Threads and Java Collections

Thread support in Java's built-in collections varies from class to class. In general we can place Java collections into three groups:

- Collections with No Thread Support
- Synchronized Collections
- Concurrent Collections

Let's take a close look at these groups.

Collections with No Thread Support

Unless a collection is explicitly documented as supporting threads, assume that no thread support is provided. This means that its internal data is not designed to handle changes from multiple threads. Using these collections in a multi-threaded program can easily result in corrupted data. Assume we have the following data structure defined:

```
static List<String> testList = new ArrayList<String>();
```

Consider the following code:

```
testList.add("Go Cardinal");
```

What can go wrong executing add in a multithreaded environment? We know adding an item into an `ArrayList` actually involves several steps. The computer must determine how many items are in the list, it must place the item into the array, and then it must increment the number of items in the list. In a multithreaded environment, any of these steps could end up interleaved with instructions from another thread, resulting in corrupted data. For example our thread might retrieve the number of items in the list, and then get swapped out for another thread which adds an item into the list. When our thread becomes active again it has the wrong number of items in the list, it proceeds to overwrite the data added by the other thread.

Java collections without thread support may be used in a multithreaded environment, but extreme caution should be exercised. A collection which is only accessed from a single thread, for example, should not cause any problems. You can also write your own locking code to coordinate access to the collection from multiple threads.

Synchronized Collections

Java provides a number of Synchronized Collections. These are collections in which each method is synchronized. As you may recall this means that any time a thread is running a

method on an object, other threads attempting to perform an operation on the object will block until the first thread has completed the method.

Going back to our previous example if we have a Synchronized List, then:

```
testList.add("Go Cardinal");
```

causes no problems. If one thread is adding an item to the list, no other operation may be performed on the list, until the add operation has concluded.

Available Synchronized Collections

There are two sets of Synchronized Collections. First there are two older classes—`Vector` and `Hashtable`—which are synchronized. Please note this is the `Hashtable` class only and not the newer `HashMap` or `TreeMap` classes. In addition, the `Collections` class provides access to a large number of Synchronized lists via a set of static methods. Note this is the `Collections` class (with an 's') not the `Collection` interface (without the 's'). Synchronized collections provided by `Collections` include a general collection, a list, a map, a set, a sorted map, and a sorted set. The `Collections` class static methods convert existing collections, and can also be used to create new synchronized collections. Here's an example of code which starts with an un-synchronized list and then converts it to a synchronized list.

```
List<String> origList = new ArrayList<String>();

origList.add("Leland");
origList.add("Stanford");
origList.add("University");

List<String> synchList = Collections.synchronizedList(origList);
```

Here's how you would create a new empty synchronized list:

```
List<String> synchList = Collections.synchronizedList(
                                    new ArrayList<String>());
```

The other types of synchronized collections work exactly the same. All these synchronized collections classes are anonymous classes, and you can't create them directly, only indirectly through `Collections`'s conversion methods.

Synchronized Collections Not Thread-Safe

A synchronized collection doesn't have to worry about getting corrupted internally. However, having a synchronized collection doesn't always mean that your code is going to work the way you want it to. Consider the following code snippet run on a synchronized list:

```
if(!testList.isEmpty())
        testList.remove(0);
```

`isEmpty` and `remove` are both treated as atomic operations (that is to say their internal operations cannot be interrupted or interleaved with other operations on our test list). However, that doesn't mean that the combination of the two synchronized methods can't end up interleaved with other actions. Suppose the list has only one item in it. Our thread could verify that the list is not empty, and then before it removes the item, another thread removes it first, resulting in an error when the original thread tries to remove the now non-existent item.

