Collections and Concurrency

3 Types of Collections in Java

- Thread-unsafe collections
- Thread-safe collections
  - Synchronized collections
  - Concurrent collections

Thread-unsafe Collections

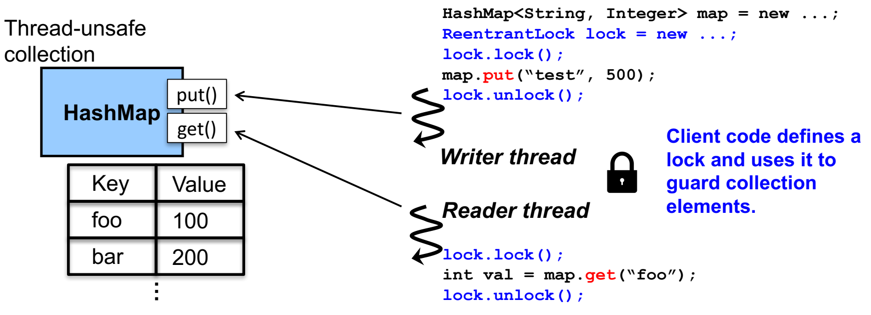
- Many collection classes are NOT thread safe.
  - e.g., `ArrayList`, `LinkedList`, `HashMap`, etc.
  - Their public methods never perform thread synchronization (i.e., locking).
- Look into Java API documentation to see if a collection is thread-safe or not.

Java API Doc on `ArrayList`

- **"Note that this implementation is not synchronized.** If multiple threads access an ArrayList instance concurrently, and at least one of the threads modifies the list structurally, it *must* be synchronized externally. (A structural modification is any operation that adds or deletes one or more elements, or explicitly resizes the backing array; merely setting the value of an element is not a structural modification.)…"

When You Use a Thread-unsafe Collection…

- You must do *client-side locking*; your client code must perform thread synchronization (i.e., locking)
  - to guard collection elements against concurrent accesses.
    - c.f. previous HWs that use thread-unsafe collections such as `ArrayList`, `LinkedList` and `HashMap`.



Key	Value
foo	100
bar	200

```
HashMap<String, Integer> map = new ...;
ReentrantLock lock = new ...;
lock.lock();
map.put(“test”, 500);
lock.unlock();
```

```
lock.lock();
int val = map.get(“foo”);
lock.unlock();
```

Client code defines a lock and uses it to guard collection elements.

- You must do *client-side locking* for compound operations as well as simple public method calls.
- Example compound operations
  - Iteration (element-traversal)
    - Repeatedly get elements one by one until a collection becomes empty
  - Navigation
    - Find/search the next element after a given element
  - Conditional operations (check-then-act)
    - e.g., Check if a `HashMap` has a key-value pair for the key `K`, and if not, add the pair `(K,V)`

Example Compound Operations

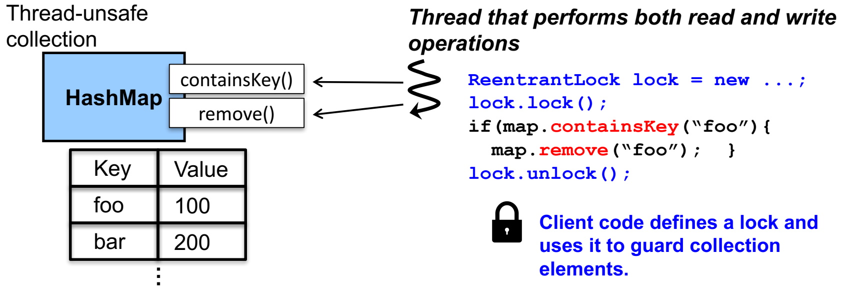
-
  ```
  List<String> list =
    Collections.synchronizedList( new ArrayList<String>());

  Iterator it = list.iterator();
  while( it.hasNext() ){              // Iteration
    doSomething( it.next() ); }
  ```

-
  ```
  for(int i=0; i<list.size(); i+=2){ // Navigation
    doSomething( list.get(i) ); }
  ```

-
  ```
  if( list.size() > 10 )              // Check-then-act
    doSomething( list );
  ```

Client-side Locking for Compound Operations

- An example “check-then-act” compound operation



Key	Value
foo	100
bar	200

Thread that performs both read and write operations

```
ReentrantLock lock = new ...;
lock.lock();
if(map.containsKey(“foo”){
  map.remove(“foo”);   }
lock.unlock();
```

Client code defines a lock and uses it to guard collection elements.

Synchronized Collections

- “Ready-made” thread-safe collections
  - Synchronized classes: `Vector` and `Hashtable`
    - All public methods perform thread synchronization (locking).
      - Only one thread can access the elements of a collection at a time.
        - » e.g., When a thread is in the middle of executing `add()` on a `Vector`, no other threads can call `get()`, `size()`, etc. on that `Vector`.
  - Synchronized wrapper classes for thread-unsafe collections
    - Created by `java.util.Collections.synchronizedXyz()`
      - Factory methods
      - `synchronizedList()`
      - `synchronizedMap()`
      - `synchronizedSet()`

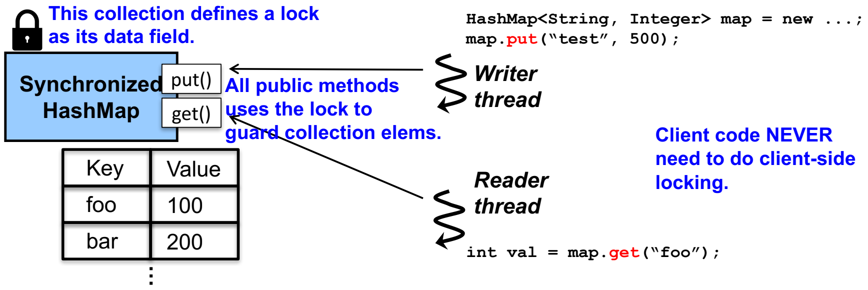
`Vector` and `Hashtable` in Single Threaded Programs

- It makes no sense to use these collections in single-threaded programs in a performance point of view.
  - They perform thread synchronization (locking) even when only one thread runs in a program.
    - Unnecessary performance loss
- Use **`ArrayList`** and **`HashMap`** instead!

