

Objects First with Java



We work with leading authors to develop the strongest educational materials in computing, bringing cutting-edge thinking and best learning practice to a global market.

Under a range of well-known imprints, including Prentice Hall, we craft high quality print and electronic publications that help readers to understand and apply their content, whether studying or at work.

To find out more about the complete range of our publishing, please visit us on the World Wide Web at: www.pearsoneduc.com



Objects First with Java

A Practical Introduction using BlueJ

David J. Barnes and Michael Kölling



An imprint of Pearson Education

Harlow, England • London • New York • Boston • San Francisco • Toronto
Sydney • Tokyo • Singapore • Hong Kong • Seoul • Taipei • New Delhi
Cape Town • Madrid • Mexico City • Amsterdam • Munich • Paris • Milan

Pearson Education Limited

Edinburgh Gate
Harlow
Essex CM20 2JE

and Associated Companies throughout the world

Visit us on the World Wide Web at:
www.pearsoneduc.com

First published 2003

© Pearson Education Limited 2003

The rights of David J. Barnes and Michael Kölling to be identified as authors of this work have been asserted by them in accordance with the Copyright, Designs and Patents Act 1988.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without either the prior written permission of the publisher or a licence permitting restricted copying in the united Kingdom issued by the Copyright Licensing Agency Ltd, 90 Tottenham Court Road, London W1P 0LP.

The programs in this book have been included for their instructional value. They have been tested with care but are not guaranteed for any particular purpose. The publisher does not offer any warranties or representations nor does it accept any liabilities with respect to the programs.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Pearson Education has made every attempt to supply trademark information about manufacturers and their products mentioned in this book. Java™ and Sun™ are trademarks or registered trademarks of Sun Microsystems Inc. Star Wars® is a registered trademark of Lucasfilm Ltd.

ISBN 0130 44929 6

British Library Cataloguing-in-Publication Data

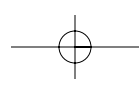
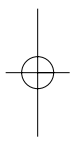
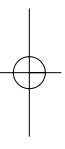
A catalogue record for this book is available from the British Library

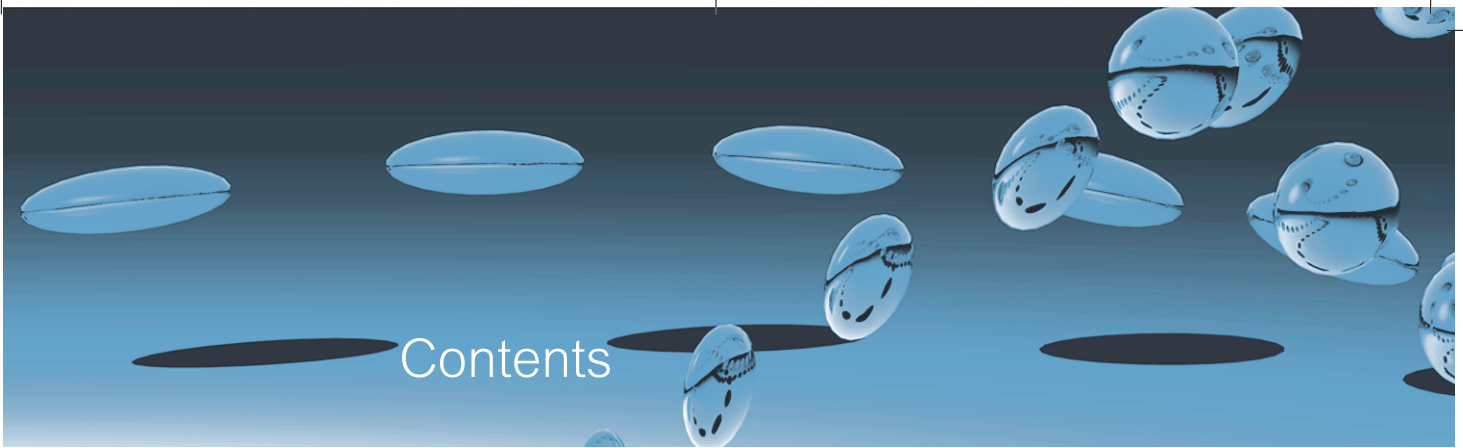
10 9 8 7 6 5 4 3 2 1
07 06 05 04 03

Typeset in 10/12pt TimesNewRomanPS by 30
Printed by Ashford Colour Press Ltd., Gosport

