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. Write your student number in the spaces provided in the top-right corner of Pages 2 through 19.
2. Attempt all the questions, and budget your time carefully. Feel free to answer the easiest questions first. Answer Questions 1 through 3 on the lined (right-hand) pages of your answer booklet. Answer Question 4 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 us to mark them**; for example, by drawing a box around the rough work and writing "Mark this" next to the box. When answering questions, you don't need to copy any of the comments or Java code from this question paper to your answer booklet.
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 instance variables.
5. **You can detach the API reference provided at the end of this question paper; however, you cannot take it with you. <u>ALL</u> PAGES OF THIS QUESTION PAPER, INCLUDING THE API REFERENCE, MUST BE TURNED IN. DO NOT REMOVE <u>ANY</u> PAGES FROM THE EXAM ROOM.**

Please read the instructions a second time.

A Java API reference is provided on the last three pages of this question paper.

Question 1 – Defining Classes [10 marks]

A software system for a library has Java classes that model books and patrons (people who have a library card. and are permitted to borrow books from the library).

Here is an incomplete implementation of the `Patron` class.

```
public class Patron
{
    /**
     * Constructor for objects of class Patron.
     */
    public Patron(String name, String cardNumber)
    { /* Code omitted. */ }

    /**
     * Returns true if this Patron's library card number equals the
     * library card number of the Patron referred to by obj;
     * otherwise returns false.
     */
    public boolean equals(Object obj)
    { /* Code omitted. */ }

    /**
     * Returns a string representation of this Patron, listing
     * that person's name and library card number.
     */
    public String toString()
    { /* Code omitted. */ }

    // Methods not required for this question have been omitted.
}
```

Every `Patron` object has two **private** fields (instance variables). Field `name` (of type `String`) is the patron's full name. Field `cardNumber` (of type `int`) is the number on the library card issued to the patron.

Write the complete implementation (field definitions, constructor and methods) of class `Patron`. Don't copy the class' Javadoc comments to your answer booklet.

Question 2 – Grouping Objects [20 marks]

Attempt Question 1 before attempting this question.

Here is an incomplete definition of the class that models library books:

```
public class Book
{
    /**
     * Constructor for objects of class Book.
     */
    public Book(String title, String author)
    { /* Code omitted. */ }

    /**
     * Attempts to loan this book to Patron p.
     */
    public void borrowBook(Patron p)
    { /* Code omitted. */ }

    /**
     * Return this book to the library.
     */
    public void returnBook()
    { /* Code omitted. */ }

    /**
     * Cancels any pending request by Patron p to borrow this book.
     */
    public boolean cancelRequest(Patron p)
    { /* Code omitted. */ }
}
```

(a) Every `Book` object has four **private** fields:

- `title`, the book's title, stored as a character string;
- `author`, the book's author, stored as a character string;
- `borrower`, which stores a reference to the `Patron` that has borrowed the book. If the book is not currently on loan, this field is assigned the symbolic value `null`. (Recall that `null` is equivalent to the `NULL` pointer in C and C++);
- `waitingList`, which refers to a list of patrons who are waiting for the book to be returned to the library, so that they can borrow it. This list is implemented using an `ArrayList`.

Write the Java statements that define these four fields, as they would appear inside the definition of class `Book`. (You must use the field names shown here. Marks will be deducted if you change the names.)

(b) The constructor is passed the book's title and author, and is responsible for initializing the `Book` object's four fields. (A newly created `Book` is not on loan, and no patrons are waiting to borrow it.) Write the constructor. Do not change the names of the two parameters, `title` and `author`. Marks will be deducted if you change these names. Don't copy the constructor's Javadoc comment to your answer booklet.

(c) Method `borrowBook` is called when a patron wants to borrow the book. Parameter `p` refers to the `Patron` that wants to borrow the book. That patron is added to the end of the waiting list if the book is currently on loan; otherwise, the book is loaned to the patron. Write the method. Don't copy the method's Javadoc comment to your answer booklet.

(d) Method `returnBook` is called when a patron returns the book to the library. When the book is returned, it is immediately loaned to the first patron in the waiting list. Write the method. Think carefully about what changes should be made to the waiting list. Remember to handle the case where no one is waiting to borrow the book when it is returned. Don't copy the method's Javadoc comment to your answer booklet.

