In this assignment you will learn more about objects by turning the program you wrote for Homework #4 into a reusable software component. Along the way, you will learn about programming graphical user interfaces and applets.

**Part 1.** In Homework #4, you wrote a class that simulated a cellular automaton. The automaton had a rule governing its evolution that we represented as follows:

```
   Rule 126
```

There are 256 distinct rules for cellular automata that take into account the value of a cell and its immediate left and right neighbors: one rule for each distinct combination of boolean values assigned to the eight possible arrangements above. Rule 126 displays a regular nested self-similar structure; other rules display other types of structures, some simpler and some more complex.

In this assignment, you will revise your `Automaton` class so that it relies upon an interface `Rule`. The specification of `Rule` is as follows:

```java
public interface Rule {
    public boolean rule(boolean lneighbor, boolean cell, boolean rneighbor);
}
```

You will alter the constructor of `Automaton` so that it takes a parameter of type `Rule` as well as a parameter of type `boolean`. The `Rule` object should be kept as a field of the `Automaton` class and used by the `generation` method to compute the state of each cell for the next generation. You will no longer need the method `rule` inside the class `Automaton`.

You will define four classes that implement the `Rule` interface: `Rule126`, `Rule30`, `Rule110` and `Rule 250`. `Rule126` will use the rule you have already written, as described above. The other three classes will use rules 30, 110 and 250 as below:

```
   Rule 30
```

```
   Rule 110
```

```
   Rule 250
```
Each Rule class should override toString to return the name of the rule, e.g., "Rule 30".
You should add a command-line argument to the Automaton class "--rule=n" which allows the user to select from among the automata above. 30, 110, 126 and 250 will be the only valid values of n, so you may implement the argument by checking for the strings "--rule=30", "--rule=110", "--rule=126" and "--rule=250", respectively. Rule 126 should remain the default.
The Automaton class should still operate from the command-line as it did in the last assignment. Use the Input class to read user input; advance a generation on Return and quit on “q”. Sample output for each of the rules is shown in Figure 1.

Part 2. We are now going to embed the Automaton program in an applet. In order to do this effectively, you will need to add a few new methods to the Automaton class:

- **reset** - this method should set the generation counter back to 0 and reinitialize the line of cells according to the current settings (randomly, or with one live cell).

- **setRule** - this method should take one Rule parameter and make it the active rule for the automaton. Changing the rule should result in the automaton being reset.

- **setRandom** - this method should take one boolean parameter indicating whether cell initialization should be randomized. Changing the randomization setting should result in the automaton being reset.

Figure 2 shows the outline for the full Automaton class.

Once you have defined these methods, you should integrate your Automaton class with the class AutomatonApplet, available from the course directory on CUNIX. The applet has four GUI components that you must write the code for:

- **comboBox** - This JComboBox contains a list of all of the available automaton rules. On a change to the selection in the list, the automaton should be updated with the new rule and then reset.

- **random** - This JCheckBox should be checked to select random initialization and unchecked to select default initialization. On a change to the state of the checkbox, the automaton should be updated with the new initialization preference and then reset.

- **gen** - This JButton will advance the automaton one generation when the user clicks on it and display the new state of the automaton in the JTextArea textArea. If the text exceeds the size of the text area, the view should be scrolled to the bottom of the text so that the newest generation is visible.

- **reset** - This JButton will reset the automaton without changing its settings. The text area should be cleared of all previous output and the initial generation of the reset automaton should be displayed.
% java Automaton

    X              0
    XXX            1
    XX XX          2
    XXXXXXX        3
    XX XX          4
    XXXX XXXX      5
    XX XX XX XX    6
    XXXXXXXXXXXXXX 7
    XX XX          8q

% java Automaton --rule=30

    X              0
    XXX            1
    XX X           2
    XX XXXX        3
    XX X X         4
    XX XXXX XXXX   5
    XX X X X X     6
    XX XXXX XXXXXX 7
    XX X XXX X     8q

