# Guidelines for Teaching Object Orientation with Java

Michael Kölling
School of Network Computing
Monash University
mik@monash.edu.au

John Rosenberg
Faculty of Information Technology
Monash University
johnr@infotech.monash.edu.au

## ABSTRACT

How to best teach object orientation to first year students is currently a topic of much debate. One of the tools suggested to aid in this task is BlueJ, an integrated development environment specifically designed for teaching. BlueJ supports a unique style of introduction of OO concepts. In this paper we discuss a set of problems with OO teaching, present some guidelines for better course design and show how BlueJ can be used to make significant improvements to introductory OO courses. We end by presenting a description of a possible project sequence using this teaching approach.

## 1. INTRODUCTION

Teaching object orientation in first year courses, especially in Java, is becoming more and more common. There is, however, still a lack of experience with teaching OO to beginning students. Software tools, teaching support material and teachers' experience all are less mature than the equivalent for structured programming. As a result of this, teachers perceive the teaching of object orientation as being difficult.

One tool to support introductory OO teaching is BlueJ, an integrated software development environment specifically designed for education [2]. BlueJ has some capabilities that are not commonly available in other Java development environments which lead to a different approach to software development. The environment and its functionality have been described in previous papers [3, 4].

Monash University and a few dozen other universities around the world have been using BlueJ in their introductory teaching for a few semesters now. It was always expected, and is now apparent from experience, that teaching with BlueJ fundamentally changes the didactic approach to teaching object orientation to beginning students.

Because the tools available to students and teachers differ from those available in other environments, a different approach to introducing important concepts can be taken. In fact, to make most of the environment, a different approach *should* be taken when using BlueJ.

In this paper we will first summarise the functionality of BlueJ. This section is really only a summary – interested readers are referred to earlier papers. The main part of this paper will then present guidelines for the development of a sequence of programming assignments specifically designed to exploit the possibilities of BlueJ. These guidelines have been used very successfully to develop the assignments used in our own course in the past two years.

Published in *The Proceedings of the 6th conference on Information Technology in Computer Science Education (ITiCSE 2001), Canterbury, 2001.*

## 2. THE BLUEJ ENVIRONMENT

BlueJ is an integrated Java development environment specifically designed for introductory teaching that presents a development front-end which offers a unique interaction style.

The strengths of the BlueJ system are built around three design goals: interactivity, visualisation and simplicity.

BlueJ presents on screen a graphical overview of a project structure in the form of a UML-like class diagram. It then allows the interactive creation of objects from any given class in a software project. Once an object has been created, it becomes visible to the user and any of its public methods can be interactively invoked by selecting it from a pop-up menu. Parameters and method results are entered and presented through dialogue windows.

The environment is carefully designed to be very simple to use. The goal is that students do not need to spend significant time struggling with the environment, but instead concentrate on the programming task.

This was achieved by a conscious trade-off: Much functionality present in other environments, but not needed in first year courses, is not included in BlueJ. This makes BlueJ less suitable for professional development, but represents a great win for introductory teaching. More detailed information about BlueJ is available in [2, 4].

## 3. BLUEJ ASSIGNMENT DEVELOPMENT GUIDELINES

BlueJ, through its unique functionality and interface, allows teachers to teach introductory courses differently than can be done without it. Standard programming examples from existing courses and textbooks can be used, and students will benefit from the greater possibility of interaction and the simplicity of the interface. To exploit the full potential of BlueJ, however, a course should be specifically designed for the functionality of BlueJ.

We will start by discussing a set of guidelines for teaching object-oriented programming. Most of the guidelines themselves are independent of BlueJ, but some are very difficult to follow without it. BlueJ's tools support activities conforming to these ideas.

## Guideline 1: Objects first.

It is now almost consensus among OO teachers that object orientation is best taught by teaching about objects from the start, rather than starting with a small scale, structured programming approach and adding objects later. While there is very little scientific evidence to support this, the anecdotal evidence is so strong that the great majority of teachers and textbooks now follow this approach.

