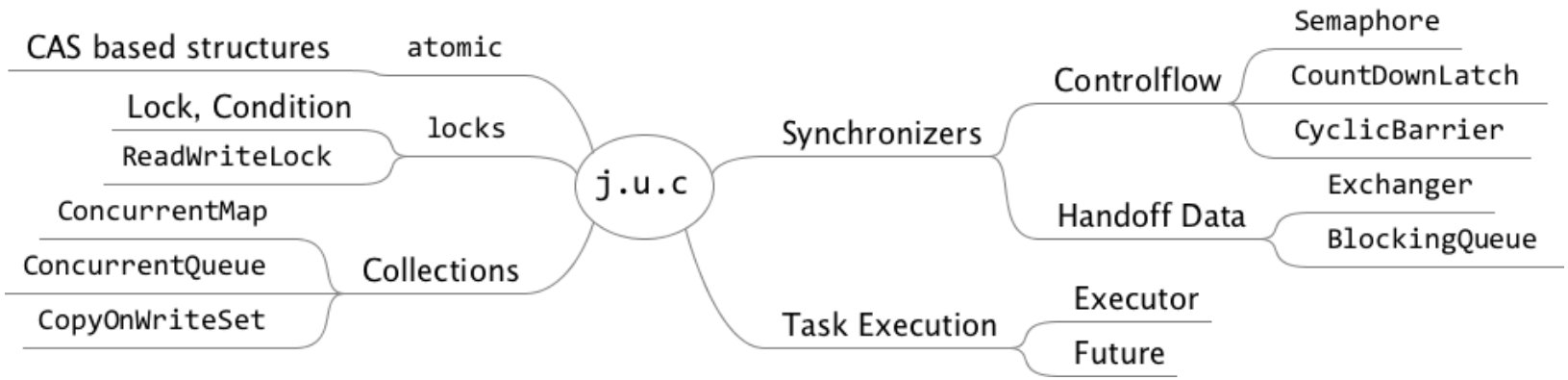
Synchronizers

- Overview java.util.concurrent
- Synchronizers
- Memory Consistency Effects

If there is a library that can save you from doing low-level multi-threaded programming, by all means use it. [Effective Java]

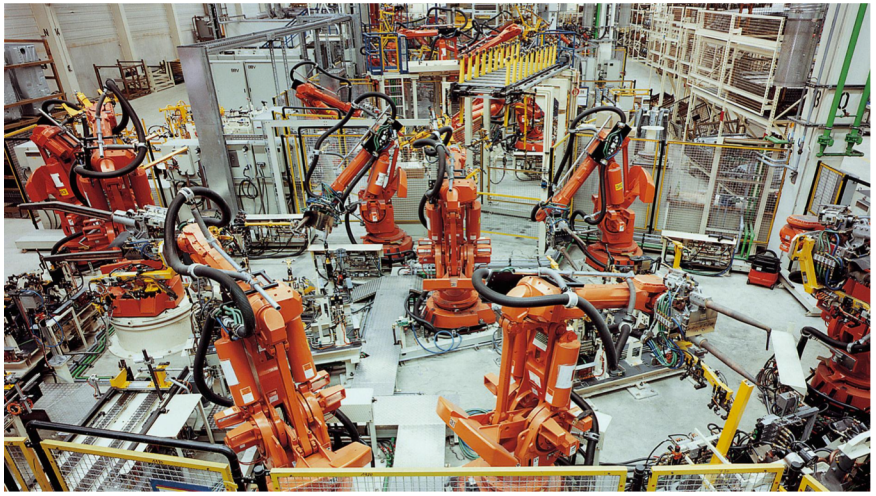
Package java.util.concurrent.*

"Utility classes commonly useful in concurrent programming. This package includes a few small standardized extensible frameworks, as well as some classes that provide useful functionality and are otherwise tedious or difficult to implement." [JavaDoc for j.u.c.]



Synchronizers

- **A synchronizer is any object that coordinates the control flow of threads based on its state.**
- **Discussed in this lecture**
  - Controlflow
    - Semaphore
    - ReadWriteLock
    - CountDownLatch
    - CyclicBarrier
  - Handoff Data
    - Exchanger
    - BlockingQueue



Delegate thread-safety to existing classes!
Just let them manage all the state.