To my family
Helen, John, Hannah, Ben and Sarah
djb

To my family
Leah, Sophie and Feena
mk





Foreword by James Gosling, Sun Microsystems	xvii
Preface to the instructor	xviii
List of projects discussed in detail in this book	xxv
Acknowledgements	xxvii

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	8
1.8	What is in an object?	9
1.9	Object interaction	10
1.10	Source code	11
1.11	Another example	13
1.12	Return values	13
1.13	Objects as parameters	13
1.14	Summary	15
Chapter 2	Understanding class definitions	17
2.1	Ticket machines	17
2.1.1	Exploring the behavior of a naïve ticket machine	18
2.2	Examining a class definition	19
2.3	Fields, constructors and methods	21
2.3.1	Fields	22

2.3.2	Constructors	24
2.4	Passing data via parameters	25
2.5	Assignment	27
2.6	Accessor methods	27
2.7	Mutator methods	29
2.7	Printing from methods	31
2.9	Summary of the naïve ticket machine	33
2.10	Reflecting on the design of the ticket machine	34
2.11	Making choices: the conditional statement	35
2.12	A further conditional-statement example	38
2.13	Local variables	39
2.14	Fields, parameters and local variables	40
2.15	Summary of the better ticket machine	41
2.16	Reviewing a familiar example	42
2.17	Summary	45
Chapter 3	Object interaction	49
3.1	The clock example	49
3.2	Abstraction and modularization	50
3.3	Abstraction in software	51
3.4	Modularization in the clock example	51
3.5	Implementing the clock display	52
3.6	Class diagrams versus object diagrams	53
3.7	Primitive types and object types	54
3.8	The <code>ClockDisplay</code> source code	56
3.8.1	Class <code>NumberDisplay</code>	56
3.8.2	String concatenation	58
3.8.3	The modulo operator	59
3.8.4	Class <code>ClockDisplay</code>	59
3.9	Objects creating objects	62
3.10	Multiple constructors	64
3.11	Method calls	64
3.11.1	Internal method calls	64
3.11.2	External method calls	65
3.11.3	Summary of the clock display	66
3.12	Another example of object interaction	67
3.12.1	The mail system example	67
3.12.2	The <code>this</code> key word	68
3.13	Using a debugger	70
3.13.1	Setting breakpoints	71

3.13.2	Single stepping	72
3.13.3	Stepping into methods	74
3.14	Method calling revisited	74
3.15	Summary	75
Chapter 4	Grouping objects	77
4.1	Grouping objects in flexible-size collections	77
4.2	A personal notebook	78
4.3	A first look at library classes	78
4.3.1	An example of using a library	79
4.4	Object structures with collections	80
4.5	Numbering within collections	82
4.6	Removing an item from a collection	83
4.7	Processing a whole collection	84
4.7.1	The while loop	85
4.7.2	Iterating over a collection	87
4.7.3	Index access versus iterators	88
4.8	Summary of the notebook example	88
4.9	Another example: an auction system	89
4.9.1	The <code>Lot</code> class	89
4.9.2	The <code>Auction</code> class	90
4.9.3	Casting	92
4.10	Fixed-size collections	94
4.10.1	A log-file analyzer	94
4.10.2	Declaring array variables	96
4.10.3	Creating array objects	97
4.10.4	Using array objects	98
4.10.5	Analyzing the log file	99
4.10.6	The for loop	100
4.11	Summary	103
Chapter 5	More sophisticated behavior	105
5.1	Documentation for library classes	106
5.2	The TechSupport system	106
5.2.1	Exploring the TechSupport system	107
5.2.2	Reading the code	108
5.3	Reading class documentation	112
5.3.1	Interfaces versus implementation	113
5.3.2	Using library-class methods	114

