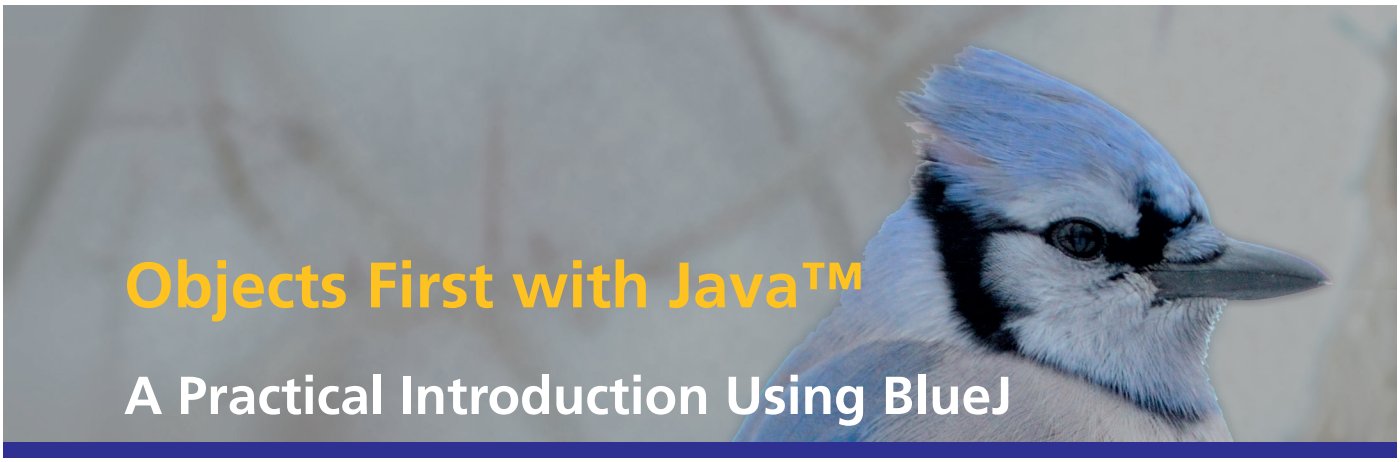


Objects First with Java™

A Practical Introduction Using BlueJ

David J. Barnes and Michael Kölling

University of Kent

Sixth Edition

PEARSON

Boston Columbus Indianapolis New York San Francisco Hoboken
Amsterdam Cape Town Dubai London Madrid Milan Munich Paris Montreal Toronto
Delhi Mexico City Sao Paulo Sydney Hong Kong Seoul Singapore Taipei Tokyo

Contents

	Foreword	xiv
	Preface	xv
	List of Projects Discussed in Detail in This Book	xxv
	Acknowledgments	xxviii
Part 1	Foundations of Object Orientation	1
Chapter 1	Objects and Classes	3
1.1	Objects and classes	3
1.2	Creating objects	4
1.3	Calling methods	5
1.4	Parameters	6
1.5	Data types	7
1.6	Multiple instances	8
1.7	State	9
1.8	What is in an object?	10
1.9	Java code	11
1.10	Object interaction	12
1.11	Source code	13
1.12	Another example	15
1.13	Return values	15
1.14	Objects as parameters	16
1.15	Summary	17
Chapter 2	Understanding Class Definitions	21
2.1	Ticket machines	21
2.2	Examining a class definition	23
2.3	The class header	25
2.4	Fields, constructors, and methods	26
2.5	Parameters: receiving data	32
2.6	Assignment	34

2.7 Methods 35
2.8 Accessor and mutator methods 36
2.9 Printing from methods 39
2.10 Method summary 42
2.11 Summary of the naïve ticket machine 42
2.12 Reflecting on the design of the ticket machine 43
2.13 Making choices: the conditional statement 45
2.14 A further conditional-statement example 47
2.15 Scope highlighting 48
2.16 Local variables 50
2.17 Fields, parameters, and local variables 51
2.18 Summary of the better ticket machine 53
2.19 Self-review exercises 53
2.20 Reviewing a familiar example 55
2.21 Calling methods 57
2.22 Experimenting with expressions: the Code Pad 59
2.23 Summary 61

Chapter 3 Object Interaction 67
3.1 The clock example 67
3.2 Abstraction and modularization 68
3.3 Abstraction in software 69
3.4 Modularization in the clock example 69
3.5 Implementing the clock display 70
3.6 Class diagrams versus object diagrams 71
3.7 Primitive types and object types 72
3.8 The `NumberDisplay` class 72
3.9 The `ClockDisplay` class 80
3.10 Objects creating objects 83
3.11 Multiple constructors 84
3.12 Method calls 84
3.13 Another example of object interaction 88
3.14 Using a debugger 92
3.15 Method calling revisited 96
3.16 Summary 97

Chapter 4 Grouping Objects 101
4.1 Building on themes from Chapter 3 101
4.2 The collection abstraction 102

4.3 An organizer for music files 103
4.4 Using a library class 104
4.5 Object structures with collections 107
4.6 Generic classes 109
4.7 Numbering within collections 110
4.8 Playing the music files 113
4.9 Processing a whole collection 115
4.10 Indefinite iteration 120
4.11 Improving structure—the `Track` class 128
4.12 The Iterator type 131
4.13 Summary of the music-organizer project 135
4.14 Another example: an auction system 137
4.15 Summary 147