Synchronized Wrapper Classes

-
  ```
  List<String> list =
    Collections.synchronizedList( new ArrayList<String>() );
  ```
- `list`: an instance of a synchronized wrapper class for **`ArrayList`**.
  - `list.getClass()` → `java.util.Collections$SynchronizedRandomAccessList`
  - The wrapper class offers “synchronized” (or thread-safe) versions of `ArrayList`’s public methods.
    - `add()`, `get()`, `remove()`, etc.

- All public methods are thread-safe in all synchronized collection classes.
- No client-side locking is necessary when client code makes a simple public method call.



This collection defines a lock as its data field.

```
HashMap<String, Integer> map = new ...;
map.put(“test”, 500);
```

All public methods uses the lock to guard collection elems.

Key	Value
foo	100
bar	200
⋮	

Client code NEVER need to do client-side locking.

```
int val = map.get(“foo”);
```

- Need *client-side locking* in *compound operations* on a synchronized collection
  - Iteration (element-traversal)
    - Repeatedly get elements one by one until a collection becomes empty
  - Navigation
    - Find/search the next element after a given element
  - Conditional operations (check-then-act)
    - e.g., Check if a **`HashMap`** has a key-value pair for the key **`K`**, and if not, add the pair **`(K,V)`**

Potential Problems in Compound Actions

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

  Iterator it = list.iterator();
  while( it.hasNext() )         // Iteration
    doSomething( it.next() );
  ```

-
  ```
  if( list.size() > 10 )       // Check-then-act
    doSomething( list );
  ```

Race conditions can occur here.

- Race conditions
- **`ConcurrentModificaionException`**
  - Raised if a writer thread tries to add/remove elements before a reader thread completes a traversal on the entire set of elements.

Client-side Locking for Compound Operations

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

-
  ```
  synchronized(list){
    Iterator it = list.iterator();
    while( it.hasNext() )         // nested locking
        doSomething( it.next() );   // nested locking
  }
  ```

-
  ```
  synchronized(list){
    if( list.size() > 10 )         // nested locking
        doSomething( ... );
  }
  ```
- **`synchronized(list)`**: acquires the lock that the **`list`** owns/uses for thread synchronization in its public methods.

-
  ```
  ReentrantLock lock = new ReentrantLock();
  lock.lock();                    // acquires lock
  Iterator it = list.iterator();
  while( it.hasNext() )           // acquires the lock that the
                                  // list owns.
      doSomething( it.next() );   // nested locking
  lock.unlock();
  ```
- **`lock`** is different from the lock that the **`list`** owns/uses for thread synch in its public methods.
  - Must make sure to use **`lock`** consistently in all client code of **`list`**.
- A thread acquires 2 locks.

Performance Implication on Client-side Locking

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

-
  ```
  synchronized(list){
    Iterator it = list.iterator();
    while( it.hasNext() )              // nested locking
        doSomething( it.next() );      // nested locking
  }
  ```

-
  ```
  ReentrantLock lock = new ReentrantLock();
  lock.lock();                        // acquires lock
  Iterator it = list.iterator();
  while( it.hasNext() )               // acquires the lock that the
                                      // list owns.
      doSomething( it.next() );       // nested locking
  lock.unlock();
  ```

- Performance penalty due to 2 lock acquisitions
  - Whether it is nested locking or not.

