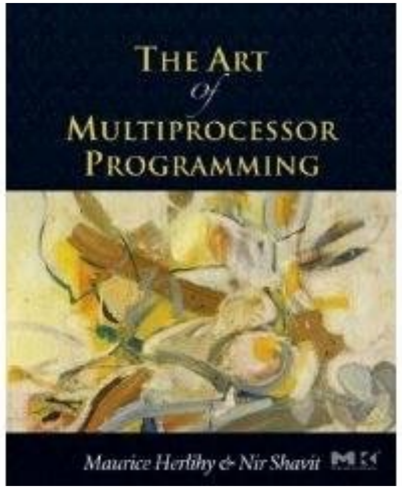
Concurrent Objects

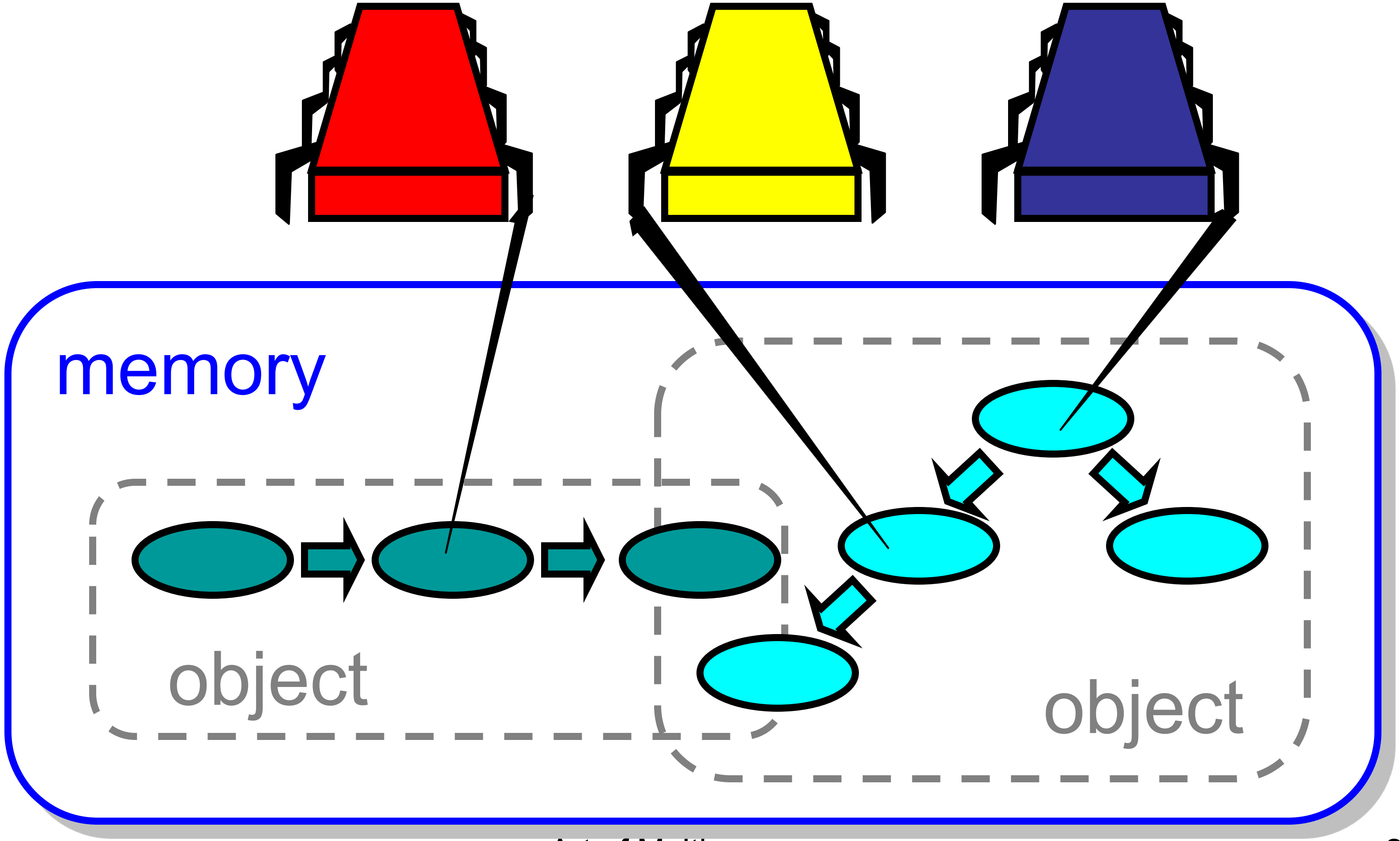


Companion slides for

The Art of Multiprocessor Programming

by Maurice Herlihy & Nir Shavit

Concurrent Computation



Objectivism

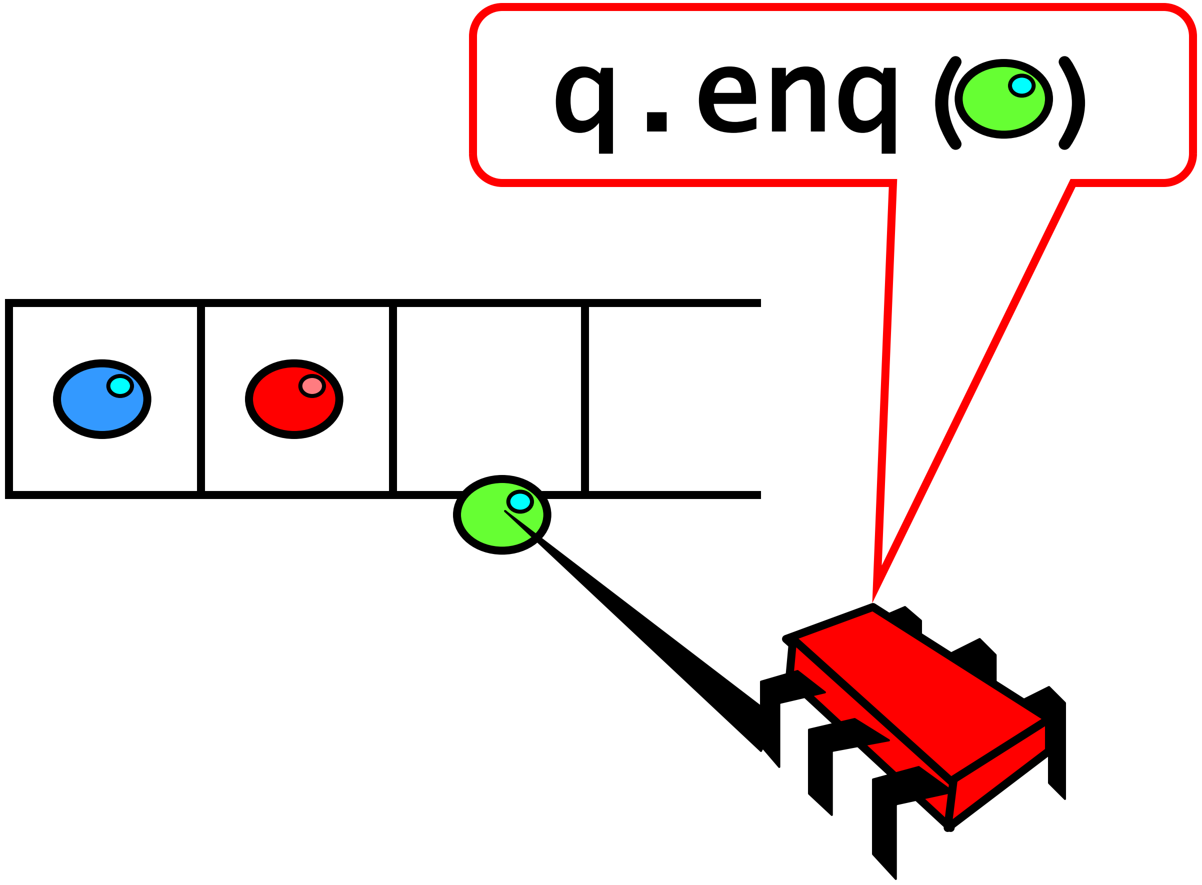
- What is a concurrent object?
  - How do we **describe** one?
  - How do we **implement** one?
  - How do we **tell if we're right**?

