American University of Armenia
CS 120 Intro to OOP
Spring 2019

Homework Assignment 6, Part 1

In this part of the homework you will update how pattern data is stored and you will extend your Game of Life to support going back through generations as well as forward (an 'undo' if you like). This will require you to copy your `World` objects, which you'll do both by copy construction and by Java's cloning (normally you wouldn't do both of course, but this is an educational activity!).

1. **(10 points)** The current implementation of `PatternStore` relies on a partially filled array of `Patterns` that wastes a lot of memory if the actual number of patterns is much less than 1000. Your first task is to fix this by switching from the partially filled array to an `ArrayList`.

   Replace the state of the `PatternStore` class

   ```
   public static final int MAX_NUMBER_PATTERNS = 1000;

   private Pattern[] patterns = new Pattern[MAX_NUMBER_PATTERNS];
   private int numberUsed = 0;
   ```

   with the single instance variable

   ```
   private ArrayList<Pattern> patterns = new ArrayList<>();
   ```

   Modify your `load` method to read the data into this new structure. As before, if a line in the file is malformed you should print out the offending line as a warning, but just continue (i.e. not kill the program).

   Your next task is to give `Pattern` an in-built/default/natural ordering, which should be by pattern name (only). You can assume that all patterns have a unique name.

   To give a natural ordering we apply the `Comparable` interface. Your `Pattern` class therefore needs to start:

   ```
   public class Pattern implements Comparable<Pattern> {
   ```

   which will require you to add a new method to `Pattern`:

   ```
   @Override
   public int compareTo(Pattern o) {
      // TODO
   }
   ```

   Implement `compareTo` such that it sorts by pattern name (only). Note `String` implements `Comparable<String>`.

   Remove the `getPatterns` method from your `PatternStore` class and implement the following method instead:

   ```
   public ArrayList<Pattern> getPatternsNameSorted() {
     // TODO: Get a list of all patterns sorted by name
   }
   ```

In implementing this, note that to sort an `ArrayList` using the natural ordering of its contents you can use `Collections.sort(mylist);` and that you must not return a reference to the internal list. To copy a list `mylist` you can use something like `ArrayList<Pattern> copy = new ArrayList<Pattern>(mylist)`. Doing a shallow copy is OK in this case since your `Pattern` class should be immutable.

Finally, make the corresponding changes in the `play` method of the `GameOfLife` class so that it uses the new `getPatternsNameSorted` method instead of the old `getPatterns`. Test your program.

2. **(10 points)** It turns out there can be multiple starting patterns that give a particular pattern, so you can't auto-generate the previous state from the current state. Instead, we need to save the state of the current world each time we move forward, i.e. copy our `World` objects.

   `GameOfLife` will need to contain a new structure to hold the saved `Worlds`. The functionality required is:

   - Entering "b" and pressing enter should move you back one generation unless we are already at generation 0, in which case you should just print the 0th generation board.
   - Entering "f" and pressing enter should take you forward one generation. However, if we have previously computed that generation, it should not be recomputed: rather the saved `World` should be reinstated, i.e. you *cache* every computation so you never repeat yourself.

   Add an `ArrayList<World>` variable `cachedWorlds` to `GameOfLife`.

   Add this method to `GameOfLife`:

```java
private World copyWorld(boolean useCloning) {
  // TODO later
  return null;
}
```

   Adapt `play()` to have the functionality described above. When you need to copy the world, put a call to `copyWorld(false)` for now. Note that `world` should be retained and should *always* reference the current object of `World` (i.e. the one being displayed). At any given time, `cachedWorlds` should contain every `World` that has been computed so far. This means that `world` will always point to a `World` object that is also referenced from within `cachedWorlds`. Note that you should *not* add any further state to `GameOfLife`: everything you need should be available from `world` or `cachedWorlds`.

3. **(10 points)** As you saw in lectures, a copy constructor is a constructor that takes an instance of the same class as an argument and matches the state of the new instance to the supplied one.

   Add suitable copy constructors to `World`, `PackedWorld` and `ArrayWorld`. Remember that you will need to deep copy some state and that `Pattern` is an immutable class.

   You should test your copy constructors by constructing a `PackedWorld` or `ArrayWorld`, making a copy, changing that copy (e.g. using `setCell`) and checking that the original remains unchanged.

   Your `GameOfLife` class stores references to `Worlds` and not to the concrete types (`ArrayWorld`, etc). This means your code has the problem discussed in lectures: you need to know the true type of an object in order to write the appropriate `new` statement. Here we will need to write something like:

```java
World copy = new ??????(world);
```

but we don't know whether to write PackedWorld or ArrayWorld (or any other world representation we might add in future) to get the correct constructor. There isn't a 'pure' OOP solution to this, although you can get round it in Java using reflection (the ability to query objects about their class information). We'll do it here, but only to emphasise it's not pretty!

Fill in copyWorld to return a reference to a copy of the world object created using the copy constructor you have written when useCloning is false. You may need to check the true type of world using instanceof or similar. Note the instanceof syntax is A instanceof B, and it returns true if A is an instance of B.

It's important to realise your solution is bad for multiple reasons: firstly, it's ugly. It's taking up a lot of unnecessary lines. Secondly, it's not extensible. Every time you create a new type of World (e.g. StringWorld), you'll need to hand-edit this chunk of code so that it can work. It would be easy to forget to do that.

