Chapter

23 Java

23.1 Core language

Programs are written in Java 1.7. Most programs are compatible with Java 1.6. Java 1.7 features we use are the diamond operator (<>), which reduces the verbosity of declaring and instantiating variables, the `Objects` utility class (which simplifies writing hash functions and comparators), and binary literals and underscores, which makes integral constants easier to read, e.g., `0b001_00001`.

Usually we declare helper classes as static inner classes of the top-level class that provides the solution. You should be comfortable with the syntax used to instantiate static and nonstatic inner classes, including anonymous classes, e.g., the comparator object in Solution 22.17 on Page 426. Our tests live in the `main()` method of the top-level class. (We also have a Junit test suite, which wraps calls to `main()` for each class.)

We follow the Google Java style guide, which you should review before diving into EPI. The guide is fairly straightforward—it mostly addresses naming and spacing conventions, which should not be a high priority when presenting your solution.

23.2 Libraries and constructs

We have elected not to use popular third-party libraries, like Apache Commons, or Google Guava—every program uses the JDK, and nothing else. We have written a very small number of utility functions in `com.epi.utils`; these are used excusively in the test code, e.g., for filling and printing collections.

The JDK container classes and interfaces that we use, in order of decreasing frequency are given in Table 23.1 on the facing page. You are strongly encouraged to review their Javadoc. Note that we do not use the legacy `Stack` class, preferring instead to use the `Deque` interface with a `LinkedList` implementation.

Wherever possible, you should use the standard library for containers. Some problems require you to write your own container classes. For example, since node

Random	List	ArrayList	Arrays	Collections
LinkedList	Set	Map	HashSet	HashMap
Deque	Objects	Comparator	PriorityQueue	TreeSet
Queue	Iterator	SortedSet	TreeMap	LinkedHashMap
BigInteger	NavigableSet	NavigableMap	BitSet	

Table 23.1: JDK classes and interfaces used in EPI. This does not include the most basic library types and methods, e.g., `Integer`, `String`, `java.util.Math`, `java.io`, etc.

ListNode	DoublyListNode	BinaryTreeNode	PostingListNode
Stack	Queue	BSTNode	

Table 23.2: EPI container classes.

objects are not exposed by `TreeSet`, you have to write your own BST class when asked to reconstruct a BST from traversal data. The container classes we implemented are described in Table 23.2 on the next page. (Note that Queue appears in both the JDK and EPI lists—this is because in some cases we are required to implement a queue, in other cases we can use an implementation of the Queue interface in JDK.)

23.3 Best Practices for Interview Code

23.3.1 EPI practices not suitable for production

First, we describe practices we use in EPI that are not suitable for production code, but are necessitated by the finite time constraints of an interview. See Section 2 on Page 13 for more insights into coding for an interview.

– We make fields public, rather than use getters and setters.
– We do not protect against invalid inputs, e.g., null references, negative entries in an array that's supposed to be all nonnegative, input streams that contain objects that are not of the expected type, etc.
– Since we want programs to be standalone for ease of readability, we have duplicate code in our codebase.
– We occasionally use static fields to pass values—this reduces the number of classes we need to write, at the cost of losing safety in the presence of concurrent modification.
– We use IO-exceptions to detect the end of a stream, rather than encoding the number of objects at the start of the stream, or using a sentinel object to mark the end.

23.3.2 EPI practices not suitable for an interview

There are some practices we follow in EPI, which are industry-standard, but we would not recommend for an interview.

– We use long identifiers for pedagogy, e.g., `queueOfMaximalUsers`. In an actual interview, you should use shorter, less descriptive names than we have in our programs—writing `queueOfMaximalUsers` repeatedly is very time-consuming compared to writing `q`.

– When specifying types, we use the weakest type that provides the functionality we use, e.g., the argument to a function for computing the *k*th smallest element in an array is a `List`, rather than `ArrayList`. This is generally considered a best practice—it enables code reuse via polymorphism—but is not essential to use in an interview. Spend your time making sure the program works, not showing off that you know when to use `RandomAccess` over `List` as an argument type.
– When raising exceptions, we use the appropriate exception type, e.g., `NoSuchElementException` when dequeing an empty queue. This is not needed in an interview, where throwing `RunTimeException` suffices.

An industry best practice that we use in EPI, which we recommend you use in an interview is explicitly creating classes for "data clumps", i.e., groups of values that do not have any methods on them, such as a starting index and an ending index into an array. Many programmers would use a generic `Pair` or `Tuple` class, but we have found that this leads to confusing, buggy programs. (It also makes the interviewers job harder, something you should avoid.)

23.4 Java Resources

Our favorite Java reference is Peter Sestoft's "Java Precisely", which does a great job of covering the language constructs with examples in a very concise manner. The definitive book for Java is "Java: The Complete Reference" by Oracle Press—this is the go-to resource for answering questions about order of initialization, resolving ambiguity, etc.

Joshua Bloch's "Effective Java" (second edition) one of the best all-round programming books we have come across. It addresses general programming principles, such as ways in which to construct objects and the pitfalls of inheritance, to Java-specific features such as `enum` and the `Executor` framework.

For design patterns, we like "Head First Design Patterns" by Freeman *et al.* It's primary drawback is its verbosity. Tony Bevis' "Java Design Pattern Essentials" conveys the same content in a more succinct, though less entertaining, manner. (Note that our programs are written at too small a scale to take advantage of design patterns.)

"Java Concurrency in Practice", by Goetz *et al.* does a wonderful job of explaining pitfalls and practices around multithreaded Java programs.