Concurrency and Correctness

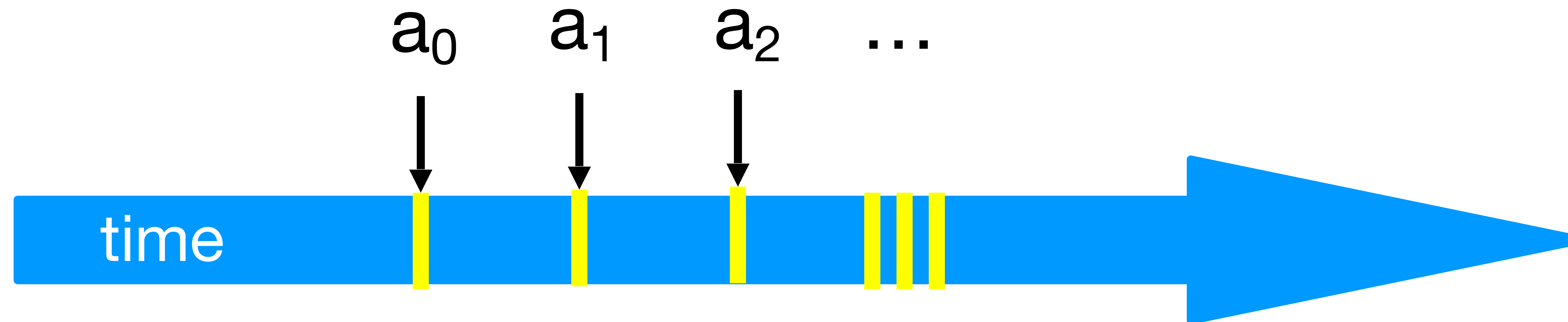
CS 475, Fall 2019

Concurrent & Distributed Systems

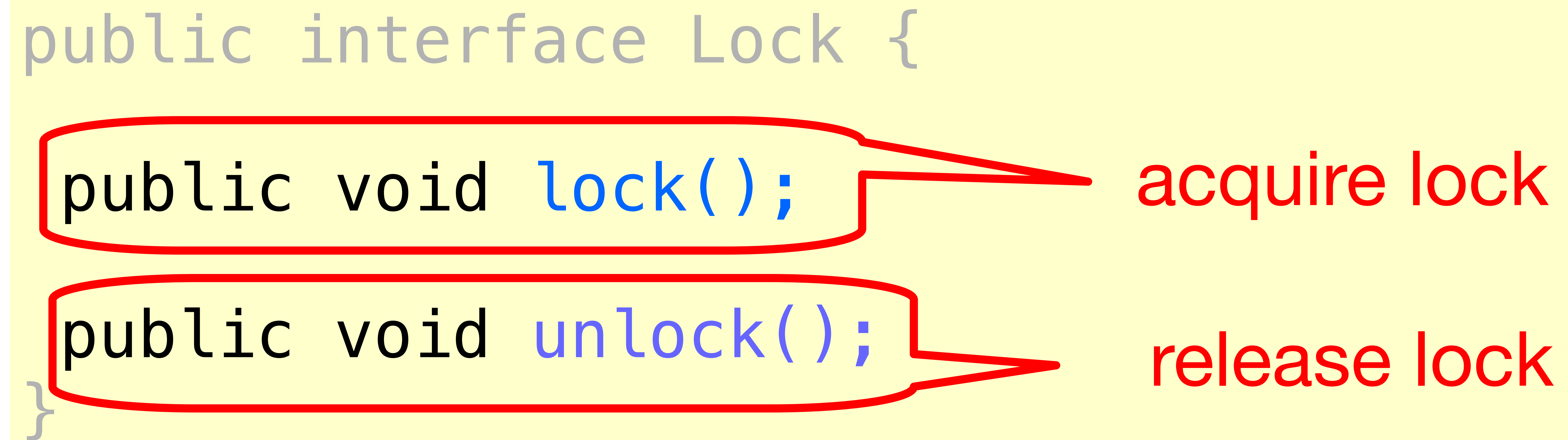
With material from Herlihy & Shavit, Art of Multiprocessor Programming

Threads

- A thread A is (formally) a sequence a_0, a_1, ... of events
  - "Trace" model
  - Notation: $a_0 \rightarrow a_1$ indicates order



Locks (Mutual Exclusion)



Peterson's Algorithm

```
public void lock() {
 flag[i] = true;
 victim  = i;
 while (flag[j] && victim == i) {};
 }
public void unlock() {
 flag[i] = false;
}
```

Announce I'm interested

Defer to other

Wait while other interested & I'm the victim

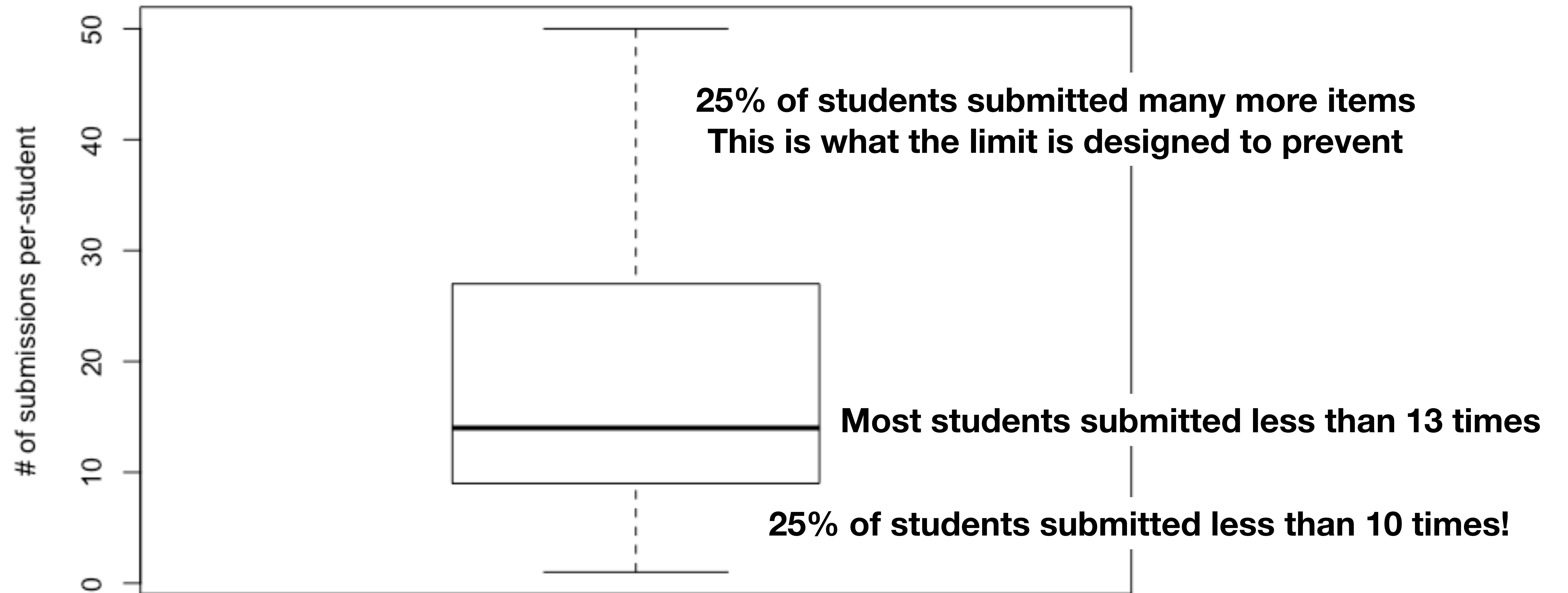
No longer interested

Where `i` is the index of the current thread, `j` is the index of the other thread