5.3.3	Checking string equality	115
5.4	Adding random behavior	116
5.4.1	The <code>Random</code> class	117
5.4.2	Random numbers with limited range	118
5.4.3	Generating random responses	119
5.5	Packages and import	121
5.6	Using maps for associations	122
5.6.1	The concept of a map	123
5.6.2	Using a <code>HashMap</code>	123
5.6.3	Using a map for the <code>TechSupport</code> system	124
5.7	Using sets	126
5.8	Tokenizing strings	127
5.9	Finishing the <code>TechSupport</code> system	129
5.10	Writing class documentation	130
5.10.1	Using <code>javadoc</code> in <code>BlueJ</code>	131
5.10.2	Elements of class documentation	131
5.11	Public versus private	133
5.11.1	Information hiding	133
5.11.2	Private methods and public fields	134
5.12	Learning about classes from their interfaces	135
5.13	Class variables and constants	138
5.13.1	The <code>static</code> key word	138
5.13.2	Constants	139
5.14	Summary	140
Chapter 6	Well-behaved objects	143
6.1	Testing and debugging	144
6.2	Unit testing within <code>BlueJ</code>	144
6.2.1	Using inspectors	148
6.2.2	Positive versus negative testing	150
6.3	Test automation	150
6.3.1	Regression testing	150
6.3.2	Automated checking of test results	153
6.4	Modularization and interfaces	154
6.5	A debugging scenario	156
6.6	Commenting and style	157
6.7	Manual walkthroughs	158
6.7.1	A high-level walkthrough	158
6.7.2	Checking state with a walkthrough	160
6.7.3	Verbal walkthroughs	162
6.8	Print statements	163

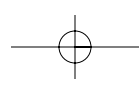
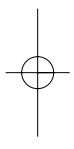
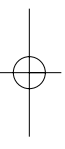
6.8.1	Turning debugging information on or off	165
6.9	Debuggers	166
6.10	Putting the techniques into practice	167
6.11	Summary	167
Chapter 7	Designing classes	169
7.1	Introduction	170
7.2	The world-of-zuul game example	171
7.3	Introduction to coupling and cohesion	173
7.4	Code duplication	174
7.5	Making extensions	177
7.5.1	The task	177
7.5.2	Finding the relevant source code	177
7.6	Coupling	179
7.6.1	Using encapsulation to reduce coupling	179
7.7	Responsibility-driven design	183
7.7.1	Responsibilities and coupling	184
7.8	Localizing change	186
7.9	Implicit coupling	186
7.10	Thinking ahead	189
7.11	Cohesion	190
7.11.1	Cohesion of methods	190
7.11.2	Cohesion of classes	191
7.11.3	Cohesion for readability	192
7.11.4	Cohesion for reuse	193
7.12	Refactoring	194
7.12.1	Refactoring and testing	194
7.12.2	An example of refactoring	195
7.13	Design guidelines	197
7.14	Executing without BlueJ	199
7.14.1	Class methods	199
7.14.2	The main method	199
7.14.3	Limitations of class methods	200
7.15	Summary	200
Part 2	Application Structures	203
Chapter 8	Improving structure with inheritance	205
8.1	The DoME example	205

8.1.1	DoME classes and objects	206
8.1.2	DoME source code	208
8.1.3	Discussion of the DoME application	214
8.2	Using Inheritance	215
8.3	Inheritance hierarchies	216
8.4	Inheritance in Java	217
8.4.1	Inheritance and access rights	218
8.4.2	Inheritance and initialization	218
8.5	DoME: adding other item types	220
8.6	Advantages of inheritance (so far)	222
8.7	Subtyping	223
8.7.1	Subclasses and subtypes	224
8.7.2	Subtyping and assignment	225
8.7.3	Subtyping and parameter passing	226
8.7.4	Polymorphic variables	227
8.8	The <code>Object</code> class	227
8.9	Polymorphic collections	228
8.9.1	Element types	229
8.9.2	Casting revisited	229
8.9.3	Wrapper classes	230
8.10	The collection hierarchy	231
8.11	Summary	232
Chapter 9	More about inheritance	235
9.1	The problem: DoME's <code>print</code> method	235
9.2	Static type and dynamic type	237
9.2.1	Calling <code>print</code> from <code>Database</code>	238
9.3	Overriding	239
9.4	Dynamic method lookup	241
9.5	Super call in methods	244
9.6	Method polymorphism	245
9.7	Object methods: <code>toString</code>	245
9.8	Protected access	248
9.9	Another example of inheritance with overriding	250
9.10	Summary	253
Chapter 10	Further abstraction techniques	255
10.1	Simulations	255
10.2	The foxes-and-rabbits simulation	256

