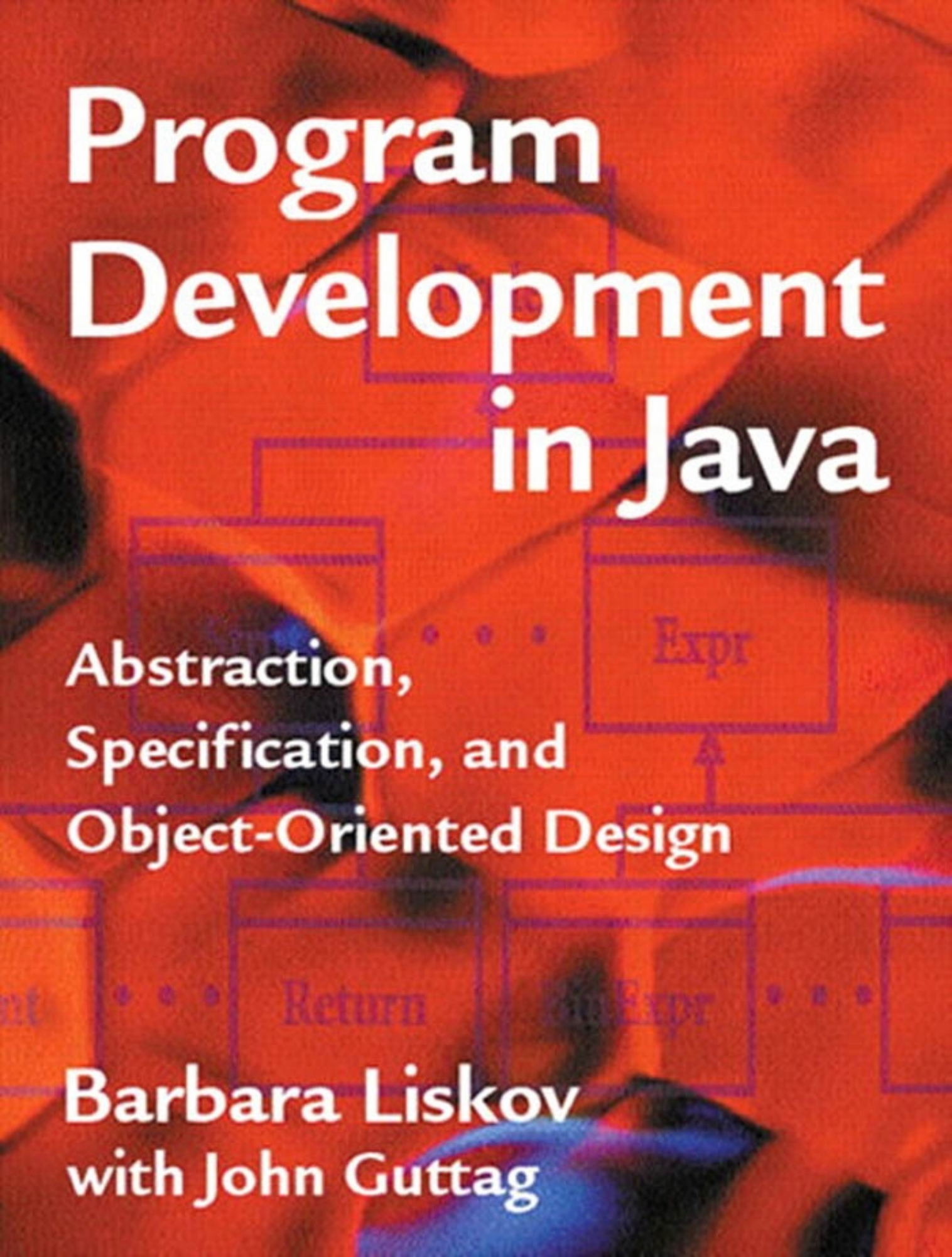
# Program Development in Java

## Abstraction, Specification, and Object-Oriented Design




Barbara Liskov
with John Guttag

# Program Development in Java

examples in this chapter. But sometimes constructors take arguments of their type, and these are *not* creators.

2. *Producers*. These operations take objects of their type as inputs and create other objects of their type. They may be either constructors or methods. For example, `add` and `mul` are producers for `Poly`.
3. *Mutators*. These are methods that modify objects of their type. For example, `insert` and `remove` are mutators for `IntSets`. Clearly, only mutable types can have mutators.
4. *Observers*. These are methods that take objects of their type as inputs and return results of other types. They are used to obtain information about objects. Examples are `size`, `isIn`, and `choose` for `IntSets`, and `coeff` and `degree` for `Polys`.