Chapter 5 Functional Processing of Collections (Advanced) 149
5.1 An alternative look at themes from Chapter 4 149
5.2 Monitoring animal populations 150
5.3 A first look at lambdas 154
5.4 The `forEach` method of collections 156
5.5 Streams 158
5.6 Summary 168

Chapter 6 More-Sophisticated Behavior 171
6.1 Documentation for library classes 172
6.2 The *TechSupport* system 173
6.3 Reading class documentation 178
6.4 Adding random behavior 183
6.5 Packages and import 189
6.6 Using maps for associations 190
6.7 Using sets 195
6.8 Dividing strings 195
6.9 Finishing the *TechSupport* system 197
6.10 Autoboxing and wrapper classes 199
6.11 Writing class documentation 201
6.12 Public versus private 204
6.13 Learning about classes from their interfaces 206
6.14 Class variables and constants 211
6.15 Class methods 214
6.16 Executing without BlueJ 216

6.17 Further advanced material 216
6.18 Summary 220

Chapter 7 Fixed-Size Collections—Arrays 223
7.1 Fixed-size collections 223
7.2 Arrays 224
7.3 A log-file analyzer 224
7.4 The for loop 230
7.5 The *automaton* project 236
7.6 Arrays of more than one dimension (advanced) 244
7.7 Arrays and streams (advanced) 251
7.8 Summary 252

Chapter 8 Designing Classes 255
8.1 Introduction 256
8.2 The *world-of-zuul* game example 257
8.3 Introduction to coupling and cohesion 259
8.4 Code duplication 260
8.5 Making extensions 263
8.6 Coupling 266
8.7 Responsibility-driven design 270
8.8 Localizing change 273
8.9 Implicit coupling 274
8.10 Thinking ahead 277
8.11 Cohesion 278
8.12 Refactoring 282
8.13 Refactoring for language independence 286
8.14 Design guidelines 291
8.15 Summary 292

Chapter 9 Well-Behaved Objects 295
9.1 Introduction 295
9.2 Testing and debugging 296
9.3 Unit testing within BlueJ 297
9.4 Test automation 304
9.5 Refactoring to use with streams (advanced) 311
9.6 Debugging 312
9.7 Commenting and style 314
9.8 Manual walkthroughs 315

9.9 Print statements 320
9.10 Debuggers 324
9.11 Debugging streams (advanced) 325
9.12 Choosing a debugging strategy 326
9.13 Putting the techniques into practice 327
9.14 Summary 327

Part 2 Application Structures 329

Chapter 10 Improving Structure with Inheritance 331
10.1 The *network* example 331
10.2 Using inheritance 343
10.3 Inheritance hierarchies 345
10.4 Inheritance in Java 346
10.5 *Network*: adding other post types 349
10.6 Advantages of inheritance (so far) 351
10.7 Subtyping 352
10.8 The `Object` class 358
10.9 The collection hierarchy 359
10.10 Summary 359

Chapter 11 More about Inheritance 363
11.1 The problem: *network's* display method 363
11.2 Static type and dynamic type 365
11.3 Overriding 368
11.4 Dynamic method lookup 370
11.5 `super` call in methods 373
11.6 Method polymorphism 374
11.7 Object methods: toString 374
11.8 Object equality: `equals` and `hashCode` 377
11.9 Protected access 379
11.10 The `instanceof` operator 381
11.11 Another example of inheritance with overriding 382
11.12 Summary 385

Chapter 12 Further Abstraction Techniques 389
12.1 Simulations 389
12.2 The foxes-and-rabbits simulation 390
12.3 Abstract classes 405

12.4 More abstract methods 412
12.5 Multiple inheritance 414
12.6 Interfaces 417
12.7 A further example of interfaces 425
12.8 The `Class` class 427
12.9 Abstract class or interface? 427
12.10 Event-driven simulations 428
12.11 Summary of inheritance 429
12.12 Summary 430

Chapter 13 Building Graphical User Interfaces 433

13.1 Introduction 433
13.2 Components, layout, and event handling 434
13.3 AWT and Swing 435
13.4 The ImageViewer example 435
13.5 ImageViewer 1.0: the first complete version 447
13.6 ImageViewer 2.0: improving program structure 461
13.7 ImageViewer 3.0: more interface components 467
13.8 Inner classes 471
13.9 Further extensions 476
13.10 Another example: MusicPlayer 478
13.11 Summary 481

Chapter 14 Handling Errors 483

14.1 The *address-book* project 484
14.2 Defensive programming 488
14.3 Server error reporting 491
14.4 Exception-throwing principles 495
14.5 Exception handling 501
14.6 Defining new exception classes 508
14.7 Using assertions 510
14.8 Error recovery and avoidance 513
14.9 File-based input/output 516
14.10 Summary 527

Chapter 15 Designing Applications 529

15.1 Analysis and design 529
15.2 Class design 536
15.3 Documentation 538

15.4 Cooperation 539
15.5 Prototyping 539
15.6 Software growth 540
15.7 Using design patterns 542
15.8 Summary 548

Chapter 16 A Case Study 551

16.1 The case study 551
16.2 Analysis and design 552
16.3 Class design 556
16.4 Iterative development 561
16.5 Another example 570
16.6 Taking things further 570