10.2.1	The foxes-and-rabbits project	257
10.2.2	The <code>Rabbit</code> class	259
10.2.3	The <code>Fox</code> class	262
10.2.4	The <code>Simulator</code> class: setup	265
10.2.5	The <code>Simulator</code> class: a simulation step	268
10.2.6	Taking steps to improve the simulation	270
10.3	Abstract classes	270
10.3.1	The <code>Animal</code> superclass	271
10.3.2	Abstract methods	271
10.3.3	Abstract classes	274
10.4	More abstract methods	275
10.5	Multiple inheritance	277
10.5.1	An <code>Actor</code> class	277
10.5.2	Flexibility through abstraction	278
10.5.3	Selective drawing	279
10.5.4	Drawable actors: multiple inheritance	280
10.6	Interfaces	280
10.6.1	An <code>Actor</code> interface	280
10.6.2	Multiple inheritance of interfaces	281
10.6.3	Interfaces as types	282
10.6.4	Interfaces as specifications	283
10.6.5	A further example of interfaces	284
10.6.6	Abstract class or interface?	285
10.7	Summary of inheritance	285
10.8	Summary	286
Chapter 11	Handling errors	287
11.1	The address-book project	288
11.2	Defensive programming	292
11.2.1	Client-server interaction	292
11.2.2	Argument checking	293
11.3	Server error reporting	295
11.3.1	Notifying the user	295
11.3.2	Notifying the client object	296
11.4	Exception-throwing principles	299
11.4.1	Throwing an exception	299
11.4.2	Exception classes	300
11.4.3	The effect of an exception	301

11.4.4	Unchecked exceptions	302
11.4.5	Preventing object creation	303
11.5	Exception handling	304
11.5.1	Checked exceptions: the throws clause	304
11.5.2	Catching exceptions: the try block	305
11.5.3	Throwing and catching multiple exceptions	307
11.5.4	Propagating an exception	308
11.5.5	The finally clause	308
11.6	Defining new exception classes	309
11.7	Error recovery and avoidance	311
11.7.1	Error recovery	311
11.7.2	Error avoidance	312
11.8	Case study: text input/output	314
11.8.1	Readers, writers and streams	314
11.8.2	The address-book-io project	315
11.8.3	Text output with <code>FileWriter</code>	317
11.8.4	Text input with <code>FileReader</code>	318
11.8.5	Object serialization	319
11.9	Summary	320
Chapter 12	Designing applications	321
12.1	Analysis and design	321
12.1.1	The verb/noun method	322
12.1.2	The cinema booking example	322
12.1.3	Discovering classes	322
12.1.4	Using CRC cards	323
12.1.5	Scenarios	324
12.2	Class design	327
12.2.1	Designing class interfaces	328
12.2.2	User interface design	329
12.3	Documentation	329
12.4	Cooperation	330
12.5	Prototyping	330
12.6	Software growth	331
12.6.1	Waterfall model	331
12.6.2	Iterative development	331
12.7	Using design patterns	333
12.7.1	Structure of a pattern	333
12.7.2	Decorator	334

12.7.3 Singleton	334
12.7.4 Factory method	335
12.7.5 Observer	336
12.7.6 Pattern summary	337
12.8 Summary	338
Chapter 13 A case study	341
13.1 The case study	341
13.1.1 The problem description	341
13.2 Analysis and design	342
13.2.1 Discovering classes	342
13.2.2 Using CRC cards	343
13.2.3 Scenarios	344
13.3 Class design	346
13.3.1 Designing class interfaces	346
13.3.2 Collaborators	346
13.3.3 The outline implementation	347
13.3.4 Testing	351
13.3.5 Some remaining issues	351
13.4 Iterative development	351
13.4.1 Development steps	351
13.4.2 A first stage	352
13.4.3 Testing the first stage	356
13.4.4 A later stage of development	357
13.4.5 Further ideas for development	358
13.4.6 Reuse	359
13.5 Another example	359
13.6 Taking things further	360
Appendices	
A Working with a BlueJ project	361
B Java data types	363
C Java control structures	365
D Operators	369
E Running Java without BlueJ	371
F Configuring BlueJ	375
Index	377





Foreword by James Gosling, Sun Microsystems

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.

