

CSE 11 Fall 2020 PA7 & PA8 - Life of Cells

These are the final assignments for the iteration of CSE 11 taught in Fall 2020 by Professors Cao and Miranda at UCSD.

PA7 (check point) and PA8 (final submission) are combined into a single theme - Life of Cells.

PA7 (Check Point)

Due date: Wednesday, Nov 25 @ 11:59PM PST

(Thursday, Nov 26 @ 11:59PM PST w/ slip day. If you submit your assignment late, the autograder will automatically use your slip day if you have any remaining. Note that the [README](#) portion of this assignment cannot be submitted late.)

Provided Files

None

Files to Submit

- Cell.java
- CellStationary.java
- CellMoveUp.java
- CellDivide.java
- CellMoveToggle.java
- CellMoveDiagonal.java
- CellMoveToggleChild.java
- PetriDish.java

Goal

The goal of PA7 (check point) is to apply inheritance to defining the relationship among different types of cells (cell classes). In addition to creating these cell classes, you will also be creating a PetriDish class that represents a 2D board of different types of cells.

Some Background

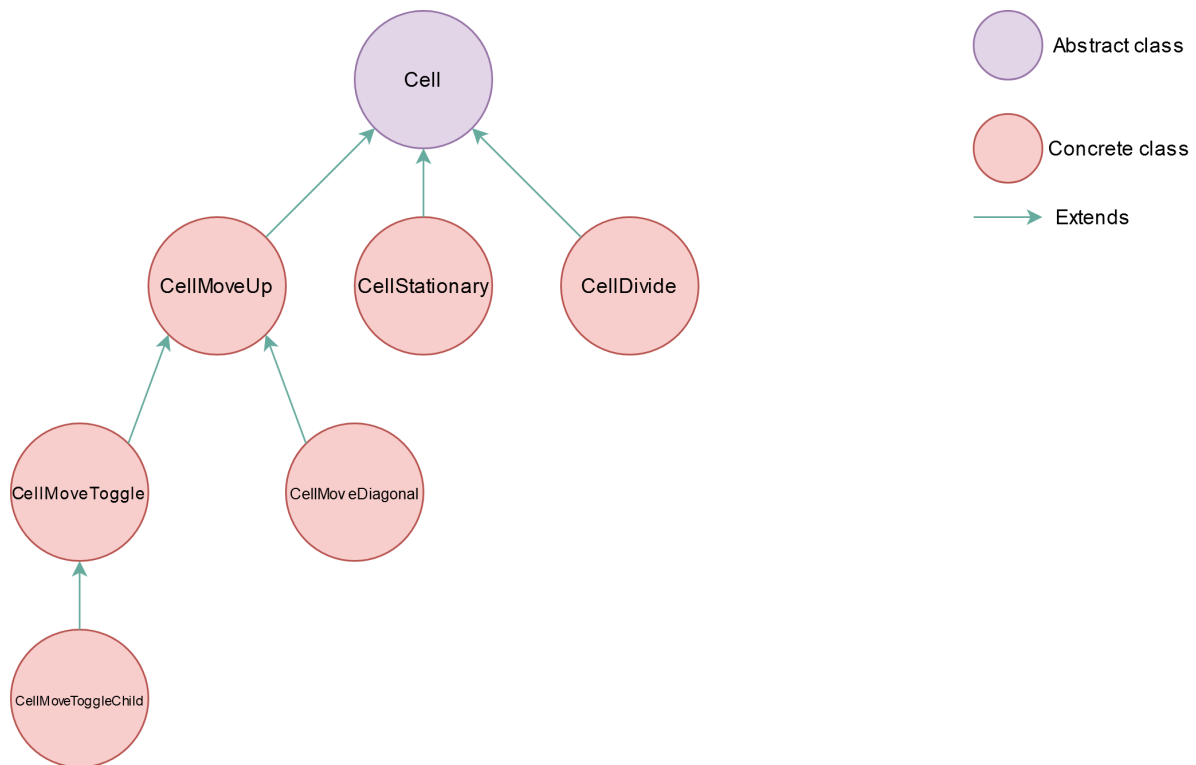
Conway's Game of Life is a cellular automaton that is simple to understand but has interesting properties. You can read more about it on [Wikipedia](#) but knowledge of it is not required for this assignment.

The program we will be implementing for PAs 7 and 8 will be similar to the Game of Life. In the original game, each cell of the board is equivalent. In our game, each cell of the board may be a different cell (in the biological sense) and may have different behaviors.

We have abstracted out all biology, except for the term "apoptosis" which means "programmed cell death" (which we will use to mean that a cell disappears). We will also, at various points, use the word "spawn" to mean for a cell to come into existence.

