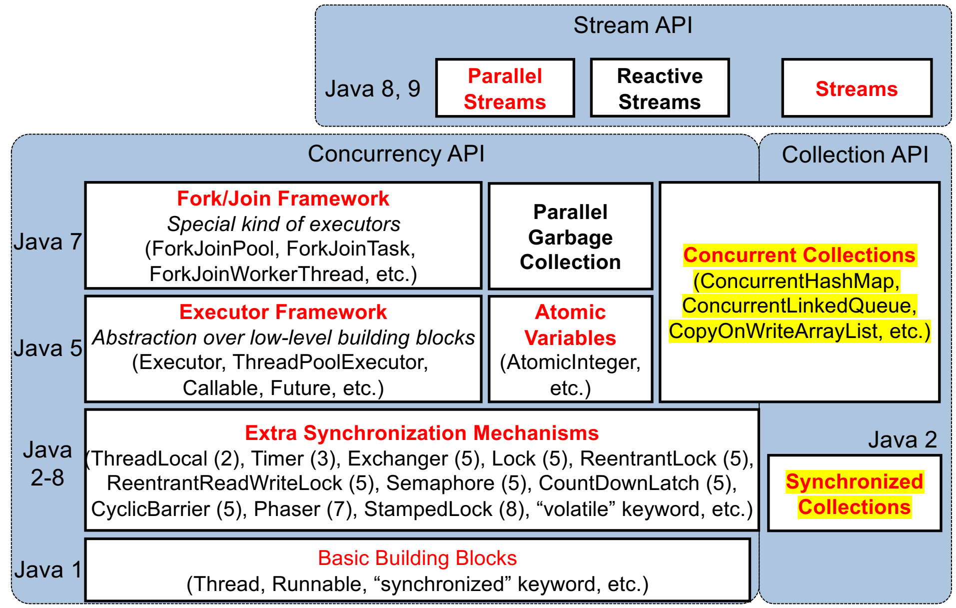
Collections and Concurrency

Concurrency API in Java



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 (locking).
- Look into Java API documentation to see if a collection is thread-safe or not.

Java API Doc on `ArrayList`

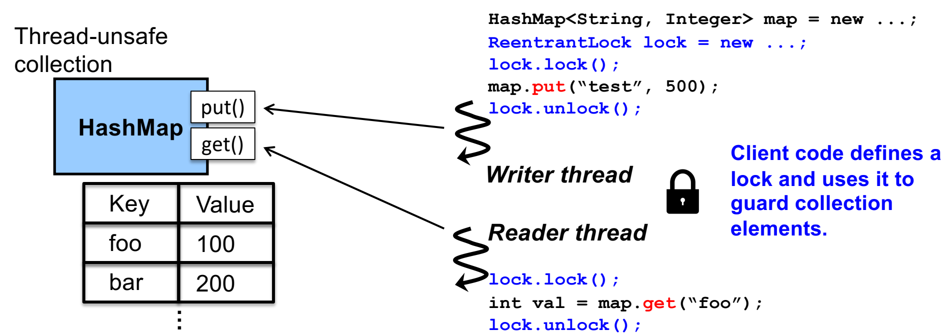
- **"Note that this implementation is not synchronized.** If multiple threads access an ArrayList instance concurrently, and at least one of the threads modifies the list structurally, it *must* be synchronized externally.
  - (A structural modification is any operation that adds or deletes one or more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

- You must do *client-side locking* for compound operations as well as simple public method calls.

- Example compound operations
  - Iteration (element-traversal)
    - Get all elements one by one
  - 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)`

When You Use a Thread-unsafe Collection…

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



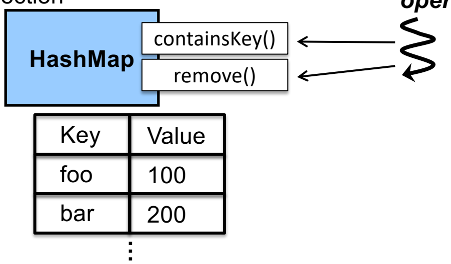
Example Compound Operations

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

  Iterator it = list.iterator();
  while( it.hasNext() ){
    doSomething( it.next() ); }            // Iteration
  ```
- ```
  for(int i=0; i<list.size(); i+=2){       // Navigation
    doSomething( list.get(i) ); }
  ```
- ```
  while( it.hasNext() ){
    if( it.next().endsWith(".jar" ){...} } // Navigation
  ```
- ```
  if( list.size() > 10 )                   // Check-then-act
    doSomething( list );
  ```

Client-side Locking for Compound Operations

- An example "check-then-act" compound operation

Thread-unsafe collection



Thread that performs both read and write operations

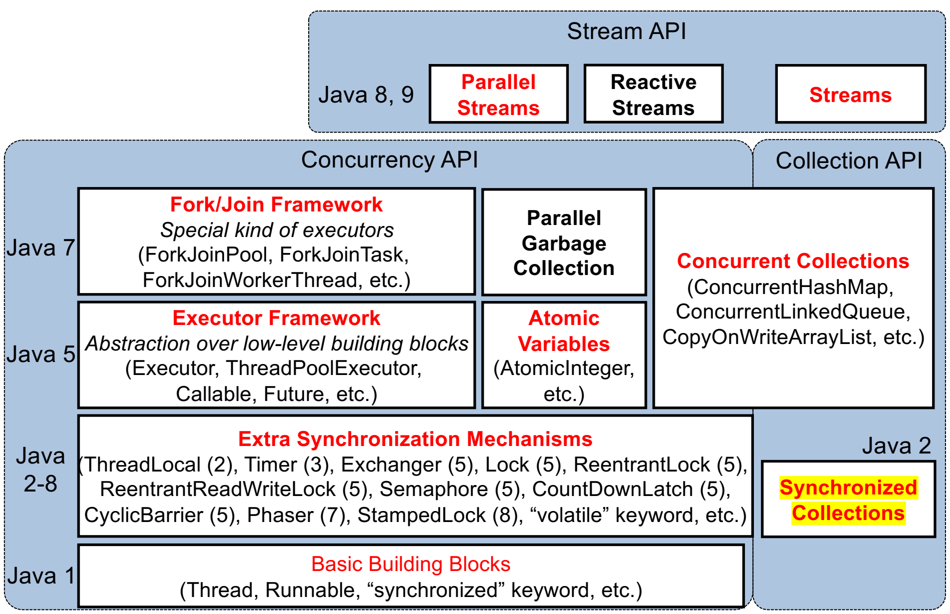
```
ReentrantLock lock = new ...;
lock.lock();
if(map.containsKey("foo"){
  map.remove("foo");  }
lock.unlock();
```

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

3 Types of Collections in Java

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

Concurrency API in Java



Synchronized Collections

- "Ready-made" thread-safe collections
  - Synchronized classes: `Vector` and `Hashtable`
    - All public methods perform thread synchronization.
      - 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()`

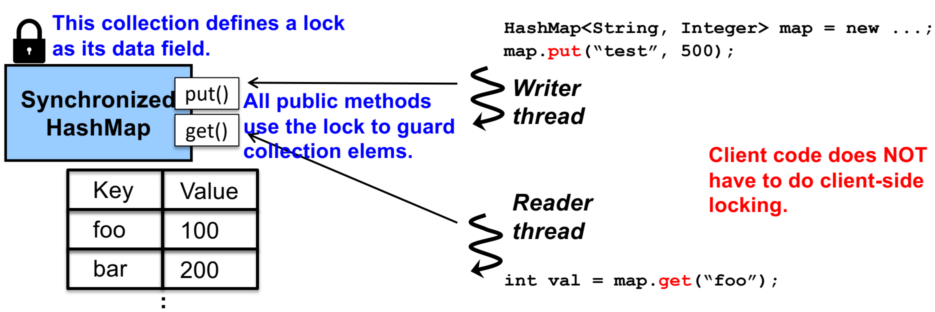
`Vector` and `Hashtable` in Single Threaded Programs

- It makes no sense to use these collections in single-threaded programs in a performance point of view.
  - They perform thread synchronization even when only one thread runs in a program.
    - Unnecessary performance loss
- Use **`ArrayList`** and **`HashMap`** instead!
  - They still exist in Java API only for backward compatibility.

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.



- Need *client-side locking* in *compound operations* on a synchronized collection
  - Iteration (element-traversal)
    - Get all elements one by one
  - 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());   // acquires the lock that the list owns
  lock.unlock();
  ```
- `lock` is different from the lock that the `list` owns 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){              // acquires the lock that the list owns
    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() );      // acquires the lock that the list owns
  lock.unlock();
  ```
- Performance penalty due to 2 lock acquisitions
  ```