% java Automaton --rule=110

    X              0
    XX             1
    XXX            2
    XX X           3
    XXXX           4
    XX X           5
    XXX XX         6
    XX X XXX       7
    XXXXXXX X      8q

% java Automaton --rule=250

    X              0
    X X            1
    X X X          2
    X X X X        3
    X X X X X      4
    X X X X X X    5
    X X X X X X X  6
    X X X X X X X X 7
    X X X X X X X X X 8q

Figure 1: Typical output from the Automaton program.
public class Automaton {
    /* Fields and constant definitions */
    ...

    /* Create an Automaton object with the given rule. */
    public Automaton(Rule rule, boolean random) {
        ...
    }

    /* Re-initialize the line of cells and revert to generation 0. */
    public void reset() {
        ...
    }

    /* Change the automaton’s rule. */
    public void setRule(Rule rule) {
        ...
    }

    /* Change the automaton’s initialization method. */
    public void setRandom(boolean random) {
        ...
    }

    /* Produce a String representation of the cell state. */
    public String toString() {
        ...
    }

    /* Advance a generation. */
    public void generation() {
        ...
    }

    /* Command-line version of the Automaton program. */
    public static void main(String[] args) {
        ...
    }
}

Figure 2: The Automaton class.
You can run the applet by loading the file `auto.html` (also available from the course directory on CUNIX) in your web browser. Unfortunately, most web browsers cannot reload an applet if you change it; you must shut down the browser, restart and then load the file again. The Java `appletviewer` program allows the user to stop, start and reload applets without quitting and restarting the program. I recommend that you use `appletviewer` as you develop your applet, and test it in your web browser of choice before submitting it. You can download a screenshot of the applet in use from the course website. (The applet may not look exactly like the screenshot on your computer. Java attempts to use “native” buttons and GUI controls whenever possible.)

The `JTextArea` class ensures that the current cursor position is always within the viewable area. To force the text area to scroll to a particular position, you can set the cursor position using the `setCaretPosition` method, inherited from `javax.swing.text.JTextComponent`. `setCaretPosition` takes a single `int` parameter specifying a position in the text. You can determine the last position in the text by calling the `String` method `length` on the text of the component. To force the cursor to the end of the text area, you can use:

```java
textarea.setCaretPosition( textarea.getText().length() );
```

Submit `Automaton.java`, `Rule.java`, `Rule30.java`, `Rule110.java`, `Rule126.java`, `Rule250.java`, `AutomatonApplet.java`, `auto.html` and a README file briefly explaining the design of your program.

**Extra Credit 1 (4 points).** The rules for the cellular automata we’ve been discussing are numbered by interpreting their description as binary code. Each case with a live cell is read as a 1 and each case with a dead cell is read as a 0, as follows:

<table>
<thead>
<tr>
<th>Rule 30</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 1 1 1 1 0</td>
</tr>
<tr>
<td>0001 1110₂ = 30</td>
</tr>
</tbody>
</table>

Instead of making `Rule` an interface, make it a class with one constructor. The constructor should take a single `int` parameter between 0 and 255, representing an automaton rule using the encoding above. If the parameter is outside of the interval [0,255], you should truncate it in a manner of your choosing.

Your `Rule` class should have the same methods as the rule classes above: `rule` and `toString`. Update the “`--random=n`” command-line argument to accept any `n` between 0 and 255. You will also have to change the `comboBox` component to use your new version of `Rule`. Allow the user to select any of the 256 automata using `comboBox`.

**Extra Credit 2 (4 points).** Modify the `Automaton` class so that it can produce a graphical representation of the line of cells, perhaps a row of black and white squares. Use this new capability to paint the generations of the automata onto a `JPanel` rather than
printing them in a text area. Besides the visual format of the automaton, the applet should behave in exactly the same way.

Submit both the textual AutomatonApplet and your separate graphical applet. They should both use the same Automaton class.