# CMSC 433
# Programming Language Technologies and Paradigms

Spring 2013

**Collections, Thread Safety, Blocking**

## Java provides many collections…

Ones that are not thread-safe like…

- Lists: LinkedList, ArrayList, PriorityQueue, Stack, …
- Maps: HashMap, TreeMap, …
- Sets: HashSet, TreeSet, …

We can use our own guarding mechanisms to have our multithreaded programs safely interact with this.

The Collections package in Java can give us some more options to wrap existing collections with some thread safety, but only some… Any compound actions still need to be synchronized by YOU (such as iterating over the collection or doing something like deleting the last item in a list).

## `java.util.concurrent`

Like other things within **`java.util`** the idea is to provide a useful and well tested library of classes.

In this case, to provide things to be used by multithreaded applications.

The collections here support multiple operations on the collection to occur with some overlap (unlike collections built using the Monitor Pattern) so there can be more concurrency, though the way you use them is a bit different…

## `ConcurrentHashMap<K,V>`

A more concurrency-friendly implementation of the **`HashMap`**.

Locking is done on clusters of hash buckets. The default locking scheme is that there will be 16 locks used, each on 1/16$^{th}$ of the buckets. You can select a different level of concurrency when creating a new **`ConcurrentHashMap`** object.

Some operations have new subtleties. If you use an **`iterator`**, they'll "tolerate" if modifications are made, but they'll basically perform a traversal based on the state when the iterator was made which means that you aren't necessarily seeing the results of modifications since then.

There is no locking method that works on the entire object. This means, for example, things like **`.size()`** and **`.isEmpty()`** don't account for operations that might be in progress.

## ConcurrentHashMap<K, V> and compound actions

Also related to the issue of not being able to lock an entire **ConcurrentHashMap** object, is the question of how to perform compound actions like "check to see if a key is mapped to, then do XYZ" since if the check was done apart from the action, the state might have changed.

For this reason, the **ConcurrentHashMap** class provides some compound methods:

```
V putIfAbsent(K key, V value) If the specified key is not
  already associated with a value, associate it with the given value.

boolean remove(Object key, Object value) Remove entry for
  key only if currently mapped to given value.

boolean replace(K key, V oldValue, V newValue) Replace
  entry for key only if currently mapped to given value.

V replace(K key, V value) Replace entry for key only if
  currently mapped to some value.
```

## Project 3

In Project 2, you might have used a **HashMap**. If you did, did you ever have to synchronize on it? If so, you might be able to replace it with a **ConcurrentHashMap** now and remove the synchronized blocks!

You might look back at Project 1 and think about the way the **HashMap** objects were used there. Were you just locking the structures to protect them from concurrent access or were you using them to guard against other things as well?

## `CopyOnWriteArrayList<E>`

The approach here is very different from the **`ConcurrentHashMap`** class. Rather than having several locks to allow some level of concurrent access, this list allows unrestricted reading and creates a new array behind the scenes if a write takes place.

Writes can not take place concurrently (a **`ReentrantLock`** is used to guard writes). A new backing array is created, everything is copied from the current array into this new one and then the new array is published as the backing array.

If (for example) an **`iterator`** is created, it will iterate through the "backing array" of this list and essentially be oblivious to any writes that happen to the replacement backing array which a write creates.

## When to use a `CopyOnWriteArrayList<E>`?

If you plan to read from (and iterate over) your list ***FAR MORE OFTEN*** than you plan to make updates to it.

You can pass the constructor an array to initially create the **`CopyOnWriteArrayList`** object. Adds can also do mass-additions in a single "write" operation.

The larger a **`CopyOnWriteArrayList`** becomes, the worse performance on any add becomes!

## Concurrent Queues

The idea with concurrent queues is essentially to implement various types of queues in an automatically thread-safe manner.

For example, there is a **`ConcurrentLinkedQueue<E>`** class that implements everything you'd expect from a regular queue, but in a thread-safe way. It handles all of the guarding of the structure itself, so you can simply **`add`**, **`poll`**, **`remove`**, etc. without doing any guarding yourself.

## Blocking Queues

There are also some blocking queues that can be useful. The idea with a blocking queue is two-fold;

- An attempt to **`take`** something off an empty queue will block until the queue is non-empty (notice this is a different method than **`poll`**). While a regular call to **`poll`** will return **`null`** if the queue is empty, calling **`poll`** and passing in a time limit will only return **`null`** if the queue is empty that long.
- You can specify a capacity and then you can use **`put`** to add an element, and it will block if the queue is full. You can also use **`offer`** (with or without a time limit) which returns a boolean based on whether the element was added. If you use the regular **`add`** then if the queue is full an exception will be thrown.

