UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA
Denkleiers • Leading Minds • Dikgopolo tša Dihlalefi

Department of Computer Science

Faculty of Engineering, Built Environment & IT

University of Pretoria

COS212 - Data Structures & Algorithms

Practical 0 Specifications:
Introduction to Java

Total Marks: 170

Contents

1 General instructions:

- If you have not yet joined the Discord server please do so now: `https://discord.gg/AEtk5GjgcH`
- This assignment should be completed individually; no group effort is allowed.
- Be ready to upload your assignment well before the deadline, as no extension will be granted.
- You may not import any of Java's built-in data structures. Doing so will result in a mark of zero. You may only make use of native arrays where applicable. If you require additional data structures, you must implement them yourself.
- If your code does not compile, you will be awarded a zero mark. Only the output of your program will be considered for marks, but your code may be inspected for the presence or absence of certain prescribed features.
- All submissions will be checked for plagiarism.
- Read the entire assignment before you start coding.
- You will be afforded unlimited upload opportunities but this will be reduced for future practicals.
- Make sure your IDE did not create any packages or import any libraries which are not allowed.

2 Plagiarism

The Department of Computer Science considers plagiarism a serious offense. Disciplinary action will be taken against students who commit plagiarism. Plagiarism includes copying someone else's work without consent, copying a friend's work (even with permission) and copying material from the Internet. Copying will not be tolerated in this course. For a formal definition of plagiarism, the student is referred to http://www.library.up.ac.za/plagiarism/index.htm. **If you have any questions regarding this, please ask one of the lecturers to avoid misunderstanding.** Also, note that the OOP principle of code reuse does not mean you should copy and adapt code to suit your solution.

3 Outcomes

Upon completing this assignment, you will have revised and should be familiar with the following concepts:

- **String Manipulation**
  - Operations on Strings such as concatenation, substring, contains and character manipulation.
- **File Input Handling**

  - Techniques for reading data from files.
  - Handling file input/output exceptions.

- **Use of Generics**
  - Implementing parameterized types in classes and methods.
  - The benefits provided by generics.

- **Implementation of Interfaces**
  - Specifically focusing on the Comparable interface.
  - Overriding the `compareTo` method for custom sorting and comparison.

- **Inheritance**
  - The concept of superclass and subclass.
  - Utilization of the Object class and method overriding (e.g., `toString`, `equals`).
  - Polymorphism.

- **Object References and Basic Data Structures**
  - Creating and manipulating object references.
  - Implementing a custom linked list.

4 Introduction

This practical is designed to familiarize students with the Java programming language. The underlying concepts are assumed prior knowledge and the only new work should be familiarizing yourself with the language. These concepts will be needed in future practicals an assignments.

To ensure a cohesive learning experience, a scenario has been developed to integrate each task. Understanding the scenario is not vital to complete the practical but it should provide context and help you to make reasonable assumptions.

This practical carries no marks and **will not contribute to your semester mark.** Its sole purpose is to illustrate the key concepts you need to understand for this module.

We have made in-depth set of videos to help you complete this practical. You will find the playlist here: `https://www.youtube.com/playlist?list=PLJB3NVIxlhqn5TdmjJid6d7wwgrkrIn7F`

5 Scenario

2024 is an Olympic year and UP has decided to host its own Olympics. Students from a variety of faculties will compete in classic UP sports such as:

- *Parking Spot Hunting*: An event where students compete to find a parking spot closest to campus. The student with the smallest distance to their first lecture wins.
- *Textbook Weightlifting*: Students compete to carry the heaviest backpacks filled with textbooks. The student with the heaviest total weight wins. Outdated editions that are no longer relevant to the course count double.
- *Class Registration Roulette*: Students randomly select courses in a high-stakes game of chance, with the aim of creating a schedule without 7:30 AM classes or Friday lectures. Each 7:30 AM lecture counts 1 penalty point and each Friday lecture counts 1 penalty point with 7:30 lectures on a Friday counting 3 penalty points. The student with the smallest number of penalty points wins.

