CSE 143: Computer Programming II — Autumn 2019

HW4 Bonus: Hangman Bonus (due Thursday, November 7, 2019 11:30pm)

This is an extra credit assignment that is worth just **one point**. It is intended for students who want to explore how to improve the HangmanManager and how to use inheritance to create this variation. There will be minimal assistance available to students working on this problem since it is extra.

In this version, you will improve on two flaws in the original HangmanManager:

- We will modify the record(char) method so that it wins immediately if it can when the user is down to one guess (making the program even more evil)
- We will modify the words() and guesses() methods so that they prevent a client from changing our internal data structures

Each of these will be worth $\frac{1}{2}$ points. You can attempt just one bonus or you can attempt both.

Both of these changes will be implemented through **inheritance**. The idea is that we want to create a minor variation of the original class and inheritance is particularly good at capturing that kind of variation. We can override particular methods and change their behavior. Remember that you can still call the superclass version of a method even if you are overriding it. If the superclass has a method foo, you can call super.foo to call the overridden method. If you need a review of inheritance covered in CSE 142, you can read this.

You should define a class called HangmanManager2 that extends HangmanManager. You will have to include a constructor for HangmanManager2 that takes the same parameters as HangmanManager. It will need to have a call on the superclass constructor as the first line of the new constructor. You call a superclass constructor by saying:

```
super(<expression>, <expression>, ..., <expression>)
```

The textbook has an example of calling the superclass constructor on pages 606 to 607 (chapter 9).

You should modify the following line of code in HangmanMain to have it use the new version of HangmanManager:

```
HangmanManager hangman = new HangmanManager2(dictionary2, length, max);
```

Notice that the only thing we have to change is the kind of HangmanManager being constructed (in this case, a HangmanManager2).

In terms of grading, if you get one of the two modifications to work correctly, you will get $\frac{1}{2}$ bonus point. To get the whole bonus point, you have to have both working and you have to use good programming style. For example, you wouldn't want to include methods in your class that don't need to be there. If you are not changing an inherited method, then it doesn't need to appear in the subclass. In terms of comments, you don't have to repeat the comments from the superclass. You should instead assume that someone is familiar with the superclass description of the methods and should describe how the behavior is changed.

Your solution should work for any legal implementation of HangmanManager. This means that you can't solve the problem by modifying your solution to HangmanManager (e.g., adding methods or making fields protected).

You should name your file HangmanManager2.java and you should turn it in electronically from the Homework tab on the class web page if you choose to do this bonus. The details for the bonus point are included below.

Even More Evil ($\frac{1}{2}$ points)

There are many improvements you can make to the evil hangman algorithm to make it even more evil. For this bonus point, you will implement one specific improvement. Think of what happens when the user has just one guess left. We go through our normal process of splitting the current set of words into buckets and then picking the bucket with the most words. By doing so, we are likely to keep the user alive to try to win the game. It would be better to just end it if we can by having the player lose.

Consider, for example, the following log using `dictionary.txt`:

```
Welcome to the cse143 hangman game.

What length word do you want to use? 5
How many wrong answers allowed? 5

guesses : 5
guessed : []
current : - - - - -
Your guess? a
Sorry, there are no a's

guesses : 4
guessed : [a]
current : - - - - -
Your guess? e
Sorry, there are no e's

guesses : 3
guessed : [a, e]
current : - - - - -
Your guess? i
Sorry, there are no i's

guesses : 2
guessed : [a, e, i]
current : - - - - -
Your guess? o
Sorry, there are no o's

guesses : 1
guessed : [a, e, i, o]
current : - - - - -
Your guess? u
Yes, there is one u

guesses : 1
guessed : [a, e, i, o, u]
current : - u - - -
Your guess? ...
```

Why let the user keep playing when there is only one guess left? Instead, we will choose the first word in the current set of words that doesn't contain the letter the user has guessed. For the log above, instead of saying that there is one u, the program instead says:

```
guesses : 1
guessed : [a, e, i, o]
current : - - - - -
Your guess? u
Sorry, there are no u's

answer = byrls
Sorry, you lose
```

