JAVA PROGRAMMING BASICS

Module 2: Java Object-oriented Programming

Training program

Module contents

Module contents

Atomic operations

- **Atomic operations** are operations that can be performed for atomic variables in a multithreaded program without blocking synchronization.
- Such operations are based on the **Compare And Swap [CAS] algorithm**, the idea of which is that instead of blocking and mutual exclusion of access of several threads to a resource (***pessimistic approach***), it is not prohibited for several threads to get simultaneous access to a resource (blocking is absent) in the hope that there will be no mutual influence (***optimistic approach***).
- If such an influence is found, then the last thread will know about it and will not be able to change the resource, but will try to do it later.
- Atomic variables provide ***memory visibility*** for multiple threads like volatile variables, but with additional atomic operations.

Compare And Swap (CAS) algorithm

There are 3 parameters for a CAS operation:

- a memory cell with value V that has to be replaced;
- old value A (in thread local variable) which was read from cell by thread last time;
- new value B (in thread local variable) which should be written by thread over V;

1) Suppose first V = 10 and there are threads 1 and 2 that want to read and increment the values in the memory cell V:

   $V = 10, A_1 = 0, B_1 = 0, A_2 = 0, B_2 = 0$

2) Threads 1 and 2 want to increase the value of V, they both read the value:

   $V = 10, A_1 = 10, B_1 = 0, A_2 = 10, B_2 = 0$

3) Threads 1 and 2 increase the read value by 1 in their local variables (also remembering the previous values):

   $V = 10, A_1 = 10, B_1 = 11, A_2 = 10, B_2 = 11$

…

Compare And Swap (CAS) algorithm

…

4) Let thread 1 access the memory cell first and **compare** the value of V with the last read value:

$V = 10, A_1 = 10, B_1 = 11, A_2 = 10, B_2 = 11$

as V = 10 <u>is equal</u> to $A_2 = 10$,
V will be **swapped** as 11:

$V = 11, A_1 = 11, B_1 = 11, A_2 = 10, B_2 = 11$

```
if  (A == V ) {
    V = B
} else  {
    operation failed
}
  return V
```

5) When thread 2 accesses a memory cell, it performs a similar operation:
In this case, V = 11 <u>is not equal</u> to $A_2 = 10$, so the value is not replaced and returns the current value of V = 11.
Thread 2 updates the last read value in A_2

$V = 11, A_1 = 11, B_1 = 11, A_2 = 11, B_2 = 11$

$V = 10, A_1 = 10, B_1 = 11, A_2 = 10, B_2 = 11$

…

Compare And Swap (CAS) algorithm

…

6) Now thread 2 will repeat the increment operation again with the values:

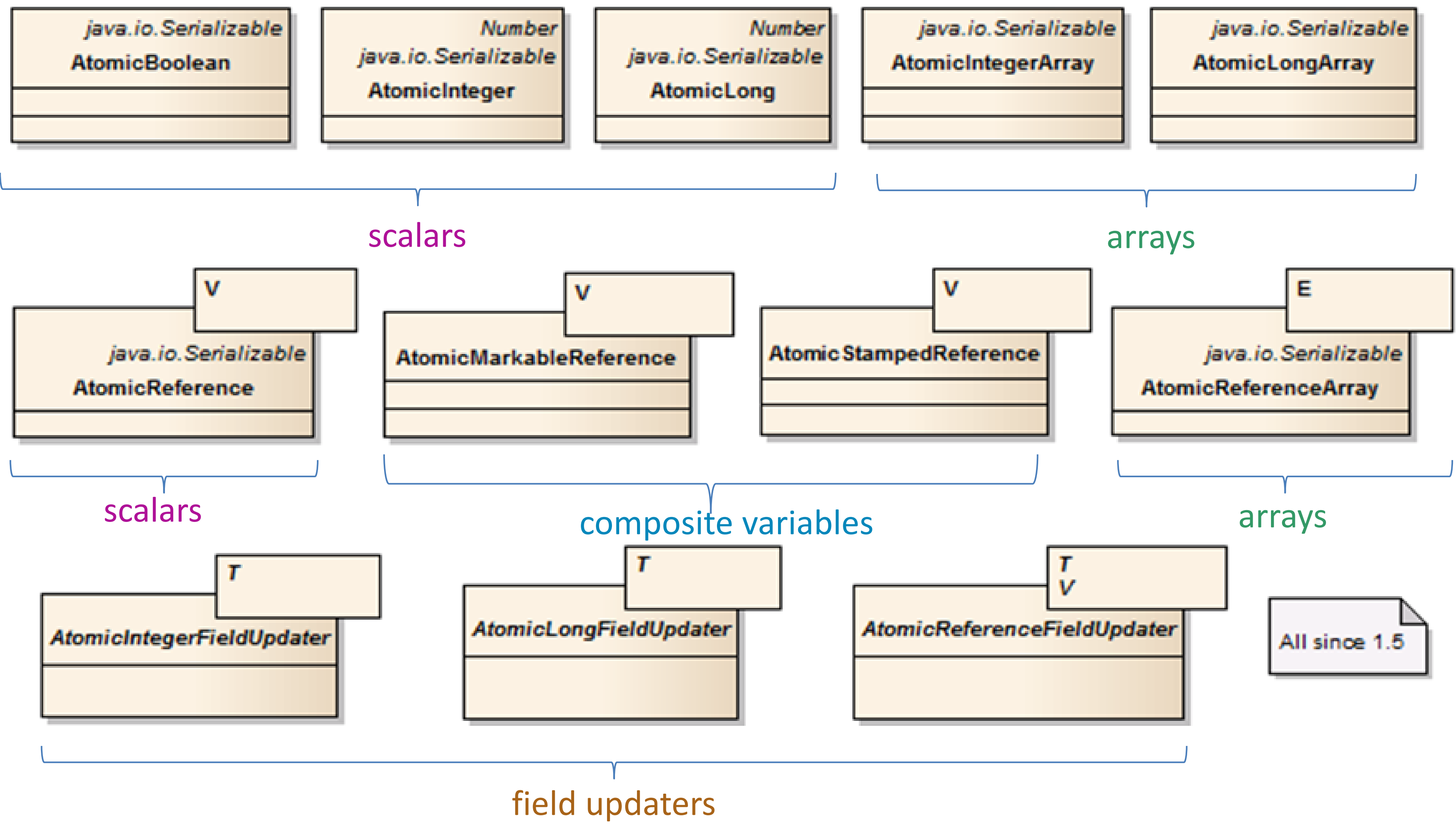
   $V = 11$, $A_1 = 11$, $B_1 = 11$, $A_2 = 11$, $B_2 = 12$

7) When thread 2 now has access to the cell and no other thread has changed its value during this time, it executes the CAS-algorithm, replaces the value of V with its incremental
   (because $A_2 = 11$ was equal to $V = 11$).
   New values will be:

   $V = 12$, $A_1 = 11$, $B_1 = 11$, $A_2 = 11$, $B_2 = 12$

```
if  (A == V ) {
    V = B
} else  {
    operation failed
}
  return V
```

Atomics java.util.concurrent.atomic



Since JDK 8: DoubleAccumulator, DoubleAdder, LongAccumulator, LongAdder were added.

See synchronizedcomparison

Atomics java.util.concurrent.atomic

Class	Description
AtomicBoolean	It is used to update boolean value atomically.
AtomicInteger*	It is used to update integer value atomically.
AtomicLong	It is used to update long value atomically.
AtomicReference<V>	An object reference that may be updated atomically.
AtomicIntegerArray	An int array in which elements may be updated atomically.
AtomicLongArray	A long array in which elements may be updated atomically.
AtomicReferenceArray<E>	An array of object references in which elements may be updated atomically.
AtomicIntegerFieldUpdater<T>	A reflection-based utility that enables atomic updates to designated **volatile** int fields of designated classes.
AtomicLongFieldUpdater<T>	A reflection-based utility that enables atomic updates to designated **volatile** long fields of designated classes.
AtomicReferenceFieldUpdater<T, V>	A reflection-based utility that enables atomic updates to designated **volatile** reference fields of designated classes.

* To simulate atomic variables of other primitive types, You can cast short or byte to and from int, and use the floatToIntBits or doubleToLongBits methods for floating point numbers

