SYSC 2004 B/C Winter 2015 Final Exam Questions (Released April, 2016)

READ THE FOLLOWING INSTRUCTIONS BEFORE ANSWERING ANY QUESTIONS.

1. Write your name and student number in the spaces provided on Page 1 of this question paper.
2. Attempt all the questions, and budget your time carefully. Read all parts of each question before starting to wrote your solutions. Answer Questions 1 through 4 on the lined (right-hand) pages of your answer booklet. Answer Question 5 on this question paper. Use the blank, left-hand pages in the answer booklet for your rough work. **Solutions written on the blank pages will be assumed to be rough work and will not be graded unless you clearly indicate that you want them to be marked**; for example, by drawing a box around the rough work and writing "Mark this" next to the box.
3. **Exam questions will not be explained, and no hints will be given**. If you think something is unclear or ambiguous, make a reasonable assumption (one that does not contradict the question), write it at the start of your solution, and answer the question. Please do not ask for help unless you believe you have found a mistake in the question paper. If there is a mistake, the correction will be announced to the entire class. If there is no mistake, this will be confirmed, but no additional explanation of the question will be provided.
4. Unless otherwise noted, you may not change any aspects of the classes, variables or methods that are presented in this question paper. For example, you are not permitted to change the visibility of private fields. You are not permitted to change constructor and method signatures; that is, change the names or types of the parameters or the method's return type. You don't have to copy any of the Java comments from this question paper to your answer booklet.
5. When you review your Java code, if you decide you need to insert additional statements, don't rewrite your entire solution. Just write them below or to the side of you code, draw a box around them, and draw an arrow to indicate where they belong.
6. **You can detach the API reference provided at the end of this question paper; however, you cannot take it with you. ALL PAGES OF THIS QUESTION PAPER, INCLUDING THE API REFERENCE, MUST BE TURNED IN. DO NOT REMOVE ANY PAGES FROM THE EXAM ROOM.**

Please read the instructions a second time.

Question 1 - Basic Class Definitions [15 marks]

Here is an incomplete implementation of a class that models a simple green-yellow-red traffic signal (also known as a *traffic light* or *stop light*). At any time, exactly one of the signal's three lights is on. The class provides three methods that determine which of the lights is on, and a mutator method to change the signal to the next light.

```
public class TrafficSignal
{
    public static final int GREEN = 0;  // Green signal light
    public static final int YELLOW = 1; // Yellow signal light
    public static final int RED = 2;    // Red signal light

    /* Records which light is currently on (GREEN, YELLOW, or RED). */
    private int light;

    /**
     * Constructs a new TrafficSignal in which the light specified by
     * parameter light is on. This parameter has value GREEN, YELLOW, or RED.
     */
    public TrafficSignal(int light) {...}

    /**
     * Advances this traffic signal to the next light in the sequence: GREEN to
     * YELLOW, or YELLOW to RED, or RED to GREEN.
     */
    public void advance() {...}

    /**
     * Returns true if this traffic signal is GREEN; otherwise returns false.
     */
    public boolean greenLightIsOn() {...}

    /**
     * Returns true if this traffic signal is YELLOW; otherwise returns false.
     */
    public boolean yellowLightIsOn() {...}

    /**
     * Returns true if this traffic signal is RED; otherwise returns false.
     */
    public boolean redLightIsOn() {...}

    /**
     * Returns a string representation of this TrafficSignal, indicating which
     * light is currently on, in the form, "LLL light" (where LLL is one of
     * "green", "yellow" or "red".
     */
    public String toString() {...}
}
```

Notice that the class defines three public constants, `GREEN`, `YELLOW`, and `RED`, which represent the possible states of the traffic signal. **In your solutions, any statements that modify or check the traffic signal's state must use these constants instead of the literal values 0, 1 and 2.**

Before answering this question, please reread Instruction 4 on Page 2.

(a) Write the complete definition of the constructor. You can assume that parameter `light` will always be one of the values: `GREEN`, `YELLOW`, or `RED`. (Your constructor should not check this.) (2 marks)

(b) Write the complete definition of the `advance` method. Hint: the simplest solution requires no `if` statements or loops. (4 marks)

(c) Write the complete definitions of the `greenLightIsOn`, `yellowLightIsOn` and `redLightIsOn` methods. (5 marks)

(d) `TrafficSignal` overrides the `toString` method it inherits from class `Object`. Write the complete definition of this method. (4 marks)

Question 2 - Collaborating Objects [15 marks]

This question uses the `TrafficSignal` class from Question 1; however, correctly answering that question is not a prerequisite for answering this one.

