Java Interfaces in CS 1 Textbooks

Thomas VanDrunen
Wheaton College, Wheaton, IL
Thomas.VanDrunen@wheaton.edu

Abstract

Java's `interface` construct allows for a clear distinction between subtype polymorphism based on a shared interface and code reuse based on class extension or inheritance. *Design Patterns* argues that in an object-oriented setting, programming should be done to an interface, not to an implementation, that class inheritance is a mechanism for code reuse rather than for subtyping and polymorphism, and that even composition should be favored over class inheritance. We conclude from this that `interface` should be introduced prior to and given more focus than abstract classes and class inheritance. We survey 27 textbooks from major publishers to show that very few available texts teach Java this way. We propose an alternative ordering of material that will promote the principles mentioned above.

Categories and Subject Descriptors K.3.2 [*Computers and Education*]: Computer and Information Science Education; D.1.5 [*Programming Techniques*]: Object-oriented Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features—Polymorphism

General Terms Program Design, Pedagogy, Programming Languages

Keywords Interfaces, Polymorphism, CS 1

1. Introduction

The Java programming language has two features supporting subtype polymorphism: class extension and `interface`[1] implementation. If a class extends another (possibly abstract) class, it inherits all the members of that class; its type is considered a subtype of the extended class's type; and it implicitly shares an interface with the extended class. If a class implements an `interface`, it must provide an implementation of all the methods of the `interface`, and its type is considered a subtype of the `interface`'s type. Java does not support multiple inheritance, so a class may not extend more than one class, but a class may implement more than one `interface`. Thus class extension can be used for subtype polymorphism and code reuse, but not multiple subtyping; `interface` implementation, on the other hand, can be used for subtype polymorphism, including multiple subtyping, but not code reuse for non-static members. (For simplicity, we omit discussing `interface`s extending other `interface`s.)

[1] Throughout this paper, we use `interface` in typewriter print to refer to the Java language construct, and interface in plain print to refer to the general concept, a set of prototypes of public methods.

Textbooks abound for introductory programming courses (CS 1). Research for this paper found 27 textbooks readily available from major publishers for Java alone (not including variant versions by the same authors), and several in their fifth or sixth edition. As one would expect, the ordering, relative prominence, and motivation for various language features, including class extension and `interface` implementation, vary among textbooks. Regardless of whether objects are introduced first, early, or late, polymorphism is a central concept and one of the most powerful features of object-oriented programming; naturally, the choice of which language features to present first and with most emphasis will affect how students understand and use polymorphism in program design, since students tend to remember best and use most what they learn first. Thus this consideration must play a part in the decision of instructors who are confronted with a wide array of choices when selecting a textbook. This paper seeks to provide some guidance on the matter.

Section 2 considers the difference between code sharing and interface sharing and relates subtyping and polymorphism to these. We conclude from this that subtype polymorphism should be based on a shared interface rather than reused implementation, and hence `interface` implementation rather than class extension should be the primary vehicle for polymorphism in well-designed Java programs. In Section 3, we survey a field of 27 textbooks, categorizing them by the relative order in which subclassing, abstract classes, and `interface`s are introduced and by the way `interface`s are motivated by the text. This will demonstrate that, with few exceptions, textbooks are not using `interface`s to teach polymorphism. Section 4 concludes by suggesting a practical way to implement an *interfaces-first* approach to teaching polymorphism, placing polymorphism and subtyping before inheritance and code reuse. Individual notes on each textbook surveyed can be found in an appendix.

2. Interfaces

One of the most influential works on program design in an object-oriented setting is *Design Patterns* [13]. More than the individual patterns cataloged, however, perhaps the most insightful contribution of the book is the discussion of interfaces and implementation-sharing found in the introduction, pages 11-28. There the authors defend the principle

> *Program to an interface, not an implementation.* (Page 18)

To abridge that argument (ellipses omitted for readability):

> The set of all signatures defined by an object's operations is called the **interface** to the object. A **type** is the name used to denote a particular interface. We say that a type is a **subtype** of another if its interface contains the interface of

OOPSLA'06 October 22–26, 2006, Portland, Oregon, USA.
ACM 1-59593-491-X/06/0010.

its **supertype**. Interfaces are fundamental in object-oriented systems. Objects are known only through their interfaces. An object's interface says nothing about its implementation. Two objects having completely different implementations can have identical interfaces.