See atomicref

Atomics java.util.concurrent.atomic

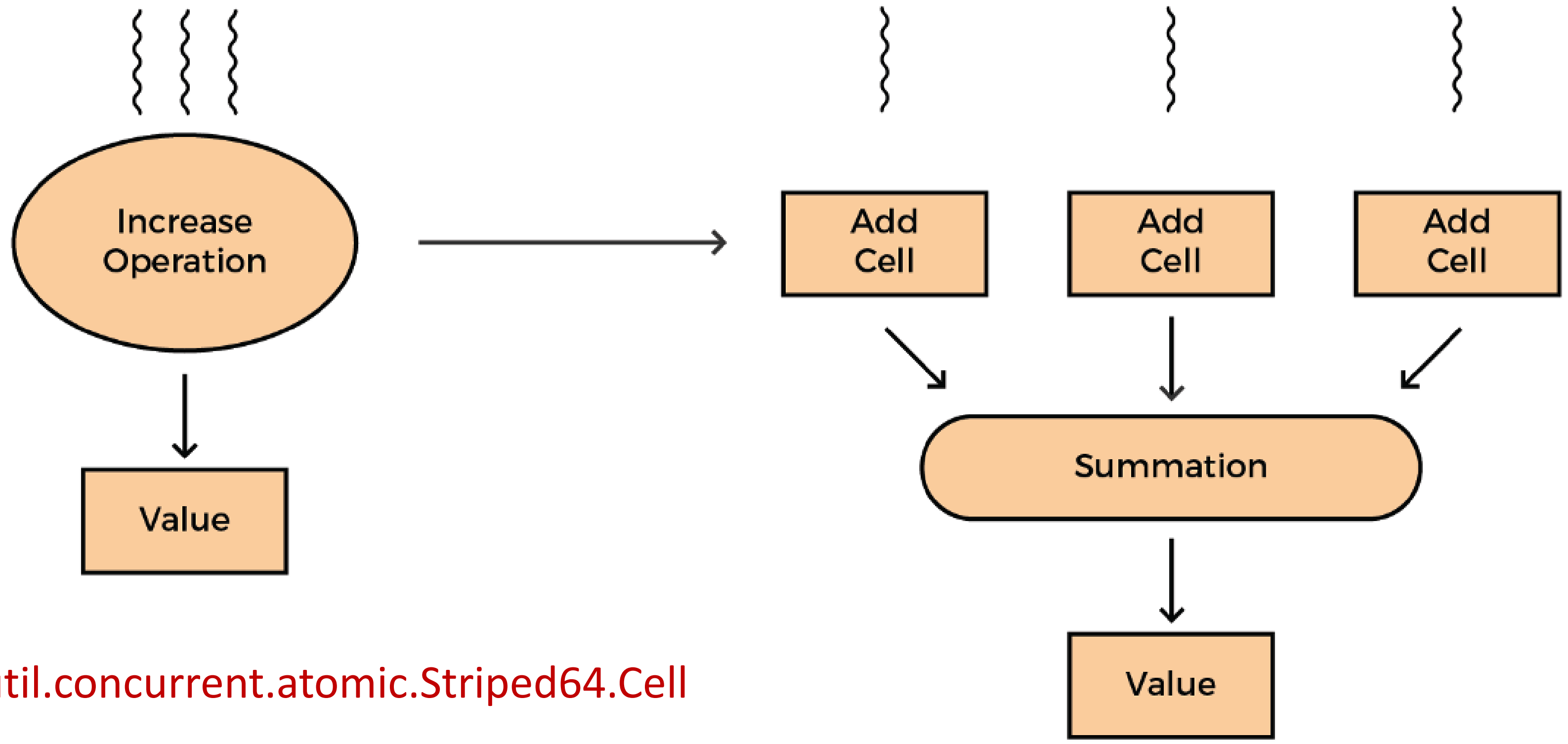
Class	Description
AtomicMarkableReference<V>*	An AtomicMarkableReference maintains an object reference along with a mark bit, that can be updated atomically.
AtomicStampedReference<V>*	An AtomicStampedReference maintains an object reference along with an integer "stamp", that can be updated atomically.
LongAdder**	This is alternative to AtomicLong can be used to consecutively add values to a number very often.
LongAccumulator**	This is a more generalized version of LongAdder. Instead of performing simple add operations the class LongAccumulator builds around a lambda expression of type LongBinaryOperator.
DoubleAdder**	It can be used to consecutively add double values to an initially zero double sum very often
DoubleAccumulator**	One or more variables that together maintain a running double value updated using a supplied function.

* ABA problem - for some algorithms (LinkedList eq.), changing V from A to B and then back to A still counts as a change that requires us to repeat some algorithmic step. Decision - instead of updating the link value, update the value pair, link and version number.

** see next slide

since JDK 8

Atomics



java.util.concurrent.atomic.Striped64.Cell

LongAdder, LongAccumulator, DoubleAdder, DoubleAccumulator
- when a shared atomic variable is repeatedly updated by several thread they spin until successful. An alternative is to have several variables (Cell instance) that maintain the updated number in a thread-safe manner rather than a single variable. As the number of threads increases, this underlying table of Cells keeps growing. After the final add, the **sum()** method can be invoked.

See adderaccumulator

Atomic variables common methods

Methods	Description
type **get**()	Gets the value from the memory, so that changes made by other threads are visible; equivalent to reading a volatile variable
void **set**(type newValue)	writes the value to memory, so that the change is visible to other threads; equivalent to writing a volatile variable
boolean **compareAndSet**(type expect, type update)	The method atomically sets the object to a new value if the current value is equal to the expected one, and returns true. If the current value changes and doesn't equal to expected, the method will return false and the new value will not be set
type **compareAndExchange**(type expect, type update)	The method atomically sets the object to a new value if the current value is equal to the expected one, and returns the current value, which will be the same as the expected value if successful

type is boolean or int or long or V

Atomic variables operations

AtomicInteger extends Number and **AtomicLong extends Number** have:

type getAndIncrement() - atomic increment of the current value and return of the old value (equivalent to the operation i++);

type incrementAndGet() - atomic increment of the current value and return of the old value after increase (equivalent to operation ++i);

type getAndDecrement() - atomic decrement of current value and return of old value (equivalent to operation i--);
type decrementAndGet() - atomic decrement of the current value and return of the old value after reduction (equivalent to the --i operation);
type addAndGet(type delta) - atomic addition of value-argument to the current one, returns a new value after addition;
type getAndAdd(type delta) - atomic addition of an argument value to the current one, returns the old value.

type is boolean or int or long or V

Atomics

AtomicIntegerArray, AtomicLongArray, AtomicReferenceArray<E> contain methods for working with array elements, similar to the methods of the classes AtomicInteger, AtomicLong, AtomicReference. The difference between these methods is in adding an additional argument that points to the index of the element in the array `i`, for example, **`boolean compareAndSet(int i, type expect, type update)`**, **`type getAndIncrement(int i)`**, etc.

...

See array

type is int or long or E

Atomics

…

AtomicIntegerFieldUpdater<T>, AtomicLongFieldUpdater<T>, AtomicReferenceFieldUpdater<T,V> contain methods for updating the values of object fields by their names using reflection, similar to the methods of the AtomicInteger, AtomicLong, AtomicReference. The difference between these methods is adding an additional argument indicating the **`obj`** object whose field is being updated, for example, **`boolean compareAndSet(T obj, type expect, type update)`**, **`type getAndIncrement method (T obj)`**, etc.

The **AtomicMarkableReference** class supports object references along with a tag bit that can be updated atomically.

The **AtomicStampedReference** class supports a reference to an object along with an integer "stamp" that can be updated atomically.

See fieldupdater

type is int or long or V

Module contents

Thread-safe collections - Immutable object

- An **immutable object** is an object with a very special characteristic. You can't modify its visible state (the value of its attributes) after its initialization. If you want to modify an immutable object, you have to create a new one.
- Its main advantage is that it is thread-safe. You can use it in concurrent applications without any thread-safe tools.
- An example of an immutable object is the `String` class in Java. When you assign a new value to a `String` object, you are creating a new string.
- **Volatile** keyword ensure atomic read and write operation for the field by several threads (besides long and double), but other operation can result in thread interference

Thread-safe Collections

By a ***thread-safe collection*** we mean a collection whose elements can be processed by concurrent threads without ever being in an inconsistent state. There are such types of thread-safe collections:

1) ***Unmodifiable collections*** cannot be changed structurally (add and remove operations are prohibited) and their elements cannot be replaced (throw an *UnsupportedOperationException*).

```
List<String> list = List.of ("Tom", "Dick", "Mary");
```

2) ***Unmodifiable views of collections*** are backed by an underlying collection, where changes in the underlying collection are reflected in the view. Such a view also cannot be modified, and only query methods are passed to the underlying collection.

```
List<Integer> list = new ArrayList<>(Arrays.asList(1, 2, 3));
List<Integer> immutableList =
                    Collections.unmodifiableList(list);   ...
```

Thread-safe Collections

…

3) The *Java Collections Framework* provides methods to create ***synchronized collections*** that are thread-safe. Actually, these collections are synchronized views of collections as they are backed which the getter and setter methods use synchronized blocks to delegate operations to the underlying collection.
4) The Java Concurrent API provides ***concurrent collections*** .

```
List<Integer> cwList = new CopyOnWriteArrayList<>
                                        (new Integer[]{1,2,3,4,5});
```

see collections/ThreadSafeCollections

Synchronized Collections

java.util.Collections has methods that return ***synchronized collections***:

- public static <T> Collection<T> **synchronizedCollection**(Collection<T> c),
- public static <T> List<T> **synchronizedList**(List<T> list),
- public static <T> Set<T> **synchronizedSet**(Set<T> s),
- public static <K,V> Map<K,V> **synchronizedMap**(Map<K,V> m),
- public static <T> SortedSet<T> **synchronizedSortedSet**(SortedSet<T> s),
- public static <K,V> SortedMap<K,V> **synchronizedSortedMap**(SortedMap<K,V> m)

Synchronized collections are static nested classes of the Collections class and have fields - a reference to the backed collection (parameter of the corresponding method from the above) and the **`Object mutex;`** whose monitor is used for synchronization.

see Collections.SynchronizedList API

Synchronized Collections issues

- Although getter and mutator methods are synchronized in a synchronized collection, this is not the case for methods that implement serial access (e.g., iteration and search). As multiple methods can be called for an operation on a synchronized collection when using the iterator, we need to ensure that these are executed as a single mutually exclusive operation. Iteration thus requires a ***coarse-grained manual synchronization at client side*** on the synchronized collection for deterministic results.
- Regardless of manually synchronizing on the synchronized collection any modification made ***directly on the underlying collection*** during serial access will result in the runtime java.util.ConcurrentModificationException.
- *A* ***compound operation*** that requires multiple operations on a synchronized collection **is not guaranteed to be thread-safe** just because the individual operations are mutually exclusive.
- Synchronous collections ***are not effective*** because of mutual exclusion of threads while it any of synchronized method of the list (they use common object-monitor).