Part 1: Cells

```
public abstract class Cell {...}
public class CellStationary extends Cell {...}
public class CellMoveUp extends Cell {...}
public class CellDivide extends Cell {...}
public class CellMoveToggle extends CellMoveUp {...}
public class CellMoveDiagonal extends CellMoveUp {...}
public class CellMoveToggleChild extends CellMoveToggle {...}
```



Since these classes are all public classes, they should be in separate `.java` files, where each file is named after the class.

Cell.java

```
public abstract class Cell {...}
```

TODO: Instance Variables for Cell

```
public abstract class Cell {
    ...
    public int currRow;
    public int currCol;
    public int mass;
    ...
}
```

- `currRow` stores the row value of the cell.
- `currCol` stores the column value of the cell.

- `mass` stores the mass of the cell.
- Valid values for all the instance variables are ones that are not negative.

TODO: Methods to Implement for Cell

```
public abstract class Cell {
    ...
    public Cell(int currRow, int currCol, int mass);
    public Cell(Cell otherCell);
    public void apoptosis();
    public int getCurrRow();
    public int getCurrCol();
    public int getMass();
    public abstract boolean checkApoptosis(List<Cell> neighbors);
    ...
}
```

public Cell(int currRow, int currCol, int mass)

- This is the constructor for `Cell`.
- Initialize all instance variables with the values passed in as arguments.
- If an argument would make the appropriate instance variable invalid, set the appropriate instance variable to 0 instead.
- Even though `Cell` is an abstract class, we need this constructor because all subclasses should use this constructor to initialize the instance variables inherited from `Cell`.

public Cell(Cell otherCell)

- This is the copy constructor for `Cell`.
- Initialize all instance variables with the instance variables of the `Cell` object passed in as an argument.
- Even though `Cell` is an abstract class, we need this copy constructor because all subclasses should use this copy constructor to copy the instance variables inherited from `Cell`.

public void apoptosis()

- This method will be called on a `Cell` when apoptosis happens.
- Set `currRow`, `currCol`, and `mass` to `-1` to indicate cell death.

Getters `getCurrRow()`, `getCurrCol()`, `getMass()`

- For each getter method, return the current value of the appropriate instance variable.

public abstract boolean checkApoptosis(List<Cell> neighbors)

- Given a `List` of `Cell`s, determine if this cell should initiate apoptosis or not. Return `true` if the condition being checked for is satisfied and `false` otherwise (more details below).
- This is an abstract method that each concrete subclass will implement with its own behavior.

Subclasses of Cell

```
public class CellStationary extends Cell {...}
public class CellMoveUp extends Cell {...}
public class CellDivide extends Cell {...}
public class CellMoveToggle extends CellMoveUp {...}
public class CellMoveDiagonal extends CellMoveUp {...}
public class CellMoveToggleChild extends CellMoveToggle {...}
```

Each of these 6 classes derive from `Cell` and will have similar (but not the exact same) behaviors.

TODO: Methods to Implement for each Subclass

They will all implement the following methods with at least the following functionality (more details about special functionality will be described afterward, you will have to modify these signatures to be appropriate per class):

```
public class Subclass extends ParentClass {
    ...
    public Subclass(int currRow, int currCol, int mass);
    public Subclass(Subclass otherSubclass);
    public String toString();
    public boolean checkApoptosis(List<Cell> neighbors);
    ...
}
```

So each class will have two constructors (one taking the three `int` parameters in the order specified here and one being a copy constructor for the current class), override the `toString()` method, and override the `checkApoptosis()` method. You will need to adjust the method parameter types (specifically for the constructors) appropriately for each class.

Note that even though some of these subclasses may have extra instance variables, the initial values of those are not passed into the constructor, so the first constructor should always only have those 3 parameters. Similarly, for a subclass with name `Subclass`, the copy constructor should take a parameter of type `Subclass`.

`public Subclass(int currRow, int currCol, int mass)`

- This is the constructor for the subclass.
- Invoke the parent class's constructor to initialize all instance variables with the values passed in as arguments.

`public Subclass(Subclass otherSubclass)`

- This is the copy constructor for the subclass.
- Invoke the parent class's copy constructor to initialize all instance variables with the instance variables of the `otherSubclass` object passed in as an argument.

public String toString()

- Return the `String` representation of the current object. Each class will have a different representation but they will each be a single character.

public boolean checkApoptosis(List<Cell> neighbors)

- Return `true` or `false` based on `neighbors` depending on the conditions for apoptosis. Each class will have different conditions for apoptosis. Only non- `null` neighbors are included in this list.
- Note that this method does NOT call `apoptosis()` as this method is only for checking if `apoptosis()` should be called later.

CellStationary Specifics

CellStationary String representation

- `"."`.

