PREFACE

Purpose and approach

This book is an introductory text on programming and software design using Java™. We assume no previous programming experience, no more mathematics than elementary algebra, and a reading level appropriate for first-year undergraduates. Though intended for a first-year undergraduate course in computer science, the text has also been used successfully in higher level courses in programming and software engineering.

The approach is object-oriented, sometimes called “objects first.” While traditional introductory programming texts approach the subject in a syntax and example driven format, we stress design and the discipline needed for developing complex software systems. The emphasis throughout the book is on problem modeling using sound software engineering principles and concepts. It takes considerable experience, of course, to acquire real proficiency in the design and construction of software systems. We hope to develop a set of fundamental skills and to introduce a point of view regarding system design that will be as useful in constructing large systems as it is in building small components.

The programming language used is Sun Microsystem’s Java, SDK version 5.0 or later. We should be clear that this is not just a text about Java. We are more concerned with teaching the design and construction of software systems than developing fluency in a particular programming language. Nevertheless as we assume our readers have no previous programming experience, we spend quite a bit of time covering Java syntax and semantics. In fact, we cover all but a few features of the language. We chose Java because it offers a good compromise between viability and semantic coherence. As the language is quite popular, adequate tools and libraries are readily available. At the core, Java is a relatively clean and small language: one that supports the concepts we present without “getting in the way.” We do not need to spend much time explaining language peculiarities that have little to do with the fundamentals of software design. We informally adopt a notation called the Unified Modeling Language (UML) for graphically denoting objects, object relationships, and system dynamics.

For introductory students, we assume topics will be further explained in lectures, discussions, or tutorials. Ideally, concepts should be reinforced with regular laboratories, where they can be reviewed and practiced.

Finally, the text includes optional interactive exercises to be done using the DrJava integrated development environment. DrJava was designed with beginning students in
mind and includes an easy to use facility for interactively evaluating Java code. DrJava is under active development by the JavaPLT group at Rice University and is freely available under an open source license from

http://drjava.sourceforge.net/

While these exercises are not intended as a substitute for a structured laboratory component, they will help students understand, through direct experience, the implications of the concepts discussed in the text. They should be particularly valuable to the reader who is not using the text in a class.

Though we find DrJava particularly suited to introductory students, the central topics and their development in the text is completely independent of any software development environment. The student can use any environment for exercises and practice.

---

**Overview**

Chapter 0 presents an introduction to object-oriented software development. The two fundamental problems we face in building software systems—that they are complex and dynamic—are introduced, along with the concepts of composition and abstraction. Virtually everything we do in the remainder of the text centers on using abstraction and composition to address the complexity and evolutionary nature of software.

The actual study of software design and the programming language Java begins in Chapter 1. A reader who has little or no experience with computers may want to read the supplements on Systems and software and Programming errors before beginning this chapter. Chapter 1 introduces the fundamental concepts of object and class, and relates them to the notions of value and type. Examples of a class definition and of a complete program to test the class offer a “preview” of Java syntax.

Chapters 2 through 8 progress from specifying, implementing, and testing simple, well-defined classes, through designing and building small collections of interacting classes. Very simple classes are defined in Chapters 2 and 3. Elementary Java expressions and statements, including arithmetic expressions, assignments, and method invocations, are explained. Chapter 3 includes the development of a complete system for testing a class. Boolean expressions and conditional statements are added in Chapter 4, while the important notion of component correctness is introduced in Chapter 5. Chapter 6 deals with testing, with particular emphasis given to “test driven” implementation. A DrJava exercise in Chapter 6 introduces the JUnit testing framework. A text-based user interface is developed in Chapter 7, and this first part of the text concludes with the design and implementation of a complete system of moderate complexity in Chapter 8.

Much of this early material of necessity deals with elementary algorithmic concepts, similar to those found in a traditional procedural approach. However the context in which topics are presented is one of class development, an approach sometimes called object-based. Classes are specified and designed with the assumption that they are to be incorporated into a larger whole. Fundamental notions such as the distinction between specification and implementation, and the architectural distinction between model and user
interface are emphasized. The basic structuring mechanism of composition is also introduced in this part.

The next three chapters deal with abstraction, and serve as an introduction to object-oriented programming. Interfaces and subtypes are introduced in Chapter 9, while class extension and polymorphism are treated in Chapter 10. The discussion of inheritance and polymorphism necessitates a presentation of some of the more baroque aspects of Java’s scoping and accessibility semantics. While it is tempting to skip much of the detail, enough must be presented to enable students to understand otherwise incomprehensible program behavior. Chapter 11 introduces abstract classes, and compares the fundamental structuring techniques of composition and extension.