Objectivism

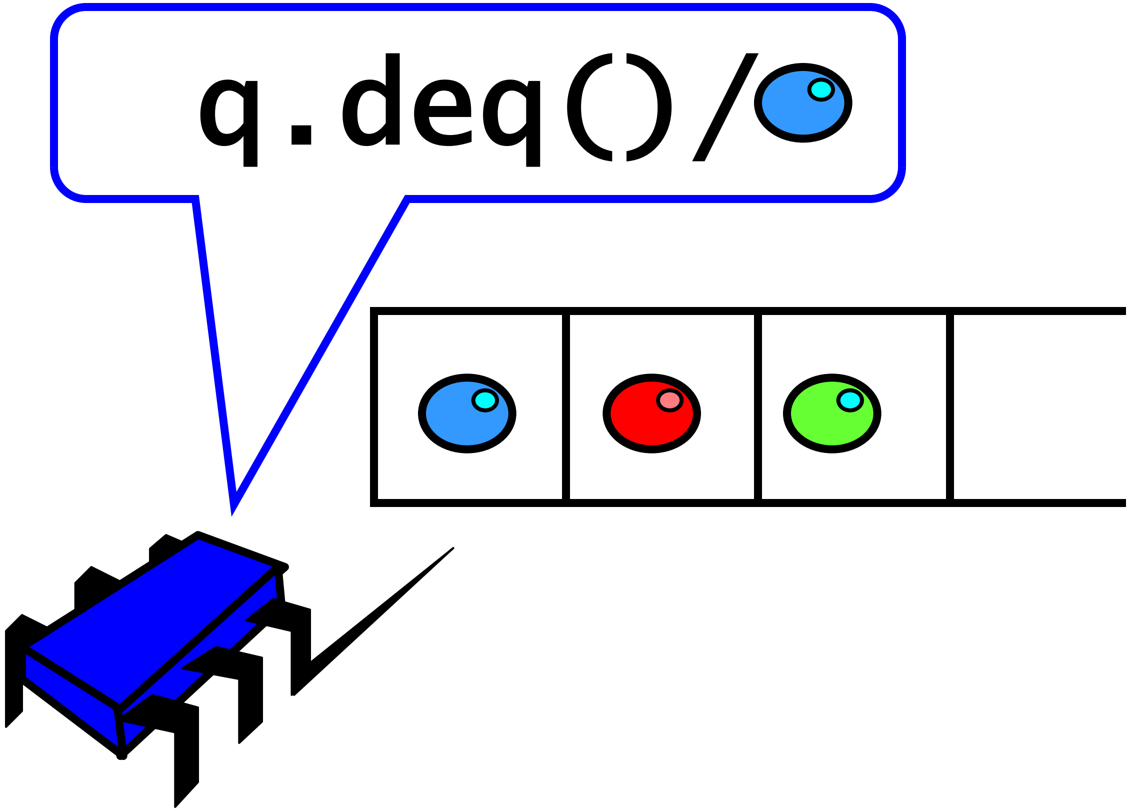
- What is a concurrent object?
  - How do we **describe** one?

  - How do we **tell if we're right**?

FIFO Queue: Enqueue Method



FIFO Queue: Dequeue Method

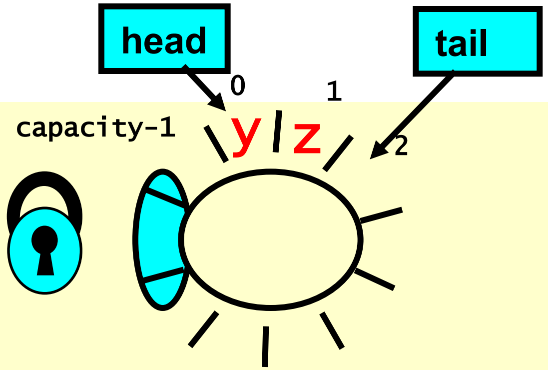


A Lock-Based Queue

```
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];
}
```

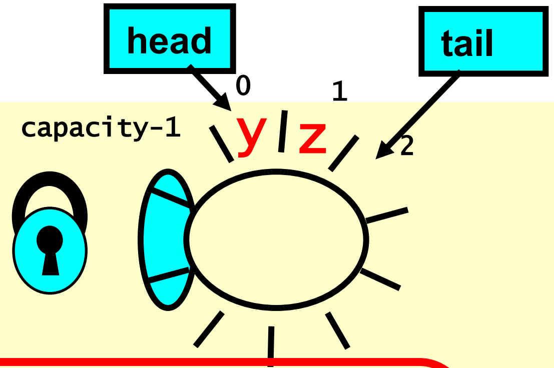
A Lock-Based Queue

```
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

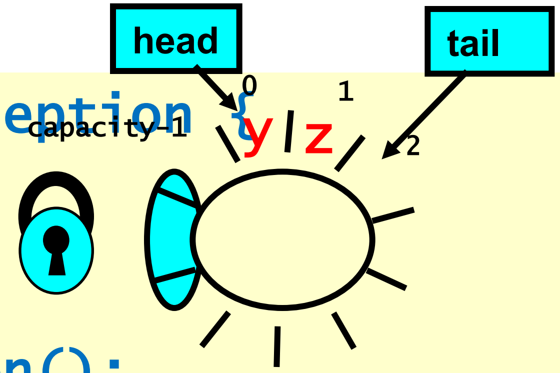
A Lock-Based Queue




