Lecture 7
Concurrent Collections
Collections in Java

• Collection objects group together other objects of the same type
  – Lists
  – Sets
  – Maps
  – Queues
  – Etc.
• They permit objects to be stored and processed later
• They support *iteration*: processing of each element in a collection
  – Iterator objects
  – for (e : collection) statements
Sample Collection Classes

- **Set interface**
  - HashSet
  - TreeSet
  - LinkedHashSet

- **List interface**
  - ArrayList
  - LinkedList

- **Map interface**
  - HashMap
  - TreeMap
  - LinkedHashMap

- **Queue interface**
  - LinkedList
  - PriorityQueue
Collections and Thread-Safety

• Previous implementations are not thread-safe
  – Insertion, deletion operations are not synchronized
  – Sharing these objects among threads can lead to erroneous data structures

• But collections are needed in thread programming!
Synchronization and the Collections Class

• The Collections class consists of static methods for processing collections
• It includes factory methods for creating synchronized versions of lists / sets / maps
  – Factory methods take relevant collections as inputs
  – They produce collections as outputs, but with all operations synchronized
• List<Integer> list =
  Collections.synchronizedList(new ArrayList<Integer>());
  – synchronizedList() produces a new list object that contains its argument as private field
  – List methods are "wrapped" inside synchronization code
  – Returned object is thread-safe as a result
Implementing `synchronizedList()`

- Create new class `SynchronizedList<T>`

```java
... class SynchronizedList<T> implements List<T> {

    final List<T> list;

    SynchronizedList<T> (List<T> list) { this.list = list; }

    public int size () {
        synchronized (this) {
            return list.size();
        }
    }

    ...
}
```

- Each method is “wrapped” with synchronization code
- Lock used is lock of wrapping object, as opposed to the wrapping object, or backing list

- Have `synchronizedList()` return an object in `SynchronizedList<T>`!

```java
public static <T> List<T> synchronizedList (List<T> list) {
    return new SynchronizedList<T> (list);
}
```
Thread Safety and Compound Actions

• Thread safety guarantees individual method invocations preserve correctness

• What if threads want to perform operations involving multiple actions?
  – Example: removing last element from a list
    ```java
    public static Object getLast (List<Object> l) {
        int lastIndex = l.size() - 1;
        return (l.get(lastIndex));
    }
    ```
  – This can lead to an `IndexOutOfBoundsException`!
    • Each thread computes `lastIndex` value
    • First thread then removes element at this position
    • Second thread will try, but position is no longer valid
Implementing Compound Actions

• Thread safety does not guarantee that compound actions will complete successfully

• Solution for synchronized collections: \textit{client-side locking}
  – Client locks data structure while compound action is performed
  – This ensures that state of data structure cannot change unexpectedly
  – Corrected \texttt{getLast()}: 
    
    \begin{verbatim}
    public static Object getLast (List<Object> l) {
      synchronized (l) {
        int lastIndex = l.size() - 1;
        return (l.get(lastIndex));
      }
    }
    \end{verbatim}

\begin{flushright}
10/15/2012 \hspace{4cm} ©2012. 2013 University of Maryland
\end{flushright}
Iteration and Synchronized Collections

• Iteration: the ultimate compound action!
  – Iteration processes all elements in a collection
  – Without synchronization:
    • One thread can start an iteration
    • Another can modify the collection while the iteration is underway
    • ConcurrentModificationException can be thrown as a result!
    • Iterators that raise this exception are called fail fast

• Solution: lock whole collection throughout iteration
  Collection<Type> c =
  Collections.synchronizedCollection(myCollection);
  synchronized(c) {
    for (Type e : c)
      foo(e);
  }
  – This keeps state of collection consistent
  – It does reduce concurrent access to collection
Hidden Iteration

• Consider the following
  – list is a List object
  – The following statement is executed without any synchronization
    System.out.println(list);
  – This can cause a ConcurrentModificationException!

• Why?
  – Implementation of toString() for collections uses iteration
  – During construction of string for list, another thread can modify it

• Moral: compound actions, especially iterative ones, require client-side locking
Concurrent Collections

• Issue with synchronized collections: overly reduced concurrent access
  – If a collection is locked during iterative processing, then no other thread can access it
  – Individual operations can also unduly impede concurrent access
    • Hash tables have several buckets
    • Why lock the whole table to access a single bucket?