(e) Method `cancelRequest` is passed a reference to any `Patron` object. If that `Patron` is currently waiting to borrow the book, the method cancels the pending request by removing the patron from the waiting list, and returns `true`. If the `Patron` currently has the book checked out, the method returns `false`. If the `Patron` doesn't have the book and is not waiting to borrow the book, the method returns `false`.

Write the method. Don't copy the method's Javadoc comment to your answer booklet. Hint: the `equals` method in class `Patron` determines if two objects represent the same patron, based on their library card numbers.

Question 3 – Class Inheritance [15 marks]

Here is a `Counter` class based on the one you used in Lab 5:

```
public abstract class Counter
{
    /** The current value of this counter. */
    private int count;

    /** The minimum value of this counter. */
    private int minimumCount;

    /** The maximum value of this counter. */
    private int maximumCount;

    /** The default minimum value of this counter. */
    private static final int DEFAULT_MINIMUM = 0;

    /** The default maximum value of this counter. */
    private static final int DEFAULT_MAXIMUM = 999;

    /**
     * Constructs a new Counter whose current value is initialized to
     * minCount, and which is intended to count from minCount to
     * maxCount, inclusive.
     *
     * @param minCount The minimum value for this counter.
     * @param maxCount The maximum value for this counter.
     */
    public Counter(int minCount, int maxCount)
    {
        minimumCount = minCount;
        maximumCount = maxCount;
        count = minimumCount;
    }

    /**
     * Constructs a new Counter whose current value is initialized to
     * DEFAULT_MINIMUM, and which is intended to count from
     * DEFAULT_MINIMUM to DEFAULT_MAXIMUM, inclusive.
     */
    public Counter()
    {
        this(DEFAULT_MINIMUM, DEFAULT_MAXIMUM);
    }
```

```
    /**
     * Returns the maximum value of this counter.
     */
    public int maximumCount()
    {
        return maximumCount;
    }

    /**
     * Returns the minimum value of this counter.
     */
    public int minimumCount()
    {
        return minimumCount;
    }

    /**
     * Returns this counter's current value.
     */
    public int getCount()
    {
        return count;
    }

    /**
     * Returns true if this counter is at it minimum value;
     * otherwise returns false.
     */
    public boolean isAtMinimum()
    {
         return (count == minimumCount);
    }

    /**
     * Returns true if this counter is at its maximum value;
     * otherwise returns false.
     */
    public boolean isAtMaximum()
    {
        return (count == maximumCount);
    }
```

```
    /**
     * Resets this counter to its minimum value.
     */
    public void reset()
    {
        count = minimumCount;
    }

    /**
     * Increments this counter by 1.
     */
    public void countUp()
    {
        count++;
    }

    /**
     * Decrements this counter by 1.
     */
    public void countDown()
    {
        count--;
    }
}
```

One of things biologists study is *cell division*, which is the process by which living organisms are constructed and repaired. Each time cell division occurs, the total number of cells doubles: 1 cell divides into 2 cells, then the 2 cells both divide, for a total of 4 cells, then the 4 cells each divide, for a total of 8 cells, and so on. When a cell dies, the total number of cells decreases by 1 (the death of one cell does not affect other cells).

A *cell division counter* mimics the cell division process. This counter always starts at 1, and incrementing the counter doubles its value. Decrementing the counter reduces its value by 1. The counter has a maximum value of 2147483647; that is, $2^{31} - 1$, which is the largest value of Java's `int` type. The counter has a minimum value of 1.

You are going to write a `CellDivisionCounter` class, which must be a subclass of `Counter`. The class has one constructor. Here is the specification:

```
    /**
     * Constructs a new CellDivisionCounter whose current count is
     * initialized to 1.
     */
    public CellDivisionCounter();
```

This question continues on the next page.