None the less, 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 it up to the instructor to figure out how to solve the problem. And leaving the instructor with the burden of relating the material being taught to the steps that students have to go through to work on the exercises. Instead, it 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 to the instructor

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 a combination of two aspects: the language design and its popularity. The Java programming language itself provides a very 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 large amount and good quality of 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

The second tool, BlueJ, deserves more comment. This book is unique in its completely integrated use of the BlueJ environment.

BlueJ is a Java development environment that was designed at Monash University, Australia, 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, DOS commands 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, again 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, for 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 is a full Java environment. It is not a cut-down, simplified version of Java for teaching. It runs on top of Sun Microsystems' 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 several 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 a developer 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, textbooks typically either

- have to work their way through this forbidding list, and only reach objects somewhere around Chapter 4; 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.

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 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 dependences between concepts.

Some teachers may not be familiar with an 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 and 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.

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.

We have tried this approach with students many times. It seems that 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 of course attempt to prepare them for something more fundamental than the language flavor of the day.

On the other hand, many Java details are important for actually doing the practical 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. That is expected and necessary. 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 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 large number and wide variety of different example projects. This will hopefully serve to capture the reader's interest, but it also helps to illustrate the variety of different contexts in which the concepts can be applied. Finding good example projects is hard. 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 the benefit of learning by reading and imitating good examples. For this to work, however, one must make sure that the examples students read are well written and worth imitating. We have tried to do this.

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 and not necessarily according to the particular Java language constructs. One indicator of this is the chapter headings. In this book you will not find many of the traditional chapter titles, such as 'Primitive data types' or 'Control structures'. Structuring by fundamental development tasks allows us to give a much 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 it much more interesting.

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. It 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. Here, we also introduce the first types of statement. The main example is an implementation of a ticket machine. We also look back to the laboratory class example from Chapter 1 to investigate that 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. 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. Most importantly, we start using collections of objects. We implement an electronic notebook and an auction system to introduce collections. At the same time, we discuss iteration over collections and have a first look at loops. The first collection being used is an `ArrayList`. In the second half of the chapter we introduce arrays as a special form of a collection, and the `for` loop as another form of a loop. We discuss an implementation of a web-log analyzer as an example for array use.

Chapter 5 deals with libraries and interfaces. We introduce the Java standard 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 6, titled *Well-behaved objects*, deals with a whole group of issues connected to producing correct, understandable and maintainable classes. It covers issues ranging from writing clear, understandable code – including style and commenting – to testing and debugging. Test strategies are introduced, and a number of debugging methods are discussed in detail. We use an example of a diary for appointment scheduling and an implementation of an electronic calculator to discuss these topics.

In Chapter 7 we discuss more formally the issues of dividing a problem domain into classes for implementation. We introduce issues of designing classes well, 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.

Chapters 8 and 9 introduce inheritance and polymorphism with many of the related detailed issues. We discuss a simple database of CDs and videos to illustrate the concepts. Issues of code inheritance, subtyping, polymorphic method calls and overriding are discussed in detail.

In Chapter 10 we implement a predator/prey simulation. This serves to discuss additional abstraction mechanisms based on inheritance, namely interfaces and abstract classes.

Chapter 11 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.

Chapter 12 steps back to discuss 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 13 we try to bring everything together and integrate many 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. Topics discussed in earlier chapters (such as reliability, data structures, class design, testing and extendibility) are applied again in a new context.

Discussion group

The authors maintain an active email discussion group for the purpose of facilitating exchange of ideas and mutual support for and by readers of this book and other BlueJ users. Postings to this list are archived and publicly accessible. Using this list, teachers can receive support and ideas from other teachers and the authors of this book. The mail address for this list is `bluej-discuss@bluej.org`. Interested people can join the list or browse the archives at

<http://lists.bluej.org/mailman/listinfo/bluej-discuss>

Additional material

This book includes all projects used as discussion examples and exercises on a CD. The CD also includes the Java development environment (JDK) and BlueJ for various operating systems.