Dynamic binding lets you substitute objects that have identical interfaces for each other at run-time. This substitutability is known as **polymorphism**.

An object's implementation is defined by its **class**. New classes can be defined in terms of existing classes using **class inheritance**. When a **subclass** inherits from a **parent class**, it includes the definitions of the all the data and operations that the parent class defines.

It's important to understand the difference between an object's *class* and its *type*. An object's class defines how the object is implemented. The class defines the object's internal state and the implementation of its operations. In contrast, an object's type only refers to its interface—the set of requests to which it can respond. An object can have many types, and objects of different classes can have the same type.

It's also important to understand the difference between class inheritance and interface inheritance (or subtyping). Class inheritance defines an object's implementation in terms of another object's implementation. In short, it's a mechanism for code and representation sharing. In contrast, interface inheritance (or subtyping) describes when an object can be used in place of another. (Pages 13-17)

Thus interface-sharing and class-inheritance are separate concepts. Subtype relationships used for polymorphism are best understood through interface-sharing. Class-inheritance is merely a code-reuse mechanism—and perhaps not the best mechanism at that, as the authors go on to argue for a second principle,

Favor object composition over class inheritance. (Page 20)

Gamma et al mention that it is "easy to confuse these two concepts, because many languages don't make the distinction explicit" [13], languages like C++, which was used for most of the examples in *Design Patterns*. Java, however, does make a distinction, with the language features of class-extension and `interface`-implementation. Thus, when teaching object-oriented programming in Java, `interfaces` should be given a place of prominence and be used as the primary means to demonstrate and implement subtyping and polymorphism. Subclassing should be deemphasized, discussed as one of several options for reusing code—others including composition, delegation, and aggregation.

3. Textbook survey

Categorizing textbooks is difficult and subtle, as many factors come into play: overall approach, kind and quality of pedagogy, clarity of prose, completeness, target audience, usefulness of examples, and quality of exercises, among others. None of these, however, are in view for our survey; instead, we focus on two questions: In what sequence does the textbook place `interfaces` in relation to subclassing and abstract classes? and, What is the principal motivation given for `interfaces` as a language construct? The 27 textbooks under review are characterized in relation to these two questions in the chart in Figure 1, where the textbook abbreviations are keyed to the appendix.

