C212 Lab 6:

Objectives:

- Object Oriented Design (Objects, Classes, Encapsulation)
- Object Oriented Concepts (Polymorphism, inheritance, etc)
- Continue working with Arrays and ArrayLists
- Writing of Classes and Tests

Submission:

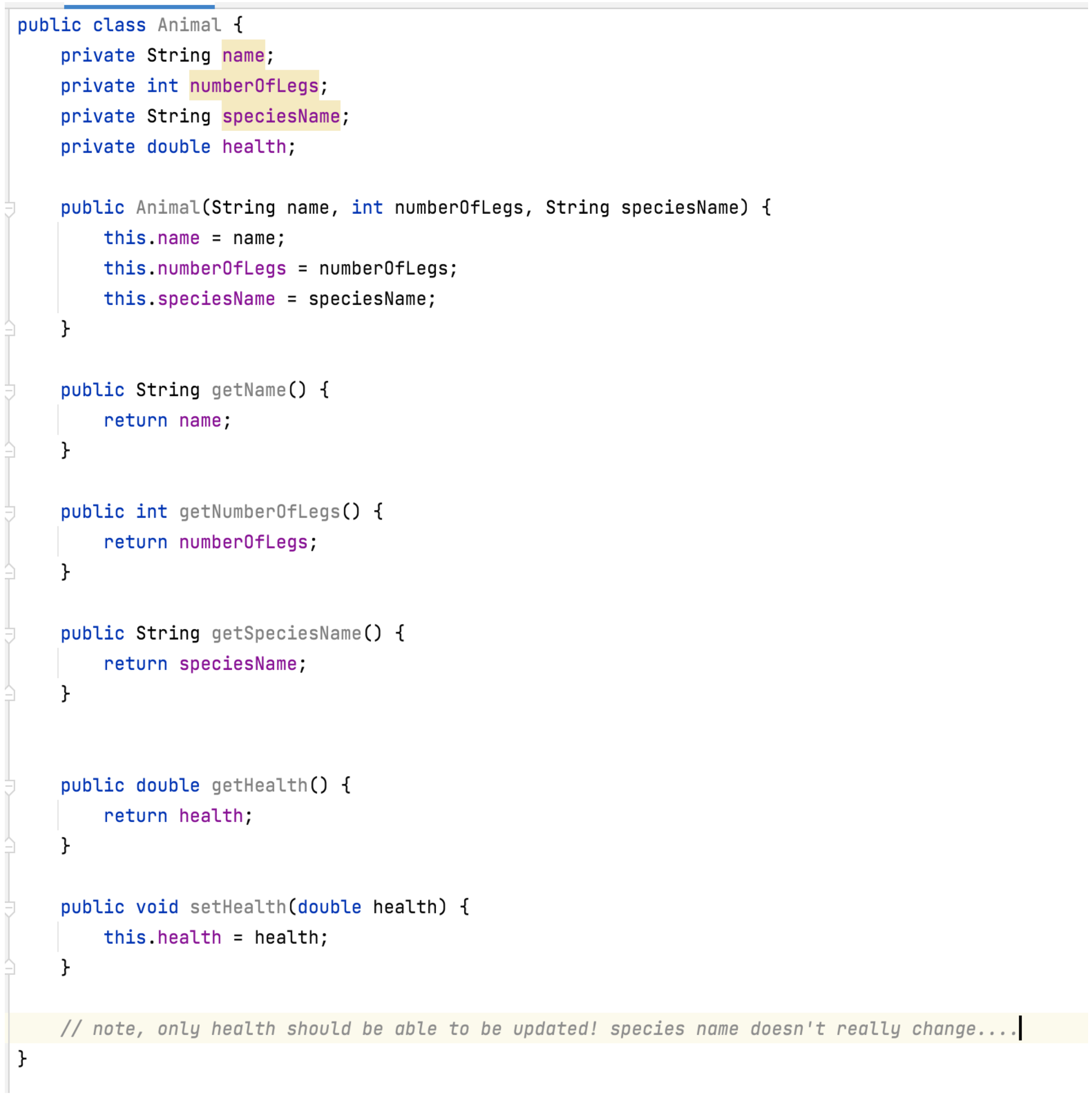
- Please submit all java files zipped together into **one ZIP file** and turn into the respective lab 6 location on canvas.