```
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];
}
```

Initially head = tail

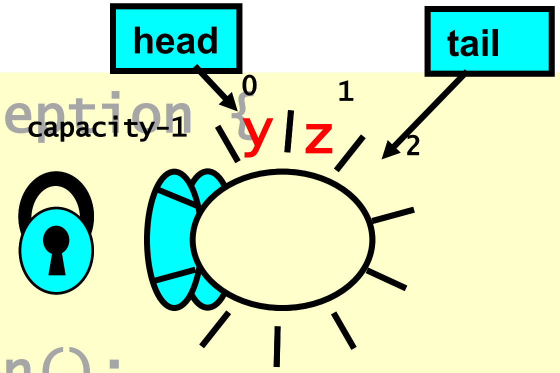
Implementation: Deq




```
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();
  }
}
```

Implementation: Deq

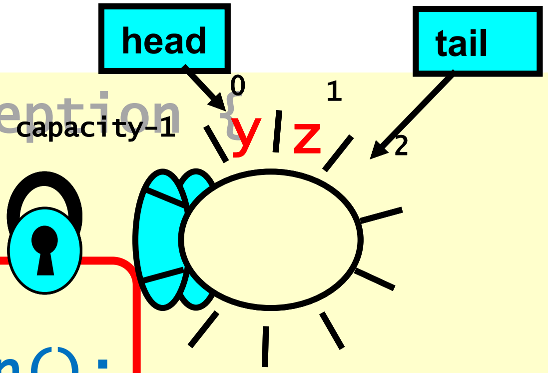
```
public T deq() throws EmptyException {
  lock.lock();
  try {
    if (tail == head)
       throw new EmptyException();
    T x = items[head % items.length];
    head++;
    return x;
  } finally {
    lock.unlock();
  }
}
```



Method calls mutually exclusive

Implementation: Deq

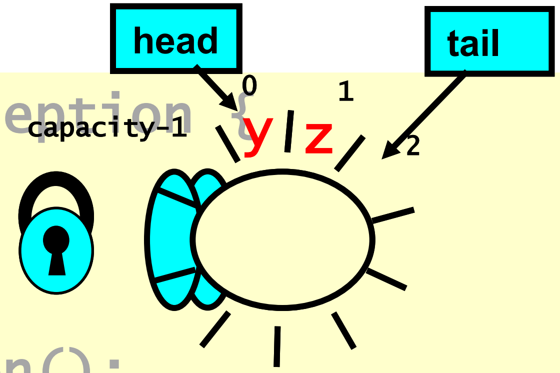
```
public T deq() throws EmptyException {
  lock.lock();
  try {
    if (tail == head)
       throw new EmptyException();
    T x = items[head % items.length];
    head++;
    return x;
  } finally {
    lock.unlock();
  }
}
```



If queue empty throw exception

Implementation: Deq

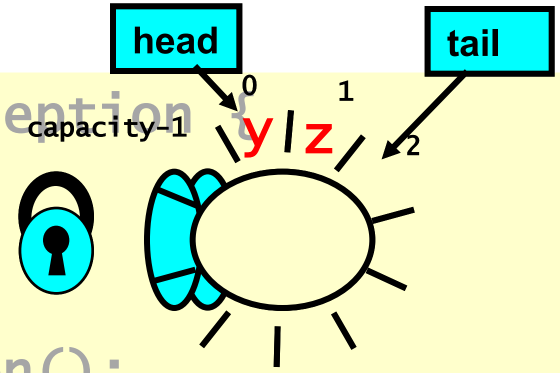
```
public T deq() throws EmptyException {
  lock.lock();
  try {
    if (tail == head)
       throw new EmptyException();
    T x = items[head % items.length];
    head++;
    return x;
  } finally {
    lock.unlock();
  }
}
```



Queue not empty: remove item and update head

Implementation: Deq

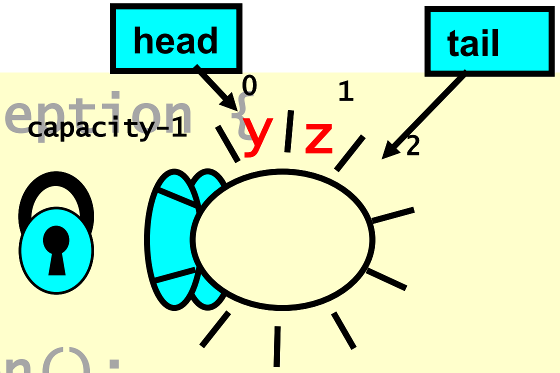
```
public T deq() throws EmptyException {
  lock.lock();
  try {
    if (tail == head)
       throw new EmptyException();
    T x = items[head % items.length];
    head++;
    return x;
  } finally {
    lock.unlock();
  }
}
```



Return result

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();
  }
}
```



Release lock no matter what!

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

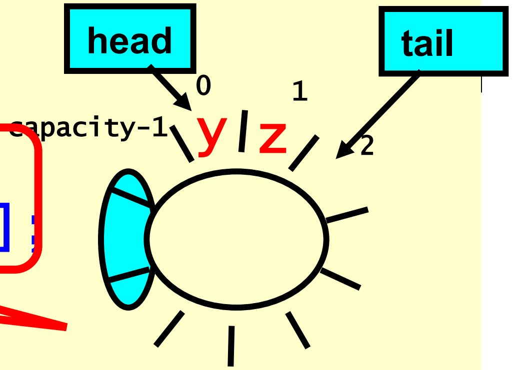
- The same thing without mutual exclusion
- For simplicity, only two threads
  - One thread enq only
  - The other deq only

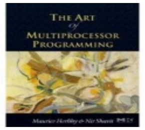
Wait-free 2-Thread Queue

```
public class WaitFreeQueue {

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

  public void enq(Item x) {
    if (tail-head == capacity) throw
         new FullException();
    items[tail % capacity] = x; tail++;
  }
  public Item deq() {
     if (tail == head) throw
         new EmptyException();
     Item item = items[head % capacity]; head++;
     return item;
}}
```

