More on Java Synchronization

D. Thiebaut
CSC352 Fall 2013

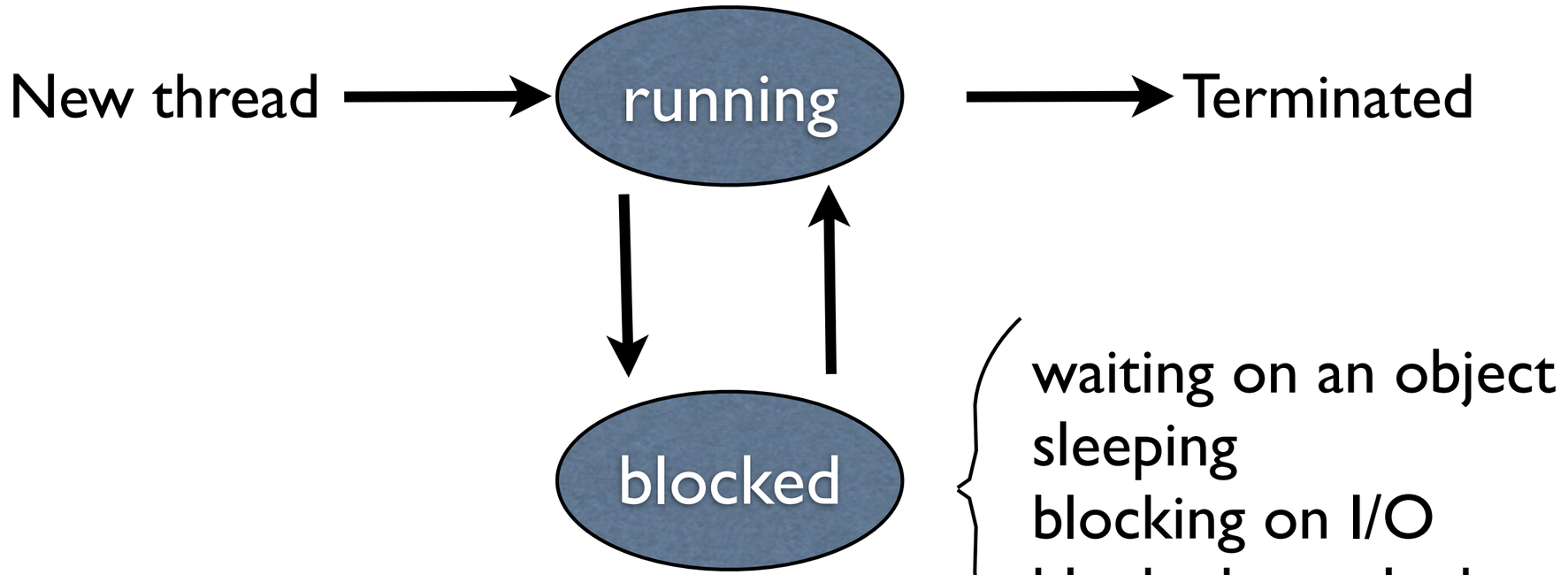
Outline

- Basics of Thread Operations (new stuff)
- Thread States
- Thread Scheduling
- Threads and I/O
- Producer/Consumer Pattern
- wait()/notify()
- Thread-Safe Data Structures
- Processing Lab

The Basics

- **Threads**
  - `run()`/`start()`
  - `yield()`
  - `sleep()`
  - `join()`
  - `wait()` and `notify()`, and also `notifyAll()`

States of a Thread



How to get the state?