Concurrent Collections

- The ***concurrent collections* provided** by the java.util.concurrent package use special-purpose locking mechanisms to enable multiple threads to operate concurrently, without explicit locking on the collection and with minimal contention.
- They provide *thread-safety* and *atomicity* of operations and avoid *memory consistency errors* by defining a *happens-before relationship* that essentially states that an action by a thread to place an object into a concurrent collection happens-before a subsequent action to access or remove that object from the collection by another thread.
- The concurrent collections and maps in the can be categorized as follows:

  - Concurrent collections

  - Concurrent maps

  - Blocking queues

  - Copy-on-write collections

Concurrent Collections features -1

The following aspects about collections can be crucial in selecting an appropriate collection for maintaining the elements:

- **Non-blocking or blocking**. Operations on a *non-blocking* collection do not use any locking mechanism (they are *lock-free*) to access elements of the collection, allowing multiple threads to execute concurrently, in contrast to a *blocking* collection in which operations can block to acquire a lock before they can proceed, because only one thread at a time is allowed. Collections with **Concurrent** in their name are all <u>non-blocking</u>.
- **Unbounded and bounded**. A *bounded* collection has a fixed capacity that defines the maximum number of elements allowed in the collection (usually specified when the collection is created). When the collection is full, some operations, such as adding an element, will block until there is space in the collection. *Unbounded* collections are not bounded by any capacity restriction. *Optionally bounded collections* can be instantiated to behave either as bounded or as unbounded. **...**

Concurrent Collections features -2

…

- **The null value**. All concurrent collections, blocking queues, and concurrent maps *do not allow the null value* as elements, whereas the copy-on-write collections *do*.
- **Duplicate elements**. All collections that embody the concept of a Set do not allow duplicates. All Maps do not allow duplicate keys which would violate the concept of hashing values. However, all queues allow duplicates.
- **Ordering**. Elements in some concurrent sets and maps do not have any ordering, for example, a CopyOnWriteArraySet or a ConcurrentHashMap. Some sets and maps keep their elements and entries in sort order, according to either their natural ordering or a total ordering defined by a comparator on the elements or the keys, respectively. Examples of concurrent sorted collections are a ConcurrentSkipListSet and a ConcurrentSkipListMap.

…

Concurrent Collections features -3

…

- **Iterator behavior during serial access**. Iterators can be classified according to the response they provide on modifications that are made to the collection, outside the control of the iterator, which is referred to as *concurrent modification*:

  - ***Weakly consistent (fail-safe) iterator***. It is created on a clone of the collection, and will always traverse elements that existed at the time the iterator was constructed only once. It may not reflect all subsequent modifications made to the collection after the iterator is constructed. It will also not throw a ConcurrentModificationException and can execute concurrently with other operations on the collection. Weakly consistent iterators also provide guarantees against repeated elements occurring, and guard against a variety of errors and from infinite loops occurring during traversal.

…

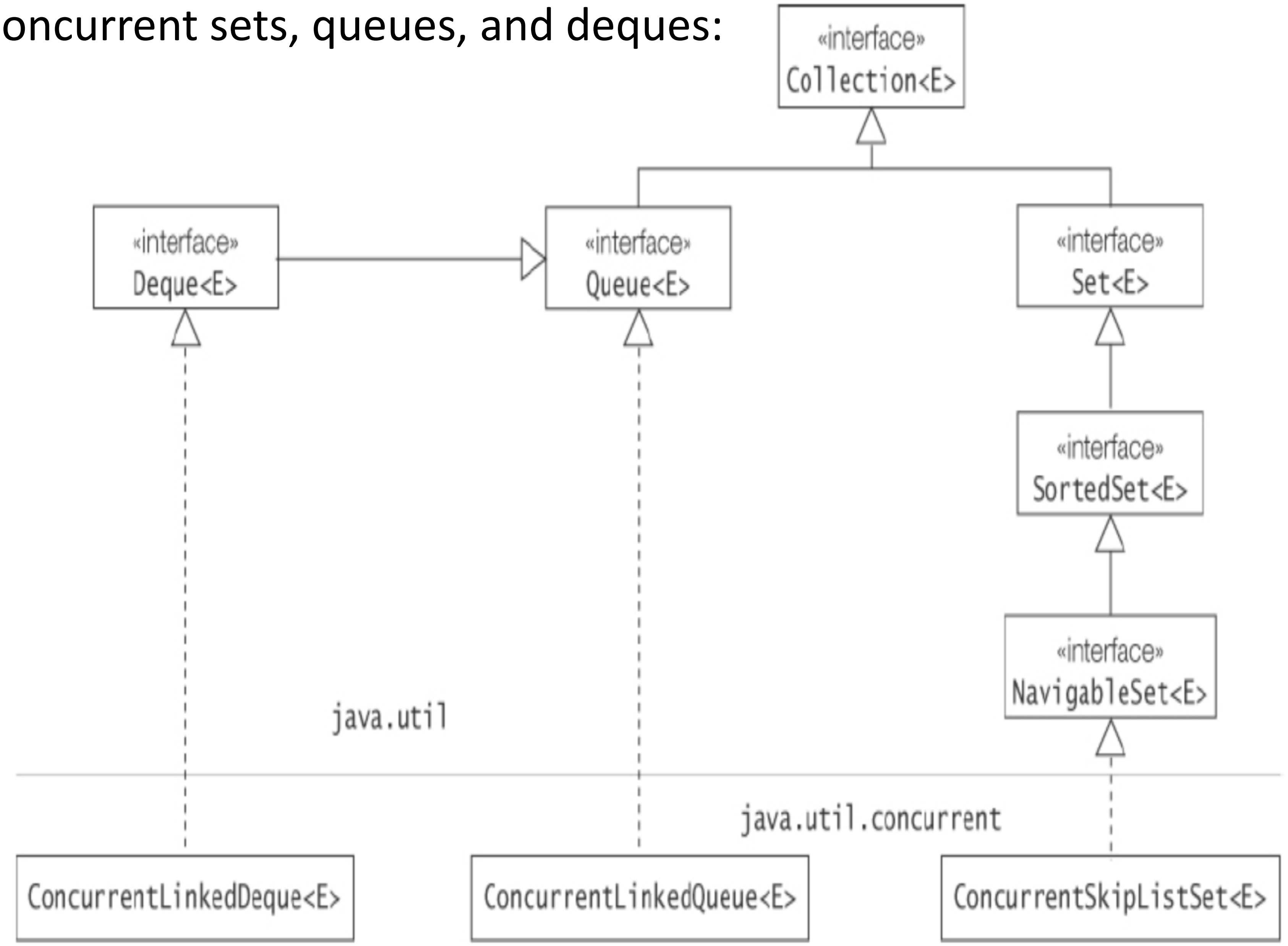
Concurrent Collections features - 4

…

- ***Snapshot-style iterator***. A CopyOnWriteArrayList uses snapshot-style iterators for serial access. It iterates over a snapshot of the underlying array taken at the time the iterator is created. Concurrent multiple read operations on this array snapshot are thread-safe as this snapshot of the array cannot be changed, but any write operation is done on a new copy of the array. An iterator is always created on the most current modified array. A snapshot iterator does not reflect any changes made in the copy of the array - in contrast to a weakly consistent iterator that may reflect such changes.
- ***Fail-fast iterator***. At the start of each iteration, a fail-fast iterator detects whether the collection has been modified. If that is the case, it throws a ConcurrentModificationException. Thread-unsafe collections from the java.util package exhibit this behavior, as do the PriorityBlockingQueue and DelayDeque.

Concurrent Collections

Concurrent sets, queues, and deques:



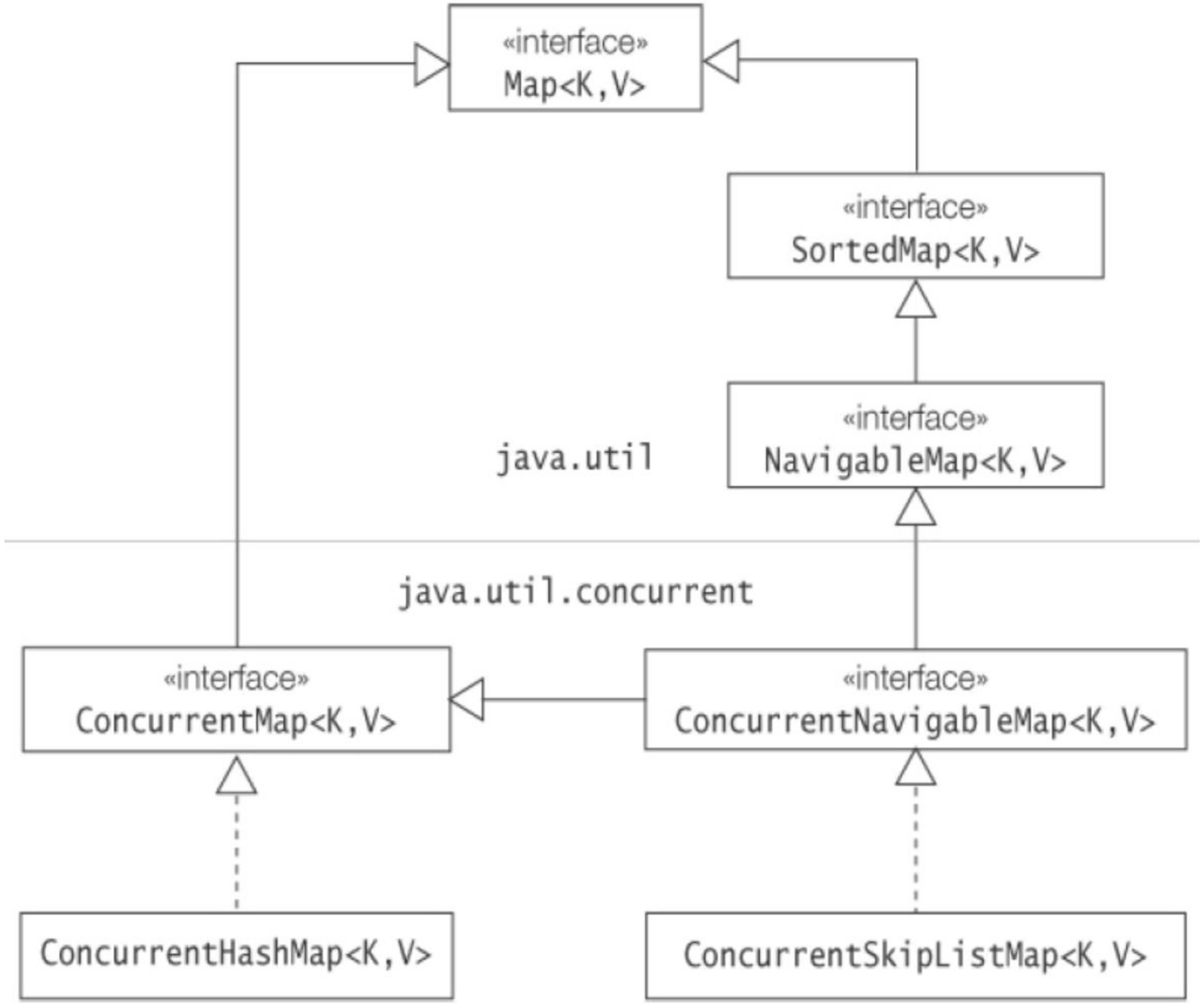
Concurrent Collections

Concurrent collection	null value	Dupli-cates	Ordering	Kind of Iterator
ConcurrentSkipListSet<E> **extends** AbstractSet<E> **implements** NavigableSet<E>	No	No	Sort order	Weakly consistent
ConcurrentLinkedQueue<E> **extends** AbstractCollection<E> **implements** Queue<E>	No	Yes	FIFO order	Weakly consistent
ConcurrentLinkedDeque<E> **extends** AbstractQueue<E> **implements** Deque<E>	No	Yes	FIFO/ LIFO order	Weakly consistent