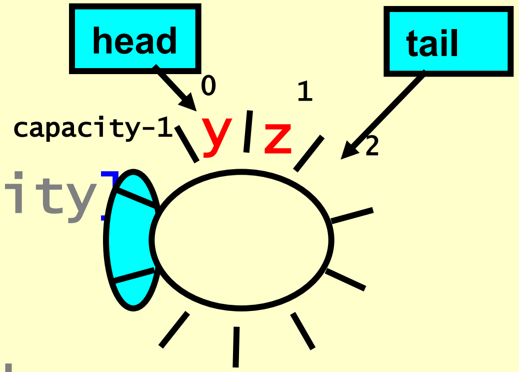
Wait-free 2-Thread Queue

```
public class WaitFreeQueue {

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

  public void enq(Item x) {
    if (tail-head == capacity) throw
         new FullException();
    items[tail % capacity] = x; tail++;
  }
  public Item deq() {
     if (tail == head) throw
         new EmptyException();
     Item item = items[head % capacity]; head++;
     return item;
}}
```





Wait-free 2-Thread Queue




```
public class WaitFreeQueue {

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

  public void enq(Item x) {
    if (tail-head == capacity) throw
        new FullException();
    items[tail % capacity] = x; tail++;
  }
  public Item deq() {
     if (tail == head) thr
        new EmptyEx
    Queue is updated                    ++;
     return item;
}}
```

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

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

Let's 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

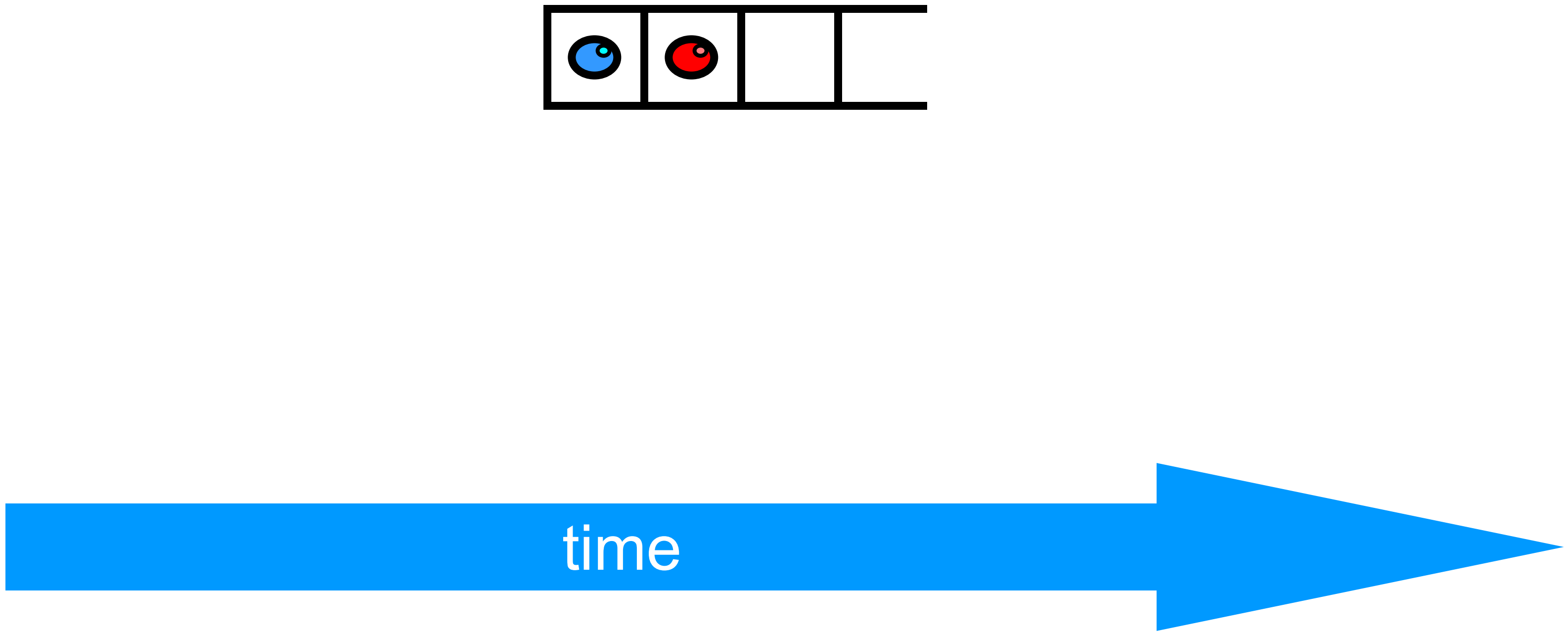
Why Sequential Specifications Totally Rock

