

## Java Concurrency

### Goetz: Chapter 5 Building Blocks

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

General-purpose Implementations					
Interfaces	Implementations				
	Hash table	Resizable array	Tree	Linked list	Hash table + Linked list
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Queue					
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.

## Wrapper Implementations

Wrapper implementations delegate all their real work to a specified collection but add extra functionality on top of what this collection offers. For design pattern fans, this is an example of the *decorator* pattern. Although it may seem a bit exotic, it's really pretty straightforward.

These implementations are anonymous; rather than providing a public class, the library provides a static factory method. All these implementations are found in the [Collections](#) class, which consists solely of static methods.

## Synchronization Wrappers

---

The synchronization wrappers add automatic synchronization (thread-safety) to an arbitrary collection. Each of the six core collection interfaces — [Collection](#), [Set](#), [List](#), [Map](#), [SortedSet](#), and [SortedMap](#) — has one static factory method.

```
public static <T> Collection<T>
    synchronizedCollection(Collection<T> c);
public static <T> Set<T>
    synchronizedSet(Set<T> s);
public static <T> List<T>
    synchronizedList(List<T> list);
public static <K,V> Map<K,V>
    synchronizedMap(Map<K,V> m);
public static <T> SortedSet<T>
    synchronizedSortedSet(SortedSet<T> s);
public static <K,V> SortedMap<K,V>
    synchronizedSortedMap(SortedMap<K,V> m);
```

Each of these methods returns a synchronized (thread-safe) collection backed up by the specified collection. To guarantee serial access, *all* access to the backing collection must be accomplished through the returned collection. The easy way to guarantee this is not to keep a reference to the backing collection. Create the synchronized collection with the following trick.

```
List<Type> list =
    Collections.synchronizedList(new ArrayList<Type>());
```

A collection created in this fashion is every bit as thread-safe as a normally synchronized collection, such as a [Vector](#).

In the face of concurrent access, it is imperative that the user manually synchronize on the returned collection when iterating over it. The reason is that iteration is accomplished via multiple calls into the collection, which must be composed into a single atomic operation. The following is the idiom to iterate over a wrapper-synchronized collection.

```
Collection<Type> c =
    Collections.synchronizedCollection(myCollection);
synchronized(c) {
    for (Type e : c)
        foo(e);
}
```

If an explicit iterator is used, the `iterator` method must be called from within the synchronized block. Failure to follow this advice may result in nondeterministic behavior. The idiom for iterating over a `Collection` view of a synchronized `Map` is similar. It is imperative that the user synchronize on the synchronized `Map` when iterating over

any of its `Collection` views rather than synchronizing on the `Collection` view itself, as shown in the following example.

```
Map<KeyType, ValType> m =
    Collections.synchronizedMap(new HashMap<KeyType, ValType>());
    ...
Set<KeyType> s = m.keySet();
    ...
synchronized(m) { // Synchronizing on m, not s!
    while (KeyType k : s)
        foo(k);
}
```

One minor downside of using wrapper implementations is that you do not have the ability to execute any *noninterface* operations of a wrapped implementation. So, for instance, in the preceding `List` example, you cannot call `ArrayList`'s [ensureCapacity](#) operation on the wrapped `ArrayList`.

## Unmodifiable Wrappers

---

Unlike synchronization wrappers, which add functionality to the wrapped collection, the unmodifiable wrappers take functionality away. In particular, they take away the ability to modify the collection by intercepting all the operations that would modify the collection and throwing an `UnsupportedOperationException`. Unmodifiable wrappers have two main uses, as follows:

- To make a collection immutable once it has been built. In this case, it's good practice not to maintain a reference to the backing collection. This absolutely guarantees immutability.
- To allow certain clients read-only access to your data structures. You keep a reference to the backing collection but hand out a reference to the wrapper. In this way, clients can look but not modify, while you maintain full access.