You have been approached to create a system to manage acceptance, facilitate events, and declare winners.
Students have submitted results containing:

- Their personal details including:
  - name
  - surname
  - age
  - degree
- An application message or "bio"
- The event they competed in
- Their best result for their chosen event
  - *Parking Spot Hunting* will show the smallest distance.
  - *Textbook Weightlifting* will have a comma-separated list of normal book weights then a dollar sign ($), then a comma-separated list of outdated book weights.
  - *Class Registration Roulette* will have a comma-separated list of 7:30 classes then a dollar sign ($), then a comma-separated list of Friday classes.

These data points will be separated by hashtags (#).

6 Tasks

6.1 Task 1

The goal of this task is to create a set of methods (a "method" is the name for a function in Java) that will be used to determine which applicants are accepted to compete in the UP Olympics.

To be accepted, the applicants' bio must meet at least one of three conditions.

1. The bio contains the phrase "no cap" (case sensitive)
2. The bio is exactly 42 characters long
3. The bio (with spaces and case and ignored) contains a palindrome [1].

For now, these methods should be written in the `Task1And2` class whose functionality you will reuse later in section 6.3 (Task 3). Below is a UML class diagram to which your implementations should adhere. You may add your own methods but you may not remove any given methods or change any given method signatures.

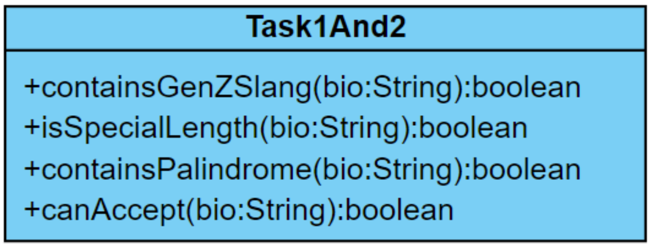

Task1And2
+containsGenZSlang(bio:String):boolean
+isSpecialLength(bio:String):boolean
+containsPalindrome(bio:String):boolean
+canAccept(bio:String):boolean

Figure 1: Task 1 methods

6.1.1 Task 1.1

In this sub-task, you will implement the method which checks if the bio contains the phrase "no cap". Below is the method signature in Java.

```
public boolean containsGenZSlang(String bio){
//implement here
}
```

This method should return *true* if the bio contains "no cap" and *false* if not.

6.1.2 Task 1.2

In this sub-task, you will implement the method which checks if the bio is 42 characters long. Below is the method signature in Java.

```
public boolean isSpecialLength(String bio){
//implement here
}
```

This method should return *true* if the bio is exactly 42 characters long and *false* if not.

[1] A palindrome is a word that is spelled the same forwards as backward. "kayak", and "rotor" are examples of palindromes. For our purposes, a palindrome must contain at least two distinct letters so "a" and "bb" are not palindromes but "abba" and "race car" are palindromes.

6.1.3 Task 1.3

In this sub-task, you will implement the method that checks if the bio (with spaces ignored) contains a palindrome.
A palindrome is a word that is spelled the same forwards as backward. "kayak", and "rotor" are examples of palindromes. For our purposes, a palindrome must contain at least two distinct letters so "a" and "bb" are not palindromes but "abba" and "race car" are palindromes. Since we ignore spaces and cases "Race Car" and "Race car" are both palindromes.
Given the definition above, the word "event" contains a palindrome ("eve") and the phrase "life fun" contains a palindrome ("fe f"). Below is a pseudo-code implementation for the `containsPalindrome` method.
Note: This pseudo-code is just a suggestion. Your method can be different as long as the output remains correct. Just make sure that you know how to read pseudo-code

Algorithm 1 Pseudo-code for `containsPalindrome` method

```
function containsPalindrome(bio)
    beginCharacterIndex := 0
    endCharacterIndex := 0
    while beginCharacterIndex<length(bio) do
        while endCharacterIndex<length(bio) do
            substringToCheck := substring(bio,beginCharacterIndex,endCharacterIndex)
            if isPalindrome(substringToCheck) do
                return true
            end if
            endCharacterIndex := endCharacterIndex + 1
        end inner while
        beginCharacterIndex := beginCharacterIndex + 1
        endCharacterIndex := beginCharacterIndex
    end outer while
    return false
end function
```

This method should return *true* if the bio contains a palindrome and *false* if not. You are responsible for implementing the `isPalindrome` method. Note that the `isPalindrome` method should ignore spaces and case.