- 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

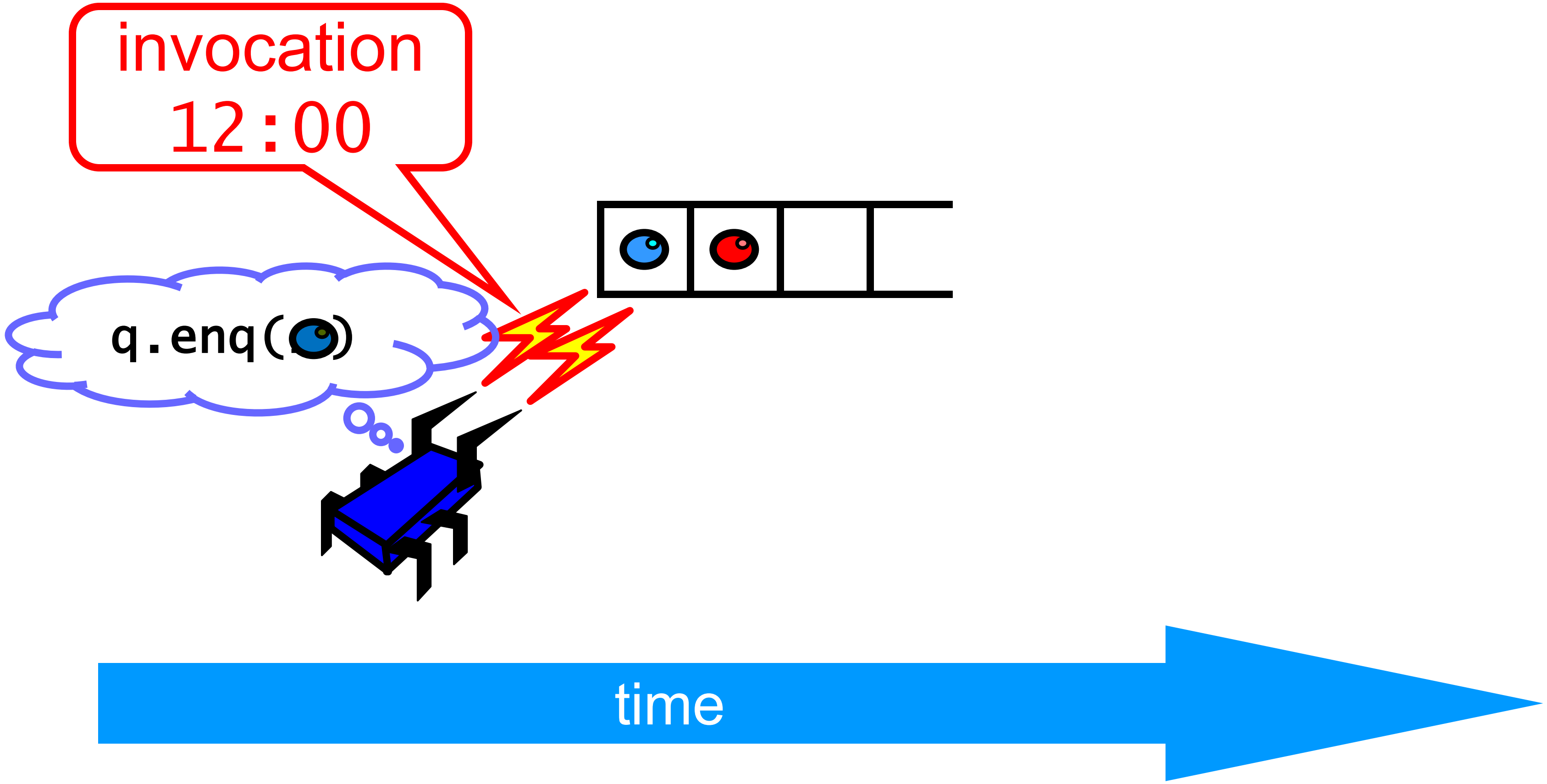
What About Concurrent Specifications ?

- Methods?
- Documentation?
- Adding new methods?

