

# Design Patterns in JDK Collections

## Motivation:

Teaching Java is more challenging than teaching C++ or C, since instructors must at least survey various bundled and closely-related toolkits, beginning with the JDK API; for example, Swing, AWT, JDBC, JAXP, and the internationalization and collections frameworks, among others. Students of Java must learn when and how to leverage them; no one whose first instinct would be to implement a parser for some "little language" from scratch would find a job, let alone survive a first course in Java. Whatever sidebars introducing Design Patterns most texts offer are motivated by coverage of Swing, Java collections, or EJB. This is a tacit recognition that understanding design patterns is more critical to understanding these frameworks than, say, mechanics of the Java language.

All these frameworks exemplify dependence on design patterns. Even a textbook such as Horstmann and Cornell<sup>1</sup>, although mentioning GOF patterns in connection with Java collections/Swing, merely scratches the surface, instead of structurally integrating them. The collections framework, in particular, offers a fairly well-defined and accessible domain within which design patterns enable achieving key architectural goals. Following the development of the collections framework through successive releases offers a case study on how design patterns interact with supporting graceful software evolution. At a tactical level, the collections framework provides a rich repository of decisions about how to implement design patterns, demonstrating how features of the Java language which may appear unmotivated when presented in textbook examples facilitate resolving implementation issues.

## Background:

Designing a framework to provide basic data structures support presents one of the most daunting object-oriented design challenges, despite or perhaps partly because various procedural solutions are readily available. Eiffel, many years ago, featured an elegant but little-used and little-known object-oriented collections framework. By contrast, all commercially-viable high-level languages offer data structures support, but most avoid designing a framework; Perl, for example, has hard-wired arrays and hashes; Visual Basic and Python offer similar facilities. Very early versions of Java made available only the stand-alone classes `Hashtable` (with its convenient and still widely-used subclass `Properties`), and `Vector` (with its "please-don't-notice-me" subclass `Stack`, poster child for the antipattern "How to Misuse Public Inheritance"). Starting with JDK 1.2, however, Java established an elaborate collections framework, even managing to integrate the pre-existing `Vector` and `Hashtable`; and has followed through with consistent expansions; for example, the `PriorityQueue` interface and a

---

1. Cay S. Horstmann and Gary Cornell, *Core Java 2, Volume 1: Fundamentals*, 5th edition, Prentice Hall PTR, 1999; and *Volume II; Advanced Features*, 7th edition, Prentice Hall PTR, 2004.

new subpackage encapsulating sophisticated concurrency control in JDK 1.5. Java was somewhat influenced by the very carefully-designed collections framework in Standard C++, which integrates a library of modular and extensible generic algorithms together with generic data structures. Stroustrup notes that design constraints for the standard C++ library were much more stringent than for regular applications, enumerating among others:

- Invaluable and affordable to essentially every student and professional programmer.
- Used directly or indirectly by every programmer for everything within the scope of the library.
- Primitive in the mathematical sense That is, ... individual components [should be] designed to perform only a single role.
- Convenient, efficient, and reasonably safe for common use.
- Complete at what they do,
- Supportive of commonly-accepted programming styles.

In particular, feel the tension between these goals: primitive operations are often the opposite of complete or convenient; what is maximally efficient is often minimally safe; commonly-accepted programming styles may be unusable or unaffordable to those at either end of the newbie-to-professional programmer continuum. This explains why design patterns are so crucial to the architecture of the Java collections framework; they help provide solutions to these problems, all of which are common, but not usually confronted at the same time, as here. In particular, we will focus on Iterator and Template Method, then briefly look at support for concurrency.

Let's begin with Iterator, which satisfies nearly all the goals just listed: it is used by everyone who uses JDK collections; convenient and supportive of commonly-used programming styles; reasonably safe; and complete, yet based on only a few primitive operations. That is, the Iterator design pattern resolves the tension between the demanding and mutually-conflicting design goals for a standard framework.