Appendix A: Working with a BlueJ Project 571

A.1 Installing BlueJ 571
A.2 Opening a project 571
A.3 The BlueJ debugger 571
A.4 Configuring BlueJ 571
A.5 Changing the interface language 572
A.6 Using local API documentation 572
A.7 Changing the new class templates 572

Appendix B: Java Data Types 573

B.1 Primitive types 573
B.2 Casting of primitive types 574
B.3 Object types 574
B.4 Wrapper classes 575
B.5 Casting of object types 575

Appendix C: Operators 577

C.1 Arithmetic expressions 577
C.2 Boolean expressions 578
C.3 Short-circuit operators 579

Appendix D: Java Control Structures 581

D.1 Control structures 581
D.2 Selection statements 581

D.3 Loops 583
D.4 Exceptions 585
D.5 Assertions 587

Appendix E: Running Java without BlueJ 589
E.1 Executing without BlueJ 589
E.2 Creating executable `.jar` files 591
E.3 Developing without BlueJ 591

Appendix F: Using the Debugger 593
F.1 Breakpoints 594
F.2 The control buttons 594
F.3 The variable displays 595
F.4 The Call Sequence display 596
F.5 The Threads display 596

Appendix G: JUnit Unit-Testing Tools 597
G.1 Enabling unit-testing functionality 597
G.2 Creating a test class 597
G.3 Creating a test method 597
G.4 Test assertions 598
G.5 Running tests 598
G.6 Fixtures 598

Appendix H: Teamwork Tools 599
H.1 Server setup 599
H.2 Enabling teamwork functionality 599
H.3 Sharing a project 599
H.4 Using a shared project 599
H.5 Update and commit 600
H.6 More information 600

Appendix I: Javadoc 601
I.1 Documentation comments 601
I.2 BlueJ support for javadoc 603

Appendix J: Program Style Guide 605
J.1 Naming 605
J.2 Layout 605

J.3 Documentation 606
J.4 Language-use restrictions 607
J.5 Code idioms 608

Appendix K: Important Library Classes 609

K.1 The `java.lang` package 609
K.2 The `java.util` package 610
K.3 The `java.io` and `java.nio.file` packages 611
K.4 The `java.util.function` package 612
K.5 The `java.net` package 612
K.6 Other important packages 613

Index 615

Foreword

by James Gosling, creator of Java

Watching my daughter Kate and her middle-school classmates struggle through a Java course using a commercial IDE was a painful experience. The sophistication of the tool added significant complexity to the task of learning. I wish that I had understood earlier what was happening. As it was, I wasn't able to talk to the instructor about the problem until it was too late. This is exactly the sort of situation for which BlueJ is a perfect fit.

BlueJ is an interactive development environment with a mission: it is designed to be used by students who are learning how to program. It was designed by instructors who have been in the classroom facing this problem every day. It's been refreshing to talk to the folks who developed BlueJ: they have a very clear idea of what their target is. Discussions tended to focus more on what to leave out, than what to throw in. BlueJ is very clean and very targeting.

Nonetheless, this book isn't about BlueJ. It is about programming.

In Java.

Over the past several years Java has become widely used in the teaching of programming. This is for a number of reasons. One is that Java has many characteristics that make it easy to teach: it has a relatively clean definition; extensive static analysis by the compiler informs students of problems early on; and it has a very robust memory model that eliminates most "mysterious" errors that arise when object boundaries or the type system are compromised. Another is that Java has become commercially very important.

This book confronts head-on the hardest concept to teach: objects. It takes students from their very first steps all the way through to some very sophisticated concepts.

It manages to solve one of the stickiest questions in writing a book about programming: how to deal with the mechanics of actually typing in and running a program. Most books silently skip over the issue, or touch it lightly, leaving the instructor with the burden of figuring out how to relate the book's material to the actual steps that students have to go through to solve the exercises. Instead, this book assumes the use of BlueJ and is able to integrate the tasks of understanding the concepts with the mechanics of how students can explore them.

I wish it had been around for my daughter last year. Maybe next year . . .

Preface

New to the sixth edition

This is the sixth edition of this book, and—as always with a new edition—the content has been adapted to the latest developments in object-oriented programs.

Many of the changes this time can, on the surface, be attributed to a new version of Java: Java 8. This version was released in 2014 and is now very widely used in practice. In fact, it is the fastest adoption of any new Java version ever released; so it is time also to change the way we teach novice students.

The changes are, however, more than merely the addition of a few new language constructs. The most significant new aspects in Java 8 center around new constructs to support a (partial) functional programming style. And it is the growing popularity of functional programming that is driving this change. The difference is much deeper, and much more fundamental, than just adding new syntax. And it is the renaissance of the functional ideas in modern programming generally—not only the existence of Java 8—that makes it timely to cover these aspects in a modern edition of a programming textbook.

The ideas and techniques of functional programming, while fairly old and well known in principle, have seen a marked boost of popularity in recent years, with new languages being developed and selected functional techniques being incorporated into existing, traditionally imperative languages. One of the primary reasons for this is the change in computing hardware available, and—with it—the changing nature of problems we wish to tackle.