Like synchronization wrappers, each of the six core `Collection` interfaces has one static factory method.

```
public static <T> Collection<T>
    unmodifiableCollection(Collection<? extends T> c);
public static <T> Set<T>
    unmodifiableSet(Set<? extends T> s);
public static <T> List<T>
    unmodifiableList(List<? extends T> list);
public static <K,V> Map<K, V>
    unmodifiableMap(Map<? extends K, ? extends V> m);
public static <T> SortedSet<T>
    unmodifiableSortedSet(SortedSet<? extends T> s);
public static <K,V> SortedMap<K, V>
    unmodifiableSortedMap(SortedMap<K, ? extends V> m);
```

## Compound Actions on Vector that may produce confusing results

```
package net.jcip.examples;

import java.util.*;

/**
 * UnsafeVectorHelpers
 * <p/>
 * Compound actions on a Vector that may produce confusing results
 *
 */
public class UnsafeVectorHelpers {

    public static Object getLast(Vector list) {
        int lastIndex = list.size() - 1;
        return list.get(lastIndex);
    }

    public static void deleteLast(Vector list) {
        int lastIndex = list.size() - 1;
        list.remove(lastIndex);
    }
}
```

If thread A calls `getLast` while thread B calls `deleteLast`, thread A may see an `IndexOutOfBoundsException` since the last element may have been deleted. Very confusing and unexpected to the writer of thread A.

To avoid this problem, use client-side locking via synchronization. Will the following work?

```
public class UnsafeVectorHelpers {

    public synchronized static Object getLast(Vector list) {
        int lastIndex = list.size() - 1;
        return list.get(lastIndex);
    }

    public synchronized static void deleteLast(Vector list) {
        int lastIndex = list.size() - 1;
        list.remove(lastIndex);
    }
}
```

## Client Side Locking that solves the problem.

The problem we are trying to solve is that between the call to `size()` and the call to `get()`, the list may change. Here we must **use the same lock** as the one used by the thread-safe Vector collection.

```
package net.jcip.examples;

import java.util.*;

/**
 * SafeVectorHelpers
 * <p/>
 * Compound actions on Vector using client-side locking
 */
public class SafeVectorHelpers {
    public static Object getLast(Vector list) {
        synchronized (list) {
            int lastIndex = list.size() - 1;
            return list.get(lastIndex);
        }
    }

    public static void deleteLast(Vector list) {
        synchronized (list) {
            int lastIndex = list.size() - 1;
            list.remove(lastIndex);
        }
    }
}
```

## Iteration and Synchronized Collections

The problem that there may be change in the vector between the call to `size()` and the call to `get()`, can also occur during iteration.

```
for (int i=0; i < vector.size(); i++) {
    doSomething(vector.get(i) );
}
```

We can solve this by using

```
synchronized(vector) {
    for (int i=0; i < vector.size(); i++) {
        doSomething(vector.get(i) );
    }
}
```

BUT, this will lock the vector and for large vectors will pose liveness problems.

## Java Collections and ConcurrentModificationException

The newer Java Collections provide Iterators to iterate over their collections and partially address this problem by throwing a ConcurrentModificationException when the JVM detects a change to the underlying collection while iteration is occurring.

```
List<Widget> widgetList =
    Collections.synchronizedList(new ArrayList<Widget>());

// may throw ConcurrentModificationException
for (Widget w : widgetList)
    doSomething(w);
```

But again, if the collection is large, locking the collection as in:

```
synchronized(w) {
    for (Widget w : widgetList)
        doSomething(w);
}
```

may severely affect performance.

## Alternative: copy the collection

- An alternative to locking is to clone the collection and then iterate over the clone (but the collection must be locked during the cloning process).
- This will always avoid a ConcurrentModificationException but there is the issue that the iterator will see a stale version of the collection.

## Hidden Iterators

```
package net.jcip.examples;

import java.util.*;

import net.jcip.annotations.*;

/**
 * HiddenIterator
 * Iteration hidden within string concatenation
 *
 */
public class HiddenIterator {
    @GuardedBy("this") private final Set<Integer> set =
        new HashSet<Integer>();

    public synchronized void add(Integer i) {
        set.add(i);
    }

    public synchronized void remove(Integer i) {
        set.remove(i);
    }

    public void addTenThings() {
        Random r = new Random();
        for (int i = 0; i < 10; i++)
            add(r.nextInt());
        System.out.println("DEBUG: added ten elements to " + set);
    }
}
```

The problem here is that when `set.toString()` is called, the JVM will iterate over each entry in the set. If the set is modified during this time, a `ConcurrentModificationException` will be thrown.