Here is an incomplete implementation of a class that models a street intersection. An intersection has two traffic signals: an east-west signal and a north-south signal.

```
public class Intersection
{
    private TrafficSignal eastWest;
    private TrafficSignal northSouth;

    /**
     * Constructs a new Intersection that has two traffic signals. The
     * east-west traffic signal is initially green and the north-south traffic
     * signal is initially red.
     */
    public Intersection() {...}

    /**
     * Advances one or both of this intersection's traffic signals.
     */
    public void change() {...}

    /**
     * Returns a string representation of this Intersection that describes
     * the current colours of the intersection's two traffic signals; for example:
     *   "east-west: green light   north-south: red light"
     */
    public String toString() {...}
}
```

When answering parts (a) through (c), remember that the `GREEN`, `YELLOW` and `RED` constants defined in `TrafficSignal` are public and can therefore be used in class `Intersection`.

(a) Write the complete definition of the constructor. (4 marks)

(b) Write the complete definition of the `change` method. Hint: think carefully about the different cases this method must handle, by considering how the traffic signals at a real street intersection work. Sometimes this method should cause only one of the traffic signals to change colour, and sometimes it should cause both traffic signals to change colour. Remember, when one signal is green or yellow, the other signal must be red. (8 marks)

(c) `Intersection` overrides the `toString` method it inherits from class `Object`. Write the complete definition of this method. Hint: this method is very short - read the API for `TrafficSignal` before writing the method. (3 marks)

Question 3 - Object Equality [10 marks]

Attempt Questions 1 and 2 before you answer this question.

Traffic control software needs to be able to determine if the traffic signals at two interactions have the same state.

(a) Define an `equals` method for the `TrafficSignal` class from Question 1. This method overrides the `equals` method that is inherited from class `Object`. The method signature is:

```
boolean equals(Object obj);
```

Two `TrafficSignal` objects are considered to be equal if the same light is on in both signals.

Don't copy your entire solution from Question 1. Just write the `equals` method. (5 marks)

(b) Define an `equalsEastWest` method for the `Intersection` class from Question 2.

The method signature is: `boolean equalsEastWest(Intersection in);`

Consider this code fragment:

```
Intersection bronsonSunnyside = new Intersection();
Intersection bankSunnyside = new Intersection();
        ...
 if (bronsonSunnyside.equalsEastWest(bankSunnyside)) {...}
```

In this example, the `equalsEastWest` method will return `true` if the east-west signals at the Bronson-Sunnyside and Bank-Sunnyside intersections have the same state; otherwise it will return `false`. Hint: remember that a method defined in `Intersection` class can access the private fields of another instance of the same class.

Don't copy your entire solution from Question 2. Just write the `equalsEastWest` method. (5 marks)

Question 4 – Class Inheritance [20 marks]

In Lab 6 you developed a simple hierarchy of Java classes that modelled two types of bank accounts; namely, savings accounts and chequing accounts. The superclass of both types of accounts is an abstract class called `BankAccount`. Here is a complete implementation of that class:

```
public abstract class BankAccount
{
    private double balance;

    public BankAccount(double initialBalance)
    {
        balance = initialBalance;
    }

    public void deposit(double amount)
    {
        balance = balance + amount;
    }

    public void withdraw(double amount)
    {
        balance = balance - amount;
    }

    public double getBalance()
    {
        return balance;
    }

    public abstract void monthEnd();
}
```

Suppose the bank wants to provide a "value" account for people who make a small number of transactions in a month. This type of account earns no interest. The bank normally charges a $5 monthly administration fee for this account. If the minimum monthly balance is $1000 or higher, the administration fee is not charged for that month and all withdrawals during the month are free. If the minimum monthly balance is less than $1000, the administration fee is charged and first 5 withdrawals each month are free, but there is a 25 cent transaction fee for every additional withdrawal until the end of the month.

Define a Java class called `ValueAccount`. The class should provide the following constructor:

- `public ValueAccount(double initialBalance)` – the constructor initializes the account balance to the specified amount.

This question continues on the next page.

You should be able to call the following methods on a `ValueAccount` object:

- `public void deposit(double amount)` – increases the account balance by the specified amount.
- `public void withdraw(double amount)` – decreases the account balance by the specified amount. No check is made to see if the account will be overdrawn (i.e., a negative balance).
- `public double getBalance()` – returns the account balance.
- `public void monthEnd()` – calculates the month's administration fee and transaction fee, based on the minimum monthly balance and the number of withdrawals that month, and withdraws that amount from the account. This method also performs any additional "housekeeping" chores so that the account's methods will work correctly during the next month.

