Software – Concepts & Tools (1998) 19: 141–145

Why Java is not my favorite first-course language

László Böszörményi

Institut für Informatik, Universität Klagenfurt*, Universitätsstraße 65–67, A-9020 Klagenfurt, Austria;
http://www.ifi.uni-klu.ac.at/cgi-bin/staff_home?laszlo, E-mail: laszlo@ifi.uni-klu.ac.at

Abstract. The choice of the first-course programming language for a university-level computer science curriculum has pedagogical ramifications in terms of comprehensibility and mastery of fundamental concepts. This paper compares the merits of Java and Modula-3 as a first-course language.

Key words: Programming education – Java – Modula-3

1 Introduction (and justification)

The programming language Java [1, 7] is now very fashionable, and to argue against fashion is very similar to fighting against the wind. Nevertheless, I would like to use the ongoing debate about using Java as a first-course programming language as a pretext to discuss once more the purposes of an introductory course in a computer science curriculum at universities. (It is important to stress that I am speaking about universities, because the aims of other educational institutions may be different. Even universities may differ to a large extent, but I think that some common denominators can be found). Thus this paper is not primarily about Java, nor even about computer science, but about teaching. Most computer scientists seldom write (and still more seldom read) papers about teaching. Considering the fact, however, that most computer scientists are also teachers, it seems to need no justification to speak about teaching (actually this statement is a justification, as I realize while writing it). This topic is also my excuse for a more personal style than what is usually tolerated in computer science publications (which is worth another discussion, because the usual impersonal style is often just a way to suggest scientific value that cannot be found otherwise in the given publication).

2 Didactic aims of a first programming course

2.1 Analogy to mother tongue

How we learn something for the first time seems to have particular importance. The best example is surely the learning of the mother tongue. Most people are able to learn foreign languages very well, but almost never perfectly, and the role of the mother tongue always remains a special one. For people having two (or even three) mother tongues, this is true for both (or all three) mother tongues. The analogy between the first programming language and the mother tongue is surely not perfect, because we learn the mother tongue in a very different way and at a very different age than a programming language. In the phase of learning the mother tongue, we have no idea about the grammatical concepts of the mother tongue, not even about the existence of a grammar; we just learn to speak by intuitively grasping the language through unbelievable creative use (Humboldt calls this the *energetic* phase). As Wittgenstein put it, we learn to play a game, without knowing its rules, without even knowing that it has rules. This is definitely not the way that adults can learn a second language or a programming language. Rather, they have to learn the concepts and their use in a long dynamic process. Nobody can learn a programming language properly just by using it, and nobody can learn a programming language without using it. Nevertheless, it is true that the first programming language serves as a reference for learning additional programming languages.

* The paper was written on a sabbatical leave at the State University of New York at Stony Brook.

2.2 No best paradigm – the importance of a historical point of view

Meyer argues for the early introduction of an object-oriented curriculum [4]: "If you think object-oriented development is the right way to go, there is no reason to delay". Unfortunately, I don't think that "object-oriented development is the right way to go". First, for many purposes it is definitely the wrong approach, and, e.g., the concept of *modularization* is much more fundamental than that of object-orientation. I don't know any software of non-trivial size where modularization was not indispensable to handling complexity, whereas I know many cases where object-orientation does not help much. Of course, object-orientation includes modularization, but this is what makes modularization is a simpler and more basic concept. Second, and more importantly, I don't think that there is one single "right way" that universities should try to find and teach. Quite the contrary, university education should show that the approach we think today to be the best (if there is one) was not the first and probably will not be the last. The university should not try to teach ultimate wisdom; it should rather teach students to think about different possibilities. This can be best achieved with a certain historical view. It is surely not possible to always tell the whole story, but it is important to show that – as with so many other things – programming *has* a history and a development, and continues to develop.

The historical point of view reveals another – didactically very important – aspect. The direction of historical development has probably not been an accident. For example, it is surely no accident that programming started with such basic (I deliberately avoid *simple*) concepts as the notions of constant, variable, algorithm and function (procedure) and not with the notion of objects. Much time passed before scientists understood the basic concepts enough to come to the highly sophisticated abstraction of classes and objects. The first object-oriented language, Simula-67, appeared in the mid-sixties, yet almost two decades passed before its significance was generally recognized. Why should we expect that students will be able grasp these notions in a few weeks?

