For example, the interest on $6000 at 2% for three years is (1.02)3 \* 6000 - 6000 or 367.
4. For the Account Array class, create:
   1. a main() method.
   2. an array to store 10 objects of either the Savings or Checking class. Use 2% (.02) for the savings account rate. Use account numbers 100 through 109. Use initial balances of 1000 through 10000.
   3. a for loop that will instantiate 5 objects of the Savings class and 5 objects of the Checking class. Store these objects in a single array created above.
   4. a for loop that prints the data in all 10 objects of the array using the overridden toString() method and also prints the interest amount computed for each object. Use 3 for the period. The results are displayed in the following figure



**Assignment 4 – Polymorphism Assignment**

## Objectives

* Be able to derive a class from an existing class
* Be able to define a class hierarchy in which methods are overridden and fields are hidden
* Be able to use derived-class objects
* Be able to use polymorphism

## Introduction

For this assignment, you will be writing software in support of a Dessert Shoppe which sells candy by the pound, cookies by the dozen, ice cream, and sundaes (ice cream with a topping). Your software will be used for the checkout system.

To do this, you will implement an inheritance hierarchy of classes derived from a **Dessert Item** *abstract* superclass.

The **Candy**, **Cookie**, and **Ice-cream** classes will be derived from the **Dessert Item** class.

The **Sundae** class will be derived from the **Ice-cream** class.

**You will also write a Checkout class which maintains a list (Vector) of Dessert Item’s.**

**Task 1: Create the Dessert Item Class**

The **DessertItem** class is an *abstract superclass* from which specific types of **DessertItems** can be derived. It contains only one data member, a name. It also defines a number of methods. All of the **DessertItem** class methods except the **getCost()** method are defined in a generic way in the file, **DessertItem.java.**

**Create a class named “DessertItem.java” in your project. Code is here:**

**// DessertItem.java - Dessert Item abstract superclass**

/\*\*

\* Abstract superclass for Dessert Item hierarchy

\* @author Ritu Ashar

\*/

public abstract class DessertItem {

protected String name;

/\*\*

\* Null constructor for DessertItem class

\*/

public DessertItem() {

this("");

}

/\*\*

\* Initializes DessertItem data

\*/

public DessertItem(String name) {

if (name.length() <= DessertShoppe.MAX\_ITEM\_NAME\_SIZE)

this.name = name;

else

this.name = name.substring(0,DessertShoppe.MAX\_ITEM\_NAME\_SIZE);

}

/\*\*

\* Returns name of DessertItem

\* @return name of DessertItem

\*/

public final String getName() {

return name;

}

/\*\*

\* Returns cost of DessertItem

\* @return cost of DessertItem

\*/

public abstract int getCost();

}

The getCost() method is an *abstract method* that is not defined in the DessertItem class because the method of determining the costs varies based on the type of item. Tax amounts should be rounded to the nearest cent. For example, the calculating the tax on a food item with a cost of 199 cents with a tax rate of 2.0% should be 4 cents.

**Task 2: Create the DessertShoppe Class**

It contains constants such as the tax rate as well the name of the store, the maximum size of an item name and the width used to display the costs of the items on the receipt. Your code should use these constants wherever necessary! The DessertShoppe class also contains the **cents2dollarsAndCents** method which takes an integer number of cents and returns it as a String formatted in dollars and cents. For example, 105 cents would be returned as "1.05".

**Create a class named “DessertShoppe.java” in your project. Code is here:**

// DessertShoppe.java - constants and method to format cents as dollars and cents

public class DessertShoppe {

public final static double TAX\_RATE = 6.5; // 6.5%

public final static String STORE\_NAME = "Ritu's Dessert Shoppe";

public final static int MAX\_ITEM\_NAME\_SIZE = 25;

public final static int COST\_WIDTH = 6;

public static String cents2dollarsAndCents(int cents) {

String s = "";

if (cents < 0) {

s += "-";

cents \*= -1;

}

int dollars = cents/100;

cents = cents % 100;

if (dollars > 0)

s += dollars;

s +=".";

if (cents < 10)

s += "0";

s += cents;

return s;

}

}

**Task 3: Create the derived classes (Candy, Cookie, IceCream, Sundae)**

All of the classes which are derived from the **DessertItem** class must define a constructor. Please see the **TestCheckout.java**, to determine the parameters for the various constructors. Each derived class should be implemented by creating a file with the correct name, eg., **Candy.java**.

