Name: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. For part 1a-1d, consider the following incomplete class declaration for a Date class:

public class Date {

private int day;

private int month;

private int year;

public Date() { … } default

public Date(int mo, int day, int yr) { **part (b)**

**month = mo;**

**this.day = day;**

**year = yr;**

} //constructor

public int month() { return this.month; … } //returns month of Date

public int day() { return this.day;… } //returns day of Date

public int year() { return this.year… } //returns year of Date

//returns String representation of Date as “m/d/y”

public String toString() { **//part (d)**

**String s = “”;**

**S += month() + “/” + day() + “/” + year() + “/”;**

**Return s;**

}

}

1. Write a piece of code that declares and instantiates a Date object named today with today’s date.

Date today = new Date(10,20,2022);

1. Complete the method body of the non-default constructor:

**month = mo;**

**this.day = day;**

**year = yr;**

1. A client program has been written that involves Date objects. It contains a snippet of code whose intention is to print out the month of each Date object in a Date array:

Date[] myEvents = new Date[10];

//...code to instantiate objects into array

for(int i = 0; i < myEvents.length; i++) {

System.out.println(myEvents[i].month);

}

This code will not compile. Explain the error, how it can be fixed, and what feature of object-oriented programming is being featured in this piece of code.

The variable month is private so it cannot be accessed by these means. The accessor function month missing the parentheses. This can be fixed by adding them.

myEvents[i].month()

This code is showcasing data encapsulation.

1. Write the method body for the toString() method of the Date class:

**String s = “”;**

**S += month + “/” + day + “/” + year + “/”;**

**Return s;**

1. Questions 2a-2c involve the Student class that is partially defined below:

public class Student implements Comparable {

//data

private String name;

private int id;

private double average;

//constructors

public Student() { //not shown }

public Student(String n, int i, double a) { //not shown }

//methods

public int compareTo(Student other) { **//parts a and b**}

//accessor methods getName(), getID(), and getAverage() not

//shown

//mutator methods setName(String s), setAverage(double d), setID(int i) not shown.

}

1. Write a method body for the compareTo method of the Student class that compares the average of Student objects. The method returns a positive integer if this object has a greater average than other, a negative integer if this object has a lesser average than other, and zero if they have the same average.

public int compareTo(Object other) {

if ( this.getAverage() > other.getAverage()){  
 return 1;

}

Else if ( this.getAverage() < other.getAverage()){  
 return -1;

}

Else{  
 return 0;

}

1. Write another method body for the compareTo method of the Student class. This one compares the names of Student objects. The method returns a positive integer if this object comes later in a dictionary than other, a negative integer if this object comes before other in a dictionary, and zero if they have the same name.

public int compareTo(Object other) {

if(this.getName().compareTo(other) > other.compareTo(this.getName()){  
 return 1;

else if(this.getName().compareTo(other) < other.compareTo(this.getname()){

return -1;

else{  
 return 0;

1. Overwrite the method equals from the Object class. This method will return true if two Student objects have the same id, false otherwise. (You must also write the method header!)

Public boolean equals(Student other){  
 if( this.getID() == other.getID()){  
 return true;

Else{  
 return False

}

1. For 3a through 3d, consider the following AnnoyingStudent class:

public class AnnoyingStudent extends Student {

private double volumeLevel;

private boolean hasDetention;

public AnnoyingStudent() { //not shown }

**//part a – Write constructor for parameterized constructor**

//accessor methods getVolumeLevel() and getHasDetention(); //mutator method setVolumeLevel(double d) are not shown

/\*\*

\* Returns true if this Student’s volume level is over 50,

\*false otherwise

\*@return this Student’s detention status

public boolean giveDetention() {

**//part b**

}

}

1. Write the definition for a parameterized constructor for the AnnoyingStudent class. All data items in this class need to be initialized.

Public AnnoyingStudent(String n, int i, double a, double v, Boolean h){

Super(n, i, a);

volumeLevel = v;

hasDetention = h;

}

1. Write the method definition for the giveDetention method. giveDetention returns true if this object’s volumeLevel is greater than 50; false otherwise.

