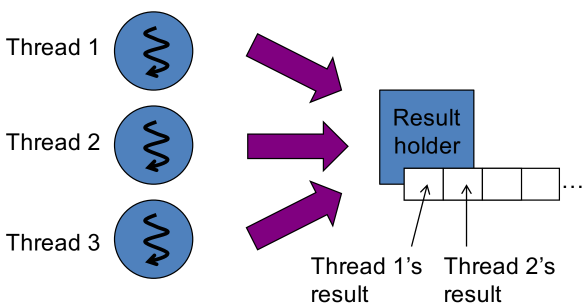
Thread-Specific Storage (TSS)

Thread-Specific Storage (TSS)

- Storage (variable) allocated/reserved per thread.
  - The storage/variable is not accessible by other threads.

- java.lang.ThreadLocal<T>

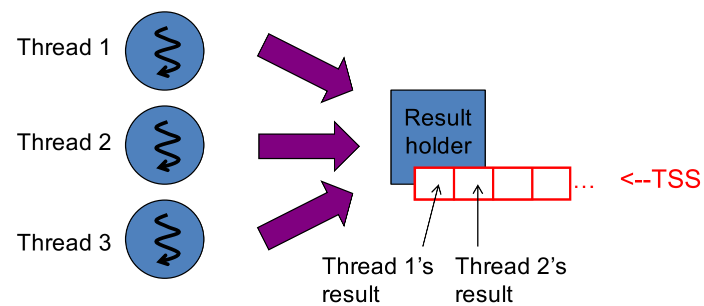
Imagine this Scenario…



- Different threads
  - generate different data of the same time (T)
  - store them in a result holder
  - read them from the result holder.

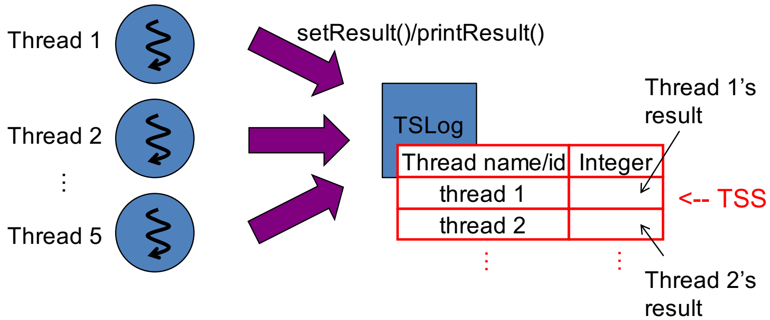
- Need to guard the result holder from threads.
  - Locking required. Maybe read-write lock.

When does a TSS Work?



- If each element is paired with a thread and accessed only by the thread…
  - TSS works well.
    - Easier-to-read code
    - Safer code

Sample Code: TSLog.java



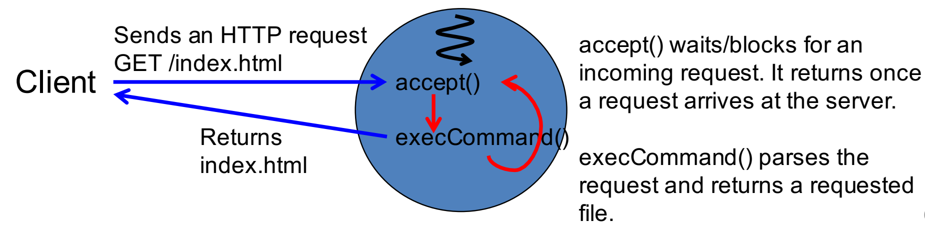
- Locking is encapsulated in ThreadLocal
- No ways to access other threads' TSS.

- No locking in client code
  - Shorter (easier-to-understand) code
  - No worry about race conditions and deadlocks.