- These concurrent collections are unbounded and non-blocking.
- These concurrent collections use Compare-And-Swap algorithm for non-blocking multi-threaded access.
- Their bulk operations (addAll(), removeIf(), forEach()) are not guaranteed to perform atomically.

see conclinkqueue

Concurrent maps



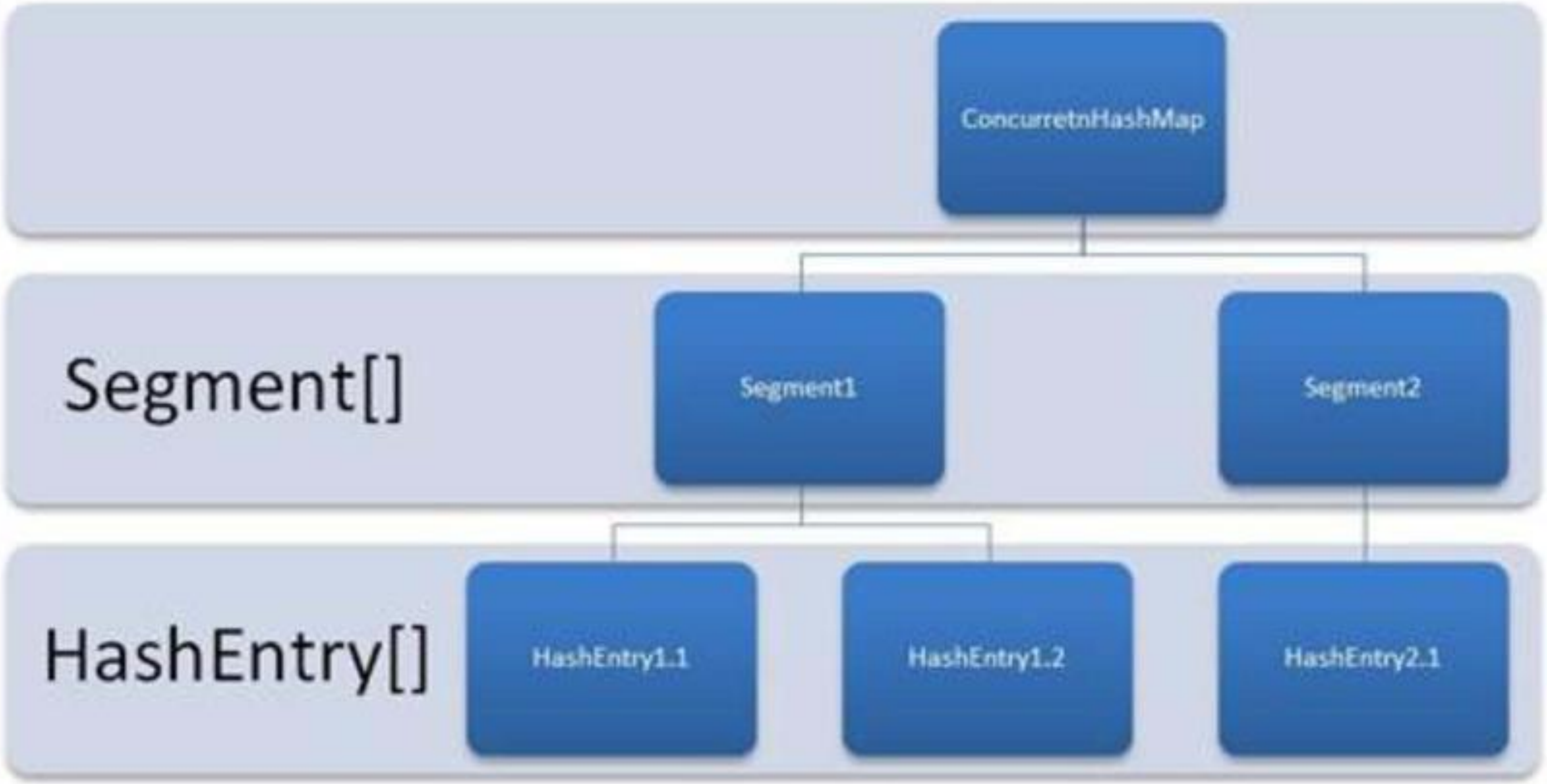
Concurrent Maps

Concurrent map	null value	Dupli-cates	Orde-ring	Kind of Iterator
ConcurrentHashMap<K,V> **extends** AbstractMap<K,V> **implements** ConcurrentMap<K,V>	Not allowed as key or values	Unique keys	No order	Weakly consistent
ConcurrentSkipListMap<K,V> **extends** AbstractMap<K,V> **implements** ConcurrentNavigableMap<K,V>	Not allowed as key or values	Unique keys	Key sort order	Weakly consistent

- The implementation of the ConcurrentHashMap<K,V> divides the map into segments (sub-maps) that can be independently locked by a thread, thus allowing several threads to perform operations on the map concurrently.
- The ConcurrentSkipListMap<K,V> is organized as a hierarchy of linked lists with subsequences of elements that are sorted, allowing efficient traversal, but at the extra cost of insertions in the skip list.