CellStationary conditions for checkApoptosis

- Checks for whether this cell has fewer than or equal to 7 and greater than or equal to 3 neighbors.
-

CellMoveUp Specifics

CellMoveUp String representation

- `"^"`.

CellMoveUp conditions for checkApoptosis

- Checks for whether this cell does not have exactly 4 neighbors.
-

CellDivide Specifics

CellDivide direction instance variable

- `CellDivide` will have an extra instance variable, `public int direction`, that will represent the direction the cell will divide into. The variable is public for grading purposes.
- The (non-copy) constructor for this class should default `direction` to `1`.

CellDivide String representation

- `"+"`.

CellDivide conditions for checkApoptosis

- Checks for whether this cell has exactly 3 neighboring cells.
-

CellMoveToggle Specifics

CellMoveToggle toggled instance variable

- `CellMoveToggle` will have an extra instance variable, `public boolean toggled`, that will represent if the cell is currently "toggled" or not. It is made public for grading purposes.
- The (non-copy) constructor for this class should default `toggled` to `true`.

CellMoveToggle String representation

- "T" if "toggled" and "t" otherwise.

CellMoveToggle conditions for checkApoptosis

- Checks for whether this cell has fewer than 2 or greater than 5 neighbors.
-

CellMoveDiagonal Specifics

CellMoveDiagonal orientedRight and diagonalMoves instance variables

- CellMoveDiagonal will have two extra instance variables, public boolean orientedRight, which will represent if the cell is currently oriented to the right or not (to the left), and public int diagonalMoves, which will count the number of moves this cell has made (there will be more about "moving" in the PA8 portion).
- The (non-copy) constructor for this class should default orientedRight to true and diagonalMoves to 0.

CellMoveDiagonal String representation

- "/" if oriented right and "\" otherwise.

CellMoveDiagonal conditions for checkApoptosis

- Checks for whether this cell has greater than 3 neighbors.
-

CellMoveToggleChild Specifics

CellMoveToggleChild numAlive static variable

- CellMoveToggleChild will have an extra static variable, public static int numAlive, that will represent the number of instances of this cell type that are currently alive (so it should be 0 when there are no instances of this cell).
- The constructor for this class should increment this value by 1 each time. This behavior should apply to both the normal constructor and the copy constructor.
- This class should also override the apoptosis() method to not only call its parent's apoptosis() method but to also decrement numAlive by 1.

CellMoveToggleChild String representation

- This class should not override CellMoveToggle's toString() method.

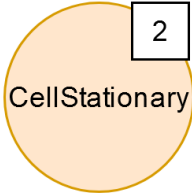
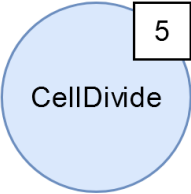
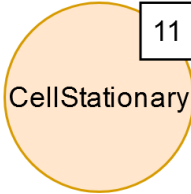
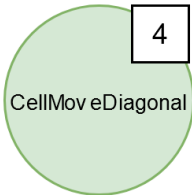
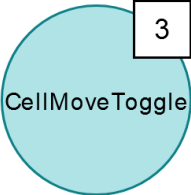
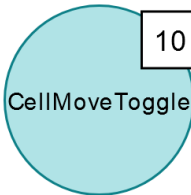
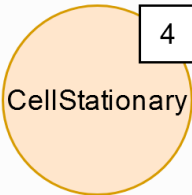
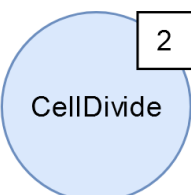
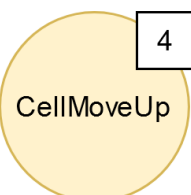
CellMoveToggleChild conditions for checkApoptosis

- Checks for whether CellMoveToggle's conditions for apoptosis are satisfied AND there are fewer than 10 CellMoveToggleChilds alive.

Part 2: PetriDish

```
public class PetriDish {...}
```

A **PetriDish** object contains a board that holds all the cells growing on the **Petridish**.

TODO: Instance Variables for PetriDish

```
public class PetriDish {  
    ...  
    public Cell[][] dish;  
    ...  
}
```

- **dish** represents the petri dish in the current iteration. A **null** value in the array represents an "empty space" and a non- **null** value represents an "alive" cell. The board is represented with the 0th row at the top and the 0th column at the left.