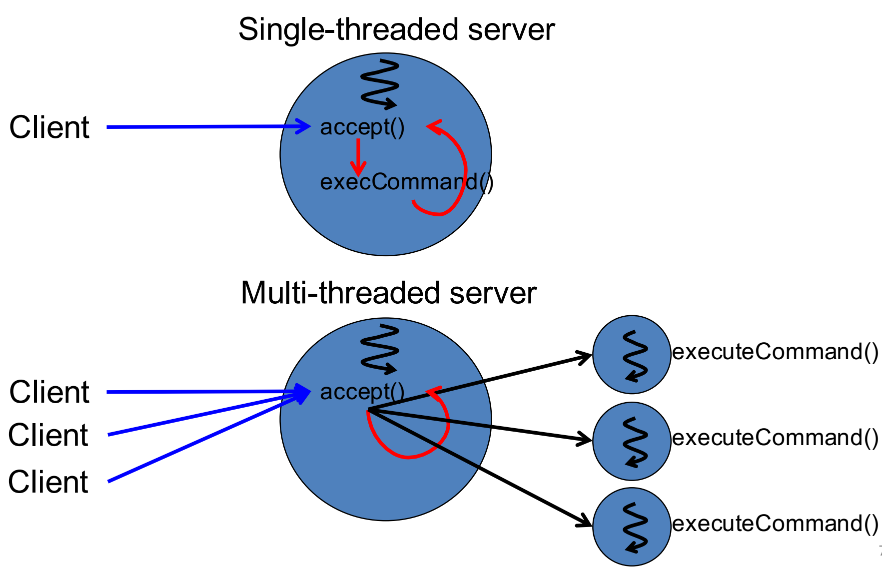
Potential Use Case: Concurrent Web Server

- Suppose you implement your own web server.
  - Receives a request that a client (browser) transmits to ask for an HTML file.
  - Returns the requested file to the client.
- What if the server receives multiple requests from multiple clients at the same time?
  - If the server is single-threaded, it processes requests *sequentially*.

Single-threaded server

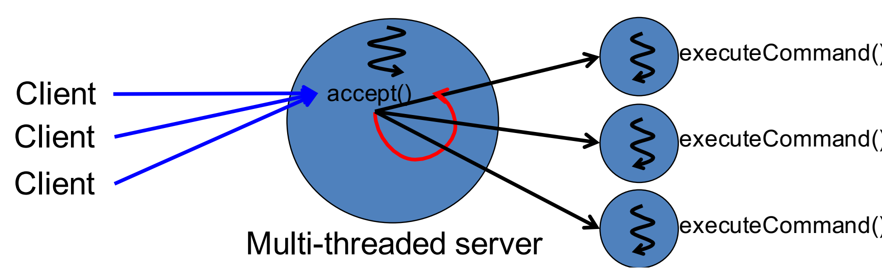


Concurrent (Multi-threaded) Web Server

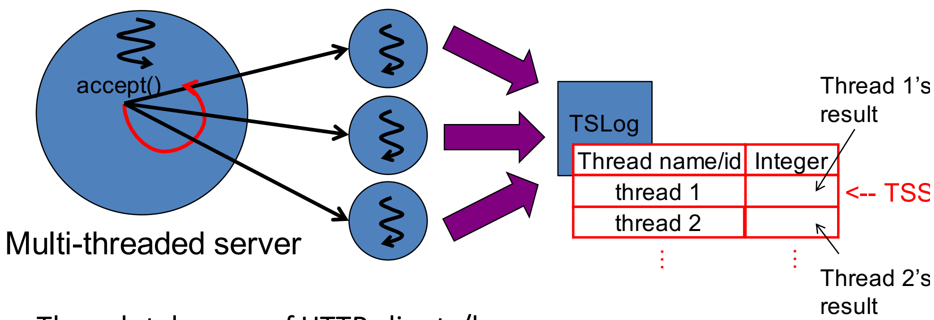


Thread-per-request Concurrency

- Thread-per-request
  - Once the web server receives a request from a client, it creates a new thread.
    - The thread parses the incoming request and returns a requested file.
    - The thread terminates once the requested file is sent out to the client.



TSS in Concurrent Web Server



- Threads take care of HTTP clients/browsers.
  - Parsing an incoming HTTP command, retrieving a target content, caching it, incrementing its access counter, logging, etc. etc.
  - They may…
    - Need customer info (e.g. customer ID) from browser cookies to display some personalized content.
    - Need client-specific information (e.g., client OS name/type and browser name/type) to display some client-specific content.

