Assignment 7: Mutable Worlds

Goals: Further practice with ArrayLists; mutable worlds; refactoring in the face of mutation.

Instructions

This assignment is long. Start early.

As always, be very careful with your naming conventions.

The submissions will be organized as follows:

- **Homework 7 Problem 1:** The Automata.java file.
- **Homework 7 Problem 2:** The rest of your Centipede game

Due Dates:

- Problem 1 – Thursday, March 12th at 9:00pm
- Problem 2 – Tuesday, March 24th at 9:00pm

Problem 1: Cellular Automata and Mutable World Programs

A *cellular automaton* is a simplified model of a biological cell: it has a *state*, a short lifespan, and can produce a child cell in some state in response to its own state and to the state of its neighbors. When a collection of cellular automata are arranged in some fashion (in a line, or on a plane, or in a 3-d grid...), interesting collective behaviors can emerge. In this problem, we're going to work with the simplest kind of cellular automaton, which has only two states (on or off), and we're only going to work with a 1-dimensional collection of cells. As it turns out, even this simplified setting is enough for impressive patterns to emerge.

Before you begin the assignment below, add the following lines to the top of your Automata.java file:

```
import tester.*;                // The tester library
import javalib.worldimages.*;   // images, like RectangleImage or OverlayImages
import javalib.impworld.*;      // the abstract World class and the big-bang library for imperative worlds
import java.awt.Color;          // general colors (as triples of red,green,blue values)
                                // and predefined colors (Red, Green, Yellow, Blue, Black, White)
```

Be sure to use impworld, not funworld.

7.1 Individual cells

An individual cell should implement the following interface:

```
interface ICell {
  // gets the state of this ICell
  int getState();

  // render this ICell as an image of a rectangle with this width and height
  WorldImage render(int width, int height);

  // produces the child cell of this ICell with the given left and right neighbors
  ICell childCell(ICell left, ICell right);
}
```

Every cell has a state, but it might be computed in various ways, so `ICell`s provide a `getState` method to compute it. We make the return type `int` in case you want to experiment with more sophisticated cell states than just on/off. For now, use `0` for an "off" cell, and `1` for an "on" cell.

Additionally, every cell needs to produce a picture of itself somehow. The `render` method should produce a picture that fits into a rectangle of the given width and height. You will use this method to render a collection of cells at various locations. You may draw cells however you like; in the pictures below, an "on" cell is drawn as a black square, while an "off" cell is white.

Finally, the `childCell` method lets a cell produce its child cell, in response to its own state and its neighbors' states. We'll describe this method in more detail, below.

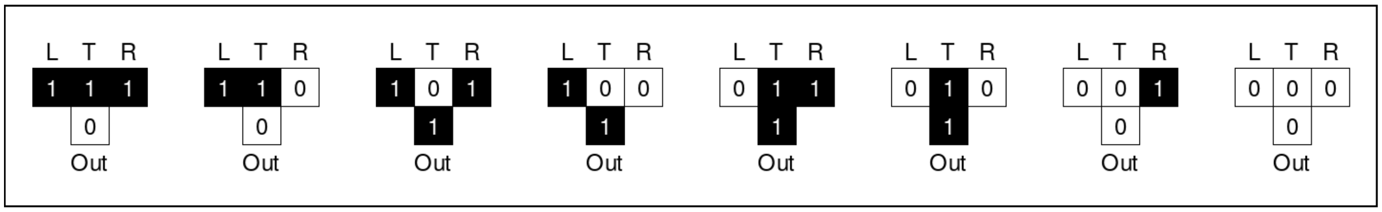
7.2 Different kinds of cells

7.2.1 Inert cells

Define a class `InertCell` that implements `ICell`, and is always off. Its state is always `0`. When it produces a child, it ignores its neighbors entirely and produces another `InertCell`.