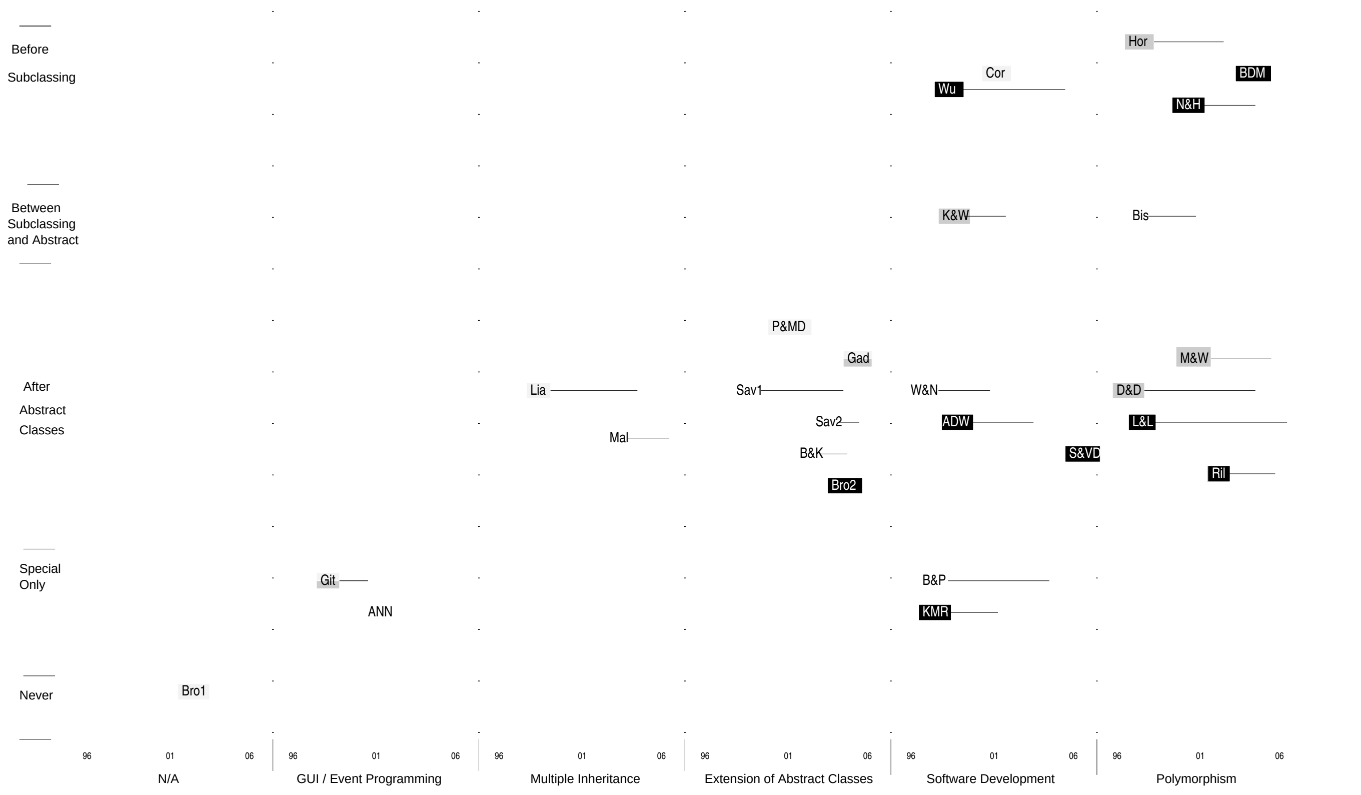
The sequencing of discussions of subclassing, abstract classes, and `interfaces` is plotted by the vertical axis. This categorization required judgment calls in a few cases, because some textbooks discuss `interfaces` in several contexts. Typically in those cases, in addition to the context of subclassing, inheritance, and polymorphism, `interfaces` are casually introduced earlier in the textbook in order to explain the Java Collections classes or GUI implementation with AWT or Swing. Occasionally `interfaces` are discussed as a means to enforce a class's conformity to a specification or for use with a design pattern, which uses we lumped together as "software development." Four textbooks (Git, ANN, B&P and KMR) present `interfaces` *only* in specialized topics, not in relation to subclassing and abstract classes. For the other textbooks, effort was made to discern where their *primary* discussion of `interfaces` comes. Since abstract classes normally are discussed after subclassing (though we propose an alternative in Section 4), the three main categories represent introducing interfaces before subclassing, between subclassing and abstract classes, and after abstract classes. Only five (Cor, Wu, Har, N&H, and BDM) present `interfaces` as an independent language construct before subclassing. Placing `interfaces` between subclassing and abstract classes is a non-obvious arrangement, and only two (K&W and Bis) fall there. The pack (15 of 27) place `interfaces` after abstract classes.

The motivation for `interfaces` is categorized along the horizontal axis. This categorization is also difficult because a language construct can be used for more than one purpose. Therefore effort again was made to discern the *primary* motivation for `interfaces`, especially as they are compared to subclassing and abstract classes. The news is better in this consideration, as a large number (eight) of texts recognize polymorphism as the main use of `interfaces`, though just as many base their discussion on software development techniques, as explained above. Among texts that discuss `interfaces` after abstract classes, the most popular motivation was an extension of the notion of an abstract class—that is, an `interface` is essentially a class that is purely abstract. For two textbooks (Lia through the fourth edition and Mal), `interfaces` allow for multiple inheritance. This terminology is a bit misleading, as some consider *inheritance* to mean class inheritance, for which Java does not allow multiple inheritance, as opposed to *interface* inheritance; it is consistent, however, with Gamma et al [13].

Two other variables are shown on the chart: years in print and self-identified approach. A textbook's horizontal placement within a motivation category is based on the publication year of its first edition, keyed to labels along the horizontal axis. For textbooks in multiple editions, a bar extends from the abbreviation to the year of the most recent edition. This shows that the categorization is not correlated to the time of publication (though it is interesting to note that no first editions came out in 1999). Both new and old books can be found in nearly every categorization.

