Assignment 3: Programming Butterflies

Due Date: Sunday, October 6

*“Happiness is a* ***butterfly****, which when pursued, is always just beyond your grasp, but which, if you will sit down quietly, may alight upon you.”* — Nathaniel Hawthorne

# Administrivia

## Academic Integrity

The utmost level of academic integrity is expected of all students. Please read the following webpages carefully. The second one is a great resource for both students and faculty, looking at Academic Integrity from all viewpoints and answering many questions about procedures.

* <http://cuinfo.cornell.edu/Academic/AIC.html>
* <http://www.theuniversityfaculty.cornell.edu/AcadInteg/>

In general, you can comply with the rules of academic integrity if you obey the following basic rules.

* Never look at or be in possession of assignment code of another student. This includes code from previous semesters.
* Never show or give your assignment code to another student.
* Never post assignment code on Piazza.Of course, it is allowed and encouraged that you ask general questions about assignments on Piazza.

## Groups

You may work in groups of 2 people. If you decide to work with someone, form a group on the CMS well before the assignment deadline. Remember, both people must do something on the CMS: one person invites and the other accepts the invitation. One person then submits the assignment on the CMS, and both members of the group automatically get the same grade.

If you don’t have a partner and would like to work with someone, use the Piazza to find someone.

The members of a group should work together on the assignment. When on the computer, they should sit together, taking turns driving (handling the keyboard and the mouse). It is an academic-integrity violation to split the work into two parts, work on them independently, and then merge the results.

The assignment can be done individually. There is no significant advantage or disadvantage to being in a group. Form a group if that appeals to you, and don’t worry about it if you prefer to work on your own.

## Style Guidelines

Please read and be familiar with [CS 2110’s coding style guidelines](http://www.cs.cornell.edu/courses/CS2110/2013fa/style_guidelines.html). Coding in a stylistically consistent manner is an essential skill, especially as the projects you work on become larger and more collaborative.

It is important that you comment the fields of your class with the *class invariant*, stating what each field means and giving constraints on it. The fields should be private.

It is important that each method you write have a *javadoc specification* that completely, clearly, and precisely specifies what the method does, including any preconditions it has. Ideally, you should write this specification before you write the method body; later, if you decide to change what the method does, even slightly, change the specification and then the body. This way of working prevents many errors and makes it easier to write (and later read) the methods.

## Time Spent

We will ask you to tell us the number of hours you spent on this assignment when you submit your assignment. We will announce the mean, median, and maximum time spent. Please keep track of the time you spent.

# Overview

In this assignment A3, you will implement boustrophedonic[[1]](#footnote-1) implementation of Danaus’ method learn(). From the Greek *bous* meaning “ox” and *strephein* meaning “to turn”, “boustrophedonic” literally translates to “turning as an ox in plowing”[[2]](#footnote-2). A boustrophedonic search of a map is one that uses the pattern of an ox plowing a field: back and forth, alternating between west-to-east and east-to-west.

# Objectives

Upon completion of A3, you will have learned about and will have experience with the following.

* The Danaus project.
* Importing archived source files into an existing Eclipse project.
* Packages and their benefits.
* Non-trivial graph exploration with complex loops.
* Converting between row-column indices and xy-coordinates using strong naming conventions.

# Instructions

## Step Zero: Multidimensional Arrays

Thus far, you’ve had some experience with one-dimensional arrays. For example, the code

Integer[] int\_array = new Integer[2][[3]](#footnote-3);

System.out.println(Arrays.deepToString(int\_array));

prints

[null, null]

and can be thought of abstractly as a row of Integers:

|  |  |
| --- | --- |
| null | null |

Java also supports the construction of multi-dimensional arrays with an arbitrary number of dimensions[[4]](#footnote-4). Particularly applicable to this assignment are two-dimensional arrays. The code,

Integer[][] int\_array = new Integer[2][3];

System.out.println(Arrays.deepToString(int\_array));

prints

[[null, null, null], [null, null, null]]

and can be thought of abstractly as a two-dimensional grid of Integers with two rows and three columns:

|  |  |  |
| --- | --- | --- |
| null | null | null |
| null | null | null |

More generally, the syntax new Integer[**ROWS**][**COLS**] will instantiate an array with **ROWS** rows and **COLS** columns. Likewise, the element in row **ROW** and column **COL** can be retrieved from an array arr with arr[**ROW**][**COL**]. For example, if we wanted to populate each element in a 2x3[[5]](#footnote-5) array of Integers with a unique index, we could do so with the following code:

int i = 0;

for (int row = 0; row < 2; ++row) {

for (int col = 0; col < 3; ++col) {

int\_array[row][col] = i++;

