International Journal of People-Oriented Programming
Volume 6 • Issue 2 • July-December 2017

Book Review

Objects First With Java:
A Practical Introduction Using BlueJ (5th Edition)

Reviewed by Steve Goschnick, School of Design, Swinburne University of Technology, Melbourne, Australia

Objects First with Java: A Practical Introduction Using BlueJ (5th Edition)
David Barnes & Michael Kölling
(c) 2012 by Pearson Education, Inc.
546 pp.
$17.99
ISBN 978-0-13-283554-1

This is a book specifically for teaching Java that would be suitable to students in either: the middle to upper end of high school, or even a first-year university or college course. It is also about teaching object-oriented programming (OOP), with a leaning towards software engineering by way of introducing several analysis and design methods, and the unit testing of code. It takes a considerably different approach to teaching it than most textbooks on Java that I've encountered, of which I've probably evaluated at least a dozen of the main players in the past for various teaching assignments. (n.b. I've built two subjects from zero on Java and OOP: one for 2nd year university students of Information Systems, and a much earlier course in 1998 for 3rd year Computer Science & Software Engineering students. I've also developed and taught two days short courses on the subject, both to Apple University Consortium conference goers and to all-comers).

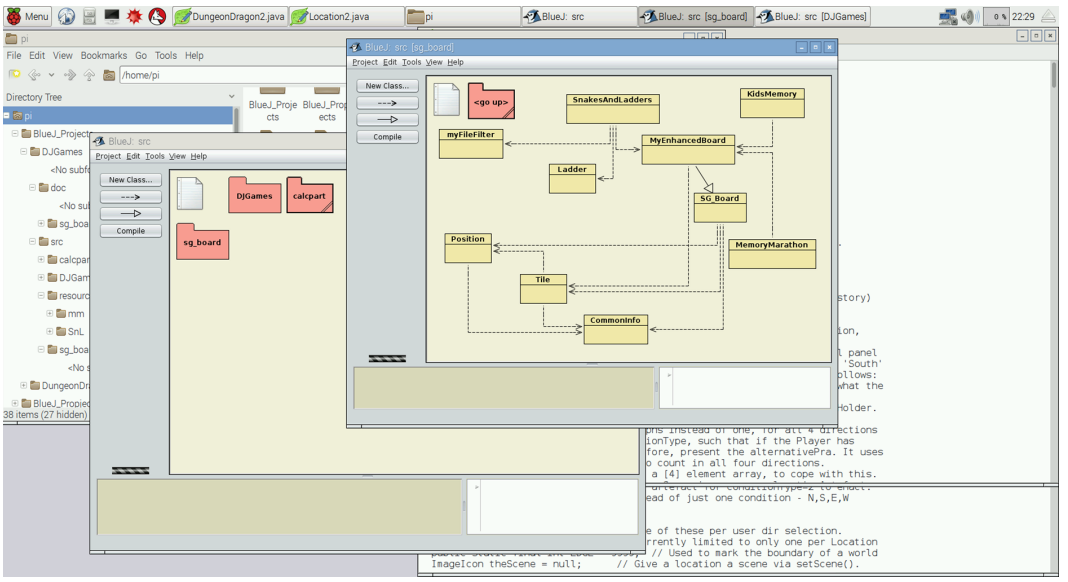
Apart from the different approach to teaching Java and OOP the book is also unique in being intertwined with a Java IDE (Integrated Development Environment) called BlueJ, designed for teaching OOP - so this review must also include something of BlueJ too. Indeed, BlueJ precedes the book - which has now been used for over 15 years across Editions to teach students around the planet. My previous encounter with BlueJ was way back in 1998/9, when I used it to try and teach Java to my own

two eldest children (then both pre-teen). More interesting and substantial, is advice in the Foreword to the book, by none other than James Gosling the creator of Java. He laments that his own daughter and her middle school classmates had struggled through a Java course using a commercial IDE. He finishes his Foreword with "I wish it (BlueJ + book) had been around for my daughter last year (at her school). Maybe next year …"

Imagine my delight when I first fired up my Raspberry Pi and discovered BlueJ as one of the pre-installed IDEs along with the full version of Java 8 onboard. BlueJ has come a long way since my first encounter with it. It is now a full-blown IDE - these sorts of resources on the Raspberry Pi underline just what extraordinary value the Pi is as a computer for teaching. (See: Figure 1 - a medium-sized Java project I imported and continue to develop, without any hit-a-brick-wall incidents). Furthermore, the book has an active community forum for teachers at: http://blueroom.buej.org

