Which use for Java in introductory courses ?

Jean-Pierre Jacquot

Which use for Java in introductory courses?

J.-P. Jacquot
Université Henri Poincaré – LORIA
B.P. 239 — 54506 Vandœuvre-lès-Nancy — France

Abstract

In the eye of educators, Java looks like a very promising tool. This paper discusses how we can use Java in introductory computer science courses. We first present and analyze how Java fits in our requirements in this context. Then, we present our strategy, based on using functional then object programming.

1 Introduction

In many contexts, the "first time" is very important. The imprint of first experience will last long, for the best or for the worse. During the years, we have made two observations regarding first exposure to computer science. First, it has a strong and lasting impact on the conceptual framework built by students to organize and to understand computational concepts, and on the way students consider programming. Second, the programming language has a strong impact on learning.

Both observations can be easily explained. Learning is about fitting new concepts in a coherent frame. Three basic strategies can be used in case of mismatch: ignoring the new concept, tweaking and reducing it to a variant of a pre-existing one, or rebuilding a new frame. Mental blocks in problem solving [14] or slow acceptance of such useful techniques as, for instance, object oriented programming [12] or UML [6] in industry have in part the same explanation. If the first exposure to computing conveys a feeling of black art, funny tinkering, or rigorous engineering, we can expect most students to adapt their view accordingly. Bachelard [4] observed that by introducing chemistry with flashy experiments, he lead many students to think that chemistry was only about producing funny bangs and dangerous experiments.

Which is the best first programming language is a topic hotly debated. Two extreme views are often confronted. Utilitarians state that the best language is the most used one so that students can learn upfront the real trade and put a known name on their vitæ. The danger of this view is to induce students to focus on the idiosyncratic features of the language rather than on the underlying concepts. Idealists state that programming languages are minor tools, algorithms should be presented in their own abstract language and then translated into programs. While correct for experts, this view overflows many beginners with too many things to learn at once: two languages, translation rules, and underlying concepts.

In our view, introductory computer science courses must focus on basic computational concepts and lay out a sound foundation for subsequent courses like algorithm design, data-structures, data-base, operating system, and so on. The programming language is the

medium through which the concepts are presented and learned. Hence its importance: concepts should have a straight and simple expression in the language, and the computational model should be intuitive and consistent with the concepts. At this stage of learning it is essential that students could reify and play with the concepts. So the programming environment, taken in a broad sense, is important. In a negative way though: its technicalities should not get in the way or burden students.

Beyond the hype, Java [3] is a good candidate as a medium for introductory informatics courses. If the following, we will present a summary of our requirements for an introductory language. Then, we will present where Java falls short and where it leads to improvement in this usage. Last, we will describe our pedagogic strategy, based on OCaml and Java.

2 Language requirements for introductory computer science

In [10], we discussed six requirements for an introductory language.

Conceptual neatness. Beginners need to put concrete forms to new concepts so they can grasp them "physically" before grasping them intellectually. In our discipline, concepts will be syntactic expressions first. So, a one-to-one mapping of concepts to notations is an important feature.

Consistency with common usage. In a scientific university, beginners in computer science are not beginners in computation. Actually, students are fluent in (basic) calculus, algebra, formal and numerical computations. We use this previous experience on two levels: on the surface level, to present students with problems they quickly understand, and on a deeper level, to link computing concepts to previously acquired concepts. However, arbitrary deviances from their usual notations should be avoided to prevent confusion.

Incrementallity. The difficult part in course design is to order the presentation of concepts. An ideal language should grow with the progression for two reasons: students should be able to write and run programs they can fully understand right after the first course, and not-yet-seen constructions should be unavailable. Curiosity and exploration are attitudes to promote, but after ground concepts have been laid out.

Profiling and probing. Programming is about specifying correct and efficient computations. Efficiency is a tricky concept, in which students have a genuine interest. It would be good if the language and its environment allowed for easy observations of this matter.

Observability of execution mechanism. Understanding why and how a machine computes a correct (or incorrect) result is an essential part of learning computer science. Computation mechanisms in "glass boxes" would be a great plus.

Enforceability of good habits. Good writing style (clarity, readability, nice presentation) and systematic documentation are habits we would like to induce from the beginning. Of course, we set up examples and we give encouragements, but a student struggling with a program will soon forget. Ideally, the language or its environment should not allow "dirty" programs.

Today, these requirements are still valid, and we are still looking for the ideal first language. On the one hand, professional languages are designed for experts, who have much different needs than beginners. On the other hand, developing a "beginners' language" requires a huge investment with no economic incentive. ISetl [5] and Scheme [1] meet some of our requirements, but, as we discovered by using them for several years, lack some important concepts like explicit typing for instance.

The question is to find how we can use existing languages for introductory courses. Java is appealing. The following analyses how it fits our requirements.