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

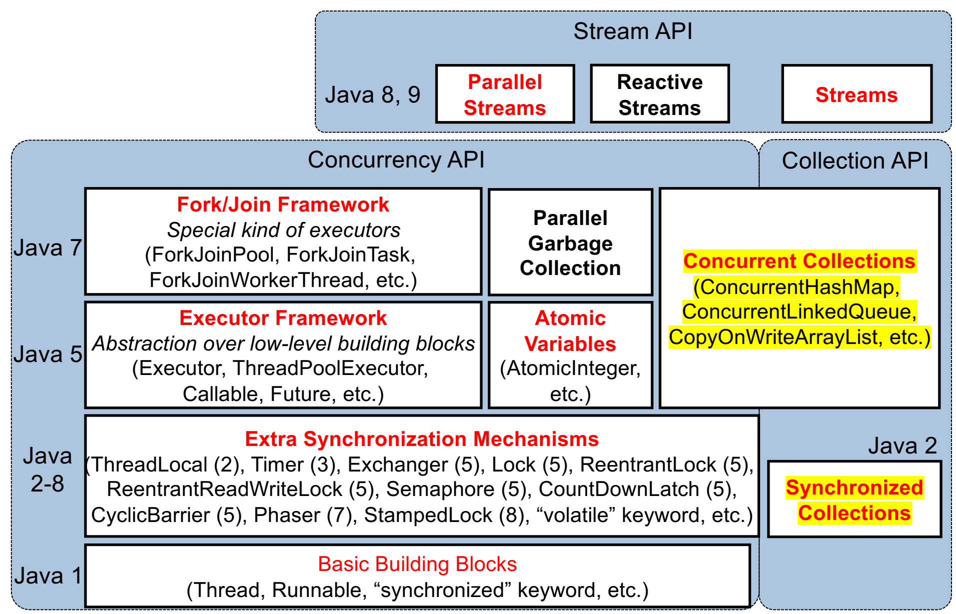
- Performs client-side locking for **`ArrayList`**, which is NOT thread-safe.
  - The client code is thread-safe although **`hasNext()`** and **`next()`** are not.
- More efficient than the previous client code.
  - Only 1 lock acquisition (not 2 acquisitions)
  - Use a read-write lock if you have many “reader” threads.

Summary: Thread-unsafe Collections v.s. Synchronized Collections

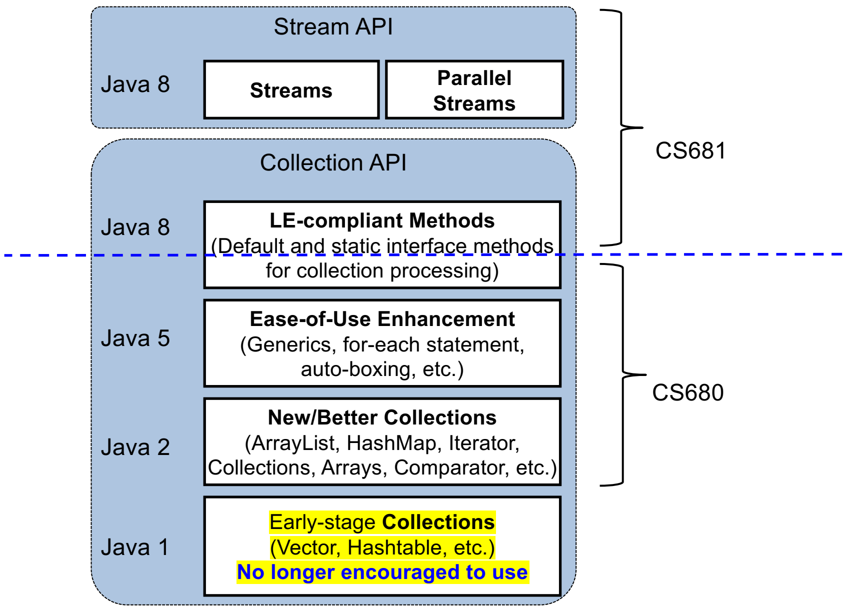
- Thread unsafe collections
  - Client code always must perform thread synchronization.
    - For both simple public method calls and compound operations
    - Client-side locking always requires 1 lock acquisition.
- Synchronized collections
  - Client code does NOT have to perform thread synchronization for simple public method calls.
  - However, it needs client-side locking for compound operations.
    - Client-side locking requires 2 lock acquisitions.

- Thread unsafe collections
  - Use them consistently.
- Synchronized collections
  - Forget about them.
  - Never use **`Vector`** and **`Hashtable`** in single-threaded programs