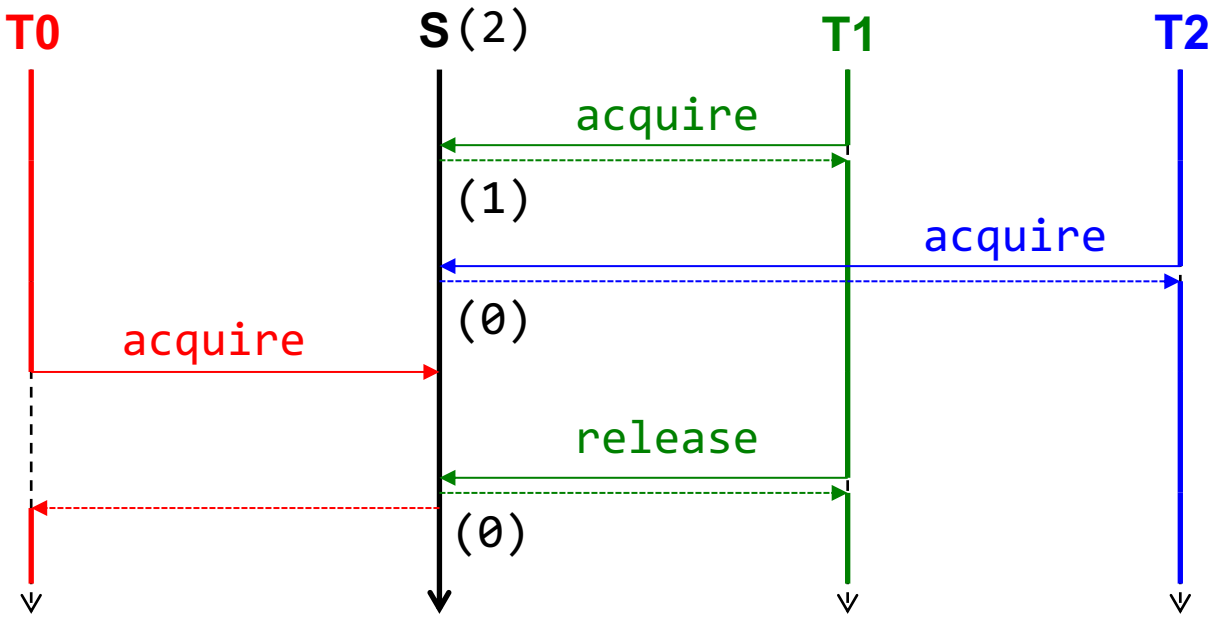
Semaphore

- **A semaphore is an integer variable that represents a resource counter (conceptually: maintains a set of permits)**
  1. The semaphore is initialized with the number of permits
  2. acquire()
     - If (permits > 0) Decrements permits and proceeds
     - Else blocks until a permit is available
  3. release() increments the permits, potentially releasing a waiting acquirer

```
public class Semaphore {
    public Semaphore(int permits) {...}
    public void acquire() throws InterruptedException {...}
    public void acquireUninterruptibly() {...}
    public void release() {...}
}
```

- **Usage: Restrict the number of threads than can access some (physical or logical) resource**

Semaphore - Behavior



Semaphore Example: CarPark

```
class SemaphoreCarPark implements CarPark {
    private final Semaphore sema;

    public SemaphoreCarPark(int places) {
        sema = new Semaphore(places);
    }

    public void enter() {
        sema.acquireUninterruptibly();
        log("enter carpark");
    }

    public void exit() {
        log("exit carpark");
        sema.release();
    }
}
```

wait-notify: CarPark

```
public class CarPark4 implements CarPark {
    private int places;
    public CarPark4(int places){ this.places = places; }

    public synchronized void enter() {
        while(places == 0) {
            try { wait(); } catch (InterruptedException e) {}
        }
        log("enter carpark");
        places--;
    }

    public synchronized void exit() {
        log("exit carpark");
        places++;
        notify();
    }
}
```