## Concurrent Collections

Java 5.0 provides concurrent collections which improve on the synchronized collections with a technique called lock striping which provides a greater degree of concurrency. They also support methods that previously would require client-side locking, such as:

<a href="#">V</a>	<a href="#">putIfAbsent</a> ( <a href="#">K</a> key, <a href="#">V</a> value) If the specified key is not already associated with a value, associate it with the given value.
-------------------	--

If the specified key is not already associated with a value, associate it with the given value. This is equivalent to

```
if (!map.containsKey(key))
    return map.put(key, value);
else
    return map.get(key);
```

Except that the action is performed atomically.

### ConcurrentHashMap

Retrieval operations (including `get`) generally do not block, so may overlap with update operations (including `put` and `remove`). Retrievals reflect the results of the most recently *completed* update operations holding upon their onset. For aggregate operations such as `putAll` and `clear`, concurrent retrievals may reflect insertion or removal of only some entries. Similarly, Iterators and Enumerations return elements reflecting the state of the hash table at some point at or since the creation of the iterator/enumeration. They do *not* throw [ConcurrentModificationException](#). However, iterators are designed to be used by only one thread at a time.

The allowed concurrency among update operations is guided by the optional `concurrencyLevel` constructor argument (default 16), which is used as a hint for internal sizing. The table is internally partitioned to try to permit the indicated number of concurrent updates without contention. Because placement in hash tables is essentially random, the actual concurrency will vary. Ideally, you should choose a value to accommodate as many threads as will ever concurrently modify the table. Using a significantly higher value than you need can waste space and time, and a

significantly lower value can lead to thread contention. But overestimates and underestimates within an order of magnitude do not usually have much noticeable impact. A value of one is appropriate when it is known that only one thread will modify and all others will only read. Also, resizing this or any other kind of hash table is a relatively slow operation, so, when possible, it is a good idea to provide estimates of expected table sizes in constructors.

```
public interface ConcurrentMap<K, V>
    extends Map<K, V>
```

A [Map](#) providing additional atomic `putIfAbsent`, `remove`, and `replace` methods.

This interface is a member of the [Java Collections Framework](#).

**Since:**

1.5

## Nested Class Summary

Nested classes/interfaces inherited from interface `java.util.Map`

[Map.Entry](#)<K, V>

## Method Summary

<a href="#">V</a>	<a href="#">putIfAbsent</a> ( <a href="#">K</a> key, <a href="#">V</a> value) If the specified key is not already associated with a value, associate it with the given value.
boolean	<a href="#">remove</a> ( <a href="#">Object</a> key, <a href="#">Object</a> value) Remove entry for key only if currently mapped to given value.
<a href="#">V</a>	<a href="#">replace</a> ( <a href="#">K</a> key, <a href="#">V</a> value) Replace entry for key only if currently mapped to some value.
boolean	<a href="#">replace</a> ( <a href="#">K</a> key, <a href="#">V</a> oldValue, <a href="#">V</a> newValue) Replace entry for key only if currently mapped to given value.

## ConcurrentHashMap

### Constructor Summary

[ConcurrentHashMap](#)()

Creates a new, empty map with a default initial capacity, load factor, and concurrencyLevel.

[ConcurrentHashMap](#)(int initialCapacity)

Creates a new, empty map with the specified initial capacity, and with default load factor and concurrencyLevel.

[ConcurrentHashMap](#)(int initialCapacity, float loadFactor, int concurrencyLevel)

Creates a new, empty map with the specified initial capacity, load factor, and concurrency level.

[ConcurrentHashMap](#)(Map<? extends [K](#),? extends [V](#)> t)

Creates a new map with the same mappings as the given map.