Concurrency API in Java



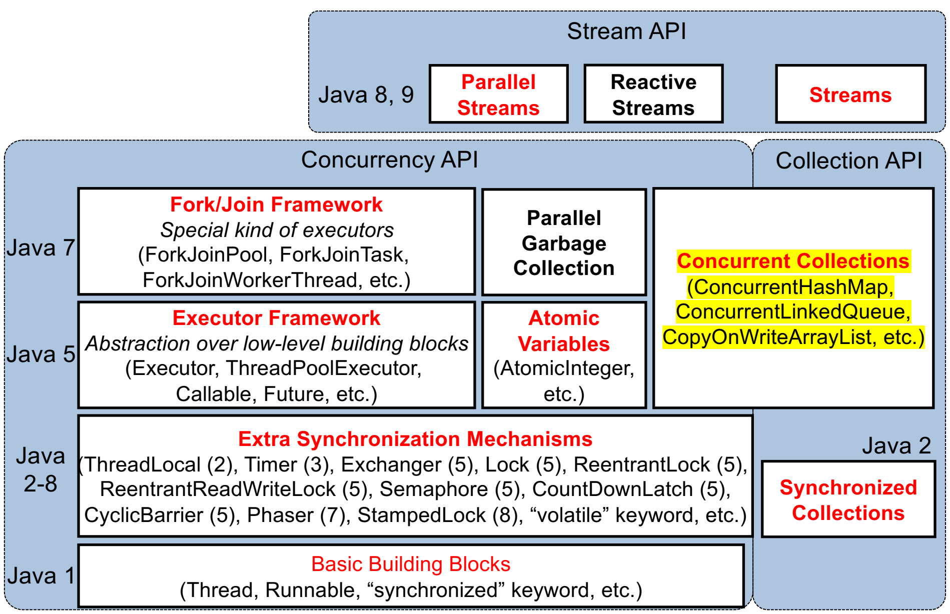
Collection API in Java



3 Types of Collections in Java

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

Concurrency API in Java



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`

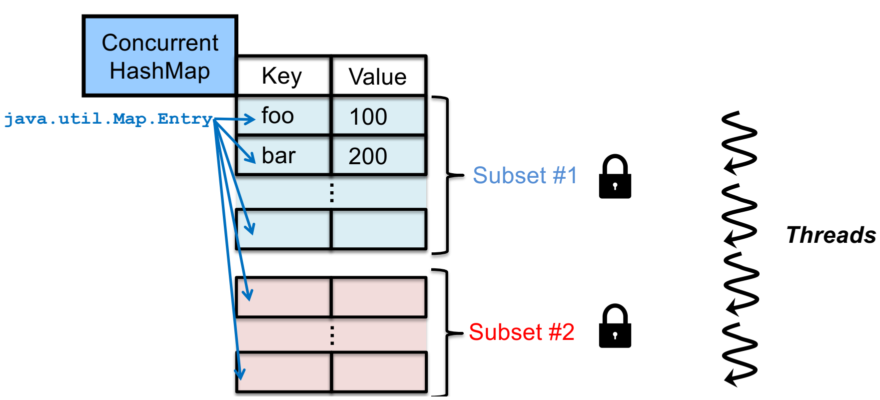
- Indented to replace synchronized collections.
  - e.g., `ConcurrentHashMap` is indented to replace `SynchronizedMap`.
- Make public methods thread-safe.
  - e.g., `get()` and `put()` in `ConcurrentHashMap`
  - Client code does NOT have to perform thread synchronization for simple public method calls.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

ConcurrentHashMap

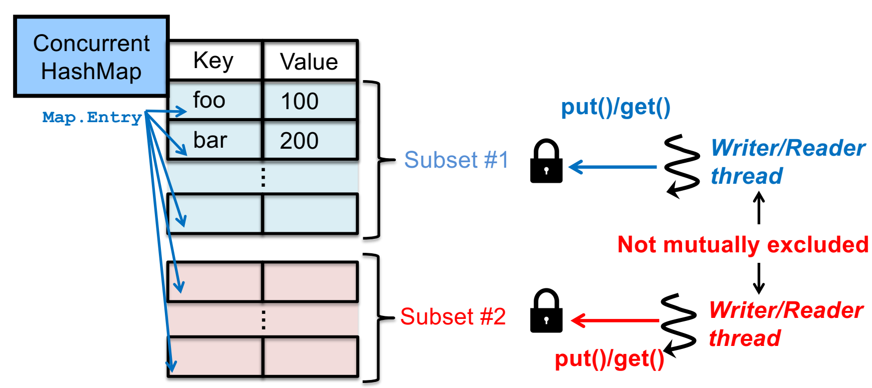
- Provides thread-safe public methods.
  - e.g., `get()` and `put()`
  - Client code does NOT have to do client-side locking.
- A replacement for synchronized hash-based `Map` implementations (e.g., `Hashtable` and synchronized `HashMap`).
- Implements public methods in an efficient manner.
  - Performs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

Lock Stripping

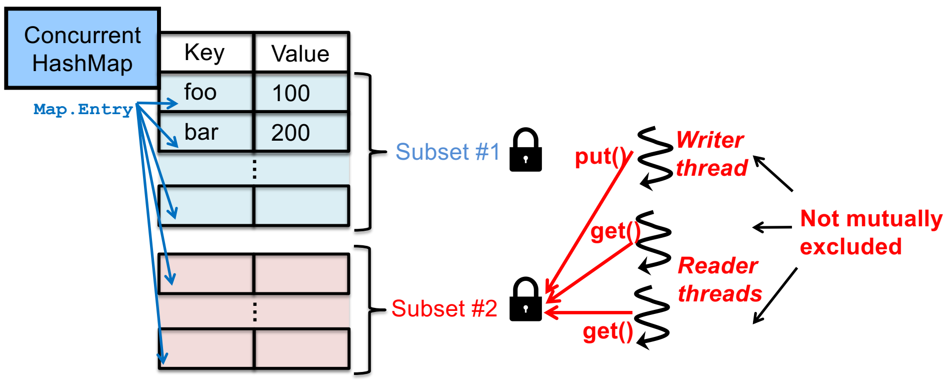
- `ConcurrentHashMap` uses *multiple* locks to guard a table (i.e., key-value pairs).
  - 16 locks by default
    - Configurable with the “concurrencyLevel” parameter in a constructor.
  - Each lock is associated with a subset of the table.



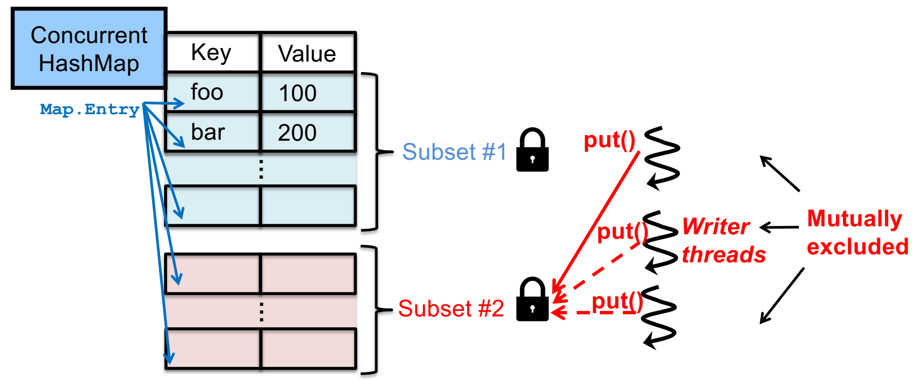
- Threads are not mutually excluded with each other
  - as far as they access different subsets of the table with different locks.



