Class Notes 8

Today's Agenda:
1. Review HW assignment
2. Classes and Objects
3. Class Questions
4. What is the point of Object-Oriented Programming?
5. Software Design--solo vs. team

Object Oriented Programming

OOP is a programming *paradigm*.

A paradigm in simple definition is "a way of doing things." More applicable to computer science, a paradigm is the set of guidelines that we use to approach problem-solving given a set of tools.

For example, in procedural programming, our paradigm is to take a larger task and break it down into smaller subtasks through a process called *procedural abstraction*. This allows us to write *methods* that *encapsulate* the functionality that would otherwise be written line-by-line, and repeated when necessary. Methods (or procedures, or functions) can be called repeatedly without having to type in the code that they perform again.

In structured programming, our paradigm is to write many lines of code that manipulate the data until we achieve our objective. While this works for smaller programs, it is incredibly inefficient for large-scale software development.

OOP is concerned less with the order in which we do things, but much more concerned with the relationship of all the pieces of data in a system. OOP is more about logical organization of concepts, and less about the specific order in which they are written in the code. OOP asks us to use the constructs of our programming language to create objects that simulate the problem we are exploring and allows us to experiment with solutions that may not be entirely apparent from the first glance.

Case Study: Programming Computer Games

Back in the 1980's, many games were written by a single programmer, who also took on the role of graphics artist, interface designer, story developer, writer, creative director, composer of music, etc., or they worked with very small teams (2-4 people at most).

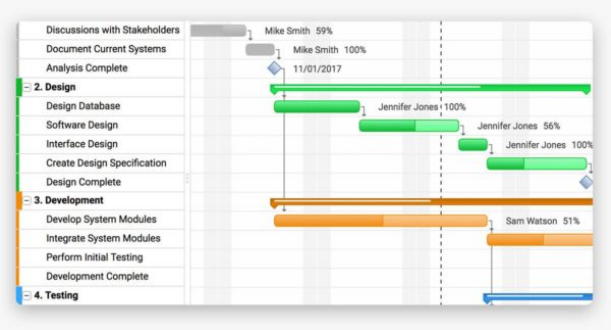
Games like *Galaga* on the Atari 2600 were simple enough that only a few people were needed to handle the game's entire development process. Indeed, the code, graphics, and sound didn't take up that much space, and the game system it ran on could only display so many colors and play so many sounds at one time that in effect, developers had to do as much as possible in a very limited amount of space (processing speed, memory, storage, etc.). Today, independently developed software like this is much rarer, and only really exists in the realm of apps for tablets and smartphones.

Modern day AAA gaming titles (AAA is an informal classification used for video games that are produced and distributed by a major publisher that has a budget associated more with movies and television than smaller computer gaming firms).

AAA titles like Warcraft from Blizzard Entertainment are developed by large teams of programmers, writers, artists, testers...thousands of people who all make individual contributions to a much larger overall product. In order to do this, people cannot indiscriminately contribute snippets of code whenever they feel like it! It must be tracked carefully and the need for *project management* is critical.

As a result of people sharing code, the need for *code conventions*, or common ways of writing code so that it is readable by multiple programmers, is essential to collaboration.

When collaborating, programmers engage in a practice called *pair programming* or *peer review*, where two or more programmers solving a single problem will either code or critique, and swap roles frequently during the development process. By putting more eyes on the code, this allows errors to be caught more quickly and results in higher quality code and fewer errors down the line as a work is in development.



The image on the left is a portion of a screenshot of project management software, in a view called a *timeline* that shows major portions of the project, the time frame in which it will be worked on, and the primary contact person responsible for that portion of the project. A good project manager will be able to delegate the portions of the project to the people who will accomplish the goals for each segment the best.

Case Study: Disease Research

Organizations like the CDC (Center for Disease Control) use software to run simulations on the spread of diseases in a population. This activity takes place all the time because researchers want to see what the potential outcome could be if a population is affected by some new mutation of a particularly virulent disease. This also allows for testing to see how effective treatments can be in preventing the spread of new cases.

A popular mobile game, *Plague, Inc.*, is based on this kind of simulation. In in it, a player assumes the role of a disease of some sort and attempts to infect and eradicate the human population of planet Earth. Each human being in the game is an object that can be infected, or can be inoculated against the disease.