Some authors refer to the Java Synchronized Collections as *conditionally thread-safe*. Individual operations on these collections are safe, but compound actions on them still require the programmer using them to take extra precautions. Because the synchronized collections use locks on the collection objects themselves, we could fix our code like this:

```
synchronized(testList) {
   if(!testList.isEmpty())
      testList.remove(0);
}
```

Now no other threads can execute methods on the `testList` between the `isEmpty` call and the `remove` call.[1]

Iterators and Synchronized Lists

As with standard collections, the synchronized collections support iterators. However, if one thread is using an iterator for a collection and another thread modifies the collection, the iterator will throw a `ConcurrentModificationException` when the iterator is used to retrieve the next item. You can get around this by locking down the collection as we did above with our `isEmpty-remove` example. However, keep in mind that while the collection is locked, no other threads can access the collection. If you're iteration is taking a long time, this could have a severe impact on efficiency.

Concurrent Collections

Concurrent Collections add much more sophisticated support for concurrency. These collections can be found in `java.util.concurrent`. Java 1.5 adds a number of Queues supporting concurrency along with `ConcurrentHashMap`, `CopyOnWriteArrayList`, and `CopyOnWriteArraySet`. Java 1.6 adds an additional queue type and also adds `ConcurrentSkipListMap` and `ConcurrentSkipListSet`. Mac users (and those

[1] Note that this code only works because a thread can acquire a lock on an item more than once. It's okay for our thread to grab testList's intrinsic lock once for the explicit synchronized(testList) and then again when calling isEmpty.

building applications for Mac users) should be aware that Java 1.6 is only widely available on Mac OS X 10.6.

CopyOnWriteArrayList and CopyOnWriteArraySet

As their name implies these collections create a new copy of themselves when operations are performed which changes the array. Iterators act on copies of the array when the iterator was created. Therefore they do not need to worry about changes made during their iteration. Any changes made to the list or set after the iterator is created will not be visible to the iterator.

Because duplicating the underlying array is expensive, these classes should only be used in situations in which iteration through the list or set happens frequently, but changes to the list or set happen only rarely.

ConcurrentHashMap

This class is much more efficient than its synchronized counterpart. Retrieval operations do not block and can therefore occur simultaneously. Some write operations can take place simultaneously (the underlying hash table is broken down into sections and simultaneous writes can occur if they modify different sections of the table).

This map supports operations found in the `ConcurrentMap` interface in addition to those in ther standard Map interface. These additional operations are atomic versions of compound operations commonly performed on maps. For example the `putIfAbsent` operation can be carried out on a standard map by performing a `containsKey` call followed by a `put` call. However, in a concurrent environment the map could change between the `containsKey` call and the `put` call, resulting in the overwriting of data. With the new `ConcurrentMap` `putIfAbsent` call, both checking for a key and placing data associated with the key happen in a single atomic action that cannot be interrupted.

Iterators on `ConcurrentHashMap` are better designed to support concurrency and will not throw `ConcurrentModificationException`. However, they may or may not reflect modifications in the list since the iterator was created.

BlockingQueue

Blocking Queues are particular interesting data structures as they can be used both to store information and coordinate actions between threads. You'll get an opportunity to study them on the upcoming Homework #4. They fit very naturally in producer consumer situations.

Normally if you have one or more threads producing data and one or more threads consuming information you will need communications mechanisms such as semaphores to coordinate between the two sets of threads. The semaphores will track when the queue is empty or when it is full. A blocking thread can handle much of the communication overhead.

The standard `put` operation used to place items into the queue is modified to block if the queue is full. The `take` operation removes an item from the queue if one is available, otherwise it blocks until an item is available.

`BlockingQueue` itself is an interface. There are a number of classes which support the `BlockingQueue` behavior. `LinkedBlockingQueue` and `ArrayBlockingQueue` provide standard FIFO behavior. There is also a `PriorityBlockingQueue`. The `SychronousBlockingQueue` is particularly interesting in that it doesn't actually store data, it's designed to allow threads to communicate using `put` and `take` without actually involving any real data storage. The storage size is zero, threads calling `put` pass data directly to threads calling `take`.