The Use of Java as an Introductory Programming Language

by Jason Hong

Introduction

During the fall of 1996, the Georgia Insitute of Technology initiated the transition from teaching procedural programming to teaching object-oriented programming at the introductory level. Our main goal was to produce an introductory class that taught solid programming and design skills, using Java as the programming language. We will look at some of the problems we encountered, as well as some of the issues involved in using Java to teach students object oriented programming at the introductory level.

Our introductory programming curriculum is divided into two consecutive courses. The first class, *Introduction to Computing*, deals with such fundamental concepts in computing as abstraction, modularity, algorithms, dynamic data structures, and complexity theory. Implementation is done in a formalized pseudocode only, with no compilation at all. The idea behind only using pseudocode is to allow students to understand, design, and implement the underlying logic of a program without having to fight a compiler at the same time.

The second class, *Introduction to Programming*, is a more traditional programming course. Actual programs are constructed, with emphasis placed on the concepts introduced in the previous course. Reusability, readability, and documentation are also strongly stressed in this class. In this article we will focus on this second class, Introduction to Programming.

Why Java?

In the spring of 1996, there were many discussions between faculty members and the teaching assistants (TA's) for our *Introduction to Programming* class as to whether or not we were adequately preparing our students for the future. We had been teaching procedural programming using Pascal as our introductory language for over ten years. Our primary concern was whether or not students were learning appropriate and useful skills about modern programming practices and technologies.

We felt we needed a language that had modern programming practices built-in. Some of the technologies that were lacking in Pascal include safe references (as opposed to pointers), garbage collection, and exception handling. More importantly, while Pascal is a procedurally based language, we felt that objects were a better design paradigm. Bertrand Meyer suggests that ``if you agree with its [object-oriented designs'] goals and techniques, there is no reason to delay bringing it to your students; you should teach it as the first approach to software development" [3].

We also needed a practical programming language. For some students at Georgia Tech, the only courses in which they learn about designing and constructing programs are the introductory courses we offer. Some of these students may have to construct programs in other courses at Georgia Tech. These students may also be involved with the design of computer systems later in their careers. For these reasons, the programming language we use had to be of general purpose, easily available, and widely used. Java fit most of these requirements quite well: it is a relatively simple language; it has most aspects of object-oriented design; it has several modern technologies designed into the language; and it has also become a marketable skill.

Current Infrastructure

Georgia Tech is on the quarter system, which means that the course lasts for a total of eleven weeks. We have around 25 TA's to handle approximately 200 students per quarter. Some TA's handle multiple lab sections, while others grade the programs submitted by students. Some of these TA's even handle managerial responsibilities exclusively (such as creating programming assignments, creating quizzes, answering questions on the class newsgroup, and setting the course topics).

The students are given programming assignments every week, consisting of program requirements, sample data sets, sample answer sets, and a sample demo. There is also an optional extra program that can replace the lowest program grade. Students are given two weeks to complete both the design and construction of each program. A proposed design for each programming assignment must be submitted a week before the assignment is due (because of scheduling constraints, sometimes there is an overlap between when one program is available and when the next program is due). This design consists of a high-level overview of what classes will be created, what data structures each object will have, and what methods each object will have. These design documents are essentially skeleton programs with full documentation.

In order to give students more exposure to programming concepts and to the programming language, weekly labs are used to supplement these programming assignments. A few simple concepts are explained in the first part of a lab through sample source code that can be compiled and executed. The other part of a lab reinforces these concepts by having the students complete a skeleton program, which is later submitted and graded by a lab TA. However, to reduce the burden on teaching assistants, we have been working on implementing an autograder for the labs submitted by students.

The Transition Plan

It should be noted that there is a subtle but significant difference between creating a course that teaches programming *with* Java, as opposed to programming *in* Java. Our overall goal was to produce a course that taught the fundamentals of programming using Java as the implementation language, as opposed to a course that taught the specifics about programming in the Java language.

The plan to move from Pascal to Java started with a pilot program of a few students and TA's learning about Java. Our pilot class consisted of experienced programmers taken from our regular introductory programming course. The primary goals were to gain more experience with the Java language, and to create a first draft of the topics and lectures that would be used to introduce Java to the other students. This phase took one quarter to accomplish.

The next part of the plan was to scale up and have a few sections from our regular course using Java. This time, we had two class sections (about forty people) assigned to using Java as the instructional language. This phase took two quarters to accomplish.

The last part of the plan was to have the entire course using Java. The entire transition period took one academic year, and we are now teaching our *Introduction to Programming* course using Java as the implementation language.