While the idea sounds good, it is not easy with traditional environments to get to objects very quickly. To properly interact with an object, a student typically has to write a main method (code that makes mention of a static method and array parameters besides other concepts!), use a typed variable, create an object and write a method call. These are not trivial concepts and, as a result, it typically takes several hours of instruction (or several chapters in a textbook) to reach the point where object calls can be made.

With BlueJ we can really interact with objects as the very first thing we do. Since objects can be created interactively, the first activity for students should be to open an existing project, create a few objects, make method calls on these objects and inspect the objects' state. Here, we really interact with objects before introducing any other concept. Objects come truly first.

This activity illustrates some very important concepts:

- A Java program has classes representing the program's components.
- Objects are created from classes.
- Many objects can be created from a single class.
- All objects of the same class have the same structure; objects of other classes have different structure.
- The state (variable values) of each object is different.
- Objects have operations (methods) which can be invoked.
- Methods may have parameters and results.

These concepts are illustrated through demonstration or tutorial activity before a student writes any code. Students can experience these concepts through direct interaction.

## Guideline 2: Don't start with a blank screen.

One of the most common mistakes in teaching object-oriented programming is to let students start from scratch.

Starting with a blank screen is a very difficult exercise. Writing a new class involves design. One has to decide what class(es) should exist, and what the methods should be. Even if students are initially told what classes and methods to write, they still do an activity that they at this stage cannot fully comprehend. They cannot recognise *why* the design decisions have been made as they are.

Instead, a student should start by making small changes to existing code. This way, they can go through a sequence of exercises of which they can understand each step.

If they are expected to write some new code, they can do so as part of an existing class. This has been described as an educational pattern under the name "Fill in the blanks" [1].

BlueJ facilitates this by making it easy to provide and execute projects that students can extend. The class diagram provides students with an overview of what is there, and students quickly become accustomed to the idea that some classes are provided and don't need to be changed.

## Guideline 3: Read code.

One of the most puzzling aspects of the way programming is commonly taught is that students start writing code without ever reading any code.

Programming must be one of very few disciplines (the only one?) where this is done. Imagine a course in essay writing. We would be very surprised to see someone expecting students to start writing essays without ever having read one. Without, in fact, knowing what an essay really is.

But often it is even worse than that: Students not only write code before reading, but often don't read code at all. There are many courses where all the programs that students ever see are those that they have written themselves. This really misses a great opportunity. Students can learn a lot from studying well written programs and copying styles and idioms. It is important, though, that all examples students read are well written and are worth being copied.

## Guideline 4: Use "large" projects.

A serious problem with teaching object orientation to beginners is that object-oriented programs have some syntactic overhead. The overhead is roughly proportional to the number of classes (or constant for each class).

This is a problem only because the reason for having this overhead is not immediately apparent to beginners. This problem seems worse the shorter the program is. If the intention is to execute a single line of code, then the syntactical overhead is, relatively speaking, large. It appears more acceptable in longer programs.

Some of the benefits of object orientation only become apparent in larger programs. Object-oriented constructs in a program of one class with one method simply do not make sense. This means that showing students programs that consist of a single method is showing them bad examples (and thus contradicts our guideline 3).

Students should see sensible examples from the start. In particular they should see that a Java application consists of a set of cooperating classes. Thus, we should show them "large" (in student terms) examples, which consist of several classes with a sensible number of methods. Using large projects can have a variety of other related benefits for students, such as practising the reading of code, understanding the need for clarity of code and clear documentation, learning to read class interfaces, possibly team work, and more [6].

## Guideline 5: Don't start with "main".

Java's "main" method has nothing to do with object orientation. Its only reason for being is to connect the application to the operating system. It contains code that does not logically belong to any class or object. It certainly does not implement an operation on an object.

Thus starting to teach about object orientation by showing the "main" method is starting with an exception to the rule, rather than starting with a typical example.

Unfortunately, most Java environments force users to start by studying the main method. This is not true for BlueJ. Since any

method can be invoked, there is no need to use a "main" method at all.

This does not mean that we do not teach about "main" at all. The goal still is that students can, at the end, write programs in any standard environment. It means, however, that a teacher can now choose when to introduce the "main" method instead of being forced to do it as the very first thing. One would not normally choose to introduce concepts such as static methods or arrays in the first lecture.