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

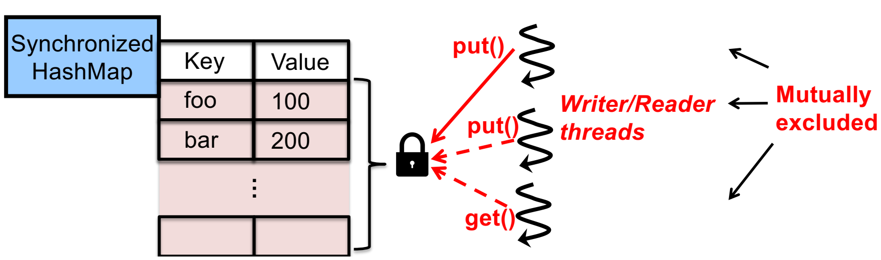


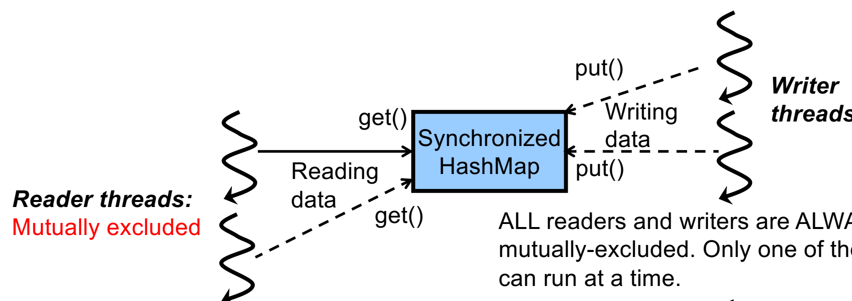
- To access a subset of the table,
  - Writer threads ARE mutually excluded with each other.



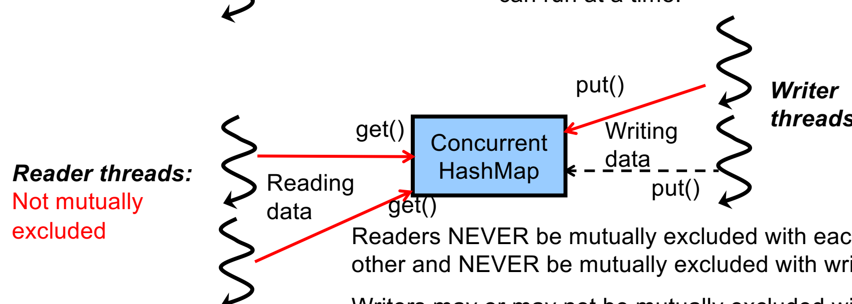
Synchronized Hash-based Map Impls

- A *single* lock is used to guard the *entire* table in `Hashtable` and synchronized `HashMap`
  - No lock stripping.
- All writer/reader threads ARE ALWAYS 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.

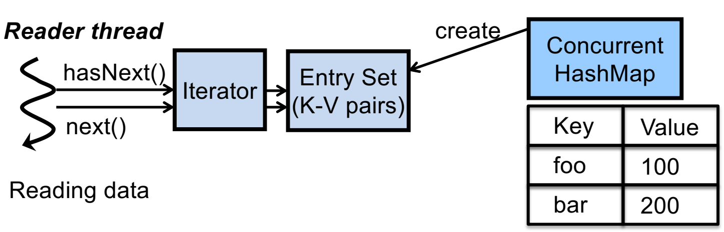
You, as a collection user, cannot tell if the writers access the same subset of the table.

Thread-safe Compound Operations

- Supports thread-safe iteration
  - No client-side locking is necessary.
    - c.f. Thread-unsafe collections and synchronized collections require client-side locking for iteration
      - Because it is a compound operation.

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

- Pros
  - No client-side locking is necessary.
    - No need to mutually exclude readers.



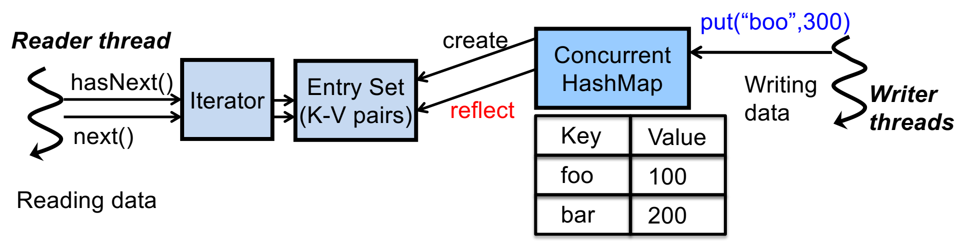
Concurrent Iterators

- 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() );
```

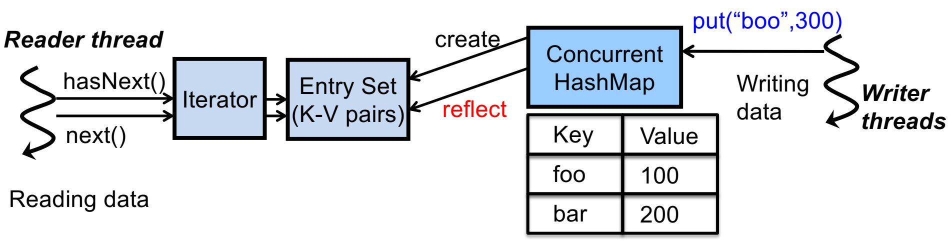
```
Iterator it = aConcurrentHashMap.entrySet().iterator();
while( it.hasNext() )
  doSomething( it.next() );
```

- Pros
  - No client-side locking is necessary in client code.
    - No need to mutually exclude readers.
    - No need to mutually exclude readers and writers.
      - » 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.

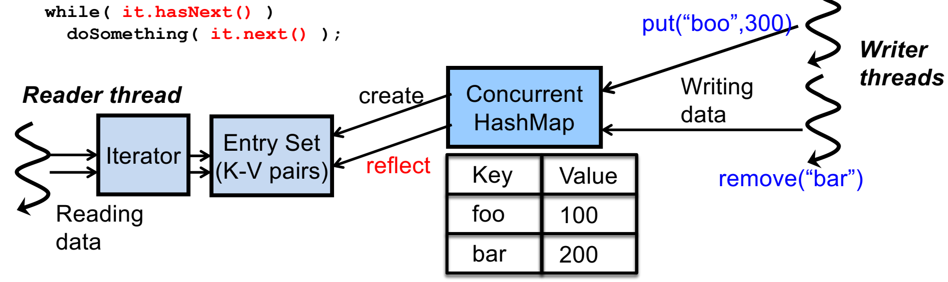