We will accomplish this by overriding the `record` method. It will still call the superclass version of the method to do most of the work. But before doing so, it will check to see if the user has just one guess left. If so, then it will go through the current set of words and pick the first word that does not contain the letter being guessed. If such a word can be found, then it calls the `clear` method on the current list of words and then adds this word back into the current set of words. That way when the superclass version of record is called, it will find just one word to work with. That one word will cause the user to lose the game immediately. If the number of guesses left is not 1 or if no such word can be found, then it simply calls the superclass version of the `record` method so that the behavior is unchanged.

Protecting Internal Structures ($\frac{1}{2}$ point)

The obvious way to write the `words()` and `guesses()` methods is to return a reference to fields of your `HangmanManager` object and we asked you to solve it that way in writing `HangmanManager`. This is very dangerous, because it means that clients have the ability to damage the internal state of your objects. We could make copies of them to return, but copying is expensive. A better approach is to return an unmodifiable version of each that allows the client to perform "read only" operations but prevents the client from performing any mutating operations.

The `Collections` class in `java.util` has a series of methods that can be used to construct unmodifiable versions of various collections. For example, if you look at `HangmanMain`, you will see that it includes this line of code:

```
List<String> dictionary2 = Collections.unmodifiableList(dictionary);
```

It passes the modifiable dictionary to the method `Collections.unmodifiableList` which returns a reference to a new list that is a "wrapper" for the original. It doesn't make a copy, but it prevents a client from making any changes to the new version of the structure. There is a similar method called `Collections.unmodifiableSet` that can be used to create unmodifiable versions of the words and guesses sets.

For this bonus point, you will override the `words()` and `guesses()` methods to return unmodifiable versions of the sets returned by calls on the superclass versions of these methods.

One simple way to do this would be to override the `words()` and `guesses()` methods by always constructing a new unmodifiable object. That would be inefficient because it constructs a brand new object every time. At the other extreme, you could construct unmodifiable versions once in your constructor and then never change them. That won't work, though, because the set objects might change. The algorithm we are using for evil hangman involves replacing our old set of words with a new set of words and sometimes that will involve having a new set object. So whatever approach we use has to take into account the fact that sometimes we have a new set object to work with but not always.

To get this bonus point, you have to implement this in the efficient way. In particular, your class should create new unmodifiable versions of each set if and only if there is a new object being returned for the original set. For example, suppose that on 10 calls to the `words()` method, there are four different actual set objects returned. Then your class should create four different unmodifiable sets. If you create fewer than four, then your class won't work. If you create more than four, then you won't receive the credit.

Keep in mind that what matters is whether the set is a brand new object, not whether it has a new state. If I add or remove values from a set but I'm still using the same set, then that is a single object. But sometimes the change involves the object itself (a different set object). So your class has to figure out when the underlying set object has changed. You might guess that this happens whenever the record method is called, but you aren't allowed to make assumptions like that. For example, the superclass might never change the set of words. You also might think that a change in the size of the set would signal that it is a new set but once again, this is an assumption you aren't allowed to make. It is possible that the same set object will have different sizes at different times.

You should solve this problem by comparing object references. Recall that you can compare object references for equality. For example, consider the following code:

```
Set<String> s1 = new TreeSet<String>();
Set<String> s2 = new TreeSet<String>();
Set<String> s3 = s1;
```

This constructs two set objects but introduces three set variables. The variables `s1` and `s3` refer to the same object while the variables `s1` and `s2` refer to different objects. You can test that with code like the following:

```
if (s1 == s2) {
    System.out.println ("yes");
} else {
    System.out.println("no");
}

if (s1 == s3) {
    System.out.println("yes");
} else {
    System.out.println("no");
}
```

This prints "no" followed by "yes." The variables `s1` and `s2` aren't equal because they refer to different objects. But the variables `s1` and `s3` are equal because they refer to the same object. You should use tests like these to implement the second bonus point to detect when you have a new set.