The Java™ Tutorials

Trail: Collections

The Java Tutorials have been written for JDK 8. Examples and practices described in this page don't take advantage of improvements introduced in later releases.

Lesson: Implementations

Implementations are the data objects used to store collections, which implement the interfaces described in the Interfaces section. This lesson describes the following kinds of implementations:

- **General-purpose implementations** are the most commonly used implementations, designed for everyday use. They are summarized in the table titled General-purpose-implementations.
- **Special-purpose implementations** are designed for use in special situations and display nonstandard performance characteristics, usage restrictions, or behavior.
- **Concurrent implementations** are designed to support high concurrency, typically at the expense of single-threaded performance. These implementations are part of the `java.util.concurrent` package.
- **Wrapper implementations** are used in combination with other types of implementations, often the general-purpose ones, to provide added or restricted functionality.
- **Convenience implementations** are mini-implementations, typically made available via static factory methods, that provide convenient, efficient alternatives to general-purpose implementations for special collections (for example, singleton sets).
- **Abstract implementations** are skeletal implementations that facilitate the construction of custom implementations — described later in the Custom Collection Implementations section. An advanced topic, it's not particularly difficult, but relatively few people will need to do it.

The general-purpose implementations are summarized in the following table.

General-purpose Implementations

Interfaces	Hash table Implementations	Resizable array Implementations	Tree Implementations	Linked list Implementations	Hash table + Linked list Implementations
`Set`	`HashSet`		`TreeSet`		`LinkedHashSet`
`List`		`ArrayList`		`LinkedList`	
`Queue`					
`Deque`		`ArrayDeque`		`LinkedList`	
`Map`	`HashMap`		`TreeMap`		`LinkedHashMap`

As you can see from the table, the Java Collections Framework provides several general-purpose implementations of the `Set`, `List` , and `Map` interfaces. In each case, one implementation — `HashSet`, `ArrayList`, and `HashMap` — is clearly the one to use for most applications, all other things being equal. Note that the `SortedSet` and the `SortedMap` interfaces do not have rows in the table. Each of those interfaces has one implementation (`TreeSet` and `TreeMap`) and is listed in the `Set` and the `Map` rows. There are two general-purpose `Queue` implementations — `LinkedList`, which is also a `List` implementation, and `PriorityQueue`, which is omitted from the table. These two implementations provide very different semantics: `LinkedList` provides FIFO semantics, while `PriorityQueue` orders its elements according to their values.

Each of the general-purpose implementations provides all optional operations contained in its interface. All permit `null` elements, keys, and values. None are synchronized (thread-safe). All have *fail-fast iterators*, which detect illegal concurrent modification during iteration and fail quickly and cleanly rather than risking arbitrary, nondeterministic behavior at an undetermined time in the future. All are `Serializable` and all support a public `clone` method.

The fact that these implementations are unsynchronized represents a break with the past: The legacy collections `Vector` and `Hashtable` are synchronized. The present approach was taken because collections are frequently used when the synchronization is of no benefit. Such uses include single-threaded use, read-only use, and use as part of a larger data object that does its own synchronization. In general, it is good API design practice not to make users pay for a feature they don't use. Furthermore, unnecessary synchronization can result in deadlock under certain circumstances.

If you need thread-safe collections, the synchronization wrappers, described in the Wrapper Implementations section, allow *any* collection to be transformed into a synchronized collection. Thus, synchronization is optional for general-purpose implementations, whereas it is mandatory for legacy implementations. Moreover, the `java.util.concurrent` package provides concurrent implementations of the `BlockingQueue` interface, which extends `Queue`, and of the `ConcurrentMap` interface, which extends `Map`. These implementations offer much higher concurrency than mere synchronized implementations.

As a rule, you should be thinking about the interfaces, *not* the implementations. That is why there are no programming examples in this section. For the most part, the choice of implementation affects only performance. The preferred style, as mentioned in the Interfaces section, is to choose an

implementation when a `Collection` is created and to immediately assign the new collection to a variable of the corresponding interface type (or to pass the collection to a method expecting an argument of the interface type). In this way, the program does not become dependent on any added methods in a given implementation, leaving the programmer free to change implementations anytime that it is warranted by performance concerns or behavioral details.

The sections that follow briefly discuss the implementations. The performance of the implementations is described using words such as *constant-time*, *log*, *linear*, *n log(n)*, and *quadratic* to refer to the asymptotic upper-bound on the time complexity of performing the operation. All this is quite a mouthful, and it doesn't matter much if you don't know what it means. If you're interested in knowing more, refer to any good algorithms textbook. One thing to keep in mind is that this sort of performance metric has its limitations. Sometimes, the nominally slower implementation may be faster. When in doubt, measure the performance!

Previous page: Previous Lesson
Next page: Set Implementations