Chapter 12 presents the sequential list as the paradigmatic container class. (“List” is simply a more fundamental and abstract notion than “array.” Arrays are hardware architectural features—see the IBM 704—bubbling up into the language syntax.) Iteration is discussed extensively in Chapter 12.

The remaining chapters are much less sequentially dependent on each other. Chapter 13 introduces arrays. Basic sorting and searching algorithms are presented in Chapter 14. Chapter 15 expands on the programming by contract methodology, addressing the complex problem of error handling. Java’s exception mechanism is introduced here. Chapter 16 discusses stream i/o, and surveys the fundamental java.io classes.

Chapters 17 and 18 provide an overview of the Java Swing library and the model-view-controller architecture. The discussions of MVC and Swing allow us to examine specific examples of additional fundamental design patterns such as composite and observer.

The algorithmic topic of recursion is the topic of Chapter 19. Backtracking and object recursion are also covered here. Chapter 20 returns to consider the generic structures introduced in Chapter 12 in more detail. Finally, Chapter 21 investigates linked list implementations, while Chapter 22 presents iterators as an efficient means of accessing list elements.

The following are key points of the presentation.

• Methods are categorized as commands that change state or queries that return some aspect of an object’s state.
• Class specification is clearly distinguished from class implementation.
• System and class design are distinguished from algorithm design.
• Design by contract, with preconditions and postconditions, is employed throughout.
• Assertions, in particular invariants, are used to design, document, and test classes.
• The fundamental role of testing in the design-specify-implement cycle is emphasized.
• An architecture that distinguishes the model from the user interface is employed regardless of system size.
• The class List is used to provide a basic container abstraction, and several implementations are developed including array-based and linked implementations.
• The notion of a software pattern is introduced and used whenever a tried design solution is available. Several common patterns are presented.
Preface

Optional material, DrJava

Some sections in the text are shaded gray and are marked with a blue asterisk (✽). These sections are optional, and can be skipped without affecting the general flow of the presentation. Subsequent material does not depend on these sections. Some optional sections contain less commonly used material included “for completeness.” Others include topics that might require a bit more background than can be generally assumed of introductory students. (Exercises that depend on optional material are also marked with a blue asterisk.)

There are also DrJava exercises throughout the text marked with ✿ and shaded blue. Though we encourage readers to do these exercises, subsequent topics are not dependent on them and they can be skipped without loss of continuity.

Supporting material

Supporting material can be accessed through the Wiley server at

http://www.wiley.com/college/nino

or through our server at

http://www.cs.uno.edu/~fred/nhText

For students

The following material is available online: libraries used in the text, including source and documentation; supporting code for DrJava exercises; source code for examples used in the text.

For instructors

Slides to accompany the text are available through the Wiley site listed above. Solutions to the exercises are available to adopting instructors. To obtain the exercise solutions, instructors should contact their local Wiley sales representative. Finally, laboratory material used in our courses is generally available through the UNO site listed above. Instructors should feel free to contact the authors if necessary.

Tools and documentation

Software tools and documentation can be found at these sites.

• The Java Software Development Kit
  http://java.sun.com/javase/downloads/

• The DrJava development environment
  http://drjava.sourceforge.net/

• The JUnit testing framework
  http://www.junit.org
To the student

This book is an introductory text on programming and software design using Java. As we said, we assume no previous programming experience, no more mathematics than elementary algebra, and a reading level appropriate for first-year undergraduates. The only additional requirement is dedication and willingness to practice. To develop programming skill, you must program, and program often.

Our goal is to teach the basic discipline needed to create substantial programming applications that can be understood, maintained, and modified. To that end, the text emphasizes program design and the development process. For instance, you will produce not only working code, but a test-bed that can be used to verify the correctness of the code and serve as a benchmark for future extensions and modification. From the start you will be concerned with program design issues and the iterative development process. Mastery of these topics will pay major dividends as you learn to develop larger and more complex programs.

You may very well already have some programming experience, acquired on your own, in other courses, or on the job. While this experience should help you quickly learn the language syntax, we suggest that while reading this book, you try to put aside whatever
ideas you have previously acquired about software development. Over the years, many programming strategies have been developed and written about. Some are useful, others have proven to be of limited value. The approach we take here has been used successfully in the software industry for well over a decade. It can legitimately be called “best current practice.” As you read and practice with this text, follow the strategies as they are presented. Once you have a good foundation in fundamentals, you can easily adapt your approach and practice to suit your needs and preferences.