http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/Thread.State.html

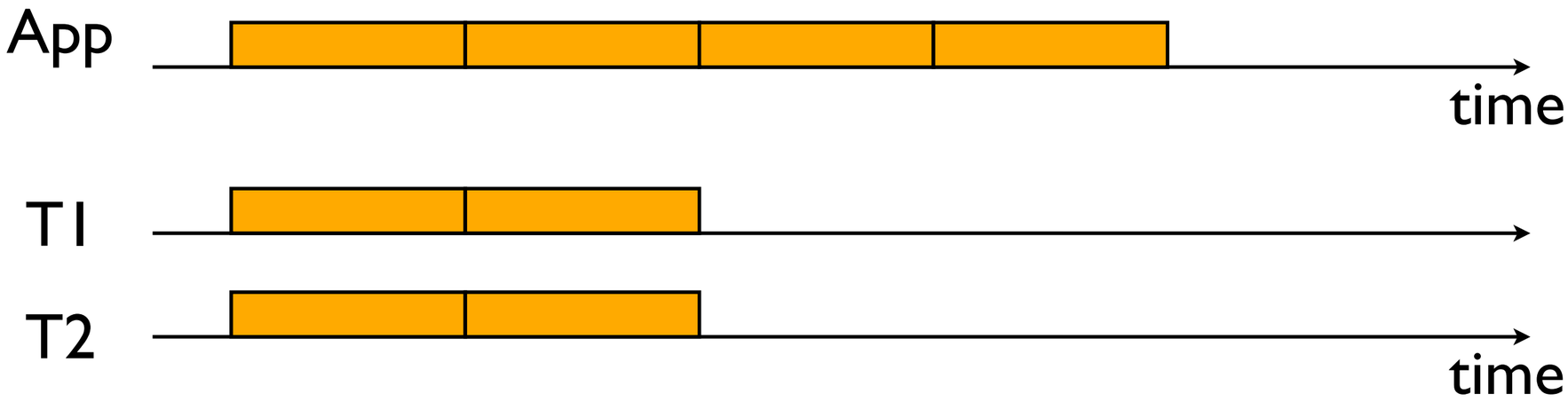
- NEW
- RUNNABLE
- BLOCKED
- WAITING
- TIME_WAITING
- TERMINATED

```
getState()
```