```
Iterator it = aConcurrentHashMap.entrySet().iterator();
while( it.hasNext() )
  doSomething( it.next() );
```

– Cons:
  - *Weak (or best-effort)* consistency
    - There is no guarantee about how soon a change (on 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.




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



- The iterator "it" may traverse
  - {(foo,100), (bar,200)},
  - {(foo,100), (bar,200), (boo,300)},
  - {(foo,100)}, or
  - {(foo,100), (boo,300)}.
- It is guaranteed that elements are never get corrupted.
  - Corrupted data like (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` does not implement perfect consistency.

- `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 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 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`

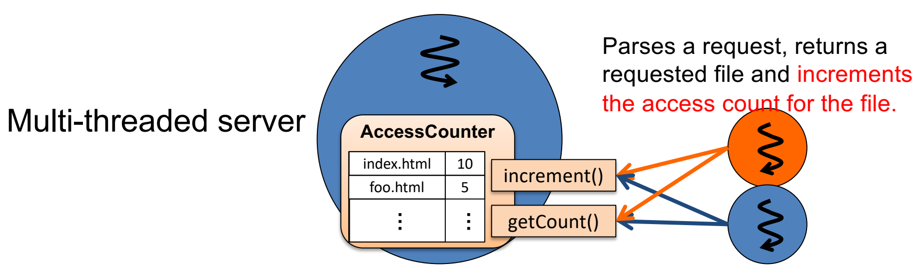
- ```
  public interface Iterable<T>{
      Iterator<T> iterator();
      Spliterator<T> spliterator();
      ... }
  ```
- `java.util.Spliterator<E>`
  - "Split" + "iterator"
  - Can *split* (or subset) the entire set of elements and traverse a subset of the elements.
    - ```
      Iterator it = concurrentHashMap.keySet().iterator();
      it.forEachRemaining( (String k)
                           ->{System.out.println(k);} );
      ```
    - ```
      Spliterator st1 = concurrentHashMap.keySet().spliterator();
      Spliterator st2 = st1.trySplit();
      st1.forEachRemaining( (String k)
                           ->{System.out.println(k);} );
      st2.forEachRemaining( (String k)
                           ->{System.out.println(k);} );
      ```
    - `st1` and `st2` cover two different subsets.

Other Thread-safe Compound Operations

- Supports common *compound operations* in a thread-safe manner
  - 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

Exercise: Concurrent Access Counter

- `AccessCounter`
  - c.f. HW 11 (w/ `HashMap` and `ReentrantLock`)
  - c.f. HW 13 (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.



- Revise the access counter with `ConcurrentHashMap`
  - `increment()`
    - ```
      Integer oldCount = concurrentMap.get(aFilePath);
      int newCount = oldCount==null? 1: oldCount+1;
      concurrentMap.put(aFileName, newCount);
      ```
    - This client code is NOT thread-safe.
      - because it performs a compound operation.
      - although `get()` and `put()` are thread-safe.
      - Need client-side locking with a `ReentrantLock`
- Alternatively, use `putIfAbsent()` and `AtomicInteger`
  - ```
    ConcurrentHashMap<Path, AtomicInteger> map = new ...;
    map.putIfAbsent(aFilePath, new AtomicInteger(0));
    map.get(aFileName).incrementAndGet();
    ```

Thread-safe Methods that Accept Lambda Expressions

- `compute(K, BiFunction<K,V,V>)`
  - Accepts a key (`K`) and a lambda expression (LE)
  - Applies the LE on a key-value pair that is associated with the supplied key (`K`).
  - ```
    map.compute(aFilePath, (Path k, Integer v)->
                                 {return v==null? 1: ++v;})
    ```
    - If a key-value pair that is associated with `K` does not exist, `null` is passed to the 2nd parameter of the lambda expression.
- `computeIfAbsent(K, Function<K,V>)`
  - `map.computeIfAbsent(aFilePath, (Path k)->{return 1;})`
- `computeIfPresent(K, BiFunction<K,V,V>)`
  - ```
    map.computeIfPresent(aFilePath, (Path k, int v)->
                                     {return ++v;})
    ```

Important General-Purpose Functional Interfaces

	Params	Returns	Example use case
Function<T,R>	T	R	Get the price (R) from a Car object (T) Generate a function (R) from another (T)
Consumer<T>	T	void	Print out a collection element (T)
Predicate<T>	T	boolean	Has this car (T) had an accident?
Supplier<T>	NO	T	A factory method. Create a Car object and return it.
UnaryOperator<T>	T	T	Logical NOT (!)
BinaryOperator<T>	T, T	T	Multiplying two numbers (*)
BiFunction<U,T>	U, T	R	Return TRUE (R) if two params (U and T) match.

HW 17

- Revise your concurrent access counter (HW 11 solution) with `ConcurrentHashMap`
  - Eliminate client-locking (to guard a hash map) in `increment()` and `getCount()`
    - i.e., `AccessCounter` no longer needs a `ReentrantLock` data field.
  - Use lambda expressions whenever possible.

Bulk Operation Methods

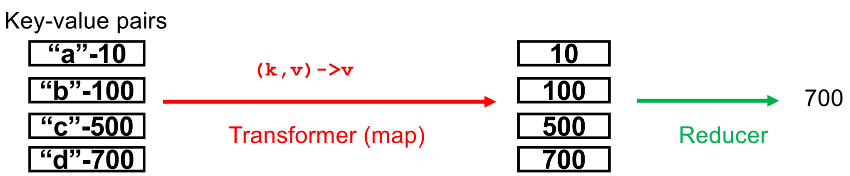
- Apply a given lambda expression on each key-value pair with multiple threads in a thread-safe manner.
  - *forEach*: `forEach()`, `forEachEntry()`, `forEachKey()`, `forEachValue()`
  - *Search*: `search()`, `searchEntries()`, `searchKeys()`, `searchValues()`
  - *Reduce*:
    - `reduce()`, `reduceToDouble()`, `reduceToInt()`, `reduceToLong()`
    - `reduceEntries()`, `reduceEntriesToDouble()`, `reduceEntriesToInt()`, `reduceEntriesToLong()`
    - `reduceKeys()`, `reduceKeysToDouble()`, `reduceKeysToInt()`, `reduceKeysToLong()`
    - `reduceValues()`, `reduceValuesToDouble()`, `reduceValuesToInt()`, `reduceValuesToLong()`