Semaphore Example: Lock

```
class SemaphoreLock {
    private final Semaphore mutex = new Semaphore(1);

    public void lock() {
        mutex.acquireUninterruptibly();
    }

    public void unlock() {
        mutex.release();
    }
}
```

- **Binary semaphores can be used as mutex (lock)**
- **No notion of the owner of a lock**
  - No reentrancy
  - T2 may release a lock acquired by T1

ReadWriteLock Motivation

```
class KeyValueStore {
   private final Map<String, Object>  m = new TreeMap<>();
   private final Lock l = new ReentrantLock();

   public Object get(String key) {
      l.lock(); try { return m.get(key); } finally { l.unlock(); }
   }
   public Set<String> allKeys() {
      l.lock(); try { return new HashSet<>(m.keySet()); } finally { l.unlock(); }
   }
   public void put(String key, Object value) {
      l.lock(); try { m.put(key, value); } finally { l.unlock(); }
   }
   public void clear() {
      l.lock(); try { m.clear(); } finally { l.unlock(); }
   }
}
```

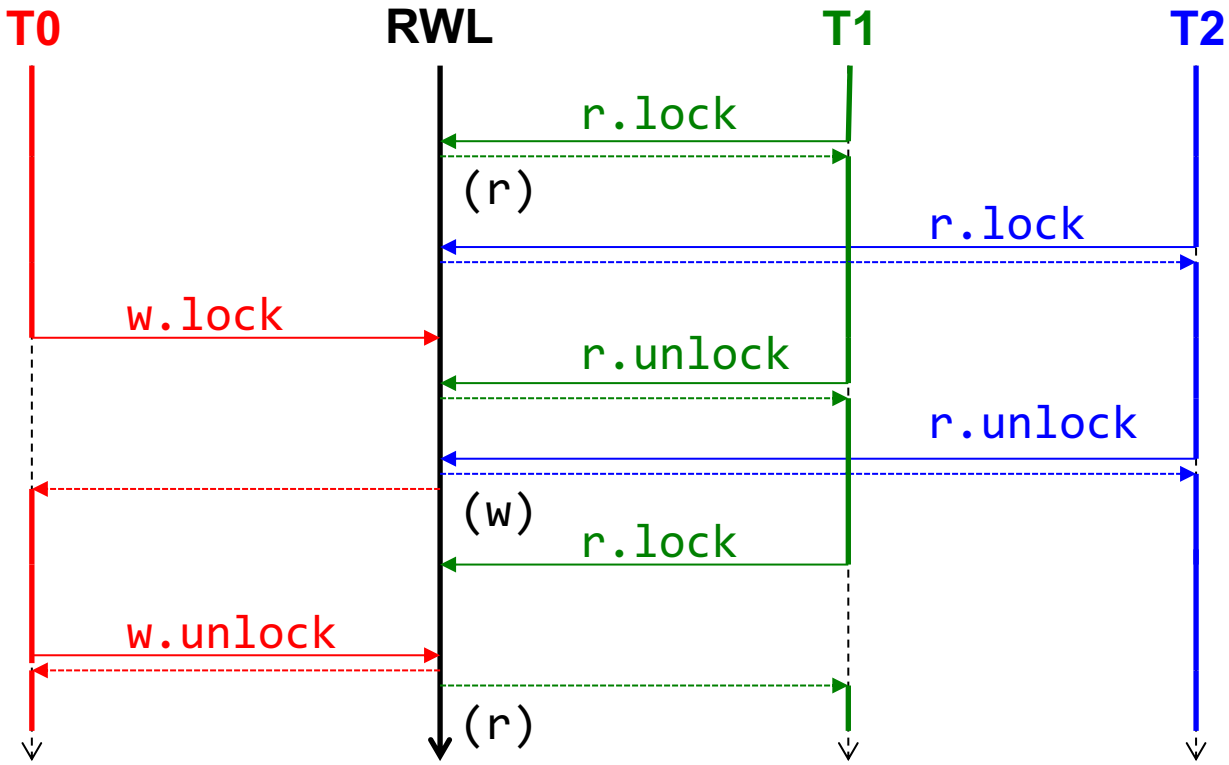
ReadWriteLock

- **A ReadWriteLock maintains a pair of associated locks**
  - Read lock
    - lock for reading, may be held simultaneously by multiple readers
  - Write lock
    - lock for writing, is exclusive

```
public interface ReadWriteLock {
    Lock readLock();  // allows for concurrent reads
    Lock writeLock(); // writes are exclusive
}
```

