Note: Student work submitted as part of this course may be reviewed by Oxford College and Emory College faculty and staff for the purposes of improving instruction and enhancing Emory education.

Course Syllabus for
CS 170Q - *Introduction to Computer Science*
Fall 2015: Section 11J/L-B (Course# 4896 & 4898)

Lecture and Integrated Lab: TuTh 11:50 – 1:30 PM & Tu 1:40 – 2:40 PM in Pierce 206

Instructor: Paul Oser
Email: poser3@emory.edu
Office: Pierce 120B
Hours: TuTh 2:40-5:00 PM along with an "open-door" policy

Web: http://www.oxfordmathcenter.com/drupal7/node/10

Textbook: None required. All notes and coding examples will be available online at the website above. (If students wish to have a text for reference, Daniel Liang's *Introduction to Java Programming* is a good resource – but this text will not be explicitly addressed in the course.)

Overview: This course is an introduction to computer science for the student who expects to make serious use of the computer in course work or research. Topics include: fundamental computing concepts, general programming principles, and the Java programming language. Emphasis will be on algorithm development with examples highlighting topics in data structures.

Goals for Student Learning: Students at the conclusion of this course should be able to...

- Effectively use primitive data types and common pre-made objects in the Java language
- Effectively use program-flow-control concepts (i.e., "for"-loops, "while"-loops, "if"-statements, etc.)
- Effectively use arrays and strings for storing and manipulating a large amount of data
- Build classes and objects of their own design
- Effectively use subclasses and interfaces to facilitate Object-oriented design
- Begin to become familiar with event-driven programming

Prerequisites:

There are no official prerequisites although some familiarity with email and web browsers will be helpful. Knowledge of high school algebra and basic problem solving skills are assumed. This course is the first of a two-semester sequence for computer science majors and is followed by CS 171.

The "Ways of Inquiry" at Oxford:

"Ways of Inquiry" courses are designed to introduce students to the specific ways knowledge is pursued in each discipline through active engagement in the discipline's methods of analysis. INQ courses start with questions, are student-centered and often collaborative, and they place increasing responsibility on students for their own learning. Students not only experience each discipline's distinctiveness but also move beyond its boundaries to understand connections with other disciplines and fields.

The "Ways of Inquiry" Used in this Course:

Writing a computer program is an act of inquiry. There is no "recipe" that can be given to students so charged. On the contrary (and in a quite literal sense) – with every program they write, students "create their own recipe" to accomplish the task in question.

Students will be given many opportunities to write programs – especially through the lab assignments. They will have a goal (a task their program must perform); they will have the tools they need (the language specification and API); and how they get to that goal is up to them. Students will attack these tasks in a variety of ways, some elegant, some brutish, but all will have to pull up their sleeves and get down in the trenches of figuring out how this new thing can be done with what they know.

The instructor's job in this course is to demonstrate the requirements and capabilities of the Java language, give students some good guiding principles, and then largely get out of their way – letting them discover how to use this language to do what they want to do. The instructor will also play a supporting role: 1) helping students see how certain "fundamental" questions can guide their efforts in accomplishing the goals given to them; 2) driving students away from inefficient solutions; and 3) revealing to students (through questioning) cases they haven't considered, that might cause their program to behave in a manner contrary to what they intended – and thus alerting them to the need to debug as necessary.

Students will have the opportunity to pair up with other students as they engage in many of their programming tasks – and the tasks themselves will often connect in some way to interesting real-world problems or subfields of mathematics, cryptography, and/or other disciplines.

How to Approach Programming Assignments in this Course

Students should plan on spending a considerable amount of time in front of a computer working on their assignments outside of class this semester. Students should make every effort to work all of the programs assigned, as programming is a skill best learned by "doing".

Assignments will involve designing, coding, testing and debugging programs based on a written assignment specification. These programs will involve a conceptual understanding of language features and require skill with various software tools. With programming it is important to "work smarter, not harder." Brute-force approaches often lead to long, tedious, unsuccessful hours of work.