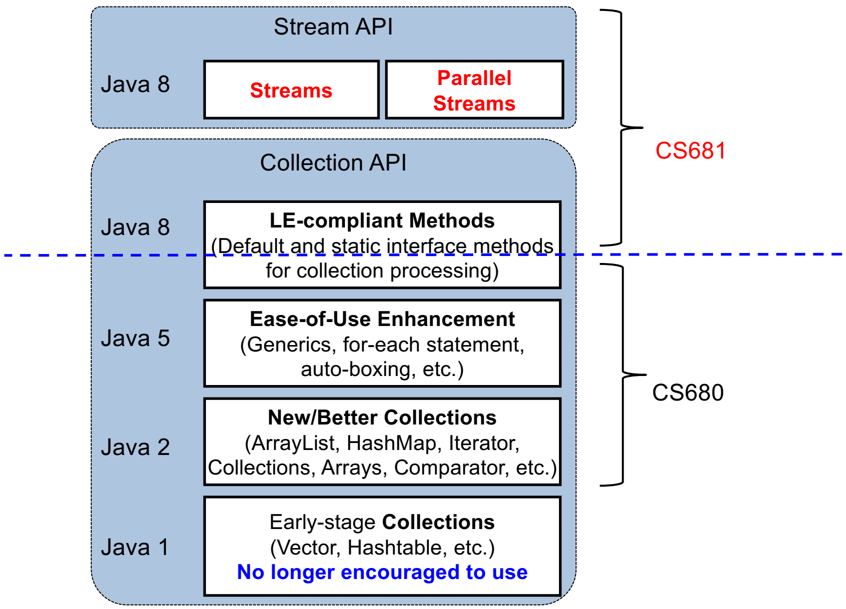
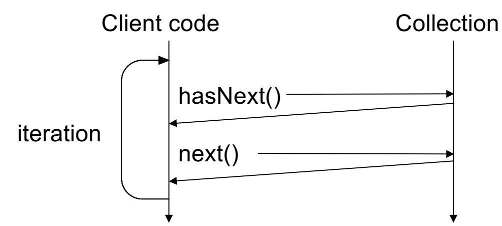
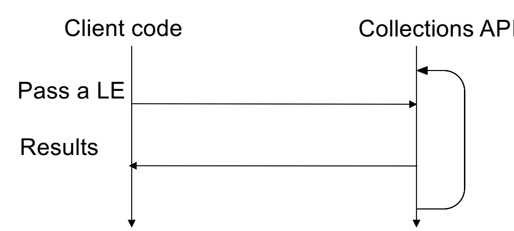
- Each bulk operation method receives a "parallelism threshold" as the first parameter and creates extra threads if necessary.
  - Executed with extra threads if the # of key-value pairs >= this threshold.
    - `Long.MAX_VALUE`: Suppress concurrency/parallelism. Use a single thread.
    - 1: Maximize concurrency/parallelism.
- The number of threads to be used:
  - If (# of key-value pairs) < threshold
    - No extra threads to be used
  - If (# of KV pairs) / threshold >= (# of CPU cores)
    - Use (# of CPU cores)
    - c.f. `Runtime.availableProcessors()`
  - If (# of KV pairs) / threshold < (# of CPU cores)
    - Use (# of KV pairs) / threshold
      - Rounded to an int number

`forEach` Bulk Operations

- Performs a given lambda expression (LE) on each key-value pair.
  - ```
    map.forEach(500,
                (k,v)->{if(v>10000)
                          System.out.println(k + "->" + v) )
    ```
    - 1st param: parallelism threshold (500)
    - 2nd param: lambda expression (`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 (specified as a LE)
  - Returns a non-null result if found. Returns null otherwise.
  - Skips further search when the first search fit is returned.
  - `Path path = map.search(500, (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.
  - `Integer count = map.searchValues(500, (v)->{v>10000? v: null} )`
    - Second param: search function (`Function`)
    - 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(500,
                                  (k,v)-> v,
                                  0,
                                  (max, v)-> v>max? v: max );
```

  - 2nd param: transformer (`BiFunction`)
    - Expected to work like a map operation
  - 4th param: reducer (`BinaryOperator`)
  - 3rd param: the initial value for accumulated value (`max`).




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

  - 4th param: reducer (`BinaryOperator`)
  - 3rd param: the initial value for accumulated value (`max`).

- Generalized form of reduce operations with the Streams API

```
– T result = aStream.reduce(initValue, (result, element)-> ... );

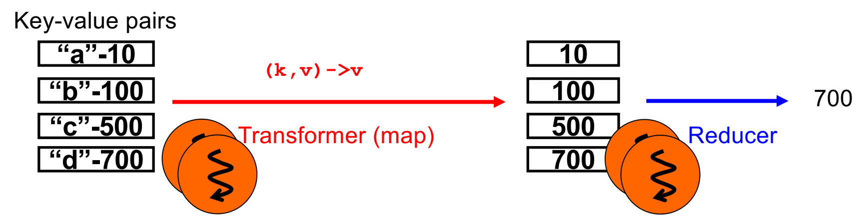
– T result = initValue;
  for(T element: collection){
      result = accumulate(result, element);   }
```

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 (weak consistency)
    - Write threads can modify key-value pairs when read threads are reading key-value pairs.
      – It is guaranteed that readers and writers do not corrupt key-value pairs.
      – c.f. Concurrent iteration

Recap: Collection Processing with LEs

- Java 8 made major improvements to the Collection API by
  – Adding new static and default methods in existing interfaces
    - e.g., `Iterable.forEach()`



Recap: External v.s. Internal Iteration

- ***External*** iteration: Iterate over a collection and performs an operation on each element in turn.
  - ```
    Iterator<ArrayList> iterator = strList.iterator();
    while( iterator.hasNext() ) {
     System.out.print(iterator.next()); }
    ```
- Iteration occurs outside of a collection.
- Need to write a boilerplate code whenever you need to iterate over the collection.



- The loop mixes up *what you want to do on a collection* and *how you do it*.
  - May not be that maintainable because "what" is often obscured by "how." ("How" is often emphasized too much than "what.")
  - ```
    Iterator<ArrayList> iterator = strList.iterator();
    while( iterator.hasNext() ) {
     System.out.print(iterator.next()); }
    ```
- Inherently serial
  - Hard to make it concurrent/parallel.

- ***Internal*** iteration:
  - ```
    int maxCount = map.reduceToInt(50,
                                   (k,v)->v,
                                   0,
                                   (max, v)-> v>max? v: max );
    ```
    - Does not return an `Iterator` that externally controls the iteration
    - Creates an equivalent object, which works inside of a collection.
      - A collection internally uses the iterator-like object to perform iteration


    ```

- Client code simply states "what" you want to do on a collection. "How" is hidden.
  - Collection processing looks more declarative, not procedural.
    - c.f. SQL statements

- `ConcurrentHashMap` *internally* uses threads to perform bulk operations.