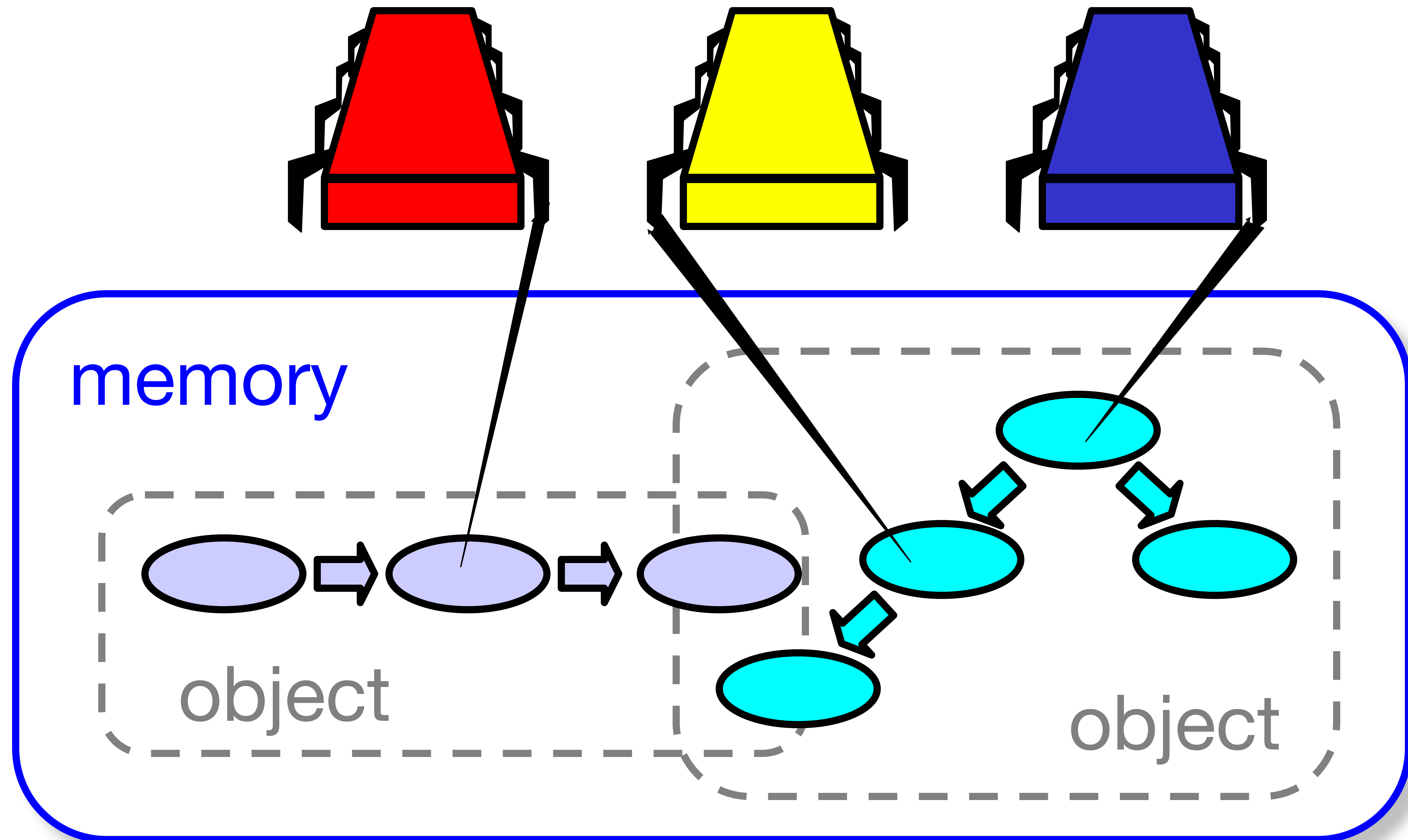
Today

- Concurrent specifications
  - Linearizability
  - Reasoning about correctness
- Reading: H&S 2.1-2.3
- Note: HW1 posted: https://www.jonbell.net/gmu-cs-475-fall-2019/homework-1/