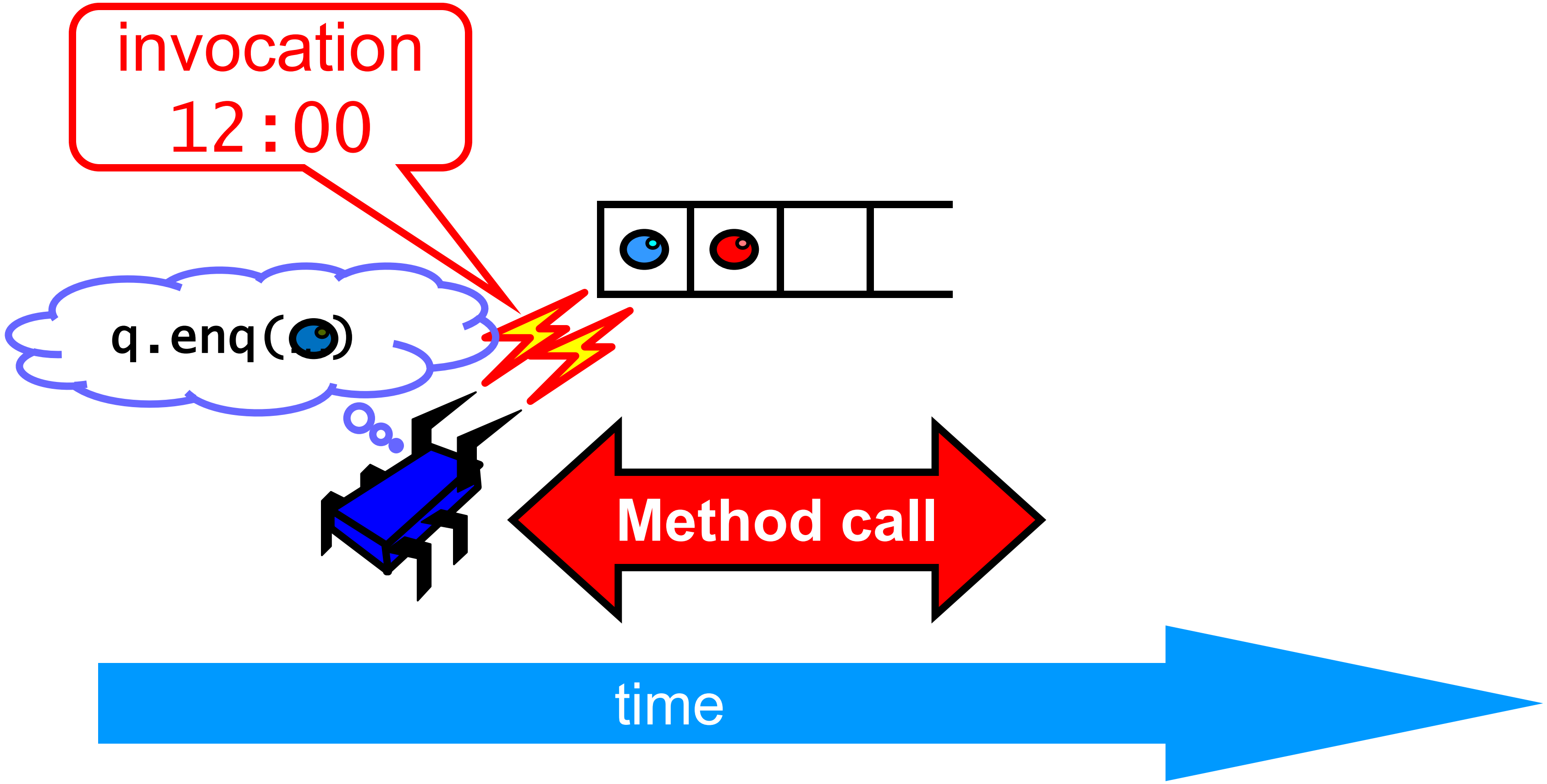
Methods Take Time



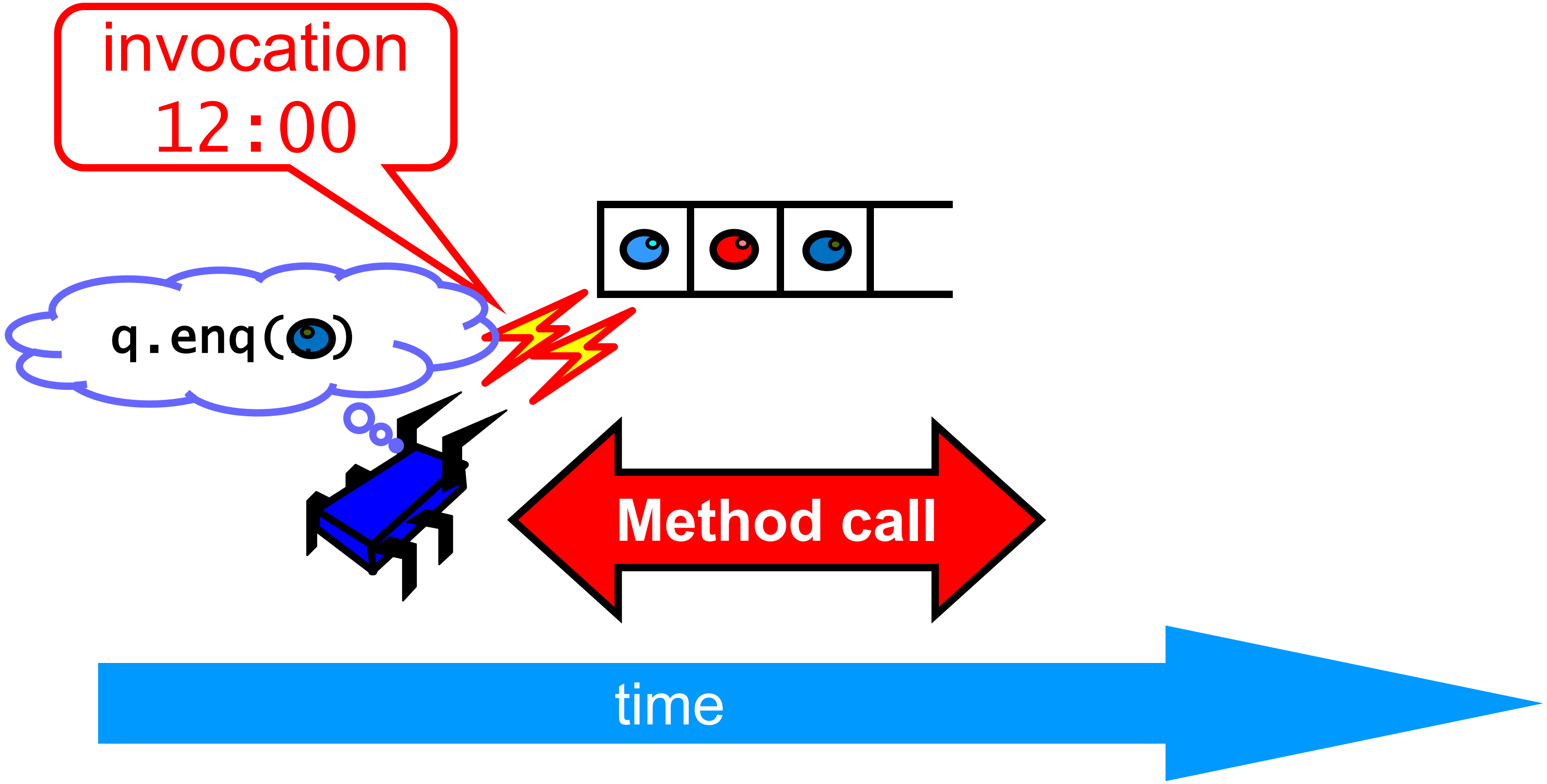
Methods Take Time



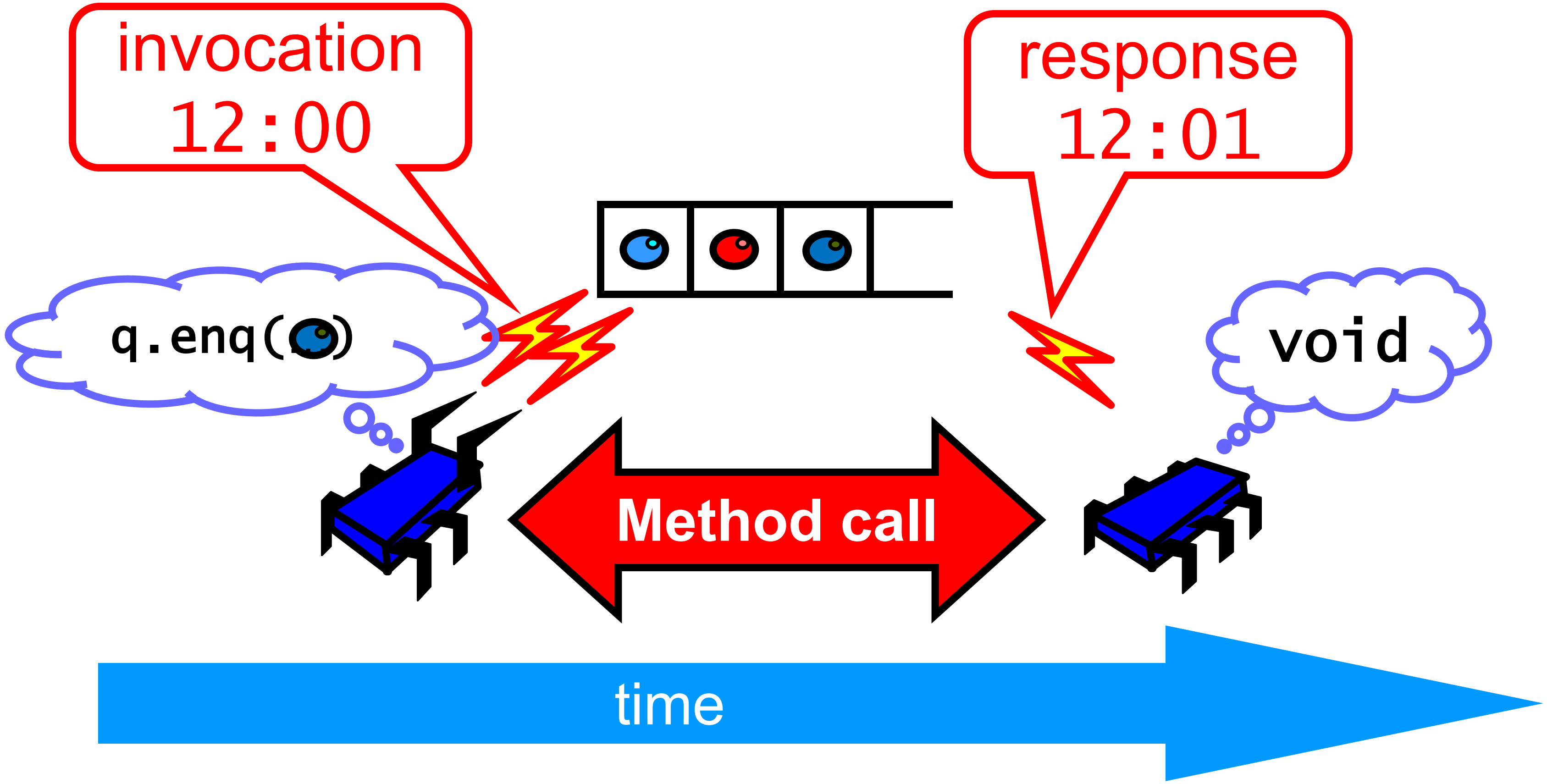