Shading shows how various textbooks identify themselves with respect to frequently-advertised approaches to teaching introductory programming. Not all textbooks made such an identification (some, in fact, identify themselves as usable with several approaches), but others could be broadly labeled as *objects-first*, *objects-early*, or *fundamentals-first*, shaded dark, medium, and light, respectively. Two textbooks (Git and Gad) were released in parallel versions, objects-early and fundamentals-first, and thus have two shadings. Those that did not self-identify in terms close to these have no shading. Such a categorization has limited use since what one author considers objects-early may seem like objects-first to another, but it does demonstrate that a given sequencing or motivation is not restricted to a specific approach, as we find representatives of various approaches throughout our categories.

Five authors (Bronson, Gittleman, Gaddis, Savitch, and Malik) each have been involved with two textbooks. In the case of Savitch (Sav1 and Sav2) and Bronson (Bro1 and Bro2), these are completely separate works and so appear independently in the chart. For the others, either the two books are variant versions of each other—so with Gittleman (Git) and Gaddis (Gad)—or the later has



Points represent textbooks. The vertical axis shows where the principle discussion of `interfaces` occurs in the textbook. The horizontal axis shows the motivation given for `interfaces`. The line extending from a point shows the year for the first and most recent edition, as keyed on the horizontal axis. Shading shows self-identified approach: light for fundamentals-first, medium for objects-early, dark for objects-first.

Figure 1. Scatterplot showing when and why various textbooks discuss Java `interfaces`.

clear continuity with the former—in the case of Malik (Mal)—and are therefore represented only once in the chart.

4. Conclusions

Our chart shows that with the exception of three (Hor, N&H, and BDM), introductory programming textbooks are not heeding the recommendations from Gamma et al [13]. Subclassing is given first billing, confusing the notions of code reusing and subtyping. Interfaces are reserved until after abstract classes (though in some cases they are given “equal time”), and their purpose is often misplaced or unfocused. In extreme cases, ANN and Git discuss interfaces only when forced by the needs of GUI programming; Sav1 considers `interfaces` optional material through the third edition (this designation, though removed in the fourth edition, is preserved, apparently by error, in Displays 7.15 and 7.16); Lia through the fourth edition and Sav2 put `interfaces` on the same level as inner classes; and Bro1 does not mention them at all.

We propose a new order of presentation for core object-oriented features. After students have learned to write simple, stand-alone classes for the purposes of inventing new types, packaging data and functionality together, and data hiding, and once good encapsulation practices are in place, the next conceptual step is substitutability and interoperability. The instructor will point out that if several certain classes have overlapping interfaces, they could, in theory, be substituted for each other in the code.

There are many standard examples which readily can be employed for this. Several classes modelling various animals may all have public methods `boolean feed(String food)`, `int weigh()`, and `String provoke()`, yet they accept or reject different food offerings, grow at different rates and based on different foods given, and make different noises when provoked. Various shape classes will all have methods like `double perimeter()` and `double area()`. Classes `Polynomial`, `Rational`, and `Exponential`, modeling different kinds of mathematical functions, will have methods `double evaluate(double x)` and `double integrate(double lower, double upper)`. None of these are likely to involve any code sharing (except that a `Rational` may contain two `Polynomials`, an obvious example of composition); their commonality, rather, is in their interface.

The Java `interface` will then be introduced as the means to put this theory into practice. By writing the classes mentioned above to implement the `interfaces Animal`, `Shape`, and `Function`, the students will learn subtype polymorphism by using variables of the interface type and dynamically dispatching methods on expressions whose static type is the `interface`. Adding the method `Function derivative()` to the `Function` interface will also illustrate how the value returned from a method can be a subtype of the declared return type.

As we alluded already, opportunities to discuss composition, delegation, and aggregation as means to reuse code should come naturally. Once students are comfortable with `interface` implementation, the time will be right to present inheritance/extension as a built-in, convenient way to share code in certain designs. This concept will be introduced by examples where a few of the classes which implement a specific `interface` also have duplicate instance variables and identical implementation of methods. An abstract class—either as a level between the classes and the `interface` or in place of the `interface`—will be used to extract commonality from the classes. Then the more intricate matters, such as overriding, shadowing, and `super`, can be discussed. Finally, for completeness, the instructor will show that non-abstract classes also may be extended; it is our contention, however, that if students have been shown topics in the order presented here and if they adhere to good object-oriented design principles, they will find very infrequent use for extending non-abstract classes.

