Homework 3

Data Struct. and {Abstraction, OOP} (Term III/2021–22)

built on 2022/05/18 at 09:34:59 **due:** *fri may 27 @ 11:59pm*

This assignment is aimed at giving you practice on working with larger programs, using object-oriented techniques we've discussed in class, including cohesion/coupling considerations and use of interfaces, generics, and higher-order functions. You will implement a few Java programs, test them thoroughly, and hand them in. There is a starter package, which you must download to begin working on this assignment.

Collaboration

We interpret collaboration very liberally. You may work with other students. However, each student ***must*** write up and hand in his or her assignment separately. Let us repeat: You need to write your own code. You must not look at or copy someone else's code. You need to write up answers to written problems individually. The fact that you can recreate the solution from memory will be taken as proof that you actually understood it, and you may actually be interviewed about your answers.

Logistics: The Starter Package and Handing In

We've created a starter zip file for you to get started with this assignment. It is available for download from the course website. When unzipped, the starter file at the top level contains *two* folders, each corresponding to a task in this assignment. For every task, we've set up Gradle for you, so each of the task folder has a `build.gradle`, as well as other Gradle-related files.

```
Top Level
└─ deque-palindrome
   ├─ build.gradle
   └─ ⟨other files for deque etc.⟩
└─ zuul
   ├─ build.gradle
   └─ ⟨other files for zuul⟩
```

In your IDE (e.g., IntelliJ), you will want to import three separate projects, one for each task. The use of Gradle is highly encouraged as this will allow for easy JUnit testing, plus this will be how we're going to run your code. This means you're creating each project from existing source and point your IDE to `build.gradle`—not just the folder.

What to Hand In? You will hand in one file (`a3.zip`) to Canvas. To avoid huge submissions full of junk, you will run the `clean` task in Gradle for each task. This can be done either using the Gradle panel in IntelliJ or on the command line via "gradlew clean".

Task 1: Deque and Generalized Palindromes (18 points)

You will extend one of your Deque implementations from the previous assignment to conform to an interface and use it to solve "real-world" problems. When you download the starter package, the folder for this problem contains the following files:

- `CharacterComparator.java` — an interface for comparing characters.
- `TestPalindrome.java` — a class containing JUnit tests for Palindrome.
- `TestOffByOne.java` — a class containing JUnit tests for OffByOne.

IMPORTANT: In the starter files, the JUnit test files were commented out because they depend on your implementation of certain classes and methods. Throughout this task, you'll be asked to uncomment the test files to activate them as you go along.

We are also providing the following empty files for you to save your work:

```
Deque.java    Palindrome.java      OffByOne.java      OffByN.java
```

Subtask I: Deque Interface

The first subtask is mechanical; it will help solidify our understanding of interfaces.

In this subtask, you will create an **interface** in a file named `Deque.java` that contains all of the methods that appear in both `ArrayDeque` and `LinkedListDeque`. See the Deque API section from the previous assignment.

Now that you have defined the interface, there are 2 more to-do items:

1. Modify your `LinkedListDeque` and/or `ArrayDeque` so that they implement the `Deque` interface. This means adding `implements Deque<T>` to the line declaring the class, where `T` is the name of your type parameter.
2. Add `@Override` tags to each method that overrides a `Deque` method.

NOTE: If you did not have a working `Deque` from the previous assignment, create a file called

```
LinkedListDeque.java
```

and put in the following code:

```java
public class LinkedListDeque<T> extends LinkedList<T> implements Deque<T> {}
```

This black magic will give you an implementation of a `Deque`, based on Java's built-in `LinkedList`.

Subtask II: From A Word To A Deque

This subtask involves programming inside `Palindrome.java`. Write a method with the following signature:

```java
public Deque<Character> wordToDeque(String word)
```

where `Deque` refers to the interface you have just created in the previous task.

Given a `String`, the method `wordToDeque` should return a `Deque` where the characters appear in the same order as in the given string. For instance, if the word is "meow", then the `Deque` you return should have "m" at the front, followed by "e", and so on.