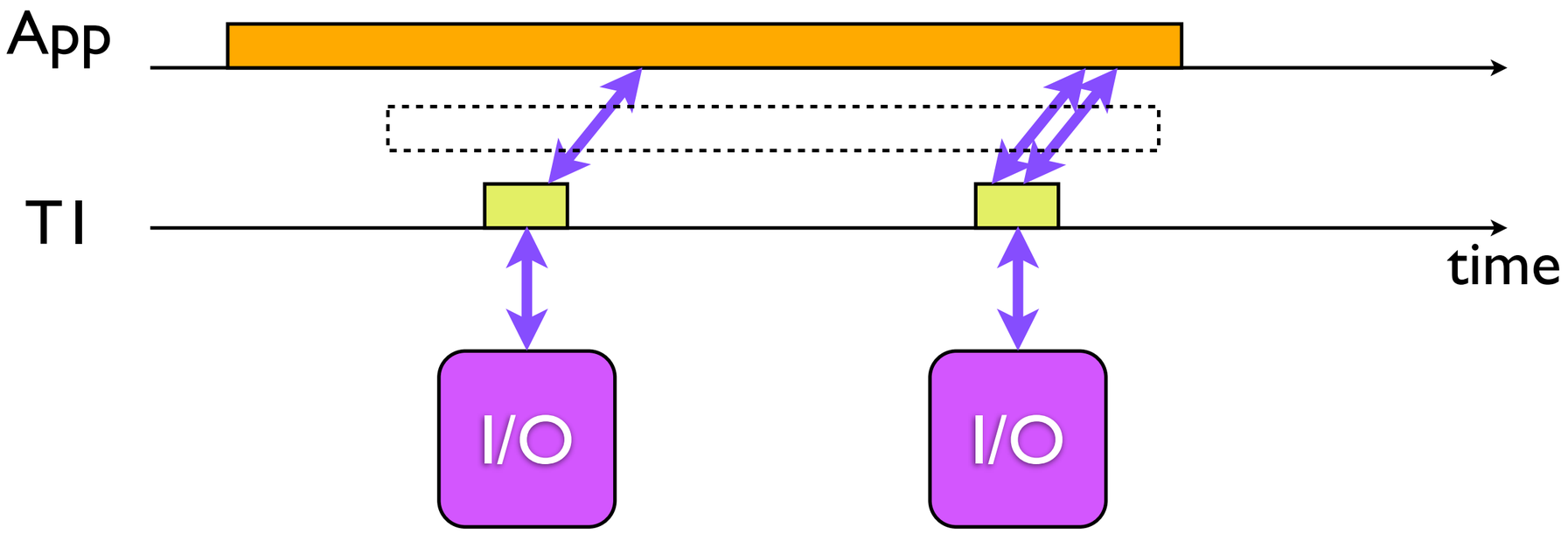
Thread Scheduling

- What is the policy?
  - Java doc says: Implemented in the JVM, preemptive, based on priority. (No mention of time-slices.)
  - 1= low priority, 5 = main, 10 = high priority
  - `getPriority()` & `setPriority()`
  - However, most OS implement time-slices (quanta), roughly 1ms, preemptive, and round-robin ==> JVMs do the same

Threads good not only for speedup



Threads good not only for speedup But also to simplify code



Important Concepts

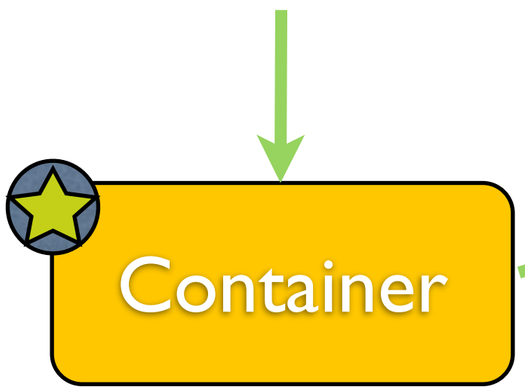
- CPU Bound Processes/Threads
- I/O Bound Processes/Threads

Time Scale

- Why I/O recognizing I/O-bound process is important
  - CPU cycle: 1 ns
  - RAM cycle: 100-500 ns
  - Disk access = seek + latency
    - seek = 1 ms
    - latency = 1/2 rotation, at 7,000 RPM
  - Question: How long does the processor wait for data from disk?