A slow transition was necessary due to two constraints: the number of Java-enabled computers available, and the number of TA's that were proficient with object technology and with Java. The hardware requirements to run a Java Virtual Machine are fairly steep. Most of our PCs had to be upgraded before enlarging the size of the class. Multi-user computers would not have worked well with a large class since many students would likely be compiling and running Java programs at the same time.

Another problem we had was that few TA's had significant experience programming in Java. To address this problem, we initiated two classes to instruct our teaching assistants in using Java. These classes consisted of biweekly programming assignments with a weekly recitation.

Problems Encountered

We have encountered several problems while teaching Java as an introductory language. Some of these problems have to do with the syntax of Java. Others are related to the current state of Java tools and resources. However, the majority of the student difficulties are conceptual problems in learning how to design and construct a program in general. We will address syntax problems first.

Input and Output

In Java, output can be created simply with `System.out.println()`. However, for input you either have to use `System.in.read()`, or create an instance of a `DataInputStream` or an `InputStreamReader`. In either case, you will have to catch a possible `IOException`, which would involve teaching the students about exceptions in order to do something as simple as reading in a single character.

We avoided this problem by providing drivers to the students for the first few programming assignments. We provided the Application Programming Interfaces (API's) that the students must have in their programs in order to use our drivers. Essentially, the design for the class is already done. All the students needed to do was to write the code for a Java class, and the driver would create an instance of the class and test its functionality for the student. The results were threefold. First, the students were given more time to learn basic syntax without having to fully understand higher level concepts at an early stage. Second, the students could still do input, but we deferred until later teaching students how to do something as simple as reading in a number. These drivers handled all the input required for the first few assignments. Finally, when the students had to handle their own input, they had several examples on which to base their work.

We also created our own set of routines to read input from both the standard input stream and from files. We did not want the students to have to deal with the low-level details involved in something as conceptually simple as input, and so we abstracted it for them. The routines we wrote allowed students to read in data (such as characters, integers, lines of text, and words of text) relatively easily. These routines also caught all possible exceptions, so students did not have to handle exceptions at an early stage.

AWT

Java's Abstract Windowing Toolkit (AWT) presents another problem. Using the AWT involves learning about many concepts and unnecessarily adds another level of complexity. The programming assignments are designed to instruct students about specific concepts, and AWT hampers inexperienced students from completing the required tasks. Also, AWT is Java specific, and our primary goal is to teach programming and design skills using Java as our programming language, not how to program in Java. Our solution was to restrict the use of AWT until the end of the quarter, and only then confine its use to relatively simple programs.

Java Implementation

One problem faced by anyone new to a programming language is the cryptic error messages provided by compilation errors. Unfortunately, most implementations of Java continue in this tradition with such brief error messages as ``Can't make forward reference'' and ``Statement expected''. However, in some implementations of Java (e.g. Sun's JDK), we have discovered that all of the possible error messages are stored in a file called *javac.properties*. This file can be edited not only to provide more useful error messages, but can also direct the students to resources which describe typical reasons why the error would occur, some examples demonstrating the error, and most importantly, tips on how to fix the error. We have been compiling a list of common errors and sample solutions in order to effect this change.

Resources Available to Students

The fact that there are so many Java resources accessible to students is both a benefit and a hindrance. Java source code is freely available at many Internet sites, which can assist the students in learning about Java, but it also produces a strong temptation to plagarize. There is also no guarantee that the source code they examine contains good programming practices! Another set of resources is the Java newsgroups, which can assist students in trying to learn Java. Far too often on these kinds of newsgroups, though, a student posts a message asking how to do his or her homework. The best policy in both of these cases is to make it clearly known, in advance, how each resource can and cannot be used.

One serious problem we have encountered is that several Java decompilers are freely available on the Internet. Compiled class Java files can be decompiled into listings nearly identical to the original source code. This fact changes the way that we give compiled class files to the students.

The demo program that we provide to the students can be decompiled if the compiled class files were made available to the students. To avoid this problem, we simply made a script that executes the demo program, which is hidden in another directory. However, this strategy does not work with demo applets available on the web. In this case, it is probably best to make the applet difficult to reverse engineer through some kind of obfuscator.

We planned to have the students reuse their own components created earlier in the quarter. If a student does not complete an assignment, compiled class files with full API's can be given out (at a slight penalty). The fact that decompilation exists, though, has altered these plans. Currently, we do not give out any class files, but we do make the source code available after the program has been submitted.

