Teaching Java –with OO first

David Gries
Computer Science
Cornell University

1. Introduction

Good morning! I thank you for inviting me to give this presentation. It is a pleasure to be here.

This talk is aimed at educators, especially those who teach the first programming course, although I hope everyone will find it interesting and provocative. I was led to talk about the teaching of programming because of the debates that have been taking place on whether OO should be taught before procedural concepts —or even whether OO *can* be taught first! I have strong opinions on the matter, which are based on 41-years of teaching, research in the field of formal programming methodology, and ten years of experience teaching programming using Java. My ideas are not widely known, and I hope that this presentation will spur debate and change some opinions.

Here is a list of references on “OO first”: portal.acm.org/citation.cfm?id=1189136.1189183. A SIGCSE debate on the issue [i] was preceded and followed by much mail on the SICGSE mailing list, and Kim Bruce wrote a summary of the discussion [j].

But rather than summarize what others have said, I prefer to tell you directly my views on how to teach OO.

In preparation for this talk, I spent a great deal of time looking at Java texts[1]. It was an eye opener, and it caused me to review the principles on which I base my teaching of programming. So, I will begin by discussing *some* principles and give examples from the literature that violate these principles. These almost universal violations, I argue, contribute to the state of affairs in teaching programming —whether OO based or not.

2. Some pedagogical principles

Let me turn your attention to the excellent PhD thesis [h] of Michael Caspersen of Aarhus, completed in June 2007. This thesis presents research and conclusions on the teaching of programming that is based on sound theories of pedagogy and cognition as well as experience. It deals with teaching OO, but its findings are applicable in more general domains. It is the best study on this topic that I have seen.

[1] Throughout, *text* means *introductory programming text*, and *computer science* is abbreviated as *CS*.

Among other things, Caspersen discusses two important principles concerning teaching novices:

Principle 1: *Reveal the programming process, in order to ease and promote the learning of programming.*

Principle 2: *Teach skills, and not just knowledge, in order to promote the learning of programming.*

Teaching methodological programming skills cannot be done in a lecture or two. There may be lectures on stepwise refinement (or stepwise *improvement*, according to Caspersen), loop invariants, and OO design. But *every* lecture should deal in some way with programming skills —e.g. when the teacher develops first the spec of a method and then the method body, or discusses local variables and where they should be declared, or writes a class invariant for every class, or develops a suite of test cases in a JUnit class. The development of skill, rather than simply transfer of knowledge, should be our aim.

From my point of view, few texts treat methodological skills appropriately. They teach *programs*, not *programming*. They focus largely on knowledge, not skill. Indeed, I would claim that computer scientists in general do not think much about the programming process and have little idea about how to teach programming.

In this sense, the field of formal programming methodology has failed. All through the 1970’s and 1980’s research went on in this field, research that helped *me* learn how to teach programming skills, often in an informal manner but based on a formal theory of programming. The field has not embraced the idea, to say the least. And I find it a tragedy that most students in CS can graduate without ever having heard the term *loop invariant*, much less used it.

I won’t mention principals (1) and (2) again in my lecture, for I focus on how OO can be taught first. But rest assured that these principals are always in my mind when teaching. Here are a few more principles.

Principle 3: *Present concepts at the appropriate level of abstraction.*

We all believe that abstraction is important in CS, perhaps more so than in other disciplines. Abstraction is discussed in many places. Colburn and Shute [c] conclude that “abstraction through information hiding is a primary factor in CS progress and success.” For Burg and Thomas [d], abstraction is “arguably the most fundamental intellectual activity in the field of CS.” Gunter Gorz [e] discusses abstraction as a fundamental concept in teaching CS. One well-known text says that, “a good abstraction hides the right details at the right time so that we can manage complexity.”

I was co-author of the influential report *Computing as a discipline* [f], which discussed the three major paradigms, or cultural styles, used in CS: *theory*, *abstraction* (modeling), and *design*.

But, in spite of this talk, most texts do not use abstraction appropriately. For example, many texts describe variables and assignment in terms of computers[2]:

> "A variable is a name for a memory location used to hold a value of some particular data type."
>
> "When [the assignment statement is] executed, the expression is evaluated … and the result is stored in the memory location …."
>
> "The computer must always know the type of value to be stored in the memory location associated with a variable."
>
> "An object reference variable actually stores the address where the object is stored in memory."

This computer-centric view gives the impression that only a computer can execute a program. If students are taught hat only a computer can execute a statement, how can they learn to hand-trace execution?