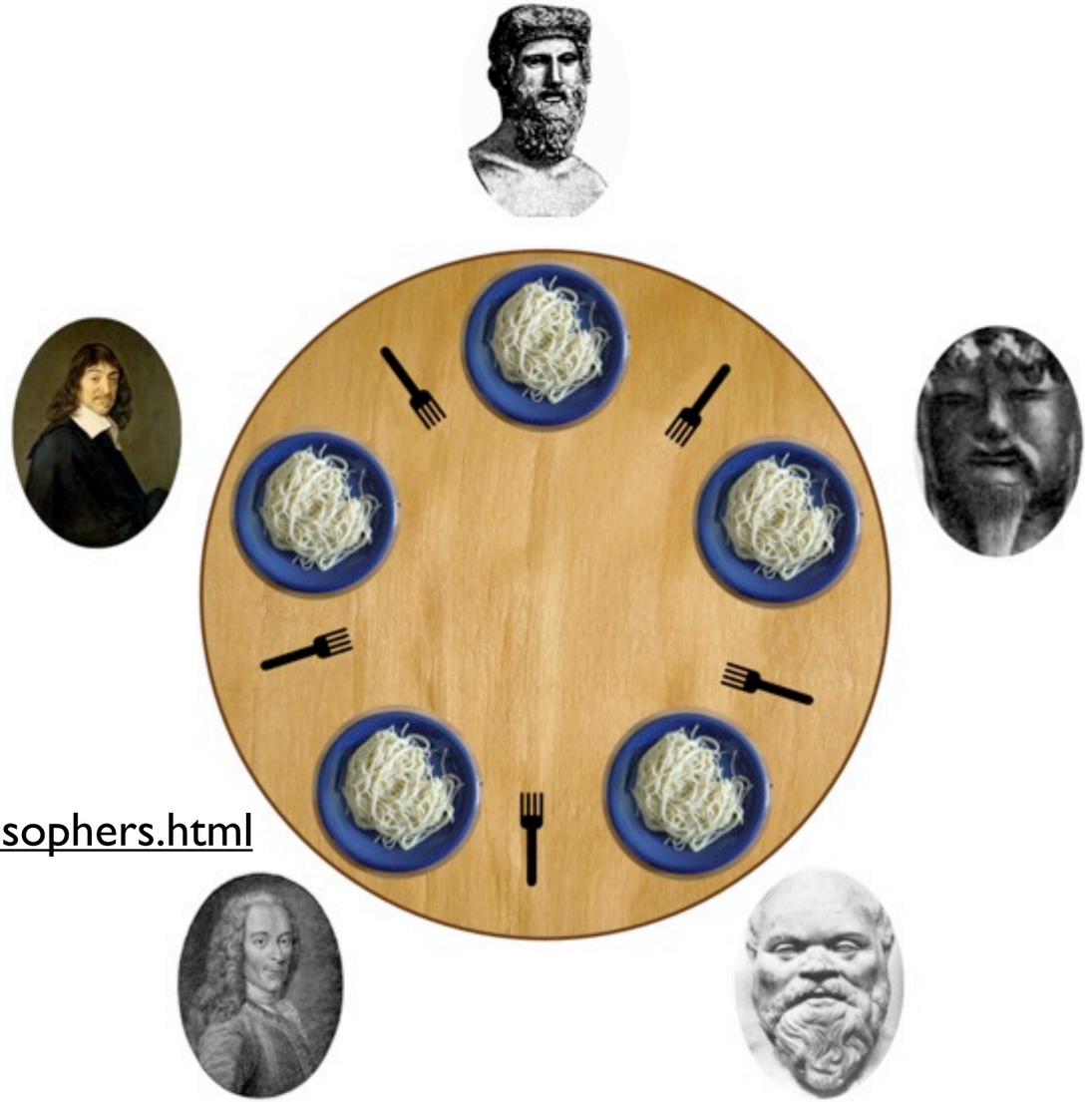
Wait/Notify Example
(Producer-Consumer)

```
// producer code
synchronized( lock ) {
   while ( !container.isEmpty() ) {
      try {
         lock.wait();
      } catch (InterruptedException e)
      {}
   }
   container.put( newItem );
}
```




```
//consumer code
synchronized( lock ) {
   if ( !container.isEmpty() ) {
      item = container.getItem();
      consume( item );
      lock.notify();
   }
}
```

Beware of Deadlocks!



- The Dining-Philosophers Problem
  http://en.wikipedia.org/wiki/Dining_philosophers_problem