Books

We have not found a good book about programming with Java at the introductory level. We have evaluated dozens of texts, and nearly every one is about how to program in Java, not how to construct programs using Java as the programming language. A few books even jump directly into using AWT. This makes it much more difficult for the students since they are busy fiddling around with AWT instead of trying to solve the problem. In addition, these books rarely emphasize good design, good object-oriented techniques, reusability, and solid software engineering techniques. As a result, we have been forced to create a whole set of class notes for the students to use.

One common problem in all of these books is the terminology used to describe the way variables are passed into methods. The typical terms used are "pass-by-value" and "pass-by-reference." Java is passed-by-value with respect to primitive data types. However, some books describe objects as being passed-by-reference, while others refer to the reference itself as being passed-by-value. While both are technically correct, we have abandoned using the term "pass-by-reference" for objects, as we have found the term "pass-by-constant-reference" easier for students to understand. While the term may not be accurate in terms of the actual implementation, we have found that it is a good conceptual model for the students.

Conceptual Problems

Perhaps the most difficult problem that students face with Java syntax is that there are so many concepts to grasp at the same time. For example, method `main` is declared as:

```
   public static void main (String[] argv)
```

This deceptively simple statement involves the concepts of methods, visibility modifiers, class and instance methods, return types, method names, formal parameters, and arrays!

We attacked this complexity problem in four different ways. First, part of our problem was overcome by the fact that all students were required to have taken the *Introduction to Computing* course, which meant they should already understand subroutines, visibility modifiers, return types, arrays, and formal parameters.

Second, on the first day of class, this method header was shown to the students. It was simply explained as something that they must have, word for word, in order for their programs to execute properly. As we progressed through the quarter, a few more of the concepts were exposed. Eventually all of the parts involved become clearer.

Third, we avoided this problem temporarily by providing a skeleton program for the first lab and first programming assignment, with the students simply filling in the blanks. This way, students gain experience with basic Java syntax without having to deal with more advanced concepts.

Lastly, and most importantly, method `main` was contained in the drivers we provided to the students. This way, the students were aware that they must have it in their code, but we handled it for them the first few times.

Primitives and References

Another conceptual problem lies in the differences between primitive data types and references. For example, what are the differences between an `int` and an `Integer`? Both of them hold numbers, so why are there two ways to do what appears to be the same thing? The task a student wants to accomplish is to create a variable that holds a number. However, one is a primitive data type while the other is a reference to an object. Few students will initially understand the subtle distinction between the two, and introducing these two concepts at the same time could cause confusion. Our solution to this problem was to use only primitive data types for the first programming assignment. References were explained later, with both references and primitive data types used for the second and subsequent programming assignments.

Class versus Instance

Our experience has shown that two of the most difficult concepts for students to understand are the difference between a class method and an instance method, and between a class variable and an instance variable. Perhaps the most common compilation error a person new to Java will see is:

```
Can't make static reference to method aMethod() in class aClass.
```

We restrict the students' use of the keyword `static` to constants and in the method `main` until the concept of class methods and class variables has been properly introduced. Even then we restrict their use to a few specific cases. If class methods and class variables are introduced too early, before students have a grasp on the basics of Java syntax, students will typically add the keyword `static` to remove the static reference compilation error, thus compounding their problems.

Design

Our belief is that the greatest problem with using Java as an introductory language is not the use of Java itself, but the design of good programs. In past courses using Java, most of the students created programs that were poor in terms of modularity, coupling, abstraction, and reusability. This is not a problem restricted to object technology, as poor designs were still prevalent while we were using Pascal. Some students seem to have a natural talent for designing good programs, while others struggle with it. Some students acquire good design skills after a lot of practice, while others lag behind. Is there a way to teach students how to design good programs? What can we as educators do better to teach the students how to design good programs?

We have two ideas to improve the designs that students create:

- Show students examples of good designs and good programs, so that they can learn by example. Essentially, we would like to show them what a good program is instead of telling them what a good program is. We want to show them how programs are constructed instead of telling them how to construct good programs.
- Improve the processes used by the students to create their designs

Examples of good design

One way we are doing this is by constructing programs during lecture, where the lecturer walks students through a simple program. More importantly, however, the lecturer interacts with the students about what decisions are being made and why they are being made. Students can see how an expert programmer breaks down a problem and accomplishes tasks. The goal here is to have the students emulate the way an expert programmer thinks and performs.