The creators usually produce some but not all objects; for example, the `Poly` creators (the two constructors) produce only single-term polynomials, while the `IntSet` constructor produces only the empty set. The other objects are produced by producers or mutators. Thus, the producer `add` can be used to obtain `Polys` with more than one term, while the mutator `insert` can be used to obtain sets containing many elements.

Mutators play the same role in mutable types that producers play in immutable ones. A mutable type can have producers as well as mutators; for example, if `IntSet` had a `clone` method, this method would be a producer. Sometimes observers are combined with producers or mutators; for example, `IntSet` might have a `chooseAndRemove` method that returns the chosen element and also removes it from the set.

### 5.8.3 Adequacy

A data type is *adequate* if it provides enough operations so that everything users need to do with its objects can be done both conveniently and with reasonable efficiency. It is not possible to give a precise definition of adequacy, although there are limits on how few operations a type can have and still be useful. For example, if we provide only the `IntSet` constructor and the `insert` and `remove` methods, programs cannot find out anything about the elements in the set (because there are no observers). On the other hand, if we add just the `size` method to these three operations, we can learn about elements in the

set (for example, we could test for membership by deleting the integer and seeing if the size changed), but the type would be costly and inconvenient to use.

A very rudimentary notion of adequacy can be obtained by considering the operation categories. In general, a data abstraction should have operations from at least three of the four categories discussed in the preceding section. It must have creators, observers, and producers (if it is immutable) or mutators (if it is mutable). In addition, the type must be *fully populated*. This means that between its creators, mutators, and producers, it must be possible to obtain every possible abstract object state.

However, the notion of adequacy additionally must take context of use into account: a type must have a rich enough set of operations for its intended uses. If the type is to be used in a limited context, such as a single package, then just enough operations for that context need be provided. If the type is intended for general use, a rich set of operations is desirable.

To decide whether a data abstraction has enough operations, identify everything users might reasonably expect to do. Next, think about how these things can be done with the given set of operations. If something seems too expensive or too cumbersome (or both), investigate whether the addition of an operation would help. Sometimes a substantial improvement in performance can be obtained simply by having access to the representation. For example, we could eliminate the `isIn` operation for `IntSet`s because this operation can be implemented outside the type by using the other operations. However, testing for membership in a set is a common use and will be faster if done inside the implementation. Therefore, `IntSet` should provide this operation.

There can also be too many operations in a type. When considering the addition of operations, you need to consider how they fit in with the purpose of the data abstraction. For example, a storage abstraction like `Vector` or `IntSet` should include operations to access and modify the storage, but not operations unrelated to this purpose, such as a `sort` method or a method to compute the sum of the elements of the vector or set.

Having too many operations makes an abstraction harder to understand. Also, implementation is more difficult, and so is maintenance, because if the implementation changes, more code is affected. The desirability of extra operations must be balanced against these factors. If the type is adequate, its operations can be augmented by standalone procedures that are outside the type's implementation (i.e., static methods of some other class).

## 5.9 Locality and Modifiability

The benefits of locality and modifiability apply to data abstractions as well as to procedures. However, these benefits can be achieved only if we have abstraction by specification.

Locality (the ability to reason about a module by just looking at its code and not any other code) requires that a representation be *modifiable* only within its type's implementation. If modifications can occur elsewhere, then we cannot establish the correctness of the implementation just by examining its code; for example, we cannot guarantee locally that the rep invariant holds, and we cannot use data type induction with any confidence.

Modifiability (the ability to reimplement an abstraction without having to reimplement any other code) requires even more than locality—all access to a representation, even to immutable components, must occur within its type's implementation. If access occurs in some other module, we cannot replace the implementation without affecting that other module. This is why all the instance variables must be declared `private`.

Thus, it is crucial that access to the representation be restricted to the type's implementation. It is desirable to have the programming language help here so that restricted access is guaranteed provided the implementor does not expose the rep. Otherwise, restricted access is another property that must be proved about programs. Java provides support for restricted access via its encapsulation mechanisms.

Sidebar 5.6 summarizes the discussion about locality and modifiability.

### Sidebar 5.6 Locality and Modifiability for Data Abstraction

- A data abstraction implementation provides locality if using code cannot modify components of the rep; that is, it must not expose the rep.
- A data abstraction implementation provides modifiability if, in addition, there is no way for using code to access any part of the rep.

## 5.10 Summary

This chapter has discussed data abstractions: what they are, how to specify their behavior, and how to implement them, both in general and in Java. We discussed both mutable abstractions, such as `IntSet`, and immutable ones, such as `Poly`.