Public Boolean giveDetention(){

If(this.getVolumeLevel() > 50){

Return true;

Else{  
 return false;

}

3c) Consider the following part of a client class that contains a method swap, whose intent it is to swap two Student objects:

public static void main(String[] args) {

Student s1 = new Student(“John”, 1, 64.9);

Student s2 = new Student(“Sam”, 2, 99);

System.out.println(s1 + “ “ + s2); //print info

swap(s1, s2);

}

public static void swap(Student first, Student second) {

Student temp = first;

first = second;

second = temp;

}

The method does not work as intended. Explain what this method does and why it does not actually swap the values of the two original Student objects s1 and s2.

This method does not change anything because Java is pass by value. It only changes first, second, and temp, but they get deleted afterwards leaving s1 and s2 pointing to the same object.

3d) Write the method swap that will swap the values of the two Student objects s1 and s2.

Public static void swap(Student s1, Student s2){

Student temp = new Student(s1.getName(),s1.getID(),s1.getAverage());

S1.setName(s2.getName());

S1.setID(s2.getID());

S1.setAverage(s2.getAverage());

S2.setName(temp.getName());

S2setID(temp.getID());

S2.setAverage(temp.getAverage());

}

1. Design an interface called Lockable that includes the following methods: setKey, lock, unlock, locked. The setKey, lock, and unlock methods take an integer parameter that represents the key. The setKey method establishes the key. The lock and unlock methods lock and unlock the object, but only if the key used is correct. The locked method returns a boolean of true for locked and false for unlocked. A Lockable object is an object whose regular methods are protected: if the object is locked, the methods cannot be invoked; if it is unlocked, they can be invoked.
2. Write the interface Lockable below:

Public interface Lockable{

Public void setKey(int key);

Public boolean lock(int key);

Public boolean unlock(int key);

Public Boolean locked(int key);

}

1. Write the definition for a class named Account, which implements Lockable. An account has the following information: an account number, which is an int, a balance, which is a double, a name, which is a String, and a constant double rate of .035, which represents an interest rate. It requires one constructor to initialize all instance variables. It has the following methods:

/\*\*

\* Deposits the specified amount into the account. Returns the new

\* balance.

\* @param amount the amount to be deposited (assume it is positive)

\* @return the new balance

\*/

public double deposit(double amount)

/\*\*

\* Withdraws the specified amount from the account. Returns the new

\* balance. If the new balance is negative, it must be set to zero, and a

\* message must be printed about the new balance.

\* @param amount the amount to be withdrawn

\* @return the new balance

\*/

public double withdraw(double amount)

/\*\*

\* Adds interest to the account and returns the new balance

\* @return the new balance with added interest

\*/

public double addInterest()

You also must define all methods from the Lockable interface. No specific Account methods can be used if the object is locked. If a calling method attempts to use any method of the Account class when it is locked, an error message must be printed to the screen, and a -1 must be returned. Do not worry about how a client class would handle the -1. Do not worry about validating input amounts (depositing or withdrawing negative amounts…).

Hint: You will need to declare an integer to represent the key to lock the object.

Complete the Account class below.

Class Account implements Lockable{  
 private int accountNumber;

Private double balance;

Private String name;

Private const double rate = 0.35;

Private int key;

Public Account(int a, double b, String n){  
 accountName = a;

Balance = b;

Name = n;

}

Public double deposit( double amount){  
 this.balance += amount;

Return this.balance;

}

Public double withdraw(double amount){

This.balance -= amount;

If(this.balance < 0){  
 System.out.println(“Balance went into negatives, setting to 0);

This.balance = 0;

}

Return this.balance;

}

Public double addInterest(){

Double interest = this.balance \* this.rate;

Return this.balance + this.interest;

}

Public void setKey(int key){  
 this.key = key;

}

Public void lock(int key){  
 if ( object == Unlocked){

Return True;

Else{  
 return false;