We are also showing students how to construct programs by having them do small case studies. Similar to the approach taken at UC Berkeley [1], these case studies consist of multiple approaches to solving the same problem. Students evaluate these different approaches at multiple levels. In the beginning, we want them to focus on readability and understandability. Later on, we want them to focus on such characteristics as quality of design, amount of abstraction, and reusability. One approach may be better than another, but the most important question to ask is: *Why* is one approach better than another?

Lastly, we are also providing students our source code for the sample program after they turn in their assignment, so that they can see how an expert programmer approached the same task. Although this forces us to create new programming assignments every quarter, we believe that it is very worthwhile for the students.

Improve design processes

As educators, we typically grade only the program itself, the end-product of a student's efforts, rather than the process that was used to create it. In other words, we are only seeing a text file that contains source code listings. What we are missing is how much effort was put into understanding the concepts, analyzing the requirements, creating a solid design, implementing the design, and debugging and testing the implementation. Without this information, it is extremely difficult to assess how well the students are progressing, how well the students really understand the concepts presented, and what things the students can be doing to improve.

Our experience is that novice students have a very poor process in constructing programs. Too many students do not take enough time to understand the concepts, understand the requirements, and create a design. Typically, the first thing a student does is start hacking out code! Thus, they spend a disproportionate amount of time in a vicious cycle of coding and debugging.

Our goal is to have the students be more cognizant of their own programming process. To accomplish this, we have created a system tentatively called the Educational Software Process (ESP). Loosely based on Watts Humphrey's Personal Software Process (PSP) [2], it has been specially tailored and enhanced for a student population. The key point behind ESP is that there are differences between novice programmers and expert programmers in the mental models and processes used. ESP tries to make novice programmers more aware of these mental models and processes in two ways.

The first way is by teaching them industry best practices. There are specific things that expert programmers do that novice programmers do not necessarily do. For example, if an expert programmer does not understand a concept, then they typically create a small test program to make sure the concept matches their mental model. However, this is not something that is taught in a class. This is something that the expert either re-created on their own or was taught by another programmer. What we are trying to do is determine specifically ``what things do expert programmers do that make them experts?'' Can we gather these things and teach them to the students?

The second way ESP tries to improve the students' skills is by having them assess their own processes. Where did they spend most of their time? What were the major problems they faced? What caused these problems? What can they do to fix these problems? We want the students to be aware that it is not a problem to make mistakes at this stage of their careers, but that they must also learn from their mistakes and avoid them in the future.

Conclusions

Successfully transitioning to using Java at the introductory level involves a great deal of time and effort. Our advice for anyone considering using Java at the introductory level is as follows:

- Make sure that you are teaching programming **with** Java, not programming **in** Java. Programming languages come and go, but the fundamentals do not.
- Keep things simple for the students. Objects are a complex technology, and Java does not shield novices from it. Concepts need to be developed slowly for the students.
- Make sure you have someone who really understands Java, and who really understands object design on your team. You don't want to teach procedural design using Java.
- Be extremely flexible with the students the first few times around.

Java presents some significant problems as an introductory programming language. Some of the problems will be solved as the language, its tools, and the resources for it mature. However, the fundamental problems involved in using Java have nothing to do with the programming language at all. How do you teach students good design and evaluation skills? How do you teach good program construction skills? How do you instill in students good processes in constructing programs? For these problems, there is no clear and easy solution. However, we suggest that **showing** students how to create programs is much more effective than **telling** them how to create programs, and that students need to be made more aware of what they do when they construct programs.

Acknowledgements

Thanks to all of the students that went through GA Tech's *Introduction to Programming* course; and to Rich LeBlanc, Russ Shackelford, and all of the teaching assistants that made it happen. The class website for Georgia Tech's Introduction to Programming is located at http://www.cc.gatech.edu/classes/cs1502/

References

1

Clancy, M.and Linn, M. 1992. The Case for Case Studies of Programming Problems. *Communications of the ACM*. 35, 3 (March), 121-132.

2

Humphrey, W. 1995. A Discipline for Software Engineering. Addison-Wesley.

3

Meyer, B. 1993. Towards an Object-Oriented Curriculum. *Journal of Object- Oriented Programming,* (May).

Jason Hong (jasonh@cs.berkeley.edu) is a first year graduate student in Computer Science at the University of California at Berkeley. His research interests lie in Software Engineering and Human Computer Interaction, specifically in mobile and ubiquitous computing. Jason received his B.S. degree in both Computer Science and Discrete Mathematics in 1997 from the Georgia Institute of Technology. He was an instructor for the Introduction to Programming course in Spring of 1997.