

TURING

图灵原版计算机科学系列

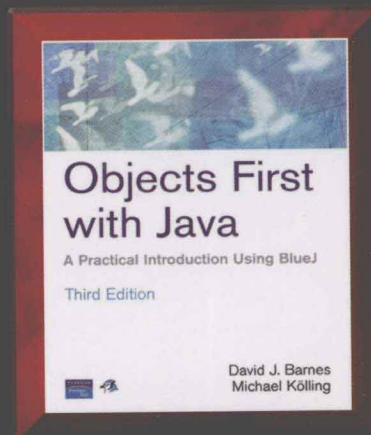
Java之父
James Gosling
作序推荐

Objects First with Java
A Practical Introduction Using BlueJ
Third Edition

Java面向对象程序设计

(英文版·第3版)

[英] David J. Barnes 著
Michael Kölling



人民邮电出版社
POSTS & TELECOM PRESS

TURING

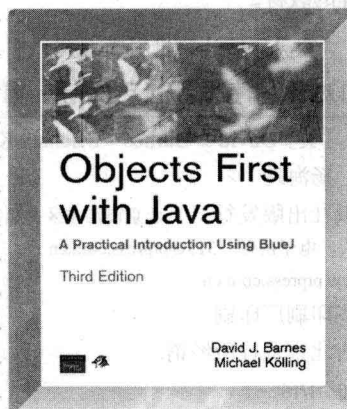
图灵原版计算机科学系列

Objects First with Java
A Practical Introduction Using BlueJ
Third Edition

Java面向对象程序设计

(英文版·第3版)

[英] David J. Barnes 著
Michael Kölling



人民邮电出版社
北京

图书在版编目 (CIP) 数据

Java面向对象程序设计: 第3版/ (英) 巴恩斯 (Barnes, D. J.); (英) 科灵 (Kölling, M.) 著. —北京: 人民邮电出版社, 2008.4

(图灵原版计算机科学系列)

ISBN 978-7-115-17515-1

I. J… II. ①巴… ②科… III. JAVA语言—程序设计—英文 IV. TP312

中国版本图书馆CIP数据核字 (2008) 第006045号

© Pearson Education Limited 2003, 2005, 2006.

This translation of *OBJECTS FIRST WITH JAVA: A PRACTICAL INTRODUCTION USING BLUEJ, Third Edition* is published by arrangement with Pearson Education Limited.

内 容 提 要

本节主要从软件工程的角度介绍面向对象和程序设计的基本概念, 侧重于讲解面向对象程序设计原理, 而不是Java语言细节。书中从面向对象的基础知识讲起, 介绍了对象和类; 然后深入到应用结构, 讲解了继承、抽象技术、构建图形用户界面、错误处理; 最后给出了一个完整的案例。书中使用两个工具实际运用所介绍的概念: Java编程语言以及Java编程环境BlueJ。全书按照项目驱动的方式来展开, 讨论了大量的程序项目, 并且提供了很多练习。

本书内容由浅入深, 适合初学者快速入门, 也适合高级程序员和专业人士学习参考, 可作为计算机相关专业“面向对象程序设计”课程的教材。

图灵原版计算机科学系列

Java 面向对象程序设计 (英文版·第3版)

- ◆ 著 [英] David J. Barnes Michael Kölling
责任编辑 杨海玲
- ◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街14号
邮编 100061 电子函件 315@ptpress.com.cn
网址 <http://www.ptpress.com.cn>
北京顺义振华印刷厂印刷
新华书店总店北京发行所经销
- ◆ 开本: 800×1000 1/16
印张: 32
字数: 666千字 2008年4月第1版
印数: 1-3000册 2008年4月北京第1次印刷
著作权合同登记号 图字: 01-2007-3071号
ISBN 978-7-115-17515-1/TP