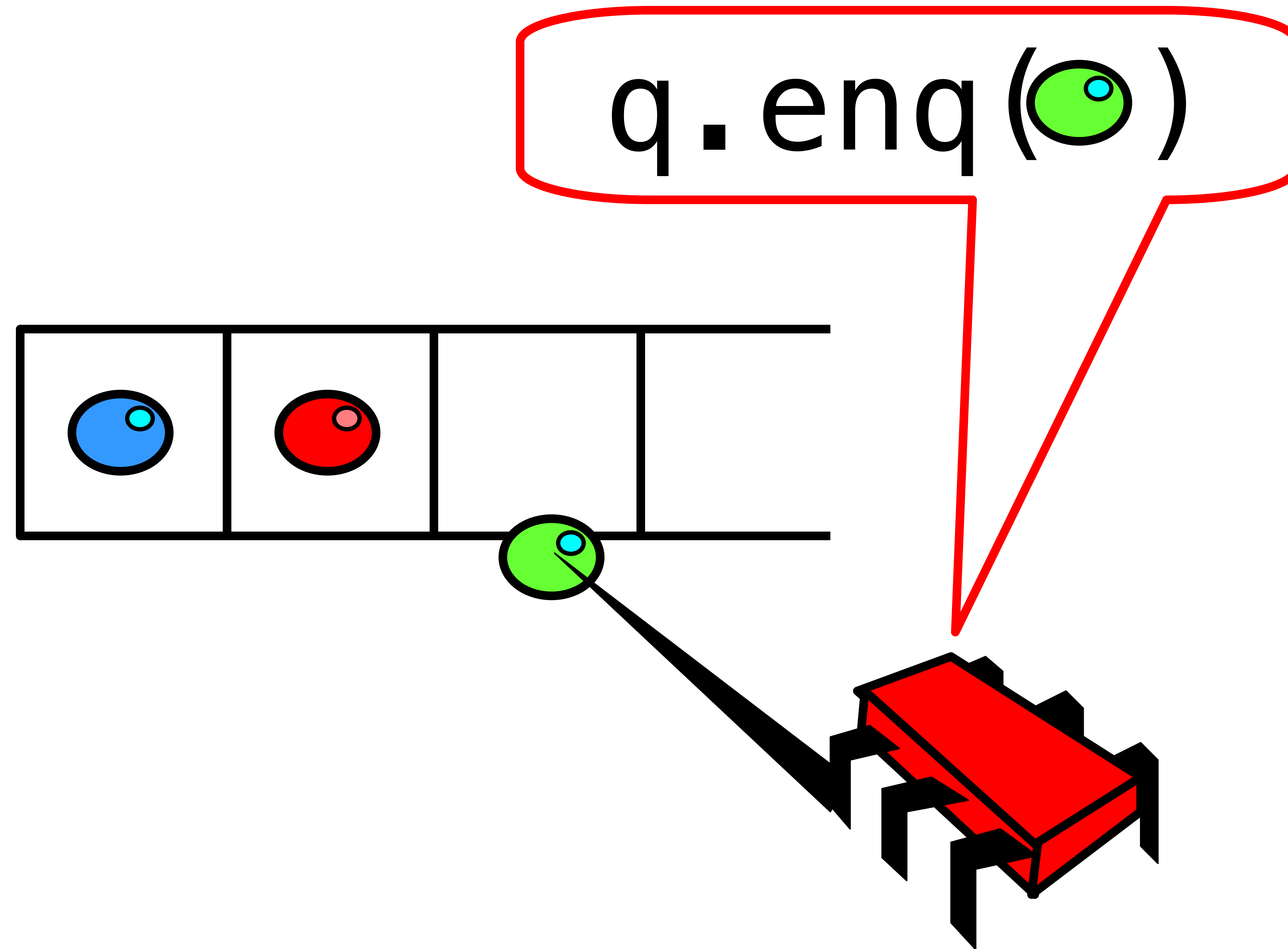
HW1 - Autolab Limit Note



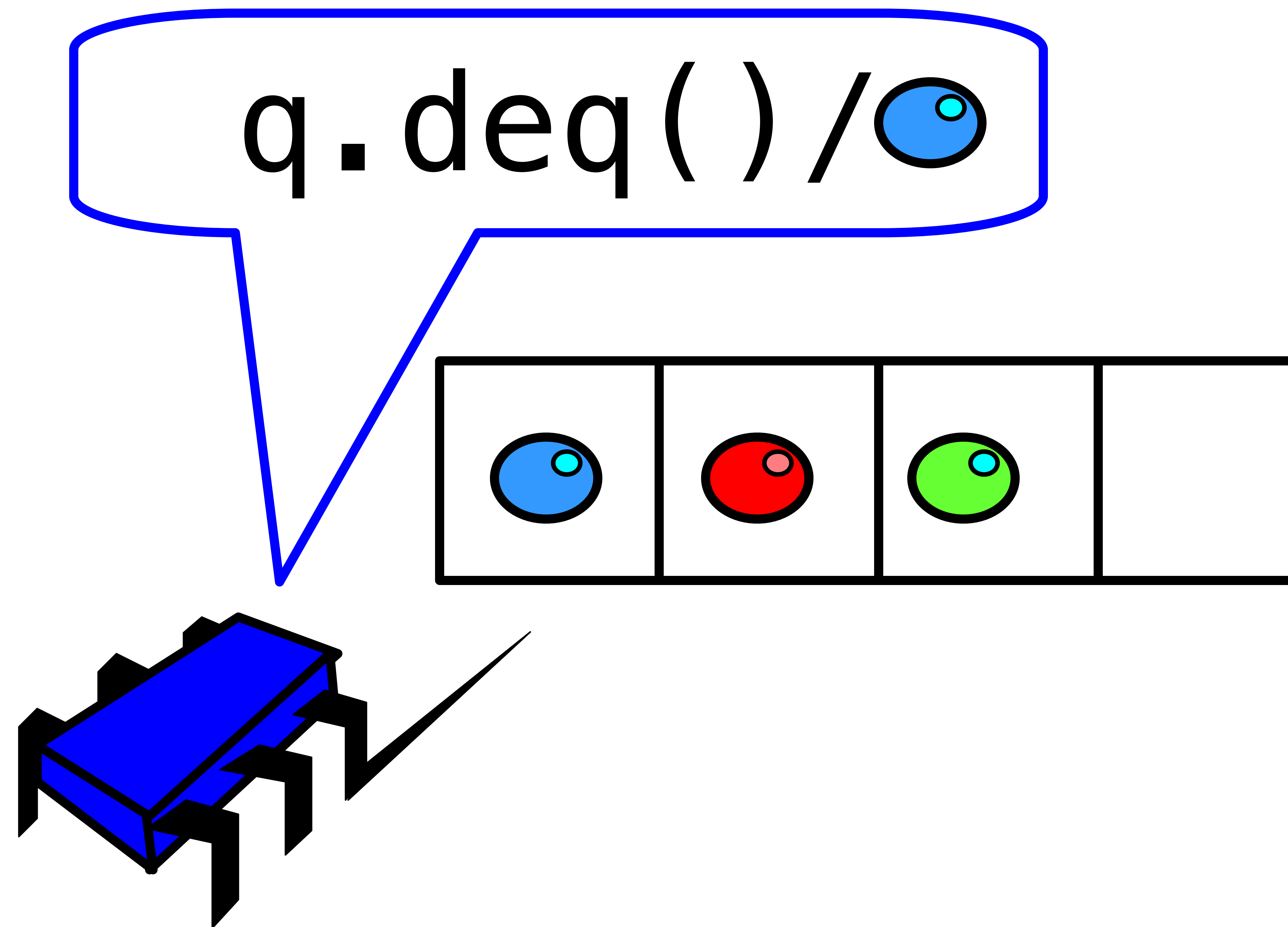
Concurrent Computation



FIFO Queue: Enqueue Method



FIFO Queue: Dequeue Method



A Lock-Based Queue

```
class LockBasedQueue<T> {
  int head, tail;
  T[] items;
  Lock lock;
  public LockBasedQueue(int capacity) {
    head = 0; tail = 0;
    lock = new ReentrantLock();
    items = (T[]) new Object[capacity];
}
```

A Lock-Based Queue

```
class LockBasedQueue<T> {
  int head, tail;
  T[] items;
  Lock lock;
  public LockBasedQueue(int capacity) {
    head = 0; tail = 0;
    lock = new ReentrantLock();
    items = (T[]) new Object[capacity];
}
```

Queue fields protected by single shared lock

Implementation: Deq (Enq is similar)

```
public T deq() throws EmptyException {
  lock.lock();
  try {
    if (tail == head)
       throw new EmptyException();
    T x = items[head % items.length];
    head++;
    return x;
  } finally {
    lock.unlock();
  }
}
```

Implementation: Deq

```
public T deq() throws EmptyException {
  lock.lock();
  try {
    if (tail == head)
       throw new EmptyException();
    T x = items[head % items.length];
    head++;
    return x;
  } finally {
    lock.unlock();
  }
}
```

Should be correct because modifications are mutually exclusive…

Now consider the following implementation

- The same thing without mutual exclusion
  - Remember Amdahl’s law?
- For simplicity, only two threads
  - One thread enq only
  - The other deq only

Wait-free 2-Thread Queue

```
public class WaitFreeQueue {

  int head = 0, tail = 0;
  items = (T[]) new Object[capacity];

  public void enq(Item x) {
    while (tail-head == capacity); // busy-wait
    items[tail % capacity] = x; tail++;
  }
  public Item deq() {
     while (tail == head);     // busy-wait
     Item item = items[head % capacity]; head+
+;
     return item;
}}
```

Wait-free 2-Thread Queue

```
public class LockFreeQueue {

  int head = 0, tail = 0;
  items = (T[]) new Object[capacity];

  public void enq(Item x) {
    while (tail-head == capacity); // busy-wait
    items[tail % capacity] = x; tail++;
  }
  public Item deq() {
     while (tail == head);     // busy-wait
     Item item = items[head % capacity]; head+
+;
     return item;
}}
```

Lock-free 2-Thread Queue

```
public class LockFreeQueue {

  int head = 0, tail = 0;
  items = (T[]) new Object[capacity];

  public void enq(Item x) {
    while (tail-head == capacity); // busy-wait
    items[tail % capacity] = x; tail++;
  }
  public Item deq() {