7.2.2 Rule-based cells

A rule-based cell takes the states of its left and right neighbors, along with its own state, into account when producing a child cell. Since each of these three cells can be in one of two states, there are eight possible situations. For each of these eight possible situations, the child cell could be either on or off. Accordingly, there are $2^8 = 256$ possible "rules" for implementing a rule-based cell. We can conveniently name these rules by reading off the outputs as bits, in order from left to right, and treating them as an 8-bit binary number. For example, here is rule 60:

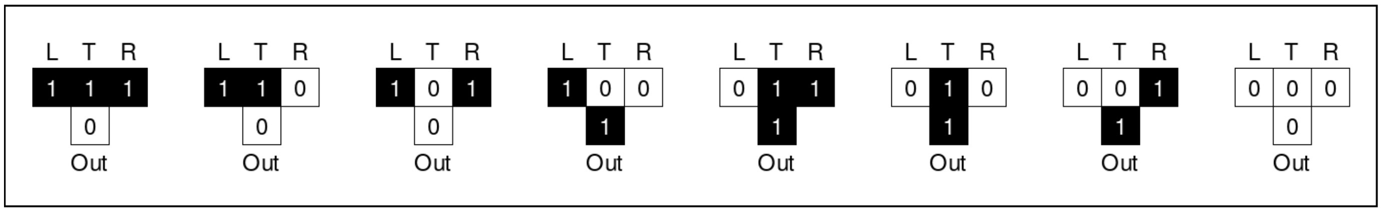


Read each of these images as saying (for example): "if the *left* cell is in state `1`, *this* cell is in state `0`, and the *right* cell is in state `1`, then output a child cell in state `1`."

(We get the number 60 by reading the outputs as bits from left to right, so $0 * 2^7 + 0 * 2^6 + 1 * 2^5 + 1 * 2^4 + 1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 0 * 2^0 = 60$. We get the exponents by reading off the *inputs* as bits of a binary number, too.)

Design a class `Rule60` that `implements ICell`, and implements the rule above.

Design a class `Rule30` that `implements ICell`, and implements the rule below:



(Again, we get the number 30 by reading the outputs as bits from left to right, so $0 * 2^7 + 0 * 2^6 + 0 * 2^5 + 1 * 2^4 + 1 * 2^3 + 1 * 2^2 + 1 * 2^1 + 0 * 2^0 = 30$.)

Abstract as much as you can. Implementing new classes, such as `Rule182` or `Rule54`, should be much easier than copy-paste-modifying your existing code! Instead, look carefully for repeated code in these rule-based classes, and abstract it into an abstract base class (that still `implements ICell`). Then, simplify your concrete classes as much as possible. You may find this easier to do once you've finished the rest of the assignment and have it working, and can confirm (via tests!) that you won't break anything.

7.3 Populations of cells

A population of cells will be a `CellArray`, which takes in just an `ArrayList<ICell>` in its constructor. Design a method `CellArray nextGen()` that produces a new generation of cells from the current population. Cells at the start and end of the list are treated as having `InertCell`s as left and right neighbors, respectively.

It should also have a `draw` method, which takes in two numbers, representing the width and the height of an indiviual cell.

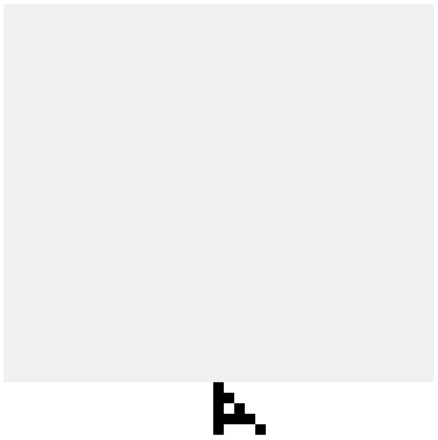
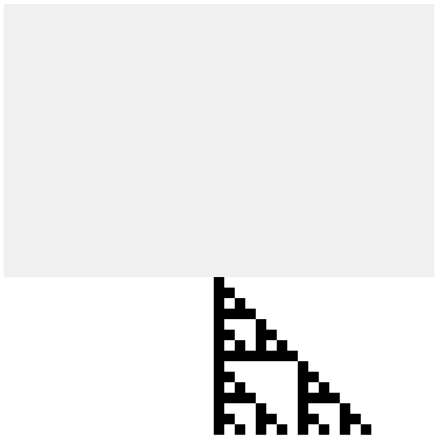
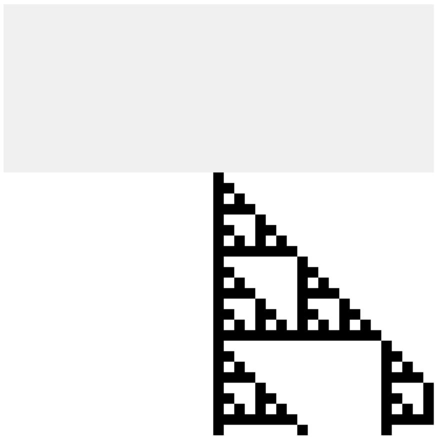
7.4 Animating your world