## **Iterator**

Iterators are so ubiquitous that it's hard to imagine Java collections without them. The benefits of applying the Iterator design pattern can be appreciated more fully from the perspective of a similar, but more ordinary, set of requirements: Suppose you were working an application requiring several specialized collections-like classes: perhaps for customer, or inventory control, or network equipment records. In the context of your application, you might focus on organizing, navigating, and manipulating these elements according to their specific interfaces and interrelationships; in particular, whatever supported some smoothly-running web of objects above and beyond the RDBMS from which they would be retrieved/saved.

If this approach were applied to the basic collections classes provided by the JDK, each type of collections class would have its own customized access methods: direct-access collections based on integer positions or generic parameters, sequential collections doing forward and/or backward fetches based on current position. To traverse a whole collection, you would need to know what sort of collection it was, then either initiate ordered/unordered accesses of all keys, or get the first/last element and fetch in a loop.

This would satisfy the goal of supporting commonly-accepted programming practices: there's comfort in writing application code completely controlling customized access. However, it is not complete since as it lacks a common mechanism for accessing all elements in any collection, which has to be reinvented in each case; it also lacks convenience, and potentially efficiency and safety, depending on each particular implementation devised. In response, commonly-named methods could be introduced to each collections class; however, because their overall strategies and specific method signatures would differ, complete traversals would retain dependencies on the type of collection.

The option of delegating traversals to collection class implementations exposes the binding to previously implicit decisions about extrafunctional concerns including scalability, safety, and robustness being hard-coded in applications code. For example, to support several non-concurrent traversals of the same collection -- say, in nested loops -- either all application code or each collection instance needs to keep track of state corresponding to in-process traversals. Iterator, of course, solves this dilemma: each iterator encapsulates the state for one traversal. What remains to be resolved by applying the Iterator design pattern is the exact demarcation between the iterator class and its corresponding collection: obviously, the iterator needs to know intimate details of its collection in order to perform effective and efficient traversals; this will be considered further under implementation issues below.

Consider more sophisticated extrafunctional concerns, such as whether traversal is permitted to modify a collection, and insure or require integrity guarantees in the presence of potential access by multiple concurrent threads. These even more clearly should not be left to haphazard resolution in multiple hard-coded implementations scattered across applications code and/or collections classes. Establishing a standard mechanism by supplying Iterator across all collections is one prerequisite for finding a good solution.

The introduction of the JDK 1.2 collections framework did not avoid one "antipattern," which subverts the key motivation of using Iterator to represent a common access and traversal protocol for all aggregate instances. The original JDK collections, **Vector** and **Hashtable**, were unified in their use of the **Enumeration** interface, which offered only minimal functionality. The GOF description<sup>1</sup> lists the basic Iterator operations as: **First()**, **Next()**, **IsDone()**, and **CurrentItem()**. JDK **Enumeration** subsumes **First()** into the Factory Method **elements()** from which an **Enumeration** instance is obtained; and provides only **hasMoreElements()** and **nextElement()**, which merges **Next()** and **CurrentItem()**, as suggested by a footnote in GOF. JDK 1.2 introduced the semantically equivalent interface **Iterator**, with methods **hasNext()** and **next()**, corresponding to **hasMoreElements()** and **nextElement()**, respectively; as well a mutating **remove()**. Thus, Java newbies are confronted with legacy code that uses **Enumeration** and current code that uses **Iterator** -- defeating the goal of a common protocol. This was mitigated by retrofitting **Iterator** to be available from **Vector**, **Hashtable** (and **Properties**); thus, the issue can be reduced to understanding legacy code; all new code can use the now-

---

1. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns CD*, Addison Wesley Longman, 1997.

standard Iterator.

Examining more closely how JDK Iterators handle the "Implementation Issues" raised by the original GOF exposition exposes interesting details of applying Iterator as a design pattern, in the context of a Java implementation.