     while (tail == head);
     Item item = items[head % c
```

```
}}
```

Queue is updated without a lock!

How do we define "correct" when modifications are not mutually exclusive?

Wait-free 2-Thread Queue

Argument for why this is OK (for now):
No two threads ever write the same variable

```
public c
                               ;
                           ct[capacity];

  public void enq(Item x) {
    while (tail-head == capacity); // busy-wait
    items[tail % capacity] = x; tail++;
  }
  public Item deq() {
     while (tail == head);     // busy-wait
     Item item = items[head % capacity]; head++;
     return item;
}}
```

Writes items, writes tail

Writes head

Defining concurrent queue implementations

- Need a way to specify a concurrent queue object
- Need a way to prove that an algorithm implements the object's specification
- Lets talk about object specifications …

Correctness and Progress

- In a concurrent setting, we need to specify both the safety and the liveness properties of an object
- Need a way to define
  - when an implementation is correct
  - the conditions under which it guarantees progress

Lets begin with correctness

Sequential Objects

- Each object has a **state**
  - Usually given by a set of **fields**
  - Queue example: sequence of items
- Each object has a set of **methods**
  - Only way to manipulate state
  - Queue example: **enq** and **deq** methods

Sequential Specifications

- If (precondition)
  - the object is in such-and-such a state
  - before you call the method,
- Then (postcondition)
  - the method will return a particular value
  - or throw a particular exception.
- and (postcondition, con't)
  - the object will be in some other state
  - when the method returns,

Pre and PostConditions for Dequeue

- Precondition:
  - Queue is non-empty
- Postcondition:
  - Returns first item in queue
- Postcondition:
  - Removes first item in queue

Pre and PostConditions for Dequeue

- Precondition:
  - Queue is empty
- Postcondition:
  - Throws Empty exception
- Postcondition:
  - Queue state unchanged

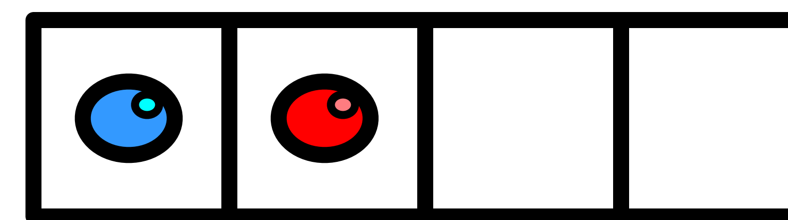
Why Sequential Specifications Totally Rock

- Interactions among methods captured by side-effects on object state
  - State meaningful between method calls
- Documentation size linear in number of methods
  - Each method described in isolation
- Can add new methods
  - Without changing descriptions of old methods

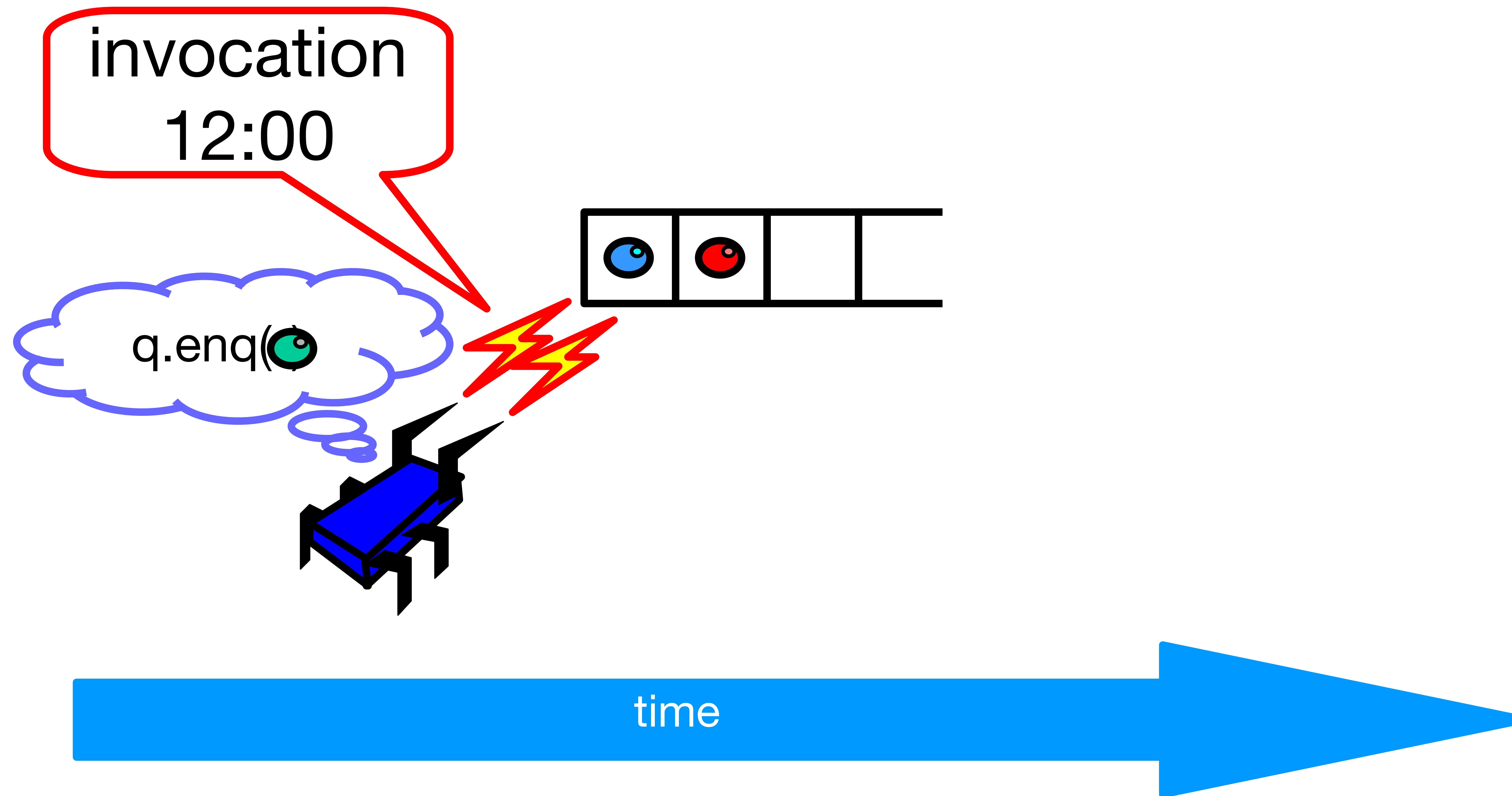
What About Concurrent Specifications ?