- **Usage: Allows for a greater level of concurrency in accessing shared data than that permitted by a mutual exclusion lock**
  - Performance improvement depends on access patterns

ReadWriteLock- Behavior



ReadWriteLock Example

```
class KeyValueStore {
   private final Map<String, Object>  m = new TreeMap<>();
   private final ReadWriteLock rwl = new ReentrantReadWriteLock();
   private final Lock r = rwl.readLock();
   private final Lock w = rwl.writeLock();

   public Object get(String key) {
      r.lock(); try { return m.get(key); } finally { r.unlock(); }
   }
   public Set<String> allKeys() {
      r.lock(); try { return new HashSet<>(m.keySet()); } finally { r.unlock(); }
   }
   public void put(String key, Object value) {
      w.lock(); try { m.put(key, value); } finally { w.unlock(); }
   }
   public void clear() {
      w.lock(); try { m.clear(); } finally { w.unlock(); }
   }
}
```

CountDownLatch

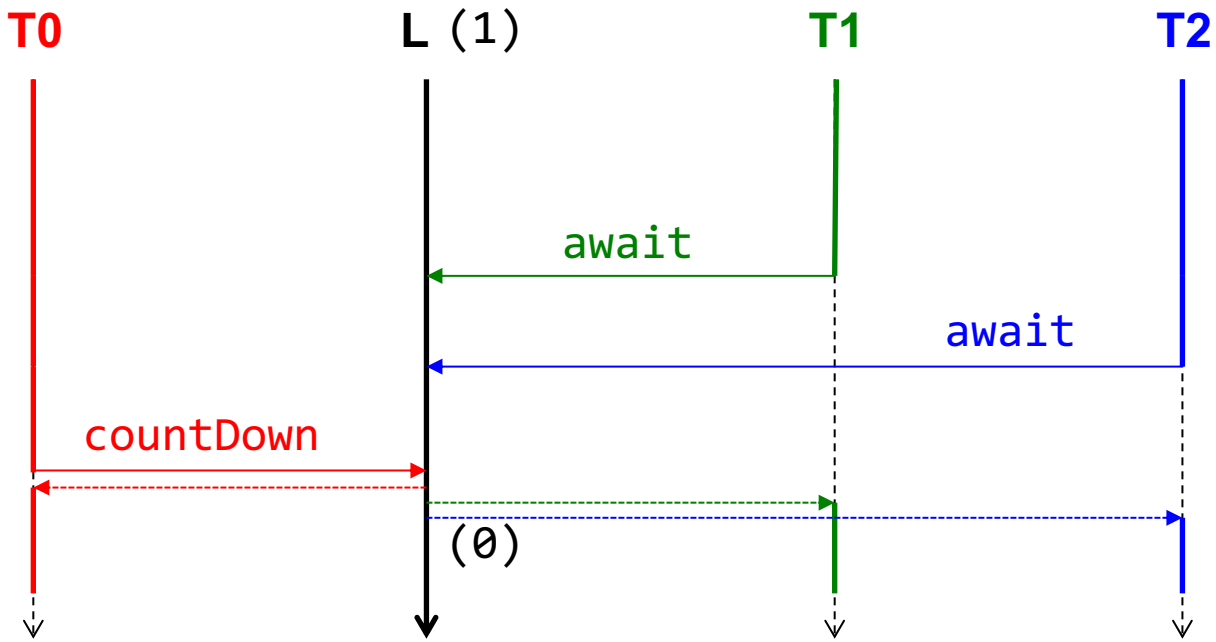
- **Delays the progress of threads until it reaches its terminal state**
  1. Latch is initialized with a count
  2. Threads calling await() are blocked if count is positive
  3. Calls to countDown() decrement the count
  4. If count reaches 0 all waiting threads are released

```
public class CountDownLatch {
    public CountDownLatch(int count) {...}
    public void await() {...}
    public void countDown() {...}
    ...
}
```