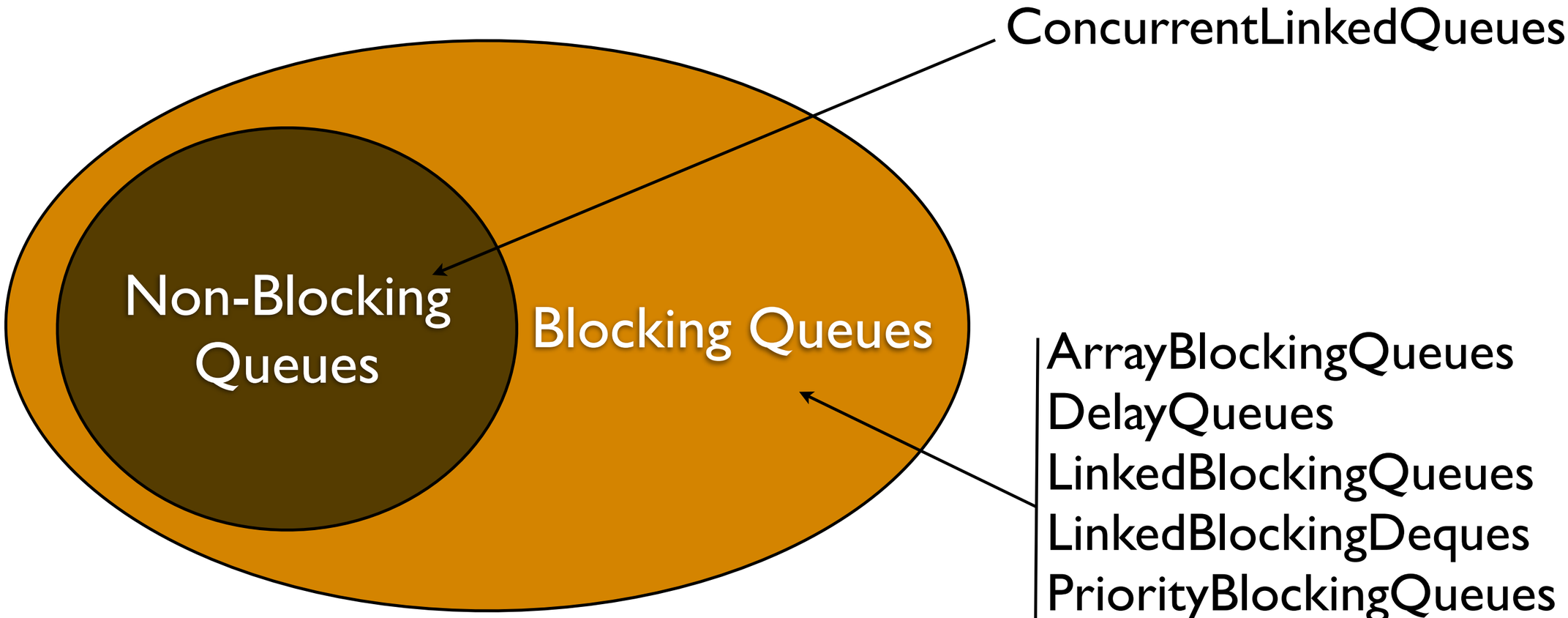
- The Applet

  http://elvis.rowan.edu/~hartley/ConcProgJava/Applets/diningPhilosophers.html

Rule #1 for Preventing Deadlocks

- Grab all the data-structures that you need first
- If you can't, release them all
- Wait a random amount of time and try again

Thread-Safe Data Structures



ConcurrentLinkedQueue

Method Summary	
boolean	**add**(E e) Inserts the specified element at the tail of this queue.
boolean	**contains**(Object o) Returns `true` if this queue contains the specified element.
boolean	**isEmpty**() Returns `true` if this queue contains no elements.
Iterator<E>	**iterator**() Returns an iterator over the elements in this queue in proper sequence.
boolean	**offer**(E e) Inserts the specified element at the tail of this queue.
E	**peek**() Retrieves, but does not remove, the head of this queue, or returns `null` if this queue is empty.
E	**poll**() Retrieves and removes the head of this queue, or returns `null` if this queue is empty.
boolean	**remove**(Object o) Removes a single instance of the specified element from this queue, if it is present.
int	**size**() Returns the number of elements in this queue.
Object[]	**toArray**() Returns an array containing all of the elements in this queue, in proper sequence.