- Methods?
- Documentation?
- Adding new methods?

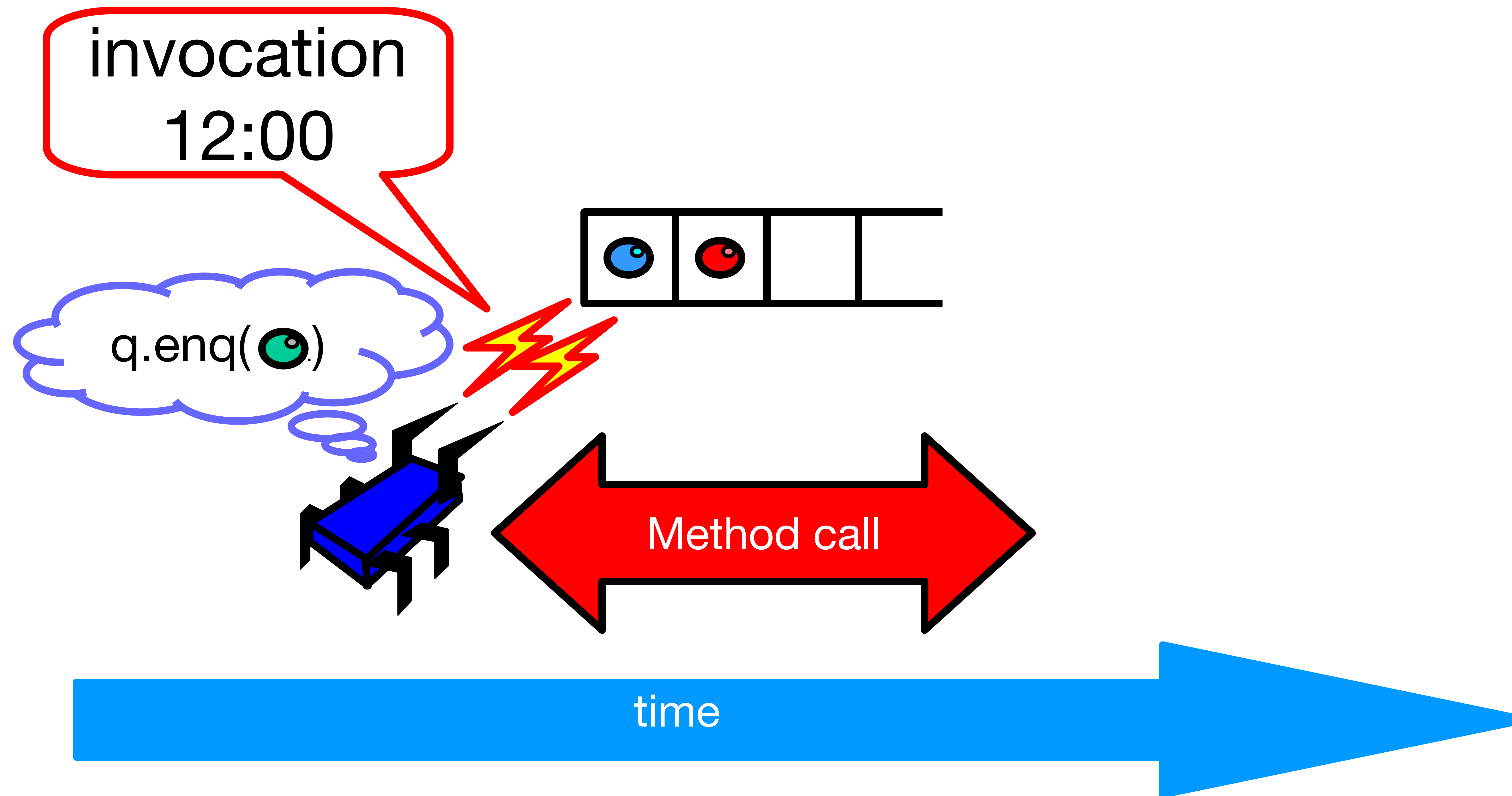
Methods Take Time



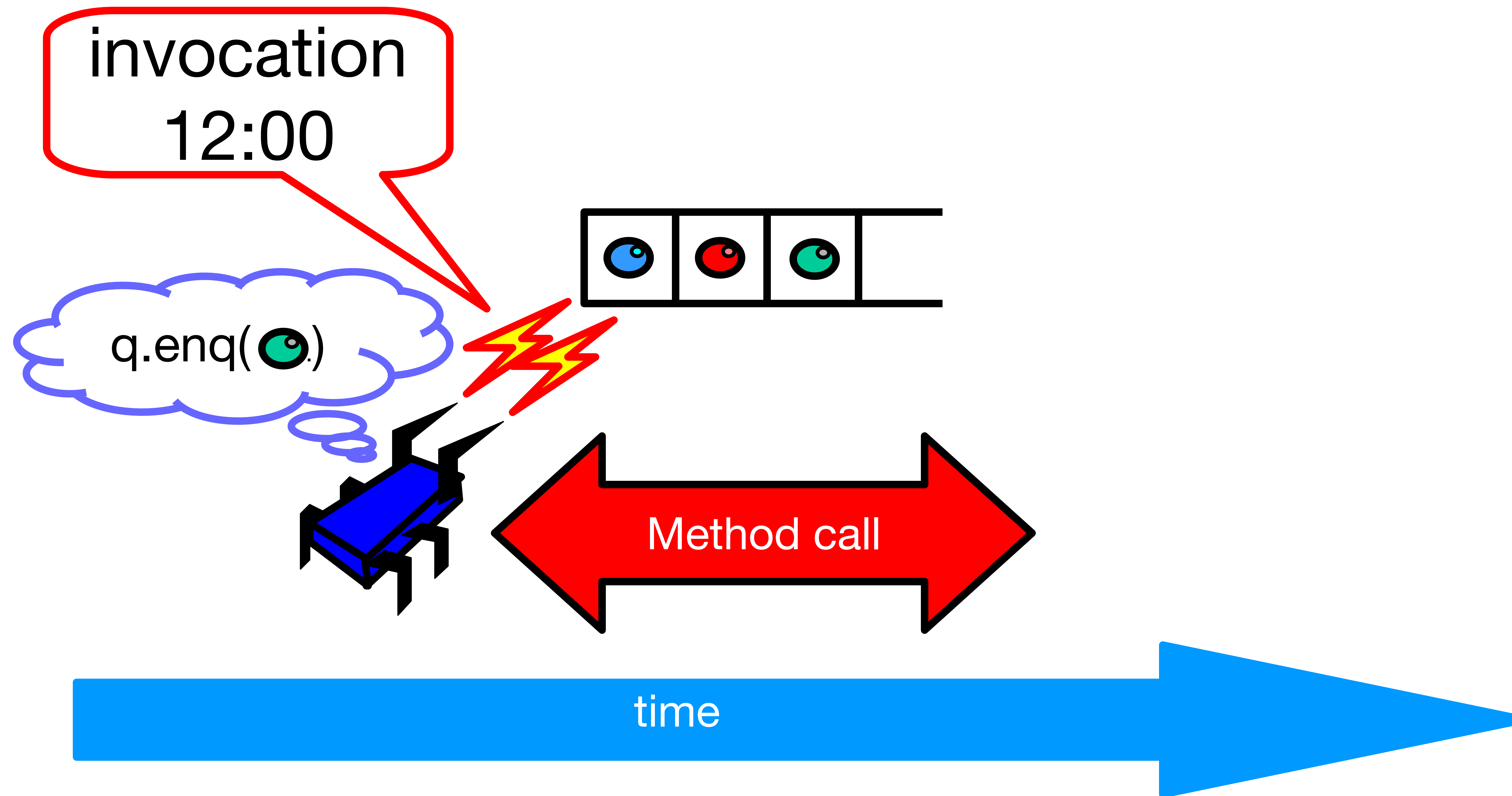
Methods Take Time



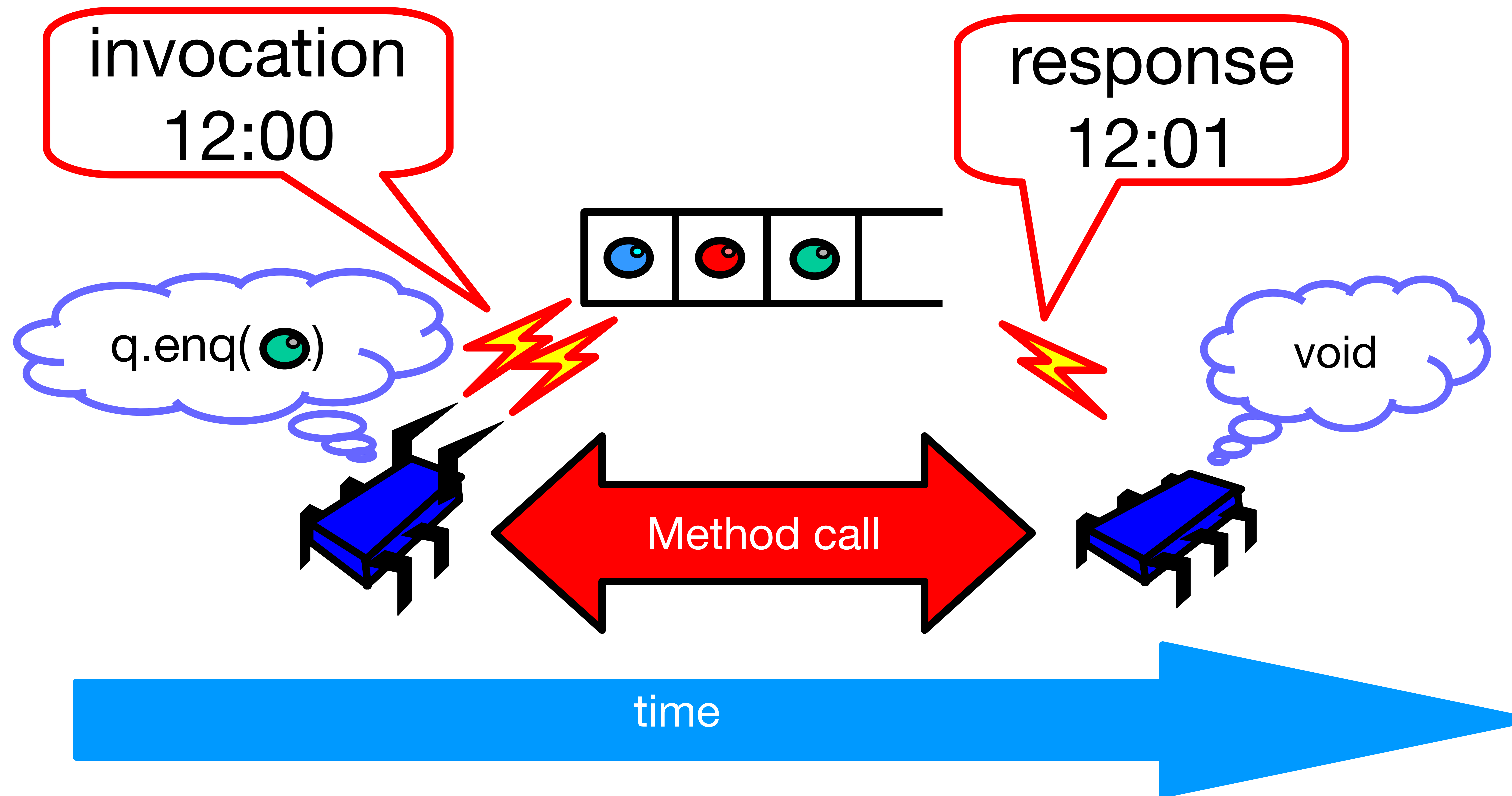
Methods Take Time



Methods Take Time



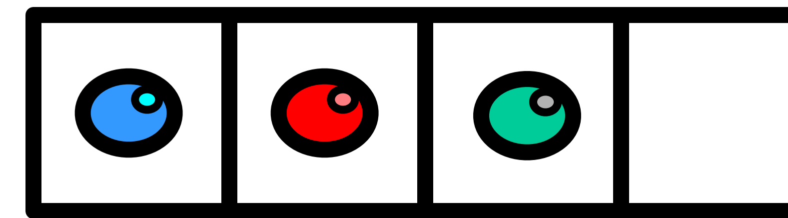
Methods Take Time



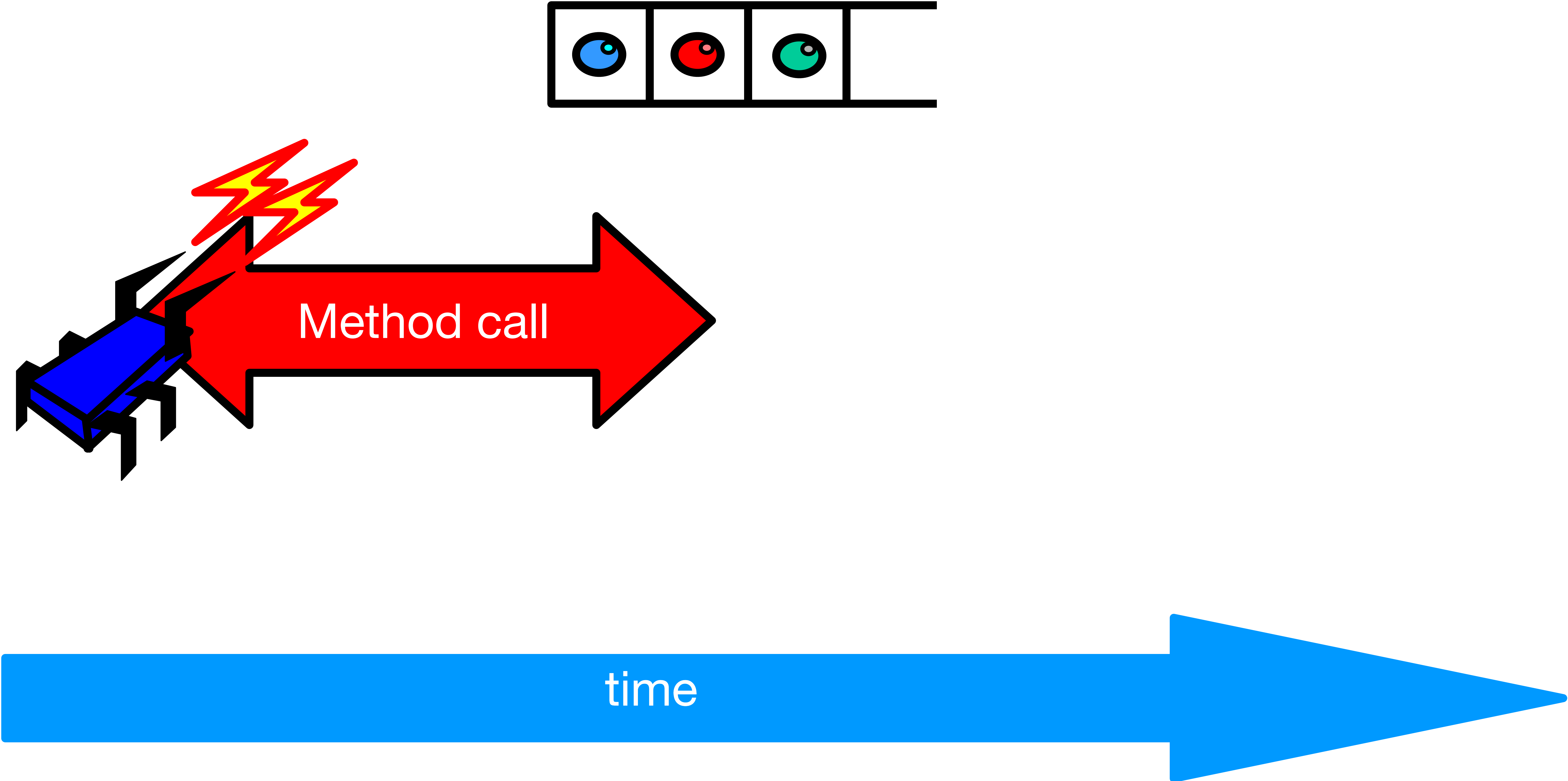
Sequential vs Concurrent

- Sequential
  - Methods take time? Who knew?
- Concurrent
  - Method call is not an event
  - Method call is an interval.

