# Computer Science 171
# Introduction to Computer Programming

Dr. Stephen Bloch
office 203 Post Hall
phone 877-4483
email `sbloch@adelphi.edu`
Web page `http://www.adelphi.edu/sbloch/`
Class Web page `http://www.adelphi.edu/sbloch/class/171/`
office hours MWF 12:00-1:00, T 10:00-3:00

January 23, 2013

## 1 Subject Matter — Overview

Computers do many of the same things people do, only faster, more accurately, and without getting bored. Accordingly, the task of programming a computer is essentially the task of figuring out exactly how *you* would solve a particular problem, and then explaining it to the computer. Unfortunately, computers are literal-minded and completely devoid of intuition, so your explanation must be much more precise and unambiguous than if you were explaining things to a human being. This course, therefore, is about how to figure out precisely and unambiguously what problem you're trying to solve, figure out precisely and unambiguously how to solve it, and explain all this to a computer.

In a broader sense, this course is about problem-solving, and about a particular way of approaching the world which we may call *algorithmic thinking*. I consider computer science a "liberal art" providing training in how to think, regardless of whether you choose it as a profession.

## 2 Who Should Take This Course

If you're a CS or CMIS major, you should take this course as early as possible, preferably in your first or second semester at Adelphi. If you're a CS minor or

a math or Physics major, you should take this course in your first two years at Adelphi. If you're planning on any other major but are curious about how computer programs and programmers work, you are welcome to take this course, but you might be better served by CSC 160; again, consult with the professor. If you have no interest in writing programs of your own, but simply want to *use* Web browsers and search engines, spreadsheets, databases, word processors, *etc.* and perhaps write your own Web pages, you should take CSC 170 instead.

This course does not assume that you've ever written a computer program before, but even if you have, you'll probably learn new ideas here. It's a 4-credit course, meeting for 3 hours of lecture and 3 hours of lab every week; you should budget an additional 6 hours per week outside class for homework and reading. I'm not kidding.

# 3 Goals of the Course

By the end of this course, you should

- be comfortable using a program development environment (like BlueJ) to carry out a design-code-test-debug cycle
- Know the Java syntax for defining and using common programming components such as variables, methods, classes, conditionals, recursion, and looping
- Be able to parse algebraic expressions in Java and ordinary algebra by hand, recognize which operator governs which sub-expressions, and infer types and values of expressions
- Be able to trace the execution of a program by hand, in particular the values of variables at different times and in different method invocations
- Internalize habits of clear coding: good choices of names for variables, methods, and classes; indentation and blank space; commenting; named constants; code re-use and factoring
- Design and implement simple graphical user interfaces, including coordinate-based drawing and some event-handling
- Be able to read a problem specification, abstract away inessential details, and identify important types, relationships, and operations
- Be able to plan the development of a complex program as a sequence of testable versions

- Design methods and classes to be easily re-used without modification, and enhanced without breaking what already works
- Follow step-by-step recipes for designing methods and classes
- Choose and use appropriate test cases at every level of a program; test-driven development, bottom-up testing and debugging

Naturally, you will continue learning many of these skills throughout your computer science career, but you should make a good start on all of them this semester.

# 4 Subject Matter — More Details

## 4.1 "Good" Programs

What distinguises a "good program" from a "bad program"? Obviously, a good program has to work correctly and reliably — a difficult goal in itself, as we'll see. But this is far from enough. In practice, very few programs are written once, used for a while, and discarded: much more often, a program is used until the need for it changes, the program is *modified* (often by a different programmer) to handle the new requirements, the modified program is used for a while, and the cycle repeats. Thus a "good program" must be not only correct the first time around, but structured in such a way that it can easily be modified to accomodate likely changes in requirements. To get across the point of modifiability, I may occasionally **change the assignment slightly on the day that it's due**. So whenever you get an assignment, you should immediately start thinking "how is Dr. Bloch likely to change this at the last minute?" and prepare for such a change. If implementing the change takes you an hour or more, you didn't design the program well for modifiability.

There are other criteria for a "good" program, in addition to correctness and modifiability: fault-tolerance, efficiency, user-friendliness, *etc.* You'll learn more about these in subsequent computer science courses.

## 4.2 Kinds of Knowledge

A first programming course is in some sense an almost impossible task. In one semester, you'll be asked to learn several different kinds of knowledge:

1. How to use the computers and the software on them
2. The grammar, punctuation, and vocabulary of a programming language
3. How to analyze a problem and design a program to solve it, so that the program is both correct and easy to write, read, modify, and repair

4. How to plan your time, and what sequence of steps to go through, in designing, writing, testing and debugging a program

5. Domain-specific knowledge (*e.g.* to write a program that draws geometric shapes on the screen, you have to know something about geometry.)