Theory

Object-oriented design is fundamental knowledge to your success as a Java (or C#, Python, C++, etc.) developer. There are four OOP (object-oriented programming) principles that you must be aware of and actively implement in your programs, though we won't talk about polymorphism at the moment:

1. Encapsulation: Encapsulation is **restricting access by default** to class members (fields and methods). By default, your class members should be using the strictest (least visible) access modifiers possible. Typically, this means that class fields should be private, and you should add getters and

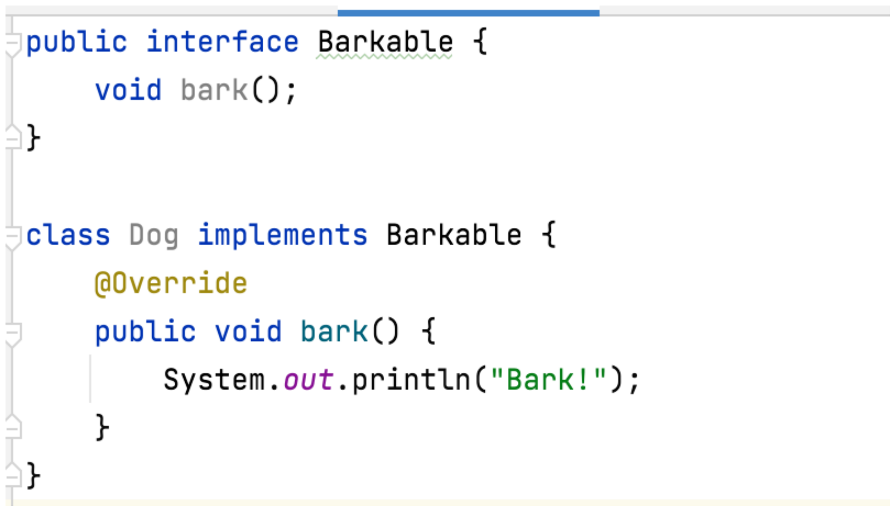
setters to access them. An example is below:



```java
public class Animal {
    private String name;
    private int numberOfLegs;
    private String speciesName;
    private double health;

    public Animal(String name, int numberOfLegs, String speciesName) {
        this.name = name;
        this.numberOfLegs = numberOfLegs;
        this.speciesName = speciesName;
    }

    public String getName() {
        return name;
    }

    public int getNumberOfLegs() {
        return numberOfLegs;
    }

    public String getSpeciesName() {
        return speciesName;
    }


    public double getHealth() {
        return health;
    }

    public void setHealth(double health) {
        this.health = health;
    }

    // note, only health should be able to be updated! species name doesn't really change....
}
```

2. Abstraction: Abstraction in OOP means that classes can "be" an extension of something else, but can also hide the internal details of the parent. For example, all classes in Java extend from the Object class, and thus all objects are instances of the Object class, but Object's methods are hidden to you by default, so you don't know how Object.toString works.
3. Inheritance: Inheritance is similar to abstraction, but you can think of interfaces (through which inheritance is done) as a specification of what is to

be done.



```java
public interface Barkable {
    void bark();
}

class Dog implements Barkable {
    @Override
    public void bark() {
        System.out.println("Bark!");
    }
}
```

Now is also a good time to learn about the **protected** keyword: as opposed to private, which is only accessible to the class a member is declared in, protected is accessible to all subclasses of a class, in addition to the class itself. You will be using protected or private members in this lab, where appropriate.

Part 1:

Create a **Person** class that is the parent class of all of the following classes. The person class should have:

- String firstName
- String lastName
- String birthdate

Create another class named **Athlete** which inherits from the Person class. This class will be the parent class of all classes below it.
The athlete class should be an **abstract class** and have:

- String athleteId
- String athleteTeam
- String athleteType (Soccer, Basketball, Hockey)
- toString(): returns a string
- Abstract method: public abstract int getTotalTimePlayed():

Relationship:
The Athlete class inherits from the Person class, and is the parent class for the following classes: **HockeyAthlete, BasketballAthlete,** and **SoccerAthlete.** Again, these three classes should all inherit from the Athlete class.

The HockeyAthlete class should have:

String arenaType
int numOfPeriods
int minutesPerPeriod
toString():
getTotalTimePlayed(): returning a int

The BasketBall Athlete class should have:
String arenaType
int numOfQuarters
int minutesPerQuarter
toString():
getTotalTimePlayed(): returning a int

The SoccerAthlete class should have:
String arenaType
int numOfHalves
int minutesPerHalf

toString():
getTotalTimePlayed(): returning a int

For each class do the following:

- Declare private instance variables
- Declare public constructors, getters, and setters
- Provide a no argument constructor initializing fields to default values.

The abstract method getTotalTimePlayed() needs to be implemented in all child classes of the Athlete class.

Implement getTotalTimePlayed() as below:
HockeyAthlete: numOfPeriods * minutesPerPeriod
SoccerAthlete: numOfHalves * minutesPerHalf
BasketballAthlete: numOfQuarters * minutesPerQuarter

Implement toString() as below:
Return total time played as a string

Now write a test program to test all of your classes in a main method. This should be a separate java file. Also write a JUnit test for getTotalTimePlayed() by creating objects of all three athlete classes.