```
– int maxCount = map.reduceToInt(2,
                                 (k,v)->v,
                                 0,
                                 (max, v)-> v>max? v: max);
```

- Client code does not have to *externally* use threads.



Benefits of Using Lambda Expressions

- Can make your code more concise (less repetitive)
- Can enjoy the power of functional programming
  - e.g., higher-order functions
- Can gain a new way to access collections
  - "Internal" iteration as opposed to traditional "external" iteration
    - e.g., Map-Reduce data processing
- Can simplify concurrent programming (multi-threading) in Java

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`.
  - c.f. lock stripping in `ConcurrentHashMap`
- Supports thread-safe, weakly-consistent iteration with an iterator and a spliterator(s)
  - just like `ConcurrentHashMap`
- Provides thread-safe methods for the other types of compound operations
  - i.e. Navigation and check-and-act operations
  - c.f. `putIfAbsent(key, value)` and `compute()` in `ConcurrentHashMap`
- Provides thread-safe methods for bulk operations.

Some Major Concurrent Collections

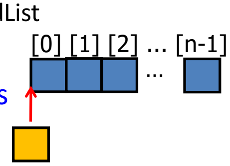
- `ConcurrentLinkedQueue`
  - Concurrent implementation of `Queue`
    - FIFO (First-In-First-Out) queue
- `ConcurrentLinkedDeque`
  - Concurrent implementation of `Deque`
    - LIFO (Last-In-First-Out) queue
- `ConcurrentSkipListMap`
  - An implementation of `ConcurrentNavigableMap`.
  - Map entries 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 sets 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.

Concurrent Array List???

- No concurrent collection class for lists in Java API
  - No such class like `ConcurrentArrayList`.
  - Why? Performance-wise, lists are not that great collections for multi-threaded programs
    - Insertion and removal are expensive for non-tail elements
      - c.f. Slides on List v.s. Queue and ArrayList v.s. LinkedList
    - Hard to implement index-based random access in a concurrent and efficient manner.
      - e.g., Weakly consistent iteration cannot be more efficient than perfectly consistent iteration, if insertion/removal often occurs for non-tail elements.

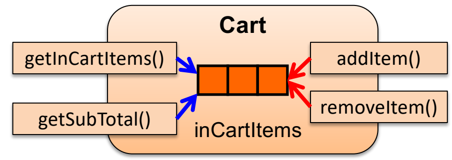


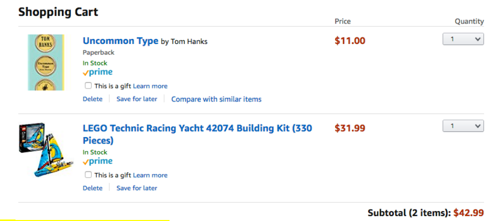
– If you need/want to use an `ArrayList`, you have to craft thread-safe client code for it yourself with client-side locking.

– Try a concurrent `Queue` class (e.g. `ConcurrentLinkedQueue`) or `Set` class (e.g. `ConcurrentHashMap.KeySetView<K, Boolean>`) if you like to take advantages of concurrent collections.

– FYI: `CopyOnWriteArrayList`
  - Concurrent collection class for ArrayList
  - Thread-safe, but not that efficient in general (to be explained)
    – Can be efficient only in limited use cases

Exercise: Online Shopping Cart

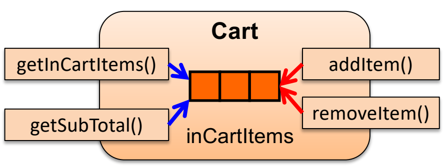


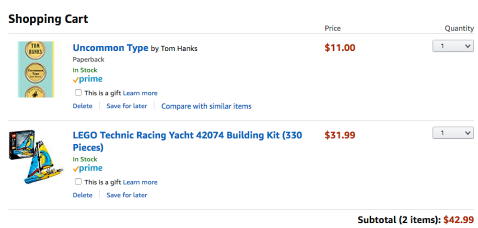


- Can use `ConcurrentLinkedQueue<Product>` for `inCartItems`
  – It is originally typed with `LinkedList`

- Thanks to `ConcurrentLinkedQueue`, all methods of `Cart` do not need thread synchronization to access `inCartItems`.

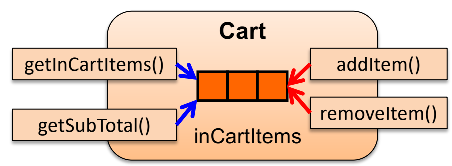
-
```
addItem(Product item){
  inCartItems.offer(item); }
  // No thread synch!
```
```
getInCartItems(){
    return inCartItems; }
  // No thread synch!
```

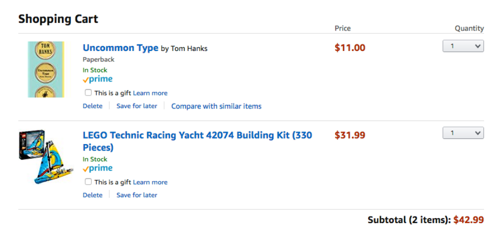




- Thanks to `ConcurrentLinkedQueue`, all methods of `Cart` do not need thread synchronization to access `inCartItems`.

-
```
getSubTotal(){
  float subtotal;
  for(Product item: inCarItems){
    subtotal += item.getPrice(); }
  return subtotal; }}
  // No thread synch!
  // Weakly consistent iteration
```



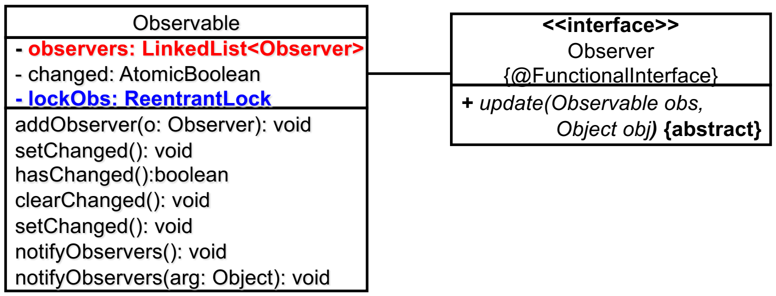


- `removeItem()` used an index number for a `LinkedList`.
  – `public void removeItem(int productIndex)`
  – Need to change the method signature
    - `public void removeItem(Product item)`
      `public void removeItem(String productId)`