ATTENTION: This is a good time to uncomment the `TestPalindrome` test file. You will be asked to write more tests soon. For now, the file contains a simple test for your `wordToDeque` implementation.

Subtask III: Is This A Palindrome?

Inside `Palindrome.java`, write a method

```java
public boolean isPalindrome(String word)
```

The `isPalindrome` method should return `true` if the given word is a palindrome, and `false` otherwise. As you already know, a *palindrome* is defined to be a word that is the same whether it is read left-to-right or right-toleft. For example, "a", "racecar", and "noon" are all palindromes. But "horse" and "aaab" are not. You might also notice that any word of length 1 or 0 is always a palindrome. Your function treats the input as case sensitive, so we treat "A" and "a" as different.

Extra Requirements. You **must** use a `Deque` in implementing `isPalindrome`. This means using `wordToDeque` to convert the given string into a `Deque`. After that, it should be obvious how to read from the front and the back of a `Deque` and compare them. You really should *not* be using the `get` method of `Deque`; it will just make things unnecessarily complicated

Write Tests. The final part of this subtask involves writing JUnit tests for your `isPalindrome` method. Add a few tests to `TestPalindrome` to make sure your implementation of `isPalindrome` works correctly.

Subtask IV: Generalizing Palindrome

The ultimate goal of this subtask is to implement a method

```java
public boolean isPalindrome(String word, CharacterComparator cc)
```

inside the `Palindrome` class. The method will return `true` if the word is a palindrome according to the character comparison test provided by the `CharacterComparator` passed in as `cc`. To compare whether characters `x` and `y` are equal, instead of comparing them with `x == y` as usual, we say that `x` and `y` are equal if the given `CharacterComparator` says they are equal. A character comparator is defined as:

```java
public interface CharacterComparator {
    /** Returns true if characters are equal by the rules of the
                implementing class. */
    public boolean equalChars(char x, char y);
}
```

Here are all the steps required of you for this subtask:

1. Create a class called `OffByOne`, which implements `CharacterComparator` such that `equalChars` returns `true` for characters that are different by exactly one. For example:

   ```java
   OffByOne obo = new OffByOne();
   obo.equalChars('a', 'b');  // ==> true
   obo.equalChars('r', 'q');  // ==> true
   obo.equalChars('a', 'e');  // ==> false
   obo.equalChars('z', 'a');  // ==> false
   obo.equalChars('a', 'a');  // ==> false
   ```

   Tips: Char values in Java are really just numbers. For example, `'a'` is actually just another way of writing 97, and `'b'` is another way of writing 98.

2. Uncomment the test file `TestOffByOne`. Write plenty of tests in `TestOffByOne` to make sure your implementation is correct.

3. Write the new `isPalindrome` method. For this part, to allow for odd-length palindromes, do *not* check the middle character for equality with itself. So “flake” is an off-by-1 palindrome,even though the letter a is not one character off from itself. This is consistent with our earlier `isPalindrome` method, any zero or 1 character word is always a palindrome.

4. Add plenty of your own tests for `isPalindrome`. They should go into `TestPalindrome.java`.

Subtask V: Off by N

Finally, you will implement a class `OffByN`, which should implement the `CharacterComparator` interface, as well as a constructor that takes an integer. Hence, the public methods and constructors of this class are:

```
public OffByN(int N)
public boolean equalChars(char x, char y)
```

The `OffByN` constructor should create an object whose `equalChars` method return **`true`** if and only if the characters are off by `N`. For example:

```
OffByN offBy5 = new OffByN(5);
offBy5.equalChars('a', 'f');  // ==> true
offBy5.equalChars('f', 'a');  // ==> true
offBy5.equalChars('f', 'h');  // ==> false
```

Writing Tests. It’s always a good habit to write tests. Although you aren’t required to hand in the tests for `OffByN`, we recommend that you write plenty of good tests for your own benefits. A natural place to do so would be in a file `TestOffByN.java` that you will create.

Task 2: Zuul (12 points)

World-of-zuul is a simple, rudimentary implementation of a text-based adventure game, designed by David Barnes and Michael Kölling to illustrate a few concepts related to OO class design. The original design, as you will see, is intentionally horrifyingly bad.