It is easy for a student (or a professor or a textbook author, for that matter) to get caught up in the details of the first two at the expense of the rest. I urge you *not* to fall into this trap, because the specific kinds of computers and software, and to some extent the language, you learn this semester will almost certainly be obsolete by the time you leave Adelphi. The much more interesting and lasting knowledge is at levels 3, 4, and 5 (and I'll try to minimize the time we spend on level 5 because it's not specific to computer science). In short, although all five kinds of knowledge are necessary in order to write a good program, I'll try to concentrate on levels 3 and 4.

## 4.3 Language and design recipes

Programming a computer requires that, to some extent, you learn the computer's language. Computers "understand" a lot of different languages, and the choice of language affects how you approach programming.

For this semester and next, we'll be using the popular Java programming language, and the BlueJ development environment, which was designed for first-year programmers and which you can download for free at `http://www.bluej.org`.

Throughout the semester, we'll pay a lot of attention to **design patterns**, which are step-by-step "recipes" for getting from a vague English-language description of a problem to a working computer program. Every year, some students skip the recipes when they're in a hurry, and invariably find themselves wasting more time as a result. To prevent this, **you will be graded** on, among other things, how well and thoroughly you use the recipes.

## 4.4 Knowing your Tools

If you wanted to learn carpentry, you would start by studying the characteristics and capabilities of each of the common tools carpenters use, so you use the right tool at the right time for things that it does well. In programming, although you have editors, compilers, etc. to help, your main tool is your mind; accordingly, it makes sense to study the characteristics and capabilities of your mind. Indeed, it can be studied and measured just as scientifically as a bouncing ball in Physics class. By the end of the semester, you should have a much better idea (backed up with hard numbers) of how *you* work as a programmer, and hence the ability to accurately estimate how long *you* will need to complete a specified programming task. This

allows you to plan ahead and complete assignments on time — an invaluable skill for a professional programmer, and applicable to the (non-programming) rest of your life as well!

# 5 Texts

We're teaching this course a little differently this semester, and haven't found a textbook that exactly matches what we want to do. Dr. Bloch's book *Picturing Programs* covers a lot of the same topics, but in the Racket language rather than Java. There are lots of books out there about the Java language; some of my favorites (in alphabetical order) are

- "Learning the Java Language" (free tutorial from the makers of Java, on the Web at `http://docs.oracle.com/javase/tutorial/java/TOC.html`)
- Barnes and Kölling, *Objects First with Java: a Practical Introduction Using BlueJ*, see `http://www.amazon.com/Objects-First-Java-Practical-Introduction/dp/0132492660`
- Gee, *Java – Objects First: an Introduction to Computer Programming using Java and BlueJ*, free on the Web at `http://www.people.okanagan.bc.ca/rgee/javabook.pdf`
- Horstmann, *Big Java*, see `http://www.amazon.com/Big-Java-Compatible-Edition-ebook/dp/B005HGFFL6`
- Sierra and Bates, *Head First Java*, available from O'Reilly at `http://shop.oreilly.com/product/9780596009205.do`

This is a 4-credit course: 3 hours/week of lecture and 3 hours/week of lab. The usual rule of thumb for out-of-class work is twice the lecture time, so you should plan on spending an additional **6 hours/week** on reading and homework. In particular, you'll need to read, on average, around **40 pages a week**, and the programming homework will take a significant amount of time. You are responsible for **everything in the reading assignments, whether or not I discuss it in a lecture.** You are also responsible for checking the class Web page and Moodle page at least once a week or so; I often post assignments, corrections to assignments, solutions to assignments, *etc.* there.

# 6 Grading

Your grade will be computed from several sources, broken down approximately as follows:

- (**40%**) There will be 5–8 major programming assignments, to be turned in by e-mail, graded in detail and returned to you by e-mail. Within a week after I've returned the graded assignment, you may turn in a "second chance" version. Your original grade will be averaged together with 80% of the "second chance" grade, so if you've only made a small improvement, or if you got at least an 80% the first time around, you can only hurt yourself by re-submitting so don't bother. Note also that the last assignment will be too close to the end of the semester to allow for a "second chance".

  A major programming assignment turned in with a time stamp after midnight on the due date is late, and will receive partial credit depending on how late it is. Anything turned in after midnight on May 8, the last day of class, will be ignored.

- (**20%**) There will be 27 lab sessions, with (probably) a lab programming assignments for each one. These will be much shorter than the major programming assignments. You'll either show these to Dr. Stemkoski in lab, or turn them in to us before midnight on the same day. They will be graded as either "satisfactory" or "unsatisfactory"; don't bother turning them in late.