References

[1] J. Adams, L. Nyhoff, and J. Nyhoff. *Java: An Introduction to Computing*. Prentice Hall, Upper Saddle River, NJ, 2001.

[2] D. Arnow, S. Dexter, and G. Weiss. *Introduction to Programming Using Java: A Object-Oriented Approach*. Addison-Wesley, second edition, 2004.

[3] D. Barnes and M. Kölling. *Objects First with Java: A Practical Introduction using BlueJ*. Prentice Hall, Harlow, England, second edition, 2005.

[4] D. Bell and M. Parr. *Java for Students*. Prentice Hall, Harlow, England, fourth edition, 2005.

[5] J. Bishop. *Java Gently*. Addison-Wesley, third edition, 2001.

[6] G. Bronson. *A First Book of Java*. Brooks/Cole, Pacific Grove, CA, 2002.

[7] G. Bronson. *Object-Oriented Program Development using Java*. Thomson Course Technology, Boston, MA, 2004.

[8] K. Bruce, A. Danyluk, and T. Murtagh. *Java: An Eventful Approach*. Prentice Hall, Upper Saddle River, NJ, 2006.

[9] B. Cornelius. *Understanding Java*. Addison-Wesley, 2001.

[10] H. Deitel and P. Deitel. *Java: How to Program*. Prentice Hall, Upper Saddle River, NJ, sixth edition, 2005.

[11] T. Gaddis. *Starting Out with Java 5: Early Objects*. Addison-Wesley, 2005. Originally published by Scott Jones.

[12] T. Gaddis. *Starting Out with Java 5: From Control Structures to Objects*. Addison-Wesley, 2005. Originally published by Scott Jones.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[14] A. Gittleman. *Computing with Java: Programs, Objects, Graphics*. Addison-Wesley, second edition, 2001. Originally published by Scott Jones.

[15] A. Gittleman. *Computing with Java: Programs, Objects, Graphics*. Addison-Wesley, alternate second edition, 2002. Originally published by Scott Jones.

[16] C. Horstmann. *Computing Concepts with Java Essentials*. Wiley, Hoboken, NJ, third edition, 2003.

[17] S. Kamin, D. Mickunas, and E. Reingold. *An Introduction to Computer Science using Java*. McGraw Hill, New York, NY, second edition, 2002.

[18] E. Koffman and U. Wolz. *Problem Solving with Java*. Addison Wesley, second edition, 2002.

[19] J. Lewis and W. Loftus. *Java: Software Solutions*. Addison Wesley, fifth edition, 2007.

[20] D. Liang. *Introduction to Java Programming*. Prentice Hall, Upper Saddle River, NJ, fifth edition, 2005. Comprehensive version.

[21] D. Malik. *Java Programming: From Problem Analysis to Program Design*. Thomson Course Technology, Boston, MA, second edition, 2006.

[22] D. Malik. *Java Programming: Program Design including Data Structures*. Thomson Course Technology, Boston, MA, 2006.

[23] R. Morelli and R. Walde. *Java, Java, Java: Object-Oriented Problem Solving*. Prentice Hall, Upper Saddle River, NJ, third edition, 2006.

[24] J. Niño and F. Hosch. *An Introduction to Programming and Object-Oriented Design*. Wiley, Hoboken, NJ, second edition, 2005.

[25] I. Pohl and C. McDowell. *Java by Dissection: The Essentials of Java Programming*. Addison-Wesley, 2000.

[26] D. Riley. *The Object of Java*. Addison-Wesley, second edition, 2006.

[27] K. Sanders and A. van Dam. *Object-Oriented Programming in Java: A Graphical Approach*. Addison-Wesley, 2006. Preliminary edition.

[28] W. Savitch. *Java: An Introduction to Problem Solving and Programming*. Prentice Hall, Upper Saddle River, NJ, fourth edition, 2005.

[29] W. Savitch. *Absolute Java*. Addison Wesley, second edition, 2006.

[30] P. Winston and S. Narasimhan. *On to Java 2*. Addison-Wesley, third edition, 2001.

[31] T. Wu. *An Introduction to Object-Oriented Programming with Java*. McGraw Hill, New York, NY, fourth edition, 2006.