This algorithm could be considered inefficient. As an extra exercise try to come up with a mathematical function that shows how many times the `isPalindrome` method is called based on the number of characters in the `bio` parameter.

6.1.4 Task 1.4

In this final sub-task, you will implement the method that checks whether or not an applicant can be accepted based on their bio. The method will take in a String called `bio` and return a boolean.Below is the method signature in Java.

```
public boolean canAccept(String bio){
//implement here
}
```

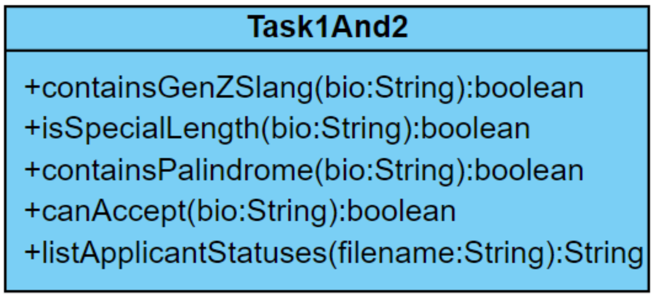
6.2 Task 2

In this task, you will create a method called `listApplicantStatuses` in the `Task1And2` class. It should read the list of applicants from a text file, determine their acceptance status, and return a String with each student's acceptance status in the following format:
student name<space>*student surname*<colon><tab>*accepted/not accepted*<new line>

Each student's status should appear on a new line. Hint: To achieve this you should add "\n" to the end of each student's status. Note that after the last student's status has been printed the cursor should be at the beginning of the next blank line.

Example Output:
Bob Smith:	accepted
John Doe:	not accepted
Jane Price:	not accepted
Abby Arnold:	accepted

You should make use of the method you created in section 6.1 (Task 1). The `String.split` method may also be useful.
Below is a UML class diagram to which your implementations should adhere. You may add your own methods but you may not remove any given methods or change any given method signatures.



Task1And2

+containsGenZSlang(bio:String):boolean
+isSpecialLength(bio:String):boolean
+containsPalindrome(bio:String):boolean
+canAccept(bio:String):boolean
+listApplicantStatuses(filename:String):String

Figure 2: Task 1 and Task 2 methods

The text file layout is as follows:
name<hashtag>*surname*<hashtag>*age*<hashtag>*degree*<hashtag>*bio*<hashtag>*chosenEvent*<hashtag>
eventResult
For example:
Alice#Smith#20#Electrical Engineering#I'm gonna win, no cap.#Parking Spot Hunting#15m

6.3 Task 3

In this task, you will create a `Competitor` class to hold the details common to each student. You will then create sub-classes to represent competitors for each event.

6.3.1 Task 3.1

Create the `Competitor` class, it should adhere to this UML class diagram:

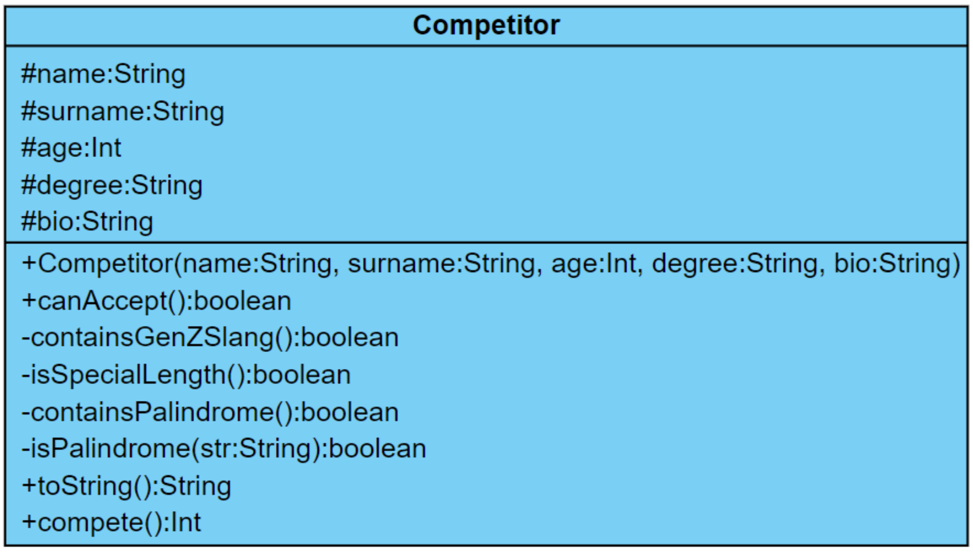

Competitor
#name:String
#surname:String
#age:Int
#degree:String
#bio:String
+Competitor(name:String, surname:String, age:Int, degree:String, bio:String)
+canAccept():boolean
-containsGenZSlang():boolean
-isSpecialLength():boolean
-containsPalindrome():boolean
-isPalindrome(str:String):boolean
+toString():String
+compete():Int

Figure 3: UML Class diagram for Competitor

- Members
  - name:String
    * This is the name of the student who has applied as given in the text file
  - surname:String
    * This is the surname of the student who has applied as given in the text file
  - age:int
    * This is the age of the student who has applied as given in the text file
  - degree:String
    * This is the degree in which the student is enrolled as given in the text file
  - bio:String
    * This is the student's bio as given in the text file
- Methods
  - Competitor(name:String, surname:String, age:Int, degree:String, bio:String)
    * This is the constructor for a Competitor.
    * It takes in parameters corresponding to each member variable.
    * The parameters should be used to initialise the relevant member variable.

- canAccept():boolean
    - This method returns true if the competitor can be accepted based on their bio.
    - The implementation is the same as in 6.1 (Task 1).
    - Slight changes need to be made to the signature from Task 1
- containsGenZSlang():boolean
    - This method returns true if the bio contains "no cap".
    - The implementation is the same as in 6.1 (Task 1).
    - Slight changes need to be made to the signature from Task 1
- isSpecialLength():boolean
    - This method returns true if the bio is exactly 42 characters long.
    - The implementation is the same as in 6.1 (Task 1).
    - Slight changes need to be made to the signature from Task 1
- containsPalindrome():boolean
    - This method returns true if the bio contains a palindrome.
    - The implementation is the same as in 6.1 (Task 1).
    - Slight changes need to be made to the signature from Task 1
- isPalindrome(str:String):boolean
    - This method returns true if the input String is a palindrome.
    - This method should ignore spaces and ignore case.
- toString():String
    - This method returns a String representation of a Competitor
    - The String should be in the following format:

      name<space>*surname*<comma><space>*age*<space><open bracket>*degree*<close bracket>
    - For example: John Doe, 20 (BSc Computer Science)
- compete():int
    - This method should just return -1.
    - It will be overridden by sub-classes
    - Note: It would be best practice to make this method abstract with no implementation. However, for marking purposes, it has been stubbed instead.

Note that the `canAccept`, `containsGenZSlang`, `isSpecialLength` and `conatainsPalindrome` methods no longer take a String parameter. They will use the `bio` attribute from the `Competitor` class instead. Do not remove these methods from the `Task1And2` class, just copy them across and make the appropriate changes as shown in the UML class diagram.

6.3.2 Task 3.2

In this sub-task, you will create the specialisations of the `Competitor` class. There are three specialisations: `ParkingHunter`, `TextbookLifter`, and `ClassOrganiser`. These classes will override certain methods of the base `Competitor` class. Your implementations should adhere to the below UML class diagram. Note that the constructor for each sub-class has been left out of the class diagram to save space. However, it will still be listed in the required methods.