All this does not directly imply that we cannot start the teaching of programming with object-orientation, but it surely follows that we do not have to start with it. Actually, I am rather against starting with object orientation because it puts an unnecessary burden on the students, who might fail to thoroughly understand the basics for a long time. Recall the world-wide pedagogical disaster with using set theory in the elementary school. The idea of deriving almost all of mathematics from set theory is surely a most interesting mathematical effort. However, it is *didactically* a terribly bad idea. To use another example, it is certainly true that every sequential program can be regarded as a special case of a parallel program. Still, this observation does not imply that it is a good idea to start teaching with parallel programming and derive sequential programming as a special case thereof. I do not want to say that it is impossible to start with object-orientation, but my experience suggests that it is better to start with the classical basic concepts and to introduce object-orientation on top of them. A student need not be familiar with all current, fashionable buzzwords already after the first semester. It is much more important that he/she be able to understand and solve a variety of different problems after finishing a computer science major.

In summary, I require the following from the first programming course:

1. It must teach the most basic concepts of programming.
2. It must teach these basic concepts in a way that the student can use them as a reference for advanced concepts and advanced programming notations.
3. It must teach these basic concepts in a way that makes students understand that programming paradigms and notations were not created by God, but were invented by humans who err, and so the concepts are therefore under steady development and (hopefully) improvement.

To put it in another way, an introductory course on programming at a university must do much more than just teaching programming in a given language: it must develop the basic capability in the students to learn any other programming language and any other programming paradigm.

From these requirements it follows that the quality of such an introductory course cannot be evaluated by itself. We have to evaluate it in the context of a whole curriculum. The most important question is: how well does it support later understanding of more advanced concepts?

2.3 Java as a first-course language

My basic criticism of using Java as a first-course language is that a number of basic programming concepts are supported only in an indirect way, and thus the language suggests a limited view of programming. I do not criticize Java as a programming language itself. Although it has its weaknesses, it is certainly a clean and good language and is by far the best among the "popular" programming languages (such as Fortran, Cobol, C and C++). Java might even be cleaner (and is definitely more powerful) than Pascal. In the following discussion I will compare Java to Modula-3 [2, 3] (my favorite first-course language), a high-end member of the Pascal family. The two languages are comparable in their semantic concepts (this is more than an accident; to my knowledge Modula-3 directly influenced the design of Java, although this fact is rather sparsely referenced in Java documents), and they are probably equally well suitable as a programming language for large software projects. However, as a teaching language, just from the beginning, Modula-3 seems to

be superior, as I will try to show. I have, of course, absolutely nothing against teaching Java in later courses. Quite the contrary, I find this important, and my experience shows that students having Modula-3 as their "mother tongue" [5] learn Java very easily and quickly.

In Java, the notion of a module arises only as a special case of a class, the notion of a procedure as a special case of a method, and the notion of a constant as a special case of a variable. This has a certain mathematical beauty, but makes initial understanding difficult. As if my mother, instead of saying: "Please bring a chair to the table", had said "please, send a message to an instance of the class chair, which is a subclass of class furniture, to be brought to the very instance of the class table, being also a subclass of class furniture." I doubt that I could have ever learned to speak in this way.

In the following I will show the lack of a number of notions in Java, from which the reader might conclude that Java is much simpler than Modula-3. This is not the case. The Modula-3 language reference consists of roughly 50 pages, while the Java language reference is a thick book that is still steadily growing. My criticism is not that Java is too simple, but that by centering almost everything around object-orientation, a language was created that is nice to use, but difficult to learn as a first language. In Modula-3, on the other hand, the traditional basic concepts of structured programming reach an unusually mature level, which allows smooth integration of newer concepts such as object-orientation and concurrency.

2.4 The lack of the notion of a module

If you start the programming course in Modula-3, you can tell the students that a program is always constructed as a set of modules, but for a while we will construct simple programs consisting of a single module and employing another module providing basic input/output facilities. If you use Java, you cannot say this so simply, because Java has no explicit notion of a module. You can, of course, say that a Java program consists of a set of classes, and initially we will develop programs consisting of a single class that also must be called static (the explanation of why comes later). This is a minor problem.

The lack of the module concept becomes much worse a few weeks later. Using Modula-3, you can introduce the concept of an *interface* and an *implementation* of a module. The notion of *information hiding* can be easily mapped onto these language concepts – everything that is public is in an interface, everything that is hidden is not. You can introduce (*later*!) more sophisticated concepts, such as public interfaces and special interfaces for friends, which can be mapped onto the same language elements.