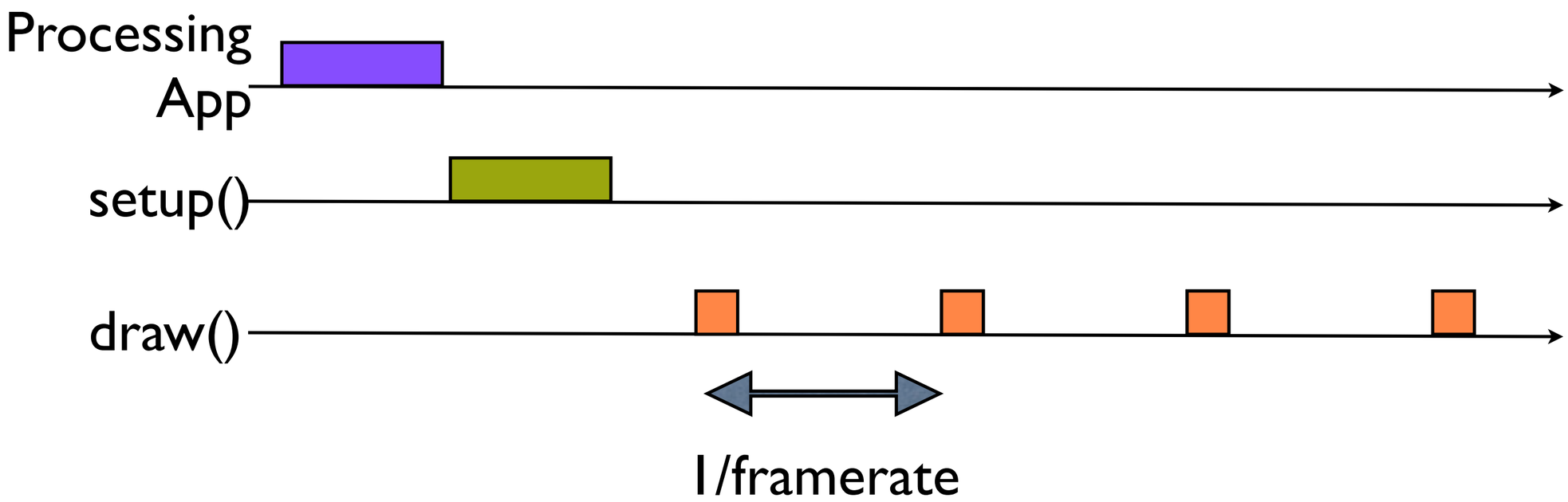
ArrayBlockingQueue

Method Summary	
void	**clear**() Removes all of the elements from this collection.
boolean	**contains**(Object o) Returns `true` if this collection contains the specified element.
int	**drainTo**(Collection<? super E> c) Removes all available elements from this queue and adds them into the given collection.
int	**drainTo**(Collection<? super E> c, int maxElements) Removes at most the given number of available elements from this queue and adds them into the given collection.
Iterator<E>	**iterator**() Returns an iterator over the elements in this queue in proper sequence.
boolean	**offer**(E o) Inserts the specified element at the tail of this queue if possible, returning immediately if this queue is full.
boolean	**offer**(E o, long timeout, TimeUnit unit) Inserts the specified element at the tail of this queue, waiting if necessary up to the specified wait time for space to become available.
E	**peek**() Retrieves, but does not remove, the head of this queue, returning `null` if this queue is empty.
E	**poll**() Retrieves and removes the head of this queue, or `null` if this queue is empty.
E	**poll**(long timeout, TimeUnit unit) Retrieves and removes the head of this queue, waiting if necessary up to the specified wait time if no elements are present on this queue.
void	**put**(E o) Adds the specified element to the tail of this queue, waiting if necessary for space to become available.
int	**remainingCapacity**() Returns the number of elements that this queue can ideally (in the absence of memory or resource constraints) accept without blocking.