TODO: Methods to Implement for PetriDish

```
public class PetriDish {  
    ...  
    public PetriDish(String[][] board);  
    public String toString(); //optional  
    ...  
}
```

public PetriDish(String[][] board)

- This is the constructor of the `PetriDish` class.
- `board` is a 2D array of Strings representing what cells the dish should be filled with.
- Each String is the string `"null"` if that position in the petri dish is an "empty space" and is of the format `"{CELL_TYPE} {MASS}"` for "alive cells."
- For example, the 2D String array could be created like this:

```
String[][] petri = new String[][]{ {"CellMoveUp 0", "CellMoveToggle 1", "CellMoveToggleChild 2", "null"},  
{"CellMoveDiagonal 3", "CellDivide 4", "CellMoveToggle 5", "null"} };
```
- Populate each position of the instance variable `dish` with references to unique objects corresponding to the specific types and masses denoted in the 2D String array.
- Assume that `board` is always valid.

public String toString()

- This can be used to print `dish` to visualize this `PetriDish` (and is optional).
- Below is an example implementation.

```
public String toString() {  
    StringBuilder sb = new StringBuilder();  
    sb.append(HORIZONTAL_BARS); //typically 2*board.length+3 would display the board nicely  
    for(int i = 0; i < board.length; i++){  
        sb.append(VERTICAL_BAR);  
        for(int j = 0; j < board[0].length; j++){  
            sb.append(board[i][j] == null ? EMPTY_STRING : board[i][j].toString());  
            sb.append(VERTICAL_BAR);  
        }  
        sb.append(NEW_LINE);  
        sb.append(HORIZONTAL_BARS);  
    }  
    return sb.toString();  
}
```

PA7 README

All the questions for the README portion of this assignment can be found on [Gradescope](#). Note that this portion of the assignment *cannot* be submitted late.

PA7 Survey

Please fill out this survey, worth 1 point to your PA grade each, to help us improve the experience of this class!

- [Weekly Reflection 7](#).

Submit PA7 Check Point

Turning in your code

Submit all of the following files to Gradescope by **Wednesday, Nov 25 @ 11:59PM PST** (Thursday, Nov 26 @ 11:59PM PST w/ slip day):

- Cell.java
- CellStationary.java
- CellMoveUp.java
- CellDivide.java
- CellMoveToggle.java
- CellMoveDiagonal.java
- CellMoveToggleChild.java
- PetriDish.java

When submitting, please wait until the autograder finishes running and read the output. **Your code must compile in the autograder in order to receive proper partial credit.**

PA7 Style

Coding style is an important part of ensuring readability and maintainability of your code. We will grade your code style in the submitted code files according to the style guidelines. Namely, there are a few things you must have in each file/class/method:

1. File headers
2. Class headers
3. Method headers
4. Inline comments
5. Proper indentation (do not intermingle spaces and tabs for indentation)
6. Descriptive variable names
7. No magic numbers (exception: `main()` for testing purposes)
8. Reasonably short methods (if you have implemented each method according to specification in this write-up, you're fine)
9. Lines shorter than 80 characters (note, tabs will be counted as 4 characters toward this limit. It is a good idea to set your tab width/size to be 4)
10. Javadoc conventions (`@param`, `@return` tags, `/**` header comments `*/`, etc.)

A full style guide can be found [here](#). In addition, an example of a properly styled Java file has been posted on [Piazza](#). If you need any clarifications, feel free to ask on Piazza.

Evaluation

1. [Correctness](#)

You will earn points based on the autograder tests that your code passes. If the autograder tests are not able to run (e.g., your code does not compile or it does not match the specifications in this writeup), you may not earn credit.

2. [Coding Style](#)

3. [README](#)

4. [Weekly Reflection Survey](#).

PA8 (Final Submission)

Due date: Wednesday, Dec 09 @ 11:59PM PST

(Thursday, Dec 10 @ 11:59PM PST w/ slip day. If you submit your assignment late, the autograder will automatically use your slip day if you have any remaining. Note that the [README](#) portion of this assignment cannot be submitted late.)

Provided Files

None

Files to Submit

- Cell.java
- CellStationary.java
- CellMoveUp.java
- CellDivide.java
- CellMoveToggle.java
- CellMoveDiagonal.java
- CellMoveToggleChild.java
- PetriDish.java
- Divisible.java
- Movable.java

Goal

The goal of PA8 (final submission) is to apply the concepts of interfaces and polymorphism to complete the Life of Cells game. New files include `Divisible.java` and `Movable.java`. Below also includes what you need to **add** to the existing classes that you worked on in the checkpoint. You need to keep/have a **correct version of the checkpoint part** in order for the final submission to work correctly.