### Who controls the Iterator?

JDK Iterators are clearly external iterators; that is, application code has the burden of explicitly stepping through all elements. The alternative, internal iterators, may be exemplified by C++ standard library algorithms, especially as customizable with function objects. External iterators, of course, offer a much more familiar style to many programmers, as well as being more obviously flexible; for example, it is very simple to step through comparing collections with disparate internal structures. In JDK 1.5, this coding burden has been lightened somewhat by the introduction of `foreach`, freeing application code from the need to define meaningless integer variables. For example: <sup>1</sup>

Code prior to JDK 1.5:

```
public boolean containsAll(Collection c) {
    for(Iterator iter = c.iterator(); iter.hasNext();) {
        if (!contains(iter.next())) {
            return false;
        }
    }
    return true;
}
```

Code for JDK 1.5

```
public boolean containsAll(Collection c) {
    for (Object o : c) {
        if (!contains(o)) return false;
    }
    return true;
}
```

### Who defines the traversal? Iterators may need privileged access.

Either the Iterator may be defined with very restricted functionality, such that all traversal logic is embedded in corresponding collection methods, or the Iterator can contain all details involved in performing the traversal, and thus most likely require access to private details within the collection. JDK finesses this distinction by providing iterators as nested classes within their corresponding collections. This leverages the Java language encapsulation loophole automatically available to nested classes, and neatly solves the need to provide every iterator with a reference to the collection it services, since Java populates this whenever each instance of a nested class gets constructed. The advantages of using a nested class to resolve this tension between iterators and collections provide a compelling example for students, who may only have been exposed to nested classes for creating Swing Listeners and Adapters.

---

1. Dion Almaer from, <http://www.oreillynet.com/pub/wlg/3332>

The collections classes and their iterators likewise provide an example of the Parallel Hierarchy/Dual Hierarchy pattern: collections implement `Collection`, iterators implement `Iterator`. Again, the general concept, that a nested class implements an interface orthogonal to the interface being implemented by its enclosing class, is worth emphasizing to students learning how to design and write Java classes.

### **Using polymorphic Iterators**

The GOF points out that iterators must be dynamically allocated by Factory Method, a design pattern which solves Gregor Kiczales' "make in not generic" problem by providing a common interface from which new instances of particular but unspecified subclasses can be obtained, instead of hard-coding constructor invocations. That's exactly how it works in the JDK: the mnemonically named `iterator()` Factory Methods in each collection class return appropriate `Iterator` implementations. Of course, Java makes the universal assumption that paying the costs of dynamic allocation far outweighs the costs of maintaining painstaking code to do faster and/or cheaper allocation tailored to some particular application's storage utilization profile. Indeed, as noted above, standard practice is to always acquire a new `Iterator`, instead of resetting a previously-created instance; this should be emphasized to students with a background in C and/or C++, because it cuts against their instincts. JDK garbage collectors continue to be focused on providing better support for an abundance of cheap, short-lived, new objects such as iterator instances.

### **Additional Iterator operations**

Here the Java collections framework provides a very nice example of how to extend interfaces; students will likely be much more familiar with extending classes. As mentioned above, `Iterator` provide only `delete()` in addition to the minimal set of operations; but `ListInterface` extends `Iterator` by adding `previous()`, `has Previous()`, `nextIndex()`, `previousIndex()`, `set()`, and `add()`. As its name suggests, this version is most appropriately available for collections backed by list-like, sequential data structures. `ListIterator` clearly defines a more powerful set of operations, distinct from any particular data structures supporting it. Simply adding extra operations to certain `Iterator` subclasses, as supplied by corresponding `Collection` subclasses capable of supporting them, misses the opportunity to make available this useful mechanism. `List Interface` further provides an example of natural "additive extension," in contrast to the problematic "reneging" -- that is, by throwing `OperationNotSupportedException` -- caused by the compromise of providing operations like `add()` at the overly general level of `Collection`, in order to prevent proliferation of intermediate interfaces.