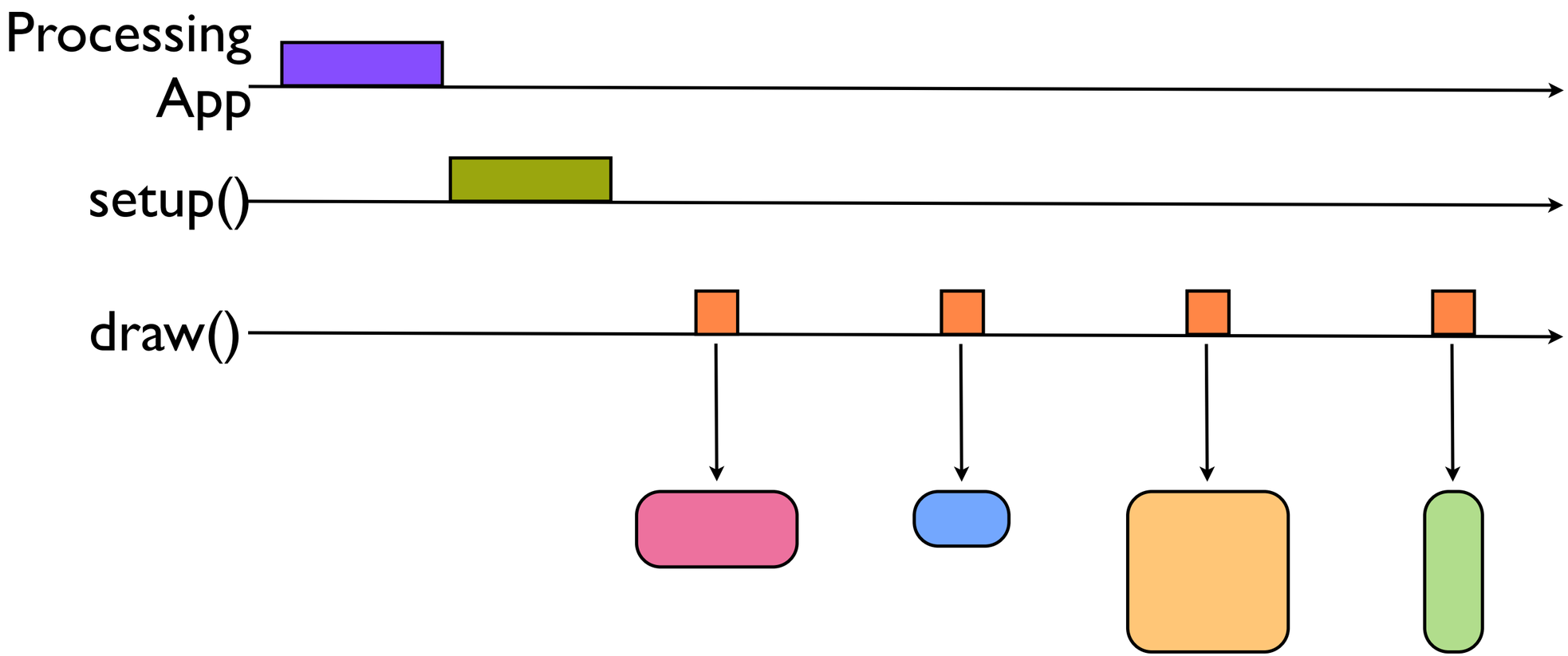
ArrayBlockingQueue (cont'd)

int	**remainingCapacity**() Returns the number of elements that this queue can ideally (in the absence of memory or resource constraints) accept without blocking.
boolean	**remove**(Object o) Removes a single instance of the specified element from this collection, if it is present (optional operation).
int	**size**() Returns the number of elements in this queue.
E	**take**() Retrieves and removes the head of this queue, waiting if no elements are present on this queue.
Object[]	**toArray**() Returns an array containing all of the elements in this collection.
<T> T[]	**toArray**(T[] a) Returns an array containing all of the elements in this collection; the runtime type of the returned array is that of the specified array.
String	**toString**() Returns a string representation of this collection.