You can find the starter code in the starter package downloadable from the course website. Your goal in this task is to fix the design and make some nontrivial extensions in the end.

Subtask I: Read The Code. Code reading is an important skill that requires practice. You first task here is to read some of the existing code and try to understand what it does. By the end, you will probably understand most of it. To start working on this task, try to skim through much of the codebase. Make note of where things are as you go along.

Subtask II: Understand Good Design and Refactoring. You will begin by watching the following YouTube videos, which walk you through the process of “upgrading” the design of Zuul through refactoring, as well as teaching a Java construct known as **`enum`**:

- `https://www.youtube.com/watch?v=chaE5EvdfZQ`
- `https://www.youtube.com/watch?v=n_hpbJlZuNc`

To recap the lecture, we spoke about *coupling* and *cohesion*. Coupling is the about the degree of dependencies between separate units of a program. If two classes depend closely on many details of each other, they are tightly coupled. We should **aim for loose coupling**: classes do not depend (too much) on the details of other classes. Encapsulation is a way to reduce coupling.

Cohesion is about how many and how diverse a single unit (e.g., a method or a class) is responsible. If a programming unit is responsible for one logical task, it has high cohesion. We should **aim for high cohesion**. In concrete terms, this means: a class should represent just one and only one well-defined entity, and a method should be responsible for one and only one well-defined task.

When we spot significant code duplication, it is a strong indicator of bad design; it is usually a symptom of low cohesion (the opposite of what we want). Refactoring is often the cure.

Subtask III: Make Small Extensions. As a little exercise to get warmed up, make some changes to the code. There is nothing to hand in for this subtask. But this will help you navigate this codebase. Example changes you can make:

- change the name of a location to something different.
- change the exits—pick a room that currently is to the west of another room and put it to the north
- add a room (or two or three, ...)

Subtask IV: Refactor for Good Design. Now that you're familiar with the code and you've studied some design principles, you will refactor the Zuul code so that it observes good design principles and follows best practices you have just learned.

Here are some glaring things, among others, that you really should fix:

- In the `Game` class, `printWelcome` and `goRoom` contains a big chunk of repeated code. The root cause seems to be that each of these methods aims to do multiple things. Improve the code by writing a separate, more cohesive method whose only task is to print information about the current location—and make them call it.
- The `Game` class makes (very) heavy use of the exit info from the `Room` class. These *public* member variables are nasty. Get rid of them. While you're at it, streamline it using a `HashMap`. Ideally, making changes local to the `Room` class should affect no other classes (loose coupling). But this is not the case here. Removing the exit variables, the code won't even compile. The classes are really tightly coupled. Aim to reduce coupling by making sure encapsulation works properly.
- `setExits` has the knowledge about possible exit directions baked into it. Reduce this coupling by instead writing `setExit(String direction, Room neighbor)`.
- When you see a wall of `if`s repeated over and over, that should cause you to question whether they can be combined or eliminated.
- Remember the enum video you just watched?

Subtask V: More Functionality. You'll extend the program to support *all* of the following:

1. Add two more directions "up" and "down". Be sure to test it by adding a few rooms that use these directions.
2. Implement a command "look". This will be in addition to the existing commands such as "go ..." and "quit". The "look" command will print the possible exits in your current location (i.e., looking around the room), similar to what you would display when you enter the room using the "go" command. To get an idea for what "look" does, suppose you're in the campus pub, the command will display:

```
You are in the campus pub
Exits: east
```

3. Implement a command “back” that takes you back to the last room you were in.
4. Add a special kind of room: a *magic transporter room* where every time you enter it you are transported to a random room in your game. *How would you represent such a room?*

Expectations: This task is open-ended and will be graded with this in mind. Our aim for this task is to get you thinking about good class design (by fixing a bad one and extending it to do wonderful things). To this end, the code you’re submitting must be professionally written (comments and indentation!) and will be graded for

- correctness
- good object-oriented design
- appropriate use of language constructs
- style (commenting, indentation, etc.)