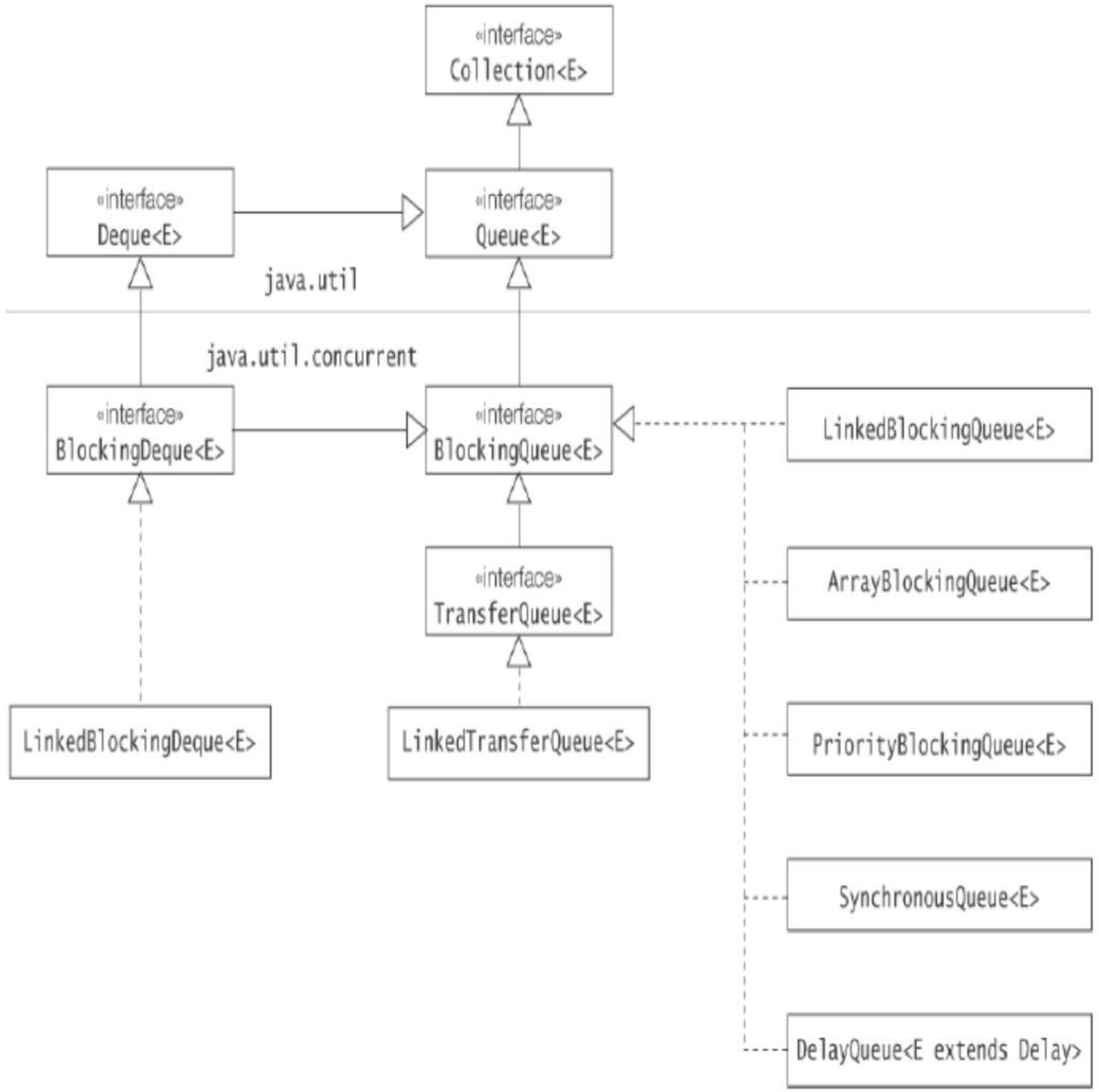
Concurrent Maps

- The ConcurrentHashMap



see conchashmap

Blocking Queues



BlockingQueue<E> implementations

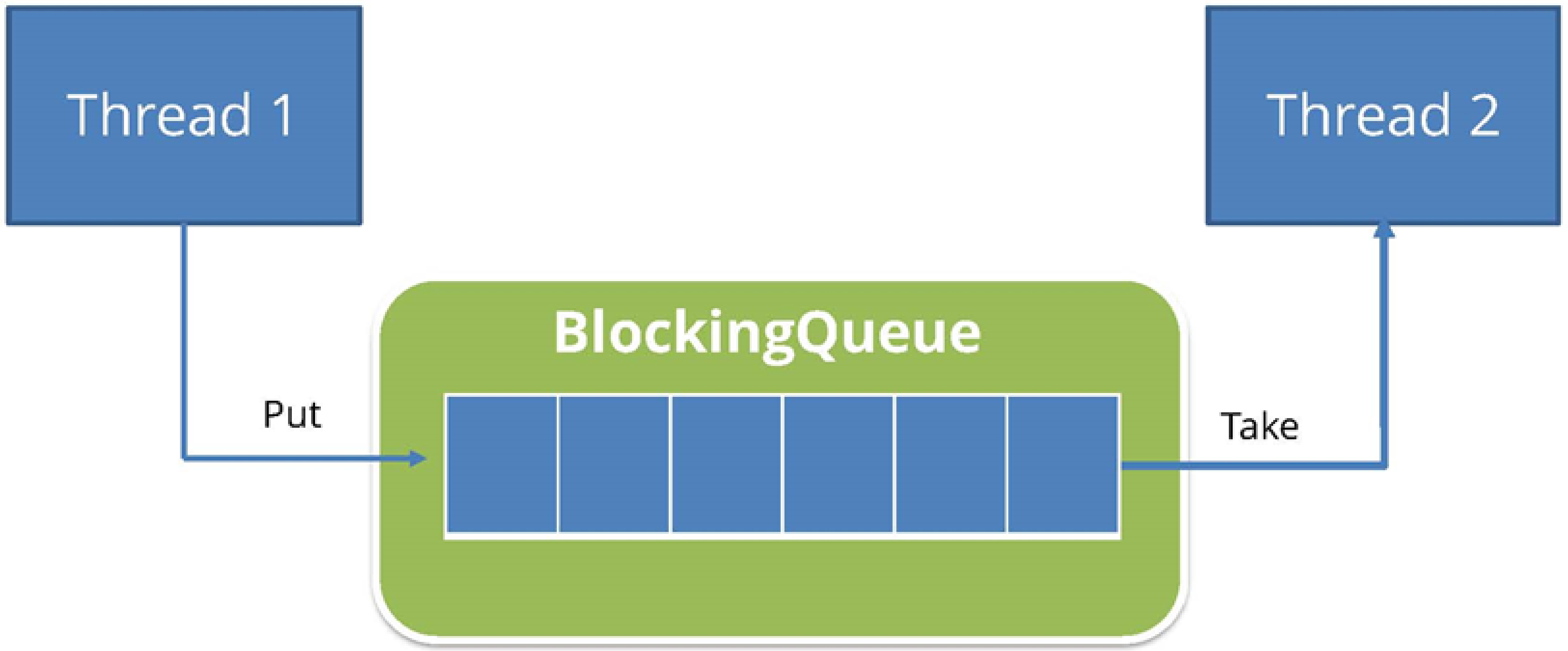
- **ArrayBlockingQueue <E>** - built on a classic circular buffer. In addition to the queue size, the ability to control the "fairness" of locks is available. If fair = false (default), then thread order is not guaranteed.
- **LinkedBlockingQueue <E>** - a queue on linked nodes, implemented on two locks: one lock for adding, the other for removing an element. In comparison with ArrayBlockingQueue, this class shows higher performance, but its memory consumption is higher. The queue size is set through the constructor and is Integer.MAX_VALUE by default.
- **PriorityBlockingQueue <E>** - multithreaded wrapper over PriorityQueue. When an element is inserted into the queue, its order is determined in accordance with the Comparator's logic or the implementation of the Comparable interface for the elements. The smallest item comes out first.
- **SynchronousQueue <E>** - works on the principle of one in, one out. Each insert operation blocks the Producer thread until the Consumer thread pulls the item out of the queue, and vice versa, the Consumer will wait until the Producer thread inserts the item.
- **DelayQueue <E extends Delayed>** - allows you to pull elements from the queue only after a certain delay, defined in each element through the getDelay method of the Delayed interface.

Blocking Queues

Blocking queue	null value	Dupli-cates	Orde-ring	Kind of Iterator
ArrayBlockingQueue<E> **extends** AbstractQueue<E> **implements** BlockingQueue<E>	No	Allowed	FIFO order	Weakly consistent
LinkedBlockingQueue<E> **extends** AbstractQueue<E> **implements** BlockingQueue<E>	No	Allowed	FIFO order	Weakly consistent
PriorityBlockingQueue<E> **extends** AbstractQueue<E> **implements** BlockingQueue<E>	No	Allowed	Natural order	Fail-fast
SynchronousQueue<E> **extends** AbstractQueue<E> **implements** BlockingQueue<E>	No	Not applicable	Not applicable	Not applicable
DelayQueue<E **extends** Delayed> **extends** AbstractQueue<E>	No	Allowed	Longest delay order	Fail-fast