More importantly, the introduction of computing concepts in terms of the computer can create unnecessary and confusing detail. Lest you think that I am alone in believing that a programming language definition should be computer independent, I quote from the definition of Algol 60 [g]:

> "The purpose of the algorithmic language is to describe computational processes. …
>
> A variable is a designation given to a single value.
>
> Assignment statements serve for assigning the value of an expression to one or several variables …. The process will in the general case be understood to take place in three steps as follows:
>
> 4.2.3.1. Any subscript expressions occurring in the left part variables are evaluated in sequence from left to right.
>
> 4.2.3.2. The expression of the statement is evaluated.
>
> 4.2.3.3. The value of the expression is assigned to all the left part variables, with any subscript expressions having values as evaluated in step 4.2.3.1.

There you have it. No reference to a computer.

[2] Quotes from texts are given without citation. These are representative of many texts. The purpose is not to denigrate individuals but simply to show the state of affairs.

The confusion really sets in when one describes objects and classes in terms of a computer. Discussions of pointers to objects in memory, heaps, and other implementation-related terms just confuse. For example consider this passage, taken from a text:

> "An object has its own unique identity, which distinguishes it from all other objects in the computer's memory … An object's identity is handled behind the scenes by the Java virtual machine and should not be confused with the variables that might refer to that object."

Far better is to have a model for classes and objects that rises above the computer, especially if that model can be based on an analogy to which students can relate. Later, I will show you such a model.

Principle 4: *Order material so as to minimize the introduction of terms or topics without explanation: as much as possible, define a term when you first introduce it.*

This seemingly obvious principle is extremely difficult to follow when teaching Java, and I cannot claim to adhere completely to it. I *do* believe that following it almost forces you to teach OO first, because almost every line of a program deals with a class or object. But even when teaching OO first, this principle is difficult to follow. In some cases, a "spiral" approach is needed. For example, one can introduce methods in a way that allows the student to start writing methods whose bodies are simple assignments or returns; later, one then explains methods in depth.

As an example of violation of Principle 4, some texts introduce applets before they discuss subclasses, so the students cannot understand the programs they are shown. In the explanation of an applet, the terms *extends* and *inheritance* are used without explanation. I have also seen texts that slip in a cast or two without ever fully explaining what casts are.

Principle 5. *Use unambiguous, clear, and precise terminology.*

Here is terminology that doesn't work.

Pointer and reference. Students do not know what a pointer is. Even after you explain, they still have difficulty understanding what an assignment statement b= c; means when c contains a pointer. With appropriate abstraction away from the computer, these terms are unnecessary.

Inheritance. In my opinion, the specification of Java makes a big mistake in its use of this term. According to the spec, private variables are not inherited, even though they appear in every object of the subclass! Only ones that are directly accessible are inherited.

I prefer to use the term the way it is used in English. A child may inherit a million dollars, but the terms of the inheritance may prohibit access to the money until the child is 21. In the same way, *all* fields and methods are inherited —they appear in every folder of a subclass. That they may not be referenced directly has no bearing on the issue of inheritance.

Formal parameter and actual parameter. These unfortunate terms were introduced in Algol 60. After introducing them, most texts tend to drop the adjective, resulting in ambiguity and confusion. Far better is *parameter* and *argument*.

One text says that, "The semantics of an assignment statement for primitive types and for objects is different," and another says that, "When an object is passed to a method, we are actually passing a reference to that object."

These are wrong and confusing —the confusion being that the authors think an object is assigned or passed to a method.

In summary, utmost clarity of terminology is necessary, for our choice of terminology can influence how easy or hard it is for students to grasp the concepts.

There are other principles that we should follow in teaching programming, but these will give you an idea of my general philosophy. Let me now turn to a consideration of teaching objects first.

3. Facilitating the teaching of objects first

Several ideas come together to facilitate the teaching of objects first. I don't have time to talk about them all, so let me just list them and talk about two of the more important ones:

- An IDE that eliminates the need for Java applications and that can be used to demo during lecture.
- A model of classes and objects in terms that students can understand.
- Closed labs, in which students follow instructions to learn something, using the computer, with TAs and consultants available for help.
- Biweekly quizzes that indicate what student should understand are currently important.
- One-on-one sessions, for all students, at least one during the semester.

I will talk a bit about the IDE, but my main emphasis will be on the model of classes and objects.

4. Removing the need for Java applications

The straightjacket of static method main for executing programs makes it difficult to teach Java, especially when trying to teach OO first. What frees us is an IDE that allows any expression or statement to be evaluated or executed immediately, without requiring an application. The use of such an IDE changes drastically what, when, and how one teaches, as I will demonstrate in this lecture, although I cannot do so in a paper such as this.

I use such an IDE. In every lecture, I demo some aspect of Java or of programming methodology. I develop programs and test them in class. My students don't learn about applications and method main until the tenth or eleventh week of the course.