-
  ```
  List<String> list = new ArrayList<String>();
  ...
  ReentrantLock lock = new ReentrantLock();
  lock.lock();
  Iterator it = list.iterator();
  while( it.hasNext() )
    doSomething( it.next() );
  lock.lock();
  ```

- Performs client-side locking for **`ArrayList`**, which is NOT thread-safe.
  - The client code is thread-safe although **`hasNext()`** and **`next()`** are not.

- More efficient than the previous client code.
  - No 2 lock acquisitions (only 1 lock acquisition)
  - Use a read-write lock if you have many "reader" threads.

Summary: Thread un-safe Collections v.s. Synchronized Collections

- Thread un-safe collections
  - Client code always must perform locking (thread synchronization).
    - For both simple public method calls and compound operations
    - Client-side locking always requires 1 lock acquisition.

- Synchronized collections
  - Client code does NOT have to perform locking for simple public method calls.
    - This client-side locking require 1 lock acquisition.

  - However, it needs client-side locking for compound operations.
    - This client-side locking requires 2 lock acquisitions.

Concurrent Collections

- "Ready-made" thread-safe collections
  - Introduced in Java 5 (2004) and enhanced in subsequent versions
    - Queue
      - **`ConcurrentLinkedQueue`** (since Java 5)
      - **`ConcurrentLinkedDeque`** (since Java 7)
      - **`ArrayBlockingQueue`** (since Java 5)
      - **`LinkedBlockingQueue`** (since Java 5)
      - **`DelayQueue`** (since Java 5)
      - **`PrioirtyBlockingQueue`** (since Java 5)
      - **`LinkedBlockingDeque`** (since Java 6)
      - **`LinkedTransferQueue`** (since Java 7)
    - Map
      - **`ConcurrentHashMap`** (since Java 5)
      - **`ConcurrentSkipListMap`** (since Java 6)
    - Set
      - **`ConcurrentSkipListSet`** (since Java 6)

  - `java.util.concurrent.CopyOnWriteXyz` classes
    - **`CopyOnWriteArrayList`**
    - **`CopyOnWriteArraySet`**

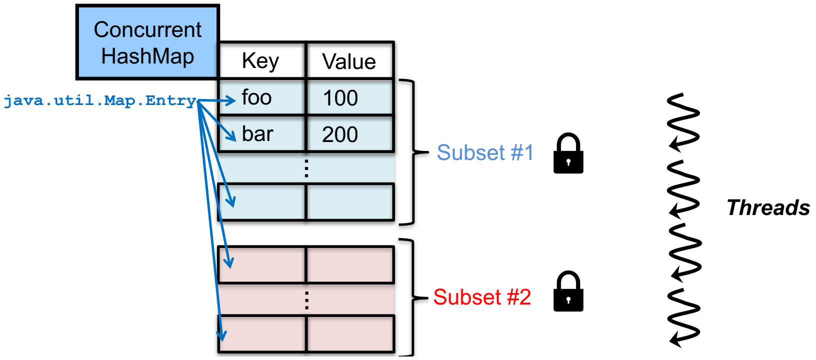
- Indented to replace synchronized collections.
  - e.g., `ConcurrentHashMap` is indented to replace `SynchronizedMap`.
- Make public methods thread-safe.
  - e.g., `get()` and `put()` in `ConcurrentHashMap`
  - Client code does NOT have to perform locking for simple public method calls.
- Implement public methods in an efficient manner.
  - Perform lock stripping
- Provide thread-safe methods to perform compound operations
  - Client code does not have to perform locking for many compound operations.

What is `ConcurrentHashMap`?

- Provides thread-safe public methods.
  - e.g., `get()` and `put()`
  - Client code does NOT have to perform locking for simple public method calls.
- A replacement for synchronized hash-based `Map` implementations (e.g., `Hashtable` and synchronized `HashMap`).
- Implement public methods in an efficient manner.
  - Performs *fine-grained* locking, called lock stripping
    - Compared to *coarse-grained* locking (i.e., lock-per-collection) in synchronized hash-based Map implementations.

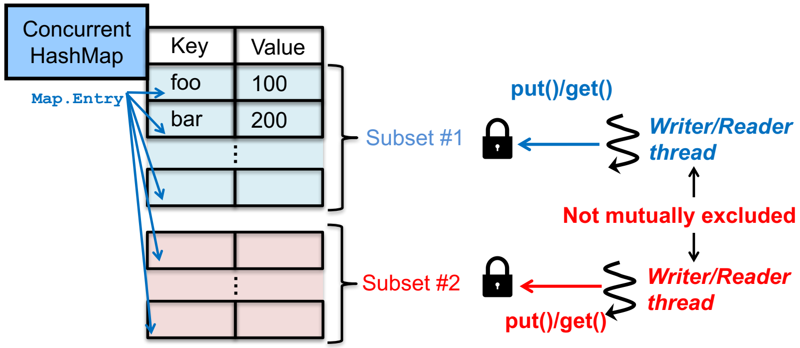
Lock Stripping

- `ConcurrentHashMap` uses *multiple* locks to guard a table (i.e., key-value pairs).
  - 16 locks by default
    - Configurable with the "concurrencyLevel" parameter in a constructor.
  - Each lock is associated with a subset of the table.



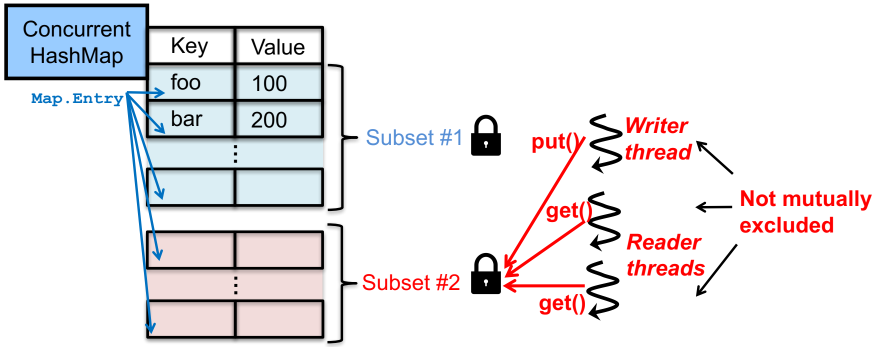
Key	Value
foo	100
bar	200

- Threads are not synchronized (i.e., not mutually excluded) with each other
  - as far as they access different subsets of the table with different locks.

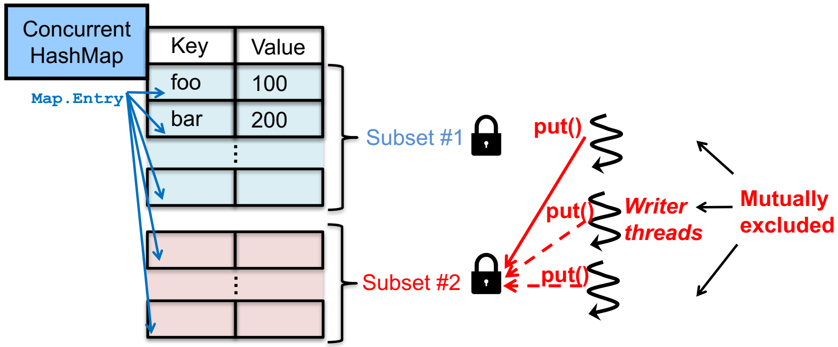


Key	Value
foo	100
bar	200

- To access a subset of the table,
  - Reader threads
    - are NOT synchronized (NOT mutually excluded) with each other.
      - c.f. read-write lock
    - are NOT synchronized (NOT mutually excluded) with writer threads.
      - c.f. inner class Node<K, V>

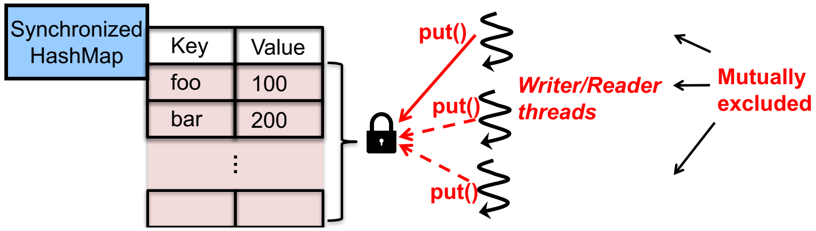


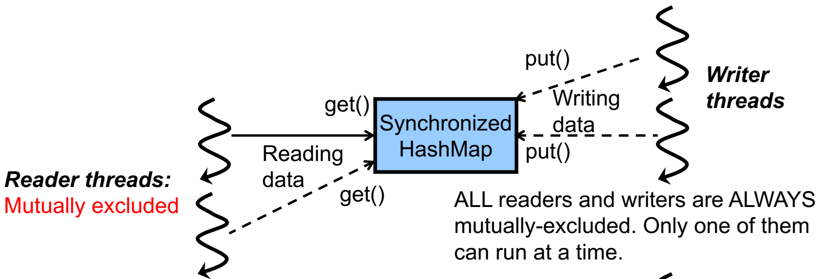
- To access a subset of the table,
  - Writer threads ARE synchronized (mutually excluded) with each other.



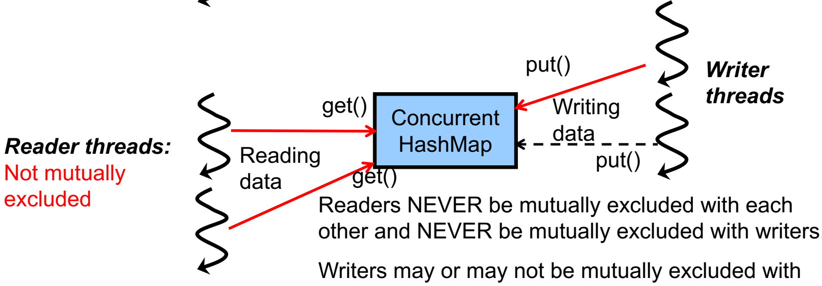
Synchronized Hash-based Map Impls

- A *single* lock is used to guard the *entire* table in `Hashtable` and synchronized `HashMap`
  - No lock stripping.
- All writer/reader threads ARE ALWAYS synchronized (mutually excluded) with each other.
  - A potential performance bottleneck as the number of key-value pairs increases and the number of threads increases.





ALL readers and writers are ALWAYS mutually-excluded. Only one of them can run at a time.



Readers NEVER be mutually excluded with each other and NEVER be mutually excluded with writers.

Writers may or may not be mutually excluded with each other.

You, as a collection user, cannot tell if the writers access the same subset of the table.

Thread-safe Iteration with Concurrent Iterators

- Supports thread-safe iteration
  - – Does not require client-side locking
  - – c.f. Thread-unsafe collections and synchronized collections require client-side locking for iteration
    - Because it is a compound operation.

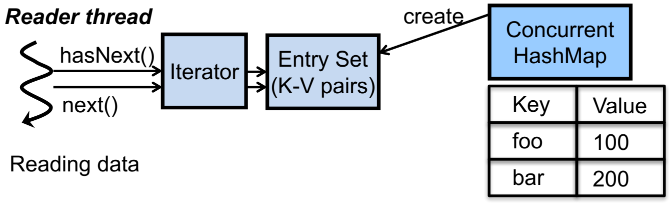
Concurrent Iterators

- Concurrent iterators are obtained through `entrySet()`, `keySet()` and `values()`.
  - `entrySet()`: Returns key-value pairs as a Set.
    - – Set<Map.Entry<K,V>>
  - `keySet()`: Returns keys as a Set.
    - – ConcurrentHashMap.KeySetView<K,V>
  - `values()`: Returns values as a Collection.
    - – Collection<V>

–
```
Iterator it = aConcurrentHashMap.entrySet().iterator();
while( it.hasNext() )
  doSomething( it.next() );
```