}

}

Public Boolean unlock(int key){

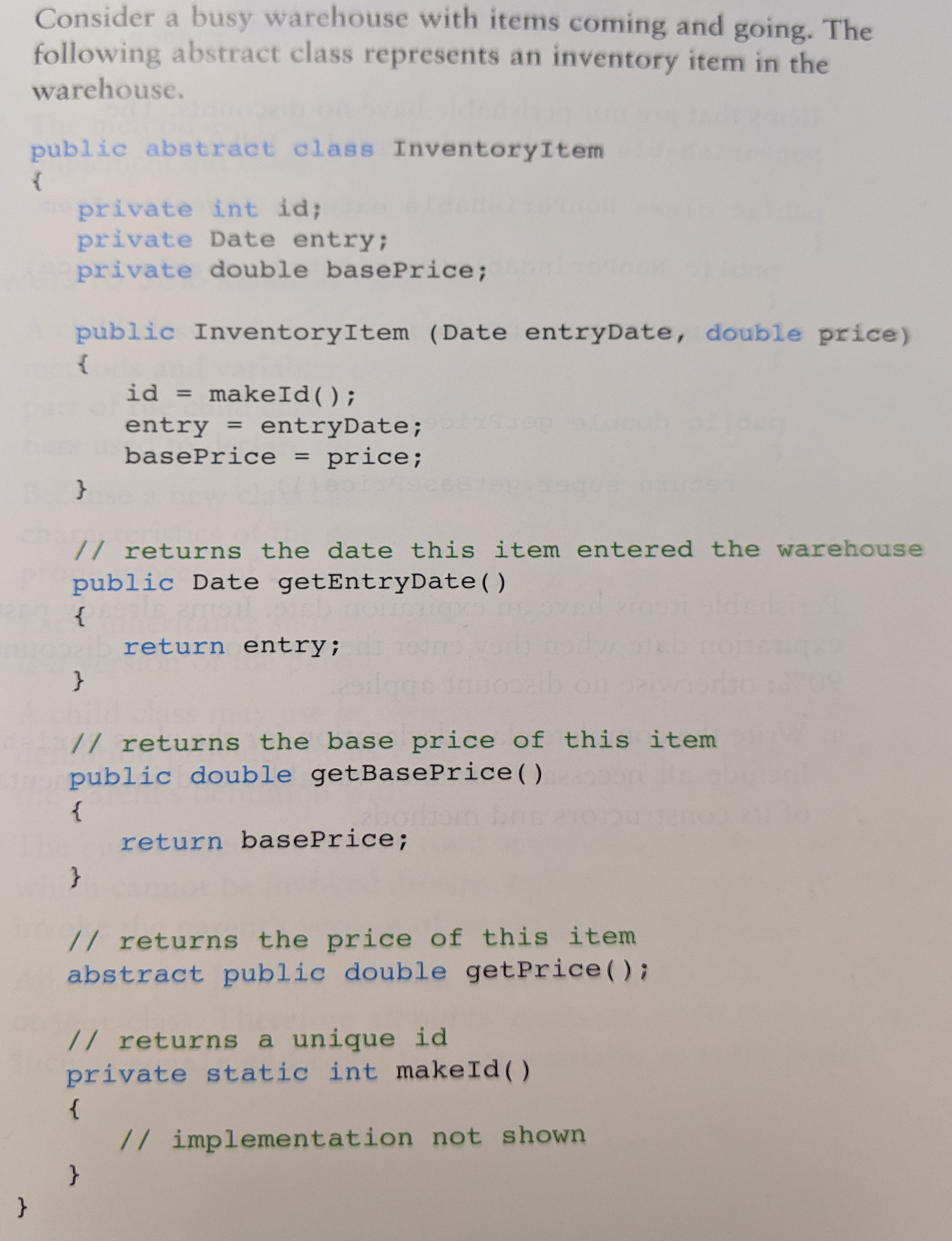
If(this.key != key){  
 return False

}

Else{  
 return True

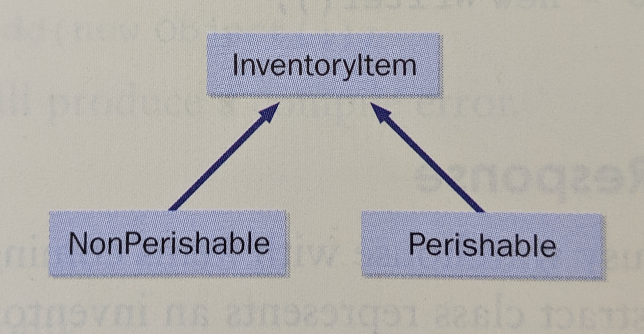
}

1. Consider a busy warehouse with items coming and going. The following abstract class represents an inventory item in the warehouse.