Almost all programming platforms now are concurrent. Even mid-range laptops and mobile phones now have processors with multiple cores, making parallel processing a real possibility on everyday devices. But, in practice this is not happening on a large scale.

Writing applications that make optimal use of concurrent processing and multiple processors is very, very difficult. Most applications available today do not exploit current hardware to a degree approaching anything that is theoretically possible.

This is not going to change much: the opportunity (and challenge) of parallel hardware will remain, and programming these devices with traditional imperative languages will not get any easier.

This is where functional programming enters the picture.

With functional language constructs, it is possible to automate some concurrency very efficiently. Programs can potentially make use of multiple cores without much effort on the side of the programmer. Functional constructs have other advantages—more elegant expression for certain problems and often clearer readability—but it is the ability to deal with parallelism that will ensure that functional aspects of programming are going to stay with us for a long time to come.

Every teacher who wants to prepare their students for the future should give them some understanding of functional aspects as well. Without it, one will no longer be able to become a master programmer. A novice certainly does not have to master all of functional programming, but a basic understanding of what it is—and what we can achieve with it—is rapidly becoming essential.

Exactly when functional techniques should be introduced is an interesting question. We do not believe that there is a single right answer for this; various sequences are possible. Functional programming could be covered as an advanced topic at the end of the traditional corpus of this book, or it could be addressed when we first encounter the topics where it is applicable, as an alternative to the imperative techniques. It could even be covered first.

An additional question is how to treat the traditional style of programming in those areas where functional constructs are now available: should they be replaced, or do both need to be covered?

For this book, we recognize that different teachers will have different constraints and preferences. Therefore, we have designed a structure that—we hope—allows different approaches, depending on the preference of the learner or teacher.

- We have not replaced the "old-style" techniques. We cover the new, functional approach in addition to the existing material. Functional constructs in Java are most prominent when working with collections of objects, and the mastering traditional approach—using loops and explicit iteration—is still essential for any programmer. Not only are there millions of lines of code out there that are written in this style—and will be continued to be written in this style—but there are also specific cases where it is necessary to use these techniques even if one generally favors the new functional constructs. Mastering both is the goal.
- We present the new functional-construct-oriented material in the book where we discuss the problems that these constructs address. For example, we address functional collection processing as soon as we encounter collections.
- Chapters and sections covering this new material are, however, clearly marked as "advanced," and are structured in a manner that they can safely be skipped on first reading (or left out altogether).
- The previous two points enable different approaches to studying this book: if time permits, it can be read in the sequence it is presented, covering the full scope of material—including functional approaches as alternatives to imperative ones—as the problems are encountered which they address. If time is short, these advanced sections can be skipped, and emphasis can be placed on a thorough grounding in imperative, object-oriented programming. (We should emphasize that *functional* is not a contradiction to *object-oriented*: whether the functional material is included in the study, or the course emphasis

is largely on imperative techniques, every reader of this book will emerge with a good understanding of object orientation!) Yet another way to approach the material is to skip the advanced sections initially, and cover them as a separate unit at a later time. They present alternative approaches to other constructs and can be covered independently.

We hope this makes clear that this book provides flexibility where readers want it, but also guidance where a reader has no clear preference: just read it in the sequence it is written.

Apart from the major changes described so far, this edition also presents numerous minor improvements. The overall structure, tone, and approach of the book is unchanged; it has worked very well in the past, and there is no reason to deviate from it. However, we continuously re-evaluate and seek to improve where we see opportunities. We now have almost 15 years of continuous experience teaching with this book, and this is reflected in the many minor improvements throughout.

This book is an introduction to object-oriented programming for beginners. The main focus of the book is general object-oriented and programming concepts from a software engineering perspective.

While the first chapters are written for students with no programming experience, later chapters are suitable for more advanced or professional programmers as well. In particular, programmers with experience in a non-object-oriented language who wish to migrate their skills into object orientation should also be able to benefit from the book.

We use two tools throughout the book to enable the concepts introduced to be put into practice: the Java programming language and the Java development environment BlueJ.

Java

Java was chosen because of both its language design and its popularity. The Java programming language itself provides a clean implementation of most of the important object-oriented concepts, and serves well as an introductory teaching language. Its popularity ensures an immense pool of support resources.

In any subject area, having a variety of sources of information available is very helpful, for teachers and students alike. For Java in particular, countless books, tutorials, exercises, compilers, environments, and quizzes already exist, in many different kinds and styles. Many of them are online and many are available free of charge. The huge amount of high quality support material makes Java an excellent choice as an introduction to object-oriented programming.

With so much Java material already available, is there still room for more to be said about it? We think there is, and the second tool we use is one of the reasons . . .

BlueJ

BlueJ deserves much comment. This book is unique in its completely integrated use of the BlueJ environment.