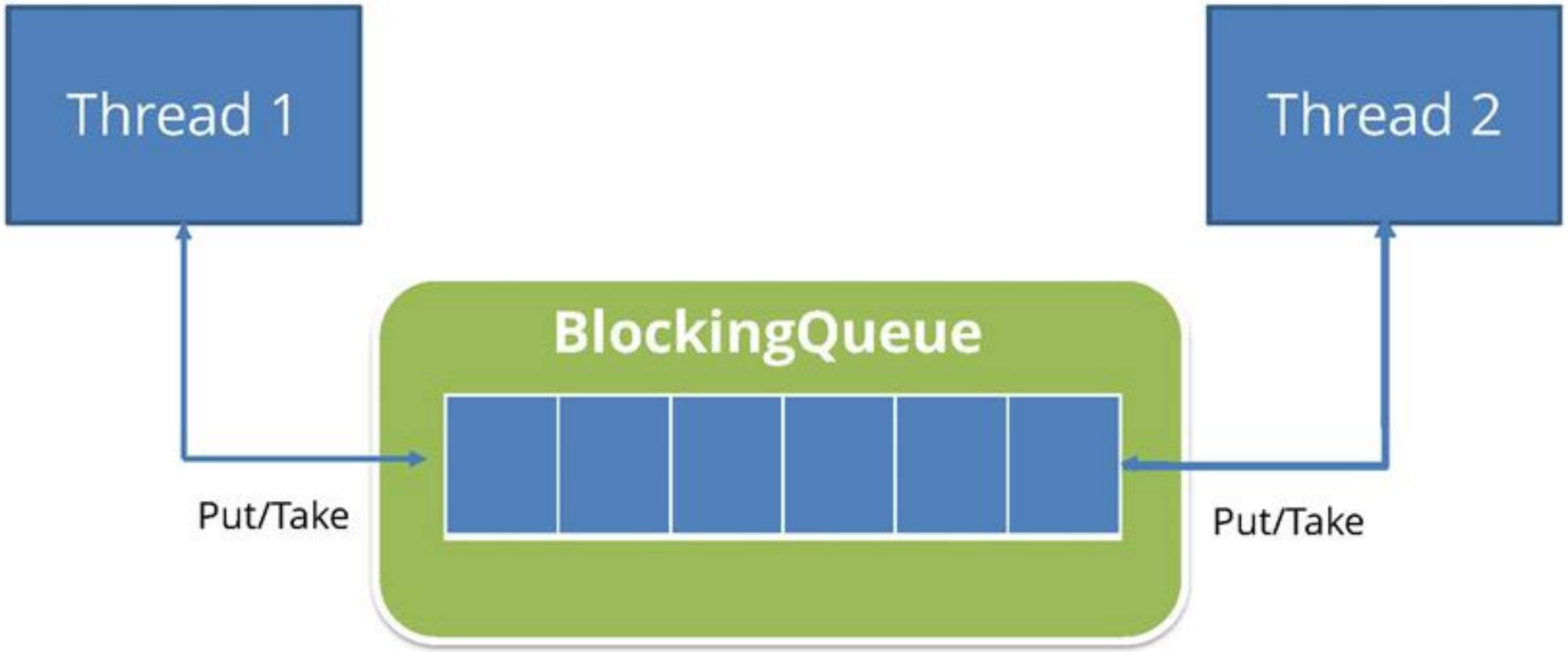
Blocking Queues

- The BlockingQueue



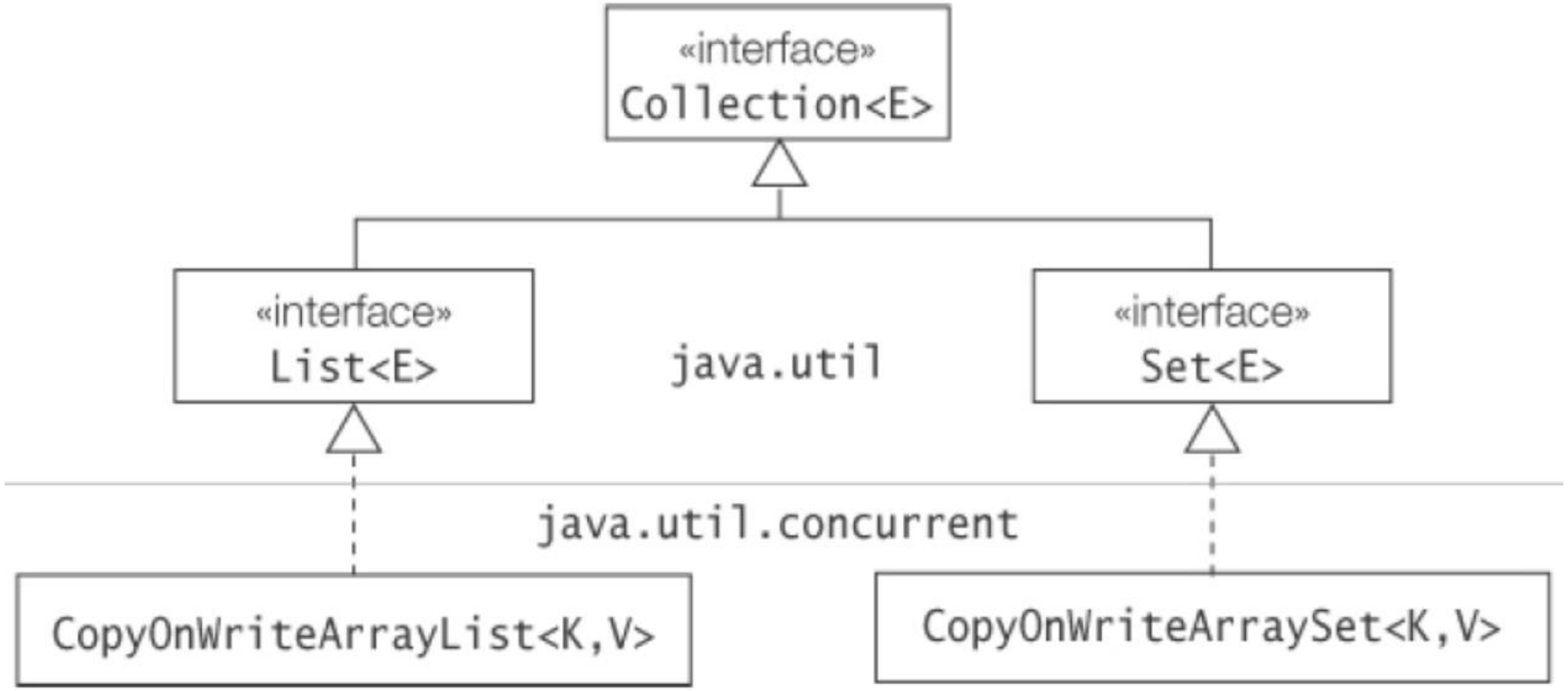
Blocking Deques

- The BlockingDeque



see blockingqueue

CopyOnWrite Collections



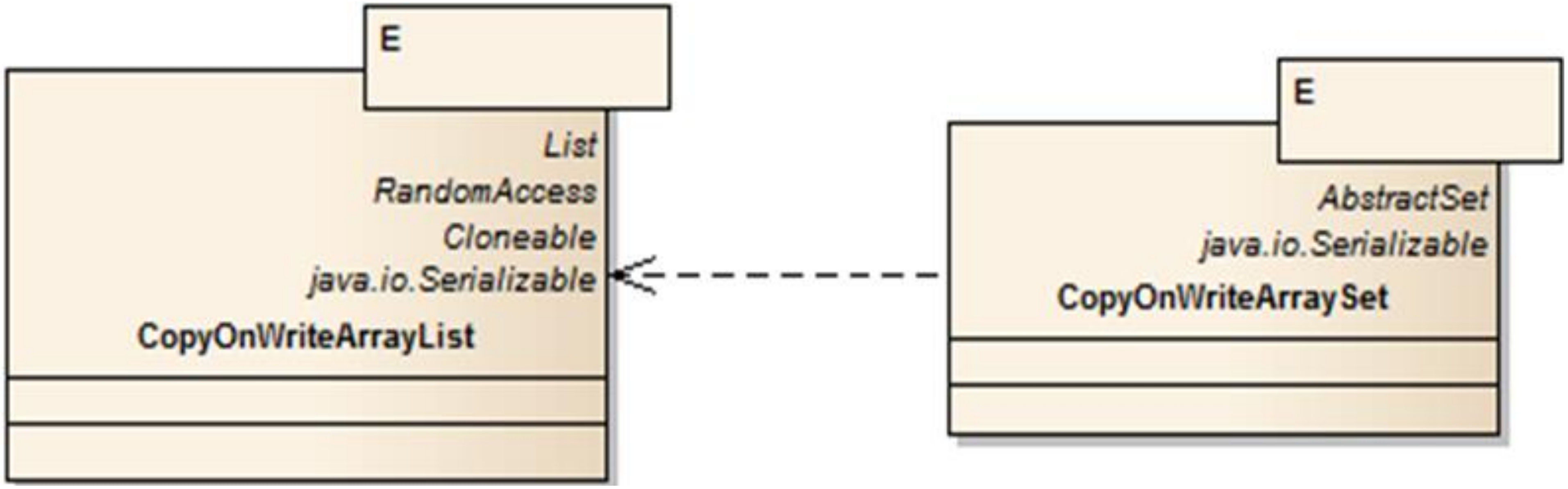
- Copy-on-write collections contain lists and special-purpose sets that are recommended when the number of read operations significantly exceeds the number of change operations on the collection's data.

CopyOnWrite Collections

CopyOnWrite collection	null value	Dupli-cates	Orde-ring	Kind of Iterator
CopyOnWriteArrayList<E> **implements** List<E>	Yes	Yes	Insert order	Snapshot-style
CopyOnWriteArraySet<E> **extends** AbstractSet<E>	Yes	No	No order	Snapshot-style

- They use an underlying array to implement its operations.
- By default, such a list/set is immutable, guaranteeing thread-safety of concurrent operations. Any mutative operation (add, set, remove, etc.) is done on a new copy of its entire underlying array which then supersedes the previous version for subsequent operations.
- For efficiency reasons, the list/set size should be kept small and modifications should be minimized, as they are expensive, incurring the copying cost of its underlying array and extra space.

CopyOnWrite collections



CopyOnWriteArrayList<E> - additional methods:

int indexOf(E e, int index)
int lastIndexOf(E e, int index)
boolean addIfAbsent(E e)
int addAllAbsent(Collection<? extends E> c)

CopyOnWriteArraySet<E> - no additioanl methods

add(E e) method of CopyOnWriteArrayList

```
private transient volatile Object[] array;

public boolean add(E e) {
        final ReentrantLock lock = this.lock;
        lock.lock();
        try {
            Object[] elements = getArray();
            int len = elements.length;
            Object[] newElements = Arrays.copyOf(elements, len + 1);
            newElements[len] = e;
            setArray(newElements);
            return true;
        } finally {
            lock.unlock();
        }
    }
```

see copyonwritearraylist

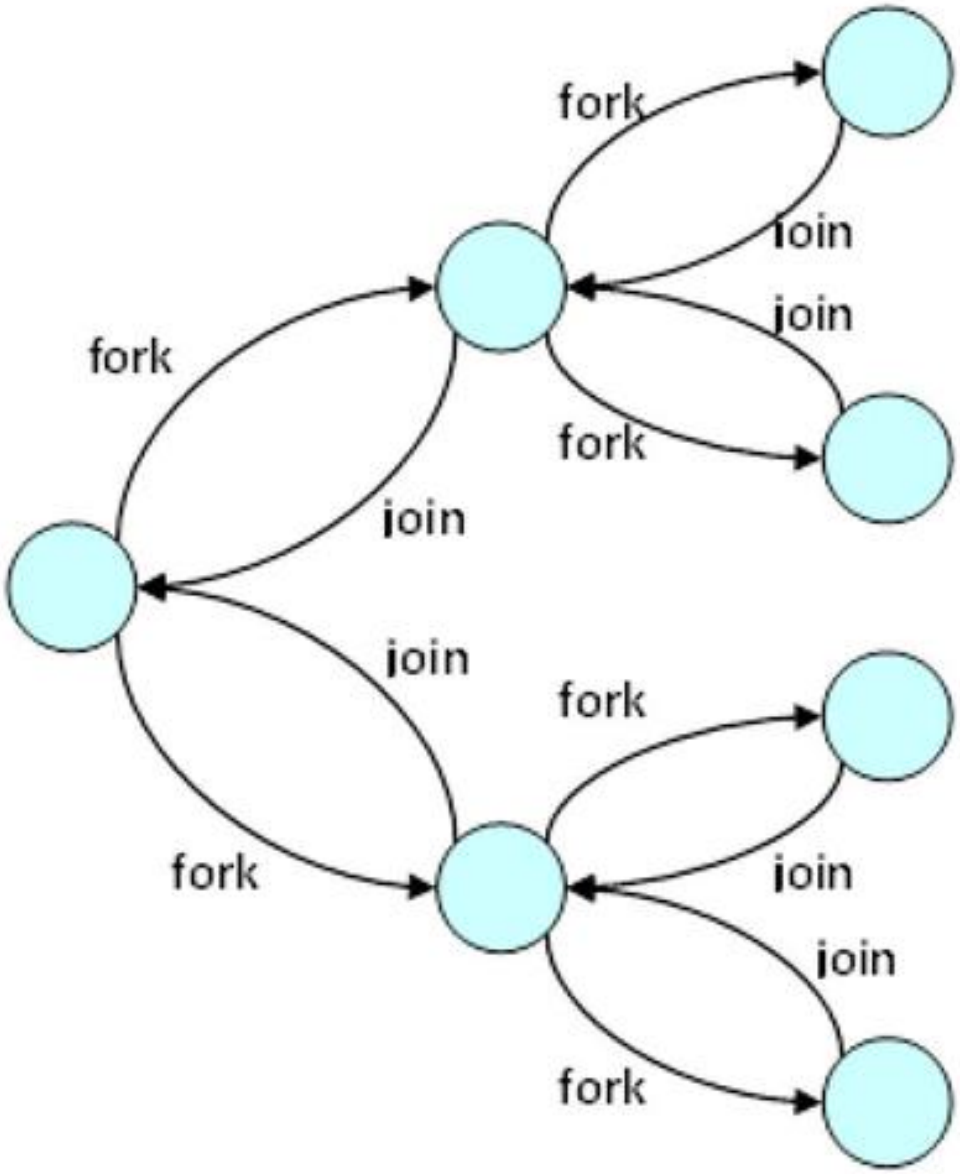
Module contents

Fork/Join Framework

- The Fork/Join Framework is specially suited for problems that can be solved by the ***divide-and-conquer problem-solving paradigm***.
- The main step in a divide-and conquer algorithm is to divide the task repeatedly into subtasks of the same type until the task size is smaller than a given threshold, such that a subtask can be solved directly, it is called the ***base case***.
- The algorithm proceeds with dividing the task into subtasks, solving each subtask recursively, and combining the results from each subtask, all the time <u>the idea being that each subtask gets smaller and moves toward the base case</u>.
- Solving a subtask recursively is referred to as ***forking***, and combing the results of subtasks is referred to as ***joining***, hence the name **Fork/Join Framework**.
- This technique is prominent in processing data in large arrays and collections—for example, in sorting and searching algorithms designing.