An (optional) better approach

We can use reflection in a better way to make our code work with *any* concrete extension of World. The trick is that the associated Class object (available via getClass()) can provide you with a Constructor object representing the correct constructor, on which you can then call a method newinstance(...) to get a copy. This is certainly an improvement, but it's still rather clunky and generally frowned upon (resorting to reflection should always be a last resort!).

However, if you are keen for a challenge, have a go at implementing this approach in copyWorld. **This is entirely optional, not necessary for your homework.**

4. **(10 points)** The neater solution in this case is to use Java's cloning. Your next task is to make ArrayWorld and PackedWorld (and therefore World) Cloneable. Here is the recipe from lectures:

   (a) Implement the Cloneable interface

   (b) Override the clone() method from Object, making it public

   (c) Call super.clone()

   (d) Perform any deep copying/cloning as needed

   Make all of your World objects cloneable. You can copy much of the code from your copy constructors, **but** be careful not to waste effort copying state that does not need to be explicitly copied (i.e. the 'shallow' state).

   Update copyWorld() to use cloning when useCloning is true.

   Set your play code to use copyWorld(true) (i.e. use cloning).

5. **(10 points)** If we generate many generations and/or have large worlds, our new history functionality will need a lot of memory. Anything we can do to reduce this is helpful, and a key observation is that knowing where the live cells are implies where the dead cells are, so storing the entire board state is *not* necessary. One optimisation is to use a *sparse array*, which would represent *only* the live cells. However, while these are more space-efficient, they add an unwanted computational cost when working out the state of a particular cell.

   Instead you will improve your ArrayWorld by removing the rows containing entirely dead cells (you will have noticed most patterns have a lot of them so this can be a significant saving).

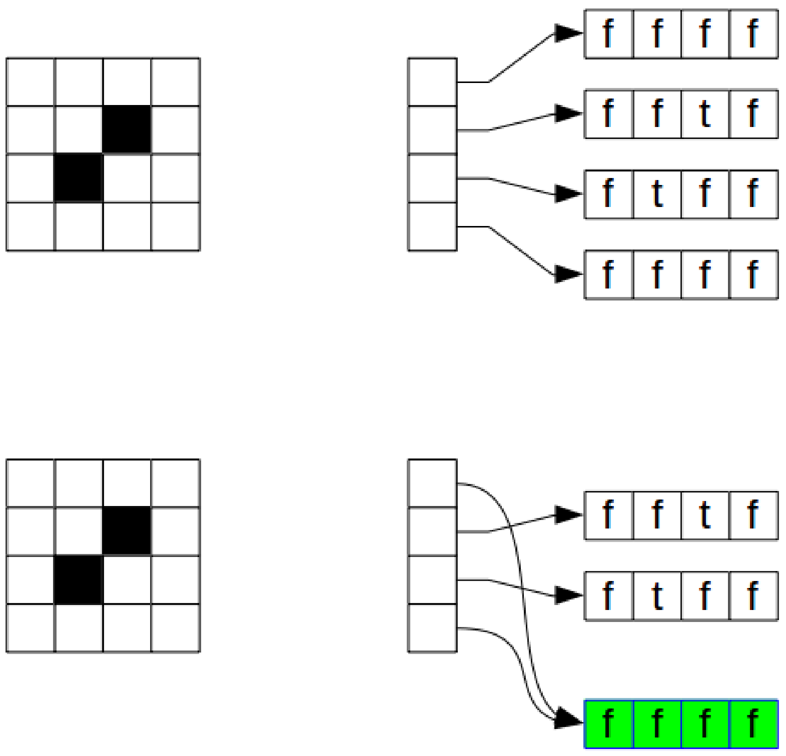

Figure 1: Reducing redundancy in `ArrayWorld`

The trick is to store a single row of dead cells (i.e. a `boolean[]` containing only `false`) and use that one array for *every* world row that is blank. For example, see Figure 1.

The top part shows the original approach, while the bottom shows the new approach. Here it only achieves a saving of one row, but across multiple generations and bigger worlds, the saving is more significant.

Add a new `boolean[]` member to `ArrayWorld` with the name `deadRow`. Initialise it to have the correct row width in the constructors.

Adapt your `ArrayWorld` constructors to set dead rows to point to `deadRow` following an initialisation from a string or pattern. The copy constructor should copy the `deadRow` reference so all copies use the same chunk of memory for it.

Adapt your `clone()` method so that any clone has all dead rows set to point to `deadRow`.

It would be nice to make the contents of `deadRow` immutable to prevent accidental edits. Unfortunately we can't do this. We *could* declare `deadRow` to be `final`: you may wish to experiment to find out what effect this has on an array. Explain your findings in a brief 1–2 paragraph report.

It may be tempting to make `deadRow` static, but in fact that's not necessary and problematic if we ever want to run multiple simultaneous `GameOfLifes` (which could have different dimensions and hence need different `deadRow` lengths). Here you should only ever construct a new `World` once per pattern selected. Thereafter you generate them by cloning, which will copy the `deadRow` reference so all `World` objects actually point to the same array.

You should now be able to move forward and backward through your Game of Life by selecting a pattern and then using "f" and "b", respectively.