  Here is an example of how the fees are calculated. Suppose that, at the start of the month, the account balance and the minimum balance are $1600. You then make five withdrawals of $100. The account balance and minimum monthly balance are now $1100. You then withdraw $200. The account balance and minimum monthly balance are now $900. You then deposit $1000. The account balance is now $1900, but the minimum balance remains $900. You then make four withdrawals of $100. The account balance is now $1500, but the minimum balance remains $900.

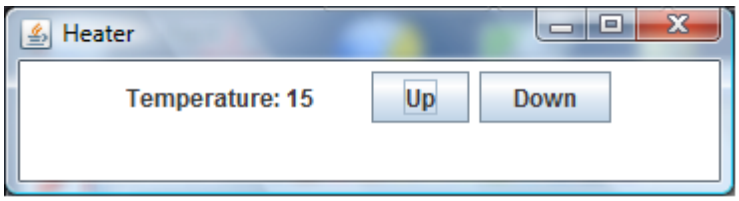
  During the month, you made a total of 10 withdrawals. Because one of the withdrawals caused the minimum balance to go below $1000, during month-end processing you will be charged the $5 administration fee as well as a transaction fee of $1.25 (10 withdrawals – 5 free withdrawals, multiplied by 25 cents per withdrawal), which reduces the account balance to $1493.75. At the start of the next month, the account balance and minimum monthly balance are $1493.75.

Note that:

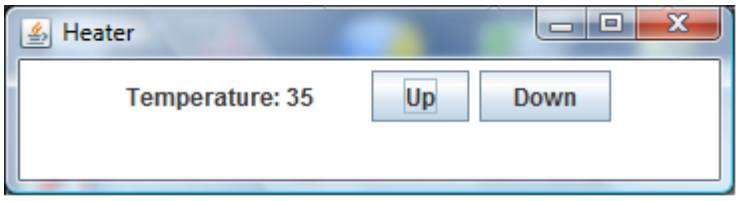
- `ValueAccount` must be a subclass of `BankAccount`;
- You are not permitted to make any changes to the `BankAccount` class. In particular, notice that the `balance` field in `BankAccount` has private visibility;
- It's up to you to decide how your `ValueAccount` class will keep track of the minimum monthly balance and the number of withdrawals during the month;
- Some of the methods that are inherited from `BankAccount` may be completely appropriate for a value account and so should not be overridden in `ValueAccount`. Other methods inherited from `BankAccount` must be overridden in `ValueAccount` to provide the required behaviour. Marks will be deducted if your `ValueAccount` class overrides methods that do not need to be overridden, and if `super.` is used in method calls where it is not required.

Question 5 - Graphical User Interfaces [10 marks]

You are going to write a program that uses the `Heater` class you developed in Lab 2. When the program is run, this window appears:



Initially, the heater's temperature setting is 15 degrees. The Up and Down buttons raise and lower the temperature setting in five-degree increments. After clicking the Up button four times, the user interface looks like this:



The program's classes are provided on the next several pages; however, several statements have been replaced by ruled lines. Your job is to read the classes and write the missing statements on the ruled lines (**not** in your answer booklet). Extra ruled lines have been provided, and a complete solution requires fewer statements than the number of ruled lines.

Class `HeaterUI`

```
import javax.swing.JFrame;

// User interface for a heater.
public class HeaterUI
{
   // Creates and displays the main program frame.
   public static void main(String[] args)
   {
      JFrame frame = new JFrame("Heater");
      HeaterPanel panel = new HeaterPanel();
      frame.getContentPane().add(panel);
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.pack();
      frame.setVisible(true);
   }
}
```

Class `HeaterPanel`

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

/**
 * A panel containing buttons to increment or decrement a heater's
 * temperature setting, and a label that displays that setting.
 */
public class HeaterPanel extends JPanel
{
    private Heater heater;
    private JLabel label;

    /**
     * Initializes the user interface.
     */
    public HeaterPanel()
    {
        // Create a Heater object whose temperature ranges from
        // 10 to 50 degrees.

        heater = new Heater(10, 50);

        // Create the label that displays the current temperature
        // setting.

        label = new JLabel("Temperature: " +
                           heater.getTemperature() + "        ");

        // Create the Up and Down buttons.
```

_____

_____

_____

_____

_____

```
        // Create one instance of ButtonListener, and register it
        // as the listener for both buttons.
        ButtonListener listener =
            new ButtonListener(_____________________________________);

_____________________________________________________________________

_____________________________________________________________________

_____________________________________________________________________

_____________________________________________________________________

        // Place the label and the buttons in the panel.