InheritableThreadLocal (Inheritable TSS)

- java.lang.InheritableThreadLocal<T>
  - Subclass of ThreadLocal<T>
  - Extends ThreadLocal to provide inheritance of values from parent thread to child thread.
    - When a child thread is created, the child receives initial values for all inheritable thread-local variables for which the parent has values.
  - Useful in the previous concurrent web server example if more than one threads are used to process a single HTTP request.

Concurrency and Collections

3 Types of Collections

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

Thread-unsafe Collections

- Many collection classes are NOT thread safe.
  - e.g., ArrayList, HashMap, etc.
  - Need to use thread synchronization (i.e., locking) to guard collection elements from concurrent accesses.
- Make sure that a collection is thread-safe or not with Java API documentation

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.)…”

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 collection state 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 class that wraps/contains an 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. etc.

A Thread Safety Issue in Synchronized Collections

- All (public) methods are thread-safe in all synchronized collection classes.
- Need *client-side locking* in *compound actions* on a synchronized collection
  - Iteration (element-traversal)
    - Repeatedly get elements until a collection becomes empty
  - Navigation
    - Find the next element after a given element
  - Conditional operations (check-then-act)
    - e.g., Check if a Map has a key-value pair for the key K, and if not, add the pair (K, V)

Example Compound Actions

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

  Iterator it = list.iterator();
  while( it.hasNext() )        // check-then-act for iterations
    doSomething( it.next() ); // navigation
  ```
- ```
  if( list.size() > 10 )       // check-then-act
    doSomething( list );
  ```

Potential Problems in Compound Actions

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

  Iterator it = list.iterator();
  while( it.hasNext() )        // check-then-act for iterations
    doSomething( it.next() ); // navigation
  ```
- ```
  if( list.size() > 10 )       // check-then-act
    doSomething( list );
  ```

  Race conditions can occur here.

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

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

Why Not Using ReentrantLock?

- ```
  synchronized(list){
    if( list.size() > 10 )
      doSomething( ... );
  }
  ```
- ```
  ReentrantLock lock = new ReentrantLock();
  lock.lock();
  if( list.size() > 10 )
      doSomething( ... );
  lock.unlock();
  ```
- `synchronized(list)`: acquires the lock that "list" owns/uses for thread synch in its public methods.
- "lock" is different from the lock that "list" owns/uses for thread synch in its public methods.

Performance Implications 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
  }
  ```
- Perfectly thread-safe.
  - Writer threads need to wait until a reader thread completes a traversal on the entire set of elements.
- Degraded performance
  - if many reader threads run.
  - if iterations occur very often.

- ```
  ReentrantLock lock = new ReentrantLock();
  List<String> list = new ArrayList<String>();
  ...
  lock.lock();
  Iterator it = list.iterator();
  while( it.hasNext() )
    doSomething( it.next() );
  lock.lock();
  ```
- This code is more efficient than the previous one.
  - No nested locking.
  - Use a read-write lock if you have many reader threads.
  ```

Concurrent Collections

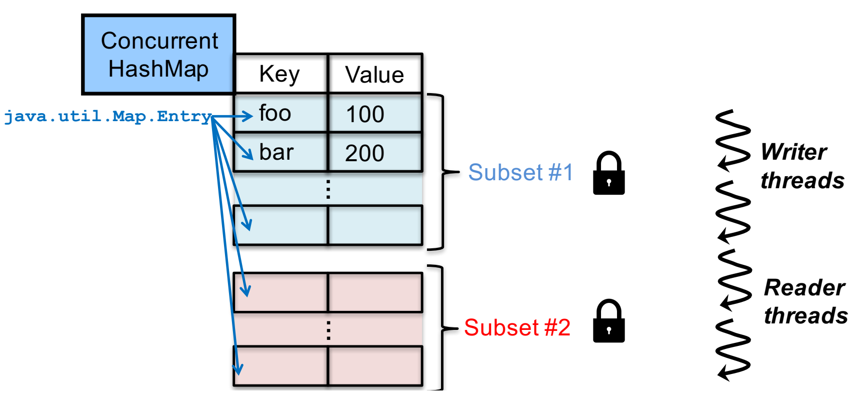
- Ready-made "thread-safe" collections
  - `java.util.concurrent.ConcurrentXyz` classes
    - ConcurrentHashMap
    - ConcurrentLinkedQueue
    - ConcurrentLinkedDeque
    - ConcurrentSkipListMap
    - ConcurrentSkipListSet
  - `java.util.concurrent.CopyOnWriteXyz` classes
    - CopyOnWriteArrayList
    - CopyOnWriteArraySet

What is ConcurrentHashMap?