ADW Arnow, Dexter, and Weiss, 2004, second edition (first edition in 1998). Includes `interface`s after abstract classes at the end of a chapter on extending class behavior. Motivates as a way "to impose the requirement that certain behavior be adhered to for objects that have no direct relationship to each other in the hierarchy." Self-described as "discussing objects right from the start." [2]

ANN Adams, Nyhoff, and Nyhoff, 2001. Discusses `interface`s only when needed for GUI and event-driven programming. Self-described as using a "spiral" approach, revisiting topics. [1]

BDM Bruce, Danyluk, and Murtagh, 2006. Introduces `interface`s seven chapters before inheritance and extension. The term polymorphism is not used (except "polymorphic method" at the end of the chapter), but is still a motivating concept, as "occasionally we wish to have a variable in a program that can refer to objects from several different classes." Self-described as "objects and events first." [8]

B&K Barnes and Kölling, 2005, second edition (first edition in 2002). Discusses `interface`s as a last topic in a chapter on "Further Abstraction Techniques" including abstract classes. `Interface`s "are similar to abstract classes in which all methods are abstract." Title identifies book as "objects first." [3]

B&P Bell and Parr, 2005, fourth edition (first edition in 1997). Discusses `interface`s not until 13 chapters after inheritance and abstract classes. "Two uses for `interface`s are in design and to promote interoperability," hence motivation is categorized as software development. Self-described as a graphics and GUI-based approach. [4]

Bis Bishop, 2001, third edition (first edition in 1997). Besides an early mention in context of `Enumeration`s, discusses `interface`s in a chapter on abstraction and inheritance, between subclassing and abstract classes. The first motivation is to "encapsulate a guarantee" that a class will provide certain methods, but the larger context is "abstraction" (partially used as a stand-in for "polymorphism"). Intended approach "some of both" of GUI (and requiring objects) first and of foundational concepts first. [5]

Bro1 Bronson, 2002. `Interfaces` completely absent. First half of the book is considered "fundamentals" and does not included object-oriented features, hence fundamentals-first. [6]

Bro2 Bronson, 2004. In a 749-page book, a total of 8 pages on inheritance and polymorphism, with abstract classes and `interfaces` together receiving less than two pages. Intended approach "introduce[s] the concept of objects and classes immediately." [7]

Cor Cornelius, 2001. Anticipates the approach suggested in this paper. "It is better to make a clearer separation between implementation inheritance and `interfaces`. I think it is important to teach `interfaces` early." However, explicit discussion of polymorphism is absent from the book, and `interfaces` are first described as "the ideal construct for documenting the set of operations for a type," hence the software development categorization. Considered here to be fundamentals-first as it consciously is not object-first. [9]

D&D Deitel and Deitel, 2005, sixth edition (first edition in 1996). Chapter on polymorphism introduces polymorphism in context of subclassing, followed by abstract classes, followed by `interfaces` as a case study, demonstrating their use for multiple subtyping. Self-described as "early classes and objects." [10]

Hor Horstmann, 2003, third edition (first edition in 1997). The clearest example of the approach advocated in this paper, introducing `interfaces` together with polymorphism two chapters before inheritance. Self-described as objects- and classes-first. [16]

Gad Gaddis, 2005. Discusses `interfaces` at the end of a chapter on inheritance, after subclassing and abstract classes. "An `interface` is similar to an abstract class that has all abstract methods." Published in two versions, "From Control Structures to Objects" (which briefly mentions `interfaces` in the context of GUI and event programming) and "Early Objects". [12, 11]

Git Gittleman, 2001/2002, second edition (first edition in 1998). Introduces `interfaces` on demand for event-driven programming; a later chapter on inheritance never mentions them. Published in two versions, one for control structures before objects and an "alternate second edition" for objects before control structures (designated here as fundamentals-first and objects-early, respectively). [14, 15]

K&W Koffman and Wolz, 2002, second edition (first edition in 1998). Introduces `interfaces` following extension and polymorphism but before abstract classes, all in a chapter on hierarchies, inheritance, and `interfaces`. The expressed motivation is "to specify a set of requirements that is imposed on a collection of classes" (hence the software development category), but polymorphism was discussed immediately before. Self-described as "classes and objects early." [18]