Part 1: Interfaces

```
public interface Movable {...}  
public interface Divisible {...}
```

Each of these interfaces should be defined in their own file with the appropriate filename.

public interface Movable

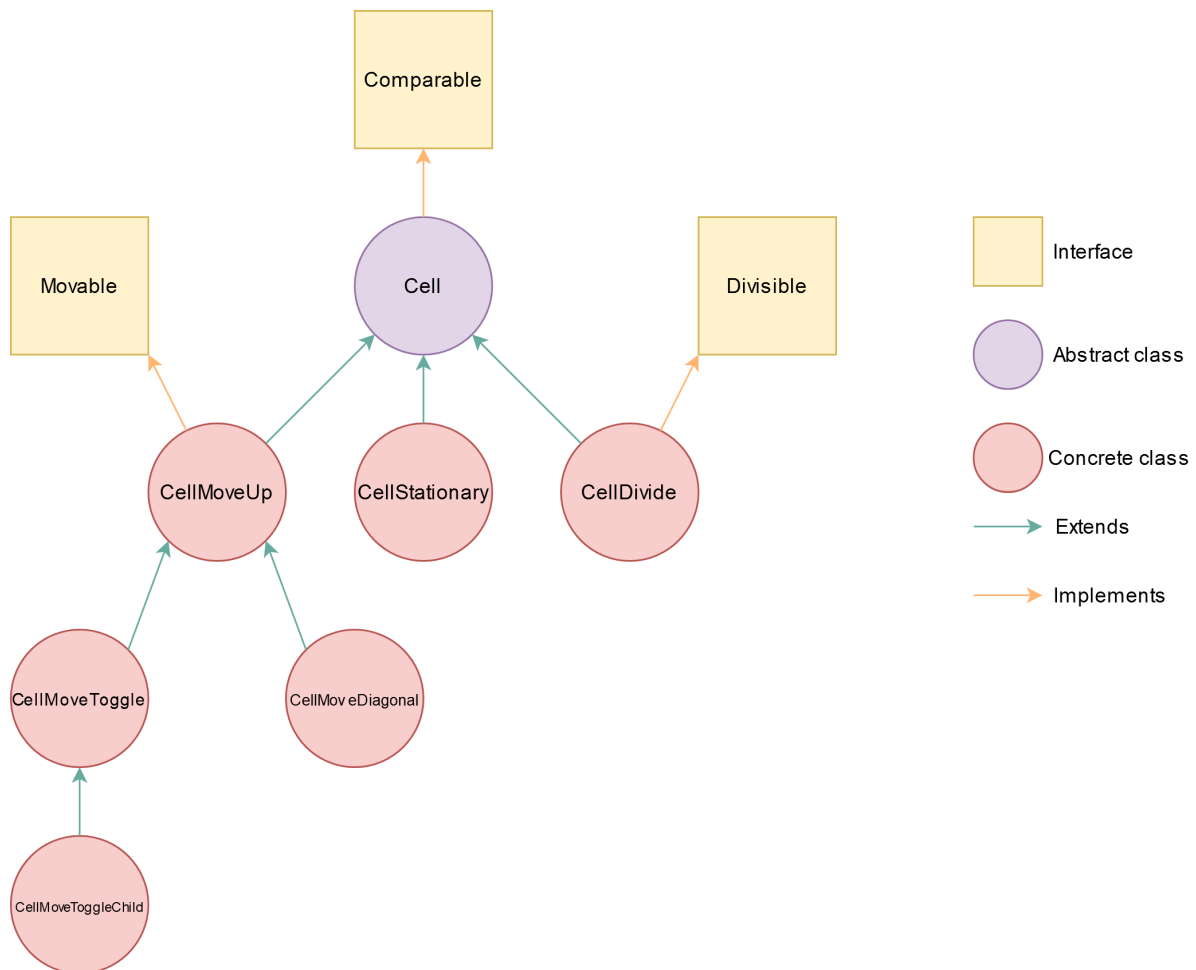
This interface has a single method, `public abstract int[] getMove()` , which will be used to determine where `Movable` objects should move to.

public interface Divisible

This interface has a single method, `public abstract int[] getDivision()` , which will be used to determine where `Divisible` objects should divide to.

Part 2: Cells - Revisited

```
public abstract class Cell implements Comparable<Cell> {...}
public class CellStationary extends Cell {...}
public class CellMoveUp extends Cell implements Movable {...}
public class CellDivide extends Cell implements Divisible {...}
public class CellMoveToggle extends CellMoveUp {...}
public class CellMoveDiagonal extends CellMoveUp {...}
public class CellMoveToggleChild extends CellMoveToggle {...}
```



Note that the signature for some of these classes have changed.

```
public abstract class Cell implements Comparable<Cell>
```

```
public abstract class Cell implements Comparable<Cell> {...}
```

TODO: New Methods to Add to Cell

```
public abstract class Cell implements Comparable<Cell> {
    ...
    public int compareTo(Cell otherCell);
    public abstract Cell newCellCopy();
    ...
}
```

public int compareTo(Cell otherCell)

- Returns whether `this` cell has a larger mass than `otherCell` according to the specifications of `compareTo()` from `Comparable`. The actual value you return is not important, as long as the sign (or zeroness) is correct.

public abstract Cell newCellCopy()

- This method's expected behavior is described below.

