Midterm
CPE 203 – Summer 2018
Instructor: Foote

Name: _____
(Please print legibly)

Section:

Notes:

When answering problems that require you to write code:

- You do NOT need to comment or import anything.
- All code must be written in Java.
- Your code will be graded on **functionality, design and efficiency**.

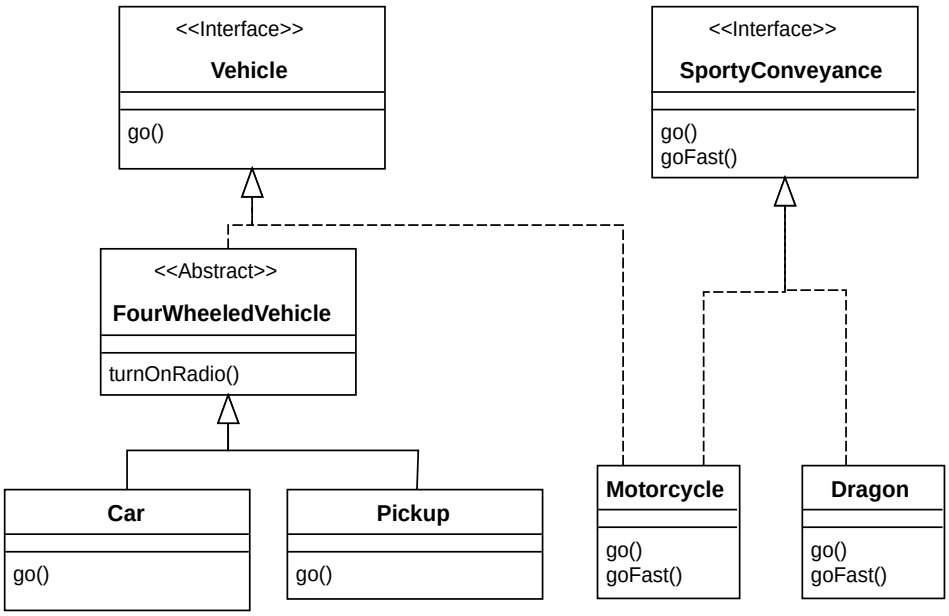
There is partial Java documentation for selected classes at the end of this exam. In the code you write, you may assume that these classes have been imported for you.

I promise not to discuss this exam with anyone who has not yet taken it.

Signature: _____

1. Classes, Interfaces and Types

Consider the following type hierarchy:



Inside a method, the following references have been declared an initialized to something not null but not shown. Other code that is not shown might include additional classes or interfaces not show in this diagram.

```
Vehicle vehicle = ... ;
FourWheeledVehicle fourWV = new ... ;
Car car = ... ;
SportyConveyance sportyConveyance = new ... ;
Motorcycle motorcycle = ... ;
Dragon dragon = ... ;
```

For each code fragment below, write **A** if the fragment will always compile and run, **M** if the fragment will compile but might fail at runtime, and **F** if the fragment will fail to compile. You may assume that the declared methods will not fail when called.

Code Fragment	A/M/F	Code Fragment	A/M/F
`motorcycle.goFast();`		`vehicle = new Pickup();`	
`dragon.goFast();`		`fourWV.go()`	
`sportyConveyance.goFast();`		`vehicle = new Dragon();`	
`vehicle.goFast();`		`car = fourWV;`	
`((Vehicle) sportyConveyance).go();`		`vehicle.turnOnRadio();`	
`motorcycle = (Motorcycle) sportyConveyance;`			
`((SportyConveyance) vehicle).goFast();`			
`((SportyConveyance) vehicle).turnOnRadio();`			
`((FourWheeledVehicle) sportyConveyance).turnOnRadio()`			

2. Creating Classes and Methods

Assume the following partial class/interface definitions for **Bank, Account,** and **Customer.** *Read all the code before you begin!*