- **Usage: Ensure that an activity does not proceed until another one-time action completes**

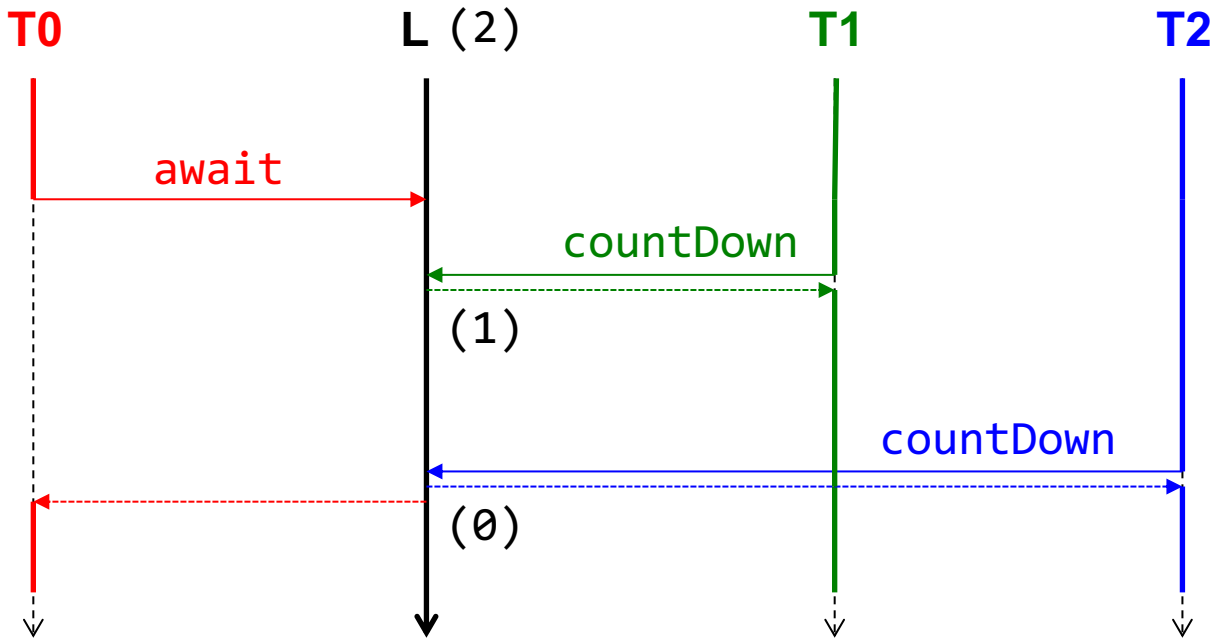
CountDownLatch

CountDownLatch- Behavior



- **T0 is starting a group (T1,T2) of related activities**

CountDownLatch- Behavior



- **T0 is waiting for a group (T1,T2) of related activities to complete**

CountDownLatch Example

```
final CountDownLatch startSignal = new CountDownLatch(1);
final CountDownLatch doneSignal  = new CountDownLatch(N);
for (int i = 0; i < N; ++i)
    new Thread() {
        public void run() {
            try {
                startSignal.await();
                doWork();
                doneSignal.countDown();
            } catch (InterruptedException ex) {}
        }
    }.start();
doSomethingElse();       // don't let them run yet
startSignal.countDown(); // let all threads proceed
doSomethingElse();
doneSignal.await();     // wait for all threads to finish
```