TODO: New Method to Implement for each Subclass

Each of these subclasses of `Cell` should already be defined in their own appropriate `.java` file. They will all implement the following new method with the same functionality.

```
public class Subclass extends ParentClass {  
    ...  
    public Cell newCellCopy();  
    ...  
}
```

public Cell newCellCopy()

- In this method, simply return a deep copy of the calling object.

TODO: New Method to Implement for Any Subclass that implements Movable

Each of these classes that implement the `Movable` interface should already be defined in their own appropriate `.java` file from PA7. Note that this applies to classes that do not implement `Movable` directly (meaning that they might extend a class that implements `Movable`). They will implement the following method with at least the following functionality (more details about special functionality will be described afterward):

```
public class Subclass implements Movable {  
    ...  
    public int[] getMove();  
    ...  
}
```

public int[] getMove()

- This method defines how the cells that are capable of moving will move around the petri dish. Return an `int[]` of length 2 that represents the intended new position of the cell, where the first element in the array is the row and the second element in the array is the column of the new position. Note that the cells themselves do not have a concept of the petri dish, so it will not do any boundary checking (i.e. values in the returned array could be negative and "out of bounds"). This method does not update any instance variables of the cell that calls it (i.e., it doesn't actually do the move, it just returns where to move).

TODO: New Method to Implement for Any Subclass that implements Divisible

Each of these classes that implement the `Divisible` interface should already be defined in their own appropriate `.java` file from PA7. They will implement the following method with at least the following functionality (more details about special functionality will be described afterward):

```
public class Subclass implements Divisible {  
    ...  
    public int[] getDivision();  
    ...  
}
```

`public int[] getDivision()`

- This method defines where a new cell that is divided from (spawned by) the original calling cell will be placed. It will return an `int[]` that represents the intended position of the spawned cell, where the first element in the array is the row and the second element in the array is the column of the intended position of the spawned cell. Note that the cells do not have a concept of the petri dish so it will not do any boundary checking (i.e. values in the returned array could be negative and "out of bounds"). This method does not actually create any new cells.

CellStationary Specifics

CellStationary behavior

- Cells of type `CellStationary` do not have any special behavior.
-

CellMoveUp Specifics

CellMoveUp `getMove` behavior

- Move up by one row.
-

CellDivide Specifics

CellDivide `getDivision` behavior

- The new cell should be placed based on the current `direction`, where `0` means down, `1` means up, `2` means left, and `3` means up. Cycle through the directions in this order after each call to this method.
-

CellMoveToggle Specifics

CellMoveToggle `getMove` behavior

- Move up by one row if `toggle` is true, else remain stationary.
 - After `getMove()` is called (regardless of what the move was), `toggle` should have its boolean value flipped.
-

CellMoveDiagonal Specifics

CellMoveDiagonal `getMove` behavior

- If the orientation is right, move to the right and up, by one column and one row, respectively. Otherwise left and up, by one column and one row, respectively. Increment `diagonalMoves` (unconditionally) and, if the number of moves is now a multiple of 4, switch the orientation (conditionally).
-

CellMoveToggleChild Specifics

CellMoveToggleChild `getMove` behavior

- This class should not override `CellMoveToggle`'s `getMove()` method.

Part 3: PetriDish - Revisited

```
public class PetriDish {...}
```

In PA7, we only gave `PetriDish` a constructor (and possibly a `toString()` method). Now, we want to add functionality to this class to drive the game/simulation.

TODO: New Instance Variables for PetriDish

```
public class PetriDish {  
    ...  
    public List<Movable> movables;  
    public List<Divisible> divisibles;  
    ...  
}
```

- `dish` represents the petri dish in the current iteration.
- `List<Movable> movables` is a list of all the `Movable` cells in the petri dish. There is no required order for the `Movable` cells in this list.
- `List<Divisible> divisibles` is a list of all the `Divisible` cells in the petri dish. There is no required order for the `Divisible` cells in this list.

TODO: New Methods to Implement for PetriDish

```
public class PetriDish {  
    ...  
    public List<Cell> getNeighborsOf(int row, int col);  
    public void move();  
    public void divide();  
    public void update();  
    public void iterate();  
    public void simulate(); //optional  
    ...  
}
```

`public List<Cell> getNeighborsOf(int row, int col)`

- Return a list of (non- `null`) cells neighboring (accounting for wrapping) the input location in the petri dish. Each cell has up to 8 neighboring cells (in the cardinal and ordinal directions). The list should have neighboring cells in the order of northwest, north, northeast, west, east, southwest, south, and southeast.
- If either of `row` or `col` is out of bounds (no wrapping for this one), return `null` instead.

