Reminder: Collections Framework, I

- **Java Collections Framework** is set of type-parameterized classes & interfaces for storing groups of objects

  - 14 interfaces
  - Many classes that implement them

- Each class is a data structure

- E.g., set, list, hash table, queue

- See `http://docs.oracle.com/javase/8/docs/technotes/guides/collections/index.html`

Reminder: Collections Framework, II

Interface / Data structure	Set	List	Deque	Map
Closed Hash	HashSet			HashMap
Array		ArrayList	ArrayDeque	
Balanced Tree	TreeSet			TreeMap
Linked List		LinkedList	LinkedList	
Open Hash	LinkedHashSet			LinkedHashMap

Aside: History

- Early (pre Java 1.2) collections are thread-safe – e.g., `Vector`, `Hashtable`

- Starting with version 1.2, new collection classes were NOT thread-safe

- Removal of thread-safety was a controversial decision

- Consequently, with version 1.5, some concurrent collection classes were added

Aside: java.util

As a result of above history, there is no single convenient place to find all concurrent collection classes:

- Package `java.util` contains all collection classes
- Package `java.util.concurrent` contains some, but not all, thread-safe collection classes

Thread-Safe Collection Classes

- In `java.util.concurrent`:

`ArrayBlockingQueue`, `ConcurrentHashMap`, `ConcurrentLinkedQueue`, `ConcurrentSkipListMap`, `ConcurrentSkipListSet`, `CopyOnWriteArrayList`, `CopyOnWriteArraySet`, `LinkedBlockingDeque`, `LinkedBlockingQueue`, `PriorityBlockingQueue`

- In `java.util` but not in `java.util.concurrent`:

`Vector`, `Hashtable`

- Other collections outside `java.util.concurrent` (e.g., `ArrayList`) are probably NOT thread-safe

- Important qualification: many (but not all) non-thread-safe collections have corresponding thread-safe counterparts

Aside: Blocking Queues

All “blocking queue” classes solve the bounded buffer producer-consumer problem automatically, without need for condition variables or wait/signal

Thread-Safe Counterparts, I

- Static "wrapper" methods in `Collections` class create thread-safe counterparts of non-thread-safe collection classes

- Example: `HashSet` is non-thread-safe collection class

- To make a thread-safe set based on `HashSet`:

```
Set s = Collections.synchronizedSet(new HashSet(...));
```

Thread-Safe Counterparts, II

Q: What's going on here?

A: Simple: "synchronized set" just wraps each `HashSet` method inside a `synchronized` block

Thread-Safe Counterparts, III

Wrapper methods in `Collections` class:

```
Collection<T>  synchronizedCollection(Collection<T> c)

List<T>  synchronizedList(List<T> list)

Map<K,V>  synchronizedMap(Map<K,V> m)

Set<T>  synchronizedSet(Set<T> s)

SortedMap<K,V>  synchronizedSortedMap(SortedMap<K,V> m)

SortedSet<T>  synchronizedSortedSet(SortedSet<T> s)
```

Thread-Safe Counterparts, IV

- Notice: only `List`, `Set`, and `Map` interfaces have synchronized counterparts — `Queue` does not

- But there are many classes for synchronized queues:

`LinkedBlockingQueue`

`ArrayBlockingQueue`

`LinkedBlockingDeque`

`PriorityBlockingQueue`

`DelayQueue`

- Remember ... `SynchronousQueue` is not really a queue!

Why Not Make All Collections Thread-Safe?

- To be "thread-safe" means: each method, by itself, is atomic

- Thread-safety does not help with compound operations

- Suppose thread-safe class has atomic methods A, B, C

- Suppose a method in your code requires atomic execution of 2 calls to A and 1 call to B

- In this case, your code must implement its own atomicity; e.g., with `Lock`

Why Not Make All Collections Thread-Safe?