BlueJ is a Java development environment that is being developed and maintained by the Computing Education Research Group at the University of Kent in Canterbury, UK,

explicitly as an environment for teaching introductory object-oriented programming. It is better suited to introductory teaching than other environments for a variety of reasons:

- The user interface is much simpler. Beginning students can typically use the BlueJ environment in a competent manner after 20 minutes of introduction. From then on, instruction can concentrate on the important concepts at hand—object orientation and Java—and no time needs to be wasted talking about environments, file systems, class paths, or DLL conflicts.
- The environment supports important teaching tools not available in other environments. One of them is visualization of class structure. BlueJ automatically displays a UML-like diagram representing the classes and relationships in a project. Visualizing these important concepts is a great help to both teachers and students. It is hard to grasp the concept of an object when all you ever see on the screen is lines of code! The diagram notation is a simple subset of UML, tailored to the needs of beginning students. This makes it easy to understand, but also allows migration to full UML in later courses.
- One of the most important strengths of the BlueJ environment is the user's ability to directly create objects of any class, and then to interact with their methods. This creates the opportunity for direct experimentation with objects, with little overhead in the environment. Students can almost "feel" what it means to create an object, call a method, pass a parameter, or receive a return value. They can try out a method immediately after it has been written, without the need to write test drivers. This facility is an invaluable aid in understanding the underlying concepts and language details.
- BlueJ includes numerous other tools and characteristics that are specifically designed for students of software development. Some are aimed at helping with understanding fundamental concepts (such as the scope highlighting in the editor), some are designed to introduce additional tools and techniques, such as integrated testing using JUnit, or teamwork using a version control system, such as Subversion, once the students are ready. Several of these features are unique to the BlueJ environment.

BlueJ is a full Java environment. It is not a cut-down, simplified version of Java for teaching. It runs on top of Oracle's Java Development Kit, and makes use of the standard compiler and virtual machine. This ensures that it always conforms to the official and most up-to-date Java specification.

The authors of this book have many years of teaching experience with the BlueJ environment (and many more years without it before that). We both have experienced how the use of BlueJ has increased the involvement, understanding, and activity of students in our courses. One of the authors is also the development lead of the BlueJ system.

Real objects first

One of the reasons for choosing BlueJ was that it allows an approach where teachers truly deal with the important concepts first. "Objects first" has been a battle cry for many textbook authors and teachers for some time. Unfortunately, the Java language does not make this noble goal very easy. Numerous hurdles of syntax and detail have to be overcome before

the first experience with a living object arises. The minimal Java program to create and call an object typically includes

- writing a class;
- writing a main method, including concepts such as static methods, parameters, and arrays in the signature;
- a statement to create the object ("new");
- an assignment to a variable;
- the variable declaration, including variable type;
- a method call, using dot notation;
- possibly a parameter list.

As a result, most textbooks typically either

- have to work their way through this forbidding list, and only reach objects somewhere around the fourth chapter; or
- use a "Hello, world"-style program with a single static main method as the first example, thus not creating any objects at all.

With BlueJ, this is not a problem. A student can create an object and call its methods as the very first activity! Because users can create and interact with objects directly, concepts such as classes, objects, methods, and parameters can easily be discussed in a concrete manner before looking at the first line of Java syntax. Instead of explaining more about this here, we suggest that the curious reader dip into Chapter 1—things will quickly become clear then.

An iterative approach

Another important aspect of this book is that it follows an iterative style. In the computing education community, a well-known educational design pattern exists that states that important concepts should be taught early and often.[1] It is very tempting for textbook authors to try and say everything about a topic at the point where it is introduced. For example, it is common, when introducing types, to give a full list of built-in data types, or to discuss all available kinds of loop when introducing the concept of a loop.

These two approaches conflict: we cannot concentrate on discussing important concepts first, and at the same time provide complete coverage of all topics encountered. Our experience with textbooks is that much of the detail is initially distracting, and has the effect of drowning the important points, thus making them harder to grasp.

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

In this book we touch on all of the important topics several times, both within the same chapter and across different chapters. Concepts are usually introduced at a level of detail necessary for understanding and applying to the task at hand. They are revisited later in a different context, and understanding deepens as the reader continues through the chapters. This approach also helps to deal with the frequent occurrence of mutual dependencies between concepts.

Some teachers may not be familiar with the iterative approach. Looking at the first few chapters, teachers used to a more sequential introduction will be surprised about the number of concepts touched on this early. It may seem like a steep learning curve.

It is important to understand that this is not the end of the story. Students are not expected to understand everything about these concepts immediately. Instead, these fundamental concepts will be revisited again and again throughout the book, allowing students to get a deeper understanding over time. Since their knowledge level changes as they work their way forward, revisiting important topics later allows them to gain a deeper understanding overall.

We have tried this approach with students many times. Sometimes students have fewer problems dealing with it than some long-time teachers. And remember: a steep learning curve is not a problem as long as you ensure that your students can climb it!

No complete language coverage

Related to our iterative approach is the decision not to try to provide complete coverage of the Java language within the book.

The main focus of this book is to convey object-oriented programming principles in general, not Java language details in particular. Students studying with this book may be working as software professionals for the next 30 or 40 years of their life—it is a fairly safe bet that the majority of their work will not be in Java. Every serious textbook must attempt to prepare them for something more fundamental than the language flavor of the day.

On the other hand, many Java details are essential to actually doing practical programming work. In this book we cover Java constructs in as much detail as is necessary to illustrate the concepts at hand and implement the practical work. Some constructs specific to Java have been deliberately left out of the discussion.