The **Candy** class should be derived from the **DessertItem** class. A **Candy** item has a *weight* and a *price per pound* which are used to determine its *cost*. For example, 2.30 lbs.of fudge @ .89 /lb. = 205 cents. The cost should be rounded to the nearest cent.

The **Cookie** class should be derived from the **DessertItem** class. A **Cookie** item has a *number* and a *price per dozen* which are used to determine its *cost*. For example, 4 cookies @ 399 cents /dz. = 133 cents. The cost should be rounded to the nearest cent.

The **IceCream** class should be derived from the **DessertItem** class. An **IceCream** item simply has a *cost*.

The **Sundae** class should be derived from the **IceCream** class. The *cost* of a Sundae is the *cost of the IceCream* plus the *cost of the topping*.

**Task 4: Create the Checkout Class**

The **Checkout** class, provides methods to enter dessert items into the cash register, clear the cash register, get the number of items, get the total cost of the items (before tax), get the total tax for the items, and get a String representing a receipt for the dessert items. The **Checkout** class must use a Vector to store the DessertItem's. The total tax should be rounded to the nearest cent. The complete specifications for the **Checkout** class are as follow:

**Class Name:** Checkout

**Class Description:** Maintains a list of DessertItem references There is no limit to the number of DessertItem's in the list

**Constructor:** Default constructor which creates a Checkout instance with an empty list of DessertItem's

**Methods:**

**1.** public int **numberOfItems**()

Returns the number of DessertItem's in the list

2. public void **enterItem**(DessertItem item)

A DessertItem is added to the end of the list of items

3. public void **enterItem**(DessertItem item)

A DessertItem is added to the end of the list of items

4.public int **totalCost**()

Returns total cost of items in cents (without tax)

5. public int **totalTax**()

Returns total tax on items in cents

6. public java.lang.String **toString**()

**Overrides:**

toString in class java.lang.Object

Returns a String representing a receipt the current list of DessertItem's with the name of the Dessert store, the items purchased, the tax, and the total cost, for example.

Ritu's Dessert Shoppe

--------------------

Strawberry Ice Cream 1.45

Caramel Sundae with

Vanilla Ice Cream 1.55

1.33 lbs. @ .89 /lb.

Gummy Worms 1.18

4 @ 3.99 /dz.

Chocolate Chip Cookies 1.33

1.50 lbs. @ 2.09 /lb.

Salt Water Taffy 3.14

3.00 lbs. @ 1.09 /lb.

Candy Corn 3.27

Tax .77

Total Cost 12.69

**Task 5: Test your application with TestCheckout class**

A simple testdriver class to test your application is here. Save following code as “TestCheckout.java”

// TestCheckout.java - main method to test Checkout class

// and GroceryItem hierarchy

public class TestCheckout {

public static void main(String[] args) {

Checkout checkout = new Checkout();

checkout.enterItem(new Candy("Peanut Butter Fudge", 2.25, 399));

checkout.enterItem(new IceCream("Vanilla Ice Cream",105));

checkout.enterItem(new Sundae("Choc. Chip Ice Cream",145, "Hot Fudge", 50));

checkout.enterItem(new Cookie("Oatmeal Raisin Cookies", 4, 399));

System.out.println("\nNumber of items: " + checkout.numberOfItems() + "\n");

System.out.println("\nTotal cost: " + checkout.totalCost() + "\n");

System.out.println("\nTotal tax: " + checkout.totalTax() + "\n");

System.out.println("\nCost + Tax: " + (checkout.totalCost() + checkout.totalTax()) + "\n");

System.out.println(checkout);

checkout.clear();

checkout.enterItem(new IceCream("Strawberry Ice Cream",145));

checkout.enterItem(new Sundae("Vanilla Ice Cream",105, "Caramel", 50));

checkout.enterItem(new Candy("Gummy Worms", 1.33, 89));

checkout.enterItem(new Cookie("Chocolate Chip Cookies", 4, 399));

checkout.enterItem(new Candy("Salt Water Taffy", 1.5, 209));

checkout.enterItem(new Candy("Candy Corn",3.0, 109));

System.out.println("\nNumber of items: " + checkout.numberOfItems() + "\n");

System.out.println("\nTotal cost: " + checkout.totalCost() + "\n");

System.out.println("\nTotal tax: " + checkout.totalTax() + "\n");

System.out.println("\nCost + Tax: " + (checkout.totalCost() + checkout.totalTax()) + "\n");

System.out.println(checkout);

}

}

**Expected output of this application is as follow:**

Number of items: 4

Total cost: 1331

Total tax: 87

Cost + Tax: 1418

Ritu's Dessert Shoppe

--------------------

2.25 lbs. @ 3.99 /lb.