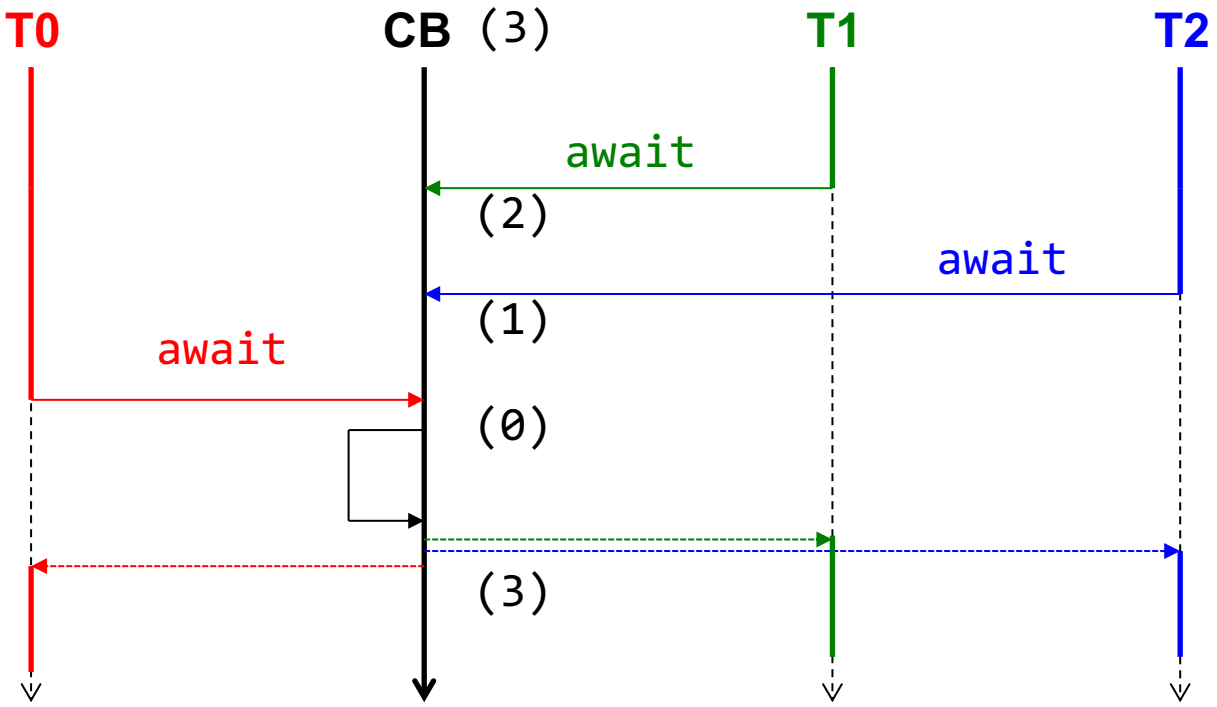
CyclicBarrier

- **Allows a set of threads to all wait for each other to reach a common barrier point.**
  1. Barrier is initialized with the number of threads (nThreads) which must wait for another
  2. The first (nThreads - 1) threads calling await() are blocked
  3. The nth thread calling await causes all other threads to proceed after the optional barrier action is executed
  4. After the barrier is released it can be reused (thus *cyclic)*

```
public class CyclicBarrier {
    public CyclicBarrier(int nThreads) {...}
    public CyclicBarrier(int nThreads, Runnable barrierAction)
    public void await() {...}
}
```

- **Usage: Allow a fixed size party of threads to occasionally wait for each other**

CyclicBarrier- Behavior



CyclicBarrier Example

```
final CyclicBarrier barrier = new CyclicBarrier(N);
for(int i = 0; i < N; i++) {
    final int segment = i; // final handle to i
    new Thread() {
        public void run() {
            try {
                while (true) {
                    prepare(segment);
                    barrier.await(); //wait for all other threads
                    display(segment);
                }
            } catch (Exception e) { /* ignore */ }
        }
    }.start();
}
```