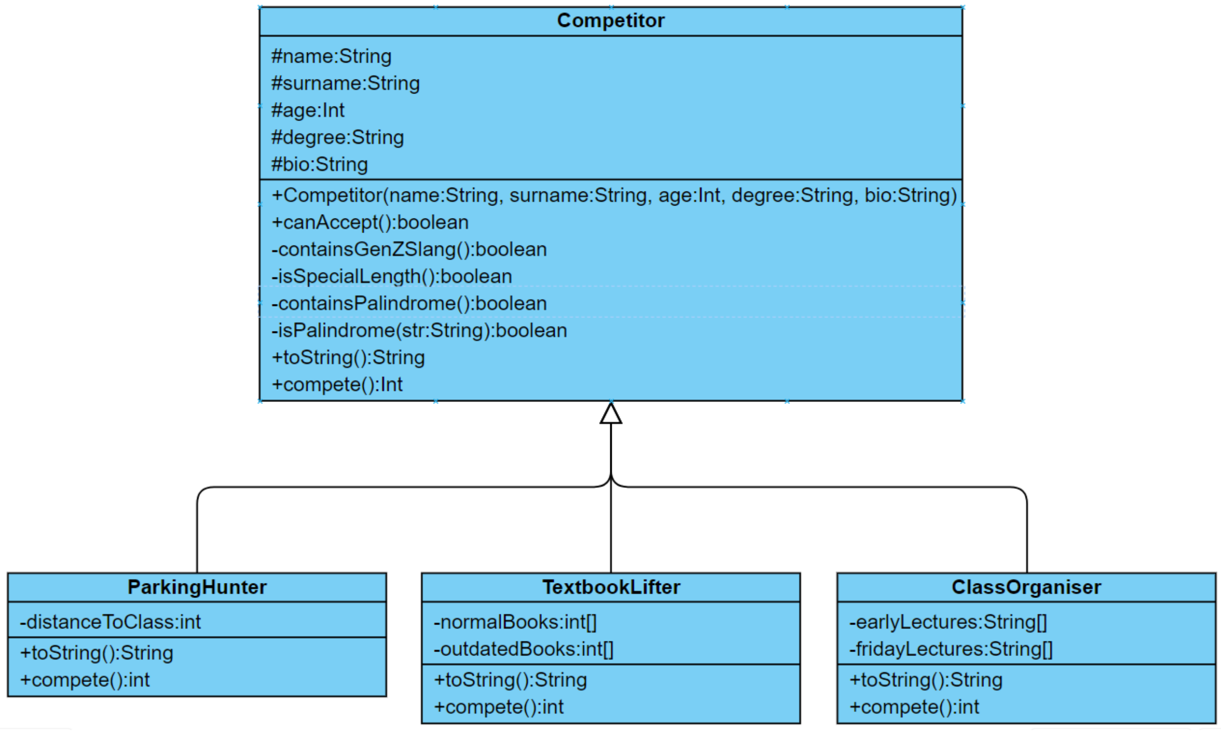


Figure 4: Subclasses of Competitor

6.3.2.1 ParkingHunter

- Members
  - distanceToClass:int
    - ∗ This holds the smallest distance to the competitor's first lecture.
- Methods
  - ParkingHunter(name:String, surname:String, age:Int, degree:String, bio:String, result:String)
    - ∗ This is the constructor for a ParkingHunter.
    - ∗ It should use the relevant parameters to initialise the base class.
    - ∗ The final parameter (`result:String`) should be used to initialise the `distanceToClass` attribute.
    - ∗ For Parking Spot Hunters, the result string will be in the following format:
      <smallestDistance>m
      For example:
      10m
  - toString():String

    * This calls the base classes toString() method and returns its result with this event name (Parking Spot Hunting), the competitor's score in brackets, a colon, and a space prepended.
    * *event name*<bracket>*compete score*<bracket><colon><space><base class toString>
    * Note that there is no newline at the end
    * For example:
      Parking Spot Hunting(5): John Doe, 20 (BSc Computer Science)
  - compete():int
    * This method returns the distance from the competitor's first class so that it can be compared to other competitors in the same event.

6.3.2.2 TextbookLifter

- Members
  - normalBooks:int[]
    * This holds the weight of each normal book in the competitors' backpack as an array of integers.
  - outdatedBooks:int[]
    * This holds the weight of each outdated book in the competitors' backpack as an array of integers.
    * Recall that outdated books count double.
- Methods
  - TextbookLifter(name:String, surname:String, age:Int, degree:String, bio:String, result:String)
    * This is the constructor for a TextbookLifter.
    * It should use the relevant parameters to initialise the base class.
    * The final parameter (`result:String`) should be used to initialise the `nomalBooks` and `outdatedBooks` attributes. (The `String.split` method will be helpful)
    * For Textbook Weightlifters, the result string will be in the following format:
      <normalBook1>kg<comma><normalBook2>kg<comma>...<normalBookN>kg<dollar-sign><outdatedBook1>kg<comma><outdatedBook2>kg<comma>...<outdatedBookN>kg
      For example:
      2kg,1kg,1kg$2kg,3kg,4kg
  - toString():String
    * This calls the base classes toString() method and returns its result with this event name (Textbook Weightlifting)the competitor's score in brackets, a colon, and a space prepended.
    * *event name*<bracket>*compete score*<bracket><colon><space><base class toString>
    * Note that there is no newline at the end