-
```
Iterator it = aConcurrentHashMap.entrySet().iterator();
while( it.hasNext() )
  doSomething( it.next() );
```

  - – Pros
    - No client-side locking is necessary in client code.
      - – Readers are fully concurrent (not mutually excluded).
      - – Writers can add/remove elements while readers read elements.
        - » It is guaranteed that writers and readers do not corrupt elements.
    - The iterator "it" is *backed by* the map, so changes to the map are reflected in the set.



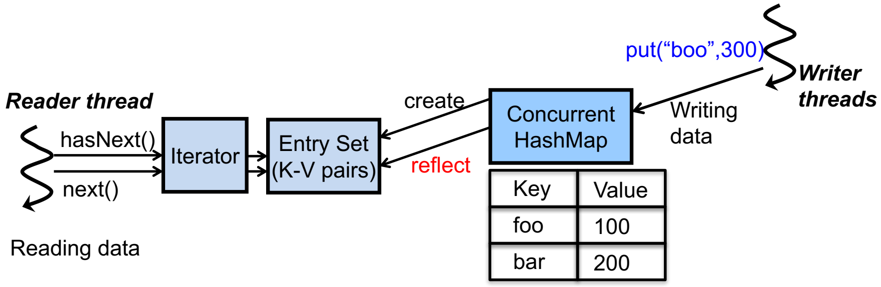
Reader thread

Reading data

Key	Value
foo	100
bar	200

-
```
Iterator it = aConcurrentHashMap.entrySet().iterator();
while( it.hasNext() )
  doSomething( it.next() );
```

  - – Pros
    - No client-side locking is necessary in client code.
      - – Readers are fully concurrent (not mutually excluded).
      - – Writers can add/remove elements while readers read elements.
        - » It is guaranteed that writers and readers do not corrupt elements.
    - The iterator "it" is *backed by* the map, so changes to the map are reflected in the set.



Reader thread

Reading data

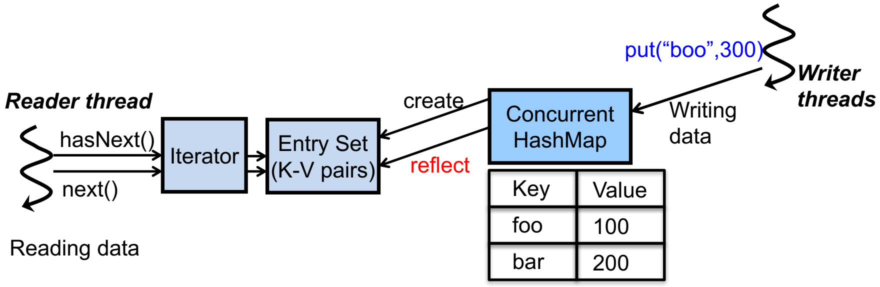
Writer threads

Writing data

Key	Value
foo	100
bar	200

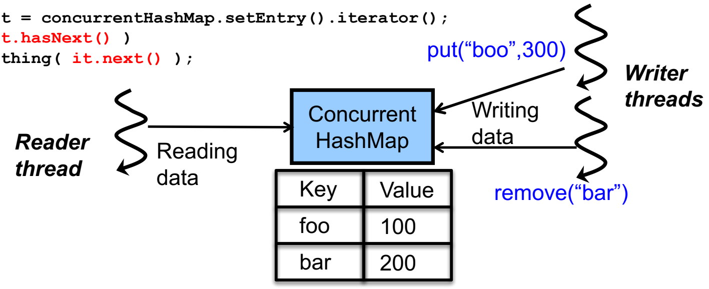
```
• Iterator it = aConcurrentHashMap.entrySet().iterator();
  while( it.hasNext() )
    doSomething( it.next() );
```

– Cons: *weak (or best-effort) consistency*
  - There is no guarantee about how soon a change (to the hash map) to be reflected into the entry set.
  - The iterator "it" may or may not traverse the most up-to-date key-value pairs in the map.



Key	Value
foo	100
bar	200