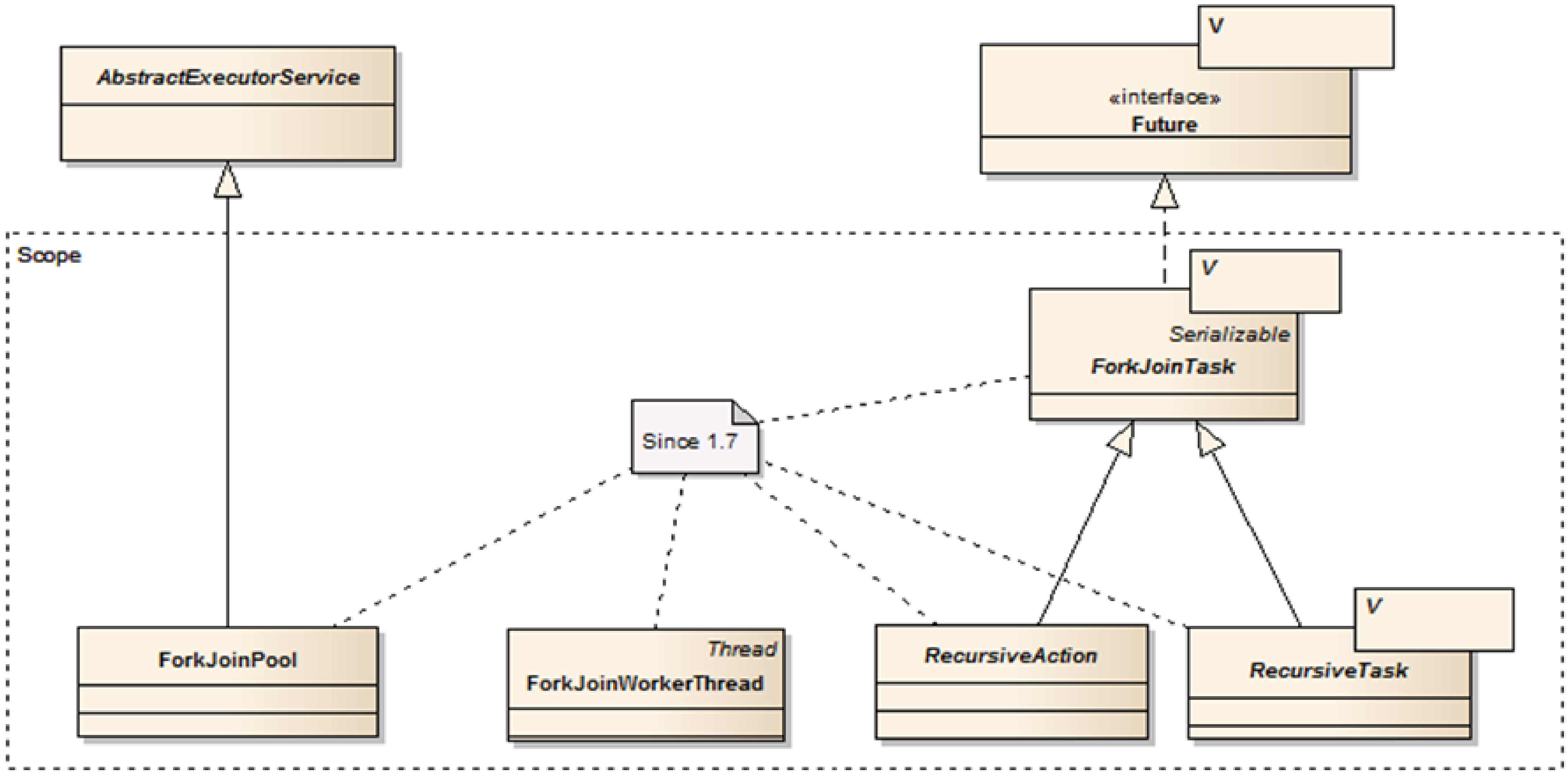
since JDK 7

Fork/Join Framework



since JDK 7

Fork-Join Framework



Fork/Join Framework

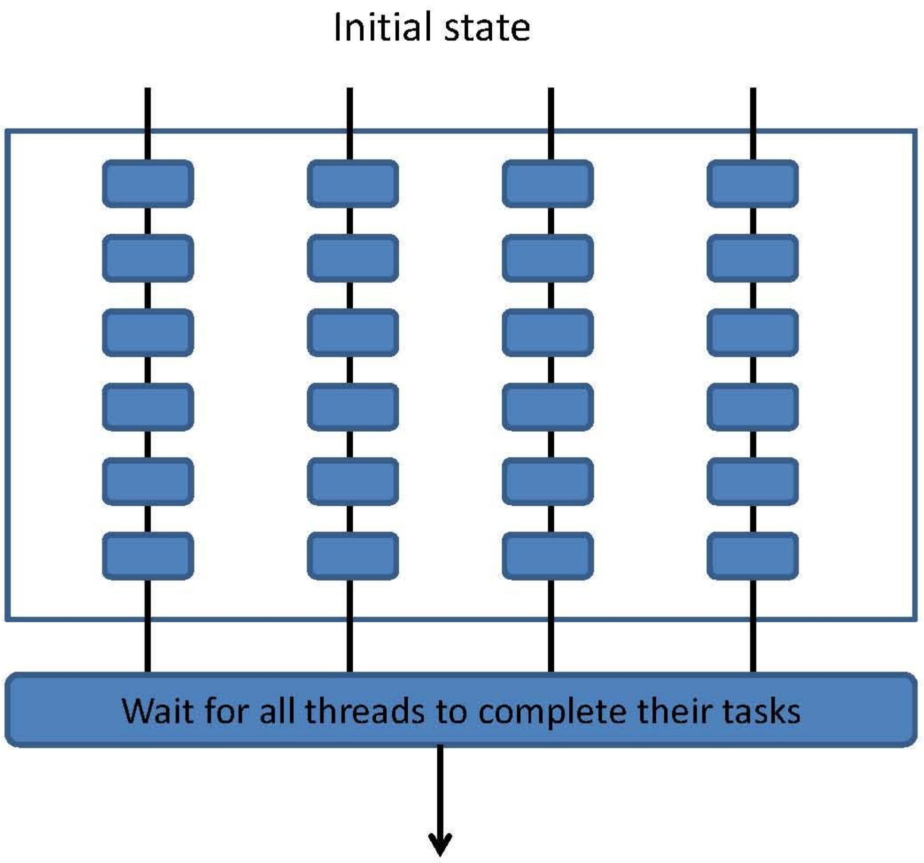
- **ForkJoinPool** - represents a special kind of executor for running the root (main) **ForkJoinTask** task. Subtasks are launched through the methods of the root task. By default, a thread pool is created with the number of threads equal to the number of processors (cores) available to the JVM.
- **ForkJoinTask<V>** - an abstract class for tasks that run within a **ForkJoinPool**. Its key methods include:
  `ForkJoinTask<V>` **`fork()`** - adds a task to the queue of the current **ForkJoinWorkerThread** for asynchronous execution;
  `V` **`invoke`**`()` - starts a task execution awaits it completion (synchronous) and returns result;
  `V` **`join`**`()` - waits for the asynchronous subtask to complete and return the result;
  `void` **`invokeAll`**`(ForkJoinTask<?>... tasks)`- combines all three previous operations, performing two or more tasks in one go;
  `ForkJoinTask<?>` **`adapt`**`(Runnable/Callable o)` **-** creates a new **ForkJoinTask** from Runnable or Callable objects.

Fork/Join Framework

- **RecursiveTask<V>** - an abstract class - a descendant of **ForkJoinTask**, with a declaration of the `V` **`compute`**`()` method in which the asynchronous operation should be performed.
- **RecursiveAction** - an abstract class - differs from **RecursiveTask<V>** in that it does not return a result.
- **ForkJoinWorkerThread** - used as the default implementation in **ForkJoinPool**.