}

}

System.out.println(Arrays.deepToString(int\_array));

This prints

[[0, 1, 2], [3, 4, 5]]

and can be thought of as

|  |  |  |
| --- | --- | --- |
| 0 | 1 | 2 |
| 3 | 4 | 5 |

Alternatively, we can iterate through the array with the following code:

int i = 0;

for (int col = 0; col < 3; ++col) {

for (int row = 0; row < 2; ++row) {

int\_array[row][col] = i++;

}

}

System.out.println(Arrays.deepToString(int\_array));

which prints

[[0, 2, 4], [1, 3, 5]]

and can be thought of as

|  |  |  |
| --- | --- | --- |
| 0 | 2 | 4 |
| 1 | 3 | 5 |

## Step One: A Pair of Docs

Before you read any further, take a moment to read Danaus.pdf. It provides an in-depth explanation of the Danaus project. A good understanding of Danaus is imperative to do well on the assignments. Once you have read Danaus.pdf, take a break, then read it again.

Specifically, you will be implementing method learn(). You should have a strong understanding of learn(), and you should be familiar with the butterfly API to implement it.

## Step Two: Installation

The following instructions walk you through the installation and setup of Danaus source code.

1. Open a browser and navigate to [CMS](http://cms.csuglab.cornell.edu).
2. Download Danaus.jar and store it somewhere safe within your file system. Likely, the download will be a Downloads directory; this works great.
3. Open Eclipse.
4. Select menu item File->New->Java Project. A windowed menu will appear.
5. Enter Danaus as the Project name near the top of the window.
6. Click Finish near the bottom of the window.
7. Select menu item File->Import.
8. Select General->Archive File and click Next >.
9. Click Browse... to the right of From archive file
10. Navigate to Danaus.jar that you previously stored on your hard drive. Select it and press OK.
11. Click Finish.

You should now have a project set up with all the files and resources you need. Feel free to explore source code, its art assets, its documentation, etc. A thorough exploration of Danaus is by no means required, but if you complete the challenge, you will likely reach an enlightened state of butterfly programming. Also, you’ll likely receive high grades on the assignments.

## Step Three: Subclass

In your newly created Danaus project, you should see a source folder src. Inside this folder, you should see two packages: danaus and student.

### An Aside on Packages

As you know from a recitation, packages help us bundle together logically interdependent files. For example, all of the Danaus implementation is bundled together in package danaus. On the other hand, your work is placed in package student. Naming sets of related files can help you think about the modularization of a program and help you find files in a large library. For example, it is not difficult to guess that pacakge java.io contains classes to facilitate input and output operations.

Packages help us manage access to files. You are already familiar with the access specifiers **public**, **protected**, and **private**. There is a default access category, which we could call *package*; it is used when no access modifier is present. For example, many methods within Danaus are not labeled as **public**, **protected**, or **private**, so they are *package*, which means that they can be referenced in any class that is in the package but not in classes outside the package. Thus, the files within package danaus can access certain *package* entities that your butterfly class, in package student, cannot.

Packages also help eliminate naming conflicts. Two classes or methods in different packages can have the same name. For example, you are free to define a method named max even though a method max is also defined in package java.lang in class Math. In languages without packages, modules, or namespaces, such as C, it is common to have library functions prefixed with the name of the library to avoid naming collisions. Packages help avoid this tedium.

### Creating Subclass Butterfly

In package student, create a subclass of AbstractButterfly named Butterfly. This is the class in which you will implement method learn(). It is *absolutely critical*that you create this subclass in package student, *not* in package danaus. All code you write for the Danaus project should go in package student. Adding code to package danaus violates the *package* privacy of package danaus, as described above.

## Step Four: learn()

### Background

We assume you have read Danaus.pdf and have a good understanding of method learn().

The maps in A3 are simplified versions of the maps you will see later on in this project. A2 maps have no Water tiles, and Cliff tiles are sparse and rarely contiguous. There are no Forests or Flowers. All these features will be introduced in later assignments.

Your butterfly also has an infinite amount of energy. In later assignments, your butterfly will have a limited amount of energy and will have to efficiently explore the map without running out of energy to fly.

### Getting Started

Before you begin implementing the boustrophedonic search, it may be helpful to play around with Danaus and acclimate to its environment. This section provides some brief suggestions of things to try with Danaus. If you already have a strong familiarity with the system, feel free to ignore this section.