We also discussed some important aspects of data type implementations. In general, we want all objects of the class to be legal representations of the abstract objects; the rep invariant defines the legal representations. The abstraction function defines the meaning of the rep by stating the way in which the legal class objects represent the abstract objects. Both the rep invariant and the abstraction function should be included as comments in the implementation (in the private section of the class declaration). They are helpful in developing the implementation since they force the implementor to be explicit about assumptions. They are also helpful to anyone who examines the implementation later since they explain what must be understood about the rep. Furthermore, the rep invariant and abstraction function should be implemented (as `repOk` and `toString`, respectively) since this makes debugging and testing easier.

In addition, we explored some issues that must be considered in designing and implementing data types. Care must be taken in deciding whether or not a type is mutable; an immutable abstraction can have a mutable rep, however, and observers can even modify the rep, provided these modifications are "benevolent" (i.e., not visible to users). Also, care is needed in choosing a type's operations so that it serves the needs of its users adequately. We also discussed data type induction and how it is used to prove properties of objects. Furthermore, we discussed how having an encapsulated rep is crucial for obtaining the benefits of locality and modifiability.

## Exercises

5.1 Implement a `toString` method for `Polys` (as part of the implementation in Figure 5.7).

5.2 Suppose `IntSets` were implemented using a `Vector` as in Figure 5.6, but the `els` component was kept sorted in increasing size. Give the rep invariant

and abstraction function for this implementation. Also implement `repOk` and `toString`.

5.3 Suppose `Polys` (Figure 5.4) were implemented with the zero `Poly` represented by the empty array. Give the rep invariant and abstraction function for this implementation, and implement `repOk` and `toString`.

5.4 Suppose we wanted a way to create a `Poly` (Figure 5.4) by reading a string from a `BufferedReader`. Specify and implement such an operation. Does the operation need to be implemented inside the `Poly` class (e.g., the one in Figure 5.7), or can it be in a separate class?

5.5 Bounded queues have an upper bound, established when a queue is created, on the number of integers that can be stored in the queue. Queues are mutable and provide access to their elements in first-in/first-out order. Queue operations include

```
IntQueue(int n);
void enq(int x);
int deq ( );
```

The constructor creates a new queue with maximum size `n`, `enq` adds an element to the front of the queue, and `deq` removes the element from the end of the queue. Provide a specification of `IntQueue`, including extra operations as needed for adequacy. Implement your specification. Give the rep invariant and abstraction function and implement `repOk` and `toString`.

5.6 Implement sparse polynomials. Be sure to include the rep invariant and abstraction function and to implement `repOk` and `toString`.

5.7 Specify and implement a rational number type. Give the rep invariant and abstraction function and implement `repOk` and `toString`.

5.8 Consider a `map` data abstraction that maps `Strings` to `ints`. Maps allow an existing mapping to be looked up. Maps are also mutable: new pairs can be added to a map, and an existing mapping can be removed. Give a specification for maps. Be sure your data type is adequate, and if any operations throw exceptions, explain whether they are checked or unchecked. Also implement your specification. Give the rep invariant and abstraction function and implement `repOk` and `toString`.

5.9 Discuss whether the implementations of bounded queues and maps should provide their own implementations of `equals` and `clone` and implement these operations if they are needed.

5.10 Give an informal argument that the implementation of `Poly` in Figure 5.7 preserves the rep invariant.

5.11 Give an informal argument that the implementation of `Poly` in Figure 5.7 is correct.

5.12 Give an informal argument that the following abstract invariant holds for `Polys`:

```
p.degree > 0 => p.coeff(p.degree) ≠ 0
```

5.13 Provide correctness arguments for your implementations of the types mentioned previously (rational numbers, sparse polynomials, bounded queues, and maps).

5.14 Suppose we wanted to evaluate a `Poly` (Figure 5.4) at a given point:

```
int eval(Poly p, int x) throws NullPointerException
  // EFFECTS: If p is null throws NullPointerException else
  //    returns the value of p at x, e.g., eval(x^2 + 3x, 2) = 10.
```

Should `eval` be an operation of `Poly`? Discuss.

5.15 A student proposes a type `matrix` with operations to add and multiply matrices and to invert a matrix. These matrices are mutable; for example, the `invert` operation modifies its argument to contain the inverse of the original matrix. A second student claims that a matrix abstraction ought not to be mutable. Discuss.

5.16 A student says that as long as programs outside a type's implementation cannot modify the rep, we have achieved as much as is possible from data abstraction. Discuss.