```
Iterator it = concurrentHashMap.setEntry().iterator();
  while( it.hasNext() )
    doSomething( it.next() );
```



Key	Value
foo	100
bar	200

- The iterator "it" may traverse
  - {(foo,100), (bar,200)},
  - {(foo,100), (bar,200), (boo,300)},
  - {(foo,100), (boo,300)}, or
  - {(foo,100)}.
- Elements are guaranteed not to get corrupted.
  - e.g., {(foo,300)} is never put to the map.

Consistency v.s. Performance

- `ConcurrentHashMap` trades *perfect consistency* for *performance improvement*.
  - If you can live with weak consistency, it's a great collection class for you.
    - Pros: performance improvement
    - Cons:
      - » Iterators may or may not traverse the most up-to-date key-value pairs in the map.
      - » `mappingCount()` and `isEmpty()` are not perfectly reliable.
        - The value returned is an estimate; the actual value may differ if there are concurrent insertions or removals.
  - If you cannot, craft your own thread-safe hash map with `HashMap` and `ReentrantLock`.
    - `ConcurentHashMap` has no built-in scheme to lock the entire map.

- `int size()`
  - Returns the total number of key-value pairs with `int`.
    - int: 32-bit signed integer:- 2147483647 to 2,147,483,647
    - What if you want/need to have more than 2.15 billion pairs?
- `long mappingCount()` [since Java 8]
  - Returns the total number of key-value pairs with `long`.
    - Long: 64-bit signed integer: -9223372036854775808 to 9,223,372,036,854,775,807
  - Use this method, rather than size(), when you maintain a huge number of key-value pairs.

A New Method in `Iterator`

- `java.util.Iterator<E>`
  - Used to traverse individual elements in a collection

    -
      ```
      Iterator it = concurrentHashMap.setEntry().iterator();
        while( it.hasNext() )
          doSomething( it.next() );
      ```
- **`forEachRemaining()`** [since Java 8]
  - Accepts a lambda expression (LE) and applies the LE to each remaining element
    - until all elements have been processed or the action throws an exception.

      -
        ```
        Iterator it = concurrentHashMap.setEntry().iterator();
        it.forEachRemaining( (Map.Entry<String, AtomicInteger> e)
                                 ->{System.out.println(e);} );
        ```

      -
        ```
        Iterator it = concurrentHashMap.keySet().iterator();
        it.forEachRemaining( (String k)
                                 ->{System.out.println(k);} );
        ```
  - No client-side locking is required to call **`forEachRemaining()`**

A Specialized Iterator: `Spliterator`

- **`java.util.Spliterator<E>`**
  - “Split” + “iterator”
  - Can *split* (or subset) the entire set of elements and traverse a subset of the elements.

    -
      ```
      Iterator it = concurrentHashMap.keySet().iterator();
      it.forEachRemaining( (String k)
                               ->{System.out.println(k);} );
      ```

    -
      ```
      Spliterator st = concurrentHashMap.keySet().spliterator();
      Spliterator st2 = st.trySplit();
      st.forEachRemaining( (String k)
                               ->{System.out.println(k);} );
      st2.forEachRemaining( (String k)
                               ->{System.out.println(k);} );
      ```
    - **`st`** and **`st2`** cover two different subsets.

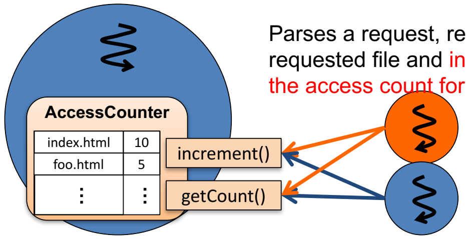
Thread-safe Compound Actions

- Supports common *compound actions* in a thread-safe way
  - put-if-absent: **`putIfAbsent(key, value)`**
    - Insert a pair of `key` and `value` as a new entry if `key` is not already associated with a value.
  - Conditional remove: **`remove(key, value)`**
    - Remove the entry for `key` if `key` is associated with `value`.
  - Conditional replace: **`replace(key, value)`**
    - Replace the entry for `key` if `key` is associated with some value.
  - Conditional replace: **`replace(key, oldValue, newValue)`**
    - Replace the entry for `key` with `newValue` only if `key` is associated with `oldValue`.
  - No client-side locking is necessary
  - c.f., **`ConcurrentMap`** interface

Exercise: Concurrent Access Counter

- **`AccessCounter`**
  - c.f. HW 12 (w/ HashMap and ReentrantLock) and HW 15 (w/ HashMap and ReentrantReadWriteLock)
  - Maintains a map that pairs a relative file path and its access count.
  - **`increment()`**
    - accepts a file path and increments the file’s access count.
  - **`getCount()`**
    - accepts a file path and returns the file’s access count.

Multi-threaded server



Parses a request, returns a requested file and increments the access count for the file.

- Imagine a CHM-based access counter
  - `increment()`

    -
      ```
      Integer oldCount = map.get(aFilePath);
      int newCount = oldCount==null? 1: oldCount+1;   // unboxing
      map.put(aFileName, newCount);                   // autoboxing
      ```
    - This client code is not thread-safe.
      - because it performs a compound operation.
      - although `get()` and `put()` are thread-safe.
      - Need client-side locking
- Alternative: Use `putIfAbsent()` and `AtomicInteger`

  -
    ```
    ConcurrentHashMap<String, AtomicInteger> map = new ...;
    map.putIfAbsent(aFilePath, new AtomicInteger(0));
    map.get(aFileName).incrementAndGet();
    ```

Thread-safe Methods that Accept Lambda Expressions

- `compute()`
  - Accepts a key and a function (as a lambda expression).
  - Applies the function on a pair of the specified key and its value.
    - If the pair does not exist, the value contains null.
  - The function (re)maps the key and value.

  -
    ```
    map.compute(aFilePath, (String k, Integer v)->
                               {return v==null? 1: ++v;})
    ```
- `computeIfAbsent()`