As is the case with most disciplines, however, the more one knows the right questions to ask when approaching a given task, the better off one will be. (*Note: it is for this reason that the awareness and practice of asking a discipline's fundamental questions is a key hallmark of inquiry courses*.)

Focusing on the fundamental questions that should always be at the forefront of any programmer's mind can help you write correct, easy-to-understand, and efficient code with minimal effort. Examples of some of these fundamental questions are given below.

- **Input** What information does the program need to know to perform its task? How can the program get this information -- from command line arguments? ...from standard input? ...via a dialog box? ...via mouse movements? ...from a file? etc...
- **Output** What output needs to be created? What form should this output take? Should something be communicated to the user? ...how? Should information be sent back to the console? Should a file be created? ...a graphic? ...other media?
- **Variables & Declarations** What information needs to be remembered in the course of the algorithm to be utilized, and what data type is the most appropriate to use? What type of information is it -- something

numerical? ...text? ...a single character? ...a condition? ...an object? In the case of numerical values, what range of values will be dealt with? Are there a bunch of similar values that must be remembered that could be consolidated into a list? ...a table? ...a three-dimensional (or higher) array? Should client code be able to see this information, or is it for internal use only? Is the information tied to a particular instance of an object? ...or should it be static?

- **Initializing and Updating** What should the value of each of my variables be initially? How should each variable's value be updated throughout the program?
- **Calculation** Does a value need to be calculated? Do the calculations need to be exact, or will close approximations suffice? Does efficiency of calculation matter in this context (i.e., are massive amounts of data involved)? What is the running-time cost of this calculation? Is there a more efficient way to accomplish the same thing?
- **Conversions** Does a particular value (or information) need to be in a different form for the next thing the program must do? How can this be accomplished? ...by promotion? ...by casting? ...by an existing method? ...by a custom method of my own design?
- **Conditionals** Will the program need to do one thing if some condition applies, and something else if another condition applies? What conditions must be considered? What actions must be taken? What's the most appropriate conditional to use in light of these answers? ("if", "if-else", "?", "switch")
- **Loops** Does something need to be done multiple times in a row? ...a fixed number of times? ...will it have to be done at least once? ...until some condition holds? ...what condition? What's the most appropriate looping structure to use in light of these answers? ("for-loop", "while-loop", "do-loop")
- **Methods** Does something need to be done multiple times in different places in the code? Would it help readability and code maintenance to consolidate all code related to a common task into a method? Should the method produce output? If so, what type of output should it create? What inputs will it need to do it's job?
- **Using Arrays** Does something need to be done to every element of an array? What type of looping is required (for one dimensional arrays, a single for-loop might do; for higher dimension arrays, nested loops are necessary)? Can what needs to be done be modeled after one of the standard simple tasks for which arrays are helpful? (finding the position of an element, counting occurrences of an element, summing elements, finding a maximum or minimum, etc...)
- **Designing Classes** What instance variables should be associated with objects of this class - what information describes the state of the objects in this class? What should happen during the construction of objects in this class? What instance methods are required - what should the objects of this class be able to do? What should the visibility of each instance variable or method be -- public, protected, default, private? What methods of this class should be static? What public static constants should be made available to users of this class?
- **Abstract Classes and Subclasses** Are there existing classes that already accomplish most of what needs to be done that one can extend (or include via instance variables)? Can redundancy of code be reduced by introducing abstract classes? Must several variants of a single class be stored together or passed to a method via the same argument?
- **Interfaces** Are there behaviors that different classes will share? Must objects of dissimilar classes (but ones that share one or more methods with the same signature) be stored together? ...or passed to a method via the same argument?
- **Event Driven Programming** Which objects should be generating events? Which objects should be listening for events? What types of listeners will be required?