KMR Kamin, Mickunas, Reingold, 2002, second edition (first edition in 1997). Includes `interfaces` independently for what is essentially the Strategy pattern [13]. Inheritance as a whole (including abstract classes) is only touched on in a short chapter along with packages and exceptions, and polymorphism is absent from the book. New to the second edition is that "object-oriented programming is used from the beginning of the book." [17]

Lia Liang, 2005, fifth edition (first edition in 1998). The latest edition (subtitled "Comprehensive Version") puts `interfaces` as the second half of a chapter including abstract classes and moves towards treating `interfaces` as a variation on abstract classes. Our chart shows the fourth edition, which puts `interfaces` on the same level as inner classes and motivates them by noting "sometimes it is necessary to derive a subclass from several classes." Self-described as "fundamentals-first." [20]

L&L Lewis and Loftus, 2007, fifth edition (first edition in 1997). Describes `interfaces` early (two chapters before subclassing), but without discussion of polymorphism. Thorough treatment of `interfaces` occurs in chapter on polymorphism, after abstract classes. Identified here as "objects-first" because of intent that "all processing should be discussed in object-oriented terms"; however, it claims to "use a natural progression. . . [to] real object-oriented solutions." [19]

Mal Malik, 2006, second edition (first edition in 2003). Mentions `interfaces` early for GUI and event programming. Discussion in chapter on inheritance and polymorphism happens after abstract methods are motivated by the need for multiple inheritance. No clear self-identification of approach. Published with and without CS 2 data structures material. [21, 22]

M&W Morelli and Walde, 2006, third edition (first edition in 2000). `Interfaces` touched on early (before subclassing of polymorphism) in a GUI section. Primary discussion of `interfaces` occurs as the last new information in a chapter on inheritance and polymorphism, after abstract classes, "a third form of polymorphism," though given about equal time with abstract classes. Self-described as "objects-early." [23]

N&H Niño and Hosch, 2005, second edition (first edition in 2001). As advocated here, introduces `interfaces` before inheritance/extension. The motivation of subtyping is shown soon, and hence the categorization with polymorphism, though explicit connection to polymorphism is not made until the chapter on inheritance, in connection with method overriding. Self-described as "objects-first." [24]

P&MD Pohl and McDowell, 2000. Discusses `interfaces` near the end of a chapter on inheritance, after subclassing and abstract classes. Suggests thinking of one as "a pure abstract class," though multiple inheritance is also talked about. Approach "begins by explaining how basic data types and control statements are used traditionally, then progresses to the object-oriented features." [25]

Ril Riley, 2006, second edition (first edition in 2002). Discusses `interfaces` after abstract classes at the end of a chapter on polymorphism, following a chapter on inheritance/extension. Although mentioning that an `interface` "is a bit like the ultimate abstract class," polymorphism is the context and first motivation. Self-described as "object-centric." [26]

Sav1 Savitch, 2005, fourth edition (first edition in 1998). Touches on `interfaces` early (before subclassing or polymorphism) in an optional GUI section. Primary discussion of `interfaces` describes them as an "extreme case of an abstract class," comprising two pages in a 65-page chapter on subclassing, inheritance, and polymorphism. Self-described as flexible, usable for several classroom approaches. [28]

Sav2 Savitch, 2006, second edition (first edition in 2003). Introduces `interfaces` five chapters after polymorphism and abstract classes in a chapter with inner classes, as "the extreme case of an abstract class." Intentionally organized as a reference book rather than for a particular pedagogical approach. [29]

S&VD Sanders and van Dam, 2006. Devotes a chapter to `interfaces` following a chapter on inheritance that includes abstract classes. `Interfaces` "specify a set of responsibilities. . . [and] allow us to highlight the different roles played by each class," hence software development category; however, "`interface` polymorphism" is discussed in the next chapter, though after "inheritance polymorphism." Self-described as objects- (and graphics-) first. [27]

W&N Winston and Narasimhan, 2001, third edition (first edition in 1996). Discusses `interface`s in dedicated chapter, following ones on abstract classes and class hierarchies. Motivates as way "to enforce requirements and document," hence software development category. Seems to be a quick introduction for those with experience in another language. [30]

Wu Wu, 2006, fourth edition (first edition in 1998). Very little on `interface`s, but most of it comes before subclassing to explain the `List interface` and to juggle several versions of a class for incremental design. "[T]akes a full-immersion approach to object-oriented programming." [31]