  -
    ```
    map.computeIfAbsent(aFilePath, (String k)->{return 1;})
    ```
- `computeIfPresent()`

  -
    ```
    map.computeIfPresent(aFilePath, (String k, int v)->
                                        {return ++v;})
    ```

HW 19

- Revise your concurrent access counter (HW 12 solution) with `ConcurrentHashMap`
  - Eliminate client-locking (to guard the map) in increment() and getCount()
    - i.e., AccessCounter no longer has a ReentrantLock data field.
  - Use lambda expressions whenever possible.

Bulk Operations

- Repeatedly apply a given lambda expression on key-value pairs.
  - *forEach*: `forEach()`, `forEachEntry()`, `forEachKey()`, `forEachValue()`
  - *Search*: `search()`, `searchEntries()`, `searchKeys()`, `searchValues()`
  - *Reduce*:
    - `reduce()`, `reduceToDouble()`, `reduceToInt()`, `reduceToLong()`
    - `reduceEntries()`, `reduceEntriesToDouble()`, `reduceEntriesToInt()`, `reduceEntriesToLong()`
    - `reduceKeys()`, `reduceKeysToDouble()`, `reduceKeysToInt()`, `reduceKeysToLong()`
    - `reduceValues()`, `reduceValuesToDouble()`, `reduceValuesToInt()`, `reduceValuesToLong()`
- Receives a “parallelism threshold” as the first parameter.
  - Executed with threads if the # of key-value pairs exceeds this threshold.
    - `Long.MAX_VALUE`: Suppress concurrency/parallelism. Use a single thread.
    - 1: Maximize concurrency/parallelism.

- The number of threads to be used:
  - If (# of key-value pairs) < threshold
    - No threads to be used
  - If (# of KV pairs) / threshold >= (# of CPU cores)
    - Use (# of CPU cores)
  - If (# of KV pairs) / threshold < (# of CPU cores)
    - Use (# of KV pairs) / threshold
      - Rounded to an int number

`forEach` Bulk Operations

- Performs a given action on each key-value pair.

```
– map.forEach( 50,
               (k,v)->{if(v>10000)
                          System.out.println(k + “->” + v) )
```

  - First param: parallelism threshold (50)
  - Second param: action (`BiConsumer`)
  - Printing out the key-value pairs that have more than 10,000 access count.

`search` Bulk Operations

- Searches a key-value pair that satisfies a given search criterion, which is specified as a lambda expression
  - Returns a non-null result if found. Returns null otherwise.
  - Skips further search when a result is returned.

```
– Path path = map.search(50, (k,v)->{v>10000? v: null} )
```

  - Second param: search function (`BiFunction`)
    - Is there at least one file that has more than 10,000 access count?
  - If there exists such a file, `path` contains a path to that file.
  - Only the first search hit is returned. (Extra search hits are ignored)

```
– Integer count = map.searchValues(50, (v)->{v>10000? v: null} )
```

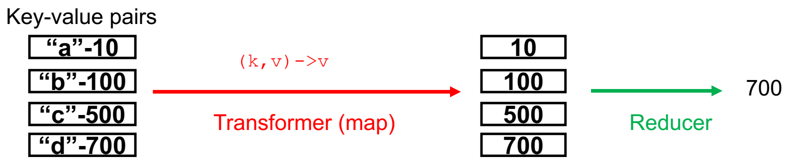
  - Second param: search function (`Function`)
    - Is there at least one file that has more than 10,000 access count?
  - If there exists such a file, `count` contains the access count of that file.

`reduce` Bulk Operations

- Reduces key-value pairs to a single value.

```
– Int maxCount = map.reduceToInt(50,
                                  (k,v)-> v,
                                  0,
                                  (max, v)-> v>max? v: max );
```

  - Second param: transformer (`BiFunction`)
    - Expected to work like a map operation
  - Fourth param: reducer (`BinaryOperator`)
  - Third param: the initial value for accumulated value (`max`).



Key-value pairs

“a”-10

“b”-100

“c”-500

“d”-700

(k,v)->v

Transformer (map)

10

100

500

700

Reducer

700

-
```
int maxCount = map.reduceToInt(50,
                               (k,v)->v,
                               0,
                               (max, v)-> v>max? v: max );
```
  - Second param: transformer (`BiFunction`)
    - Expected to work like a map operation
  - Fourth param: reducer (`BinaryOperator`)
  - Third param: the initial value for accumulated value (`max`).

- Generalized form of reduce operations with the Streams API
  -
```
T result = aStream.reduce(initValue, (result, element)-> ... );
```
  -
```
T result = initValue;
for(T element: collection){
    result = accumulate(result, element);   }
```

Notes on Bulk Operations

- Bulk operations
  - are thread-safe in that they never corrupt key-value pairs.
  - may not be performed on the most up-to-date key-value pairs.
    - Write threads can modify key-value pairs when read threads are reading key-value pairs.
      - It is guaranteed that read and write threads do not corrupt key-value pairs.
      - c.f. Concurrent iteration
    - Read threads can be fully concurrent as if a read-write lock is used in a CHM.

Recap: Concurrent Collections

- “Ready-made” thread-safe collections
  - Introduced in Java 5 (2004) and enhanced in subsequent versions
    - Queue
      - `ConcurrentLinkedQueue` (since Java 5)
      - `ConcurrentLinkedDeque` (since Java 7)
      - `ArrayBlockingQueue` (since Java 5)
      - `LinkedBlockingQueue` (since Java 5)
      - `DelayQueue` (since Java 5)
      - `PriorirtyBlockingQueue` (since Java 5)
      - `LinkedBlockingDeque` (since Java 6)
      - `LinkedTransferQueue` (since Java 7)
    - Map
      - `ConcurrentHashMap` (since Java 5)
      - `ConcurrentSkipListMap` (since Java 6)
    - Set
      - `ConcurrentSkipListSet` (since Java 6)
  - `java.util.concurrent.CopyOnWriteXyz` classes
    - `CopyOnWriteArrayList`
    - `CopyOnWriteArraySet`

- Every concurrent collection…
  - Implements its public methods in thread-safe and performance-aware manners, just like `ConcurrentHashMap`.
  - Supports thread-safe, weak-consistent iteration.
    - May or may not provide thread-safe methods tailored for the other types of compound operations
      - c.f. `putIfAbsent(key, value)` and `compute()` in `ConcurrentHashMap`

Some Major Concurrent Collections

- **ConcurrentLinkedQueue**
  - Concurrent implementation of `java.util.Queue`
    - FIFO (First-In-First-Out) queue
- **ConcurrentLinkedDeque**
  - Concurrent implementation of `java.util.Deque`
    - LIFO (Last-In-First-Out)
- **ConcurrentSkipListMap**
  - An implementation of `ConcurrentNavigableMap`.
  - Map entries are kept sorted according to the natural ordering of their keys or by a custom `Comparator`.
- **ConcurrentSkipListSet**
  - A concurrent implementation of `NavigableSet`.
  - Set elements are kept sorted according to the natural ordering or by a custom `Comparator`.

Concurrent Hash Set???

- No concurrent collection class for hash-based set in Java API
  - No such class like `ConcurrentHashSet`.
- You can "emulate" it through `ConcurrentHashMap`:
  - `Set<String> set = ConcurrentHashMap<String, Integer>.newKeyset();`
    - Values (`Integer`s) are ignored.
    - `set` is a hash-based set that contains a series of `String` data.
    - `newKeySet()` is a static factory method to generate a `ConcurrentHashMap.KeySetView<K, Boolean>`
      - `ConcurrentHashMap.KeySetView<String, Boolean>` in the above example.

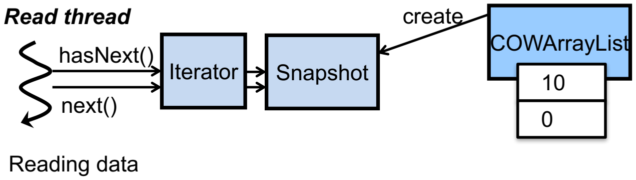
Copy-On-Write (COW) Collections

- `CopyOnWriteArrayList`
- `CopyOnWriteArraySet`
  - Concurrent replacements of synchronized wrappers for `ArrayList` and `ArraySet`.
- Key features
  - Thread-safe public methods
    - No client-side locking is necessary to call them.
  - Read threads are never mutually excluded with each other.
  - Read threads and write threads are never mutually excluded.
  - Write threads are never mutually-excluded.

Snapshot-based Iteration in COW Collections

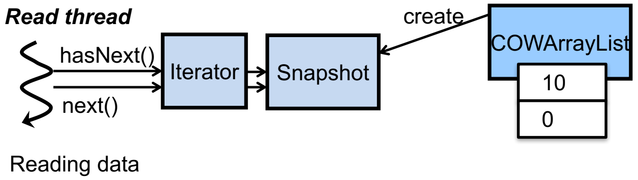
- Support thread-safe, snapshot-based iteration.
  - A read thread references and operates on a *collection snapshot*, which is a collection that is up-to-date when an iterator is created.