One should keep in mind that different questions will have more or less importance in different contexts, and answering these questions doesn't automatically (i.e., mechanically) create the program sought. However,

continually asking and answering these questions will both help one start the creation of a program and, later in the process of developing a program, suggest good directions to go next.

Students will see the fundamental questions above (and others as well) being asked in almost every class period by the instructor to drive the development of programs explored in class. This is done in the hope that students will then emulate this process when they develop programs on their own.

The Importance of Comments and Style

The importance of good comments and style in programming cannot be understated. Consequently, a sizable portion of students' grades will depend directly on how well these two elements get addressed.

Students will be asked to include detailed comments threaded throughout their code **for all programs submitted**. These comments should articulate both what the code is doing at each point in the program, and -- even more importantly -- *why* it is being done.

Doing this will produce a couple of consequences: First, this will greatly help with the readability and maintainability of one's code -- which is paramount for people that program. Secondly, by describing "what you are doing and why you are doing it" for each line or section of the code, students are forced to confront code they don't really understand and figure out how it works, as opposed to semi-randomly making modifications to their code until it produces the correct output and then moving on -- without really knowing what they did, or why it worked.

To additionally aid with the readability of the code they produce, students will be given guidance on "good style" conventions to follow when programming. These will include best practices with regard to naming conventions, indentation, constants, etc.. Such practices will greatly assist with debugging programs later, so students should do their very best to adopt these habits as early as possible.

Readings:

Students should read all of the class readings as soon as they are assigned, unless otherwise indicated. These readings will introduce you to the basic structures of the Java language. Most often in class, after a brief introduction to the content, one of two things will be transpiring -- either 1) the instructor will be modeling how to use these structures along with the fundamental questions and good habits mentioned above to create a program that accomplishes some given task, or 2) students will be taking their own crack at creating a program to accomplish some given task. Having prior familiarity with the content of the readings will facilitate both of these endeavors.

Labs:

The labs will constitute the bulk of the graded programming assignments for the course. Each lab may consist of one or more programs that students will need to submit. Generally, students will be given one week from the date of the assignment to submit their programs for grading, although there may be exceptions to this.

In the occasions when students are allowed to work in pairs (i.e., for both on the labs and the final project), students should avoid the temptation to "divide and conquer" their assignments. Students giving into this temptation will only end up "doing" half of the work, and consequently, only really learning half of the material (which invariably gets revealed on test day). Allowing students to work in pairs is intended to create conversations between students as they *both* develop each program in their own way -- conversations that will reveal when one approach is better than another, and should be adopted by both.

In keeping with the inquiry nature of this course, lab assignments will not only test students understanding of core programming concepts, but also seek to connect to some larger context. For example, some of the labs will focus on "real-world" applications, such as photo processing, game creation, or mechanically harvesting data from the internet. Others will connect to other disciplines, like the mathematics behind fractals, number theory, or cryptology.

"Administrivia":

The Java-related software on the laptops in Pierce 206 should also be available to you on the specialty computers in the Library and in the Kaleidoscope Lab, should you not have access to a computer of your own.

Programming assignments should be completed individually or as otherwise directed, although you are welcome to discuss general principles and concepts about the Java language with other students and the instructor.

Students will be asked to periodically turn in java programs via an SFTP upload to the math center server (at mathcenter.oxford.emory.edu). The programs graded may include programs that were partially fleshed out in class, or programs left to the student to complete solely on their own.

Students are required to regularly backup their workspace to protect against the loss and/or catastrophic failure of their computer.

Students should check their Blackboard account for this class **daily**. All announcements and assignments for this class will be posted there.

Exams:

There will be three closed-book tests and a final exam that will test student understanding of the material. The tests will emphasize reading and debugging code more than writing code (students' capability in doing the latter is primarily measured through their performance on the labs). That said, some questions on the tests may also require students to write code. Doing well on these exams will strongly correlate to having read and understood the notes online and other reference material provided, and having worked in earnest -- and successfully -- on the programs assigned up to that point in the class.

