Midterm 2 Sample (Part 2) With Solution Key[*]

CSS 162—Computer Programming II

Directions

This part of the exam is closed book and closed notes. No books, notes, electronic devices, or any material or information is allowed. Write neatly and use proper syntax. Partial credit may be awarded. It is highly recommended that you write supplementary explanations if you cannot produce a final answer. Unless the question explicitly asks you to write comments, you do not need to do so.

The first two questions are each worth 5 points. The remaining three questions are each worth 10 points. Please answer the questions on separate sheet(s) of paper, **one question per sheet of paper.** Please turn in this exam sheet with your answer sheet(s).

Questions

1. Given this portion of a definition of a `Date` class:

   ```
   public class Date {
       private int month, day, year;

       public Date(Date inDate) {
           this.month = inDate.getMonth();
           this.day = inDate.getDay();
           this.year = inDate.getYear();
       }

       [... more of the class definition ...]
   }
   ```

   write a good javadoc comment block for the `Date` class' constructor that is shown. You need to also use at least one "@" tag.

   Solution:

[*]As of May 20, 2017. Acknowledgment: Portions are copied from previous UWB CSS exams.

```
/**
 * Deep-copy constructor for Date object.
 *
 * The Date object stores the month, day, year of a date.
 * This constructor is a deep-copy constructor for the
 * class.  The object stores the month, day, and year as
 * ints.
 *
 * @param   inDate   Input object of class Date.
 * @return           Nothing.
 */
```

2. Give the Big-$\mathcal{O}$ notation that best describes the following functions. (By best, I mean simplest and lowest bound using Big-$\mathcal{O}$ notation, for n large. Put another way, give me the Big-$\mathcal{O}$ notation based on what we talked about in class as the correct way to find Big-$\mathcal{O}$.)

   (a) $23n^2 + 2n^2 + n$
   (b) $4n + 11$
   (c) $7n + (6n \log_2 n)$
   (d) $(12/n^3) + 2$
   (e) $3 + 4n^3 + 1799n^2 + 22n \log_2 n$

   You do not need to provide the constants n_0 or c that are part of the formal definition of Big-$\mathcal{O}$.

   Solution:
   (a) $23n^2 + 2n^2 + n \rightarrow \mathcal{O}(n^2)$
   (b) $4n + 11 \rightarrow \mathcal{O}(n)$
   (c) $7n + (6n \log_2 n) \rightarrow \mathcal{O}(n \log_2 n)$
   (d) $(12/n^3) + 2 \rightarrow \mathcal{O}(1)$
   (e) $3 + 4n^3 + 1799n^2 + 22n \log_2 n \rightarrow \mathcal{O}(n^3)$

3. A car manufacturer wants to outfit its high-end sports car with a remote ignition lock. It decides to purchase the code for a generic `Lock` class from a vendor and use it create a new version of the class for the ignition lock of its sports car.

   All instance variables in `Lock` are private. It has the following public methods:

   - `public Lock(Configuration input)`: Constructs an empty `Lock` object according to the settings given in `input`.
   - `public void lock(String code)`: Sets the pertinent instance variables controlling the lock to a "locked" state. The parameter `code` is passed in as a "key" to lock the lock. This method also declares an `IllegalAccessException` is thrown.

- `public void unlock(String code)`: Sets the pertinent instance variables controlling the lock to an “unlocked” state. The parameter `code` is passed in as a “key” to unlock the lock. This method also declares an `IllegalAccessException` is thrown.
- `public void changeKey(String newKey, String code)`: Changes the “key” for locking and unlocking to the value in `newKey`. The value of `code` must match the current “key” in order for the change to be successfully made. This method also declares an `IllegalAccessException` is thrown.
- `public String accessTimes()`: Returns a list of the times of the last 50 attempts to lock or unlock the lock.
- `public String toString()`: Returns a string summarizing the current settings and state of `Lock`.

You are to write a class called `IgnitionLock` that extends the `Lock` class using inheritance, providing the same functionality as `Lock` but that also prevents an `IgnitionLock` object from locking or unlocking the ignition if the transmission is not in Park. Attempts to lock or unlock the ignition lock if the tranmission is not in Park should return a `IllegalAccessException` (part of `java.lang`) — this is a checked exception and thus has to follow the “Catch or Declare” rule.