We are aware that some instructors will choose to cover some topics that we do not discuss in detail. However, instead of trying to cover every possible topic ourselves (and thus blowing the size of this book out to 1500 pages), we deal with it using *hooks*. Hooks are pointers, often in the form of questions that raise the topic and give references to an appendix or outside material. These hooks ensure that a relevant topic is brought up at an appropriate time, and leave it up to the reader or the teacher to decide to what level of detail that topic should be covered. Thus, hooks serve as a reminder of the existence of the topic, and as a placeholder indicating a point in the sequence where discussion can be inserted.

Individual teachers can decide to use the book as it is, following our suggested sequence, or to branch out into sidetracks suggested by the hooks in the text.

Chapters also often include several questions suggesting discussion material related to the topic, but not discussed in this book. We fully expect teachers to discuss some of these questions in class, or students to research the answers as homework exercises.

Project-driven approach

The introduction of material in the book is project driven. The book discusses numerous programming projects and provides many exercises. Instead of introducing a new construct and then providing an exercise to apply this construct to solve a task, we first provide a goal and a problem. Analyzing the problem at hand determines what kinds of solutions we need. As a consequence, language constructs are introduced as they are needed to solve the problems before us.

Early chapters provide at least two discussion examples. These are projects that are discussed in detail to illustrate the important concepts of each chapter. Using two very different examples supports the iterative approach: each concept is revisited in a different context after it is introduced.

In designing this book we have tried to use a lot of different example projects. This will hopefully serve to capture the reader's interest, and also illustrate the variety of different contexts in which the concepts can be applied. We hope that our projects serve to give teachers good starting points and many ideas for a wide variety of interesting assignments.

The implementation for all our projects is written very carefully, so that many peripheral issues may be studied by reading the projects' source code. We are strong believers in learning by reading and imitating good examples. For this to work, however, it's important that the examples are well written and worth imitating. We have tried to create great examples.

All projects are designed as open-ended problems. While one or more versions of each problem are discussed in detail in the book, the projects are designed so that further extensions and improvements can be done as student projects. Complete source code for all projects is included. A list of projects discussed in this book is provided on page xxv.

Concept sequence rather than language constructs

One other aspect that distinguishes this book from many others is that it is structured along fundamental software development tasks, not necessarily according to the particular Java language constructs. One indicator of this is the chapter headings. In this book you will not find traditional chapter titles such as "Primitive data types" or "Control structures." Structuring by fundamental development tasks allows us to present a more general introduction that is not driven by intricacies of the particular programming language utilized. We also believe that it is easier for students to follow the motivation of the introduction, and that it makes much more interesting reading.

As a result of this approach, it is less straightforward to use the book as a reference book. Introductory textbooks and reference books have different, partly competing, goals. To a

certain extent a book can try to be both, but compromises have to be made at certain points. Our book is clearly designed as a textbook, and wherever a conflict occurred, the textbook style took precedence over its use as a reference book.

We have, however, provided support for use as a reference book by listing the Java constructs introduced in each chapter in the chapter introduction.

Chapter sequence

Chapter 1 deals with the most fundamental concepts of object orientation: objects, classes, and methods. It gives a solid, hands-on introduction to these concepts without going into the details of Java syntax. We briefly introduce the concept of abstraction. This will be a thread that runs through many chapters. Chapter 1 also gives a first look at some source code. We do this by using an example of graphical shapes that can be interactively drawn, and a second example of a simple laboratory class enrollment system.

Chapter 2 opens up class definitions and investigates how Java source code is written to create behavior of objects. We discuss how to define fields and implement methods, and point out the crucial role of the constructor in setting up an object's state as embodied in its fields. Here, we also introduce the first types of statement. The main example is an implementation of a ticket machine. We also investigate the laboratory class example from Chapter 1 a bit further.

Chapter 3 then enlarges the picture to discuss interaction of multiple objects. We see how objects can collaborate by invoking each other's methods to perform a common task. We also discuss how one object can create other objects. A digital alarm clock display is discussed that uses two number display objects to show hours and minutes. A version of the project that includes a GUI picks up on a running theme of the book—that we often provide additional code for the interested and able student to explore, without covering it in detail in the text. As a second major example, we examine a simulation of an email system in which messages can be sent between mail clients.

In Chapter 4 we continue by building more extensive structures of objects and pick up again on the themes of abstraction and object interaction from the preceding chapters. Most importantly, we start using collections of objects. We implement an organizer for music files and an auction system to introduce collections. At the same time, we discuss iteration over collections, and have a first look at the *for-each* and *while* loops. The first collection being used is an `ArrayList`.

Chapter 5 presents the first advanced section (a section that can be skipped if time is short): It is an introduction to functional programming constructs. The functional constructs present an alternative to the imperative collection processing discussed in Chapter 4. The same problems can be solved without these techniques, but functional constructs open some more elegant ways to achieve our goals. This chapter gives an introduction to the functional approach in general, and introduces a few of Java's language constructs.

Chapter 6 deals with libraries and interfaces. We introduce the Java library and discuss some important library classes. More importantly, we explain how to read and understand the library documentation. The importance of writing documentation in software development

projects is discussed, and we end by practicing how to write suitable documentation for our own classes. `Random`, `Set`, and `Map` are examples of classes that we encounter in this chapter. We implement an *Eliza*-like dialog system and a graphical simulation of a bouncing ball to apply these classes.