定价: 59.00元

读者服务热线: (010) 88593802 印装质量热线: (010) 67129223

反盗版热线: (010) 67171154

Objects First with Java

**A Companion Website accompanies *Objects First with Java*,
Third Edition by David Barnes and Michael Kölling**

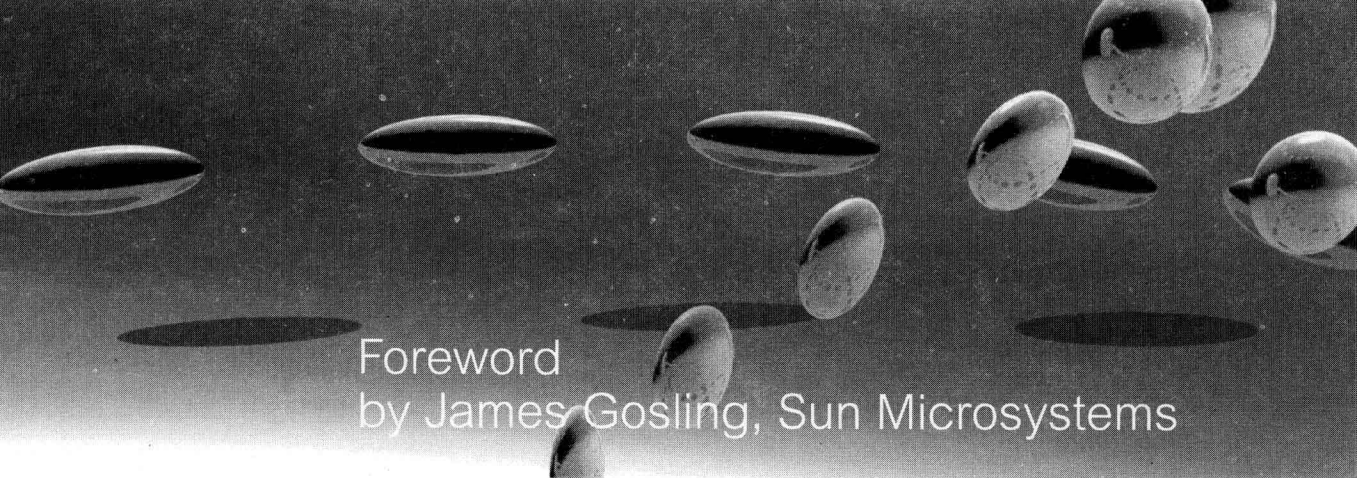
Visit the *Objects First with Java* Companion Website at
www.pearsoned.co.uk/barnes to find valuable learning materials
including:

For students:

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

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

To my family
Leah, Sophie and Feena
mk



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 is being developed and maintained at the University of Southern Denmark, Deakin University, Australia, and 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, 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 at 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 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 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 in the section, "List of projects discussed in detail in this book".

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 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. 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 introduces two new examples: an image viewer and a sound player. Both examples serve to discuss how to build graphical user interfaces (GUIs).

Chapter 12 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 an error-prone case study.

Chapter 13 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 14 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.

Third Edition

This is the third edition of this book. Several things have been changed from previous editions. The second edition saw the introduction of JUnit and a chapter on GUI programming. In this edition, the most obvious change is the use of Java 5 as the implementation language. Java 5 introduced new language constructs, such as generic classes and enumeration types, and almost all of our code examples have been changed to make use of these new features. The discussions in the text have, of course, also been rewritten to take account of this. Overall, however, the concept and style of this book remain unchanged.

Feedback we received from readers of prior editions was overwhelmingly positive, and many people have helped in making this book better by sending in comments and suggestions, finding errors and telling us about them, contributing material to the book's web site, contributing to the discussions on the mailing list, or translating the book into foreign languages.

Overall, however, the book seems to be 'working.' So this third edition is an attempt at improvements in the same style, rather than a radical change.

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.

Discussion groups

The authors maintain two email discussion groups for the purpose of facilitating exchange of ideas and mutual support for and by readers of this book and other BlueJ users.

The first list, **bluej-discuss**, is public (anyone can subscribe) and has a public archive. To join, or to read the archives, go to

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

The second list, **objects-first**, is a closed list for teachers only. It can be used to discuss solutions, teaching tips, exams, and other teaching-related issues. For instructions to join, please look at the book's web site (see above).

A decorative header featuring several dark, reflective spheres of varying sizes floating against a light, textured background. The spheres are arranged in a way that suggests depth and movement, with some appearing closer and larger, and others further away and smaller. The word "Acknowledgements" is centered in the middle of the page, below the decorative header.

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 now a Deputy Vice-Chancellor at Deakin 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. We have appreciated his friendship and collaboration immensely.

Several other people have helped to make BlueJ what it is: Bruce Quig, Davin McCall, and Andrew Patterson in Australia, and Ian Utting and Poul Henriksen in England. All have worked on BlueJ for many years, improving and extending the design and implementation in addition to their other work commitments. Without their work, BlueJ would never have achieved 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 through the first editions, and to Simon Plumtree who brought this edition to the light of day. Thanks also to the rest of the team, which includes Bridget Allen, Kevin Ancient, Tina Cadle-Bowman, Tim Parker, Veronique Seguin, Fiona Sharples, and Owen Knight. We are bound to have forgotten someone, and we apologize if we have.