You should be able to call several methods on any `CellDivisionCounter` object. Here are their specifications:

```
/**
 * Doubles this counter's current value. When the count reaches the
 * counter's maximum value, it should remain at that value until
 * countDown or reset is called.
 */
public void countUp();

/**
 * Decrements this counter by 1. When the count reaches the
 * counter's minimum value, it should remain at that value until
 * countUp is called.
 */
public void countDown();

/**
 * Returns the maximum value of this counter.
 */
public int maximumCount();

/**
 * Returns the minimum value of this counter.
 */
public int minimumCount();

/**
 * Returns this counter's current value.
 */
public int getCount();

/**
 * Returns true if this counter is at it minimum value;
 * otherwise returns false.
 */
public boolean isAtMinimum();

/**
 * Returns true if this counter is at its maximum value;
 * otherwise returns false.
 */
public boolean isAtMaximum();
```

```
/**
 * Resets this counter to its minimum value.
 */
public void reset();
```

Write the complete implementation of `CellDivisionCounter`. You must decide which methods should be defined in `CellDivisionCounter`, and which methods can simply be inherited from `Counter` and do not need to be overridden in `CellDivisionCounter`.

Marks will be deducted if you define any fields (instance variables) or methods that aren't necessary; however, you can define local variables in the constructor and methods.

Assume that these two constants have been defined (you don't need to copy these to your class definition):

```
public static final int MIN_COUNT = 1;

public static final int MAX_COUNT = 2147483647;
```

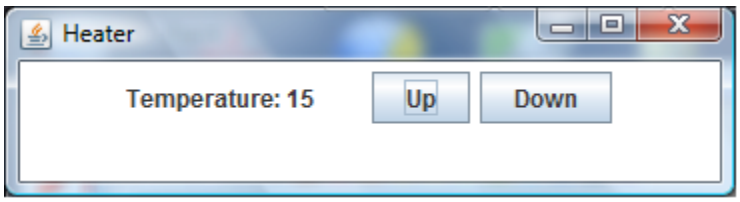
You are not permitted to make any changes to the `Counter` class presented earlier.

Here are some examples of how a `CellDivisionCounter` counts:

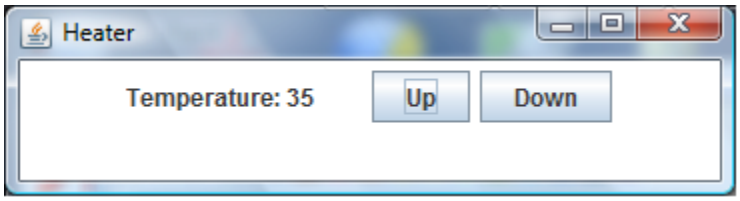
```
CellDivisionCounter counter = new CellDivisionCounter();
// The count is currently 1
counter.countUp();    // count is now 2
counter.countUp();    // count is now 4
counter.countUp();    // count is now 8
counter.countDown();  // count is now 7
counter.countUp();    // count is now 14
counter.reset();      // count is now 1

CellDivisionCounter2 counter = new CellDivisionCounter();
// The count is currently 1
counter2.countUp();    // count is now 2
counter2.countDown();  // count is now 1
counter2.countDown();  // count remains 1
```

Question 4 - 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 java.util.ArrayList<E>
 */

// Constructs an empty ArrayList that stores references to objects of type E.
public ArrayList<E>()

// Appends o (of type E) to the end of this list. Returns true if
// successful, otherwise returns false.
boolean add(E o);

// Removes all the objects from the list. The list will be empty when
// this method returns.
void clear();

// Returns true if this list has at least one occurrence of an object that is equal
// to o (of type E), as determined by the equals method; otherwise returns false.
boolean contains(E o);

// Returns the object (of type E) at the specified position (index)
// in the list. Parameter index must be >= 0 and < size().
// The object is not removed from the list.
E get(int index);

// Returns true if this list has no objects, otherwise returns false.
boolean isEmpty();

// Removes the object (of type E) at the specified position (index)
// in the list. index must be >= 0 and < size(). Shifts any subsequent
// elements to the left (subtracts one from their indices).
// Returns the object that was removed from the list.
E remove(int index);

// Removes the first occurrence of an object that is equal to o (of type E), as
// determined by the equals method. Shifts any subsequent elements to the left
// (subtracts one from their indices).
// Returns true if an object was removed from the list; otherwise returns false.
boolean remove(E o)

// Replaces the object (of type E) at the specified position (index) in
// this list with the specified element. index must be >=0 and < size().
// Returns the object that was previously stored at the specified position.
E set(int index, E element);

// Returns the number of objects stored in this list.
int size();
```

This API reference continues on the next page.

```
/**
 * 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.