To the instructor

As mentioned above, the text is intended for beginning undergraduates in computer science, software engineering, or computer engineering. No previous programming experience is assumed, and the only mathematics required is elementary college algebra. We do, however, assume a university level reading ability and a certain seriousness of purpose. Our presentation is straightforward: there is no attempt to dazzle the reader or “trick” her into learning.

Procedural approach or objects first?

We believe there is considerable benefit in beginning with the same fundamental methodology we expect students to be using when they complete the curriculum. This creates far fewer problems than starting with a procedural approach and asking students to “shift paradigms” (apologies, Thomas Kuhn) after they’ve experienced some initial success. With the procedural approach we’ve used in the past, students started by developing complete, self-contained solutions to simple, well-defined problems. This had the advantage of introducing fundamental algorithmic constructs and providing a grounding in the specifics of some programming language in a nicely confined context. The difficulty was that students found that these problems yield to an undisciplined, ad hoc approach, in which all constructs have essentially global scope. When later confronted with more substantial, less well-defined problems—problems for which partitioning is essential and for which design issues involving complex structural relations between components must be effectively addressed—they floundered, unwilling to abandon the “code now, think later” approach that had served them so well. We will go so far as to say that having students start by writing small, complete, self-contained programs is detrimental to their later development as competent programmers.\(^1\),\(^2\)

We must, of course, begin by teaching fundamental algorithmic constructs in small well-defined components, even though the payoff will be in understanding how to con-

2. Another point of view is that this is complete hokum; the only useful programs are written by very clever people, and there’s not much we can do to ruin them.
struct large, evolving systems. But the software elements we ask students to construct are carefully specified components of a conceptually large system. (We note that instruction in many other disciplines, from architecture to automotive technology, proceeds in exactly this way.)

It has been claimed in some quarters that the objects first approach has failed, and a number of instructors have returned to the procedural approach—though using Java. But we should recall the “lame and impotent conclusion” of the procedural approach that led to the development and adoption of objects first. We suggest that perceived difficulties result more from the manner of presentation than from the method. An objects first approach requires extensive refashioning of instructional methodology and a complete overhaul of the instructional tool kit. Lectures, laboratories, examples, exercises must be carefully designed to explicate the methodology. Many departments have had success with the object oriented approach, and evidence of the value of the methodology in the software industry is overwhelming.

Our experience

We have taught the material for more than eight years as part of an introductory sequence in programming and software design. Our students are first and second year computer science majors with no previous computer science coursework. We are not a selective university, and our students can best be described as “average.” Our experience has been that students do not find fundamental object-oriented concepts difficult, and have no more trouble than with a traditional procedural approach to programming. On the other hand, they develop a higher degree of facility with abstract design, and generally produce better structured programs in upper-level courses.

We enjoy two advantages, however, that are not available to every instructor. First, our introductory software design sequence—essentially CS1 and CS2—has always been composed of three semester-long courses and has included a substantial “software engineering” component. Thus there were few logistical difficulties in incorporating an objects first approach into the curriculum, and we are able to proceed at a somewhat more leisurely pace than if we had, say, only a quarter or two to work with. Second, since the sequence is intended for computer science majors, we can define performance goals in terms of the complete sequence. There is no pressure for students to acquire a complete, self-contained set of “useful” skills after a single semester.

The role of the text

We should make clear our view that the text is only one part of an introductory student’s learning environment. Many of the ideas presented are not easy to grasp on a casual first reading. (Though certainly not so difficult as concepts found in freshman calculus or mechanics!) Regular lectures, discussions, or tutorials in which topics are elaborated and examples developed are essential.

Furthermore, it is fundamental that students, particularly at the introductory level, be “immersed in code.” Regular homework assignments and laboratory work are indispensable to a student’s progress. And it is as important for the novice programmer to read well-
organized code as to write his or her own. Thus some exercises should involve reading and understanding, and then modifying or extending, existing code. To this end, a substantial number of exercises are included in the text, and source and supporting code for examples and exercises used in the text are available on line. But here, clearly, the role of the text is ancillary. Much of the work and all of the feedback, alas, falls to the instructor.