Chapter 7 concentrates on one specific—but very special—type of collection: arrays. Arrays processing and the associated types of loops are discussed in detail.

In Chapter 8 we discuss more formally the issues of dividing a problem domain into classes for implementation. We introduce issues of good class design, including concepts such as responsibility-driven design, coupling, cohesion, and refactoring. An interactive, text-based adventure game (*World of Zuul*) is used for this discussion. We go through several iterations of improving the internal class structure of the game and extending its functionality, and end with a long list of proposals for extensions that may be done as student projects.

Chapter 9 deals with a whole group of issues connected to producing correct, understandable, and maintainable classes. It covers issues ranging from writing clear code (including style and commenting) to testing and debugging. Test strategies are introduced, including formalized regression testing using JUnit, and a number of debugging methods are discussed in detail. We use an example of an online shop and an implementation of an electronic calculator to discuss these topics.

Chapters 10 and 11 introduce inheritance and polymorphism, with many of the related detailed issues. We discuss a part of a social network to illustrate the concepts. Issues of code inheritance, subtyping, polymorphic method calls, and overriding are discussed in detail.

In Chapter 12 we implement a predator/prey simulation. This serves to discuss additional abstraction mechanisms based on inheritance, namely interfaces and abstract classes.

Chapter 13 develops an image viewer and a graphical user interface for the music organizer (first encountered in Chapter 4). Both examples serve to discuss how to build graphical user interfaces (GUIs).

Chapter 14 then picks up the difficult issue of how to deal with errors. Several possible problems and solutions are discussed, and Java's exception-handling mechanism is discussed in detail. We extend and improve an address book application to illustrate the concepts. Input/output is used as a case study where error-handling is an essential requirement.

Chapter 15 discusses in more detail the next level of abstraction: How to structure a vaguely described problem into classes and methods. In previous chapters we have assumed that large parts of the application structure already exist, and we have made improvements. Now it is time to discuss how we can get started from a clean slate. This involves detailed discussion of what the classes should be that implement our application, how they interact, and how responsibilities should be distributed. We use Class/Responsibilities/Collaborators (CRC) cards to approach this problem, while designing a cinema booking system.

In Chapter 16 we to bring everything together and integrate topics from the previous chapters of the book. It is a complete case study, starting with the application design, through design of the class interfaces, down to discussing many important functional and non-functional characteristics and implementation details. Concepts discussed in earlier chapters (such as reliability, data structures, class design, testing, and extendibility) are applied again in a new context.

Supplements

VideoNotes: VideoNotes are Pearson's new visual tool designed to teach students key programming concepts and techniques. These short step-by-step videos demonstrate how to solve problems from design through coding. VideoNotes allow for self-paced instruction with easy navigation including the ability to select, play, rewind, fast-forward, and stop within each VideoNote exercise.

VideoNotes are located at `http://www.pearsonhighered.com/barnes_kolling`. Six months of prepaid access are included with the purchase of a new textbook. If the access code has already been revealed, it may no longer be valid. If this is the case, you can purchase a subscription by going to `http://www.pearsonhighered.com/barnes_kolling/` and following the on-screen instructions.

Book website: All projects used as discussion examples and exercises in this book are available for download on the book's website, at `http://www.bluej.org/objects-first/`. The website also provides links to download BlueJ, and other resources..

Companion website for students: The following resources are available to all readers of this book at its Companion Website, located at `http://www.pearsonhighered.com/barnes_kolling`:

- Program style guide for all examples in the book
- Links to further material of interest
- Complete source code for all projects

Instructor resources: The following supplements are available to qualified instructors only:

- Solutions to end-of-chapter exercises
- PowerPoint slides

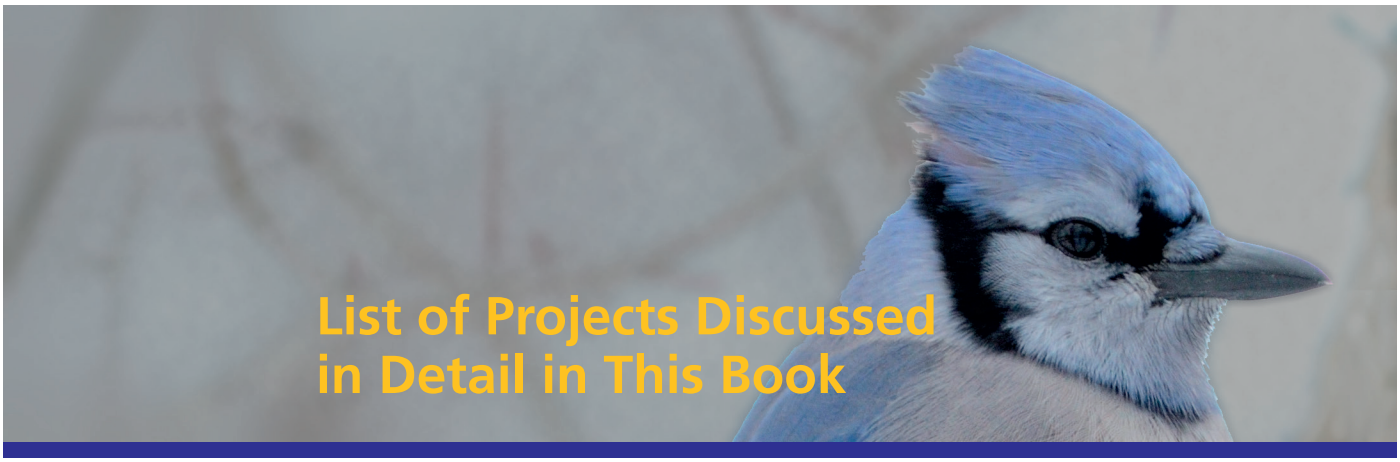
Visit the Pearson Instructor Resource Center at www.pearsonhighered.com/irc to register for access or contact your local Pearson representative.