Methods Take Time



Methods Take Time



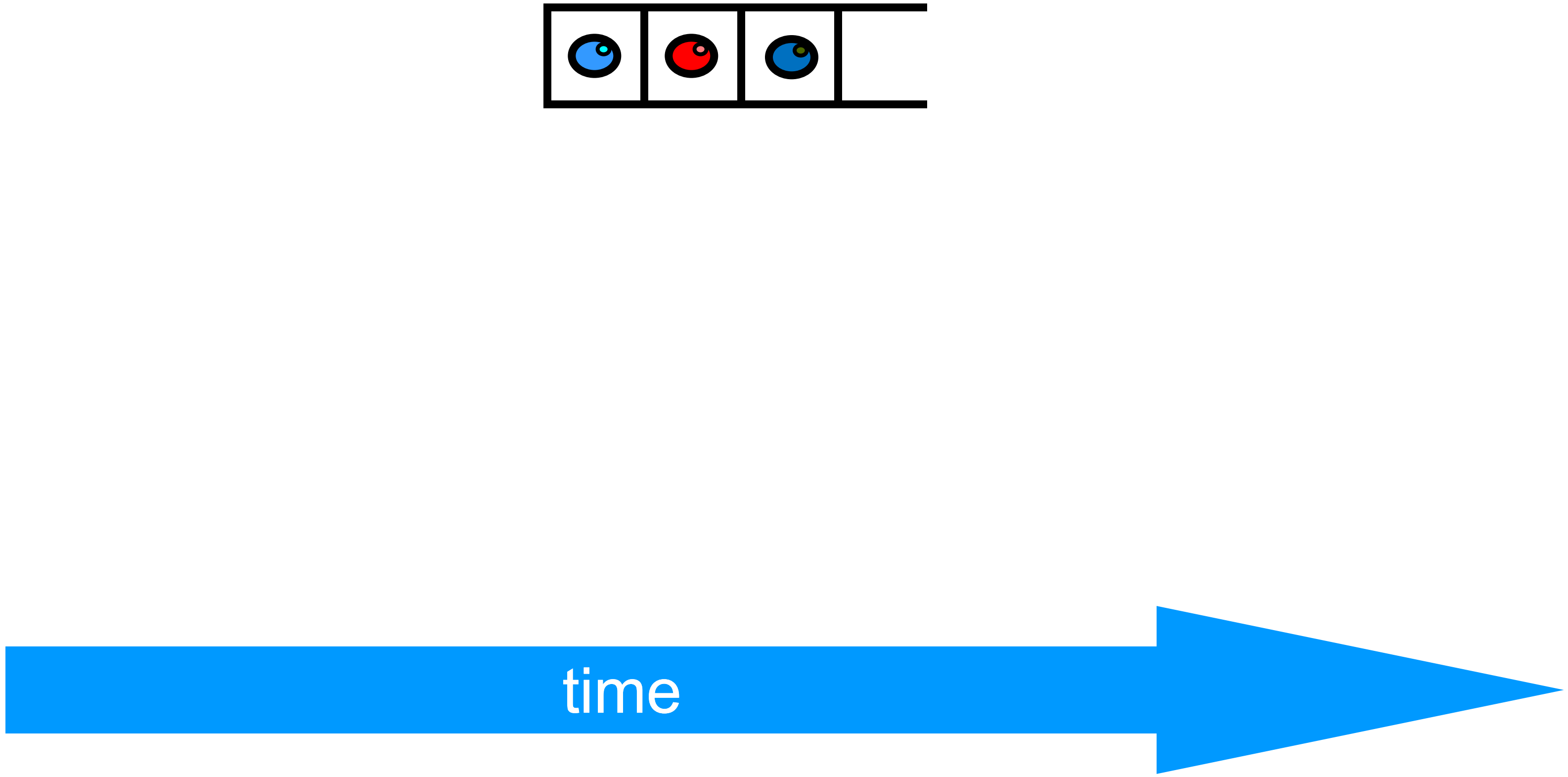
Methods Take Time



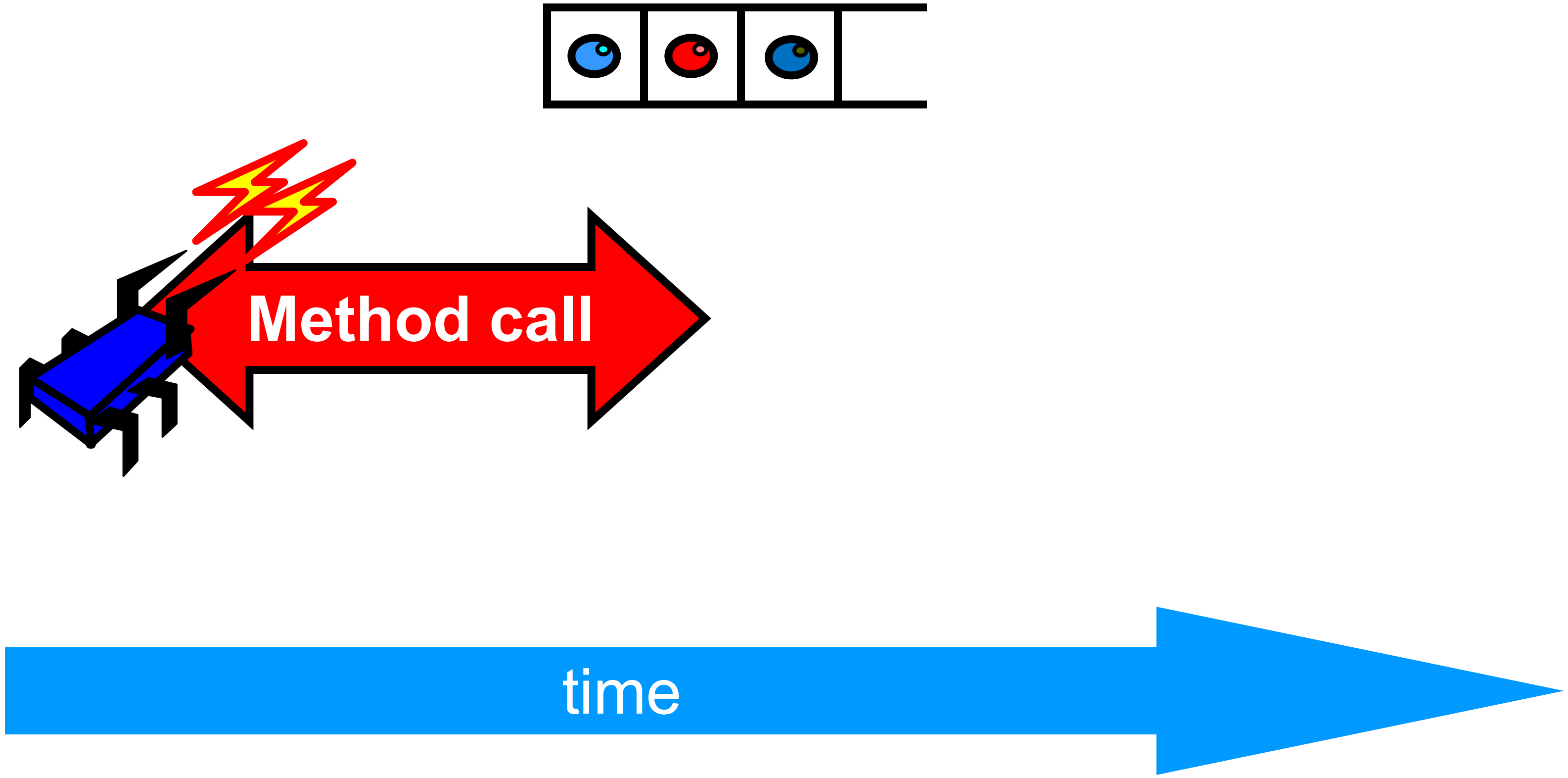