3 Java as a beginner's language

Although our department was an early adopter of object oriented languages, through Eiffel [13, 9, 11], for teaching programming in computer science curricula, we did not make the move in introductory courses. The major reason is that Eiffel is a rather ascetic language, good for future professional but rebutting for beginners. The advent of Java made us reconsider our position.

3.1 Conceptual neatness

As far as cleanness is concerned, the designers of Java have made an excellent job. An important concern for us is the syntax: major concepts are associated to specific notations onto which students can anchor. It is also possible to use intuition to explain and understand the notation. Objects, inheritance, visibility control, and so on are really upfront.

However, Java has an important shortcoming: genericity. Its simulation with the Object class and casting introduces unwelcome difficulties. The problem comes mostly from the cast, a "dirty" notion with two meanings: type conversion when applied to objects or value conversion when applied to native values. The distinction between native values and objects is generally a bit disappointing for beginners.

3.2 Consistency with common usage

On the language itself, there is not much to say about this form of consistency. The different numerical types (`int`, `float`, `double`, etc.) are still a bit confusing. The `+` operator which converts anything into strings is annoying, particularly when we want to explain the difference between notations (syntax) and values (semantics).

The major quality of Java lies in its library and user interface model. Of course, we don't expect students to program a GUI, but we can easily provide them with a framework into which they just have to plug in a few methods to get an application which mimics the ones they are used to on the computer. Students' motivation is greatly improved when they work on something which looks real in their first laboratory sessions.

3.3 Incrementallity

Because of its pure object orientation, Java raises two difficult issues for beginners. The first is the absence of the "usual" functions such as *sine* for instance. What is the nature of `java.lang.Math`? How to reconcile this strange notion of function with the (same) notion they use in mathematics or physics? Functions are such an important tool in science and such a complex notion that we have a special responsibility for *not* confusing students on this topic.

The second issue is the scaffolding one must build to run a program. Five lines of magic about `static`, `void`, `main`, `String[]`, etc. must be written. Of course, we do provide the lines, but the magic remains, and we think it is a bad thing. Curious students

will want to know, but we cannot explain; for less-motivated students the advertisement "not to be understood now" on the very first line they see is not really motivating; all students will consume a lot a energy trying to learn by rote the magical words. At the very least, students must already have the notion that programming implies respecting some technical constraints from the language and the machine.

3.4 **Profiling and probing** and **Observability of the execution mechanism**

Java is neither worse nor better than other languages. The introspection mechanisms and library classes like Timers are beyond the grasp of beginners.

3.5 Enforceability of good habits

Modeling and structuring into objects is a good habit by itself, the uncompromising approach of Java is a major advantage. Furthermore, Java allows us to promote two fundamental ideas.

The first idea is the notion of reuse, which in turns implies the notion of reading program texts. Effective programming is rarely about building a piece of software from scratch. More often, it is a mix of reuse and intelligent adaptation of code fragments. This notion is not easy to teach formally but it can be approached through practice.

The second idea is the notion of documentation. Here the Javadoc tool is a real plus. Using documentation is part of our trade and this must be learned. Another benefit is the promotion of the idea that the *specification* of a function (or method) is what is really important. As a side effect, we also expect students to become more autonomous.

This analysis led to the strategy exposed hereafter.

4 A pedagogic strategy

In French curricula, students choose their major on the third year, after two years of general science. So, our introductory course actually spans over two years. During the first year, we follow a purely functional approach, using (O)Caml [7, 8]; the duration is 50 hours. The second year, we introduce objects and Java for a 80 hours duration. Courses are evenly divided into 1/3 hall lectures, 1/3 small classes, and 1/3 lab sessions. This overall strategy was inspired by [2].

The major emphasis of the first part is about four concepts (types, functions, recursion, and evaluation as rewriting) and the basic programming constructs (names, conditionals, tuples, lists, and recursive types). Although we don't use formal notation (like, say, inference rules for typing), our course has a strong formal tone. We insist heavily on the fact that we don't introduce new concepts, but rather study new facets and deepen concepts used in other courses. Furthermore, most of the programming exercises are adaptations of problems from mathematics (numerical algorithms) or physics. Every year, we remain surprised by the difficulty students have to understand the difference between defining and using a function. Actually, they struggle with the core concept of function. This comforts us that we must take the time to straighten this foundation concept. On the technical side, we use OCaml in its interpreted version: no delay between typing and getting a result, no "magic" tool like a compiler, and no bothering with input/output.

The second part focuses on the notion of objects as a structuring mechanism and the notion of imperative execution model. The sequence of lectures is the following: (1) Structure of a Java application (classes, objects), (2) Java conditionals and methods, how they relate to the notions in OCaml, (3) Mutability (notion of variable), `for` iterations, (4) Going from recursion to iteration, arrays, (5) Inheritance and inter-class structures, (6) Execution model with inheritance (dynamic binding, cast), (7) Java List, with emphasis on Iterators, and chaining technique, (8) Abstract classes and Interfaces. The last four lectures are an introduction to hardware (binary coding and architecture) during which students complete an ALU Java simulator and code on a machine simulator written in Java.