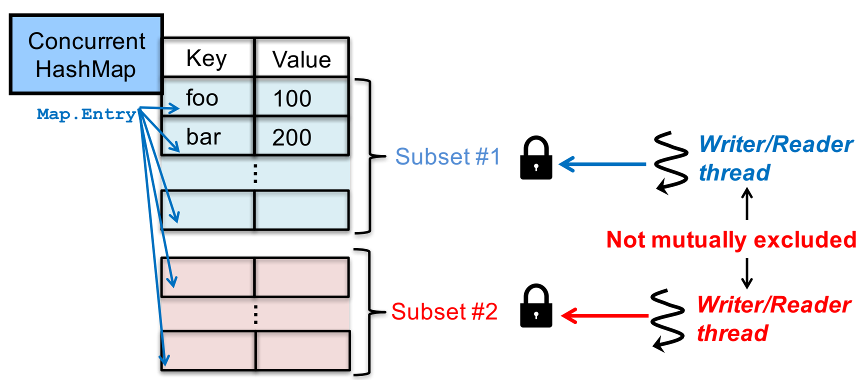
- A replacement for synchronized hash-based Map implementations (e.g., Hashtable and synchronized HashMap).
- Performs *finer-grained* locking, called "lock stripping"
  - Compared to coarse-grained (i.e., class-wide) locking in synchronized hash-based Map implementations.
- Aims
  - Greater degree of shared, concurrent access
  - Greater performance

Lock Stripping

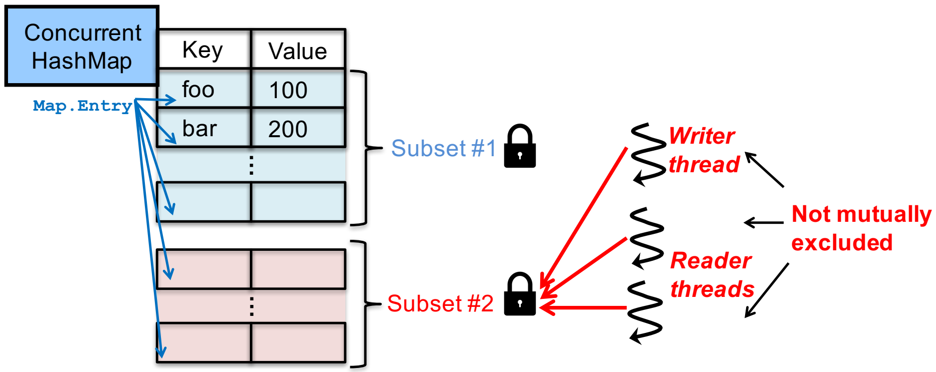
- ConcurrentHashMap uses multiple locks to guard a table (i.e., key-value pairs).
  - 16 locks by default
    - Configurable with the "concurrencyLevel" parameter in a constructor.
  - Each lock is associated with a subset of the table.