```
Iterator it = aCOWArrayList.iterator(); // A snapshot is created.
  while( it.hasNext() )
    doSomething( it.next() );
```

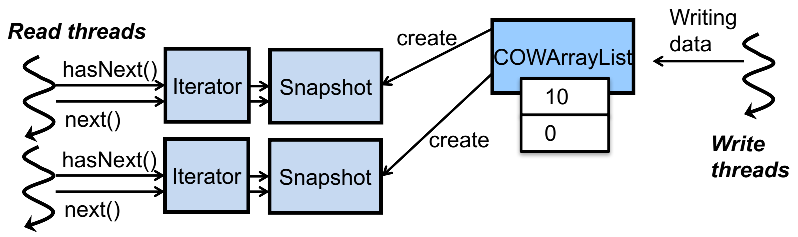


- Pros
  - No client-side locking is necessary for iteration.
    - Read threads ARE NOT mutually excluded with each other and with writer threads.
      - Each iterator has a thread-specific snapshot of List elements.
      - Different readers get different snapshots and access them concurrently.

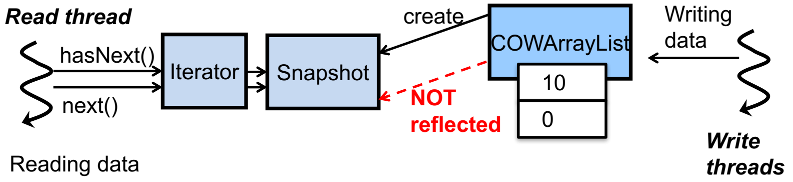
```
Iterator it = aCOWArrayList.iterator(); // A snapshot is created.
  while( it.hasNext() )
    doSomething( it.next() );
```



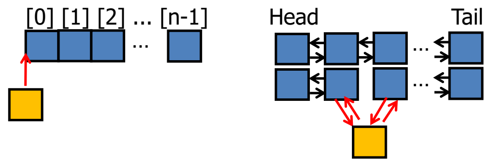
- Pros
  - No client-side locking is necessary for iteration.
    - Read threads ARE NOT mutually excluded with each other and with writer threads.
      - Each iterator has a thread-specific snapshot of List elements.
      - Different readers get different snapshots and access them concurrently.

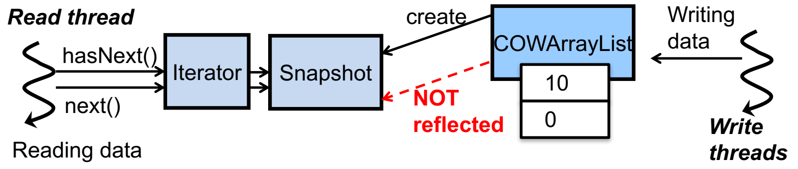


- Cons
  - The snapshot can become outdated.
    - e.g., when a write thread adds/removes collection elements after a snapshot is created.
  - No consistency preserved; Each iterator will NOT reflect additions, removals and other changes to the list after the iterator was created.
    - c.f. weak consistency in `ConcurrentHashMap` (and other non-COW concurrent collections)
- Trades consistency for performance



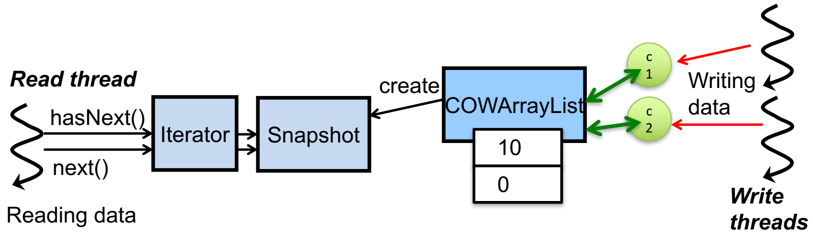
- Cons
  - No consistency preserved. Why?
  - State updates (particularly, element insertion and removal) are often expensive for lists.
    - Due to index-based element ordering



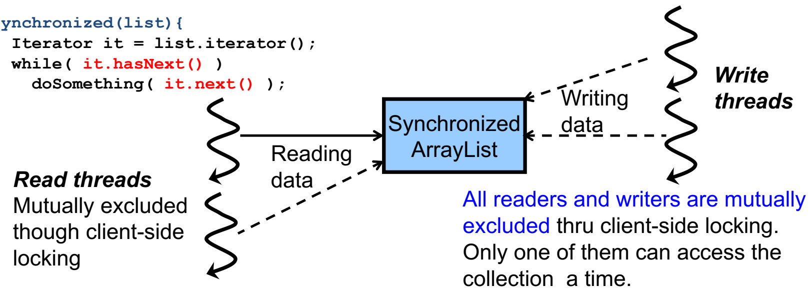


Copy-On-Write (COW)

- Making a copy of a collection when a write thread updates the collection's elements
- A write thread
  - Performs add(), remove(), set() and other state-changing (or mutative) methods on ***a duplicate copy*** of collection elements.
  - Synchronizes the updated/modified copy with the original element set.
- Write threads ARE NOT mutually excluded with each other and with read threads.



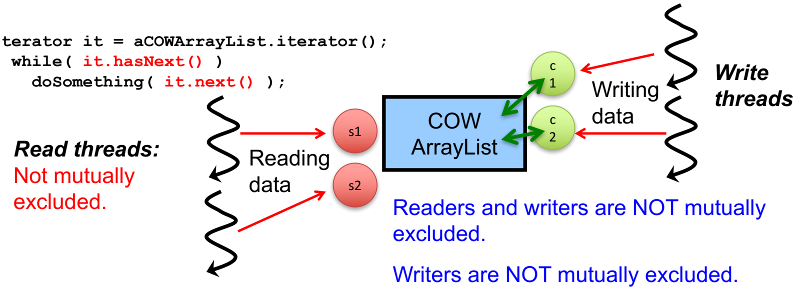
```
synchronized(list){
  Iterator it = list.iterator();
  while( it.hasNext() )
    doSomething( it.next() );
}
```



Read threads
Mutually excluded though client-side locking

All readers and writers are mutually excluded thru client-side locking. Only one of them can access the collection a time.

