Teaching Programming

Avinash Kak
Purdue University
kak@purdue.edu

First posted: July 2008; Revised (minor updates): April 2016

Abstract

Computer science educators have generally gravitated toward teaching programming using easier languages of the day under the assumption that the students would be more receptive to them. But are the students' long-term interests really served by this approach? Doesn't using the so-called easier languages for teaching programming amount to teaching serious musical composition with a banjo? Even more fundamentally, what do we mean by teaching programming? It is important to explore these issues given the critical role played by software in all facets of human existence today.

1 Introduction

Just as natural languages anchor human thought, programming languages anchor our understanding of the world of computing. So if you want to understand the potential and the limitations of the smart systems of the future, you must first come to terms with the basic vocabulary of such systems. Although there are many levels of abstraction in this vocabulary, at its base it is rooted in the nouns, the verbs, and the other qualifiers of the programming languages.[1]

Given the importance of programming languages, one would think that they would be the centerpieces of the course offerings from our educational institutions. Unfortunately, that is often not the case. Teaching a programming language — as it really should be taught — is considered to be too burdensome by most.

[1] This paragraph is from the introduction to my "*Objects Trilogy Project*" at `https://engineering.purdue.edu/kak/ObjectsTrilogy.html`. **The 17-year long project is now complete with the publication of *Designing with Objects*.**

It is, of course, true that every institution that offers computer-related education teaches programming in one form or another. At one end of the spectrum, we have educational institutions that start off the students with languages like C and then build up the programming expertise step-by-step to higher level programming and scripting languages. At the other end of the spectrum, we have educational institutions that take the path of least resistance and teach programming through a language that the professors believe would be found least difficult by the students.

It is the latter educational institutions that immediately jumped on the Java bandwagon when it first burst on the scene several years ago. The professors fell for Java's straightforward syntax (no pointers) and decided that this language was best suited for introducing students to programming and for explaining notions in data structures, control structures, GUI design, etc. I believe that these institutions are responsible for a large number of programmers today who are terrified of dealing with *real* programming issues that all too often consist of memory leaks, buffer overflows, concurrency problems, run-time exception handling, performance optimization issues, debugging, testing, etc.

2 What Does it Mean to Teach Programming?

Until about 25 years ago, this question was a no-brainer. During the past quarter century, the world of computing and programming has become so diverse and varied that it is no longer easy to define as to what constitutes core programming skills. While almost everyone today with a science or engineering degree — and many with other types of degrees as well — learns to program computers in one form or another, it is rather common to encounter graduating engineers and scientists whose programming competence is limited to scripting in Matlab and other such languages. In case the reader thinks that this state of affairs is limited to non-computer science schools, do realize that there now exist many CS schools where most of the programming skills are imparted in Java or some other easy-to-use language.

Obviously, I do not mean to imply by any stretch of imagination that scripting languages like Matlab and programming languages like Java do not deserve a place under the sun. These are all great languages — as evidenced by the reception they have been accorded in the marketplace.

To me, teaching programming in a university setting means, first and foremost, teaching how to interact with a computing machine and how to deal with all the surprises that a machine can throw at you — *notwithstanding the availability of languages capable of insulating a programmer from many*

of those surprises. While it may make good sense to develop a commercial application using a language that has type safety, range checking, and garbage collection built into it, those are exactly the features that make a language unsuitable as the primary medium for programming instruction.

More specifically, teaching programming means, of course, teaching the basic control structures of a language and the data structures it allows us to create for solving various problems. But teaching programming *at its core* must also include the art and science of implementing efficient strategies for memory management, for writing signal handlers so that a program can interact with the operating system, for input/output related to different types of data, for dealing with synchronization issues related to concurrency, for process control, etc.

Reading the above paragraph, many would say that only the systems programmers need to interact with a machine at the level stated in the paragraph. They would add that such details are unimportant for non-systems programmers. That may well be the case after a student has graduated and accepted a line of work that only requires, say, Matlab sort of programming. But when a student is being taught to appreciate and enjoy programming at school, that phase of a student's life ought not to shield him/her from what folks would commonly refer to as system-level issues.