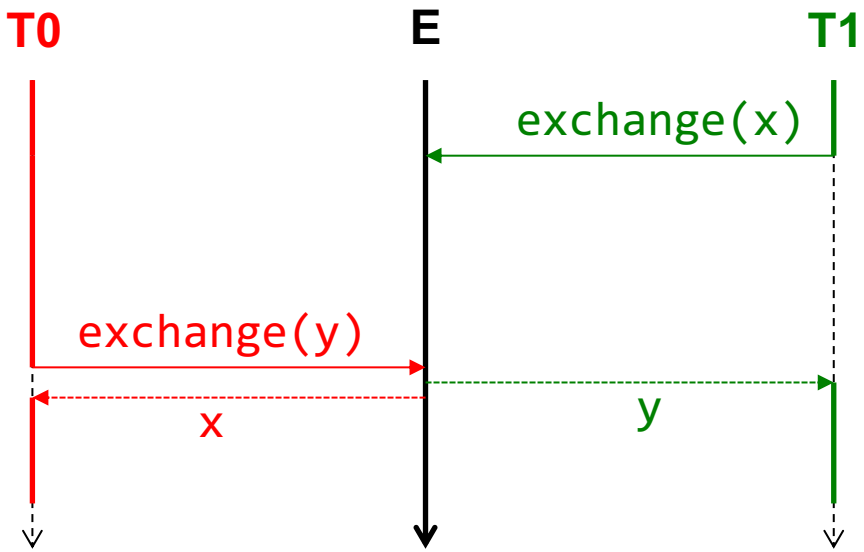
Exchanger

- **A synchronization point at which threads can pair and swap elements within pairs**
  1. The first thread offers an object to the exchange method and blocks
  2. The second thread offers an object to the exchange method and then
  3. Each thread receives the partner's object on return

```
public class Exchanger<T> {
    public T exchange(T t) {...}
    ...
}
```

- **Usage: Allows two threads to wait for each other and exchange an object**
  - Allows recycling of expensive objects

Exchanger- Behavior



Exchanger Example

```
class FillingLoop implements Runnable {
    private final Exchanger<List<Integer>> exchanger;
    private List<Integer> currentBuffer;

    FillingLoop(List<Integer> buf, Exchanger<List<Integer>> ex) {
        this.currentBuffer = buf; this.exchanger = ex;
    }

    public void run() { // exception handler omitted
        while (true) {
            if (currentBuffer.size() < MAX) {
                addToBuffer(currentBuffer);
            } else {
                // exchange full buffer for empty
                currentBuffer = exchanger.exchange(currentBuffer);
            }
        }
    }
}
```

Exchanger Example cont.

```
class EmptyingLoop implements Runnable {
    private final Exchanger<List<Integer>> exchanger;
    private List<Integer> currentBuffer;

    EmptyingLoop(List<Integer> buf, Exchanger<List<Integer>> ex) {
        this.currentBuffer = buf; this.exchanger = ex;
    }

    public void run() { // exception handler omitted
        while (true) {
            if (!currentBuffer.isEmpty()) {
                takeFromBuffer(currentBuffer);
            } else {
                // exchange empty buffer for full buffer
                currentBuffer = exchanger.exchange(currentBuffer);
            }
        }
    }
}
```

Exchanger Example cont.

```
public static void main(String[] args) {
    Exchanger<List<Integer>> exchanger = new Exchanger<>();

    new Thread(new FillingLoop(new ArrayList<Integer>(MAX),
                               exchanger)).start();
    new Thread(new EmptyingLoop(new ArrayList<Integer>(MAX),
                               exchanger)).start();
}
```

- **Only two ArrayList instances allocated**
  - Recycled between FillingLoop and EmptyingLoop
  - Avoids GC overhead
  - Reasonable for large MAX values

BlockingQueue / Producer-Consumer

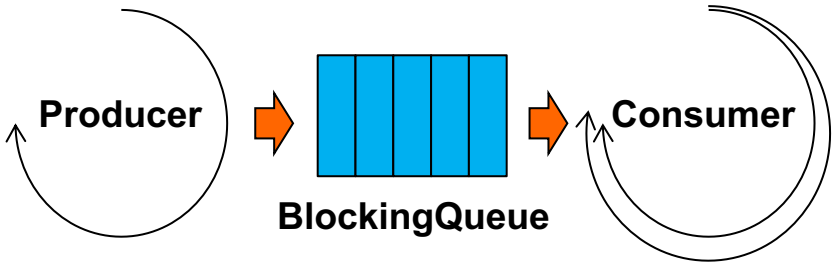
- **A queue which supports operations to wait for the queue to become non-empty when retrieving an element, and wait for space to become available when storing an element**
  1. A queue with a fixed capacity is shared between consumer and producer
  2. The producer puts elements into the queue
  3. The consumer removes elements from the queue

```
public interface BlockingQueue<E> extends Queue<E> {
    E take() throws InterruptedException;
    void put(E e) throws InterruptedException;
    ...
}
```