Sample Output:

Enter the number of Athletes: 2
Enter the following information for Athlete 1:
Name: Lionel Messi
Birthdate: June 24, 1987
Athlete Type: Soccer
Team Played for: Paris Saint-Germain
ID: 1
Total Time Played: 90 Minute.

Enter the following information for Athlete 2:

Name: Lebron James
Birthdate: December 30th, 1984
Athlete Type: Basketball
Team Played for: Los Angeles Lakers
ID: 2
Total Time Played: 48 Minutes

Part 2

You will be creating an abstract BankAccount class and two subclasses, SavingsAccount and CheckingAccount.
Savings accounts have a withdrawal/transfer transaction limit of 6 per month, while checking accounts have no such limit.

Your BankAccount should have the following fields/methods (remember that fields should be private or protected):

- String accountOwnerName
- double currentAccountBalance
- BankAccount(String accountOwnerName, double currentAccountBalance) {
  }
- double getCurrentAccountBalance()
- void setCurrentAccountBalance(double newBalance)
- String getAccountOwnerName()
- double deposit(double amountToDeposit)
- abstract double withdraw(double withdrawAmount, int currentMonth)
  If withdraw amount is greater than account balance but account balance is positive, return account balance. Otherwise, return full withdrawal amount.
- abstract void transferMoney(double amountToTransfer, BankAccount accountToTransferTo, int currentMonth)

Your CheckingAccount class should extend BankAccount and implement all methods, in addition to adding a constructor. It has no withdrawal limit. For example, a valid constructor would be:

```
public CheckingAccount(String accountOwnerName, double currentAccountBalance) {
    super(accountOwnerName, currentAccountBalance);
}
```

In addition, when transferring money from a checking account, there will be a **3%** fee charged by the bank (only 97% of the money will be transferred).

Your SavingsAccount class will contain the following additional fields and methods, in addition to adding a constructor:

- int withdrawalsThisMonth
- int currentMonth
- int getTotalWithdrawalsThisMonth()

And must update these on each withdrawal (transferring money from a savings account counts as a withdrawal).
An IllegalArgumentException should be thrown when depositing or withdrawing if the amount is 0 or negative.
An IllegalArgumentException should be thrown when withdrawing from a savings account or transferring money from a savings account if the total number of withdrawals that month is > 6.

You are expected to write comprehensive tests for this problem, including, but not limited to, testing depositing, withdrawing, and transferring money from one account to another.

Part 3

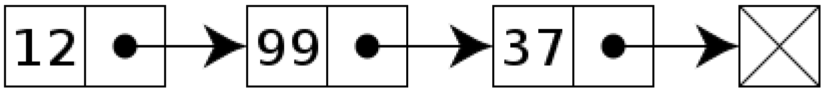
You will be creating your own data structures in Java. You will demonstrate your knowledge of OOP principles, including abstraction and encapsulation, by creating the LinkedStringList and SimpleStringSet classes.

LinkedStringList and SimpleStringSet will all inherit from an interface you will create, called StringCollection.

The StringCollection interface will have the following methods:

- int getSize()
- void add(String e)
  Adds the string to the collection
- boolean contains(String e)
  Returns true if the collection contains the string
- boolean remove(String e)
  Removes the first element e from the collection, if it exists. Returns true if the element was removed.
- void addAllFromCollection(StringCollection collection)
  Adds every element from the specified collection to this collection. If collection.getSize() is 0, throw an IllegalArgumentException.
- List<String> toList()
  Returns a java.util.List<String> list

Linked lists offer a different way to structure data as opposed to an array or array-backed structure (such as an ArrayList). It is defined by a value (head) and a reference to another linked list, if it is not the last element in the list.



(node 12 contains a reference to node 99, node 99 contains a reference to node 37, node 37 contains no reference)
You can read more about linked lists here

The StringLinkedList class will implement StringCollection and will have the following additional fields and methods:

- String value
- String getValue()
- StringLinkedList nextElement
- StringLinkedList getNextElement()
- StringLinkedList(String value) {}
  This constructor, if invoked, creates a StringLinkedList of size 1, with no nextElement (should be set to null)
- StringLinkedList(String value, StringLinkedList nextElement) { }
- void add(String e, int index)
  Add the string to the collection at the specified index

- boolean isLastElementInList()

For the method *int getSize(),* you may calculate the length of the linked list by *1 + nextElement.getSize()*

Note: you cannot remove the first element of a StringLinkedList! (this is to make it easier)

The StringSet class will implement StringCollection and will be backed by an ArrayList. It will have the following additional fields and methods:

- StringSet() {}
  Create empty arraylist
- ArrayList<String> elements
  The actual elements in the set

Additionally, *void add(String e)* will add the string to the set ONLY if it does not already exist.

You are expected to write several JUnit tests for this problem.