Just Say 'A Class Defines a Data Type'

When teaching computer programming to novices, forget "objects early," "objects later," and "back to basics."

Ask a group of graduating seniors majoring in computer science to define a data type and chances are most of them would be unable to answer even loosely beyond simply providing examples. Then add the concepts of abstract data type and (Java) class, asking about the relationship between them. The same students would also be unlikely to find an answer in a CS1 textbook. Some textbooks might not even contain terms like data type, abstract data type, or type. Yet the study of data types is fundamentally important to learning programming with a typed programming language. Moreover, if novices do not learn data types the right way—consistent across different data types and programming paradigms—some would face insurmountable difficulty learning object-oriented programming, no matter when they learn it. Without a proper characterization of data type, the textbook, as well as the instructor, would be fundamentally challenged to present everything else effectively or even correctly, no matter which approach they took.

When associating the concept of data type and a computing language, a correct definition should be consistent with the following: A data type characterizes how a set of entities is internally represented and algorithmically manipulated. Learning computer programming progresses with the level of maturity commensurate with the learner's sophistication in understanding and using data types.

Novices typically have much more difficulty learning about abstract data types, or ADTs, than about other data types. A reason for this is the lack of adequate coverage early on about data types, their structures, and the roles they play in problem solving, thus making the learning of ADTs a disconnected experience. For example, when learning built-in primitive data types, novices might naively take operations (such as +) for granted. Later, they may encounter the same + applied to strings. However, they normally miss the implication of what they see in both cases—the same apparent operation defined and implemented differently in the respective data types, depending on how the underlying data is represented.

A class (as in an object-oriented programming language like Java) defines a data type. Yet this fact is typically not mentioned in CS1 textbooks. Lacking an accurate description of a data type, the best a textbook (or an instructor) can do is say something like: A class is a template, blueprint, or pattern of an object. These characterizations suggest how a class can be viewed, not what a class is, much less what a class can do, thus making it difficult to learn the true nature of a user-defined data type.

Having learned primitive data types and standalone methods when covering ADTs, novices typically experience a drastic change in the level of abstraction (an ADT is essentially a mathematical algebra) they were unprepared for while learning primitive data types. Unfortunately, this disconnected learning experience is interpreted as "getting a mindset in procedures" by some educators. It was in this context that the notion of "teaching objects-first" was born about a decade ago and is still a reasonably popular teaching strategy for a course for beginning programmers, despite the ongoing con-

LISA HANEY

Why can't educators give their students a consistent view of a data type regardless of whether it is a primitive or user-defined?

troversy. To help with "teaching objects first," educators, notably the creators of the 3D storytelling programming software Alice and integrated programming environment BlueJ, have developed software for visualizing objects and teaching materials using visual objects (such as robots, graphics, and role-playing games). Do they work? The answer depends on who you ask. Educators must still address how novices transform their ability to manipulate visual objects into programming skills. However, the educational shortcomings attributed to poor understanding of ADTs may be rooted more deeply than not understanding what objects are and how they behave.

Textbooks have been written to support the objects-first or objects-early approach, though few of them reflect problem solving at its core. Here, I define some of the more apparent problems I have observed in recent Java texts (none cited in the references here):

Effective teaching methods. Novices learn about object instantiation and behavioral method calls from the start; they may even learn software engineering principles in chapter 1, an approach analogous to teaching babies to speak sentences when they are barely able to speak a word.

Class design. When trying to identify classes, "noun extraction" seems to be a straightforward process, as covered in some textbooks. However, abstract nouns can end up as the basis for defining data types, while a primitive data type may be all that's needed for what is otherwise a user-defined data type. Architectural data types are rarely implied by the nouns of a requirement description, and verbs may wind up defining operations of some unexpected data types. That is why identifying classes generally requires program analysis, a process that's still far from straightforward.

Consistent view. Primitive data types are value types, and user-defined types are reference types. So why can't educators give their students a consistent view of a data type regardless of whether it's primitive or user-defined? A primitive data type encapsulates data in binary format, represents data in an internal format (say, the two's complement, or the way a computer represents integers internally using 1s and 0s), and defines operations we often take for granted. Few programmers would care how + is implemented should integers not be represented by the two's complement. Thus, the level of abstraction, in which + is invoked, separates the relevant detail of "what" from the irrelevant detail of "how"—the essence of ADTs. (OO programming languages also address the "mechanical" differences through autowrapping and unwrapping features.)