In our course, we have a week towards the end of the first semester where we cover how to write and execute Java outside of BlueJ. This is done with a lecture and tutorial exercises. At this stage, students know static methods, access modifiers, parameters and arrays, and so they can really understand all necessary details in order to properly understand the "main" method.

### Guideline 6: Don't use "Hello world".

This guideline really follows from the ones above. Using "Hello world" (or an equivalent) as the first example is, however, so common that we have chosen to highlight this point separately.

Since a typical "Hello world" program breaches, simultaneously, *all* of the guidelines above, it is clear that we view it as an extremely bad example.

Lewis argues that "Hello world" is a perfectly good OO example [5], because it makes calls to an object. This, however, does not change the fact that the code that students write is not object-oriented. They write a class of which no object is ever created, and they write a method that does not implement an object operation.

### Guideline 7: Show program structure.

In discussing object-oriented programming, the classes and their relationships are a central issue. The structure of the application is crucial to the quality of a solution. Yet in most development environments, the internal program structure is not visible.

It is very hard to discuss issues in a completely abstract way. Visualising the class structure is crucial for students to develop an understanding of the important concepts.

If programming environments do not provide functionality to display that structure, teachers must take great care to present visual representations by other means as a basis for discussion. BlueJ facilitates this discussion by automatically computing and displaying the class diagram.

### Guideline 8: Be careful with the user interface.

This guideline is concerned with the interface of early student code. How should the results of the student work become visible, and how are parameters entered?

There are three commonly used alternatives: text I/O, graphical user interfaces (GUIs) or applets. All three have their own associated problems.

Text output is unproblematic. It is simple and easy to understand. The problem with text I/O is text input. Reading text from the terminal requires use of non-trivial library classes and methods. It also requires a good understanding of data types and type casting. It is clearly not possible for students to get a good understanding of these issues by the time we want them to produce their first section of code. Expecting beginning students to write text input methods invariably leads to a lot of hand-waving, telling students to just type in a given code pattern without them fully understanding what they are doing.

GUIs provide a serious distraction from the real issues underlying general programming concepts and do not serve well to illustrate general principles. GUI code is a very specific example of an object structure that has very idiosyncratic characteristics that are not common to OO in general.

Building GUIs also distracts from thinking about important issues. Students tend to regard as most important those things they spend most time doing. Building graphical interfaces is very time intensive. Students frequently end up spending a lot of time moving buttons around on the screen instead of working on the functionality or structure of an implementation. They often do not develop an appreciation of program structure and implementation design.

Applets, sometimes used as a quick way to build an interface, also do not help much to simplify things. Applets only remove the top level frame of GUI building. This is provided by the applet viewer or web browser. Other than that, applet construction has the same problems as GUI interfaces (after all, an applet *is* a GUI interface).

In addition to the above, understanding program execution with applets is not trivial. The student only implements a number of seemingly unrelated methods. To understand control flow, a student must develop an understanding of the applet execution framework provided in the viewer or browser. This also is a distraction from fundamental early programming principles.

This, now, leaves the question: What shall we do instead?

## 4. DEALING WITH I/O

To avoid the problems associated with these three kinds of interfacing, we can use one of two possible solutions.

The first is BlueJ specific: since BlueJ allows the execution of arbitrary methods and the interactive passing of parameters, this invocation mechanism can be used to execute student code and provide input. Results are also available via the environment (method results displayed in dialogues). This is the best user interaction style for early examples. All code that students write are standard Java methods – one of the most central concepts of object orientation.

The second solution is related to guideline 4 ("use large projects"). The problem lies not with students (indirectly) using text I/O or GUIs, but only with students being required to write or understand the code doing this. Students can be given partially implemented projects that already provides a user interface, textual or graphical. The student task can then be to work on parts of the project not related to I/O. Thus, the student code again communicates with its environment exclusively through method calls.

This solution is most suitable once students have a good grasp of some of the important concepts.