    * For example:
      Textbook Weightlifting(105): John Doe, 20 (BSc Computer Science)
  - compete():int
    * This method returns the final weight of all the textbooks in the competitor's backpack with the outdated books counting double.

6.3.2.3 ClassOrganiser

- Members
  - earlyLectures:String[]
    * This holds the names of the modules that have 7:30 lectures in an array of Strings.
  - fridayLectures:String[]
    * This holds the names of the modules that have Friday lectures in an array of Strings.
- Methods
  - ClassOrganiser(name:String, surname:String, age:Int, degree:String, bio:String, result:String)
    * This is the constructor for a ClassOrganiser.
    * It should use the relevant parameters to initialise the base class.
    * The final parameter (`result:String`) should be used to initialise the `earlyLectures` and `fridayLectures` attributes. (The `String.split` method will be helpful)
    * For Class Registration Roulette competitors, the result string will be in the following format:
      <earlyLecture1><comma><earlyLecture2><comma>...<earlyLectureN><dollar-sign><fri
      For example:
      COS212,COS110,COS216$COS110,COS221
  - toString():String
    * This calls the base classes toString() method and returns its result with this event name (Class Registration Roulette) a colon and a space prepended.
    * *event name*<bracket>*compete score*<bracket><colon><space><base class toString>
    * Note that there is no newline at the end
    * For example:
      Class Registration Roulette(1): John Doe, 20 (BSc Computer Science)
  - compete():int
    * This method returns the final number of penalty points for this event. Each module in the `earlyLectures` array counts 1 penalty point. Each lecture in the `fridayLectures` also counts 1 penalty point. If a module appears in both arrays it counts 3 penalty points instead of 2.

∗ For example:
earlyLectures:["COS212","COS216"]
fridayLectures:["COS210","COS212"]
This totals 5 penalty points. 1 for "COS216" and 1 for "COS210" but 3 for "COS212" as it appears in both arrays.

6.4 Task 4

In this task, you are going to create and implement interfaces. Interfaces are used when you want to specify that multiple classes must implement certain behaviors, but the classes themselves are not necessarily related through an "is-a" relationship. You will create and implement the `Cheerable` interface for each specialisation of `Competitor`. You will then implement the `Comparable` interface for each specialisation of `Competitor`. Below is a UML class diagram showing how the interfaces interact with the existing classes.