a) Fill in all the methods marked **TODO** .

```
public class Bank {
    private final String name;
    private final List<Account> accounts;

   /* TODO
    * Initialize name and accounts to an
    * empty ArrayList
    */
    public Bank(String name) {









    }

    public void addAccount(Account a){ accounts.add(a); }

    /* TODO
     * Return the total balance on deposit in the bank, in cents.
     */
    public int getTotalDeposits() {









    }

    /* TODO
     * Return the number of accounts at the bank.
     */
    public int getNumberOfAccounts() {









    }
```

```
    /* TODO
     * Returns a string containing the bank name and total number
     * of accounts, in the following format:
     *     "SLO Bank (237 accounts)"
     */
    public String toString() {









    }
}

public class Account {
    private int balance;      // Account balance in cents
    private Customer owner;

    /* TODO
     * Initialize balance and owner
     */
    public Account(int balance, Customer owner) {









    }

    public Customer getOwner() {
        return owner;
    }

    public int getBalance() {



    }

    public void addToBalance(final int amount) {




    }

    public void subtractFromBalance(final int amount) {




    }
}
```

```
public class Customer {

    // fields not shown

    public Customer(String name) {
        // Implementation not shown
    }

    public String getName() {
        // Implementation not shown
    }

    /**
     * Returns the number of years this customer has been with
     * the bank.
     */
    public int getYearsAtBank() {
        // Implementation not shown
    }

    // Other details of the Customer class not shown
}
```

(2 continued) Answer the following questions using the above classes.

b. Fill in the method below so that it creates and returns a **Bank** object with two accounts, each with a zero balance and a different owner. Use any names you like for the bank and the owners. You may assume the method is part of a class other than **Bank**; the containing class is not shown.

```
public Bank createTestBank() {









}
```

c. Add a method to the **Bank** class to charge an account maintenance fee to all accounts. It's OK for a balance to become negative. The fee is given in cents. You may assume the method is part of the **Bank** class; the containing class is not shown.

```
public void chargeMaintenanceFee(final int fee)
{









}
```

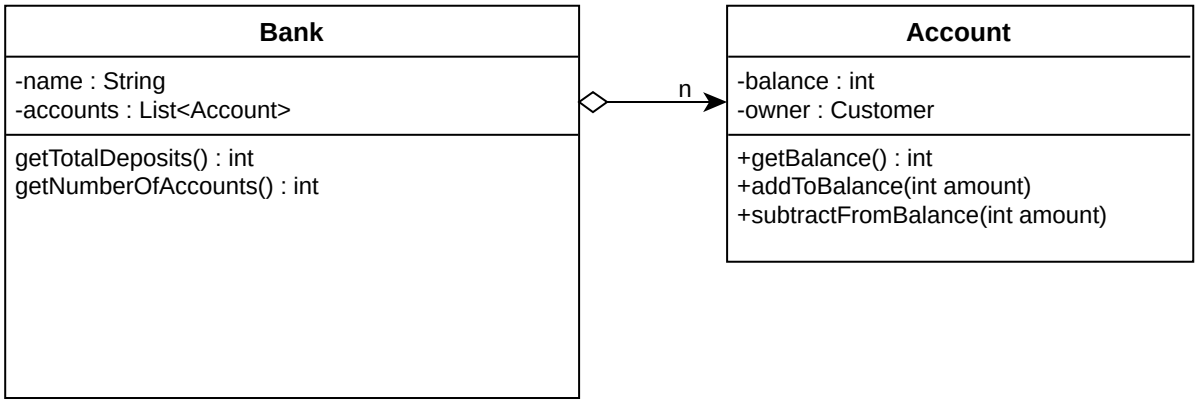
3. **Code Refactoring** – The bank using the banking system from the previous problem is planning to expand. You know that, at some point in the future, the list of accounts won't be kept in a simple list like they are now, but you don't yet know how they're going to be kept.

   You've been asked to implement new functionality on the banking system: A system that will be used for giving all accounts that have been with the bank for over 5 years a loyalty reward. If the bank weren't about to expand, you'd be tempted to just copy-paste the code from `chargeMaintenanceFee()`, above, and modify it so that it works. However, now is a good time to refactor the code so that the iteration through the list of customers occurs in only one place, so you decide to do that instead.

   You will add a new method to **Bank** to iterate through the list of accounts, and apply some operation to each account. Then you will re-write `chargeMaintenanceFee()` so that it uses your new method. Finally, you will write a new method called `applyLoyaltyReward(int minYears, int reward)` that applies a loyalty reward of `reward` cents to each account that has been with the bank for `minYears` years or more.