There is a support web site for this book at

<http://www.bluej.org/objects-first>

On this web site, updates to the examples can be found, and additional material is provided. For instance, the style guide used for all examples in this book is available on the web site in electronic form, so that instructors can modify it to meet their own requirements.

The web site also includes a password-protected, teacher-only section that provides additional material.

A set of slides to teach a course with this book is also provided.



List of projects discussed in detail in this book

shapes

Chapter 1

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

picture

Chapter 1

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

lab-classes

Chapter 1, Chapter 2, Chapter 8

A simple example with classes of students; illustrates objects, fields and methods. Used again in Chapter 8 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

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.

mail system

Chapter 3

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

notebook

Chapter 4

An implementation of a (simple) electronic notebook; used to introduce collections and loops.

auction

Chapter 4

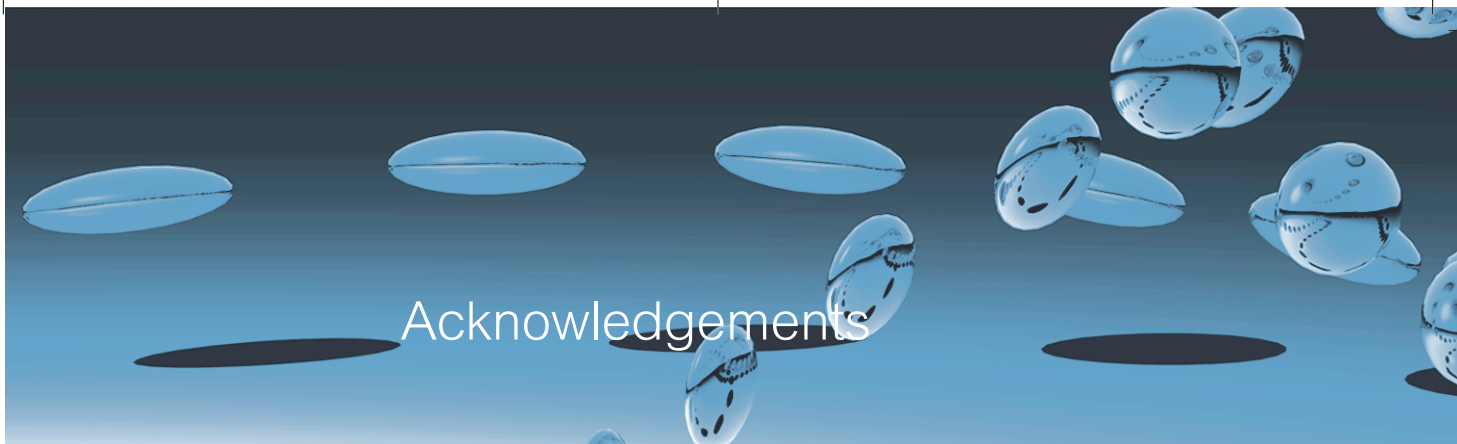
An auction system. More about collections and loops, this time with iterators.

weblog-analyzer

Chapter 4

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

- [tech-support](#) Chapter 5
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.
- [balls](#) Chapter 5
A graphical animation of bouncing balls; demonstrates interface/implementation separation and simple graphics.
- [diary](#) Chapter 6
The early stages of an implementation of a diary storing appointments; used to discuss testing and debugging strategies.
- [calculator](#) Chapter 6
An implementation of a desk calculator. This example reinforces concepts introduced earlier, and is used to discuss testing and debugging.
- [bricks](#) Chapter 6
A simple debugging exercise; models filling palettes with bricks for simple computations.
- [world-of-zuul](#) Chapter 7, Chapter 9
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.
- [dome](#) Chapter 8, Chapter 9
A database of CDs and videos. This project is discussed and then extended in great detail to introduce the foundations of inheritance and polymorphism.
- [foxes-and-rabbits](#) Chapter 10
A classic predator–prey simulation; reinforces inheritance concepts and adds abstract classes and interfaces.
- [address-book](#) Chapter 11
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 12
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](#) Chapter 13
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.