Final Project:

A key feature of this course is the increasing independence students are given in terms of their programming assignments over the course of the semester. Initially, students are given programming challenges that are clearly defined, and typically involve only one or two key concepts. However, as the semester progresses, this "scaffolding" will slowly be stripped away. More and more concepts will be integrated together in each subsequent program. Directions will slowly change from 1) "write a program named such-and-such that takes in these exact arguments, and uses these exact tools, to produce this exact output", to 2) "write a program that does such-and-such through some means of your own design to produce such-and-such output formatted in some appropriate manner", to 3) "here's a problem, write a program that solves it", and finally to 4) "come up with an idea for a cool program -- and then build it! (oh, and make sure that somewhere all of the techniques discussed in class over the past month are demonstrated in some meaningful way)"

In keeping with this, as a final project, each student (possibly with a partner) will be expected to come up an idea for a program they wish to implement. After consultation with the instructor to ensure the idea is at the proper level of difficulty, students will need to both write the program in question and articulate how their program demonstrates their mastery of the Java language and methods covered in this course.

Grading:

Lab Assignments: 25%
Exams: 45%
Final Project: 10%
Final Exam: 20%

Letter grades will be given in accordance with the following: A: 90-100%; B: 80%-89%; C: 70-79; D: 60-69%; F: 0-60%.

Letter grades of A-, B+, B-, C+, C-, D+, and D- may be given for percentages near these cutoffs at the discretion of the instructor.

Grades received on lab assignments depend not only on whether the correct output was produced or desired behavior was illustrated, but also on: 1) whether or not the programs were written with "good style" (i.e., standard coding conventions have been followed with regard to indentation, variable names, appropriate use of constants, etc.); 2) whether or not the programs have appropriate comments laced throughout their source code that clarifies, explains, and/or otherwise documents the details of these program and how they work; and 3) whether or not the source code is efficient in both its writing and execution.

Typically, all of the lab grades will be equally weighted. However, the instructor reserves the right to weigh some of these grades more heavily, so that if a particularly challenging program or lab is assigned, students can be rewarded for the extra effort involved.

Late Policy:

Students are expected to be present for all scheduled tests. Any conflicts should be brought to the instructor's attention as soon as possible. If a legitimate reason exists for missing a test – as determined by the instructor – then the test must be taken prior to the regularly scheduled date. In the unusual circumstance where taking the test early is not possible, ***students should be aware that any make-up tests given will be designed to be more difficult to offset the additional time given for study***. Students must provide written documentation in advance of any special accommodations required for testing. This includes additional time or other needs. The final exam cannot be rescheduled.

In general, late programming assignments will not be accepted; this policy will be waived only in an "emergency" situation with appropriate documentation and at the instructor's discretion.

Honor Code Policy:

All class work is governed by the Oxford College Honor Code. No collaboration is allowed on tests or exams. For the labs and final project, however, students may work in pairs if they wish. IMPORTANT: if two students do work as a pair on any given lab – BOTH students' names must appear at the top of every file submitted for that lab. So that this extra text doesn't cause an error during compilation, it should be formatted as a java comment, similar to the example below:

```
/*******************************************/
/**** Lab Partners: Mike Beck & Jon Fowler ****/
/*******************************************/
```

Lab partners may not turn in a single lab between them – each person should turn in his or her own lab assignment – even if it's substantially similar to their partner's (which in most cases would be expected).

Collaboration on lab assignments between students that are not lab partners is not allowed, painfully obvious to spot by the instructor, and does a serious disservice to all involved on test day*. Really.

** This means that if the instructor starts to get the impression that inappropriate collaboration on labs is occurring, which nullifies the capability of the labs to provide a genuine measure of students' ability to write code, the instructor will be forced to ask more questions on the tests that will require students to write code "on the spot". As such code-writing is done with pencil or pen, and without the benefits of code-completion or alerts to syntax errors like those the Eclipse IDE provides, students will undoubtedly find such tests more difficult.*