My statement regarding what is meant by teaching programming at its core takes on added significance in this day and age when even the non-technical people have acquired the rudiments of the vocabulary of computing. While general public may be a bit woolly-headed about what exactly a computer program is and how exactly it works, many of these folks are quite comfortable with such computer jargon as bits, bytes, memories of various sorts, etc. Some of these folks can also talk about the processes running in their machine — things that they see when they press the very familiar triple keys CTL-ALT-DEL. If an application seems to hang, these folks feel comfortable with bringing up the list of processes running in their computer with CTL-ALT-DEL, identifying the offending process, and then deleting it. *When it has become so common for ordinary people to interact with a computer at what is obviously the system level, that makes it all the more important for real programmers to understand the system level interactions between a program and the machine.*

Teaching programming also means talking about and demonstrating program misbehavior caused by poorly written code and getting the students excited about developing programming skills that result in good code. However this cannot be accomplished when the medium of instruction consists of a language that does not allow you to make mistakes.

3 Using Music Instruction as a Metaphor for Teaching Programming

Programming is much more than just learning the syntax of a language, in much the same way as composing a piece of music requires much more than just mastering the chords one can play on a musical instrument. Notwithstanding the fact that software engineering now provides us with various approaches for creating good code, fundamentally it is still the case that as you create each new entity (variables, data structures, classes, and so on) in a program, at the back of your head you constantly think about how it will interact with the entities you created previously and the entities yet to come. That is, as you lay down each construct in a program, you can't help but worry about how it will harmonize with the rest of the constructs, those already in place, and those yet to be thought of.

4 What Makes Teaching Programming Difficult

If teaching programming meant just explaining the syntax, the job would be easy, but also extremely boring for the students (as any student who has been taught programming by a non-programmer would testify). To instill the joy of programming in a student, the instructor must demonstrate interesting cases of what happens when good practices are not adhered to. For example, by deliberately creating large memory leaks in a loop construct, it would be easy to show how even a powerful computer can be brought down to its knees as it runs out of fast memory. Along the same lines, by deliberately overrunning the buffer, either on the stack or on the heap, it would be easy to demonstrate how an otherwise innocent looking piece of code can produce unexpectedly wrong answers.

But this kind of teaching can only be accomplished by someone who is very much involved with programming and software development. Although people of that ilk are plentiful in industry, they are rather few and far between in academia. It is a fact that much of the educational enterprise — especially in our elite universities — is driven by the needs of the researchers and most of the researchers are preoccupied with pushing the state-of-the-art in their own specialties. While these researchers do want their graduate students to write good code, they themselves do not have much of a stomach for teaching the art and science of creating good software.

5 So Where Do We Stand Today?

Today there exists a dichotomy in our universities: On the one hand, we have young people — a majority of whom with no appetite for research-focused advanced graduate degrees — hungry for programming skills that would enable them to compete in the global computing enterprise, and, on the other, a large majority of professors who have no time or desire to impart to the students those sorts of skills.

6 And, Finally, a Quote ...

Considering that this essay is on the teaching of programming languages, the following quote from Lecture 27 of my popular "Lecture Notes on Computer and Network Security" seems appropriate:

> ".... in my opinion, the individuals who bring us languages that come into widespread use are the modern deities and prophets. Obviously, hundreds if not thousands of people make important contributions to the maturation of these languages. Nonetheless, the primary credit must go to the individuals who first conceive of them and then shepherd their subsequent evolution. This pantheon obviously includes Dennis Ritchie for C, Bjarne Stroustrup for C++, James Gosling for Java, Larry Wall for Perl, Guido van Rossum for Python, Tim Berners-Lee for HTML, Rasmus Lerdorf for PHP, and several others."

About the Author

Avinash Kak is the author of *Objects Trilogy*, with all three books in the trilogy published by John-Wiley. The first, **Programming with Objects**, was published in 2003; the second, **Scripting with Objects**, in 2008; and, the last, **Designing with Objects**, in 2015. The third book is an attempt at explaining the more difficult design patterns with the help of short-story snippets from the world of Harry Potter. He is a professor of electrical and computer engineering at Purdue University.