In Java you might use abstract classes and class modifiers (there are quite a few of them) for the same purpose. However, Modula-3 *forces* us to define interfaces and corresponding implementations as separate syntactical units. Java does not force such definition, and the temptation not to do so very soon becomes too great for the students. Thus they miss one of the most important software engineering concepts, that of *modularization*. The Modula-3 student will certainly swear in the beginning because she/he is forced to make the many interfaces, but after achieving a certain skill level in this, he/she will understand the importance of the concept and will be ready to construct interfaces in other programming languages (like Java) as well. (This observation is confirmed by a student poll conducted by Prof. Henderson at the State University of New York at Stony Brook, where Modula-3 is used as a first-course language). You can, of course, force the students to always first make an abstract class in Java. In this case, however, they will not swear about the language but about you.

2.5 The lack of the notion of a procedure, modes of parameter passing and of references

In Java, the procedure (or function) is a special case of a method. This is one of the most unhappy didactic facts about Java. In my experience, the concept of a procedure is difficult enough for beginners. It is an absolutely unnecessary burden to be forced to use static and final class methods where you mean just a procedure.

A related problem in Java is the lack of an orthogonal concept of modes of parameter passing. Simple types are passed by value; objects are passed by reference. Parameter passing is defined much cleaner and safer than in C. However, there are programming languages (such as Modula-3) that manage to allow explicit and orthogonal control over the mode of parameter passing in a not less clean and safe way. In many well-known algorithms, the lack of VAR parameters in Java leads to unnecessary complications. Let us take a very trivial example: a procedure that swaps two integer values. In Modula-3 we just write:

```
PROCEDURE Swap(VAR int1, int2: INTEGER) =
    VAR x: INTEGER;
BEGIN x:= int1; int1:= int2; int2:= x
END Swap;
```

In Java this procedure is actually not implementable. We need to wrap an object around the two integers and return this as a function value, which has no justification at all. In order to provide a swap operation on two numbers, we don't need the concept of an object! If this example is not convincing enough, let's take a more real one. Most implementations of an insert (delete) operation of an AVL tree use a Boolean VAR parameter (generally called *h* or *height*) to show whether the tree was grown (reduced) at the last recursion level. To realize this in Java, we need a special object just to store a Boolean value. We may, of course, also change the algorithm, e.g., by turning the insert procedure into a function. In the case of delete, however, which is usually already a function, we need a more

sophisticated change. Anyway, in a number of well-known algorithms the lack of VAR parameters causes certain complications, which has bad consequences not only on their performance but, more important, on their understandability.

Safety arguments are also used to explain why an explicit notion of a pointer missing in Java; this is correct if we take C pointers as a basis. (Implicitly, Java does have pointers, because objects are pointers per definition.) However, there are programming languages (such as Modula-3) that do provide safe pointers (in Modula-3 they are called references) which can be used orthogonally; i.e., we can define a reference to any type. Modula-3 even provides a safe subtyping concept for references, which makes it possible to discuss the idea of a subtype, independently of the class concept, inheritance and dynamic binding. (Actually, Modula-3 provides the subtype concept even for subranges, thus allowing a discussion of the idea with easy examples already in the first weeks.)

2.6 The lack of records and the array flaw

Java does not have an explicit notion of a record. This seems to be not too bad, as we can use classes (maybe declared as final) instead. I am sure that most professional programmers have no difficulty with that. However, didactically this is not a very satisfying situation because records and classes are very different notions. (Moreover, some Java environments require putting classes into separate files, which might be justified for classes, but it is annoying if we mean only a simple record.) In Oberon [6, 8], Wirth and Gutknecht took the record as a basis and made objects to extensible records. This allows a discussion of the record concept first, without the need to introduce objects. Java – and some other orthodox object-oriented languages – do this the other way around. The effect is that many students never gain a clear understanding of a record as a static container for data (and as one of the oldest notions of programming).

Arrays in Java are objects. This has some pleasant consequences for their use, but is definitely a burden for understanding them. Arrays are a very fundamental abstraction (also one of the oldest notions) having nothing to do with object orientation. On the other hand, Java arrays always start with index value 0, which forces the programmer to normalize arrays in a number of well-known algorithms. This is certainly not so difficult, but I personally prefer arrays that can have any index bounds over arrays that also happen to be objects.

Moreover, Java arrays have the well-known covariance problem. If we have an array of type T1 and another array of type T2, where T2 is a subclass of T1, then this second array is regarded as a subclass of the first one (a kind of covariance). This has the pleasant consequence that array values of a subclass can be assigned to array variables or parameters of a superclass. However, this has also the unpleasant consequence that it may lead to unexpected run-time errors, namely, if an array variable has the run-time type of a subclass (via an assignment as described above) then it becomes incompatible with the superclass or with other subclasses of the same superclass. This fact is undetectable for the compiler, however.