This code introduces two new keywords, `static` and `final`. For now, treat them as "magic words" and use these definitions as given. A partial explanation for them is that `static` lets you define things associated with a class rather than with an object, and `final` prohibits you from modifying a value once it's been initialized. Taken together, and when used properly, they allow you to define constants like these, and the naming convention is to use `ALL_CAPS_AND_UNDERSCORES`.

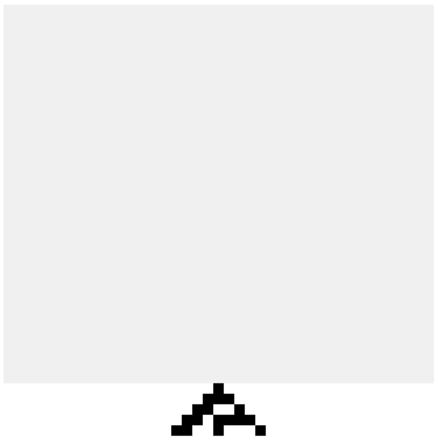
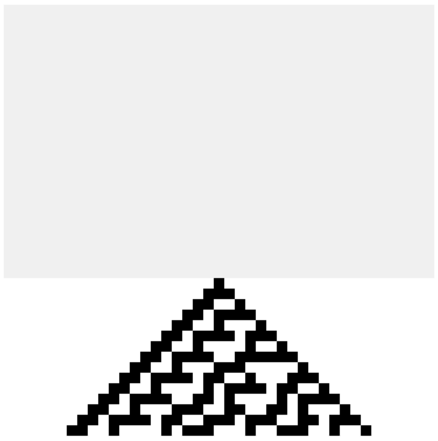
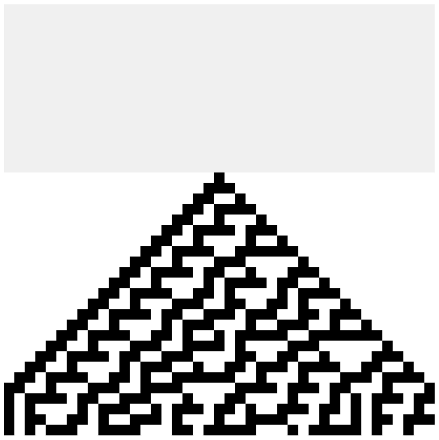
Design a class `CAWorld` that `extends World`. Complete the class definition below:

```
class CAWorld extends World {

  // constants
  static final int CELL_WIDTH = 10;
  static final int CELL_HEIGHT = 10;
  static final int INITIAL_OFF_CELLS = 20;
  static final int TOTAL_CELLS = INITIAL_OFF_CELLS * 2 + 1;
  static final int NUM_HISTORY = 41;
  static final int TOTAL_WIDTH = TOTAL_CELLS * CELL_WIDTH;
  static final int TOTAL_HEIGHT = NUM_HISTORY * CELL_HEIGHT;
  // the current generation of cells
  CellArray curGen;
  // the history of previous generations (earliest state at the start of the list)
  ArrayList<CellArray> history;

  // Constructs a CAWorld with INITIAL_OFF_CELLS of off cells on the left,
  // then one on cell, then INITIAL_OFF_CELLS of off cells on the right
  CAWorld(ICell off, ICell on) {
    //TODO: Fill in
  }

  // Modifies this CAWorld by adding the current generation to the history
  // and setting the current generation to the next one
  public void onTick() {
    //TODO: Fill in
  }

  // Draws the current world, ``scrolling up'' from the bottom of the image
  public WorldImage makeImage() {
    // make a light-gray background image big enough to hold 41 generations of 41 cells each
    WorldImage bg = new RectangleImage(TOTAL_WIDTH, TOTAL_HEIGHT,
        OutlineMode.SOLID, new Color(240, 240, 240));

    // build up the image containing the past and current cells
    WorldImage cells = new EmptyImage();
    for (CellArray array : this.history) {
      cells = new AboveImage(cells, array.draw(CELL_WIDTH, CELL_HEIGHT));
    }
    cells = new AboveImage(cells, this.curGen.draw(CELL_WIDTH, CELL_HEIGHT));

    // draw all the cells onto the background
    return new OverlayOffsetAlign(AlignModeX.CENTER, AlignModeY.BOTTOM,
        cells, 0, 0, bg);
  }

  public WorldScene makeScene() {
    WorldScene canvas = new WorldScene(TOTAL_WIDTH, TOTAL_HEIGHT);
    canvas.placeImageXY(this.makeImage(), TOTAL_WIDTH / 2, TOTAL_HEIGHT / 2);
    return canvas;
  }
}
```