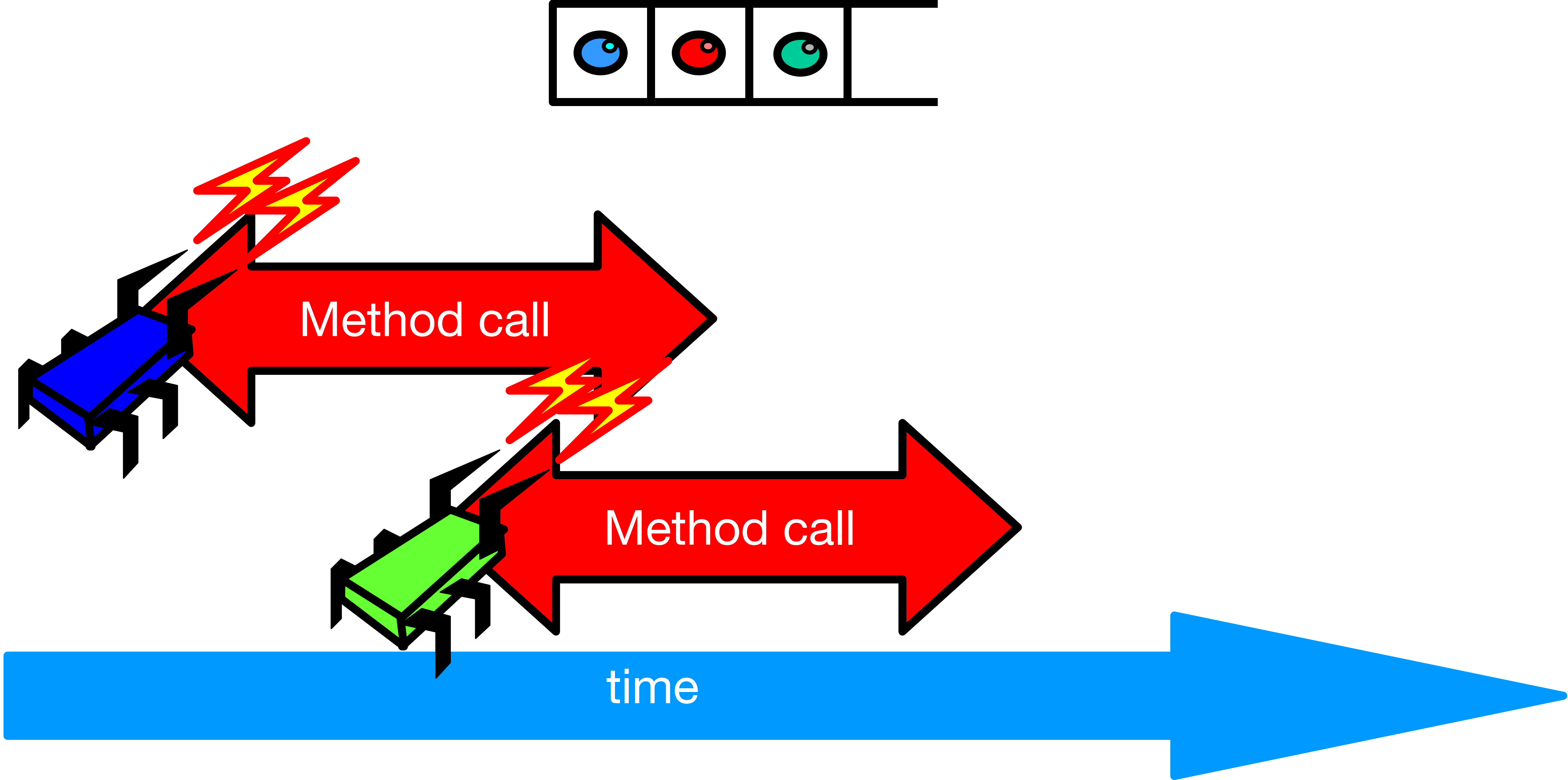
Concurrent Methods Take Overlapping Time



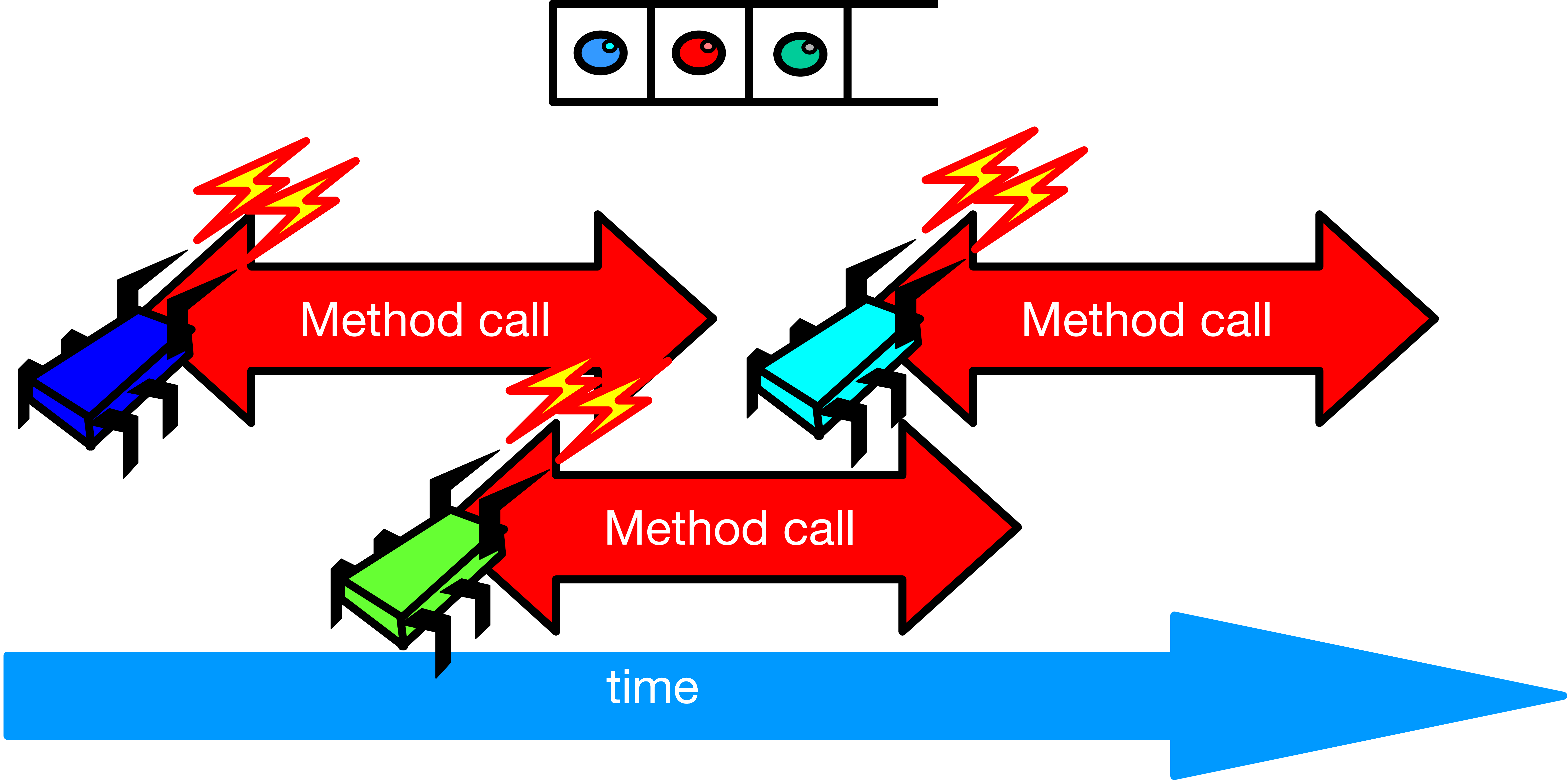
Concurrent Methods Take Overlapping Time



Concurrent Methods Take Overlapping Time



Concurrent Methods Take Overlapping Time



Sequential vs Concurrent

- Sequential:
  - Object needs meaningful state only between method calls
- Concurrent
  - Because method calls overlap, object might **never** be between method calls

Sequential vs Concurrent

- Sequential:
  - Each method described in isolation
- Concurrent
  - Must characterize **all** possible interactions with concurrent calls
    - What if two `enq`s overlap?
    - Two `deq`s? `enq` and `deq`? …

Sequential vs Concurrent

- Sequential:
  - Can add new methods without affecting older methods
- Concurrent:
  - Everything can potentially interact with everything else

Sequential vs Concurrent

- Sequential:
  - Can add new methods without affecting older methods
- Concurrent:
  - Everything can potentially interact with everything else