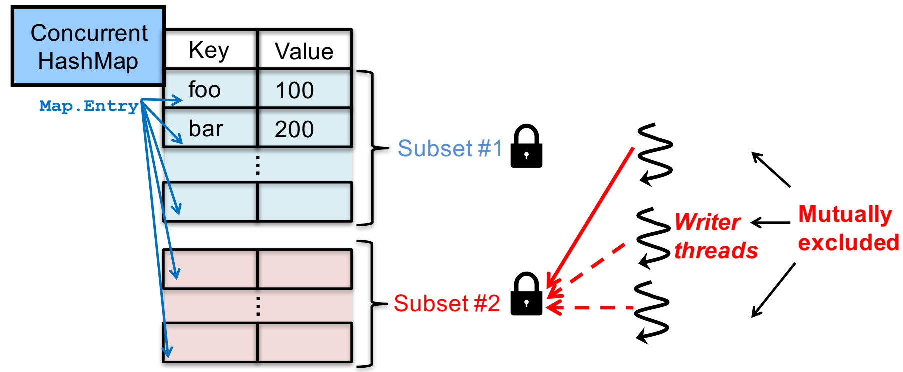
- Threads are not synchronized (not mutually excluded) with each other
  - as far as they access different subsets of the table through different locks.



- 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>

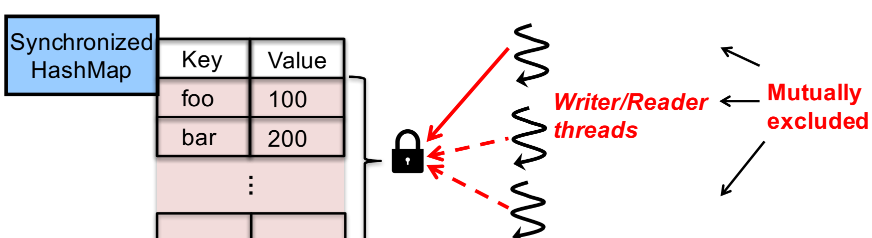


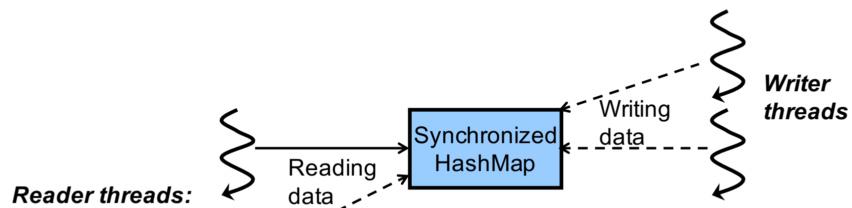
- To access a subset of the table,
  - Writer threads ARE (mutually excluded) with each other.



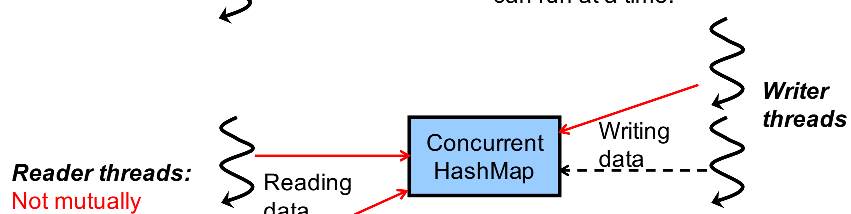
Synchronized Hash-based Map Impls

- 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.

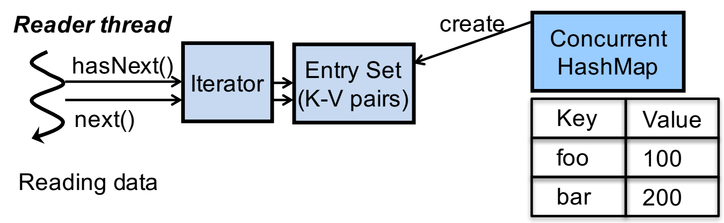
Concurrent Iteration in ConcurrentHashMap