The 5th, 15th and 25th generations for Rule 60 should look like this (shrunk to fit side-by-side here):

Likewise, the 5th, 15th and 25th generations for Rule 30 should look like this:

7.5 Optional Enhancements

Try designing a three-valued cell (states are 0, 1, 2), or a four-valued cell. Make up coloring rules for them, and make up rules for how the `childCell` method should work.

Try designing differently-sized neighborhoods, so that cells can see their four nearest neighbors (two on each side).

Problem 2: Centipede, Part 3

Goals: Adapt and enhance your existing game to meet new requirements

7.1 Instructions

As of Assignment 5, your games were now playable: the centipedes could move, the gnomes could fire darts, and the game could play to completion. In this assignment, you will add new features to the game.

Note: you are permitted to use the `impworld` library for this assignment, if you'd like to use mutation in your designs.

Note: you are *strongly encouraged* to use generics if and where you think they will help. (In particular, you are welcome to use `IList<T>` or `ArrayList<T>` instead of `ILoWhatever`, and to use function objects or appropriate loops wherever they may help.)

7.1.1 Submission format

You may find it overwhelming to keep all of your code within one file. You are permitted to split your code into multiple files, if it helps. Do *not* use any subdirectories or packages, as that will make things needlessly confusing. If you are more comfortable submitting all your code in a single file, though, that is also fine.

You will need to submit (1) all your code, (2) a screenshot of your game after it's been playing for a little while, and (3) a `README.txt` file. To submit these things together, you'll need to zip them together:

- On Windows, open Explorer, and select all the files you want to submit. Right click, and select "Send to > Compressed (Zipped) folter".
- On Mac, open Finder, and select all the files you want to submit. Right click, and select "Compress items"
- On Linux, open Nautilus, and select all the files you want to submit. Right click, and select "Compress..." Enter a filename, select the ".zip" option, and press Enter

Your `README.txt` file must be a plain text file — don't submit a Word document, or a Pages, or anything fancy! — and must include:

- A section summarizing what changes you made based on feedback on your previous Centipede submission. This section should mention which data structures, methods, or entire classes were changed, and why, and where in your code graders should look to find the updates.

- A paragraph explaining whether you chose to continue using `funworld`, or whether you switched to `impworld`, and why
- A section explaining the high-level design choices you made, what the main classes are in your design and what they are for.
- A section explaining your overall approach to testing, and what scenarios were most challenging to test for and why
- A section explaining what you would do differently, had you known about all the parts of the assignment from the beginning (rather than them being revealed over time).

7.2 Requirements

7.2.1 First, fix feedback

If you received comments on your design for part 2, you must address those concerns in part 3. We will be looking for these improvements; you can be marked down for not fixing them in part 3 even if you lost points for them in part 2.

Be sure to summarize and explain these changes in your `README.txt` file, as described above.

