The transition from one programming style to another is often a very difficult one. After reading [7], which claims that the proposal in [8] of adapting the original procedural version of the “Hello, world” program into an object-oriented version is harmful, we are inspired to make at least three remarks: that there is a need among educators to address ways to differentiate between OO programming and OO programming languages; that confusion about the fundamentals of OOP is still prevalent among educators, and this confusion is being instituted by current textbooks; and that educators should exert more influence on the quality of textbooks. Here, we address some of the issues raised by [7] and [8]; to preserve the context, we reproduce the sample code used in [8] and considered harmful by [7] in the figure appearing here.

Although there is no widely agreed-upon formal definition for the term OOP, we can safely and reasonably assert, based on current research (for example, [1]), that part1 of the fundamentals of OOP is the following:

- An object is a finite set of components. Each component of an object can be either a piece of data or an operation (over the data components by default). Objects, not classes, should be used as the data-modeling mechanism when computations are considered.
- Computations under OOP are carried out by invoking components of objects (again not classes).

Typically, an object contains both data components (fields) and operation components (methods), and the operation methods are usually defined/performed over the data fields. Nevertheless, an object need not have both operation methods and data fields present to be legitimate. That is, an object may be formed merely of data fields or operation methods. In the former case, the resulting object “degenerates” primarily to a Pascal-style record, where only primitive type or aggregate type data items are allowed to be record members. In the latter case, the resulting object (still) degenerates primarily to a ML-style record, where functions are permitted to be record members.2 Indeed, objects are a general concept that includes records as special instances. Pedagogically, introducing object concepts to first-time students would require us to start from simple objects and gradually move toward more complicated ones. So what is the simplest object? The empty object, of course. Although its usefulness is not evident, the empty object does provide the starting point of an

```
class HelloWorld
{ public static void printHello()
  { System.out.println("Hello, world");}
}

public class UseHello
{ public static void main(String args[])
  { HelloWorld myHello = new HelloWorld();
    myHello.printHello();
  }
}
```

1Other important aspects of OOP, such as inheritance and subtyping, are not addressed here.

2In both cases, to be precise, we need to ensure there are no interdependencies among the components in an object before we can call it a record.
Inductive strategy for constructing objects: an object of one component is extended from the empty object; an object of \(n\) components is extended from a \((n-1)\)-component object, for any \(n > 1\), \(n \in \mathbb{N}\).

Classes are not an essential part of the fundamentals of OOP. Due to the influence of C++ and Java, most students, some programmers, and some instructors think we must have classes in order to program in an OO style; it is not the case. In fact, there are two types of OO languages representing two different approaches to the OOP paradigm: class-based languages (for example, C++ and Java), with which we are familiar, and prototype-based languages (for example, Self [6], Kevo [5], and Omega [2]), with which we may not be familiar.

In the former, classes are used as the manufacturer as well as the type of objects. In the latter, no class notion is needed; objects can be created either from scratch or by inheritance (from another object), and the languages are much cleaner.

Consider the code shown in the figure here. Overall, it does a good job in conveying the basic ideas of OOP. Namely, it shows that an object (\(\text{myHello}\)) must be created first in order to initiate a computation; and the computation is accomplished by invoking a method of this object. Since the code is written in Java, classes are inevitable. Unfortunately, the code itself is not flawless. The loophole of the code in the figure is that the method \(\text{printHello}\) in class \(\text{HelloWorld}\) should be defined as an “object method” rather than a “class method” (that is, remove the keyword \text{static}) because it is supposed to be invoked through the object \(\text{myHello}\). Furthermore, doing so will force method \(\text{printHello}\) to be (only) invoked through an object and eliminate the chance for it to be invoked through a class (as in [7], and described later).

<table>
<thead>
<tr>
<th>Object methods and class methods.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Invoked through objects</strong></td>
</tr>
<tr>
<td>Object methods</td>
</tr>
<tr>
<td>Class methods</td>
</tr>
</tbody>
</table>

We agree with each one of the preceding points as put forth by [7]. For point 1, it contradicts clearly the sample code. As shown, the figure reveals the basic functionality of OOP by creating an object and sending a message to this object to accomplish the computation. On the other hand, the original pure procedural version of “Hello, world” is just that and has nothing to do with object-orientation.

---

1. Using classes as object types actually contributes to the issue of the confusion between subtyping (inheritance) and subtyping [3].
2. Also called instance method (Java jargon)
3. Also called static method (Java jargon)
Due to the influence of C++ and Java, most students, some programmers, and some instructors think we must have classes in order to program in an OO style; it is not the case.

For point 2, we know, by our earlier elaboration on OOP, that an object does not have to have any data members. It is perfectly fine for the class HelloWorld to contain just one method (printHello). Being trivial does not take away the essence of the example. Unfortunately, as we have pointed out, the method printHello should be defined as an object (instance) method, not a class method, to be more appropriate.

For point 3, this would be a shorter working Java code (if and only if we keep the keyword static for method printHello in the figure). However, sending messages to classes is merely a Java peculiarity; it is not an indispensable part of OOP. Even though Java is considered an OOP language, not every Java feature is an OOP construct. This point highlights the recurrent tensions between methodologies and languages.

For point 4, the purpose of the class UseHello is obvious: show how an object of HelloWorld can be created and how a message can be sent to this object to perform a computation. Of course, both HelloWorld and UseHello are toy examples. They are intended to be the very first and the simplest examples for showing OOP essentials to beginners. Nothing more should be expected of them.

We conclude with the following remarks:

• Java ≠ OOP. Java contains the minimum set of requirements of OOP, but also contains constructs that are not only nonessential to OOP, but can confuse and mislead learners of OOP as well.

• We need to separate the essence of OOP from OOP languages to avoid entangling coding and design issues.

• Unfortunately, most available textbooks (generically entitled …Object-Oriented Programming…), stress more language specifics than the fundamental ideas of OOP. They play a major role in blind-folding students in terms of the relationship and distinction between OOP and OOP languages.

• Most importantly, object-oriented design (OOD) should be given precedence over object-oriented programming in teaching OOP. The level of abstraction of OOD will help students acquire a firm basis in the OOP style.

REFERENCES

Cong-cong Xing (cmps-cx@nicholls.edu) is an assistant professor in the Department of Computer Science, at Nicholls State University, LA.

Boumediene Belkhouche (bb@eecs.tulane.edu) is a professor in the Department of Electrical Engineering and Computer Science at Tulane University, New Orleans, LA.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

© 2003 ACM 0002-0782/03/1000 $5.00