The Pearson sales team also have done a terrific job in making this book visible, managing to avert every author's worst fear – that his book might go unnoticed.

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.

Axel Schmoltitzky, who produced the excellent German translation of this book, must have been our most careful and scrupulous reader; he suggested a good number of possible improvements on sometimes very subtle points.

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 introductory OO course have always been a privilege to teach. They also provide the essential stimulus and motivation that makes teaching so much fun. Without the valuable assistance of colleagues and postgraduate supervisors, running classes would be impossible, and Simon Thompson provides outstanding support in his role as Head of Department. Outside university life, various people have supplied a wonderful recreational and social outlet to prevent writing from taking over completely: thanks to my climbing friends, Chris Phillips and Martin Stevens, who help keep me up in the air and Joe Rotchell, who helps keep my feet on the ground.

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 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 – 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.

Guided Tour

Exercises

With hundreds of these provided throughout the text, at the end of every section, students are able to practice with the concepts until they understand them.

Concept reviews

These boxes summarize along the way the concepts students have just read about – an ideal revision tool!

Integration of BlueJ throughout

Exercises and activities throughout the text make students use the BlueJ software to practice and visualize the concepts being learnt.

Code examples

Example code from carefully constructed projects is included and discussed in the text, with full versions of all projects on the accompanying CD.

316 Chapter 11 ■ Building graphical user interfaces

To get a GUI on screen, the first thing we have to do is to create and display a frame. Code 11.1 shows a complete class (already named `ImageViewer` in preparation for things to come) that shows a frame on screen. This class is available in the book projects as `imageviewer01` (the number stands for version 0.1).

Exercise 11.1 Open the `imageviewer01` project. (This will become the basis of your own image viewer.) Create an instance of class `ImageViewer`. Resize the resulting frame (make it larger). What do you observe about the placement of the text in the frame?

We shall now discuss the `ImageViewer` class shown in Code 11.1 in some detail. The first three lines in this class are import statements of all classes in the packages `java.awt`, `java.awt.event`, and `javax.swing`.¹ We need many of the classes in these packages for all Swing applications we create, so we shall always import these three packages completely in all our GUI programs.

Looking at the rest of the class shows very quickly that all the interesting stuff is in the `mainFrame` method. This method takes care of constructing the GUI. The class's constructor contains only a call to this method. We have done this so that all the GUI construction code is in a well-defined place and is easy to find later (cohesion). We shall do this in all our GUI examples.

The class has one instance variable of type `JFrame`. This is used to hold the frame that the image viewer wants to show on screen.

Let us now take a closer look at the `mainFrame` method.

The first line in this method is

```
Frame = new JFrame("ImageViewer");
```

This statement creates a new frame and stores it in our instance variable for later use.

As a general principle you should, in parallel with studying the examples in this book, look at the class documentation for all classes we encounter. This counts for all the classes we use – we shall not point this out every time from now on, but just expect you to do it.

Exercise 11.2 Find the documentation for class `JFrame`. What is the purpose of the parameter "ImageViewer" that we used in the constructor call above?

A frame consists of three parts: the *title bar*, an optional *menu bar*, and the *content pane* (Figure 11.3). The exact appearance of the title bar depends on the underlying operating system. It usually contains the window title and a few window controls.

The menu bar and the content pane are under the control of the application. To both, we can add some components to create a GUI. We shall concentrate on the content pane first.

¹ The `swing` package really is in a package called `javax` (ending with an "x"), not `java`. The reason for this is largely historical – there does not seem to be a logical explanation for it.

Concepts

Comments are placed in a frame by adding them to the frame's menu bar or content pane.

1.5 Data types 7

near the top. This is called the *signature* of the method. The signature provides some information about the method in question. The part between the parentheses (`int distance`) is the information about the required parameter. For each parameter, it defines a type and a name. The signature above states that the method requires one parameter of type `int` named `distance`. The name gives a hint about the meaning of the data expected.

1.5 Data types

A type specifies what kind of data can be passed in a parameter. The type `int` signifies whole numbers (also called "integer" numbers, hence the abbreviation "int").

In the example above, the signature of the `moveHorizontal` method states that, before the method can execute, we need to supply a whole number specifying the distance to move. The data entry field shown in Figure 1.4 thus lets you enter that number.

In the examples so far, the only data type we have seen has been `int`. The parameters of the move methods and the `changeSize` method are all of that type.

Closer inspection of the object's popup menu shows that the method entries in the menu include the parameter types. If a method has no parameter, the method name is followed by an empty set of parentheses. If it has a parameter, the type of that parameter is displayed. In the list of methods for a circle you will see one method with a different parameter type: the `changeColor` method has a parameter of type `String` (up).