With such a short and dense course, we don't expect students to be able to write a Java application, or even a complete class. Instead, we expect them to be able to read, to understand and to adapt existing Java classes. For lab-sessions, we ask students to fill in the bodies of a few methods to complete graphical applications.

We emphasize strongly the idea that we are building new concepts on top of those they already know; we are not replacing them. For instance, the very first exercise is to read the complete text (three classes) of a graphical currency converter application, to ask questions about what they don't understand, and to modify it to convert other currencies or Fahrenheit to Celsius temperatures. We avoid stressing syntactic matters, as we don't want students to focus on learning it. A few notions, like the event model or exceptions, are seen by students but presented in an optional follow-on course.

5 First observations

The OCaml part has been established a few years ago and is now working to our satisfaction. The Java part has been run for the first time this year.

Overall, we are reasonably satisfied with the results. More students than what we expected got grades over 18/20 at the final exam. This means that we set an accessible goal for students. However, the average of grades is slightly below our expectations. This means that a few steps in the progression are too steep. Apparently, we went too fast on the notions of mutability and states in objects.

Another satisfactory observation is that most students did not take long to get a correct idea of classes, objects, or inheritance. It confirms our idea that the structuring concepts can be introduced upfront, before the atomic operations. They provide learners with a solid reference frame to which new concepts can be attached. Of course, the difficult trick is to give those abstract concepts a physical representation. On that aspect, OCaml and Java are great tools, thanks for their clean design.

Students moved from the functional world to the object world quite easily. Actually, they transferred what they learned the previous year naturally. For instance, there were no need for extensive presentation of Java types, the idea that classes are types went smoothly, or recursive methods came without a problem.

Our biggest mistake concerns the dot. We followed the usual convention of calling methods in the same class by their simple name. This introduced confusion. Many students struggled a lot to figure out when the dot was needed and when it was not. We suspect that many are still not clear on the idea that methods are always attached to objects. In the future, we intend to use systematically `this.` to refer to local methods. In retrospect, we should have guessed that two different notations for the same concept would lead to confusion...

6 Conclusion

Introducing a new discipline is a major challenge for a teacher. We have to meet three qualities: attractiveness to tickle the students' interest, honesty to allow students to make rational choices for their major, and correctness to give students a useful culture. Furthermore, by its very nature, computer science is less about facts, theorems, or theories, than about reasoning. The heart of our discipline, programming, amounts to model, to structure, and to solve problems. This is the kind of concepts which are difficult both to teach and to learn.

Our experience shows that an introductory course can be built around fundamental concepts, like in any other scientific field. We can do that today because we have tools, namely programming languages, which can provide us with adequate support. Objects are an integral part of these fundamental concepts, and Java is a very good medium to introduce this notion.

Our pedagogic strategy uses two languages. So, one may ask: why don't we stay with only one, since OCaml includes objects? The short, non satisfactory, answer is: fashion and compatibility with other curricula. The better answer is that Java comes with a tremendous environment. We had a lot of fun using it, and we know that students shared some of it!

References

[1] H. Abelson and G. Sussman. *Structure and Interpretation of Computer Programs*. MacGraw Hill, 1985.

[2] Th. Accart-Hardin and V. Donzeau-Gouge Viguié. *Concepts et outils de programmation*. InterÉdition, 1992.

[3] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, 1996.

[4] G. Bachelard. *La formation de l'esprit scientifique*. VRIN, 1938.

[5] N. Baxter, E. Dubinsky, and G. Levin. *Learning Discrete Mathematics with ISETL*. Springer-Verlag, New York, 1989.

[6] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.

[7] J. Chailloux, P. Maunoury, and B. Pagano. *Développement d'applications avec Objective Caml*. O'Reilly, 2000.

[8] G. Cousineau and M. Mauny. *Approche Fonctionnelle de la Programmation*. Ediscience, 1995.

[9] M. Gautier, K. Proch, and G. Masini. *Cours de programmation par Objets*. Masson, 1996.

[10] J.-P. Jacquot and J. Guyard. Requirements for an ideal first language. In P.H. Hartel and R. Plasmeijer, editors, *Proc. First International Symposium on Functional Programming Langugages in Education*, pages 51–63, Nijmegen, 1995. LNCS - Springer Verlag.

[11] G. Masini, A. Napoli, D. Colnet, and K. Tombre. *Object oriented languages*. AcademicPress, 1991. translation of *Les languages à objets*, InterEdition, 1989.

[12] B. Meyer. *Objet-Oriented Software Construction*. Prentice-Hall, 1988.

[13] B. Meyer. *Eiffel, the language*. Prentice-Hall, 1992.

[14] G. Polya. *How to Solve it*. DoubleDay, 1957.