The Blueroom

Perhaps more important than the static web site resources is a very active community forum that exists for instructors who teach with BlueJ and this book. It is called the *Blueroom* and can be found at

```
http://blueroom.bluej.org
```

The Blueroom contains a resource collection with many teaching resources shared by other teachers, as well as a discussion forum where instructors can ask questions, discuss issues, and stay up-to-date with the latest developments. Many other teachers, as well as developers of BlueJ and the authors of this book, can be contacted in the Blueroom.



List of Projects Discussed in Detail in This Book

Figures (Chapter 1)

Simple drawing with some geometrical shapes; illustrates creation of objects, method calling, and parameters.

House (Chapter 1)

An example using shape objects to draw a picture; introduces source code, Java syntax, and compilation.

Lab-classes (Chapter 1, 2, 10)

A simple example with classes of students; illustrates objects, fields, and methods. Used again in Chapter 10 to add inheritance.

Ticket-machine (Chapter 2)

A simulation of a ticket vending machine for train tickets; introduces more about fields, constructors, accessor and mutator methods, parameters, and some simple statements.

Book-exercise (Chapter 2)

Storing details of a book. Reinforcing the constructs used in the ticket-machine example.

Clock-display (Chapter 3)

An implementation of a display for a digital clock; illustrates the concepts of abstraction, modularization, and object interaction. Includes a version with an animated GUI.

Mail system (Chapter 3)

A simple simulation of an email system. Used to demonstrate object creation and interaction.

Music-organizer (Chapter 4, 11)

An implementation of an organizer for music tracks; used to introduce collections and loops. Includes the ability to play MP3 files. A GUI is added in Chapter 11.

Auction (Chapter 4)

An auction system. More about collections and loops, this time with iterators.

Animal-monitoring (Chapter 5)

A system to monitor animal populations, e.g., in a national park. This is used to introduce functional processing of collections.

Tech-support (Chapter 6, 14)

An implementation of an *Eliza*-like dialog program used to provide "technical support" to customers; introduces use of library classes in general and some specific classes in particular; reading and writing of documentation.

Scribble (Chapter 6)

A shape-drawing program to support learning about classes from their interfaces.

Bouncing-balls (Chapter 6)

A graphical animation of bouncing balls; demonstrates interface/implementation separation and simple graphics.

Weblog-analyzer (Chapter 7, 14)

A program to analyze web access log files; introduces arrays and for loops.

Automaton (Chapter 7)

A series of examples of a cellular automaton. Used to gain practice with array programming.

Brain (Chapter 7)

A version of *Brian's Brain*, which we use to discuss two-dimensional arrays.

World-of-zuul (Chapter 8, 11)

A text-based, interactive adventure game. Highly extendable, makes a great open-ended student project. Used here to discuss good class design, coupling, and cohesion. Used again in Chapter 9 as an example for use of inheritance.

Online-shop (Chapter 9)

The early stages of an implementation of a part of an online shopping website, dealing with user comments; used to discuss testing and debugging strategies.

Calculator (Chapter 9)

An implementation of a desk calculator. This example reinforces concepts introduced earlier, and is used to discuss testing and debugging.

Bricks (Chapter 9)

A simple debugging exercise; models filling pallets with bricks for simple computations.

Network (Chapter 10, 11)

Part of a social network application. This project is discussed and then extended in great detail to introduce the foundations of inheritance and polymorphism.

Foxes-and-rabbits (Chapter 12)

A classic predator–prey simulation; reinforces inheritance concepts and adds abstract classes and interfaces.

Image-viewer (Chapter 13)

A simple image-viewing and -manipulation application. We concentrate mainly on building the GUI.

Music-player (Chapter 13)

A GUI is added to the *music-organizer* project of Chapter 4 as another example of building GUIs.

Address-book (Chapter 14)

An implementation of an address book with an optional GUI interface. Lookup is flexible: entries can be searched by partial definition of name or phone number. This project makes extensive use of exceptions.

Cinema-booking-system (Chapter 15)

A system to manage advance seat bookings in a cinema. This example is used in a discussion of class discovery and application design. No code is provided, as the example represents the development of an application from a blank sheet of paper.

Taxi-company (Chapter 16)

The taxi example is a combination of a booking system, management system, and simulation. It is used as a case study to bring together many of the concepts and techniques discussed throughout the book.