Peanut Butter Fudge 8.98

Vanilla Ice Cream 1.05

Hot Fudge Sundae with

Choc. Chip Ice Cream 1.95

4 @ 3.99 /dz.

Oatmeal Raisin Cookies 1.33

Tax .87

Total Cost 14.18

Number of items: 6

Total cost: 1192

Total tax: 77

Cost + Tax: 1269

Ritu's Dessert Shoppe

--------------------

Strawberry Ice Cream 1.45

Caramel Sundae with

Vanilla Ice Cream 1.55

1.33 lbs. @ .89 /lb.

Gummy Worms 1.18

4 @ 3.99 /dz.

Chocolate Chip Cookies 1.33

1.50 lbs. @ 2.09 /lb.

Salt Water Taffy 3.14

3.00 lbs. @ 1.09 /lb.

Candy Corn 3.27

Tax .77

Total Cost 12.69

**Assignment 5 – Advanced Polymorphism Assignment**

## Objectives

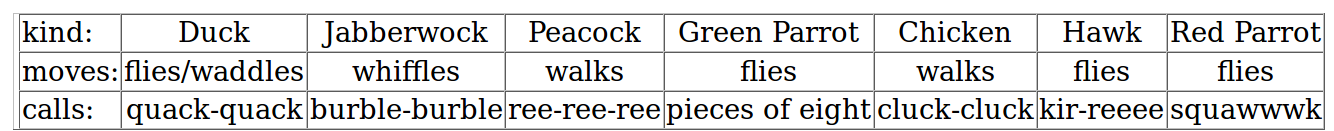
* Be able to derive a class from an abstract class
* Be able to use derived-class objects
* Be able to use polymorphism

## Introduction

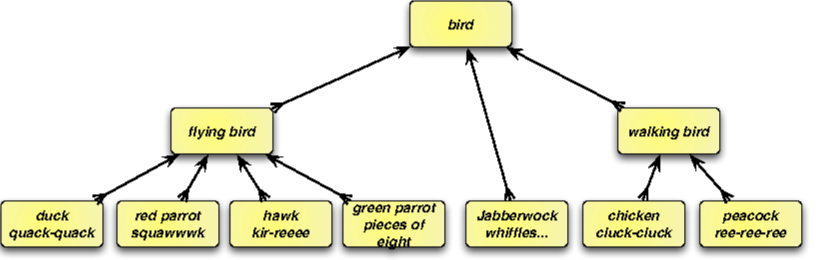
In this assignment you will design a simple simulation to illustrate the use of polymorphism. The scenario is that we are sitting in an aviary or bird sanctuary and watching the different birds walk or fly past us. The aviary has seven different birds:

* a chicken that walks and calls “cluck-cluck”;
* a duck that either flies or waddles and calls “quack-quack”;
* a green parrot that flies and calls “squawwwwk”;
* a hawk that flies and calls “kir-reeee”;
* a jabberwock that whiffles and calls “burble-burble”;
* a peacock that walks and calls “ree-ree-ree”; and
* a red parrot that flies and calls “pieces of eight, pieces of eight”.

How can you build a software representation of such an aviary? The key is (i) to recognize that some of our birds have common attributes; (ii) to create a class for each kind of bird; and (iii) to consolidate the common attributes into parent classes. Given our descriptions, we see that every bird has a kind, each moves somehow, and each has a call. We can start by drawing a table listing each bird and its attributes:



Four of our birds move by flying and two move by walking, so some of our birds move the same way. In such cases, you can consolidate the common moves attribute within a parent class:



Each bird has a call attribute, but they are different (e.g., quack-quack, kir-reee, …). Similarly, each bird has a kind attribute, but these also are different (e.g., duck, hawk, …). In circumstances like this where each object has a common attribute that differs in value, you have a choice: you can either (i) store the common attribute and an accessor method in a parent class so that child classes inherit it or (ii) avoid storing the attribute altogether and instead have each subclass define a polymorphic accessor method for that attribute. To illustrate the two approaches, you will use the first approach to store each bird’s kind attribute in a Bird parent class and use the second approach to define each bird’s call attribute in the subclasses.