Additionally, you need to override the `toString` method so that the settings/state report of the lock is appended with a description of whether or not the transmission is in Park. (The summary of the lock settings/state needs to be as detailed as the `Lock` `toString` method produces. Note you are not told how the `Lock` `toString` method works.) You are also to provide a constructor for `IgnitionLock` that accepts the same input and has the same functionality as `Lock`.

A static method `isPark` of the class `CarState` exists that returns `true` if the transmission is in Park and `false` otherwise. The `isPark` method takes no input parameters. You can assume the classes `CarState` and `IllegalAccessException` are already imported.

For this problem, do not worry about writing a no-argument constructor, a copy constructor, or an `equals` method. Finally, remember, you are only writing `IgnitionLock`. You are **not** to write `Lock`.

Solution:

```
public class IgnitionLock extends Lock {
    public IgnitionLock(Configuration input) {
        super(input);
    }

    public void unlock(String code) throws IllegalAccessException {
        if (CarState.inPark()) {
            super.unlock(code);
        } else {
            throw new IllegalAccessException("Not in Park");
        }
    }

    public void lock(String code) throws IllegalAccessException {
        if (CarState.inPark()) {
            super.lock(code);
        } else {
            throw new IllegalAccessException("Not in Park");
        }
    }

    public String toString() {
        if (CarState.inPark()) {
            return super.toString() + " " + "in Park";
        }
        else {
            return super.toString() + " " + "not in Park";
        }
    }
}
```

4. Consider the following class `LinkedList` which holds nodes that store a `String` value:

```
public class LinkedList {
    private class Node {
        private String item;
        private Node next;

        public Node(String item, Node next) {
            this.item = item;
            this.next = next;
        }
    }

    private Node head = null;

    public LinkedList(String item) {
        this.head = new Node(item, null);
    }

    public void append(String input) {
       [... code implementing this method ...]
    }
    public String toString() {
       [... code implementing this method ...]
    }

    public static void main(String[] args) {
        LinkedList a = new LinkedList("high");

        a.append("low");
        a.append("tall");
        a.append("short");

        System.out.println(a.toString());
    }
}
```

When the `main` method of the class is run, the following is output to console:

```
high, low, tall, short
```

Please write the body of the `append` method that will enable the `main` method of the class to be correctly executed. You may **not** assume the existence of any other methods or constructors of `LinkedList` or `Node` than those for which the signature is given above. (Thus, you cannot assume `LinkedList` has an `insert` method, for instance.) You do **not** have to write `toString`.

Solution:

```
public void append(String input) {
    boolean atEnd = false;
    Node position = this.head;
    while (!atEnd) {
        if (position.next == null) {
            atEnd = true;
            position.next = new Node(input, null);
        } else {
            position = position.next;
        }
    }
}
```

You didn't have to do `toString` because the logic of it is basically the same as `append`.

5. Write a static method `skipFactorial` that calculates the "factorial" of an integer by the following algorithm:

$$n!! = n(n-2)!!$$

where I made up the "!!" operator as meaning `skipFactorial`. Like the regular factorial, 0!! = 1, 1!! = 1, 2!! = 2, and !! is not defined for negative numbers. Thus, the following calls:

```
System.out.println(skipFactorial(2));
System.out.println(skipFactorial(4));
System.out.println(skipFactorial(8));
```

should produce the output:

```
2
8
384
```

If input is negative, throw an `IllegalArgumentException` (part of `java.lang`) — this is not a checked exception and thus does not need to follow the "Catch or Declare" rule. (You can assume the class is imported already.)

You may **NOT** use a `while` loop, `for` loop or `do-while` loop to solve this problem; you must use recursion.

Solution:

```
public static int skipFactorial(int n) {
    if (n < 0)
        throw new IllegalArgumentException();
    else if (n == 0)
        return 1;
    else if (n == 1)
        return 1;
    else if (n == 2)
        return 2;
    else
        return n * skipFactorial(n-2);
}
```

Note, while I put in the `n == 2` stopping case as an explicit stopping case, I don't actually need it there; the other stopping cases will give me the right value for $n = 2$.