- (**15%**) There will be 5–10 brief (15–20-minute) quizzes, to be done on paper; we'll grade them and try to get them back to you at the next class meeting. They cannot be made up; if you're not there the day of a quiz, you have a zero on it.

- (**20%**) There will be a two-hour final exam from 10:30-12:30 on May 18.

- (**5%**) We'll assign "brownie points" based on such considerations as classroom behavior, asking or answering good questions, helping your classmates without stepping over the line into cheating (see below), etc.

The final exam must be taken at the scheduled time, unless arranged in advance or prevented by a documented medical or family emergency. If you have three or more exams scheduled on the same date, or a religious holiday that conflicts with an exam or assignment due date, please notify me in writing within the first two weeks of the semester in order to receive due consideration. Exams not taken without one of the above excuses will be recorded with a grade of 0.

# 7 Program standards

Every program must contain, in the first few lines, a comment indicating the name(s) of the student(s) working on it and which assignment it is. Programs not containing this information, clearly visible, will get a zero.

Every program must be accompanied by test cases, so I can see how it actually works. Programs with inadequate or poorly-chosen test cases will lose points (we'll discuss how to choose good test cases); programs turned in with no test runs at all will lose *lots* of points.

Having done my share of programming, I know that sometimes you hit a brick wall and cannot get the thing to work for love or money. If this happens, turn in the program together with a detailed description of **how** the program fails, **what you've tried** in your attempts to fix it, and **what went wrong** in those attempts, for partial credit. Note that "how the program fails" does *not* mean saying "I got an error message": you need to tell me **which** error message you got, **when** you saw it, and **what** you think the error message means. Similarly, if the program fails by producing wrong answers, you need to tell me **when** it produces wrong answers (are they *all* wrong, or just in a few cases?), **how** they are wrong (*e.g.* are all the numbers consistently higher than you expected, are they the negatives of the correct answers, or are they all over the place with no apparent pattern?), and your speculations on **how** such an error might have arisen. I'm requiring all this not because I'm mean and horrible, but because by the time you've written all this down, you may have enough information to actually *fix* the problem, which is much better than turning it in incomplete.

I also expect you to maintain a log of what kinds of errors you encountered, how you discovered them, how long it took you to fix them, and what the actual problem was. This log must be turned in with each homework assignment. I've written some Web-based forms to make it easy to record this stuff, or you may just keep track of it yourself.

# 8 Ethics

Most homework assignments in this course involve writing, testing, and debugging one or more programs. Some of these programs are to be written individually; for others, you are to work in teams of two students, switching teams from one assignment to the next, if at all possible. (If you have a really terrible schedule and can't get together with a partner, talk to me and we'll arrange something.)

When I say "teams of two students", I don't mean "you write the first half of the assignment, and I'll write the second half"; I want *both* students working *together* on *all* of the assignment, using the techniques of Pair Programming (on which I'll give you a reading assignment). I expect people to switch partners from one assignment to the next, so you get experience working with different people.

It's hard to define what constitutes "cheating" in this sort of course. Students are encouraged to help one another with level-1 and level-2 difficulties ("how do I save this file?", "what's the syntax for defining a `struct`?", *etc.*), regardless of whether

they're on the same team, but designing, coding, testing, and debugging should be done by the one or two people whose names are on the assignment.

It's remarkably easy for a professor to notice when three different teams have turned in nearly-identical programs; if that happens, I'll grade it once and divide the credit among the three, so the best any of them can hope for is 33%. I don't try to figure out who copied from whom; it is *your* responsibility to not let anyone copy your homework. Among other things, that means don't leave it on the "Universal Share" drive, because anyone at Adelphi can copy it and even delete it.

All work on the final exam and the quizzes must be entirely the work of the one person whose name is at the top of the page. If I have evidence that one student copied from another on an exam or quiz, *both* students will be penalized; see above.

# 9 Schedule

This class meets for lecture every Monday, Wednesday, and Friday from 10:00–10:50, and for lab every Monday and Wednesday from 2:25–3:40. Both lab and "lecture" (although we hope to do very little lecturing) are in Swirbul 100.

All dates in the following schedule are tentative, except those fixed by the University; if some topic listed here as taking one lecture in fact takes two lectures to cover adequately, or *vice versa*, the schedule will shift. I'll try to keep this information up to date on the class Web page.

When I say "read" above, I mean an active process, involving not only the textbook but pencil, scratch paper, and a notebook for writing down key points. Finally, and perhaps most importantly, you'll need a computer for trying out the new ideas you find in your reading. Just as you cannot learn about cooking or driving a car just by reading about it, you cannot learn about programming just by reading about it. In short, *every* time you read about a new programming idea, *try it!*