**Laboratories and DrJava**

We are of the opinion that learning to program at the introductory level is best supported by regular participation in structured laboratories. Detailed explanation of syntactic issues, for instance, can take place in the laboratories, leaving the lecturer free to concentrate on fundamental concepts, design issues, and case studies.

We employ the DrJava software development environment in our labs. There are several advantages to this tool. First, it is relatively intuitive and easy to learn. Second, it incorporates an “interactions pane” in which arbitrary Java expressions can be evaluated. This greatly simplifies class development and initial unit testing. A student will write a method and then immediately test and experiment with it without having to construct a test driver. Finally, the JUnit testing framework is incorporated into the DrJava environment. From the beginning we insist that a unit “test plan,” expressed as a set of JUnit tests, is an essential part of the “programming product” developed in an exercise.

To give some of the flavor of a laboratory, we have incorporated a number of DrJava exercises in the text. These exercises should be of particular value to the reader who is not participating in a structured, scheduled laboratory. However, as many instructors prefer other development environments, we have included these exercises in an inessential way. They can be ignored completely without affecting the thread of topic development. The price we pay is that the DrJava material seems “pasted on” rather than integrated into the text.

**Topic organization**

Chapters 0 through 8 comprise the basic topics covered in the text. Students with little familiarity with computers should probably start by reading the supplements *Systems and software* and *Programming errors*. Students with some programming background can skim much of elementary details in these chapters.

These early chapters present fundamental algorithmic constructs from an object based perspective. While many students will be familiar with the elementary algorithmics, the object based slant will be new. In our experience, students find the approach natural and have far fewer problems with the object-oriented concepts than one might expect. A substantial system involving a number of interacting objects, with a separately identifiable model and user interface, is put together in Chapter 8.

Chapters 9, 10, and 11 cover abstraction and serve as an introduction to object-oriented programming. These chapters are a bit more conceptually advanced than the previous ones, and are the first in which more general design issues are addressed in a substantial way.
Chapter 12 introduces the “list” as the basic manifestation of “container.” The notion of a list is easy to understand, and is more fundamental and general than the notion of an array. We do the same algorithmic development with indexed lists as is conventionally done with arrays.

Chapter 13 introduces arrays. Though one of the examples in this chapter involves using arrays to implement lists and several involve generic methods, it would not be difficult to present Chapter 13 immediately after Chapter 8.

The remaining chapters are largely independent of each other.

Chapter 14 presents the traditional algorithmic topics of searching and sorting.

Chapter 15, *Exceptions*, and Chapter 16, *Stream I/O*, are mostly self-contained. Chapter 15 can be presented after Chapter 10, for instance. Chapter 16 is included mostly for completeness, and can be treated as a reference.

Chapters 17 and 18 deal with event-driven graphical user interfaces. A large part of Chapter 17 consists of “nuts and bolts” regarding Swing. Much of this material shows how to handle specific Swing components, and can be omitted or treated as a reference. Chapter 18 discusses the architecture of an event-driven system.

Chapter 19, on recursion, is again algorithmic in content. It can be presented after Chapter 14, which deals with iterative sorting and searching algorithms.

Chapter 20 takes a more in-depth look at generic structures, first introduced in Chapter 12. Wildcard types are introduced here. Much of the material could be presented immediately after, or along with, Chapter 12.

Chapters 21 and 22 again deal with lists. Chapter 21 covers linked implementations and Chapter 22 iterators.

The basic dependencies between chapters is shown in the graph below. A few minor, nonconceptual dependencies are not illustrated in the graph. To wit,

- Chapters 17 and 18 each include an example in which a try statement is used to catch a *NumberFormatException*; Chapter 18 also uses a throws clause in an example, and Chapter 20 refers to an exception in an exercise. Exceptions are introduced in Chapter 15.
- Chapters 17 and 18 use “anonymous classes” to implement *WindowListeners*. Anonymous classes are introduced in Chapter 14.
- Chapter 19 uses the observer pattern in the tower puzzle example. Observer is introduced in Chapter 18.
Sections marked with a blue asterisk (※) and shaded gray are optional. They can be omitted without affecting the flow of the presentation. In some cases, they are included “for completeness” and can be treated as references. In other cases, they contain topics that might require a bit more background or mathematical maturity than can be generally assumed of introductory students.

While the book is designed to be read sequentially, material in the later chapters can be presented in a number of ways (or ignored completely) in a course. We suggest progressing sequentially through chapters 12 or 14, followed by chapter 19, for a traditional CS1 course.