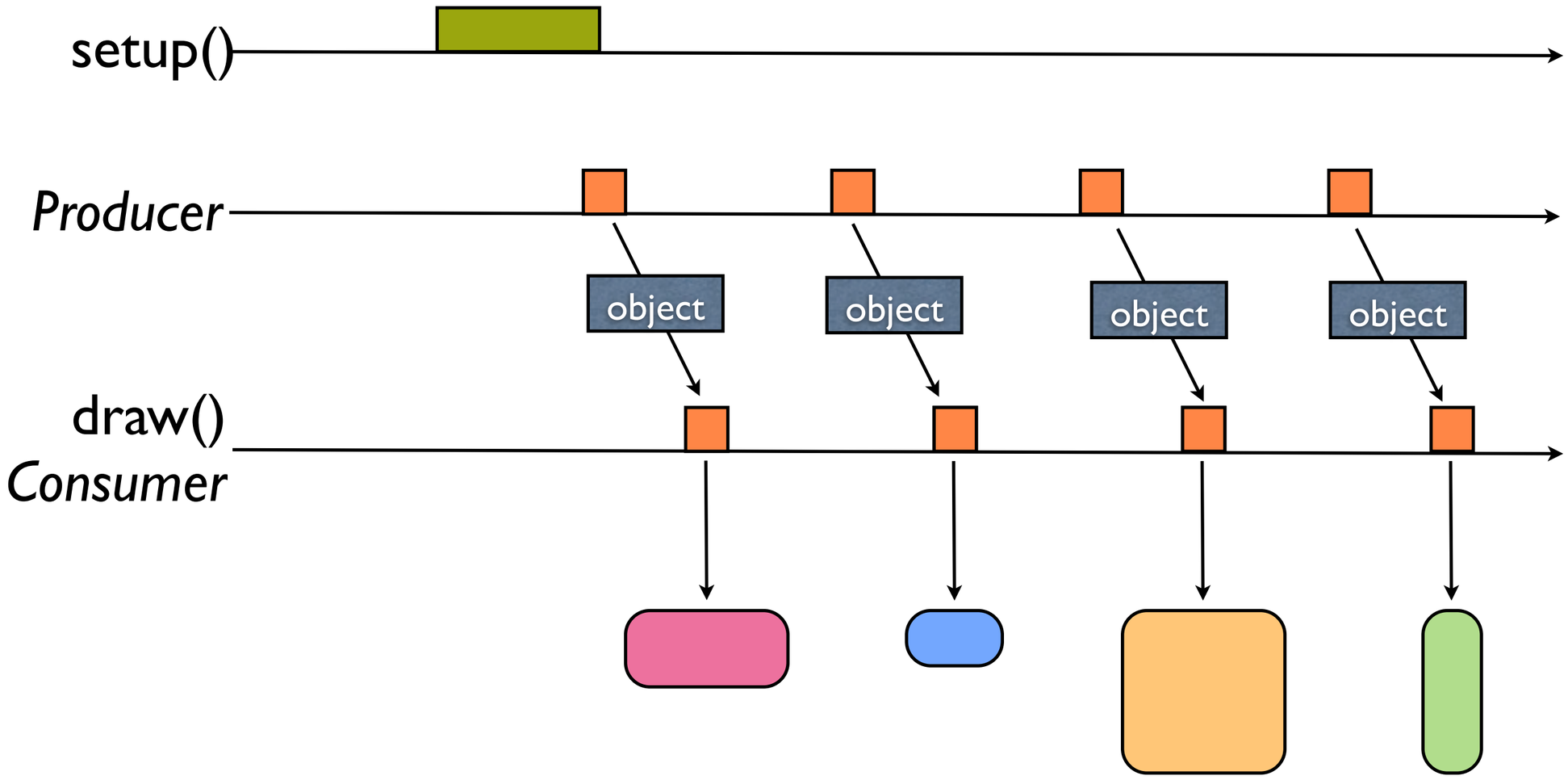
Processing Lab



Processing Lab: Version 1



Processing Lab: Version 1



Processing Lab: Version 2