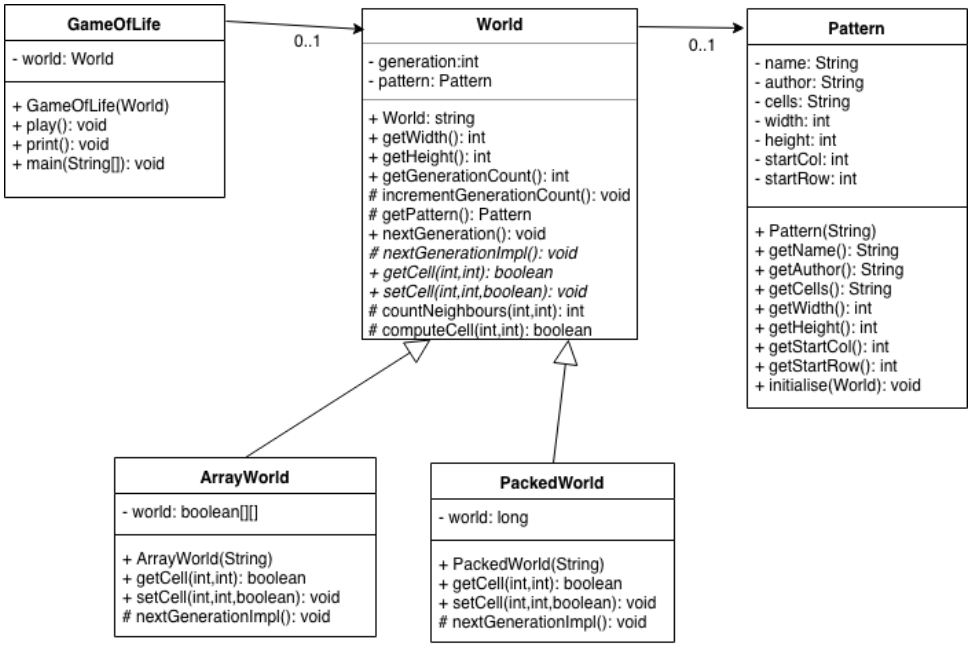
American University of Armenia
CS 120 Intro to OOP
Spring 2019

Homework Assignment 5, Part 1

In this part of the homework we will explore inheritance. So far we have created two unrelated Game of Life implementations: one using a packed long for the world, and one using an array. These two implementations share a lot of properties (they both have widths, heights, a method to query a cell, and so on). You will refactor your code to remove duplication. Here is the UML class diagram of where you end up:



If you have difficulty understanding this diagram, please refer to Section 12.1 of the "Absolute Java" textbook and additional sources on the web.

1. **(5 points)** As part of your homework, you are expected to produce a short report (1–2 paragraphs).

   In your report explain how the "has a" relationship is visualised in the UML diagram and how abstract entities are highlighted.

2. **(10 points)** To start with, we are going to extract a `World` class that will be the parent class to `ArrayLife` (which we will shortly rename to `ArrayWorld` for naming consistency). We need to tease out all of the properties associated with the abstract notion of a world, versus those associated with a concrete implementation of a world (such as an array). By comparing `ArrayLife` and `PackedLife` (and adding a couple of new things) we get to:

   ```
   State:
     int generation;
     Pattern pattern;

   Methods:
     World(String pattern)
   ```

```
int getWidth()
int getHeight()
int getGenerationCount()
void incrementGenerationCount();
Pattern getPattern();
void nextGeneration()
boolean getCell(int col, int row)
void setCell(int col, int row, boolean val)
int countNeighbours(int x, int y)
boolean computeCell(int x, int y)
void play()
```

Some notes:

- the `generation` variable is new: it's to keep track of how many generations we've passed through (i.e. it's just a counter that should be initialised to zero). The method `getGenerationCount()` should provide general access to its value and the method `incrementGenerationCount()` should be available to the class and its subclasses (but not to everything).
- the `Pattern` object stores information like the height and width for us, so we shouldn't duplicate that information by adding extra state to `World`.

In a moment you will fill in the `World` class such that it has the state and methods listed above. *However* some of the methods can only be filled in by assuming a particular representation of the world, which is not what we want to do for `World`. This is exactly the purpose of `abstract`—it says "Any non-abstract class that inherits from me must have a method with this prototype, although I'm not specifying it in any other way".

Hence `World` and some of the methods within it should be declared `abstract`.

To get you started, `getCell(...)` clearly depends on what we are using to represent the world. Therefore we can start our `World` class with:

```
public abstract class World {

  public abstract boolean getCell(int c, int r);

  // ...
}
```

Create a new class `World` based on this structure. You will need to assign each piece of state and each method an appropriate access modifier (public/protected/private). Identify the two other methods that should be abstract in `World` and declare them as such. The `World` constructor should initialise its state appropriately, including creating a `Pattern` object, but nothing more.

3. (5 points) Now we need to adapt `Pattern` to remove its assumptions that we use arrays or `long`, which are made in the `initialise` methods:

`public void initialise(boolean[][] world)`,

`public long initialise()`.

You need to have a single `initialise` method and make it act on a `World` object:

`public void initialise(World world)`

Edit your `Pattern` class to have the new `initialise` prototype and provide an appropriate body.