The Fork-Join Framework 2/12

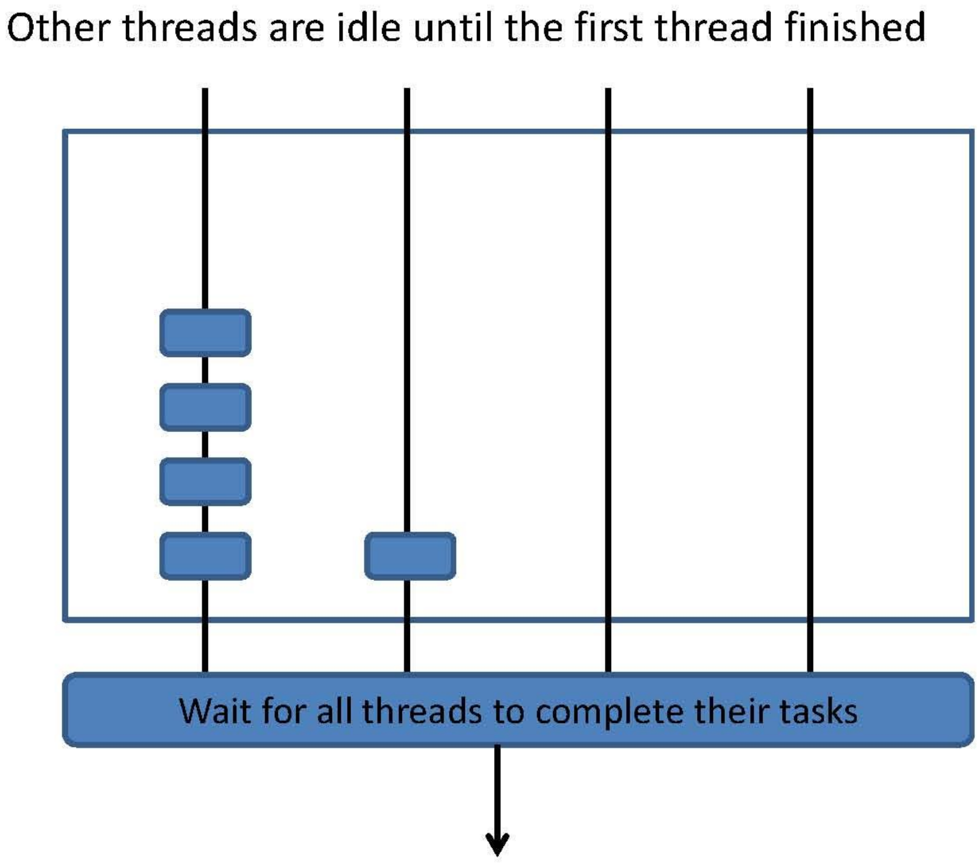
- Work Stealing



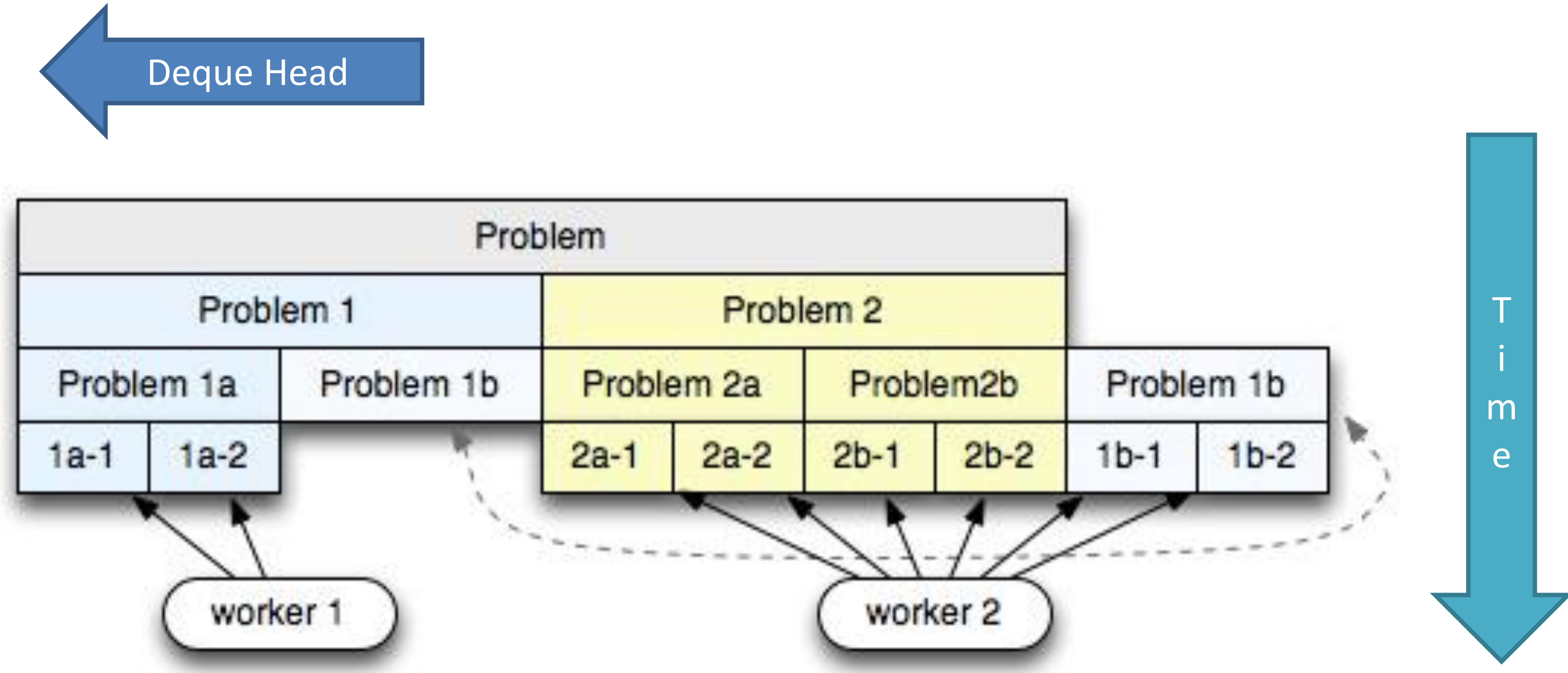
The Fork-Join Framework 3/12

- Work Stealing

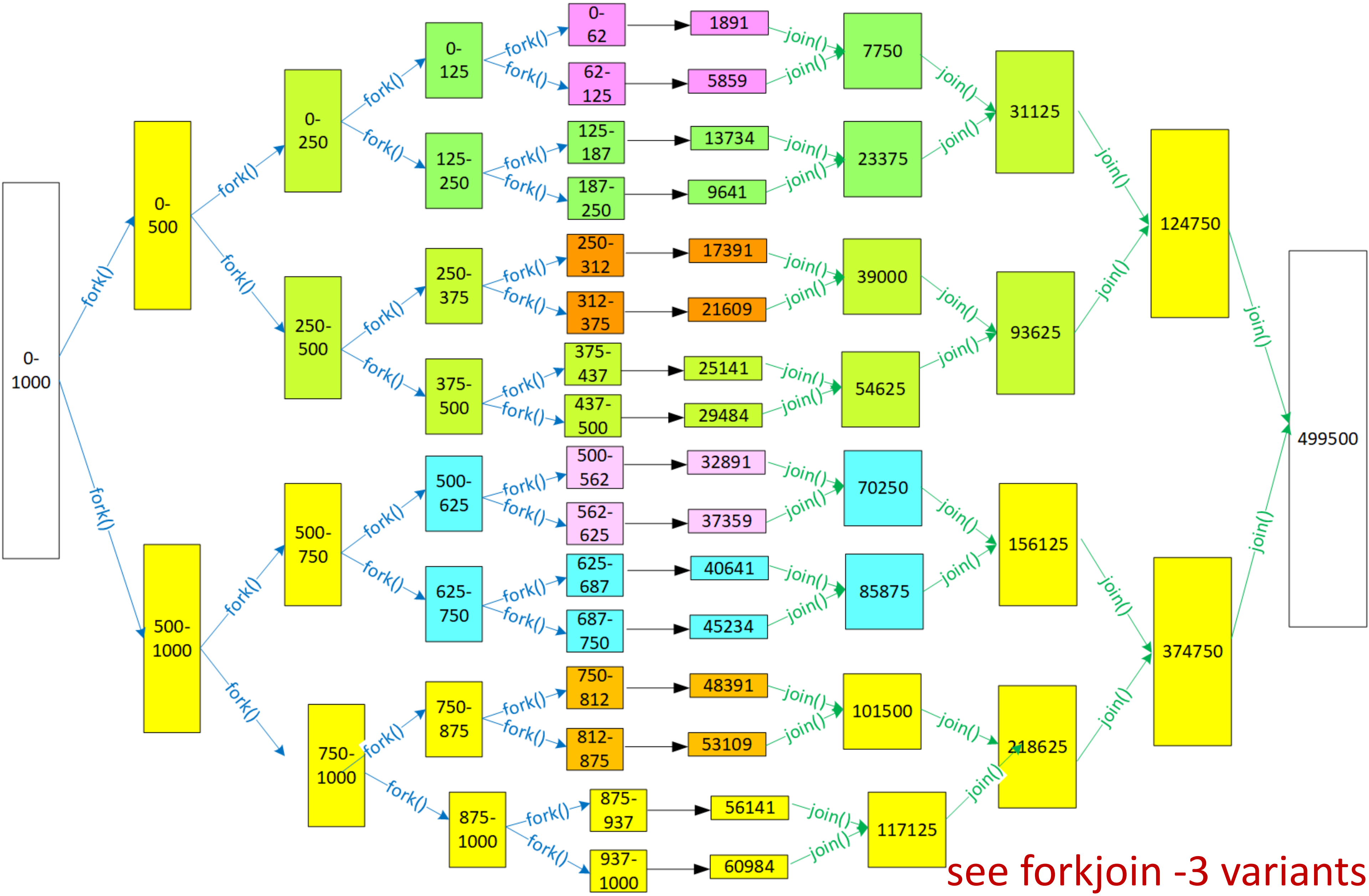
If some of the threads finish their work early, then we use them to help other threads. However, you need to be careful: "stealing" work from another worker thread will require synchronization, which will slow down processing.
The Fork-Join framework solves this problem in a clever way: it recursively divides the work into parts and uses a DEQUE structure to allocate subtasks.


Other threads are idle until the first thread finished
Wait for all threads to complete their tasks

Fork/Join Framework



Fork/Join Framework



see forkjoin -3 variants