-
```
removeItem(Product item){
  inCartItems.remove(item);}
  // No thread synch!
```
```
removeItem(String productId){
  inCartItems.removeIf(
    (Product p)->
      p.getID().equals(productid))}
  // No thread synch!
```

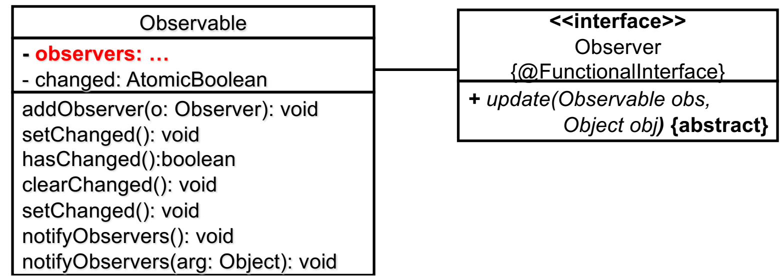
Recap: Concurrent Observer

- C.f. HW 16
  - **lockObs** is used to guard **observers** (a **LinkedList**).



HW 18

- Revise your HW 16 code to replace `LinkedList` with
  - a concurrent queue, or
    - `ConcurrentLinkedQueue`
  - a concurrent hash set
    - `ConcurrentHashMap.KeySetView<K, Boolean>`
  - Make sure that `Observable` no longer needs a `ReentrantLock` data field to guard **observers**.



Appendix: Copy-On-Write (COW) Collections

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`

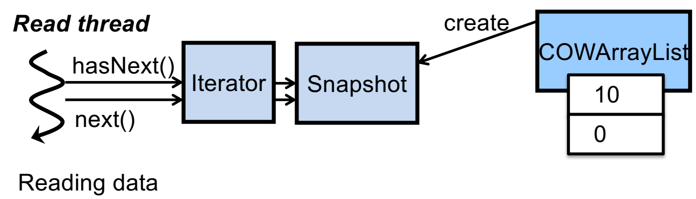
Copy-On-Write (COW) Collections

- `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

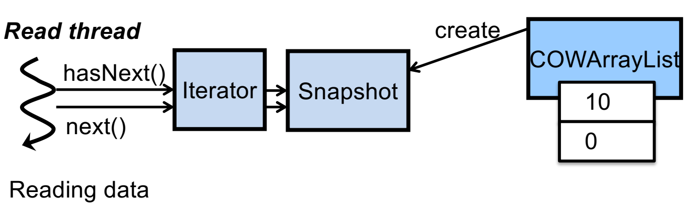
- Support thread-safe, snapshot-based iteration.
  - A read thread references and operates on a *collection snapshot*, which is a collection that is up-to-date when an iterator is created.

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

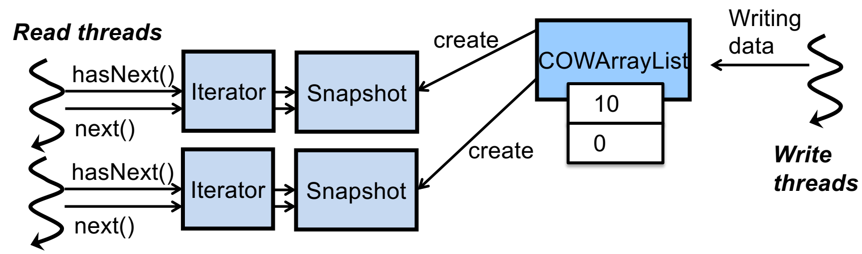


- Pros
  - No client-side locking is necessary for iteration.
    - Read threads ARE NOT mutually excluded with each other and with writer threads.
      - Each iterator has a thread-specific snapshot of List elements.
      - Different readers get different snapshots and access them concurrently.

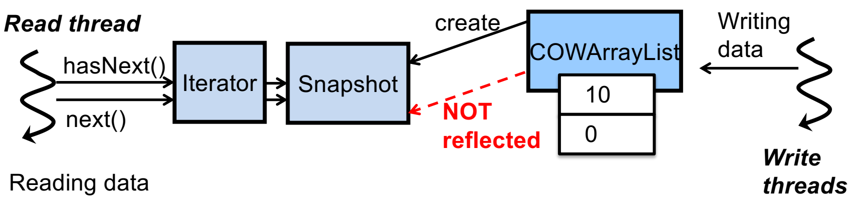
```
Iterator it = aCOWArrayList.iterator(); // A snapshot is created.
  while( it.hasNext() )
    doSomething( it.next() );
```



- 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.

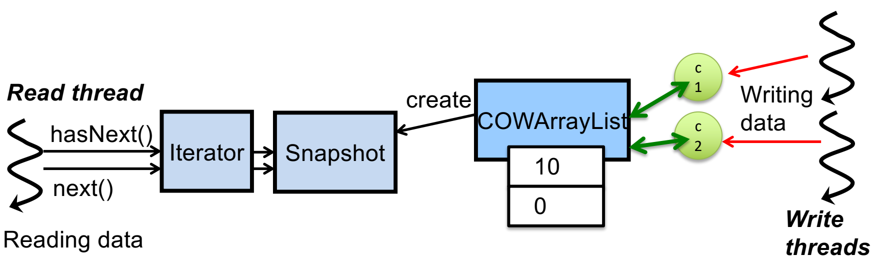


- Cons
  - The snapshot can become outdated.
    - e.g., when a write thread adds/removes collection elements after a snapshot is created.
  - No consistency preserved; Each iterator will NOT reflect additions, removals and other changes to the list after the iterator was created.
    - c.f. weak consistency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

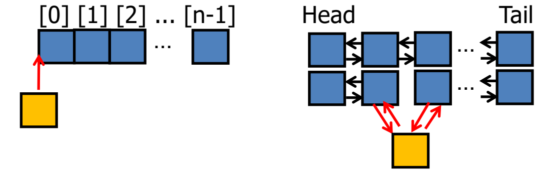


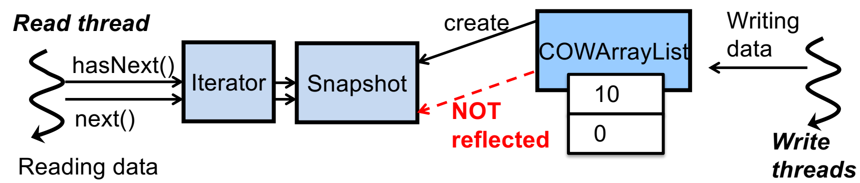
Copy-On-Write (COW)

- 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.

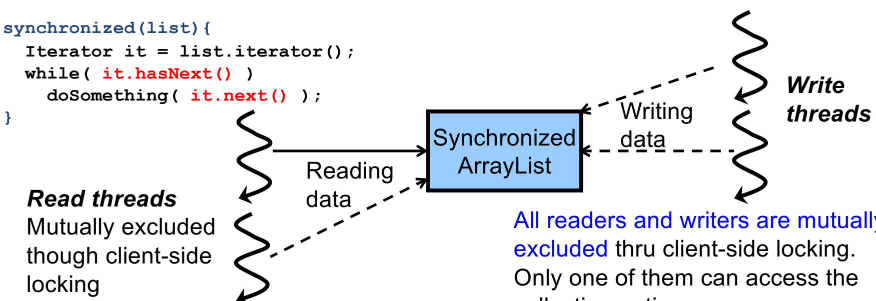


- Cons
  - No consistency preserved. Why?
  - State updates (particularly, element insertion and removal) are often expensive for lists.
    - Due to index-based element ordering



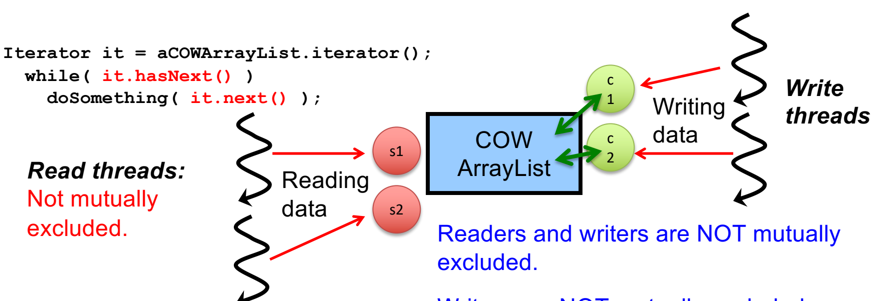



```
synchronized(list){
  Iterator it = list.iterator();
  while( it.hasNext() )
    doSomething( it.next() );
}
```



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

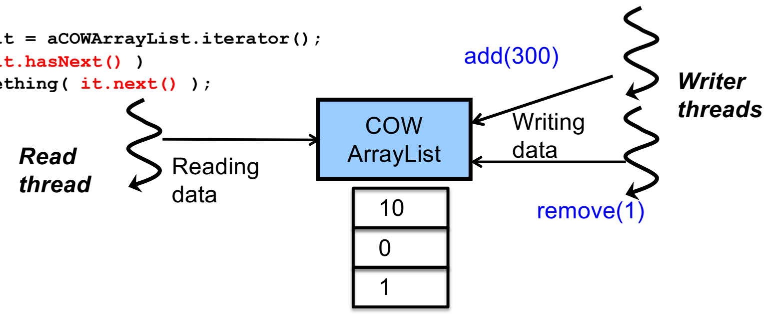
```
Iterator it = aCOWArrayList.iterator();
  while( it.hasNext() )
    doSomething( it.next() );
```



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.