Two IDEs are recommended for this purpose. First, DrJava has had its interactions since its beginning. Second, BlueJ has added a command window that gives the same facility.

I cannot impress upon you enough what a freeing experience it is not to need an application when teaching.

5. A model of classes and objects

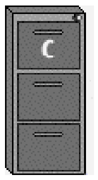
There are two aspects to a programming language: the algorithmic aspect —how one writes sequences of instructions

- The algorithmic aspect, and
- Structural/organizational aspect

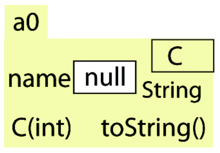
Back in the good old days, the second aspect was essentially ignored, because the main feature for organization was the subroutine, or procedure, or function. It was a rather flat organizational structure. Want more functionality? Just add some more subroutines. OO changes all that, as you know.

When teaching OO first, it is important to explain these two aspects to the students, to make clear to them that the structural aspect will be studied first, and to keep them aware of it from time to time.

In the next section, I will discuss the order of presentation of material, but here, I want to explain in some detail our model for classes and objects.



In our model, a class is a file drawer of a file cabinet, and the manila folders in the cabinet are the objects of the class. The class definition describes the contents of each manila folder (object), as one might expect. When introducing the model to students, I bring actual manila folders, filled with pieces of paper giving the fields and methods of the object.

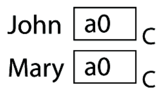


I draw an object to look like a manila folder. The name of the class appears in the upper right part of the folder. This tells us what file drawer it belongs in. The tab of the manila folder contains the name of the object —we'll talk about this in a minute.

In the folder, I place the instance variables (the type of a variable is optional) and the instance methods. I make the point that the whole method appears in the folder, but I draw only the signature of the method. The position of these components in the folder has no meaning at all; I place them wherever we want.

The name on the tab of a folder, in this case a0, is chosen by whoever creates (draws) the folder. The only rule is that different folders have different names. If the computer creates a folder, we have no say in what name is chosen, much like we have no say in the address given to a house we are constructing. If I create-draw a folder, I get to choose its name.

The class name C acts like a type. The values of the type are the names of objects of the class —the names on the manila folders of class C. In this paragraph are two variables, both containing the name of the manila folder drawn above. His makes sense, in the following way. Suppose we are modeling a dentist office and the manila folders are patient's records. Both the secretary, John, and the accountant, Mary, might have access to the folder, and they access it by opening the file drawer and finding it.



The inclusion of fields in a manila folder (or object) is natural.

We explain the inclusion of the methods as follows. In a dentist office, each staff member knows how to carry out some tasks —the secretary can create a folder and insert the patient's name and address, the nurse can make entries on a diagram of the teeth, the accountant knows how to deal with deposits. If we place the instructions in each folder, written as methods, then *anyone* can carry out any of the tasks, providing more flexibility.

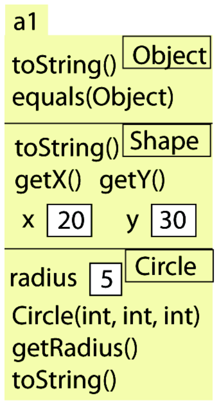
The importance of this model of classes and objects should not be underestimated, for the following reasons:

1. The analogy helps students understand what an object is, right from the beginning. It is *not* in terms of the computer.
2. The pictorial nature of an object makes the concept concrete. Students can *see* what an object is. A huge problem with current texts is that they provide *no* consistent pictorial presentation. On the other hand, we draw objects often, and we force our students to draw them. This makes other points easier to explain later, as we will see.
3. Both the fields and the methods appear in an object. The inclusion of the methods in extremely important for later understanding of how the body of a method can reference a field of the class, as we will see later.
4. The words *pointer* and *reference* are not used. Instead, we have a new type of value, the names on manila folders. Automatically, one can see what an assignment statement like

   John= Mary;

   does: it places in variable John a copy of the name that is in variable Mary.

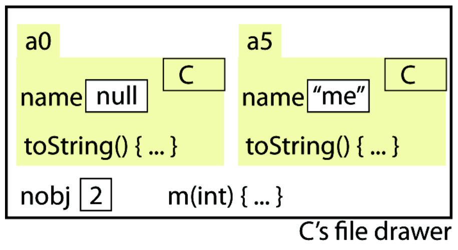
We do not use UML diagrams for objects. UML diagrams have no place for the name of the object —the name on the tab of the manila folder— and this name is a *necessary* and important part of the object. It is this name that leads to easier understanding on the part of the student.