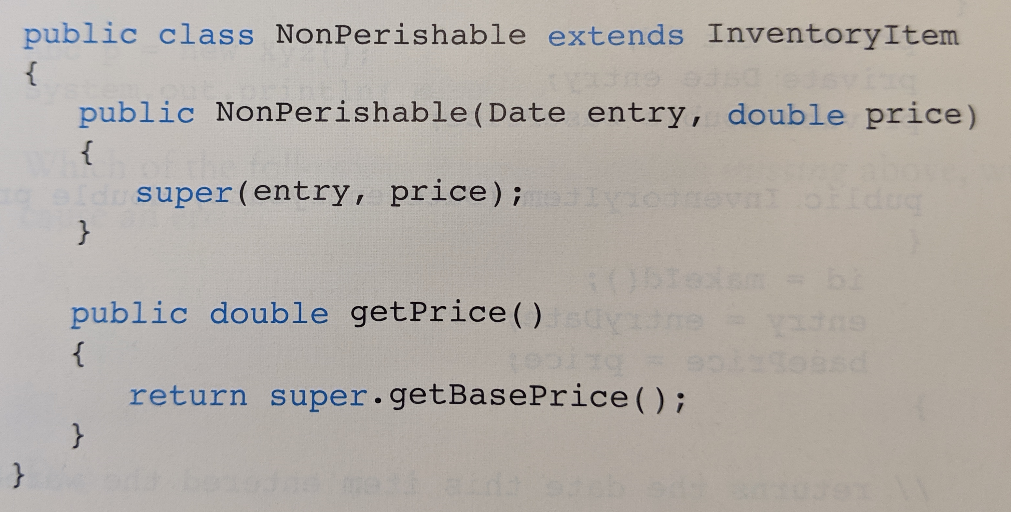
­

Each InventoryItem has a unique ID, a date of entry into the warehouse, and a base price (which may possibly be discounted). Dates are represented with a Date class (not shown), which implements the Comparable interface (larger Date objects are later Dates).

Some inventory items are perishable and some are not, giving the following class hierarchy:



Items that are not perishable have no discounts. The NonPerishable class is implemented as follows:



Perishable items have an expiration date. Items already past their expiration date when they enter the warehouse are discounted by 90%; otherwise no discount applies.

1. Write the complete class declaration for the class Perishable. Include all necessary instance variables and implementations of its constructors and methods.

Public class Perishable extends InventoryItem{

Private Date experationDate;

Private Date currentDate;

Public Perishable(Date e, double p, Date cd, Date ed){

Super(e,p);

CurrentDate = cd;

experationDate = ed;

}

Public double getPrice(){

If (this.experationDate.compareTo(this.currentDate) > this.currentDate.compareTo(this.experationDate)){

Return super.getBasePrice() \* .1; // returns 90% discount

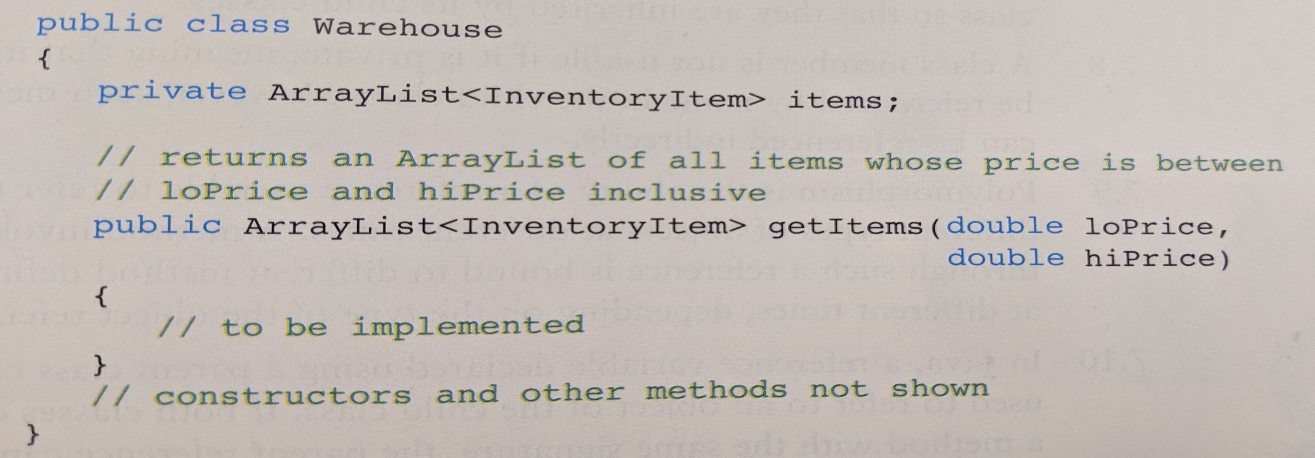
Else{  
 return super.getBaseaPrice;

}

}

}

1. Consider the following Warehouse class, which stores the list of items in the warehouse inventory:



The method getItems behaves as described in the comment. Implement getItems below.