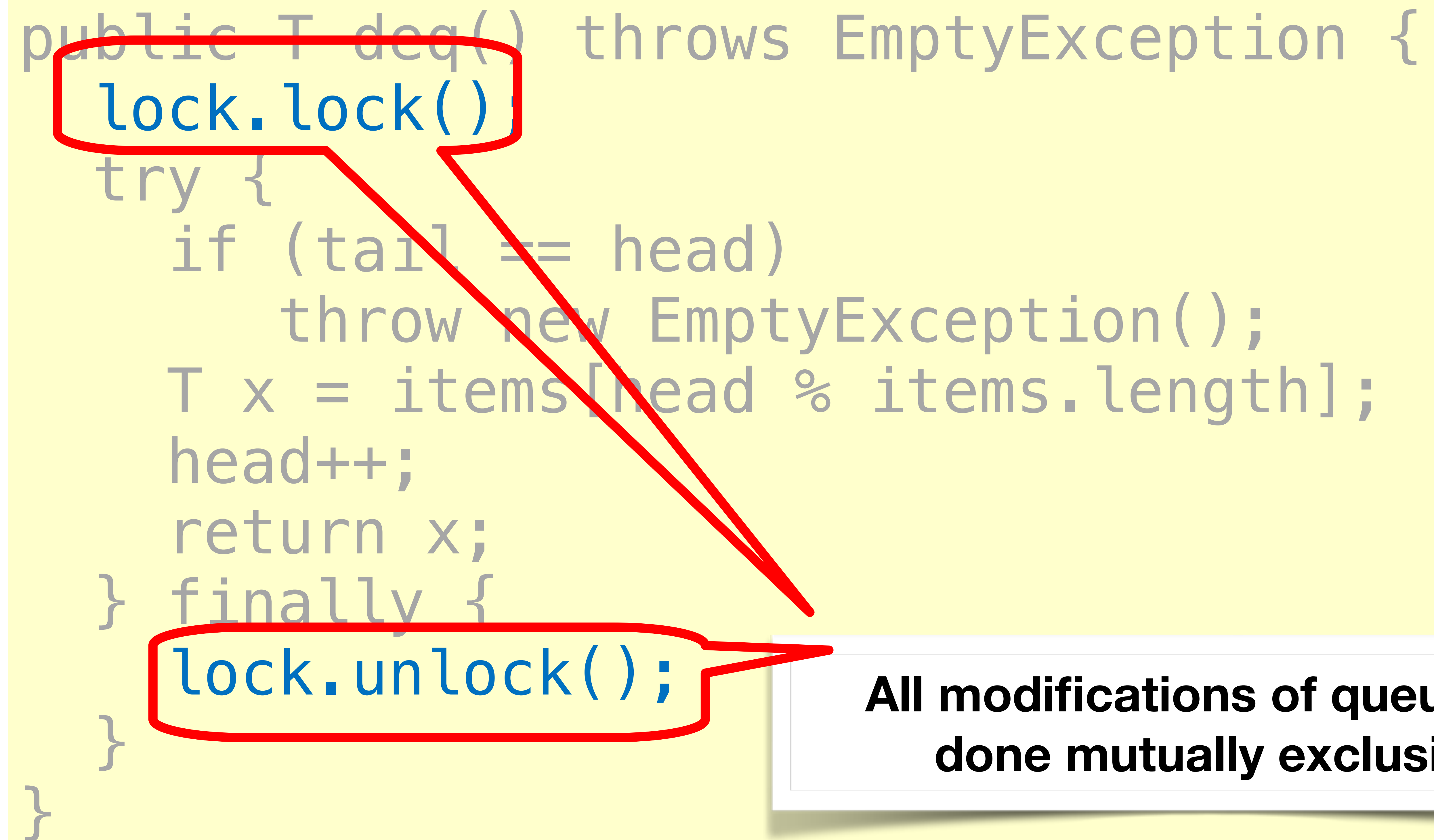
The Big Question

- What does it mean for a concurrent object to be correct?
  - What is a concurrent FIFO queue?
  - FIFO means strict temporal order
  - Concurrent means ambiguous temporal order

Intuitively...

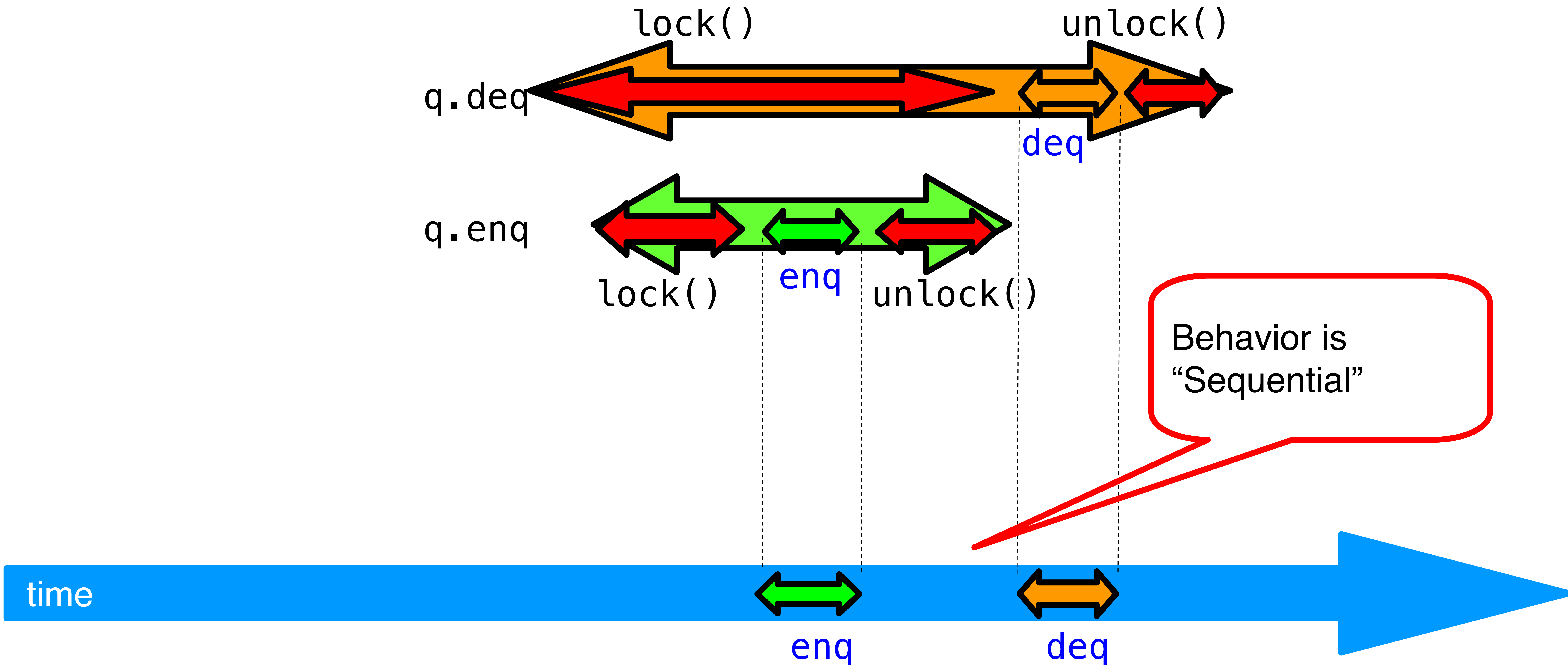
```
public T deq() throws EmptyException {
  lock.lock();
  try {
    if (tail == head)
       throw new EmptyException();
    T x = items[head % items.length];
    head++;
    return x;
  } finally {
    lock.unlock();
  }
}
```