To the right is an example of a drawing an object of subclass Circle. The partition for Circle is at the bottom, and partitions for successive superclasses are drawn above. The subclass *inherits* all the components defined in the superclasses, and you would see them all there if we had drawn them all; we omitted some to save space.

The format is used to explain overriding. Given a method call, look for a matching method starting at the bottom of the object and search upward until it is found. This *bottom-up* rule automatically find the overriding method will be found (if the method is defined in a superclass).

The figure below shows the file drawer for a class C, with two objects, or manila folder. In addition, you can see in the file drawer a static variable and a static method. So, the single copy of each static component goes directly in the file drawer.



I cannot overemphasize the value of these diagrams in helping students learn what an object is. They are not difficult to understand or draw. Once the students grasp the idea, these diagrams can be used to explain various concepts.

For example, every programming language —as well as logic— uses an *inside-out rule* to determine what references mean —I know of only one text that describes this rule. Consider an assignment v= nob;. To find variable nob, search outward through enclosing constructs until the declaration of v is found. Suppose this assignment is in method toString in object a0. Look first in the block in which the assignment appears, then in successive surrounding blocks. If not found, look in the surrounding context, which is object a0. If it is not there, look in the surrounding context, which is the file drawer.

As I said, this general inside-out rule is used in some manner, with perhaps some restrictions and changes, in every programming language. Why not explain it in this fashion to students?

Our model for classes and objects has other advantages and important ramification, for example with an explanation of inner classes, and the flattening of the class structure, which Java does when it compiles. However, we don't have time to go into this here.

Outline of teaching OO first

Now that you know the OO model, you can appreciate the order in which OO can be taught. I teach two 50-minute lectures per week, and in between the students have a 50-minute lab.

1. Expressions, variables, declarations, and assignment, with types **int**, **double**, **char**, and **boolean** and casting.

Lab 1. Practice evaluating expressions using DrJava.

2. Objects. Creation of JFrame objects and calls on methods in the objects, all demoed using DrJava. The students *see* what happens immediately.

3. Definition of a subclass of JFrame. Students see a declaration and use of two methods: a function to compute the area of a JFrame window and a function to make its width equal to its height.

Lab 2. Write similar methods in the lab.

4. Fields, getter/setter methods, JUnit testing.

5. Static variables, the class hierarchy, class Object.

Lab 3. Practice with static variables and JUnit testing.

This ends the pure structure/organization part, and a thorough study of the procedural aspect ensues, including the steps in executing a method call (like drawing a frame or activation record for the call).

Interspersed in these lectures on the procedural aspect will be sublectures on constructors in subclasses, overriding, the inside-out rule, **this** and **super**, method equals, casting, and operator **instanceof**.

By the end of lecture 11, all aspects of OO that we teach have been introduced.

The first programming assignment, due 2.5 weeks into the course, is to right a class that maintains information about rhinos —name, birth date, whether tagged or not, who their mother and father are, if known (as rhino objects). They right getter/setter methods, constructors, and comparison functions, e.g. to see whether one rhino is older than other. Only the assignment and return statements may be used —no conditional expressions.

References

[a] Westin, L.K., and M. Nordstrom. Teaching OO concepts —a new approach. ASEE/IEEE Frontiers in Education Conf., 20-23 Oct., Savannah, GA. F3C-6-11.

[b] Becker, B. Pedagogies for teaching CS1 with Java. http://www.cs.uwaterloo.ca/~bwbecker/papers/javaPedagogies.pdf.

[c] Colburn, T., and Gary Shute. Abstraction in computer science. *Minds and Machines 17*, 2 (July 2007), 169-184.

[d] Burg, J., and S. Thomas. Computer science: from abstraction to invention. www.cs.wfu.edu/~burg/ papers/AbstractionToInvention.pdf

[e] Gunther, Gortz. Abstraction as a fundamental concept in teaching computer science. www8.informatik.uni-erlangen.de/ IMMD8/staff/Goerz/rennesa.ps.gz.

[f] Denning, P., *et al.* Computing as a discipline. *CACM 22*, 2 (Feb 1989), 63-70.

[g] Backus, J.W., *et al.* Revised report on the algorithmic language Algol 60. *CACM 6*, 1 (Jan 1963), 1–17.

[h] Caspersen, M.E. *Educating Novices in the Skills of Programming*. PhD Dissertation, Computer Science, University of Aarhus, Denmark, June 2007.

[i] Astrachan, O., *et. al.* Resolved: objects early has failed. Debate at SIGCSE 2005.

[j] Bruce, K. Controversy on how to teach CS1: A discussion on the SIGCSE-members mailing list. Inroads (Dec 2004).

[k] Gries, D., and P. Gries. *Multimedia Introduction to Programming Using Java.* Springer Verlag, NY. 2005.