_____________________________________________________________________

_____________________________________________________________________

_____________________________________________________________________

        setBackground (Color.white);
        setPreferredSize (new Dimension(350, 60));
    }

    // Update the user interface to show the heater's current
    // temperature setting.

    public void refreshUI()
    {
        ____________________________________________("Temperature: " +
                                heater.getTemperature() + "        ");
    }
}
```

Class ButtonListener

```
import java.awt.event.*;
import javax.swing.*;

/**
 * The listener for the two buttons in the heater UI.
 */
public class ButtonListener implements ____________________________
```

```
{
    private HeaterPanel panel;
    private Heater heater;

    /**
     * Initializes this listener with references to the Heater and
     * the panel containing its user interface.
     */
    public ButtonListener(HeaterPanel panel, Heater heater)
    {
        this.panel = panel;
        this.heater = heater;
    }

    /**
     * Increment or decrement the heater's temperature setting,
     * depending on whether the Up or Down button was pushed,
     * then update the UI to reflect the new temperature setting.
     */
    public void actionPerformed(ActionEvent event)
    {
```

```
    }
}
```

Class `Heater` (code from Lab 2)

```
/**
 * A Heater models a simple space-heater. The operations provided
 * by a Heater object are:
 * 1. Increase and decrease the temperature setting by a set
 *    amount. The heater's temperature cannot be raised above
 *    a specified maximum value or lowered below a specified
 *    minimum value.
 * 2. Return the current temperature setting.
 */
public class Heater
{
    // The temperature setting that the heater should maintain.
    private int temperature;

    // The amount by which the temperature setting is
    // raised/lowered when warmer() and cooler() are called.
     private int increment;

    // The minimum temperature setting for the heater.
    private int min;

    // The maximum temperature setting for the heater.
    private int max;

    // The default amount by which the temperature setting is
    // increased when warmer() is called and decreased when
    // cooler() is called.
    private static final int DEFAULT_INCREMENT = 5;

    // The temperature setting for a newly created heater.
    private static final int INITIAL_TEMPERATURE = 15;

    /**
     * Constructs a new Heater with an initial temperature setting
     * of 15 degrees, and which increments and decrements the
     * temperature setting in increments of 5 degrees.
     * Parameters minTemp and maxTemp specify the minimum and
     * maximum temperature settings.
     */
    public Heater(int minTemp, int maxTemp)
```

```
    {
        temperature = INITIAL_TEMPERATURE;
        increment = DEFAULT_INCREMENT;
        min = minTemp;
        max = maxTemp;
    }

    /**
     * Returns this heater's current temperature setting.
     */
    public int getTemperature()
    {
        return temperature;
    }

    /**
     * Increases the heater's temperature setting by the increment
     * amount. Leaves the temperature setting unchanged if this
     * setting would be raised above the maximum temperature
     * setting.
     */
    public void warmer()
    {
        if (temperature + increment > max)
            return;
        temperature += increment;
    }

    /**
     * Lowers the heater's temperature setting by the increment
     * amount. Leaves the temperature setting unchanged if this
     * setting would be lowered below the minimum temperature
     * setting.
     */
    public void cooler()
    {
        if (temperature - increment < min)
            return;
        temperature -= increment;
    }
}
```

API Reference for Selected Classes from the Java Class Library

```
/**
 * class javax.swing.JPanel
 * Instances of this class are lightweight containers.
 */

/**
 * Adds the specified component to this container.
 * Returns the component argument.
 */
public Component add(Component comp)

================================================================================
/**
 * class javax.swing.JLabel
 * An instance of this class is a display area for a short text string.
 */
public JLabel(String text) // Creates a JLabel with the specified text.

/**
 * Changes the text this component will display to the specified string.
 * If the value of text is null or empty string, nothing is displayed.
 */
public void setText(String text)

================================================================================
/**
 * class javax.swing.JButton
 * An implementation of a "push" button.
 */

public JButton(String text) // Creates a button with the specified text.

// Adds the specified ActionListener to the button.
public void addActionListener(ActionListener l)

public String getText()  // Returns the button's text.

================================================================================
/**
 * interface java.awt.event.ActionListener
 *
 * The listener interface for receiving action events. The class that is
 * interested in processing an action event implements this interface, and the
 * object created with that class is registered with a component, using the
 * component's addActionListener method. When the action event occurs, that
 * object's actionPerformed method is invoked.
 */

void actionPerformed(ActionEvent e) // Called when an action event occurs.
```

This API reference continues on the next page.

```
/**
 * class java.awt.event.ActionEvent
 *
 * This event is generated by a component (such as a Button) when a
 * component-specific action occurs (such as being pressed). The event is passed
 * to every ActionListener object that registered to receive such events
 * using the component's addActionListener method.
 */

// Returns the object that generated the event.
public Object getSource()
```

This is the last page of the question paper.