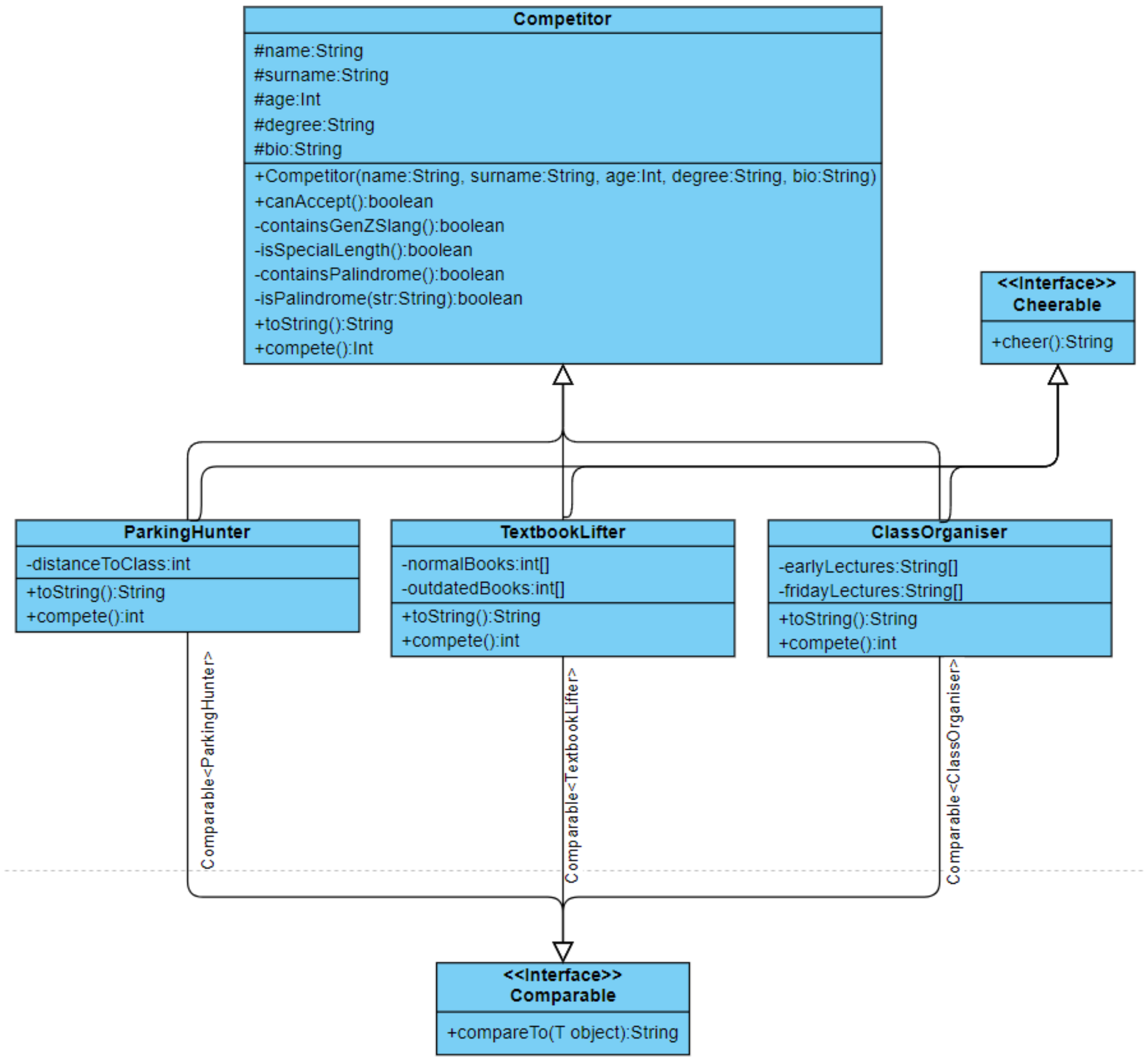


Figure 5: Competitors and their interfaces

6.4.1 Task 4.1

In this sub-task, you will create the `Cheerable` interface and implement it for each type of `Competitor`. The interface will specify one method:

```
String cheer()
```

Each type of `Competitor` should implement the `Cheerable` interface and therefore will need to implement the `cheer` method. Each implementation should return the following String:
Go<space>*name*<comma><space>go!<space>You're<space>the<space>best<space>at<space>!*event name*

Note that there is no newline at the end
For example:
Go John, go! You're the best at Parking Spot Hunting! //(for ParkingHunter)
Go John, go! You're the best at Textbook Weightlifting! //(for TextbookLifter)
Go John, go! You're the best at Class Registration Roulette! //(for ClassOrganiser)

6.4.2 Task 4.2

In this sub-task, you will implement the `Comparable` interface for each type of competitor. The comparable interface is part of the java standard library so you do not need to create it.
Competitors should be compared by the results of their `compete` methods such that if you sort in descending order the winner is first. Be careful: the winner is not always the person with the biggest score. For example in *Parking Spot Hunting* the winner is the person with the smallest distance to their first class. So when competitors or sorted in descending order the competitor with the smallest distance to their first class should come first.

Click here for a guide on how to override the `compareTo()` method. Note that the possible returns are: -1, 0 and 1. Below is the url if the link does not work.
`https://www.geeksforgeeks.org/how-to-override-compareto-method-in-java/`

6.5 Task 5

In this task, you are going to create a basic ordered linked list that can hold any class that implements the `Comparable` interface as data (this is often referred to as a generic type). This ordered linked list will only have an `insert` method and a `toString`. Data must be inserted in descending order according to the `compareTo` method. i.e. The "winner" should be the head of the list. Below is the UML class diagram your implementation should adhere to.