### **How robust is the Iterator?**

If an `Iterator` can `remove()` from the collection over which it operates, and `ListIterator` can `add()` and `set()`, then multiple instances of iterators can make inconsistent modifications simultaneously. Therefore, `Iterator` should be robust enough to prevent or at least recognize this danger. JDK 1.2 offers fail-safe iterators, which short-circuit when simultaneous attempts at modification are detected, to prevent further damage. This is somewhat similar to optimistic concurrency control, which aborts a contradictory update with a message to the user, requiring manual intervention. It's up

to the Java programmer to recognize whether simultaneous updates may be endangering a collection, then re-code the application to avoid the real or detected danger. The mechanism itself is implemented within superclasses, such as **AbstractList**, for consistent enforcement; **checkForComodification()**, the non-public method which factors out detection, is marked **final**. A count of modifications made is kept by by each iterator and the collection itself; if these counts disagree, then **ConcurrentModificationException** gets thrown. Since this is a checked **Exception**, the Java compiler enforces awareness that the situation should be handled. Of course, for a real solution to using collections from simultaneous threads, use the concurrent collections classes introduced in JDK 1.5.

## Template Method:

Template Method is one of the more neglected design patterns, perhaps because it is so fundamental. The GOF "cocktail-party" definition is: <sup>1</sup>

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Methods let subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Polymorphism is generally presented by examples overriding individual methods within easily-conceptualized class trees, such as ISDN and POTS subclasses specializing implementations of **errorRate()** and **throughput()** specified by their superclass **Circuit**. However, an equally valuable use of polymorphism enables consistent reuse of common algorithms, which encode ordering between methods as well as interrelationships between the interfaces supplying corresponding implementations. Template Method offers one realization of this more general polymorphism; and the JDK collections framework clearly uses template methods to advantage, both within abstract collection classes, and to provide sorting, a key algorithm for which many other competitive frameworks resort to more awkward callback mechanisms.

Consider **AbstractList**. Of course, it contains, as expected, abstract methods such as **get(int index)**. However, **AbstractList** also includes the Template Method **addAll(int index, Collection<? extends E> c)**, which supplies as arguments a position for insertion and another collection instance, to insert all this collection's elements into any subclass of **AbstractList**. **addAll()** relies on obtaining the **Iterator** provided by the other collection, then walking through its elements, invoking the **add()** method of the list being modified. This demonstrates Inversion of Control: **AbstractList** specifies the basic mechanism for adding another collection's elements, but depends upon implementations of **iterator()** and an **Iterator** subclass provided by that collection, as well as depending on its own subclass's implementation of **add()**. Both the **Iterator** and **ListIterator** subclasses are defined within **AbstractList**, thus making them available for use by all subclasses. Leveraging this, **AbstractList** implements several methods

---

1. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns CD*, Addison Wesley Longman, 1997

relying only on the functionality of `ListIterators`, including `indexOf()` and `lastIndexOf()`. This illustrates placing code as high as possible in the class hierarchy to maximize reuse. Students will have seen examples of implementing non-abstract methods in superclasses, colocated with member variables upon which such methods depend. However, here the superclass method leverages a shared mechanism instead of a shared variable.

Students often are warned how important it is to properly define both `equals()` and `hashCode()` when using any `Map` or `Set`; just as they are warned it is important to properly define `compareTo()`, or supply a `Comparator`, for any `SortedMap`. Understanding the dependency created by the collections' reliance on a template method which must be provided by application code explains why. A `Set`, for example, may blithely contain duplicate instances if the application inserts elements, like `PhoneNumber`, with equality-by-value semantics (as contrasted with equality-by-identity), but neglects to override the `equals()` method inherited from `Object` for `PhoneNumber`.<sup>1</sup> This is because all basic collection operations such as `find()`, `add()`, and `remove()` rely on the template method `equals()`, as implemented by the class of which the collection contains instances.