- Because:

  1. Thread-safe collection classes help in only SOME synchronization situations

  2. But they impose overhead (locking) ALL THE TIME

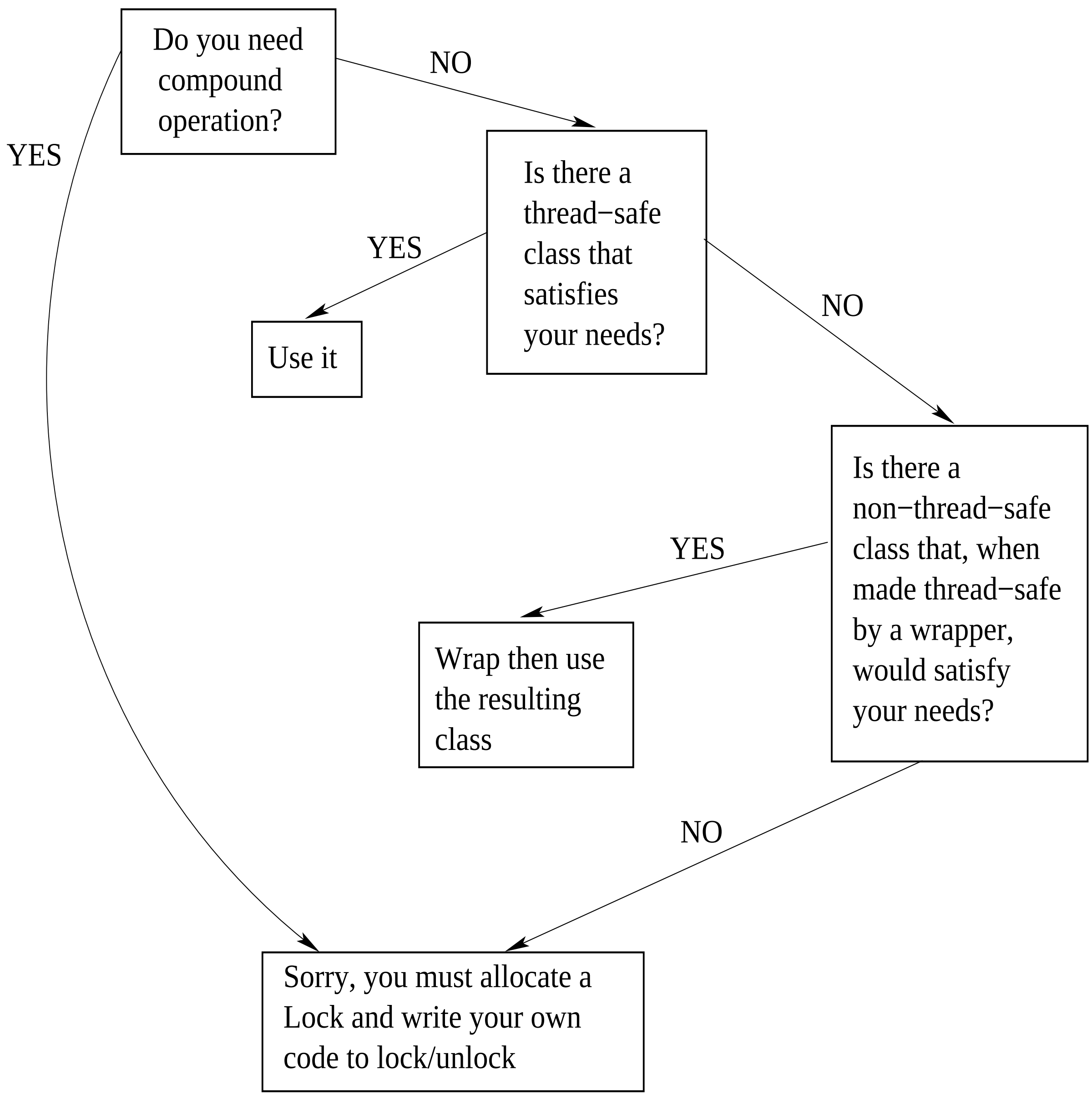
- Java library designers decided that cost (#2) outweighed benefit (#1)

- Why decision was controversial:

  1. Cost of locking overblown — just a few instructions (compare to Java's object instantiation overhead, for example)

  2. Writing correct locking code is hard, and, with this decision, must be done in every program that needs an atomic method of any collection class

How To Use All This Stuff, Depiction



How To Use All This Stuff, I

- Java library presents so many choices — it makes your head hurt!