• Package java.util.concurrent contains implementations of several concurrent collections
  – These relax the “lock the whole data structure” approach of synchronized collections
  – The gain: more concurrency
  – The price to pay: changes to some operations
ConcurrentHashMap

• A concurrent implementation of HashMap
  – Maps keys to values, like HashMap
  – Uses lock striping to improve concurrent access
    • 16 locks used to control access
    • If there are \( k \) buckets, each lock guards \( k/16 \) buckets
    • If two threads are attempting to access buckets guarded by different locks, they can do so concurrently!
    • Locks are also ReadWrite locks (will learn more about this later)

• Benefit of lock striping: more concurrent access, so better performance
• Drawback: no way to lock whole table at user level
  – This means some operations that require access to whole table (e.g. size(), isEmpty()) are approximations
  – This makes compound actions impossible to implement at user level
• Iterators are weakly consistent rather than fail-fast
  – Tolerate concurrent modification
  – Traverse elements as they existed when iterator was constructed
  – May (or may not) reflect modifications to collection after iterator is constructed
ConcurrentHashMap and Built-In Compound Actions

• There is no way to lock entire ConcurrentHashMap
• To address compound-action problem, ConcurrentHashMap implements several of these directly (K is key type, V is value type)
  – V putIfAbsent(K key, V value)
    If key is not mapped to a value in table, map it to value and return null; otherwise, return the value key is mapped to
  – boolean remove(K key, V value)
    Return true if key is mapped to value, in which case also remove mapping; otherwise, return false
  – boolean replace(K key, V oldValue, V newValue)
    Return true if key is mapped to oldValue, in which case also replace oldValue by newValue
  – V replace(K key, V newValue)
    If key is mapped to some value, replace it with newValue and return the old value; otherwise, return null
CopyOnWriteArrayList

- Another concurrent collection, this one intended to support lists
- In synchronized lists, must lock entire list to access a single element or to iterate
  - This is because another thread may modify list during processing
  - Especially for iteration, this greatly reduces concurrency
- For CopyOnWriteArrayList lists, a copy-replace mechanism is used instead
  - No locking needed to read a list
  - When a list is modified, a local copy of the list is created
  - When the update is complete, the modified list is republished
  - When an iterator is created, reference to backing array stored, so iterator sees state of list in effect when iterator was created: no ConcurrentModificationException ever thrown!
- This is a good idea when ...???
  - Most list operations do not involve modification (because no locking needed)
  - Iteration is used frequently
Queues

• Data structures allowing insertion at one end, removal at another
  – FIFO (first-in, first-out) queues: elements stored in order of insertion
  – Priority queues: elements accessed in priority order (next element to be removed is one with highest priority)

• Java Queue interface

```java
interface Queue<E> extends Collection<E> {
    boolean offer(E x); // try to insert, return true if successful, false otherwise
    boolean add (E x) throws IllegalStateException;
        // try to insert, return true if successful, throw exn if not
    E poll(); // retrieve and remove; return null if empty
    E remove() throws NoSuchElementException;
        // retrieve and remove; throw exn if empty
    E peek(); // retrieve, don’t remove, return null if empty
    E element() throws NoSuchElementException;
        // retrieve, don’t remove, throw exn if empty
}
```

• Thread-safe non-blocking implementation: ConcurrentLinkedQueue<E>
The Producer-Consumer Pattern must block on queue ops.

Producers

Consumers

Work Queue
The Producer-Consumer Pattern

• A common multi-threaded paradigm
  – Producer threads generate data to be processed
  – Consumer threads retrieve data and process it

• Issues
  – Producers might go faster than consumers
  – Want any free consumer to pick up a piece of data
  – Want producers to generate data without reference to which consumer will process it

• The Producer-Consumer Pattern
  – Use a blocking queue (work queue) to hold data!
  – Producers insert into queue; block when it is full
  – Producers retrieve data from queue; block when it is empty
Blocking Queues

• Like queues, but add new blocking operations for insertion, removal
  – void put (E e);
    Add element into queue, blocking until there is space
  – E take ();
    Remove and return lead element from queue, blocking until queue is non-empty

• Timed versions of offer, poll also available
  – boolean offer(E e, long timeout, TimeUnit unit)
    Insert element, waiting up to timeout for insertion to succeed
  – E poll (long timeout, TimeUnit unit)
    Retrieve, remove lead element, waiting up to timeout before returning null

• Null elements may not be inserted
  – NullPointerException thrown if this is attempted
  – null only used as a “sentinel value”

• Blocking queues are thread-safe
  – Implementations support multiple users
  – Specialized access pattern for queues is exploited in implementations
Blocking Queue Implementations

- **LinkedBlockingQueue**
  - FIFO
  - May be bounded or unbounded
- **ArrayBlockingQueue**
  - FIFO
  - Bounded
- **PriorityBlockingQueue**
  - Ordered by priority
  - Unbounded
- **SynchronousQueue**
  - Capacity is 0!
  - Net effect: put and take operations between threads are synchronized
  - Sometimes called a *rendezvous channel*