Concurrency & Collections
Outline

• Immutable collections
• Synchronized collections
• Concurrent collections
• Blocking collections
Immutable Collections

• The **Collections** framework provides factories to create immutable (unmodifiable) collections.
  – **static** `X unmodifiableX(X c)`
  – Where `X` can be `Collection, List, Map, Set, SortedMap, SortedSet`

• Only the *collection*, not the *elements* in it, are protected.
• Underlying collection still can change “under your feet.”

What are the classes of the objects returned by these factories?

Do these interfaces have state modifying methods?

How can immutability be maintained?
Synchronized Collections

• The Collections framework also provides factories to create *synchronized collections*.

  – `static X synchronizedX(X c)`
  – Where `X` can be `Collection`, `List`, `Map`, `Set`, `SortedMap`, `SortedSet`

If we simply wrap synchronized methods around the collection will that be enough, or do we have to impose additional rules? Do we need similar rules for the unmodifiable collections?

```java
List list = Collections.synchronizedList(new ArrayList());
...

synchronized (list) {
  Iterator i = list.iterator();
  while (i.hasNext()) {
    doSomething(i.next());
  }
}
```

What type of problem does this code exhibit? Why? How can it be fixed?
Is there be any sense in wrapping an immutable collection with synchronization?

Is there be any sense in wrapping a synchronized collection with immutability?
Synchronized collections may have performance issues because all access is serialized.

• Issues are independent of whether:
  a. We use a synchronized collection factory or
  b. We do the synchronization ourselves

• The issues may have to do with embedded, complex collection algorithms.

• Concurrent collections provide carefully defined, high performance algorithms with short-lived locks.

If we want to allow non-serialized concurrency, we have to relax some requirements, or somehow allow concurrent access.

Consider a LinkedList. How could we allow concurrent modification of the list (set value, addition, deletion)?

What are the issues with Iterators in the face of concurrent access? How could they be designed to work?

What could we say about the value returned by a size method?
The blocking queue supports a producer-consumer pattern.

Exception generating
- **boolean** `add(E e)` adds to end of queue Exception if no room.
- `E remove()` 1st element with removal Exception if queue empty.
- `E element()` 1st element w/o removal Exception if queue empty.

Non-blocking w/special return value
- **boolean** `offer(E e)` adds to end of queue **false** if no room.
- `E poll()` 1st element with removal **null** if queue is empty.
- `E peek()` 1st element w/o removal **null** if queue is empty.

Blocking
- **void** `put(E e)` adds to end of queue Waits until room.
- `E take()` 1st element with removal Waits if empty.

Timeout
- **boolean** `offer(E e, long t, TimeUnit u)`
- `E poll(long t, TimeUnit u)`

Note: offer & poll with timeout, put, and take can throw **InterruptedException**
Java provides many different types of blocking queues from basic to enhanced.

- **ArrayBlockingQueue**\(<E>\)
- **LinkedBlockingQueue**\(<E>\)
- **PriorityBlockingQueue**\(<E>\)
  - Elements ordered by comparison
- **DelayQueue**\(<E \text{ extends Delayed}>\)
  - Elements ordered by delay; not available until after delay expires
- **SynchronousQueue**\(<E>\)
  - 0 length queue, producer and consumer must exchange data
- **LinkedTransferQueue**\(<E>\)
  - Unbounded, producer can wait for consumer to get data
Interface ConcurrentMap<K, V>

Map<K, V> with atomic

boolean remove(K key, V value)
    Remove key & value iff key maps to value.

boolean replace(K key, V oldValue, V newValue)
    Replace key with newValue iff key is mapped to newValue.

V replace(K key, V value)
    Replace key with value iff key is mapped to something.
    Return previous value (or null if there was no map).

V putIfAbsent(K key, V value)
    Associate key with value if the key is not currently mapped.
    Returns null if the put succeeded, otherwise the currently mapped value.

One implementing class: ConcurrentHashMap<K, V>

Highly optimized for concurrent thread-safe access to the map data structure.
A Sampling of Other Interfaces & Classes

Double Ended Queues (Deques)

Interface BlockingDeque & Class LinkedListBlockingDeque

- `addFirst` `offerFirst` `putFirst` `offerFirst` (with timeout)
- `removeFirst` `pollFirst` `takeFirst` `pollFirst` (with timeout)
- `getFirst` `peekFirst`
- `addLast` `offerLast` `putLast` `offerLast` (with timeout)
- `removeLast` `pollLast` `takeLast` `pollLast` (with timeout)
- `getLast` `peekLast`

Classes

ConcurrentLinkedQueue<E>
- Fine granularity locks
- Low latency

CopyOnWriteArrayList<E>

CopyOnWriteArraySet<E>
- When traversals much more frequent than mutations.
- Snapshot style iteration
Read the javadocs for full information!