

**Workshop title:** “Teach Scheme, Reach Java: Introducing Object-Oriented Programming Without Drowning in Syntax”

**Brief Abstract:** Participants will use the DrScheme platform to solve several CS0/CS1 problems in Scheme, including function composition and re-use, conditionals, structures, and (if time allows) recursion on linked lists, and/or event-driven interactive graphic programming. We shall then demonstrate how the same topics can be covered in a more mainstream, but more complex, language such as Java. Participants will be invited to a week-long, NSF-funded workshop in Summer 2009, which goes into much more depth on technical, classroom, and curriculum issues.

**Presenter’s background:** Stephen Bloch has taught beginning programming for fourteen years, including both CS1/CS2 for majors and a first-and-last programming course for non-majors. He was introduced to the present approach at a 1998 workshop by Matthias Felleisen, and first implemented it in 1999-2000. Since then he has applied the same principles variously in Scheme, Java, and C++ (hint: it’s easiest in Scheme, and hardest in C++). He has led several NSF-funded week-long workshops to train high school and college faculty in these techniques, and is the PI for a current NSF grant which funds similar workshops in 2007-2010.

**Intended audience:** High school, College and University faculty who teach first-year programming courses for either majors or non-majors. Students with an interest in teaching are also welcome.

**Philosophy:** A first course in computer programming should not be about the current “hot” language in industry – which may be obsolete by the time today’s freshmen graduate – but rather about lasting, transferable concepts and practices of good programming. Yet beginning programming students spend much of their time wrestling with the language, and often mistake that as the subject of the course. The programming language distracts from the course material; on the other hand, students need a real language to write real programs that really run on real computers.

We resolve this dilemma by starting in a language with simple, consistent syntax and semantics, currently a subset of Scheme (omitting I/O, assignment, sequence, higher-order functions, and local definitions). The development environment enforces this subset, and gives error messages appropriate to the current subset, but allows students as they outgrow each subset to advance to a larger one with a few mouse clicks. Students become comfortable with fundamental programming concepts — variables, function composition, function definition, parameter passing, data types, design for reuse and modifiability, conditionals, fields (“type definition by parts”), polymorphism (“type definition by choices”), self-reference and recursion, functional abstraction, etc. — in this sheltered environment before encountering the same concepts in the more bewildering world of Java, C++, etc.

Simultaneously, students are inculcated with a step-by-step **design recipe** for software development. Each step has concrete questions and products, so students always know how far they've gotten and what to do next:

- 1) Identify the **purpose, inputs, and outputs** of the program (function, method, whatever) to be written;
- 2) Identify (and, if necessary, define) **data types** relevant to the problem at hand;

- 3) Write **examples** or **test cases** of how the program will be invoked, in legal syntax and accompanied by expected results, using the data types from step 2 as a guide;
- 4) Write a **program skeleton**, the syntax to define a function with the name and parameters chosen above;
- 5) Write an **inventory** of available and likely-to-be-needed expressions, based on parameter names and their data types;
- 6) Choose and combine items from the inventory to form a complete program **body** (the hardest part, but in practice step 5 often does most of the work);
- 7) **Test** the program by running it on the examples from step 3.

We emphasize data types throughout, not only as a fundamental concept, but as an invaluable tool in coding. To every data type correspond both a natural *coding pattern*, which (in step 5) provides a rough draft of the code and helps students avoid “blank page syndrome”, and a natural *testing pattern*, which provides guidance in building test suites (in step 3). In particular, recursion is introduced as simply the application of already-learned coding patterns to a self-referential data type. The concrete methodology also provides a handy grading rubric, which communicates to students that *every* step matters, not only the coding.

For non-majors, it's enough to convey the important programming concepts and methodology in one language. For CS majors, the course switches from Scheme to Java late in the first semester or between first and second semesters. The Java stage is not independent, but builds on and reinforces the programming concepts and methodology already learned in Scheme, with explicit discussion of similarities, differences, and the continued applicability of the concepts and methodology. Students learn to apply the same design recipe (including what Kent Beck calls “test-driven development”) in Java that they’ve been using in Scheme. The result is a student who, after a year of coursework, can approach programming problems in a principled manner (not “hack it until it works”), with understanding and perspective.

Note that although we are currently using Scheme as a first language and Java as a second, the approach is applicable to other languages. Whatever the language, however, we believe the first exposure to programming should be *functional* rather than imperative/sequential/procedural: not only do functional programs have simpler semantics, relying on the familiar model of algebraic expression evaluation rather than a load/store machine model, but it is enormously easier to write test cases for functional programs than for stateful ones. Stateful testing, along with stateful programming, can be introduced late in the first semester after students have thoroughly internalized functional techniques.

The workshop will include discussion of how to implement both the Scheme-based introduction and the Java-based followup, as well as feedback and results from instructors who have implemented our approach in their own classrooms.

**Presenter Contact:** Stephen Bloch  
Math/CS Dept  
Adelphi University  
Garden City, NY 11530  
Phone 516-877-4483  
Fax 516-877-4499  
E-mail [sbloch@adelphi.edu](mailto:sbloch@adelphi.edu)