`public void move()`

- Move all `Movable` cells in the petri dish based on each cell's respective `getMove()` behavior.
- If a cell's new position would be out of bounds, wrap the position value around on the dish (i.e if a cell moves off of the dish by moving too far up, it should wrap around to the bottom of the dish, similar to the wrapping in PA4). Also, make sure to handle wrapping for corners. This should work for any cell implementing the `Movable` interface appropriately, not just the ones we've written for this assignment.

- If there are **Movable** cells that move into the same position as non- **Movable** cells, the non- **Movable** cell will always die (and call its **apoptosis()** method).
- If there are multiple **Movable** cells in the same position after all the cells have moved, the **Movable** cell with the largest mass will stay and all other cells at that position should die (and call their **apoptosis()** methods). If there is a tie in the largest masses, all the cells at that position die.
- Remember to update all instance variables appropriately.

public void divide()

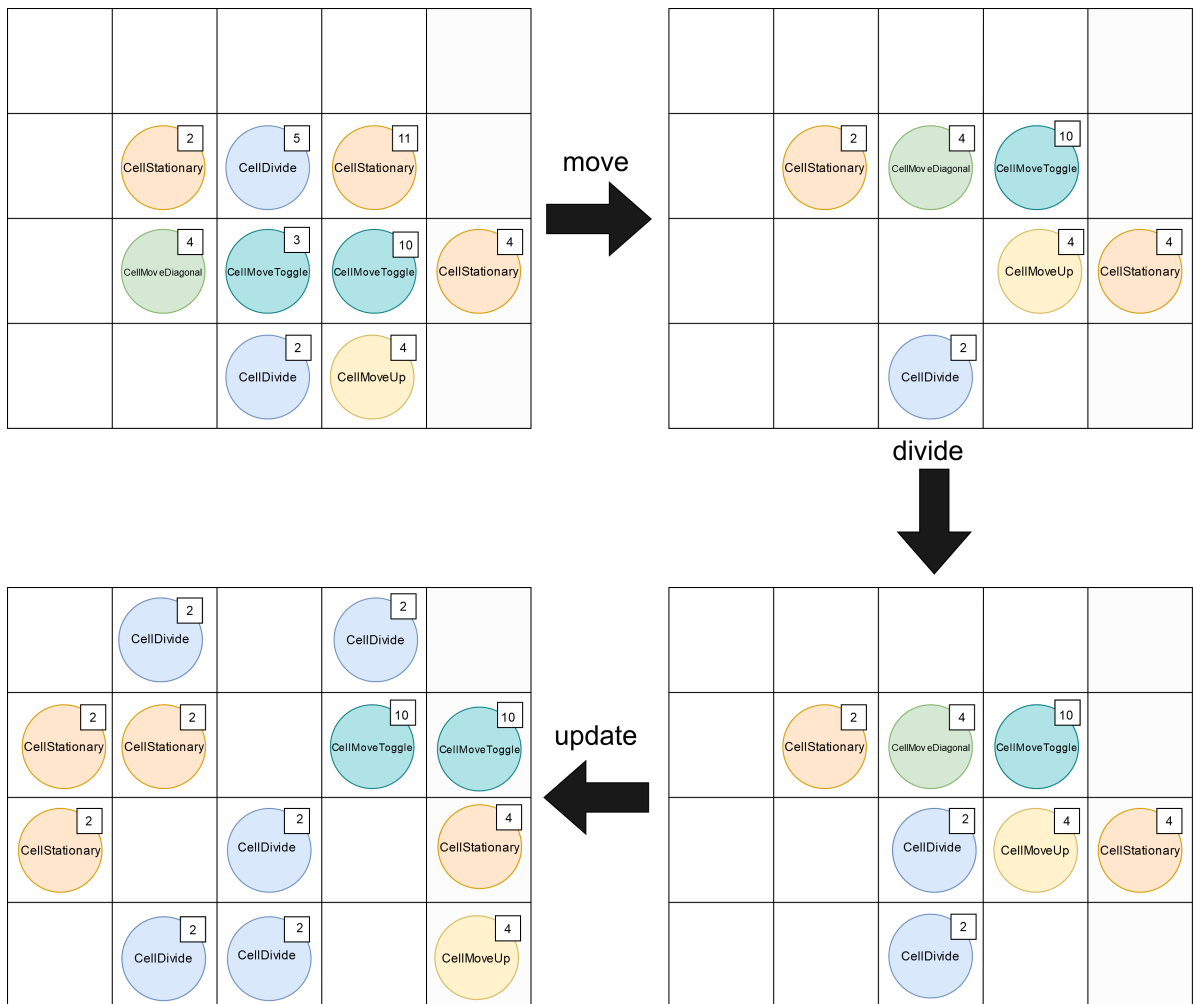
- "Divide" all **Divisible** cells in the petri dish based on each cell's respective **getDivision()** behavior.
- If a newly created cell's position would be out of bounds, wrap the position value around on the dish (i.e if a cell moves off of the dish by moving too far up, it should wrap around to the bottom of the dish, similar to the wrapping in PA4). Also, make sure to handle wrapping for corners. This should work for any cell implementing the **Divisible** interface appropriately, not just the ones we've written for this assignment.
- **Divisible** cells can only divide onto positions of the board that were empty prior to the call to **divide()** . If the position is not empty, no division onto that position will occur.
- If there are multiple **Divisible** cells on the same position after all the cells have divided, the **Divisible** cell with the largest mass will stay and all other cells at that position should die (and call their **apoptosis()** methods). If there is a tie in the largest masses, all the cells at that position die.
- Remember to update all instance variables appropriately.