Let’s begin with an empty implementation of learn(). It will look something like this.

public TileState[][] learn() {return null;}

Try running Danaus with this empty learn function. Music should begin and a windowed GUI should pop up. The GUI should contain a graphical representation of a map and butterfly, a slider to control the frame rate of the animation, information about the state of the simulation, and information about the tile most recently clicked.

Now, let’s begin flying. We can fly using function fly defined in AbstractButterfly. fly takes in two parameters, heading and s. heading is the direction in which you want your butterfly to fly. s is the speed with which the butterfly will fly. Let’s attempt to fly east at a normal speed. Our learn method will now look like this.

public TileState[][] learn() {

fly(danaus.Direction.E, danaus.Speed.NORMAL);

return null;

}

Now when you run Danaus, one of two things will happen.

1. Your butterfly successfully flies east.
2. Your butterfly collides with a cliff and throws and exception.

Clearly, we do not want to prematurely terminate our program in the event of an obstacle collision. We can remedy the situation by catching the exception. Our learn method will now look like this.

public TileState[][] learn() {

try {

fly(danaus.Direction.E, danaus.Speed.NORMAL);

}

catch (danaus.CliffCollisionException e) {}

return null;

}

Now, our learn method safely attempts to fly east. In this contrived example, we do not perform more than one fly operation, and we do not handle obstacle collisions in a meaningful way. In this assignments and in future Danaus assignments, your code will be much more complex.

Here are some exercises you can **optionally** try if you would like more experience with danaus.

1. Attempt to fly east. If you collide with a cliff, fly west.
2. Repeatedly fly in a random direction. If you collide with a cliff, simply catch the exception and continue to fly randomly.
3. Fly radially outward in increasingly larger squares. If you collide with a cliff, handle it creatively.

### Implementing learn()

Your butterfly will be placed in the top left corner of the map. It is your butterfly’s responsibility to traverse the map in a boustrophedonic fashion and construct a two-dimensional array of TileStates to represent the portion of the map your butterfly explored. The specific requirements of the search are described below.

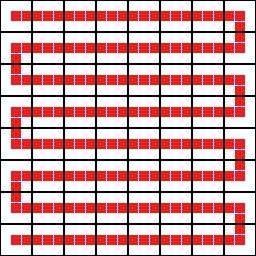
Declare a two-dimensional array of TileState of the appropriate size. Get the height and width of the map using the appropriate functions in class AbstractButterfly.

Next, have the butterfly travel east along the map using fly() or flySafe(), whichever you are more comfortable with. Continue flying until the butterfly hits either the eastern edge of the map or a cliff. Fly south one tile. In this assignment, **we guarantee that for each cliff on the map, there does not exist a cliff to the southeast or southwest of it. We also guarantee that there will not exist any cliffs on the eastern or western edge of the map.** This ensures that your butterfly will unconditionally be able to fly south in the event of a collision with a cliff or an edge of the map.

Continue flying west until the butterfly once again reaches either the western edge of the map or a cliff. Once again, fly south one tile. Repeat this process until one of three terminating conditions is met.

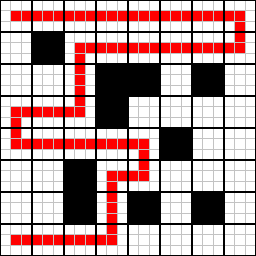
1. Your butterfly is in the bottom row of the map flying east, and it reaches the southeast corner of the map.
2. Your butterfly is in the bottom row of the map flying west, and it reaches the southwest corner of the map.
3. Your butterfly is in the bottom row of the map flying east or west and collides with a cliff.

For example, consider a simple 8x8 map with no cliffs. A correctly implemented boustrophedonic search will look like Figure 1.



**Figure 1.** A correctly implemented boustrophedonic search on a cliffless map.

On the other hand, consider an 8x8 map populated with cliffs. A correctly implemented boustrophedonic search will look like Figure 2.



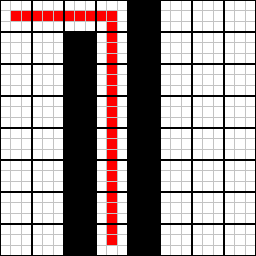
**Figure 2.** A correctly implemented boustrophedonic search on a map populated with cliffs.

Notice that the butterfly **does not** visit every tile in the map. This is acceptable (and probable and expected) for this assignment. In later assignments, you will be improving your search algorithm to visit every node in a map, even in maps with more complex topologies.