### Method Summary

void	<a href="#">clear</a> () Removes all mappings from this map.
boolean	<a href="#">contains</a> ( <a href="#">Object</a> value) Legacy method testing if some key maps into the specified value in this table.
boolean	<a href="#">containsKey</a> ( <a href="#">Object</a> key) Tests if the specified object is a key in this table.
boolean	<a href="#">containsValue</a> ( <a href="#">Object</a> value) Returns true if this map maps one or more keys to the specified value.
<a href="#">Enumeration</a> < <a href="#">V</a> >	<a href="#">elements</a> () Returns an enumeration of the values in this table.
<a href="#">Set</a> < <a href="#">Map.Entry</a> < <a href="#">K</a> , <a href="#">V</a> >>	<a href="#">entrySet</a> () Returns a collection view of the mappings contained in this map.
<a href="#">V</a>	<a href="#">get</a> ( <a href="#">Object</a> key) Returns the value to which the specified key is mapped in this table.
boolean	<a href="#">isEmpty</a> ()

	Returns <code>true</code> if this map contains no key-value mappings.
<a href="#">Enumeration</a> < <a href="#">K</a> >	<a href="#">keys</a> () Returns an enumeration of the keys in this table.
<a href="#">Set</a> < <a href="#">K</a> >	<a href="#">keySet</a> () Returns a set view of the keys contained in this map.
<a href="#">V</a>	<a href="#">put</a> ( <a href="#">K</a> key, <a href="#">V</a> value) Maps the specified <code>key</code> to the specified <code>value</code> in this table.
void	<a href="#">putAll</a> ( <a href="#">Map</a> <? extends <a href="#">K</a> , ? extends <a href="#">V</a> > t) Copies all of the mappings from the specified map to this one.
<a href="#">V</a>	<a href="#">putIfAbsent</a> ( <a href="#">K</a> key, <a href="#">V</a> value) If the specified key is not already associated with a value, associate it with the given value.
<a href="#">V</a>	<a href="#">remove</a> ( <a href="#">Object</a> key) Removes the key (and its corresponding value) from this table.
boolean	<a href="#">remove</a> ( <a href="#">Object</a> key, <a href="#">Object</a> value) Remove entry for key only if currently mapped to given value.
<a href="#">V</a>	<a href="#">replace</a> ( <a href="#">K</a> key, <a href="#">V</a> value) Replace entry for key only if currently mapped to some value.
boolean	<a href="#">replace</a> ( <a href="#">K</a> key, <a href="#">V</a> oldValue, <a href="#">V</a> newValue) Replace entry for key only if currently mapped to given value.
int	<a href="#">size</a> () Returns the number of key-value mappings in this map.
<a href="#">Collection</a> < <a href="#">V</a> >	<a href="#">values</a> () Returns a collection view of the values contained in this map.

## Blocking and Interruptible Methods (section 5.4)

Interrupting a thread means stopping what it is doing before it has completed its task, effectively aborting its current operation. Whether the thread dies, waits for new tasks, or goes on to the next step depends on the application.

Although it may seem simple at first, you must take some precautions in order to achieve the desired result. There are some caveats you must be aware of as well.

First of all, forget the *Thread.stop* method. Although it indeed stops a running thread, the method is unsafe and was [deprecated](#), which means it may not be available in future versions of the Java.

Another method that can be confusing for the unadvised is *Thread.interrupt*. Despite what its name may imply, the method does not interrupt a running thread (more on this later), as [Listing A](#) demonstrates. It creates a thread and tries to stop it using *Thread.interrupt*. The calls to *Thread.sleep()* give plenty of time for the thread initialization and termination. The thread itself does not do anything useful.

#### Listing A: Stopping a thread with Thread.interrupt()

```
class Example1 extends Thread {
    public static void main( String args[] ) throws Exception {
        Example1 thread = new Example1();
        System.out.println( "Starting thread..." );
        thread.start();
        Thread.sleep( 3000 );
        System.out.println( "Interrupting thread..." );
        thread.interrupt();
        Thread.sleep( 3000 );
        System.out.println( "Stopping application..." );
        System.exit( 0 );
    }

    public void run() {
        while ( true ) {
            System.out.println( "Thread is running..." );
            long time = System.currentTimeMillis();
            while ( System.currentTimeMillis()-time < 1000 ) {
            }
        }
    }
}
```

If you run the code in Listing A, you should see something like this on your console:

```
Starting thread...
Thread is running...
Thread is running...
Thread is running...
Interrupting thread...
Thread is running...
Thread is running...
Thread is running...
Stopping application...
```

Even after *Thread.interrupt()* is called, the thread continues to run for a while.