**Coding conventions**

We have, for the most part, adopted the conventions of The Java Language Specification [Gosling 05]. However, we have generally avoided names `getA` and `setA` to retrieve and modify the value of an attribute A. Rather, we use the older convention of using nouns or adjectival predicates to name value-returning methods (size, isEmpty), and imperative verb phases to name state-modifying methods (resetCount). The reason is that we want to discourage the view of an object as a “data record” that simply holds a collection of related values. We try to emphasize the point of view that an object’s data is there *solely to support its functionality*. If an object is simply making data available to clients via gets and sets, there is likely something seriously wrong with the coherence of the object.

We have also eschewed HTML tags and character names in doc comments. Specifically, in doc comments Java identifiers and key words and Java expressions should be wrapped in `<code>`...`</code>` or `<tt>`...`</tt>` tags, and the characters ‘<’, ‘>’, and ‘&’ should be written ‘&lt;’, ‘&gt;’ and ‘&amp;’. However, as this severely hinders the readability of the source code, we have chosen to write doc comments as plain text.

**Changes in the third edition**

In addition to many small improvements and corrections in wording, examples, exercises, etc., there are two major changes in this edition of the text.

- Chapter 13 has been completely rewritten to provide a comprehensive coverage of arrays. The chapter includes presentations on constant arrays, multidimensional arrays, and array subtyping, and optional material on static initializers and variable arity parameters.

  Almost all of the material in this chapter (Sections 13.6 and 13.7 are the exceptions) can be presented immediately after Chapter 8. However, we suggest that the more fundamental abstraction is `List`, found in Chapter 12, for which an array provides an implementation choice.

- A new Chapter 20 has been added to examine generics in detail. Chapter 12 introduces generic types and methods, and generic structures are used subsequently throughout the text. This new chapter introduces multiple and bounded generic parameters, wildcard types, and problems relating to type erasure. Some of this material was included in a supplement in the previous edition. Here it is considerably expanded with numerous examples.
We could not, of course, have completed this text without the help and support of many people. We are indebted to our colleagues for the confidence they showed in us by allowing us to overhaul the introductory programming curriculum and install an unproven approach. We are also indebted to the first few classes of students who endured our trials and errors—getting a first hand view of the design-implement-test cycle. They seem no worse for the experience, and didn’t have much of a choice anyway.

We are grateful for the conscientious efforts of our reviewers, Glenn Blank, Jonathan Bredin, Ralph L. Conrad, Tim Gegg-Harrison, Lewis Hitchner, Daniel Jinguji, Man Kong, Kenneth E. Martin, David McPherson, Michael Ondrasek, and Raymond Wisman for this edition, and Noah D. Barnette, Elizabeth Boese, Stan Kwasny, Rose M. Lowe, Blayne E. Mayfield, David T. Small, Deborah Trytten, Patrick Votruba for previous editions. Particular thanks for insightful suggestions to Kevin O’Gorman and Richard E. Pattis, and again to Robert Burton for his careful reading of the first two editions of the text. The book is much better than it ought to be because of their many helpful suggestions.

Thanks to Martin Dickey, Rob Duisberg, Laurie Poulsen, and Jonathan Pool at the University of Washington for logging errors in preliminary drafts of the second edition and offering a number of helpful suggestions.

We’d like particularly to thank our colleague Bill Greene who has labored through several versions of the text and notes on which this text was based. His observations are always constructive.

We are grateful for the help and guidance of the folks at Wiley, particularly our editor, Daniel Sayre, editorial assistant Carolyn Weisman, Ken Santor, who put up with us through production; Jeri Warner, our copy editor; and Jeof Vita who did a splendid cover.

Thanks to Gilad Bracha at Sun for patiently answering our continuous questions regarding J2SE 5.0, and to Robert Cartwright and the DrJava team at Rice for the exceptional development tool.

Finally, we are grateful for the support of our family and friends, and for Bill Joy, James Joyce, Thomas Kuhn, George Eliot, John Reynolds, Ray Witham, Stephen Kleene, Larry Landweber, Dana Scott, John Adams, Jacques Barzun, Arnold Schoenberg, Jacqueline Du Pré, Immanuel Kant, Greg Homer, and Karl Hoffman.

I should without hesitation split the infinitive and write to calmly consider.  
—Jacques Barzun

They lard their lean book with the fat of others’ works.  
—Robert Burton (1576–1640)¹

—Samuel Beckett

¹. Borrowed from the preface of [Gosling 05].