Toy objects. Objects with "toy" behavior are often included in textbooks yet solve no practical problems. Assigning appropriate methods to an object is among the most difficult lessons a novice can learn. A program design often dictates whether an object should or should not have certain behaviors due to architectural, maintenance, or efficiency concerns. For example, the statements `student.isScholarshipEligible()` and `financialAidOffice.isScholarshipEligible(student)` may be interchangeable in an application, depending on the design; the question for educators is not whether novices should reach such a level of sophistication, but how they might get there. Toy object after toy object will almost never provide a viable path to this level.

Programming paradigms. "Objects early" means "object-orientation monopoly" in many textbooks. Studies of the practice of software engineering show that object orientation is not a dominant paradigm in real-world software development [1], so why should educators act like it is when teaching novices programming? In fact, viewing software as a service, developers have found a new role for procedural data abstraction in data-intensive and highly distributed business applications. Data handling is mostly proce-

dural in nature; for example, the object-data source control of the ASP.net framework works directly with standalone methods. How can educators introduce object orientation to novices so it is simply a means for problem solving, not a hammer looking for a nail.

A debate "Resolved: Objects Early Has Failed" at the 2005 Symposium of the ACM Special Interest Group on Computer Science Education drew a sizable crowd, and in 2007 a back-to-basics textbook [2] was published. Interestingly, this sequence of "structured programming, teaching objects early, then back to basics" reminds me of the "drills, reformed calculus with technology, back to fundamentals" cycle the mathematics teaching community has experienced since the late 1980s. The critics still believe that the movement to reform the teaching of calculus fails to educate students, considering the courses watered down. Students, they say, come out of introductory calculus courses with no idea how to solve complicated mathematical problems [4]. One can argue (many CS educators indeed do) that teaching "objects first" has compromised teaching algorithm skills. Going "back to basics" in teaching computer programming may avoid some of the problems teaching "objects first" has brought but would still not solve the problem of a difficult transition from procedures to objects that teaching "objects first" sought to address in the first place.

Learning computer programming essentially involves two things: computer algorithms and how data types are defined, designed, and used to solve problems. Meanwhile, novices need a paradigm shift away from solving problems with paper, pencil, and calculator toward solving them algorithmically with conditionals and loops; they also need a gradual introduction of data types with consistency in a problem-solving context. We need a teaching approach that would better capture the essence of learning problem solving through programming with (instead) both procedural and object-oriented problem-solving paradigms. One such approach might include the following content:

- The difference between solving problems with computer programs and with paper, pencil, and calculator;
- The role of data types in problem solving through programming;
- Variables and their roles in assignment statements and elementary programming activities;
- Primitive and library-defined data types, including how they are defined, represented, and used;
- Simple control structures, more roles for variables [3], and a first look at procedures or methods;
- Problem solving with user-defined data types introduced in a context of solving meaningful problems;
- Essential-yet-simple computing algorithms;
- More control structures;
- Array data types and more computing algorithms;
- More problem solving with user-defined types, standalone procedures, or both;
- Type inheritance and its role in problem solving; and
- Problem-solving case studies.

This is neither "objects-first" nor "objects-later" but serves as an invitation to better ideas in teaching novices, not just how to program but how to solve problems through programming. We clearly need another approach to teaching programming to novice students besides "objects first" and "back to basics." C

References

1. Glass, R. One man's quest for the state of software engineering's practice. *Commun. ACM 50,* 5 (May 2007), 21–23.
2. Reges, S. and Stepp, M. *Building Java Programs: A Back to Basics Approach.* Addison-Wesley, Reading, MA, 2007.
3. Sajaniemi, J. and Hu, C. Teaching programming: Going beyond 'objects first.' In *Proceedings of the 18th Workshop of the Psychology of Programming Interest Group* (University of Sussex, Sept. 7–8). University of Sussex, Brighton, U.K., 2006, 255–265.
4. Wilson, R. A decade of teaching 'reform calculus' has been a disaster, critics charge. *The Chronicle of Higher Education 43,* 22 (Feb. 7, 1997), A12–A13.

Chenglie Hu (chu@cc.edu) is a professor of computer science at Carroll College, Waukesha, WI.

DOI: 10.1145/1325555.1325560