Intuitively…



Intuitively

Lets capture the idea of describing the concurrent via the sequential model



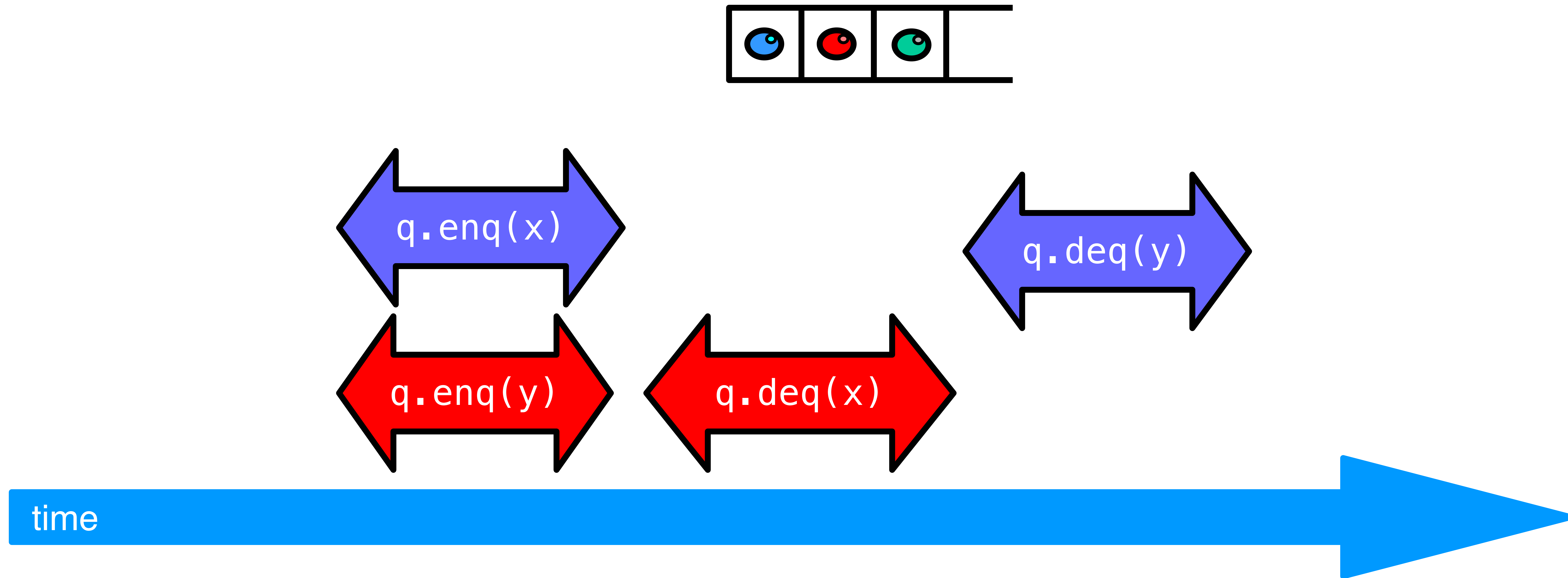
Linearizability

- Each method should
  - “take effect”
  - Instantaneously
  - Between invocation and response events
- Object is correct if this “sequential” behavior is correct
- Any such concurrent object is *Linearizable*

Is it really about the object?

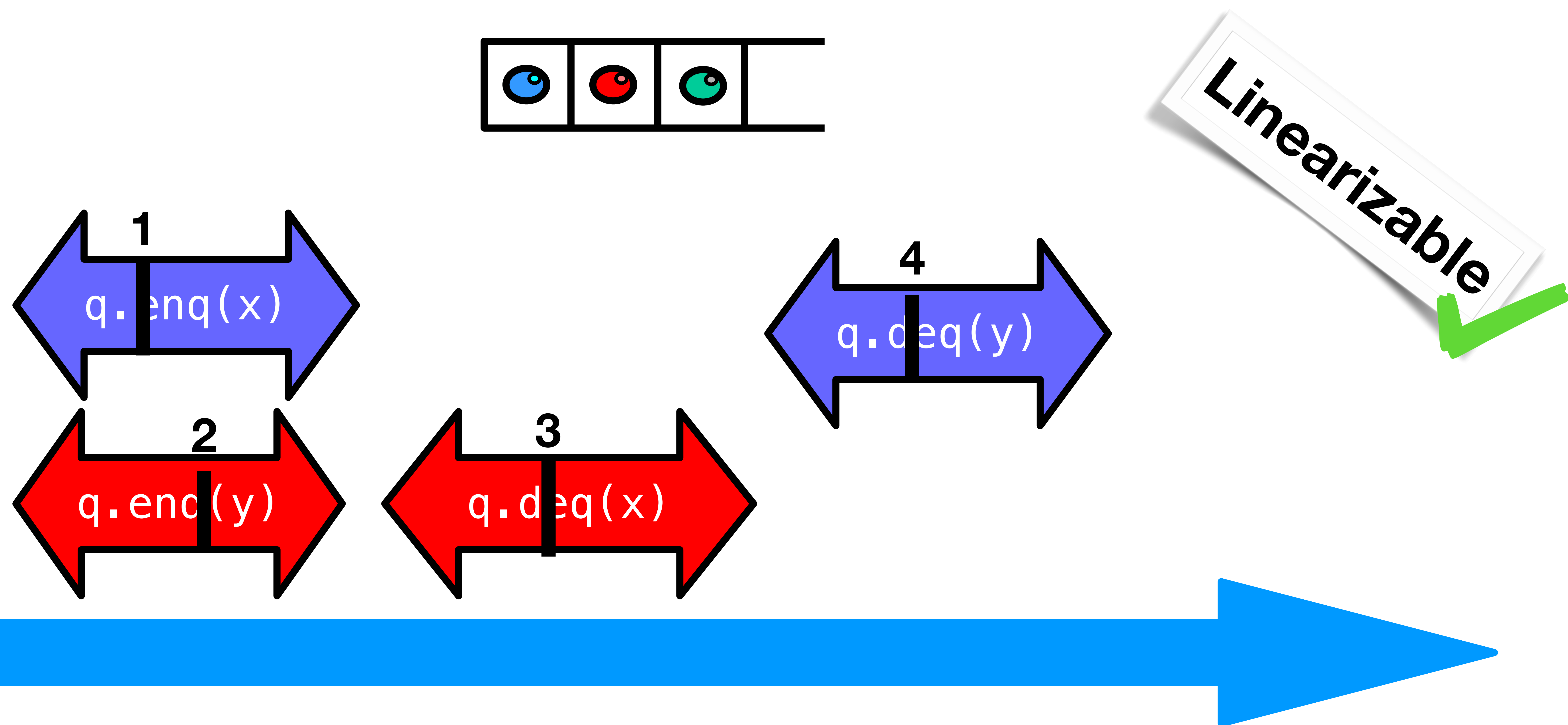
- Each method should
  - “take effect”
  - Instantaneously
  - Between invocation and response events
- Sounds like a property of an execution…
- A linearizable object: one all of whose possible executions are linearizable

Example



Example: Linearizable?

Reminder: Linearizable means: each method takes effect instantaneously, sometime in its observed time window



Reasoning About Lineraizability: Locking

```
public T deq() throws EmptyException {
  lock.lock();
  try {
    if (tail == head)
       throw new EmptyException();
    T x = items[head % items.length];
    head++;
    return x;
  } finally {
    lock.unlock();
  }
}
```

Reasoning About Lineraizability: Locking

```
public T deq() throws EmptyException {
  lock.lock();
  try {
    if (tail == head)
      throw new EmptyException();
    T x = items[head % items.length];
    head++;
    return x;
  } finally {
    lock.unlock();
  }
}
```

Linearization points are when locks are released

More Reasoning: Lock-free

```
public class LockFreeQueue {

  int head = 0, tail = 0;
  items = (T[]) new Object[capacity];

  public void enq(Item x) {
    while (tail-head == capacity); // busy-wait
    items[tail % capacity] = x; tail++;
  }
  public Item deq() {
     while (tail == head);     // busy-wait
     Item item = items[head % capacity]; head++;
     return item;
}}
```

More Reasoning

```
public class LockFreeQue
  int he     tail = 0
  iter       new Obje
     id enq(Item x) {
       e (tail-head == capacity); // busy-wait
       ems[tail % capacity] = x; tail++;
  }
  public Item deq() {
     while (tail == head);      // busy-wait
     Item item = items[head % capacity]; head++;
     return item;
}}
```

Remember that there is only one enqueuer and only one dequeuer

Linearization order is order head and tail fields modified

Socrative Activity

Go to socrative.com and select "Student Login" Room: CS475; ID is your G-Number

```
class Bouncer {
   public static final int DOWN = 0;
   public static final int RIGHT = 1;
   public static final int STOP = 2;
   private boolean goRight = false;
   private ThreadLocal<Integer> myIndex;
   private int last = -1;

   int visit() {
    1 int i = myIndex.get();
    2 last = i;
    3 if (goRight)
    4    return RIGHT;
    5 goRight = true;
    6 if (last == i)
    7    return STOP;
    8 else
    9    return DOWN;
   }
}
```

If *n* threads call visit() at the same time, how many are assigned:

1. STOP **At most 1**
2. DOWN **At most n-1**
3. RIGHT **At most n-1**

Reminder: If you are not in class, you may not complete the activity. If you do anyway, this will constitute a violation of the honor code.

What's next?

- Weds: One more consistency model: *sequential* (is included in reading for this lecture though, book covers it in a different order), plus more Java-specific implementation fun!
- Reminder for next week: HW1 due Sept 18!!!

This work is licensed under a Creative Commons Attribution-ShareAlike license

- This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/4.0/
- You are free to:
  - Share — copy and redistribute the material in any medium or format
  - Adapt — remix, transform, and build upon the material
  - for any purpose, even commercially.
- Under the following terms:
  - Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
  - ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.
  - No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

```java
class Bouncer {
   public static final int DOWN = 0;
   public static final int RIGHT = 1;
   public static final int STOP = 2;
   private boolean goRight = false;
   private ThreadLocal<Integer> myIndex;
   private int last = -1;

   int visit() {
    1 int i = myIndex.get();
    2 last = i;
    3 if (goRight)
    4    return RIGHT;
    5 goRight = true;
    6 if (last == i)
    7    return STOP;
    8 else
    9    return DOWN;
   }
}
```

```java
int visit() {
 1 int i = myIndex.get();
 2 last = i;
 3 if (goRight)
 4    return RIGHT;
 5 goRight = true;
 6 if (last == i)
 7    return STOP;
 8 else
 9    return DOWN;
}
```