Acknowledgements

Many people have contributed in many different ways to this book and made its creation possible.

First, and most importantly, John Rosenberg must be mentioned. John is Dean of the Faculty of Information Technology of Monash University, Australia. It is by mere coincidence of circumstance that John is not one of the authors of this book. He was one of the driving forces in the development of BlueJ and the ideas and pedagogy behind it from the very beginning, and we talked about the writing of this book for several years. Much of the material in this book was developed in discussions with John. Simply the fact that there are only twenty-four hours in a day, too many of which were already taken up with too much other work, prevented him from actually writing this book. John has contributed to this text continuously while it was being written and helped improve it in many ways. John is a great ambassador for Monash and an excellent host. We have appreciated his friendship and collaboration immensely.

The two other people who helped make BlueJ what it is are Bruce Quig and Andrew Patterson, both at Monash University. Both have worked on BlueJ for many years, improving and extending the design and implementation while trying to write their PhDs at the same time. Without their work, BlueJ would never have reached the quality and popularity it has today, and this book might never have been written.

Another important contribution that made the creation of BlueJ and this book possible was very generous support from Sun Microsystems. Emil Sarpa, working for Sun in Palo Alto, CA, has believed in the BlueJ project from the very beginning. His support and amazingly unbureaucratic way of cooperation has helped us immensely along the way.

Everyone at Pearson Education worked really hard to fit the production of this book into a very tight schedule, and accommodated many of our idiosyncratic ways. Thanks to Kate Brewin for her determined support for this project, and to the rest of the team, including Bridget Allen, Kevin Ancient, Tina Cadle-Bowman, Tim Parker, Veronique Seguin and Fiona Sharples.

Our reviewers also worked very hard on the manuscript, often at busy times of the year for them, and we would like to express our appreciation to Michael Caspersen, Devdatt Dubhashi, Khalid Mughal and Richard Snow for their encouragement and constructive input.

David would like to add his personal thanks to both staff and students of the Computer Science Department at the University of Kent. The students who have taken the CO309 course have always been a privilege to teach. They also provide the essential stimulus and motivation that makes teaching so much fun. My colleagues have provided superb teach-

ing support this last year: Dick, Gift, John, Mike, Paul, Peter, Ralph, Rogerio, and Tom. Ian Utting deserves particular thanks for covering my teaching during my sabbatical, when I started work on this book. My appreciation, also, to Tim and Maggie, without whose presence and laughter the department would be the poorer. Outside University life, various people have supplied a wonderful recreational outlet to prevent writing from taking over completely; thanks to my climbing friends Paul Farrant, Judy Dunkley, and (in particular) to Chris Phillips whose friendship I appreciate and whose dogged determination on a route is a wonder to behold – rockover, Chris! Thanks also to Joe Rotchell, whose style and finesse speaking French on our Belgian cycling holiday was an inspiration!

Finally, I would like to thank my wife Helen, whose love is so special; and my children, whose lives are so precious.

Michael would like to thank Andrew and Bruce for many hours of intense discussion. Apart from the technical work that got done as a result of these, I enjoyed them immensely. I like a good argument. John Rosenberg, who has been a mentor to me for many years since the start of my academic career. Without his hospitality and support I would have never made it to Australia, and without him as a PhD supervisor and colleague I would never have achieved as much as I did in my work. It is a pleasure working with him, and I owe him a lot. Thanks to Michael Caspersen, who is not only a good friend, but has influenced my way of thinking about teaching during various workshops we have given together. My colleagues in the software engineering group at the Mærsk Institute in Denmark: Bent Bruun Kristensen, Palle Nowack, Bo Nørregaard Jørgensen, Kasper Hallenborg Pedersen, and Daniel May. They have patiently put up with my missing every deadline for every delivery possible while I was writing this book, and introduced me to life in Denmark at the same time.

Finally, I would like to thank my wife Leah and my two little girls, Sophie and Feena. Many times they had to put up with my long working hours at all times of day while I was writing for this book. Their love gives me the strength to continue and makes it all worthwhile.