- Supports concurrent iterators
  - 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() );
```

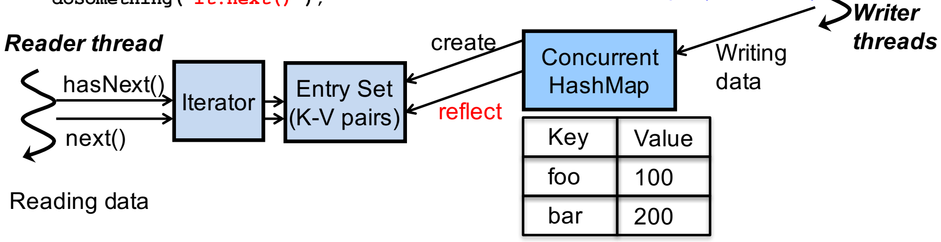
-
```
Iterator it = aConcurrentHashMap.entrySet().iterator();
while( it.hasNext() )
  doSomething( it.next() );
```
  - Pros
    - No client-side locking is necessary in client code.
      - Readers are fully concurrent (not mutually excluded).
      - Writers can add/remove elements while readers read 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.

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



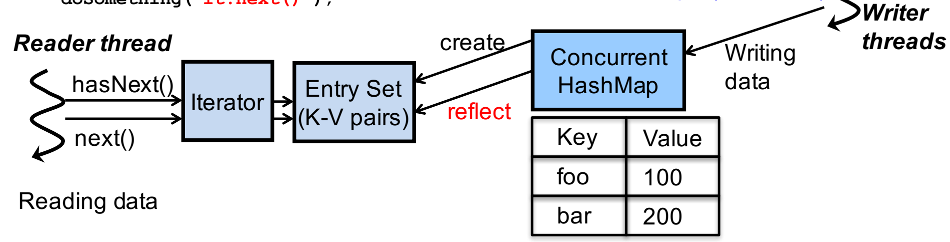
-
```
Iterator it = aConcurrentHashMap.entrySet().iterator();
while( it.hasNext() )
  doSomething( it.next() );
```
  - Pros
    - No client-side locking is necessary in client code.
      - Readers are fully concurrent (not mutually excluded).
      - Writers can add/remove elements while readers read 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.

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

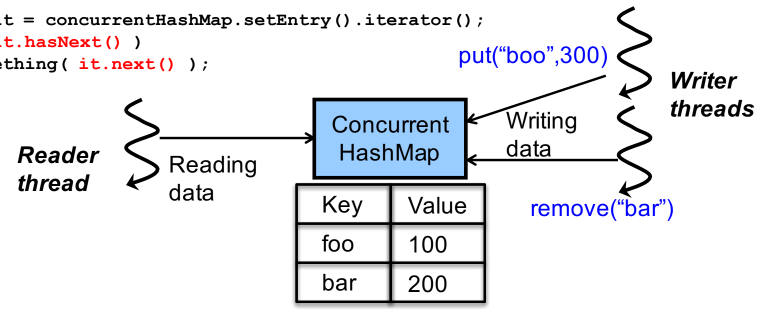


-
```
Iterator it = aConcurrentHashMap.entrySet().iterator();
while( it.hasNext() )
  doSomething( it.next() );
```
  - Cons: *weak consistency*
    - There is no guarantee about how soon the changes to be reflected in the set.
    - The iterator “it” may or may not traverse the most up-to-date key-value pairs in the map.

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



```
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)}.

- Guaranteed that elements are never corrupted.
  - e.g., {(foo,300)}

Notes

- ConcurrentHashMap trades perfect consistency for performance improvements.
  - If you can live with weak consistency, it is a great concurrent collection class.
    - Pros: performance.
    - 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, use HashMap with a ReentrantLock.
    - ConcurrentHashMap has no built-in way to lock the entire map.

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

Other Concurrent Collections

- ConcurrentLinkedQueue
  - Concurrent implementation of java.util.Queue
    - FIFO (First-In-First-Out) queue
- ConcurrentLinkedDeque
  - Concurrent implementation of java.util.Deque

- 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 impleemntation of NavigableSet.
  - Set elements are kept sorted according to the natural ordering or by a custom Comparator.

Copy-On-Write (COW) Collections

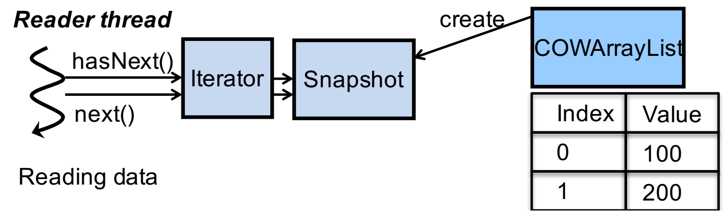
- `CopyOnWriteArrayList`
- `CopyOnWriteArraySet`

- Concurrent replacements of synchronized wrappers for ArrayList and ArraySet.
  - No client-side locking is necessary.
  - Readers are not mutually excluded.
  - Readers and writers are not mutually excluded.
    - c.f. *snapshot-based* iteration over Observers in the Observable-Observer API.

