

---

# Object-oriented Programming: Notes on Liskov's Book

Mike Spivey,  
Michaelmas Term 2007.

I've recommended the book

Barbara Liskov with John Guttag, *Program Development in Java*, Addison-Wesley, 2001.

as a companion to the course. This note clarifies which parts of the book will be especially relevant, and outlines some of the differences in detail between our approach and the one taken in the book.

Broadly speaking, the course *Object-oriented programming I* will cover material contained in the first part of the book, Chapters 1 to 10. We shall not, however, be following the book at all closely: most of the examples discussed in lectures will be different from the ones in the book. However, examples from the book are likely to appear in modified form in the problem sheets and lab exercises.

## Chapter 3: Procedural abstraction

In the course, I shall more-or-less take it for granted that students know how to specify and develop isolated procedures: this is covered in the Procedural Programming course. This course starts with Abstract Data Types (see Chapter 5), but we will in fact revisit the idea of Procedural Abstraction at later stages of the course, particularly in the context of families of procedures that share a common specification. Students who need to revise the ideas behind specifying and developing procedures will find useful material in this chapter of the book.

## Chapter 4: Exceptions

Our use of exceptions will generally be less elaborate than that of Liskov. We will not trouble to introduce many different types of exceptions, but will remain content with only a few.

For us, the specification of a method will often have a non-trivial *pre-condition*, and we will leave it formally unspecified what happens if a method is called in a state and with arguments that do not satisfy the pre-condition. This makes the specifications a lot shorter, as we do not keep having to say

“the exception *NullPointerException* is thrown if the argument  $p$  is null”.

As a matter of defensive programming, however, we will usually arrange to have methods check their pre-conditions at least partially. Usually this is done by writing an `assert` statement at the beginning of the method body. If the assertion is not satisfied (and assertions have not been disabled), this results in an *AssertionError* exception. `Assert` statements are attractive because they are short and clear, and maybe also because checking of assertions can be turned off if it turns out to be too expensive.

Typically, our programs will use this *AssertionError* exception implicitly, and also *Error* and *Failure* exceptions. The *Failure* exception is used where part of the program has yet to be implemented: for example, a first draft of a program may use a fixed-size table for some purpose, and may deal with overflow of the table by throwing a *Failure* exception. Once the rest of the program is working, we might revisit the design of the module that contains the table, and make it deal with overflow by increasing the size of the table dynamically. Perhaps we should not call a program finished until we have eliminated all instances of *Failure*; but the point is that sensible programming leads to a programs that *are* useful even before they are finished.

The *Error* exception will be used in our programs for things that ought not to happen: for example, if a certain type of object represents a set, then we might have to provide a method for fetching an element from the empty set, but that method should never be called in a properly-functioning program. An appropriate thing is for the method body to throw an exception, and we use *Error* for that. Unlike assertions, there is absolutely no point in disabling these checks, because they cost time only if the check fails. Another place where *Error* is used is in the default arm of `switch` statements: often, every case should be covered in one of the labelled branches of the `switch` statement, and it is a useful defence against programming errors to provide a default branch that signals an error. Again, there is no point in disabling this check.

### Chapter 5: Data abstraction

This chapter is the starting point for the course, and it provides useful background for the first few lectures. But we will use different examples, different notation, and different terminology!

The Oxford terminology is a bit different: we specify abstract data types in terms of an *abstract state*, and implement them by choosing a *concrete state* (which Liskov calls the “rep”) a *concrete invariant* (the “rep invariant”) and an *abstraction function* (like Liskov). In place of Liskov’s “requires” and “effects” clauses, we will specify *pre-* and *post-conditions*, usually giving them as formulas in a more mathematical style than Liskov.

### Chapter 6: Iteration abstraction

We shall cover programming with iterators over collections in later parts of the course, and this chapter will be helpful background for that. The proper specification of an iterator as an abstract data type is clear: it represents the sequence of objects that are yet to be returned, and the *hasNext* and *next*

methods of the iterator act on the sequence in the obvious ways. We shall make this specification explicit, and use it to inform our implementation of iterators and combinators on iterators.

**pp. 141-2:** The code for *OrderedIntList* given here will be a useful guide for an alternative implementation of the first lab exercise. Our version of *OrderedIntList* must support the *elementAt* method, however, so Liskov's code needs some amendment. Also, the iterator code on p. 142 sucks big time.

## Chapter 7: Type hierarchy

We will be meeting several instances during the course where several classes implement the same interface, sometimes with a non-trivial specification attached to the interface. Occasionally, we will use abstract classes that have several concrete subclasses. And just once or twice, we will meet a situation where a concrete class has a subclass: these are rarer in practice than most object-oriented books tend to suggest.

Liskov's chapter provides good background for all of these situations. Personally, I find the treatment of the substitution principle a bit vague, and one needs to interpret it afresh in each situation. Liskov does not mention the *fragile base-class problem*, but I shall give a warning about that in the lectures.

## Chapter 8: Polymorphic abstractions

Liskov's book was written before the emergence of Java 1.5, with *generics* and *autoboxing* as new features. These features make the treatment of (parametric) polymorphism much smoother to program, and we shall be using them consistently in the course.

It's true to say that the details of generics of Java are very complicated: whole books have been written about them, and it's certainly true that the chapter on generics in the recommended book *Learning Java* is incomprehensible. But simple uses are easy to follow, and the simple uses are all we shall need.

## Chapter 9: Specifications

Liskov makes some valuable remarks about specifications and how to express them. Our view will be a bit different in two ways: first, Liskov's notion of a *specificand set* differs from a more Oxford-style view. For us, the vital relationship is not between a specification and the set of programs that implements it. Instead, we talk about the relationship of refinement between a specification and another more concrete or more restrictive specification. For us, *programs are specifications* that just happen to be written in a language that can be compiled and executed.

The second difference is that Liskov seems a bit soft (here and elsewhere) on expressing specification mathematically, perhaps because she has been unfortunate in the programmers she has encountered. My own industrial experience has been that it is not too difficult to get good programmers at least

#### 4 *Object-oriented Programming: Notes on Liskov's Book*

to *read* formal specifications, even if the skill of writing a good specification is harder to acquire.