```
Iterator it = aCOWArrayList.iterator();
  while( it.hasNext() )
    doSomething( it.next() );
```

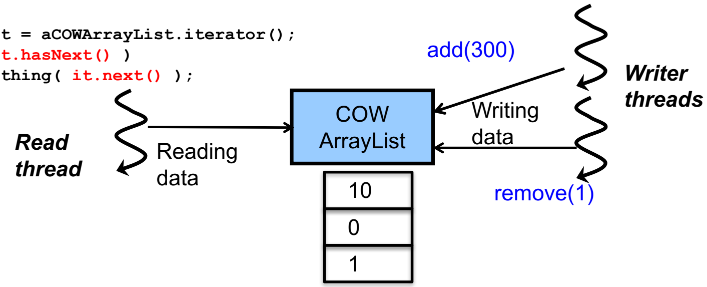


Read threads:
Not mutually excluded.

Readers and writers are NOT mutually excluded.

Writers are NOT mutually excluded.

```
Iterator it = aCOWArrayList.iterator();
  while( it.hasNext() )
    doSomething( it.next() );
```



- Guaranteed that collection elements are never corrupted.
  - Read thread obtains:
    - (10, 0, 1),
    - (10, 0, 1, 300)
    - (10, 0), or
    - (10, 0, 300)
  - It never obtains corrupted list such as (10, 300).

Pros and Cons in COW Performance

- Pros
  - No concerns about data corruption.
  - Improved performance for iteration
- Cons
  - State-changing methods (e.g., add(), remove()) are very slow.
    - Never use COW collections in single-threaded programs.
    - Their overhead grows exponentially as the number of elements increases.
      - The overhead of add() [msec]

# of elems	ArrayList	SyncArrayList	COWArrayList
1,000	0	0	14
5,000	0	0	102
10,000	0	0	409
20,000	0	0	1,712
30,000	15	16	4,566

When to Use COW Collections?

- Read operations are executed a lot more often than write operations.
- When the # of read threads is a lot greater than the # of write threads.
- When state-changing methods are rarely called.
- When the # of elements is relatively small.

In Summary…

- Be aware of the characteristics of COW collections.
- Be conservative to use them.

Lists and Concurrency

- Performance-wise, lists are not that great collections for multi-threaded programs
  - Particularly when a list has many elements.
  - Because state updates are often expensive due to index-based element ordering
    - This makes thread synchronization (locking) more expensive.
- Consider to use other collections
  - Such as queues and sets.
    - Queue: Ordered. Duplicate elements allowed.
    - Set: Not ordered. No duplicated elements allowed.

Recap: `Observable.notifyObservers()`

```
• class Observable {
   private Vector obs;             //observers
   private boolean changed = false;

   public void notifyObservers(Object arg){
     Object[] arrLocal;
     synchronized (this){          // lock.lock();
       if (!changed) return;       // balking
       arrLocal = obs.toArray();  // observers copied to arrLocal
       changed = false;
     }                             // lock.unlock();
     for (int i = arrLocal.length-1; i < =0; i--)
       ((Observer)arrLocal[i]).update(this, arg); // OPEN CALL
   }
   ……
  }
```

HW 20

- Revise your HW 18 code to replace `ArrayList` with
  - `ConcurrentLinkedQueue`,
  - concurrent hash set (`ConcurrentHashMap.KeySetView<K, Boolean>`), or
  - `CopyOnWriteArrayList`

Observable.notifyObservers()

```
class Observable {
 private ArrayList<Observer> obs;
 private boolean changed = false;
 private ReentrantLock lock;

 void addObserver(Observer o){
  lock.lock();
  obs.add(o);
  lock.lock();}

 void notifyObservers(Object arg){
  ArrayList<Observer> obsLocal;
  lock.lock();
  if(!changed) return;
   obsLocal =
     new ArrayList<Observer>(obs);
   changed = false;
  lock.unlock();
  Iterator it = obsLocal.iterator();
  while( it.hasNext() ){
    it.next().update(this, arg);
  }}
```

```
class Observable {
 pri… CopyOnWriteArrayList<Observer> obs;
 private boolean changed = false;
 private ReentrantLock lock;

 void addObserver(Observer o){
  obs.add(o);
 }

 void notifyObservers(Object arg){
  lock.lock();
  if(!changed) return;
   changed = false;
  lock.unlock();
  Iterator it = obs.iterator();
  while( it.hasNext() ){
    it.next().update(this, arg);
  }
 }
```