Unlike many textbooks on the Java language, the chapters in this book are sequenced on software concepts, rather than by Java language concepts. It is peppered with many projects - 24 of them - which have been worked into a project-based approach to the subject. The book design is based on what the authors call a project-driven approach to learning the Java language and OOP. The projects each start out relatively simple, but always arrive at a practical solution to the problems posed earlier on. Furthermore, most of the projects can be extended in a number of directions, should the student wish to go well beyond the structure that the pre-existing code provides. Some of the projects could be quite useful to a student of coding beyond the class room, such as an .mp3 music playing app. Many of the projects parallel topics I've set my own students upon in the past, not just in Java subjects, but in information modelling subjects too,

Figure 1. Visual diagram of classes in the associated BlueJ IDE on a Raspberry Pi



including: ticket-machines, online auctions, tech-support systems, a drawing program, cinema/movie booking system, and so on.

As the student (or teams of students - as there is support for teams via BlueJ) advances through the chapters, the projects gain in complexity, facilitating the learning of new language concepts, but also revisiting language constructs and their earlier usage. This approach further familiarises the student with each concept and reinforces the learning. This is a contextual approach with respect to the projects, unpacking the various language constructs as needed to solve a newly encountered logical situation or cul-de-sac.

Let me hark back to the 'object-first' approach in this book. Most authors of Java language focused textbooks do either: objects-first (or early in the proceedings), or objects-last (or much later in the proceedings). There is even one such book author (Cay Horstmann) who refuses to bet either way, and has two books selling in parallel: one with objects-first, the other with objects-last! The traditional objects-last approach typically concentrates on the language constructs (the so-called imperative language features of Java), before getting to the OOP part proper. The OOP paradigm effectively rescued the imperative languages from their earlier ineffectiveness in dealing with the complexities of modern software projects. The OOP paradigm first came to the mainstream, when most programmers were faced with GUIs (Graphical User Interfaces), around the mid-1990s. Hence, these objects-last books are typically fairly dull for the student, as they deal with character-based interfaces from the pioneering days of computing, sometimes for hundreds of pages before getting to a GUI interface and or to substantial coverage of programmed objects.

Alternatively, the objects-first books often tend to introduce the main GUI classes within Java very early on, as the preeminent objects within the code libraries that are included with the language compiler. That approach unfortunately involves complex languages features such as: abstract classes, class interfaces, event adapters and so on, quite early on in the book. This tends to loose a large number of students, before they have had a chance to gauge whether or not, they have an aptitude for coding.

This book does things much differently. It starts with Objects and Classes from the very first page, and it does so without the need to go anywhere near the GUI classes in Java - although it does cover those later in the book. It can do this because of BlueJ. In BlueJ the student defines a class and creates/instantiates an object of that class, using visual diagrams (simplified UML class diagrams), without the need to write any code first up. The book begins with some existing example code. The student is guided through a few mouse clicks to create an instance of a class and then interacts with the methods and data of that object. The student sees what a programmed object is, how its methods work, and then looks at the code that did it. Every single program created or edited in BlueJ is object-oriented with a graphical representation of the classes involved, GUI or no GUI. For example, when I imported my code in Figure 1 into BlueJ, (which was created in a different IDE on a desktop computer), the BlueJ IDE created the necessary diagrams for all the classes and packages of classes involved - of course I dragged the boxes around a bit to increase the aesthetics. Although BlueJ is

an IDE designed and has evolved as a teaching tool (See the history of it in an earlier IJPOP paper by Michael Kölling (2015), it is not a toy IDE, but is a fully-functioning tool that can be used for developing significantly sized projects. E.g. some of my projects now in BlueJ have well over 50,000 lines of Java. Beyond this book review, I've encountered no significant issue while importing these medium-sized projects into BlueJ, nor while enhancing the code within it. (One minor issue: the code line numbers down the lefthand side, only allow room for 3 digits, so for single files with more than 999 lines of code, I can't see the fourth digit.)

Back to it as a teaching book: the exercises are interspersed across a given chapter, rather than just those at the end. This places a given exercise in the best possible context of the material covered to date. By the time the student has advanced to the end of Chapter 5, most of the Java language features have been covered more than once.

As mentioned earlier, a software engineering theme also runs through the book, but particularly from Chapter 6 onward. In fact, as the authors suggest, the second half of the book can be used for a second, more advanced Java-oriented course. Chapter 6 is about designing classes, which includes concepts of coupling and cohesion - two metrics used to measure the quality of OOP code. It also covers the refactoring of code, otherwise known as repurposing or improving existing code.