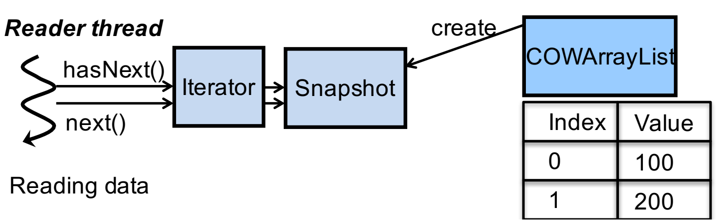
Snapshot-based Iteration in COW Collections

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

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

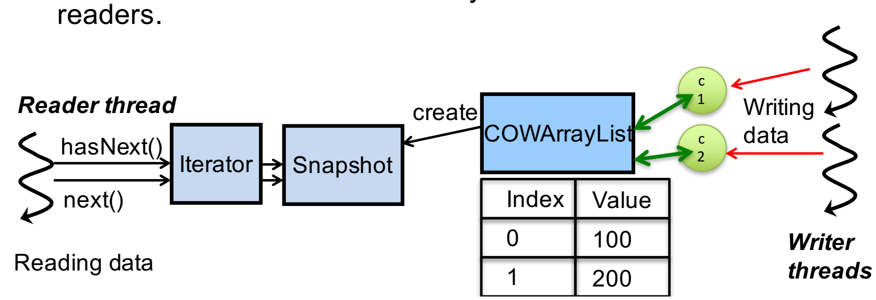


- Pros
  - No client-side locking is necessary.
    - Reader threads ARE NOT mutually excluded with each other and with writer threads.
      - The iterator "it" has a thread-specific snapshot of List elements.
      - Different readers get different snapshots and access them concurrently.
- Cons
  - The snapshot may not be perfectly consistent (maybe outdated)
    - e.g., if a writing threads add/remove collection elements after a snapshot is created.
  - The iterator will not reflect additions, removals, or changes to the list since the iterator was created.
- Trades perfect consistency for performance

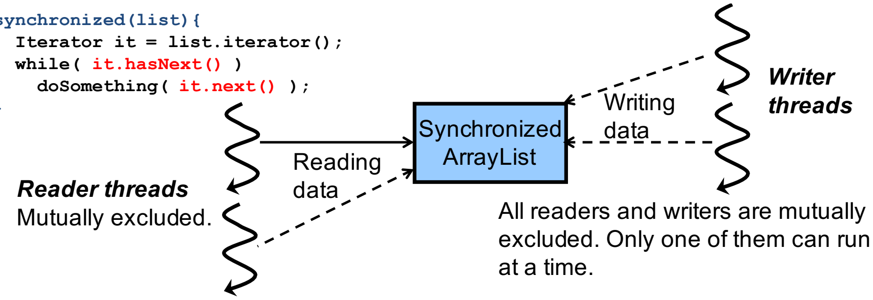


Copy-On-Write (COW)

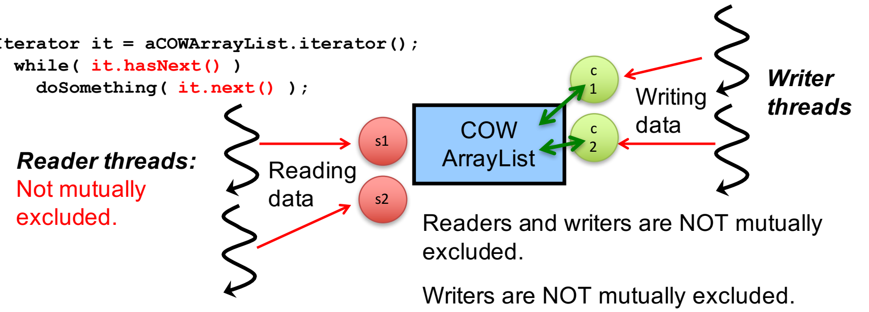
- Making a copy of a collection when a writer thread updates the collection's elements
- A writer thread
  - Performs add(), remove(), set() and other element-changing methods on ***the duplicate copy*** of collection elements.
  - Synchronizes the updated/modified copy with the original element set.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