- Follow this decision tree:

1. Does your program require atomic use of *a compound of more than one method* of some library class(es)?

Yes: you must implement atomicity yourself (e.g., with `synchronized` or `Lock`). Use whatever library classes you prefer.

No (meaning you need only atomic invocations of individual methods): there might be a library class for you. Go to step 2.

How To Use All This Stuff, II

2. Is there are a class in `java.util.concurrent` or a thread-safe class in `java.util` that satisfies your needs?

Yes: use it.

No: go to step 3.

3. Is there a class that implements the `List`, `Set`, or `Map` interface that would satisfy you, if only the class were thread-safe?

Yes: create the class you need by calling

`Collections.synchronizedList`, or

`Collections.synchronizedSet`, or

`Collections.synchronizedMap`, etc.

No: sorry, you must implement atomicity yourself.

Reminder: Iterator, I

- All children of `Collection` are also children of its superinterface, `Iterable`

- `Iterable` defines (only one) method: “`Iterator<T> iterator()`”

- Therefore, all collections can create an `Iterator` object

- `Iterator` object knows how to traverse the collection

- `Iterator` traverses in defined order if there is such an order (e.g., list), otherwise in unspecified order (e.g., hash table)

Reminder: Iterator, II

How to use `Iterator` object on a list:

```
// List is subinterface of Collection,
// therefore has iterator method
List<E> list = ...  // initialized somehow
Iterator<E> it;

for (it = list.iterator(); it.hasNext(); ) {
    E listElement = it.next();
    // do something with listElement
}
```

or:

```
it = list.iterator();
while (it.hasNext()) {
    E listElement = it.next();
    // do something with listElement
}
```

Reminder: Iterator, III

- `Iterator` object has state indicating which list element is next

- `hasNext()` method returns `boolean` indicating whether there is a next element

- `next()` method returns next element

Iterators and Threads, I

- Must protect against

  - One thread iterating thru collection, while ...
  - Other thread adds to (or deletes from) collection

- Depending exactly how the race turns out, *ANYTHING* could happen: detected problem, undetected problem (e.g., iterator returns deleted element), null pointer reference, etc.

Iterators and Threads, II

- Iterators make “best effort” to detect changes in underlying collection then throw `ConcurrentModificationException`

- (This is called **fail fast** behavior ... as opposed to silently mis-behaving; e.g., returning a deleted element or missing an added element)

- Important: “best effort” means iterator implementation does what it can — you CANNOT depend on fail-fast iterator catching 100% of problematic executions

- To be thread-safe, must synchronize on collection object, NOT iterator object!

Iterators and Threads, III

- This code correctly get/drops object lock of `list`:

```
List<E> list = ...  // initialized somehow
Iterator<E> iter;

synchronized(list) {
    iter = list.iterator();
    while (iter.hasNext()) {
        E listElement = iter.next();
        // do something with listElement
    }
}
```

- This code INCORRECTLY get/drops object lock of ONLY the iterator object:

```
List<E> list = ...  // initialized somehow
Iterator<E> iter;

synchronized(iter) {
    iter = list.iterator();
    while (iter.hasNext()) {
        E listElement = iter.next();
        // do something with listElement
    }
}
```

Iterator of Concurrent Collection

Iterator of a concurrent collection is **weakly consistent**:

- Can tolerate concurrent modification
- Does not throw `ConcurrentModificationException`
- Iterator traverses all elements that existed at moment iterator was created
- Iterator may or may not traverse some elements that were added after moment it was created

Other Weakened Semantics

- Iterator is not only concept weakened in concurrent collections

- E.g., `size` method returns an "approximation"

- Likewise, `isEmpty` method permitted to return while a concurrent insert/delete is in progress

Q: Horrors! Why is such inexactness permitted?!

A: Because methods like `size` and `isEmpty`, even if they could be exact, measure state that may have changed by time method's return value is *used*