Designing Classes

Class names should be nouns. They represent tangible people, places, or things. The convention is to start them with an uppercase letter, to distinguish them from variables, which start with a lowercase letter.

The layout of a Class should consistently follow this form:
1. variables
2. constructors
3. methods

In a Class, each constructor and method should be fully documented so that we know what their parameters are, and what kind of data is returned (if any). A brief description of the intended purpose of a method is useful as well. Using the comment format provided to us by BlueJ, we are able to automatically generate documentation for our Class.

The *Constructor* of a class is a special kind of method that is run exactly ONE time...when a new object is created from a Class. In fact, the *new* keyword in Java is what triggers this behavior. Our constructor that we write in a class is responsible for initializing all of the variables for that object when it is created.

A class on its own doesn't do anything. It has to be declared and instantiated by a main() function to be useful. We call the class that has the main() function in it the *client code*.

Class Programming Project: Life

Our programming project today is intended to be a simulation of cells, called the *Game of Life*. It was first devised by British mathematician John Horton Conway in 1970. It is a zero-player game, meaning that the evolution is determined by its initial state, and requires no further input. One interacts with the game by creating an initial configuration and observing how it evolves. Advanced programmers will create patterns using specific properties (defined by the methods).

While the program is usually run using a 2D array using only integers (1 for alive and 0 for dead), we are going to implement a more detailed version using an Array of Cell objects.

Each cell in the grid has the following behavior:
1. Any live cell with fewer than two live neighbors dies, as if by underpopulation.
2. Any live cell with two or three live nieghbors lives on to the next generation.
3. Any live cell with more than three live neighbors dies, as if by overpopulation.
4. Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

The initial pattern that we start with (a grid of live and dead cells) is the *seed* of the system. Each time we evaluate all of the cells in the grid, we call it a *generation*. Running the generations repeatedly provides us a long-term simulation of cellular growth and death.

By implementing this simulation using classes and objects, we will be able to introduce other behaviors beyond the basic simulation, for example cells that spontaneously combust (and die), zombie cells that behave differently than normal ones, and cancer-type cells that reproduce regardless of their neighbors, but kill off other cells around them.

Project Details

Create a Class **Cell** to represent an individual cell. A cell should contain a boolean value that indicates whether it is alive or not.

Create a Class **World** to contain the cells. A world is a two-dimensional plane of x width and y length, the size of which we will currently set to 20x20. It is represented by a 2D Array of Cells. Every location has a cell, which is initially alive or dead. You will prepopulate the World using your random number generator. Our displayWorld() method generates a text representation of the current world using "O" to represent live cells and "." to represent dead cells.

Our next task is to apply the four behavioral concepts that define the simulation. Each cell has to evaluate and perform the following:

1. Any live cell with fewer than two live neighbors will die, as if by *underpopulation*.
2. Any live cell with two or three live neighbors *lives* on to the next generation.
3. Any live cell with more than three live neighbors dies, as if by *overpopulation*.
4. Any dead cell with exactly three live neighbors becomes a live cell, as if by *reproduction*.

We will write a new method for the World Class called **generation()** which causes one iteration of these four behaviors to be run. After this method has completed, all of the cells in the world will have been evaluated using the four "rules" above, and a new representation of the world will be saved. It is most effective here to create a new temp 2D array which will be the new state of the world so that the original is not modified. Once the new world state has been determined, the temp array can be copied over the actual world state array.

For each cell, count how many live cells are adjacent to it, then determine whether the state of that cell (alive or dead) changes.

```
  0 1 2
0 . . 0
1 0 0 .
2 . 0 .
```

Looking at the highlighted cell, we have to look at the following positions around it:

row-1, col-1: dead
row-1, col: dead
row-1, col+1: **alive**
row, col-1: **alive**
row, col (*self--do not check!*)
row, col+1: dead
row+1, col-1: dead
row+1, col: **alive**
row+1, col+1: dead

In this case, we could count the number of cells adjacent to it that are alive. Because we have three, the cell in the middle remains alive to the next generation.

If you have cells bordering the edge, your code must check to make sure that the position exists...so if going up a row or left of a column is a place that does not exist, you want to skip checking that non-existent adjacent cell.