The Bird class will thus consolidate the kind characteristic common to all birds and provide a getKind() accessor method for it, illustrating the first approach. Each bottom level class will provide a definition for an abstract getCall() method that returns that bird’s call, illustrating the second approach.

A program that uses this hierarchy will declare an object of class Bird. Such an object will be able reference any object whose type is Bird or a subclass of Bird. As such, you need to be able to send attribute accessor messages getCall(), getMovement(), and getKind() via this object. This means that the declaration of class Bird must provide prototypes for these methods, even though Bird can only define getKind() — the definitions of getMovement() and getCall() will be left to its subclasses. This can be accomplished by declaring getMovement() and getCall() as abstract methods in class Bird.

As you learned in session, the abstract methods make Bird an abstract class, which means that no instances of class Bird can be created. This is as it should be, because in our design, only instances of subclasses like Chicken, Duck, and so on know how to respond to the getCall() message.

## Task 1: Creating Bird Class

1. Create new project in your IDE
2. Create a new class called Bird .
3. It should contain only one attribute myKind that is type String.
4. Write a constructor that takes a kind as a parameter. It should the attribute to the value specified by kind.
5. Write an accessor and define two abstract methods: getMovement() and getCall() returning String
6. Compile and debug this class.

## Task 2: Extending Bird Class

1. Create a new class called FlyingBird extending Bird. It defines the getMovement(). However, because it does not define getCall(), which was declared abstract in class Bird, the class FlyingBird as an abstract class too, and any attempt to create an instance of it will produce an error.
2. The FlyingBird constructor has a parameter for the kind attribute, which it passes along to the Bird constructor. With no attribute variables, there remains nothing else for the constructor to do. Note also that FlyingBird does not store the movement attribute in an instance variable: it simply provides a definition of getMovement() that returns the appropriate string literal. (We use the movement’s past tense for the purposes of our simulation.)
3. The declaration of class WalkingBird is similar to that of FlyingBird.
4. The Hawk constructor simply passes its kind along to its superclass (FlyingBird) constructor (which in turn passes it along to the Bird constructor). Since the Hawk class “knows” its call, it provides the definition for the getCall() method. Because Hawk defines getCall() and inherits a definition of getMovement(), all of the abstract methods have definitions, so the compiler can create instances of class Hawk. Most of the other bottom-level class declarations are similar.
5. Define all the remaining classes
6. The only bottom-level class that differs slightly from the others is the Duck class, because a Duck may either fly or waddle. As a result, the Duck class overrides the definition of getMovement() inherited from class FlyingBird with a new definition that selects randomly the inherited “flew” movement or the “waddled” movement. To elicit this random behavior, we use the Random class from Lecture-12 The remaining bottom-level classes are similar.

## Task 3: Test your application

1. Once we have our class hierarchy implemented, we are ready to write the program to simulate the aviary. The basic idea is to do the following:
   1. Define a vector of seven entries of type Bird.
   2. Using new, initialize each element of the vector to an instance of a different bottom-level class.
   3. Loop:
      1. Generate a random integer within the range of vector indices.
      2. Use that integer to index into the vector.
      3. Display the result of sending that vector element the getKind(), getMovement(), and getCall() messages.
   4. End loop.
2. Your last task is to implement the above algorithm

The sample output shows a run of the program, using the symbol ‘↩’ to represent the ENTER or RETURN character. Note that the duck randomly flies or waddles, as specified by the getMovement() method.

1. **Sample Run:**

Welcome to the Bird Aviary!

To remain in the aviary, keep pressing 'Enter';

enter any other character to leave:

↩

A jabberwock just whiffled by, calling "burble-burble"...

↩

A hawk just flew by, calling "kir-reeeee"...

↩

A red parrot just flew by, calling "squawwwwwwwwk"...

↩

A duck just waddled by, calling "quack-quack"...

↩

A chicken just walked by, calling "cluck-cluck"...