public void update()

- Simultaneously initiate apoptosis for all eligible cells and spawn new cells for eligible empty spaces. This means that all conditions are based on the petri dish right before this method is called.
- For the cells that will go into apoptosis, set the value of their position on the dish to **null** .
- For spaces in the petri dish that were initially empty, if there are between 2 to 3 (inclusive on both ends) alive cells around (taking into account wrapping around the petri dish), spawn a (deep) copy of the cell that appears first in the list returned from **getNeighborsOf()** .
- Remember to update all instance variables appropriately.

public void iterate()

- One call of **iterate()** contains one **move()** (first step), one **divide()** (second step), and finally one **update()** of the petri dish, in order.
- On the next page is an example of one iteration (with move, divide and update steps shown). Note that this is the first call to **move()** after initializing the board (so all other instance variables of the cells have the default values).



public void simulate()

- This method runs a simulation of the life of cells (and is optional).
- Below is an example implementation.

```
public void simulate(){
    Scanner sc = new Scanner(System.in);
    System.out.println(this);
    while(sc.hasNextLine()){
        String line = sc.nextLine();
        if(line.equals(EXIT_KEY)){
            break;
        }
        switch(line){
            case MOVE_KEY:
                move();
                break;
            case DIVIDE_KEY:
                divide();
                break;
            case UPDATE_KEY:
                update();
                break;
            case ITERATE_KEY:
                iterate();
                break;
            default:
```

```
        System.out.println(INVALID_MESSAGE);
        break;
    }
    System.out.println(this);
}
sc.close();
}
```

PA8 README

All the questions for the README portion of this assignment can be found on [Gradescope](#). Note that this portion of the assignment *cannot* be submitted late.

PA8 Survey

Please fill out this survey, worth 1 point to your PA grade each, to help us improve the experience of this class!

- [Weekly Reflection 8/9](#).

Submit PA8 Final Submission

Turning in your code

Submit all of the following files to Gradescope by **Wednesday, Dec 09 @ 11:59PM PST** (Thursday, Dec 10 @ 11:59PM PST w/ slip day):

- Cell.java
- CellStationary.java
- CellMoveUp.java
- CellDivide.java
- CellMoveToggle.java
- CellMoveDiagonal.java
- CellMoveToggleChild.java
- PetriDish.java
- Divisible.java
- Movable.java

When submitting, please wait until the autograder finishes running and read the output. **Your code must compile in the autograder in order to receive proper partial credit.**

PA8 Style

Coding style is an important part of ensuring readability and maintainability of your code. We will grade your code style in the submitted code files according to the style guidelines. Namely, there are a few things you must have in each file/class/method:

1. File headers
2. Class headers
3. Method headers
4. Inline comments
5. Proper indentation (do not intermingle spaces and tabs for indentation)
6. Descriptive variable names
7. No magic numbers (exception: `main()` for testing purposes)
8. Reasonably short methods (if you have implemented each method according to specification in this write-up, you're fine)
9. Lines shorter than 80 characters (note, tabs will be counted as 4 characters toward this limit. It is a good idea to set your tab width/size to be 4)
10. Javadoc conventions (`@param`, `@return` tags, `/**` header comments `*/`, etc.)

A full style guide can be found [here](#). In addition, an example of a properly styled Java file has been posted on [Piazza](#). If you need any clarifications, feel free to ask on Piazza.

Evaluation

1. [Correctness](#)

You will earn points based on the autograder tests that your code passes. If the autograder tests are not able to run (e.g., your code does not compile or it does not match the specifications in this writeup), you may not earn credit.

2. [Coding Style](#)

3. [README](#)

4. [Weekly Reflection Survey](#)