   Define a new functional interface to represent the operation to be applied to each account.

a. complete the following UML diagram so that it includes any classes, interfaces or methods you have added or modified as part of your refactoring. Any anonymous classes may be shown with the name "from XXX" where "XXX" indicates where the class is defined. Solid lines and non-italic text are fine.



b. Give the code for any new named classes or interfaces you introduce in your refactoring here:

c. Fill in the implementation of the following methods of **Bank**. You may assume these methods are part of the **Bank** class; the containing class is not shown.

```
public void applyOperation(                                          ) {




}

public void chargeMaintenanceFee(final int fee) {




}

public void applyLoyaltyReward(final int minYears, final int reward) {



}
```

3. `Object.equals()` and `Object.hashCode()`

Complete the following class:

```
public final class PersonName {

    public final String firstName;   // guaranteed to not be null
    public final String lastName;    // guaranteed to not be null

    public PersonName(String firstName, String lastName) {
        // implementation not shown
    }

    /**
     * Returns true if other is a PersonName instance with equivalent
     * values for firstName and lastName.  Otherwise returns false.
     */
    public boolean equals(Object other) {
        // TODO:  Fill in code here









    }

    /**
     * Returns a value consistent with the definition of equals()
     * such that PersonName instances can be used as the key in
     * a hash table.
     */
    public int hashCode() {
        // TODO:  Fill in code here




    }
```

Class Object

Constructor Summary	
Object()	

Method Summary	
boolean	equals(Object obj) Indicates whether some other object is "equal to" this one.
int	**h**ashCode() Returns a hash code value for the object.
String	toString() Returns a string representation of the object.

Interface List<E>

Method Summary	
boolean	add(E e) Appends the specified element to the end of this list.
void	add(int index, E element) Inserts the specified element at the specified position in this list.
void	clear() Removes all of the elements from this list.
E	get(int index) Returns the element at the specified position in this list.
int	indexOf(Object o) Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
E	remove(int index) Removes the element at the specified position in this list.
boolean	remove(Object o) Removes the first occurrence of the specified element from this list, if it is present.
E	set(int index, E element) Replaces the element at the specified position in this list with the specified element.
int	size() Returns the number of elements in this list.
void	sort(Comparator<? super E> c) Sorts this list according to the order induced by the specified Comparator.

Class ArrayList<E>

Implements *List<E>*

Constructor Summary	
ArrayList() Constructs an empty list with an initial capacity of ten.	

Method Summary	
boolean	add(E e) Appends the specified element to the end of this list.
void	add(int index, E element) Inserts the specified element at the specified position in this list.
E	get(int index) Returns the element at the specified position in this list.
int	indexOf(Object o) Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
E	remove(int index) Removes the element at the specified position in this list.
boolean	remove(Object o) Removes the first occurrence of the specified element from this list, if it is present.
E	set(int index, E element) Replaces the element at the specified position in this list with the specified element.
int	size() Returns the number of elements in this list.
void	sort(Comparator<? super E> c) Sorts this list according to the order induced by the specified Comparator.

Interface Comparator<T>

Method Summary:

int	compare(T o1, T o2) Compares its two arguments for order. Returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second.

Interface Map<K,V>

Method Summary:

boolean	containsKey(Object key) Returns true if this map contains a mapping for the specified key.
boolean	containsValue(Object value) Returns true if this map maps one or more keys to the specified value.
Set<Map.Entry<K,V>>	entrySet() Returns a Set view of the mappings contained in this map.
V	get(Object key) Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
boolean	isEmpty() Returns true if this map contains no key-value mappings.
Set<K>	keySet() Returns a Set view of the keys contained in this map.
V	put(K key, V value) Associates the specified value with the specified key in this map (optional operation).
V	remove(Object key) Removes the mapping for a key from this map if it is present (optional operation).
int	size() Returns the number of key-value mappings in this map.

java.util

Class Objects

Method Summary

Modifier and Type	Method and Description
`static <T> int`	**`compare`**`(T a, T b, Comparator<? super T> c)` Returns 0 if the arguments are identical and `c.compare(a, b)` otherwise.
`static boolean`	**`deepEquals`**`(Object a, Object b)` Returns `true` if the arguments are deeply equal to each other and `false` otherwise.
`static boolean`	**`equals`**`(Object a, Object b)` Returns `true` if the arguments are equal to each other and `false` otherwise.
`static int`	**`hash`**`(Object... values)` Generates a hash code for a sequence of input values.
`static int`	**`hashCode`**`(Object o)` Returns the hash code of a non-`null` argument and 0 for a `null` argument.
`static boolean`	**`isNull`**`(Object obj)` Returns `true` if the provided reference is `null` otherwise returns `false`.
`static boolean`	**`nonNull`**`(Object obj)` Returns `true` if the provided reference is non-`null` otherwise returns `false`.
`static <T> T`	**`requireNonNull`**`(T obj)` Checks that the specified object reference is not `null`.

(blank page)