Every time the butterfly enters a previously unexplored tile, use the interface defined by Abstract Butterfly to retrieve its TileState and store it in the appropriate spot of the two-dimensional array. Once the butterfly has finished its search and the TileState array has been properly constructed, return the array. This will end the learning phase of the simulation.

### Gotchas[[6]](#footnote-6)

* The maps in Danaus wrap around. That is, if your butterfly flies east off the eastern edge of the map, it will end up on the western edge of the map[[7]](#footnote-7). In your boustrophedonic search, the butterfly should never wrap around the map. That is, the edges of the map should be treated as obstacles.
* The location of the TileState is given in xy-coordinates, but you want to store the tileState in its proper row-column position. You may have to perform a little math, using the map’s height and width, to convert between xy-coordinates and row-column indices. Note that *x*, *y*, *row*, and *column* are integers. If you create helper functions to convert between these values, type safety alone may not be enough to save you from accidently providing a row and column to a function expecting xy coordinates or rearranging the argument order of a function expecting (row, column) to (column, row). Make sure you give your helper functions appropriate names (and also make them private) to help prevent these mistakes.  
  For example, the bottom left corner of a map has Location (1,1), but its index in an array of TileStates is [map\_height - 1][0].
* It is possible and acceptable for a butterfly to enter and exit a row without every flying horizontally within the row. For example, consider the extreme case in Figure 3. Except for the initial row, the butterfly does not fly horizontally.



**Figure 3.** A search with minimal horizontal flight.

### Forgetting run()

For A3, you do not have to implement method run(). Of course, in order for your program to run, your subclass must define run(), since it is an abstract method defined in AbstractButterfly. Just leave the body of run() empty.

# Grading

We will grade your boustrophedonic search algorithm against the course staff’s implementation. The grading steps can summarized as follows.

1. Generate map.
2. Run your butterfly on the map and generate a two-dimensional array of TileStates, M.
3. Run the course staff’s butterfly on the map and generate a two-dimensional array of TileStates, M’ that has been proven correct.
4. Compare M to M’ and generate score.

If your array M matches M’ exactly, you will receive a good score. If M is different than M’, you will receive a score based on the similarity.

# Deliverables

## Checklist

Before you submit your assignment, check to see that you have completed the following tasks.

* You have read this document and Danaus.pdf in excruciating detail.
* You have read and agreed to the academic code of integrity.
* You have reviewed CS 2110’s coding style guidelines, and your code adheres to the guidelines:
  + The fields of class Butterfly are annotated with the class invariant.
  + Each method of class Butterfly has a good javadoc specification.
* Your class Butterfly is in package student, not in package danaus.
* You have placed the total time spent on the assignment at the top of Butterfly in the following format.  
  /\* Time Spent: XX hours. \*/
* If you have a partner, you have formed a group on the CMS. This must be done before you submit.

## Submission

On the CMS, submit two files:

1. Butterfly.java. This file contains subclass AbstractButterfly, including your implementation of a boustrophedonic learn().
2. File README.txt. This file should contain a very brief explanation of your algorithm learn(). Also, give your opinion of this project. README.txt will not be graded, so you need not labor over writing a README magnum opus.

Submit Danaus.zip on the CMS.

1. A 15 character, 25 point scrabble word without multipliers, “boustrophedonic” can be played along any edge of a scrabble board to hit all three triple word scores for over 675 points. According to Hasbro, most professional scrabble players “average between 330-450” points per game. [↑](#footnote-ref-1)
2. <http://etymonline.com/?term=boustrophedon> [↑](#footnote-ref-2)
3. Arrays can, of course, be instantiated with primitives (e.g. new int[2]). However, Arrays.deepToString() only accepts Object arrays, so we use Integers here. In an int array, all nulls would be replaced with 0’s. [↑](#footnote-ref-3)
4. In reality, the number of dimensions is bounded as suggested by [this StackOverflow post](http://stackoverflow.com/a/4060594). [↑](#footnote-ref-4)
5. That is, 2 rows and 3 columns. [↑](#footnote-ref-5)
6. <http://en.wikipedia.org/wiki/Gotcha_(programming)> [↑](#footnote-ref-6)
7. Danaus maps are very similar to the Earth. Naturally, there does not exist the notion of an “eastern hemishphere”. The concept is entirely man-made. Likewise, Danaus maps do not have a natural “eastern edge”, but we artificially denote the tiles with the largest x-values as the eastern edge. The northern, western, and southern edges have the largest y-values, smallest x-values, and smallest y-values respectively. [↑](#footnote-ref-7)