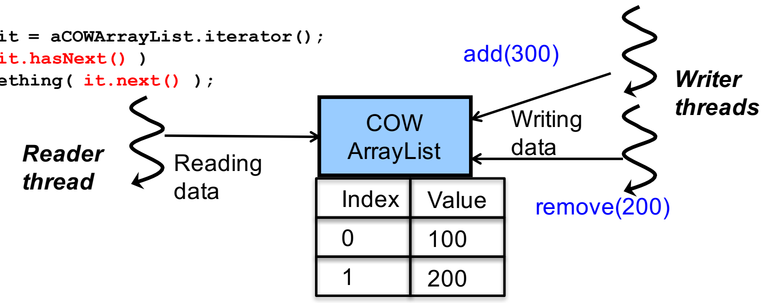
```
synchronized(list){
  Iterator it = list.iterator();
  while( it.hasNext() )
    doSomething( it.next() );
}
```




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




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



- The iterator "it" always traverse
  – (100, 200)
- Guaranteed that connection elements are never corrupted.

Pros and Cons in COW Performance

- Pros
  – No concerns about thread safety
  – Improved performance for iterators
    - Iterators can be concurrent even if writer threads exist.
    - Writer threads are perfectly concurrent (i.e. no mutual exclusions).
- Cons
  – Element-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?

- When the # of reader threads is greater than the # of writer threads.
- When element-changing methods are rarely called.
- When each writer thread updates a small number of elements at a time.
- When the # of elements is relatively small.
- When a snapshot-based element traversal is preferable over
  – Weekly-consistent traversals in ConcurrentXyz classes
  – Perfectly thread-safe (race condition free) traversals with a synchronized collection and a read-write lock.

Recap: Observable.notifyObservers()

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

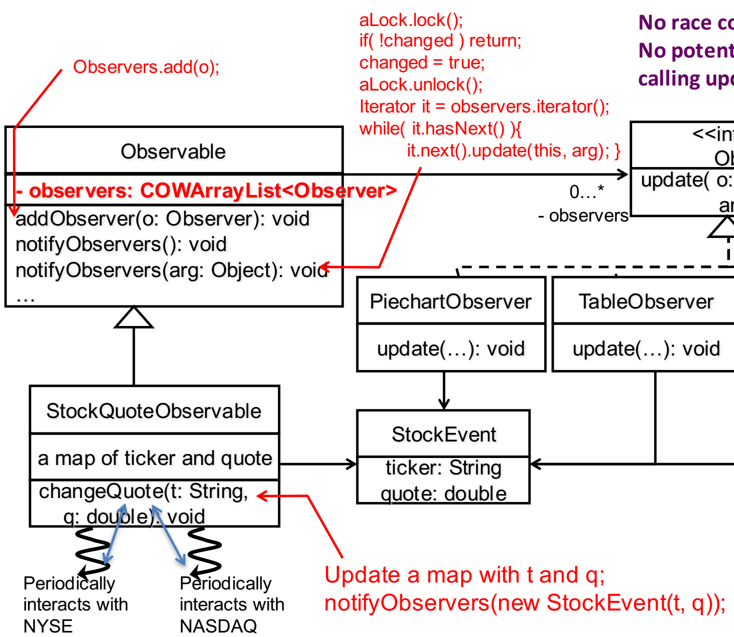
    public void notifyObservers(Object arg){
      Object[] arrLocal;
      synchronized (this){           // lock.lock();
        if (!changed) return;        // balking
        arrLocal = obs.toArray();    // observers copied to arrLocal
        changed = false;
      }                              //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

HW 24

- Implement a thread-safe version of the Obsrvable class in the concurrent Observer design pattern.
  - Use CopyOnWriteArrayList, not ArrayList
    - c.f. HW 23

Concurrent Observer

No race conditions can occur.
No potential deadlock risk by calling update()



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);
  }
 }
```