Chapter 7 is about testing and debugging your programs, something that grows in importance as the size of a code base grows. Chapters 8 and 9 are about improving the design of programs through the OOP paradigms other two key concepts of inheritance and polymorphism, and Java-specific related issues. Chapter 10 is about abstract classes and interfaces (the class kind, not the UI kind). Chapter 11 is about graphical user interfaces (the UI kind, not the class kind) - using the Swing class library - including Layout Managers for dealing with the widely variant screen sizes on modern devices, from phone and tablet sized screens to those large wall-mounted flat ones. Chapter 12 is about error handling, including defensive programming techniques and exception handling, and some related topics.

Chapters 13 is an introduction to Analysis and Design of a software system, drawing upon an example of what one might do, if you start with just natural language descriptions of some future desired system. Rather than starting with an existing set of classes to be examined, understood and enhanced, this chapter is about starting with a significant problem and no code whatsoever, to a non-trivial problem that is not yet fully described. It introduces system analysis using the Noun-Verb Method to identify candidate class names, and some of their method calls needed too. It then calls upon the CRC Cards Method to identify the functionality needed by such classes and the relationships between those classes, and it uses Scenarios to further extract the needed functionality of the overall software system being envisaged, or the iterated vision (surely thats an oxymoron?).

This chapter is useful to students in a number of ways:

- It introduces some of the methods used in software engineering to nut-out a starting set of classes and then how to improve upon their design and appropriateness regarding the functionality needed;

- It gives them a glimpse to other aspects of *information systems* beyond just coding (e.g. some of my best post-graduated *Application Analysts/Architects*, came my way as students holding Arts degrees - it would be nice for their current-day high school student equivalents to be aware of the breadth of careers in ICT, and chapters like this one help create that awareness);
- It would also help them if they have a need to develop a so-called greenfields application (think novel 'Startup' product).

What it doesn't do is substitute for a real client (beyond oneself/ or the inclusive team) with existing under-serviced needs, who is most often the source of the requirements, protocols and procedures that a software system usually addresses. One can use these (what are often called) bottom-up design methods such as Noun-Verb analysis, and CRC Cards for greenfield or self-defined projects, but further techniques, including top-down methods, are also important complementary tools to have, when dealing with third-party clients. Nonetheless the introduction of other software engineering topics, such as iterative prototyping (which can lead into a discussion of Agile development), the inadequacy of the much older waterfall method, and design patterns - makes this chapter a useful launch into another subject or book, beyond the topic of Java and OOP.

Chapter 14 is the last. It presents a Case Study which is larger than the other projects in this book. It doesn't introduce any new concepts, beyond just being on a larger scale - like one usually encountered in most commercial or organisational systems. When I've taught such a subject at university level, to either Information System students, or Software Engineering and Computing students, at least 25 percent of the assessment is drawn from a substantive project building a real-world application, via either servicing a real client and their actual needs; or else, me acting like a pseudo-client online in a moot project manner. This chapter in some small way, substitutes for such a client. Of course, this book can cater for subjects that are taught with or without a large project assignment task.

The approach in this book is not how I've taught Java in the past, but certainly a way I'd like to try in the future. With so many users of the book and the companion web site and the community of users around it, the approach clearly has a lot going for it, to both the student and the teacher of Java and Object-Oriented Programming. With respect to the Raspberry Pi, Scratch is currently a popular language choice for first comers to coding, particularly in the primary school system and lower high school. However, once you have a cohort of students in the high school system who definitely have a deeper interest in coding, perhaps even already intent on pursuing a career (or making a startup product!) in programming, or software engineering, or just in the Sciences - then Java and OOP represent an excellent language choice. The two big OOP languages in the commercial world at the moment are Java and C# - both strongly-typed languages. Having strongly-typed variables in a language, is a feature favoured by professional software developers, and even more so by their managers and those people who have to account for the time and money spent on a project (note: Python,

popular as it is, is not a strongly-typed language). C# is a Microsoft creation that was inspired by Java, and so it is relatively easy for a programmer to move between the two languages. The BlueJ IDE is an excellent teaching vehicle for the Java language and OOP. Teaching it on the Raspberry Pi, using BlueJ and this book, is probably the current best path to teaching middle to upper high school students who are destined for a career somewhere in the software world.

Postscript: There is a more recent 6th Edition of this book. It covers a few extra features of the Java language. Specifically, some coverage of the Functional Programming features added to Java in version 8. This edition of the book includes another Chapter on collections where functional programming constructs can gain advantage with multi-core processors. The content in a few other chapters is shuffled about a bit, and some other code is modified with functional programming constructs. My view on teaching Functional Programming: if you want to teach students new to programming, functional programming, use a Functional Programming language.

REFERENCES

Kölling, M. (2015). Lessons from the Design of Three Educational Programming Environments *International Journal of People-Oriented Programming*, *4*(1), 5–32. doi:10.4018/IJPOP.2015010102