## 5. A PROJECT SEQUENCE

A sequence of activities or assignments conforming to these guidelines may be structured as follows:

1. Students start by creating objects and executing methods of existing classes. They gain an understanding of classes, objects, methods, parameters and data types. Objects may be composed by passing one object as a parameter to a method of another. No code is written.
2. Students make small modifications to existing code. These modifications can initially be trivial (changes of string literals) and get increasingly more challenging. Ideally, the code they edit is very similar to statements executed interactively in (1). Students experience the edit/ compile/execute cycle, compiler errors and gain a first insight into Java code. In our course, we use an example where the code being modified corresponds very closely to statements which have been invoked interactively before. Students manage to modify and extend the given code without any prior instruction about Java syntax.
3. The next project requires students to implement or change method bodies. All required methods exist as methods skeletons with empty bodies. Students concentrate on how to implement methods, after having become familiar with using them..
4. Next, students are required to add methods to existing classes. Students must decide what methods are needed and what the signature should be.
5. After this, a project may require students to add not only statements and complete methods, but also new classes. The classes to be added to the project are fairly obvious and easy to understand.
6. As a last, advanced exercise, students develop a project from scratch. They start with an empty screen and a vague problem description. They go through the whole design process of deciding what the classes and their interfaces should be, as well as the implementation.

For our course, we have developed a sequence of assignments that spans two full semesters, following this outline.

Step number 1 (interactive execution) is done with a project called "shapes" – an application that allows the interactive creation of squares, circles and triangles. These shapes can then be moved around on the screen via interactive method calls, their size and colour may be modified. Students are encouraged to create a simple picture.

The second project is named "picture". Here, a class contains code to created, modify and display shapes to present a simple picture. Students modify this code.

For step number 3, a calculator project is used. Students are given a calculator implementation where the implementation of several methods in a class called "CalcEngine" is missing. They need to add method bodies and instance variables.

Step 4, adding methods to existing classes, is done via two projects, a game called "blocks" and an image viewer project. In the "blocks" game, students implement the game control loop; in the image viewer, students write several image modification operations, such as smoothing, thresholding and image overlay functions.

Following this, a text based adventure game is used to let students add classes. Again, a framework is given to students with the expectation that they extend the given code.

Lastly, a large discrete event simulation is designed and implemented by student groups. They are not given any code for this last assignment.

Space does not allow us to discuss these projects here in more detail. However, and extended version of this paper describing these projects, as well as the code for the projects themselves, can be downloaded from the author's web site (see below).

## 6. CONCLUSION

We have argued that BlueJ allows and benefits from the use of a different approach to introducing object-oriented programming concepts. We have discussed a list of eight guidelines for developing an introductory course and given a description of a sequence of programming tasks following the ideas described in the guidelines.

The guidelines and the assignment sequence are general and may be put to beneficial use in any course, using BlueJ or not. The use of BlueJ, however, supports this line of introduction well and makes the implementation of some of the suggested guidelines much easier.

## 7. DOWNLOADS

The BlueJ system and its documentation is freely available at http://bluej.monash.edu.

The code for the projects described here as well as an extended version of this paper with a more detailed description of the projects is available from the author's web site at http://www.netcomp.monash.edu.au/~mik.

## 8. REFERENCES


[1] J. Bergin, *Fourteen Pedagogical Patterns for Teaching Computer Science*, in Proceedings of the Fifth European Conference on Pattern Languages of Programs (EuroPLop 2000), Irsee, Germany, July 2000.

[2] M. Kölling, *BlueJ - Teaching Java,* web site at http://bluej.monash.edu, Monash University.

[3] M. Kölling, *Teaching Object Orientation with the Blue Environment,* Journal of Object-Oriented Programming, Vol. 12 No. 2, 14-23, May 1999.

[4] M. Kölling and J. Rosenberg, *BlueJ - The Hitch-Hikers Guide to Object Orientation,* to appear in Journal of Object-Oriented Programming, June 2001.

[5] J. Lewis, *Myths about Object-Orientation and Its Pedagogy*, in SIGCSE 2000 Proceedings, ACM, Austin, Texas, 245-249, March 2000.

[6] K. T. Stevens, *et al.*, *Using Large Projects in a Computer Science Curriculum (Panel)*, in SIGCSE 2000 Proceedings, ACM, Austin, Texas, 399-400, March 2000.