Specific blocking queues are the **`LinkedBlockingQueue<E>`** and **`PriorityBlockingQueue<E>`** (and others are provided in the **`java.util.concurrent`** library).

## Other Blocking Queue-like Collections

Other queue-like structures are used from time to time. In another course you may have seen something called a **deque** (pronounced "deck") which is a "double-ended queue" and allows elements to be added and removed at both ends.

In the `java.util.concurrent` library there is a collection called `LinkedBlockingDeque` that can be useful (we will actually talk about deques later in the semester when we cover Fork-Join).

There is also something called a `SynchronousQueue` whose capacity is basically set to zero which means that puts/takes end up becoming synchronized.

## Project 3

In Project 2, how did you handle lists of things like orders? Did you use wait/notify in relation to certain lists? How could a blocking collection help guard some of your lists? How could it remove the need to manually use wait and notify. Remember that some actions are blocking while others are still not blocking…

## `InterruptedException`

If a thread is sent an interrupt while running normally, an internal flag is set. Then, if within that thread any of several methods are called that check for this flag, that method will throw an **`InterruptedException`** which would then be caught or propagated.

The flag indicates to the thread that some other thread is trying to stop them. It is preferred over a call to **`.stop()`** since in general it gives you a chance to clean up and exit (possibly propagating the exception or creating a new, different type of exception).

If the thread was blocked on something then the flag isn't set but the exception is raised. The thread can go ahead and set the status flag by calling **`Thread.currentThread().interrupt();`** as part of the exception handling. This can be useful if another thread uses the **`isInterrupted()`** method to ask about the thread's state.

Of course, they can just ignore that request by having the "catch" basically brush it off…

## Producers/Consumer with a BlockingQueue: `Producer`

```
public class PC4_Producer extends Thread {
   private final BlockingQueue<Integer> sharedQueue;
   private static final Random rnd = new Random();
   public static final int MAX = 5000;

   private PC4_Producer(BlockingQueue<Integer> queueIn) {
      sharedQueue = queueIn;
   }
   public static PC4_Producer getInstance(BlockingQueue<Integer> queueIn) {
      return new PC4_Producer(queueIn);
   }

   public void run() {
      int value;

      while (true) {
         try {
            Thread.sleep(rnd.nextInt(10000)); //Pretend work is happening…
            value = rnd.nextInt(MAX);
            sharedQueue.put(value);

         } catch (InterruptedException e) {Thread.currentThread().interrupt(); return;}
      }
   }
}
```

## Producers/Consumer with a BlockingQueue: `Consumer`

```
public class PC4_Consumer extends Thread {
   private final BlockingQueue<Integer> sharedQueue;
   private static final int NUM_TO_USE = 5000;
   public long sum = 0;
   public long count = 0;

   private PC4_Consumer(BlockingQueue<Integer> queueIn) {
      sharedQueue = queueIn;
   }
   public static PC4_Consumer getInstance(BlockingQueue<Integer> queueIn) {
      return new PC4_Consumer(queueIn);
   }

   public void run() {
      int newValue;

      for (int i=0; i<NUM_TO_USE; i++) {
         try {
            newValue = sharedQueue.take();
            sum+=newValue;
            count++;

         } catch (InterruptedException e) {Thread.currentThread().interrupt(); return;}
      }
   }
}
```

## Producers/Consumer with a BlockingQueue: `Tester`

```
int NUM_OF_PRODUCERS = 1000;
BlockingQueue<Integer> dataStream = new LinkedBlockingQueue<Integer>();
PC4_Consumer consumer = PC4_Consumer.getInstance(dataStream);

PC4_Producer[] producers = new PC4_Producer[NUM_OF_PRODUCERS];
for (int i=0; i<NUM_OF_PRODUCERS; i++) {
   producers[i] = PC4_Producer.getInstance(dataStream);
   producers[i].start();
}
System.out.println("Producers started...");

consumer.start();
System.out.println("Consumer started...");
Thread.sleep(5000);
consumer.interrupt();
consumer.join();

System.out.println("Half of Max: " + PC4_Producer.MAX / 2);
System.out.println("Average: " + consumer.sum / consumer.count);

for (int i=0; i<NUM_OF_PRODUCERS; i++) {
   producers[i].interrupt();
}

for (int i=0; i<NUM_OF_PRODUCERS; i++) {
   producers[i].join();
}
```