7.2.2 Summary of Game-play Changes

- The gnome may now move up or down, somewhat.
- The game is no longer over when all segments of the centipede are destroyed, or when they get stuck at the bottom corners of the playing area.
- Pebble piles now do something, and dandelions have some new behavior.
- Centipedes no longer only move down.
- Darts are no longer the only weapon.

7.2.3 The bottom of the screen

The centipedes no longer get stuck in the corners of the playing area. Instead, they turn upward, and continue their zig-zag motion across the board. When they get back to the top of the screen, they should turn downward and continue zig-zagging just as they began. (*Hint:* this should have some consequences for your various collision behaviors, as well.)

In order for the game not to trivially be over when a centipede segment makes it to the bottom row of the screen, the gnome must have the ability to move up or down and out of the way. The gnome is restricted to the bottom three rows of the playing area, and cannot walk through any dandelions (pebbles aren’t a problem, though).

The game still ends when the gnome is hit by a segment of a centipede.

7.2.4 New levels

When the player destroys the last segment of the centipede, a new level begins. You should create a new centipede, of length 1 segment longer than the current level's starting length, and with a speed slightly faster than before. (If the player survives enough levels, it should be possible for the centipedes to become faster than the gnome.) The overall state of the garden should be maintained between each level: any dandelions or pebble piles from before are preserved, and are new obstacles for the new centipede to avoid.

7.2.5 Obstacles

As before, centipedes are blocked by dandelions, and must change rows and reverse direction just as last time. However, dandelions now have a "health" of sorts: hitting a dandelion with a dart three times will kill the dandelion and turn it into a pile of pebbles. Your graphics must visually indicate the health of each dandelion (e.g. via color, or the number of petals you draw, or something similar).

Pebble piles influence the *speed* of centipedes. Each pebble pile is actually bigger than one cell, and spills over into its four directly adjacent neighbors (up/down/left/right). If *any* segment of a centipede is on *any* of the pebble pile's cells, then the speed of that centipede is *halved*: pebbles are quite large obstacles for a centipede to crawl over, after all!

7.2.6 New weapon: water balloons

In addition to the darts, your gnomes now have the ability to throw water balloons. Once the player has hit any three targets in a row, *without missing*, they may press the 'b' key to throw a water balloon. A water balloon travels in a straight line just like a dart; the difference is when it hits a target. If it collides with anything, the balloon bursts, and soaks the collision target and everything in any of the eight adjacent cells (up/down/left/right and the four diagonally adjacent cells). Any centipede segments in those nine cells are immediately hit and turn into dandelions. Any dandelions that are caught in the eight cells of the splash zone are now watered, and come back to full health (see Obstacles above). However, if a dandelion was the direct hit of a water balloon, it drowns and turns into a pebble pile.

Your drawing of water balloons must be visually distinct from your drawing of darts. The gnome's progress towards attaining a water balloon should be indicated on the screen; once available, the gnome should be drawn to reflect that fact, as well.

Once a player uses their water balloon, they must hit another three targets in without missing before they get another balloon. If they miss a target, the count resets and they must hit another three targets without missing. Obviously, centipede segments and dandelions count as targets; pebble piles do not, though, and a balloon can fly right over them.

7.2.7 Keeping score

Every dart you fire that hits a centipede segment earns 10 points. Every dart you fire that misses everything on the board deducts 1 point from your score. Darts that hit dandelions do not change your score. Water balloons deducts 5 points just to throw (but can potentially soak multiple segments, so they still can come out ahead). Your score should be maintained across levels of the game, and must be visible somewhere on screen at all times.

When the game is over, you should include the final score in your final message to the player.

7.3 Testing

At this point, your Examples test class is probably quite difficult to work with: you likely have dozens of fields declared at the top of the class, that you initialize once, and then refer to, several hundred lines further down in the file. This is difficult to maintain and hard for anyone else to read and understand.

Refactor your code so that instead of those fields, you declare and initialize your example data in the test methods that need it. If you find yourself reusing the same examples over and over, abstract them out into a helper method — and in the purpose statement for that helper, explain what kind of example data you're building.

Graders should be able to read each test method in isolation and easily figure out what example data it's using.