Hint: you need to remove any code that assumes world is an array or a `long` (e.g. `world[x][y]=true;` and replace it with calls to the more general `setCell()` (e.g. `world.setCell(x,y,true);`)).

4. **(5 points)** Now you need to inherit from `World` to create a new `ArrayWorld`.

   Create a class `ArrayWorld` that extends `World` in an appropriate way. Using your `ArrayLife` code from last homework as a guide, fill in the abstract methods inherited from `World`. Here is a template to start:

```
public class ArrayWorld extends World {

  private boolean[][] world;

  public ArrayWorld(String serial) {
    super(serial);
    // TODO: initialise world
    getPattern().initialise(this);
  }

  // TODO: fill in the inherited formerly-abstract methods

}
```

   Make sure your `nextGeneration()` method in `ArrayWorld` increments the generation counter.

   Add the following `main` method to `ArrayWorld` and check it all works as before:

```
public static void main(String args[]) throws Exception {
  ArrayWorld pl = new ArrayWorld(args[0]);
  pl.play();
}
```

   Note: The most common problem in this homework is confusion between columns and rows. It should be world[row][col] and so created as new boolean[HEIGHT][WIDTH].

   In your report, explain what the call `getPattern().initialise(this)` does.

5. **(5 points)** You will now inherit from `World` to create a new `PackedWorld`. This process is similar to the development of `ArrayWorld` using `ArrayLife` above.

   Create a class `PackedWorld` that also extends `World` but uses a packed long for its underlying board representation (so `world` is a `long`). Note the following requirements:

   - `PackedWorld` must have one constructor that takes in a Pattern format string (*not* a `long`) and calls `getPattern().initialise(this)`
   - If the pattern cannot be represented by a `long` (i.e. more than 64 bits) the constructor should throw an `Exception` with a sensible message.
   - Your class should *not* depend on `PackedLife`, although you will almost certainly want to copy/paste functions from it.
   - As before, make sure your `PackedWorld`'s `nextGeneration()` method increments generation count found in `World`.

   Add the following `main` method to your `PackedWorld` class and check it all works:

```
public static void main(String args[]) throws IOException {
  PackedWorld pl = new PackedWorld(args[0]);
  pl.play();
}
```

6. **(15 points)** At the moment our `World`-based classes aren't a clean implementation of the world concept, since they also incorporate the concept of the gameplay. You will now restructure things so you have a class `GameOfLife` that represents everything to do with the actual game but can switch between the implementations of the world.

   Create `GameOfLife` using the template below:

```
public class GameOfLife {

  private World world;

  public GameOfLife(World w) {
    world = w;
  }

  public void play() throws IOException {
    //TODO
  }

  public void print() {
    // TODO
  }

  public static void main(String args[]) throws IOException {

    World w=null;

    // TODO: initialise w as an ArrayWorld or a PackedWorld
    // based on the command line input

    GameOfLife gol = new GameOfLife(w);
    gol.play();
  }
}
```

   Complete the `play()` and `print()` methods based on the equivalent methods in `World`, `ArrayWorld` or `PackedWorld`. Then remove the `print()` and `play()` methods from those classes, along with any other `main` methods. Adapt `print()` so that it prints the generation number at the start of the pattern, next to a "-", separated by a space. e.g.:

```
- 0
--------
---#----
-#-#----
--##----
--------

- 1
--------
--#-----
---##---
--##----
--------
```

   Complete the `main` method such that you can run:

```
> java GameOfLife --array <format string>
> java GameOfLife --packed <format string>
> java GameOfLife <format string>
```

The first form should use an `ArrayWorld` instance, the second a `PackedWorld` instance. The third form acts as a default, which should use `ArrayWorld`.

Test your implementation fully. Note that:

```
> java GameOfLife --packed "Glider:Richard K. Guy:20:20:1:1:001 101 011"
```

should produce an exception (20-by-20 is too big for a packed long), while

```
> java GameOfLife --array "Glider:Richard K. Guy:20:20:1:1:001 101 011"
```

should work fine.

Note that `GameOfLife` is exploiting *polymorphic behaviour* here. `ArrayWorld`/`PackedWorld` "is-a" `World` instance and we can reasonably write things like:

```
World w = new ArrayWorld(format);
w.nextGeneration();
```

7. **(5 points)** There's one niggling problem with your current design: you have to remember to call `incrementGenerationCount` in `nextGeneration`: nothing in the design forces you to do it. Right now, that's fine: it's in your short term memory to do that (and there were explicit instruction!). But we can tweak the design a bit to make *sure* it happens, which reduces the opportunities for errors in the future, by you or anyone who uses your code.

   The trick is to put the increment call into `World`. To do this we need to make the public `nextGeneration` method concrete (it's currently abstract of course). Then we add a new abstract and *protected* method `nextGenerationImpl` that becomes the one that is filled in within `ArrayWorld`/`PackedWorld`. So within `World` we will have:

```
public void nextGeneration() {
  nextGenerationImpl();
  generation++;
}

protected abstract void nextGenerationImpl();
```

   By making it `protected`, the new `nextGenerationImpl()` is hidden from outside classes, who see no change in the external interface to the class (i.e. there is still a public `nextGeneration()` method that does exactly what it did before.

   Adapt your `World` class to have the changes above. Change the method `nextGeneration()` to `nextGenerationImpl()` in both `ArrayWorld` and `PackedWorld` and remove the code that increments the counter in those files.