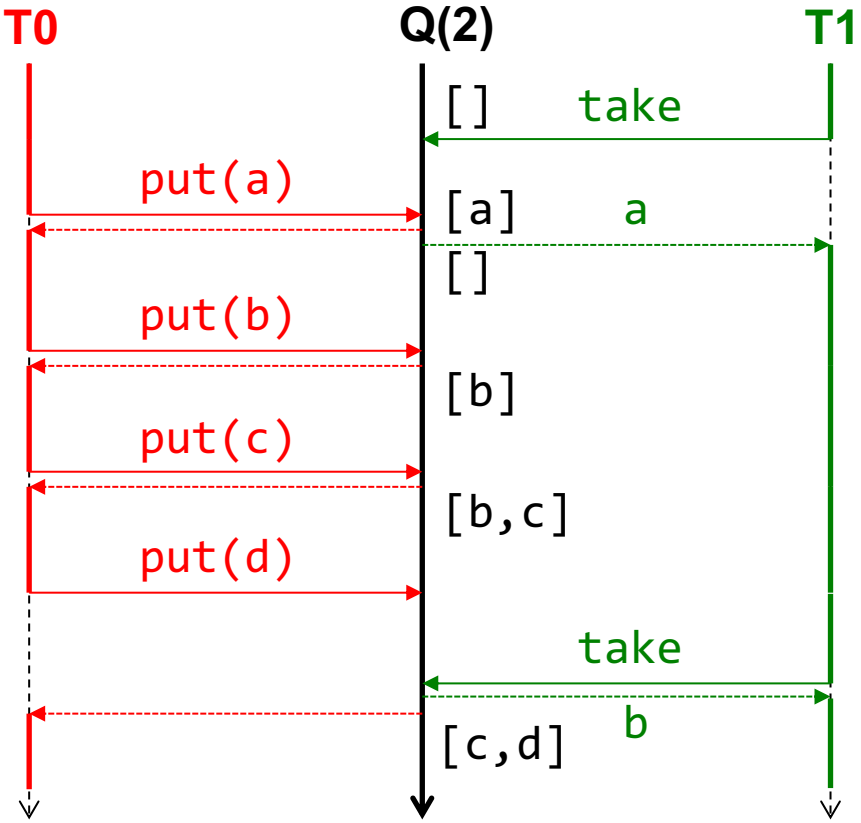
- **Usage: Decoupling producers from consumers**
  - Controlling load (e.g. Webserver)
  - Thread confined architectures (e.g. Swing Event Dispatcher)

Producer-Consumer Design Pattern

- **Separates identification of work (producers) from execution of that work (consumers) by putting work items on a todo list**
  - Decouples producer from consumer
  - Producers don't need to know how many consumers there are, and vice versa
  - ➢ Cleaner design
- **Simplifies load / resource management**
  - by limiting the size of the queue



BlockingQueue - Behavior



Producer Consumer Example

```
class Setup {
    void main() {
        BlockingQueue<Data> q = new LinkedBlockingQueue<>();
        Producer p = new Producer(q);
        Consumer c1 = new Consumer(q);
        Consumer c2 = new Consumer(q);
        new Thread(p).start();
        new Thread(c1).start();
        new Thread(c2).start();
    }
}
```

Producer Consumer Example cont.

```
class Producer implements Runnable {
    private final BlockingQueue<Data> queue;
    Producer(BlockingQueue<Data> q) { queue = q;}
    public void run() {
        try { while (true) { queue.put(produce()); }
        } catch (InterruptedException ex) {}
    }
    Data produce() { ... }
}
class Consumer implements Runnable {
    private final BlockingQueue<Data> queue;
    Consumer(BlockingQueue<Data> q) { queue = q; }
    public void run() {
        try { while (true) { consume(queue.take()); }
        } catch (InterruptedException ex) { }
    }
    void consume(Data x) { ... }
}
```