The `String` type indicates that a section of text (for example a word or a sentence) is expected. Strings are always enclosed within double quotes. For example, to enter the word `red` as a `String` type

```
"red"
```

The method call dialog also includes a section of text called a *comment* above the method signature. Comments are included to provide information to the (human) reader and are described in Chapter 2. The comment of the `changeColor` method describes what color names the system knows about.

Exercise 1.4 Increase the `changeColor` method on one of your circle objects and enter the string "red". This should change the color of the circle. Try other colors.

Exercise 1.5 This is a very simple example, and not many colors are supported. See what happens when you specify a color that is not known.

Exercise 1.6 Increase the `changeColor` method, and write the code into the parameter field without the quotes. What happens?

Pitfall A common error for beginners is to forget the double quotes when typing in a data value of type `String`. If you type green instead of "green" you will get an error message saying something like "Error: cannot resolve symbol".

Concepts

The header of a method is called its signature. The signature provides information about the method in question. The part between the parentheses (`int distance`) is the information about the required parameter. For each parameter, it defines a type and a name. The signature above states that the method requires one parameter of type `int` named `distance`. The name gives a hint about the meaning of the data expected.

2.6 Accessor methods 29

Code 2.6 The `getPrice` method

```
public class TicketMachine {
    // ...
    // Constructor method
    // ...
    /**
     * Return the price of a ticket.
     */
    public int getPrice() {
        return price;
    }
    // ...
    // Remaining methods omitted
}
```

Concepts

Methods have two parts: a header and a body. Here is the method header for `getPrice`:

```
/**
 * Return the price of a ticket.
 */
public int getPrice()
```

The first three lines are a comment describing what the method does. The fourth line is also known as the *method signature*.¹ It is important to distinguish between method signatures and field declarations, because they can look quite similar. We can tell that `getPrice` is a method and not a field because it is followed by a pair of parentheses: "()" and "}". Note, too, that there is no semicolon at the end of the signature.

The method body is the remainder of the method after the header. It is always enclosed by a matching pair of curly brackets: "{" and "}". Method bodies contain the *declarations* and *statements* that define what happens inside an object when that method is called. In our example above, the method body contains a single statement, but we shall use examples very soon where the method body consists of many lines of both declarations and statements.

Any set of declarations and statements between a pair of matching curly brackets is known as a *block*. So the body of the `TicketMachine` class and the bodies of all the methods within the class are blocks.

There are at least two significant differences between the signatures of the `TicketMachine` constructor and the `getPrice` method:

```
public TicketMachine(int ticketCost)
public int getPrice()
```

¹ This definition differs slightly from the more formal definition in the Java language specification where the signature does not include the access modifier and return type.

Figure 2.4 Parameter passing (A) and assignment (B)

A) A method call creates a new object in the memory. The object has the same price as the parameter. The object is then passed to the method. The method can then use the object's price.

B) A method call creates a new object in the memory. The object has the same price as the parameter. The object is then passed to the method. The method can then use the object's price.

Concepts

The lifetime of a variable describes how long the variable exists in memory before it is destroyed.

2.5 Assignment 27

Figure 2.4 Parameter passing (A) and assignment (B)

A) A method call creates a new object in the memory. The object has the same price as the parameter. The object is then passed to the method. The method can then use the object's price.

B) A method call creates a new object in the memory. The object has the same price as the parameter. The object is then passed to the method. The method can then use the object's price.

Concepts

A concept related to variable scope is *variable lifetime*. The lifetime of a parameter is limited to a single call of a constructor or method. Once that call has completed its task, the formal parameters disappear and the values they held are lost. In other words, when the constructor has finished executing, the whole constructor scope (see Figure 2.4) is removed, along with the parameter variables held within it.

In contrast, the lifetime of a field is the same as the lifetime of the object to which it belongs. It follows that if we want to remember the cost of tickets held in the `TicketCost` parameter, we must store the value somewhere more persistent – that is, in the price field.

Exercise 2.16 To which class does the following constructor belong?

```
public Student(String name)
```

Exercise 2.17 How many parameters does the following constructor have and what are their types?

```
public Book(String title, double price)
```

Exercise 2.18 Can you guess what types some of the `Book` class's fields might be? Can you assume anything about the names of its fields?

2.5 Assignment

In the previous section, we noted the need to store the short-lived value of a parameter into somewhere more permanent – a field. In order to do this, the body of the constructor contains the following assignment statement:

```
price = ticketCost;
```