Public ArrayList<InventoryItem> getItems(double loPrice, double hiPrice){

ArrayList<InventoryItem> between = new ArrayList<InventoryItem>();

For( InventoryItem item : this.items){

If( item.getPrice() > loPrice && item.getPrice < hiPrice){

Between.add(item);

}

Return between;

}

1. Consider an abstract class Duck, representing all available ducks. Every duck can quack and walk, which are publicly accessible functionalities of every duck. While every duck walks the same way, they quack differently which makes quacking an abstract feature of a duck. Every duck has a weight (float) which determines their ability to float (no pun intended), which ability can be checked by everybody through canFloat method. The weight of a duck can be set through a method available only for all concrete duck implementations. A RubberDuck, a kind of a duck, has all the features of a duck (yes, it quacks), but when asked to walk – it acts differently, because a rubber duck can’t walk, so it overrides a duck walk feature and does nothing. An example of a duck is also a MallardDuck that apart of quacking and walking, can also fly.

Draw a class diagram depicting classes, with names, attributes and methods along with proper relationship between classes. Skip constructors and destructors, but don’t forget to annotate elements visibility using UML notation. For the actual attributes types or method signatures use Java notation. Preceed a name with a ‘/’ symbol to indicate slanted (italicized) text.

Use identifiers (class, attributes and method names) from the following list (you can repeat them if you need it.) (Hint: you should include all of them in your solution): Duck, MallardDuck, RubberDuck, walk, quack, fly, weight, canFloat, setWeight.

Public abstract class Duck{

Private float weight;

Public void quack();

Public void walk();

Public void fly();

Public Boolean canFloat();

Public float setWeight();

}

Public class MallardDuck extends Duck{

Public void quack(){

System.out.println(“Quack!”);

}

Public void walk(){  
 System.out.println(“The mallard walks away”);

}

Public void fly(){  
 System.out.println(“The mallard flys away”);

}

Public void setWeight(float weight){

this.weight = weight;

}

Public Boolean canFloat(){

If(this.weight < 10){  
 System.out.println(“The mallard can certainly float”);

}

Else{  
 System.out.println(“The mallard sunk”);

}

}

}

Public class RubberDuck extends Duck{

Public void Quack(){

System.out.println(“Squeaky Quack”);

}

Public void walk(){

System.out.println(“Rubber ducks cannot walk but it floats away”);

}

Public void fly(){  
 System.out.println(“The rubber duck cannot fly);

}

Public Boolean setWeight(float weight){

This.weight = weight;

}

Public Boolean canFloat(){  
 If(this.weight < 0.5){

System.out.println(“The rubber duck continues to

Float”);

}

Else{  
 System.out.println(“the rubber duck sinks to the bottom of the tub.”);

}

UML Diagram:

|  |
| --- |
| Duck |
| * weight : float |
| + quack() : void  + walk() : void  + fly() : void  + setWeight : void  + canFloat : Boolean |

|  |
| --- |
| MALLARD |
| * weight : float |
| + quack() : void  + walk() : void  + fly() : void  + setWeight : void  + canFloat : Boolean |

|  |
| --- |
| RUBBER DUCK |
| * weight : float |
| + quack() : void  + walk() : void  + fly() : void  + setWeight : void  + canFloat : Boolean |

1. As best you can, define polymorphism and how it can be used to produce more robust programs. Think about the organization of your Pokemon lab. What features of Java were used in the lab assignment in order to achieve polymorphism? What do you think are its advantages and disadvantages?

Polymorphism is using an instance of a class to act like another class. One feature of java that is used is function overloading, because they allow us to have similar functions that perform differently depending on the class. One advantage is that it allows us to write much less code. One disadvantage is that it can make programs more complex.

Scrap.