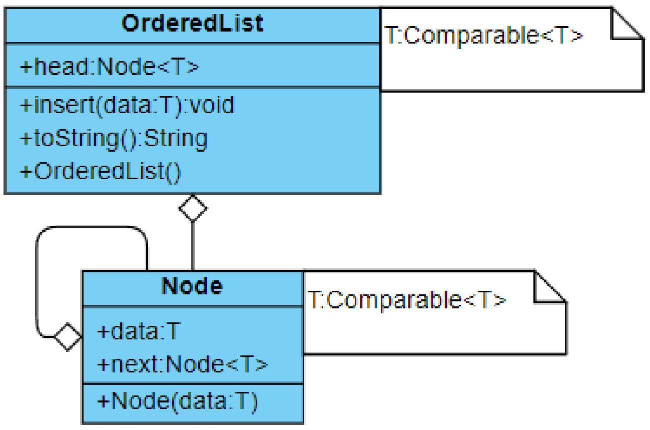


Figure 6: Ordered List UML class diagram

6.5.1 Node<T>

- Members
  - data:T
    - The data that the node holds. This will be used to place the node in the correct position in the list.
    - Note: T must be `Comparable`
  - next:Node<T>
    - The node that comes after this node in the list.
    - Note: All succeeding nodes should have data that "loses" to this node's data. Sometimes that "losing" data is "bigger" and sometimes it is "smaller". This should be handled by the *compareTo* method.
- Methods
  - Node(data:T)
    - This is the constructor for a Node.
    - It receives data of type T (which must implement the *Comparable* interface) as a parameter.

6.5.2 OrderedList<T>

- Members
  - head:Node<T>
    - The node at the beginning of a list
    - Note: All other nodes should have data that "loses" to this node's data. Sometimes that "losing" data is "bigger" and sometimes it is "smaller". This should be handled by the `compareTo` method.
    - The head node should be the winner of the event if T is a specialisation of Competitor.

- Methods
  - insert(data:T)
    - This method creates a new node using the given data and places it in the correct position in the list.
    - It receives data of type T (which must implement the *Comparable* interface) as a parameter.
    - If the data is the same according to the `compareTo` method then the new node should be placed after the existing node in the list.
  - toString():String
    - This method returns a String representation of the list.
    - The String representation should be in this format:
      Head-><square bracket>*toString of data*<square bracket>-><square bracket>*toString of data*<square bracket>...->NULL
    - Note that there is no newline at the end
    - For example if the list contains 99, 5, and 1:
      `Head->[99]->[5]->[1]->NULL`
  - OrderedList()
    - The constructor for an ordered list.
    - This should explicitly set `head` to NULL.

6.6 Suggested Test

To test that everything works as it should you can try to bring everything together. You can:

- Read all the possible competitors in from a text file
- Create `Competitor` objects of the correct sub-class
- Filter out the competitors who cannot be accepted based on their bio
- Cheer for each competitor who has been accepted
- Create three Ordered Lists, one for each event
- Insert the correct competitors into the correct lists
- Print out the contents of each list
- Print the winner of each event

This is not an exhaustive test but it should serve as a general idea of the functionality that is expected to be in place.

7 Submission instructions

Do not submit any custom function classes.
Your code should be able to be compiled and run with the following commands:
javac *.java
java Main

Once you are satisfied that everything is working, you must create an archive of all your Java code into one archive called uXXXXXXXX.{tar.gz/tar/zip} where XXXXXXXX is your student number. Submit your code for marking under the appropriate link before the deadline. **Make sure your archive consists of only java files and no folders or subfolders.**

Please ensure that you thoroughly test your code before submitting it. Just because your code runs with local testing does not mean that your code works for every situation. **DO NOT USE FITCH FORK AS A DEBUGGER**

8 Submission checklist

- Cheerable.java
- ClassOrganiser.java
- Competitor.java
- Node.java
- OrderedList.java
- ParkingHunter.java
- Task1And2.java
- TextbookLifter.java

9 Allowed imports

- import java.io.FileNotFoundException;
- import java.io.FileReader;
- import java.util.Scanner;