Really interrupting a thread

The best, recommended way to interrupt a thread is to use a shared

variable to signal that it must stop what it is doing. The thread must check the variable periodically, especially during lengthy operations, and terminate its task in an orderly manner. [Listing B](#) demonstrates this technique.

### Listing B: Signaling a thread to stop

```
class Example2 extends Thread {
    volatile boolean stop = false;

    public static void main( String args[] ) throws Exception {
        Example2 thread = new Example2();
        System.out.println( "Starting thread..." );
        thread.start();
        Thread.sleep( 3000 );
        System.out.println( "Asking thread to stop..." );
        thread.stop = true;
        Thread.sleep( 3000 );
        System.out.println( "Stopping application..." );
        System.exit( 0 );
    }

    public void run() {
        while ( !stop ) {
            System.out.println( "Thread is running..." );
            long time = System.currentTimeMillis();
            while ( (System.currentTimeMillis()-time < 1000) && (!stop) ) {
            }
        }
        System.out.println( "Thread exiting under request..." );
    }
}
```

Running the code in Listing B will generate output like this (notice how the thread exits in an orderly fashion):

```
Starting thread...
Thread is running...
Thread is running...
Thread is running...
Asking thread to stop...
Thread exiting under request...
Stopping application...
```

Although this method requires some coding, it is not difficult to implement and give the thread the opportunity to do any cleanup needed, which is an absolute requirement for any multithreaded application. Just be sure to declare the shared variable as *volatile* or enclose any access to it into *synchronized* blocks/methods.

So far, so good! But what happens if the thread is blocked waiting for some event? Of course, if the thread is blocked, it can't check the shared variable and can't stop. There are plenty of situations when that may occur, such as calling *Object.wait()*, *ServerSocket.accept()*, and *DatagramSocket.receive()*, to name a few.

They all can block the thread forever. Even if a timeout is employed, it may not be feasible or desirable to wait until the timeout expires, so a mechanism to prematurely exit the blocked state must be used.

Unfortunately there is no such mechanism that works for all cases, but the particular technique to use depends on each situation. In the following sections, I'll give solutions for the most common cases.

Interrupting a thread with *Thread.interrupt()*

As demonstrated in Listing A, the method *Thread.interrupt()* does not interrupt a running thread. What the method actually does is to throw an interrupt if the thread is blocked, so that it exits the blocked state. More precisely, if the thread is blocked at one of the methods *Object.wait*, *Thread.join*, or *Thread.sleep*, it receives an *InterruptedException*, thus terminating the blocking method prematurely.

So, if a thread blocks in one of the aforementioned methods, the correct way to stop it is to set the shared variable and then call the *interrupt()* method on it (notice that it is important to set the variable first). If the thread is not blocked, calling *interrupt()* will not hurt; otherwise, the thread will get an exception (the thread must be prepared to handle this condition) and escape the blocked state. In either case, eventually the thread will test the shared variable and stop. [Listing C](#) is a simple example that demonstrates this technique.

#### Listing C: Exiting blocked states with *Thread.interrupt()*

---

```
class Example3 extends Thread {
    volatile boolean stop = false;

    public static void main( String args[] ) throws Exception {
        Example3 thread = new Example3();
        System.out.println( "Starting thread..." );
        thread.start();
        Thread.sleep( 3000 );
        System.out.println( "Asking thread to stop..." );
        thread.stop = true;
        thread.interrupt();
        Thread.sleep( 3000 );
        System.out.println( "Stopping application..." );
        System.exit( 0 );
    }

    public void run() {
```

```

while ( !stop ) {
    System.out.println( "Thread running..." );
    try {
        Thread.sleep( 1000 );
    } catch ( InterruptedException e ) {
        System.out.println( "Thread interrupted..." );
    }
}
System.out.println( "Thread exiting under request..." );
}
}

```

As soon as *Thread.interrupt()* is called in Listing C, the thread gets an exception so that it escapes the blocked state and determines that it should stop. Running this code produces output like this:

```

Starting thread...
Thread running...
Thread running...
Thread running...
Thread running...
Asking thread to stop...
Thread interrupted...
Thread exiting under request...
Stopping application...

```