If you are now confused after this short explanation – which you likely might be unless you already knew about this phenomenon – how should we explain this to students in the first semester? Now let us consider an example:

```
public class person {                   // a person has
    String name;                        // a name
    void display() { ... } ...          // and a display method
} //person

public class student extends person {
            // student is a subclass of person
    int studentNo;
            // a student additionally has a number
    void display() { ... } ...
            // and an overridden display method
} //student
    . . .
public static void main (String argv[]) {
    person[] persons = new person [3];
            // array of persons
    student[] students = new student [3];
            // array of students
    . . .
    students[1] = new student("Paul", 2);

    persons = students;
            // persons run-time type becomes array of student
    persons[1] = new person("John");
            // leads to ArrayStoreException!
} //main
```

The last statement leads to a run-time error because we cannot assign a person-value to a variable of type student (which person[1] has become). The situation deteriorates if the critical assignment is hidden in a library procedure. We can certainly learn to live with this phenomenon, but it is most inconvenient to explain to beginners why they cannot assign a person to an element of an array of persons.

2.7 Some minor remarks

How the two languages discussed handle constants is characteristic of the languages. If we need a symbolic name for the number of certain data items, then we write in Modula-3:

CONST Number = 100. In Java we write: *static final int Number = 100.* The explanation in the first case is trivial: if you need a constant, write CONST and that's it. How do you explain – in the first or second week – why we have to write all that stuff (static final int) just to get a number?

The switch statement has inherited an ill feature of the corresponding C statement. If you forget to jump out of

a certain selection using a break statement, then control just flows on to the next selection – which is almost surely a major error. The explanation is just wasted time.

The C-like syntax might have played a key role in Java's success. I definitely do not want to trigger any religious war between the followers of Pascal-like and C-like syntax. However, in my experience, beginners have less difficulties with the more English-like syntax of the languages of the Pascal family than with the forest of braces in the languages of the C family. This is especially true if they have to *read* a piece of code, which they definitely should also learn from the beginning.

Java cannot be used as a systems programming language because it does not allow certain low-level operations (such as pointer arithmetic or disabling the garbage collection). There are good safety arguments for this; however, they are not necessarily valid. Modula-3 provides a solution by allowing modules to be declared as *unsafe*. In unsafe modules some safety checks are disabled, but through the explicit notification, such places are easy to localize in a program.

3 Conclusion

My main purpose has been to provoke a discussion about the purpose and method of teaching programming at universities. Teaching is similar to baseball (or to soccer in Europe): everybody knows better how to do it. Among the many people who are loudly proclaiming the introduction of Java as a first-course language, only a small fraction has ever written a program in Java. This is probably not how universities should make such a terribly important decision as the first programming language. Still worse, many responsible people do not even perceive the importance of what we teach in the beginning. This is also why many universities delegate the undergraduate courses to lecturers rather than professors. It is so much easier (although not easy!) to teach things properly at the beginning than to correct misconceptions later.

I have tried to show that Java is not an ideal first-course language and that there are better ones. I did not want to show that Java is a bad language or that it cannot be used as a first-course language. A university might choose the didactically non-optimal solution for several (mostly political) reasons. However, even in such a case, a technical and didactic discussion is an absolute necessity at a university that is responsible for universal education [5].

Acknowledgements. I thank Andras Ercsenyi, Michael Franz, Peter Henderson, Hanspeter Mössenböck, Markus Schordan and Niklaus Wirth for many valuable suggestions, and Jacqueline Cantwell for her careful reading.

References

1. Arnold, K., Gosling, J.: The Java Programming Language. Addison-Wesley, 1996
2. Böszörményi, L., Weich, C.: Programming in Modula-3 – An Introduction in Programming with Style. Berlin, Heidelberg, New York: Springer-Verlag, 1996
3. Nelson, G.: Systems Programming with Modula-3. Prentice Hall, 1991
4. Meyer, B.: Towards an Object-Oriented Curriculum. In TOOLS 11 (Technology of Object-Oriented Languages and Systems). Prentice Hall, 1993, pp. 585–594
5. Mittermeir, R., Böszörményi, L.: Choosing Modula-3 as "Mother Tongue". In Mössenböck H. (ed.): Modular Programming Languages. Proc. JMLC'97. Berlin, Heidelberg, New York: Springer-Verlag, LNCS 1204, 1997, pp. 336–350
6. Mössenböck, H.P.: Object-Oriented Programming in Oberon-2. Berlin, Heidelberg, New York: Springer-Verlag, 1993
7. Niemeyer, P.: Exploring Java. O'Reilly, 1996
8. Reiser, M, Wirth, N.: The Programming Language Oberon. Addison-Wesley, 1992