`HashMap` and `HashSet`, likewise, are completely dependent on invoking `compareTo()` to position their elements in a total ordering. Of course, if the collection doesn't need to be continuously maintained in a single well-defined ordering -- say, only sorted once at end of processing for display purposes, or instead requires re-sorting several times according to different collation rules -- then the algorithm `Arrays.sort()` is probably a better fit than a `SortedMap`. Note that `Arrays.sort()` relies on the template method `compareTo()` in exactly the same way. For students who have a background in C and/or C++, comparing `Arrays.sort()` against the C `qsort()` with its ugly void pointers and function type parameters, or even the elegant but esoteric C++ template function `sort()` with its generic iterator parameters, offers convincing proof that applying the template method design pattern yields a better solution for the callback mechanism required by any all-purpose sorting algorithm. Considering `Comparable` in this context further demonstrates the possibilities for establishing common protocols by means of Java interfaces, beyond extension through subclassing.

## Support for Concurrency:

From the start, Java supported multithreading. However, perspectives on who would use it, how, and why have changed greatly since then, as well as the specific mechanisms supporting it. The collections framework, in particular, offers a case study in attempting to better satisfy the demanding set of design constraints for a standard data structures framework. Initially, both `Vector` and `Hashtable` made all methods synchronized, presumably to offer universal protection against integrity problems due to inadvertent simultaneous uncoordinated manipulations. However, this violates the design goals for a standard framework: such classes may no longer be affordable by all

---

1. Joshua Bloch, *Effective Java Programming Guide*, Addison-Wesley Professional, 2001.

users, due to the costs of mandatory synchronization overhead. They also are no longer primitive, due to always providing protection against simultaneous access from multiple threads in addition to their core functionality of containing elements.

Over time, it was realized that many applications would not need protection against multithreading because they would run within an environment which provided it for them; for example, EJB code is specifically prohibited from using threads since one primary benefit of using an EJB container is how it manages all issues related to thread scheduling and resources. So, clearly, bundling this into the core JDK collections is inappropriate; yet some applications -- if only due to backwards compatibility -- will want to rely on this being available from the standard framework. With the JDK 1.2 collections framework, the Decorator design pattern provided a solution. Wrapper static methods in the `Collections` class return synchronized, as well as unmodifiable and checked versions, of all collection interfaces; for example,

```
Map<String, Circuit> ckts = Collections.synchronizedMap(new  
    HashMap<String, Circuit>);
```

Similarly, `unmodifiableMap()` and `checkedMap()` are available. Here, the Decorator pattern helps avoid a proliferation of classes; if the application needs an unmodifiable checked Map, that is readily available by layering two Decorators. The same interface is preserved, regardless of whether or by how many versions a collection is wrapped; so transparency of application usage is preserved.

The new subpackage `java.util.concurrent` introduced in JDK 1.5 ignores the design goal of being "used directly or indirectly by every programmer"; but then, that is the role of JDK subpackages, such as `java.util.text`; not all applications require pattern recognition capabilities, although some rely on it being provided by the standard library.

What matters more is that the new concurrency collections classes satisfy the goals of user convenience, efficiency, and reasonable safety, as well as being supportive of common practices. For example, users now have access to `Futures`, which can be handed off to deliver the result of having run a thread, whereas `Runnable` never supported a return code and required extra code to track its execution.

`ConcurrentHashMap` and `ConcurrentLinkedQueue` present pre-packaged solutions to maintaining data integrity in unmanaged multithreaded environments.

Moreover, the concurrency collections also satisfy the design goals of being "primitive" and "affordable to all" by making available the mechanisms on which such high-level data structures are based. `Exchanger`, `SynchronousQueue`, `CyclicBarrier`, `CountDownLatch`, and `Semaphore` comprise implementations of the most commonly useful concurrency design patterns, and could easily provide the backbone for a seminar in object-oriented concurrency theory and practice.

Christine Bouamalay  
19 September 2005