Sequential vs Concurrent

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

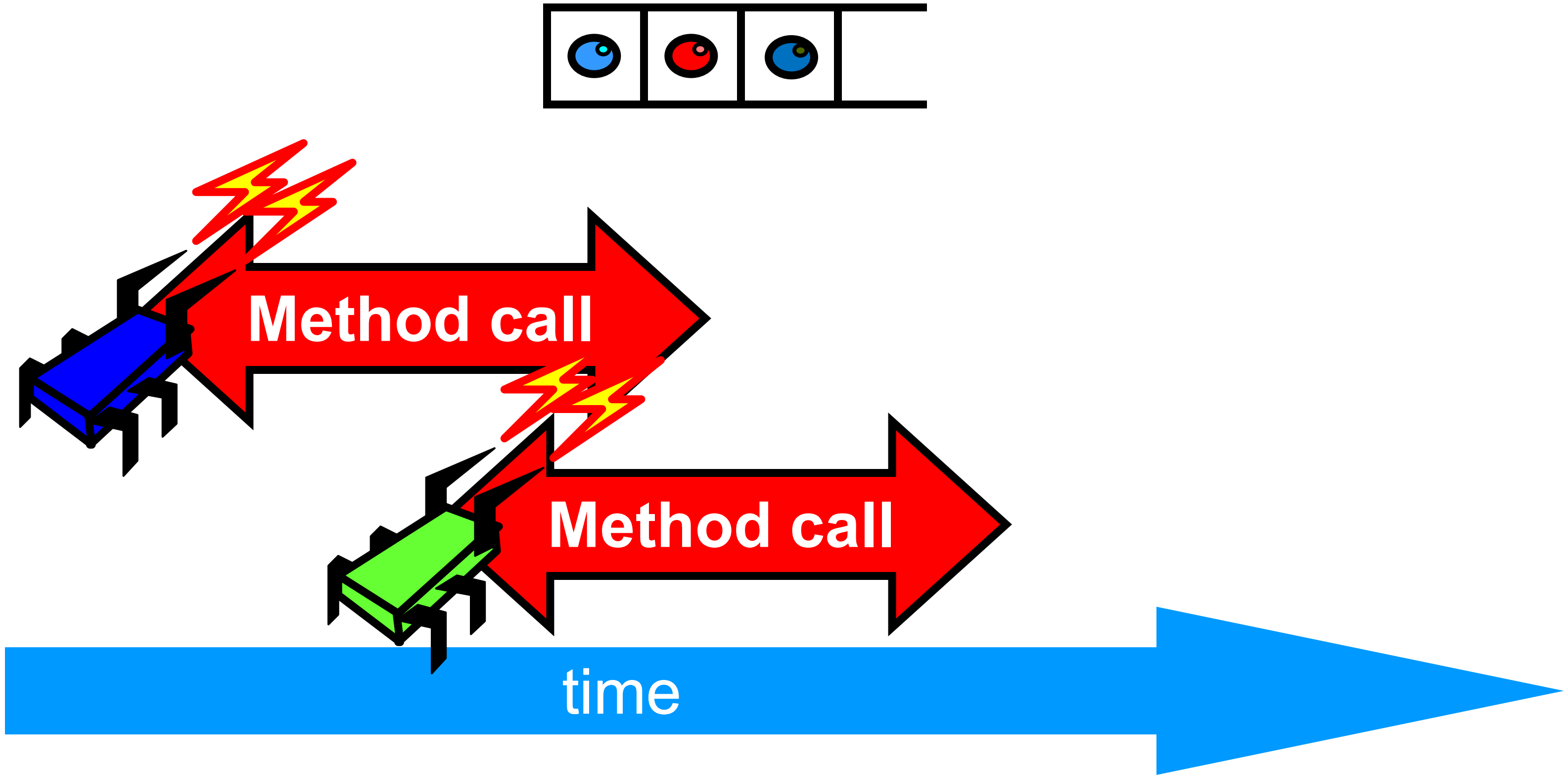
Concurrent Methods Take Overlapping Time



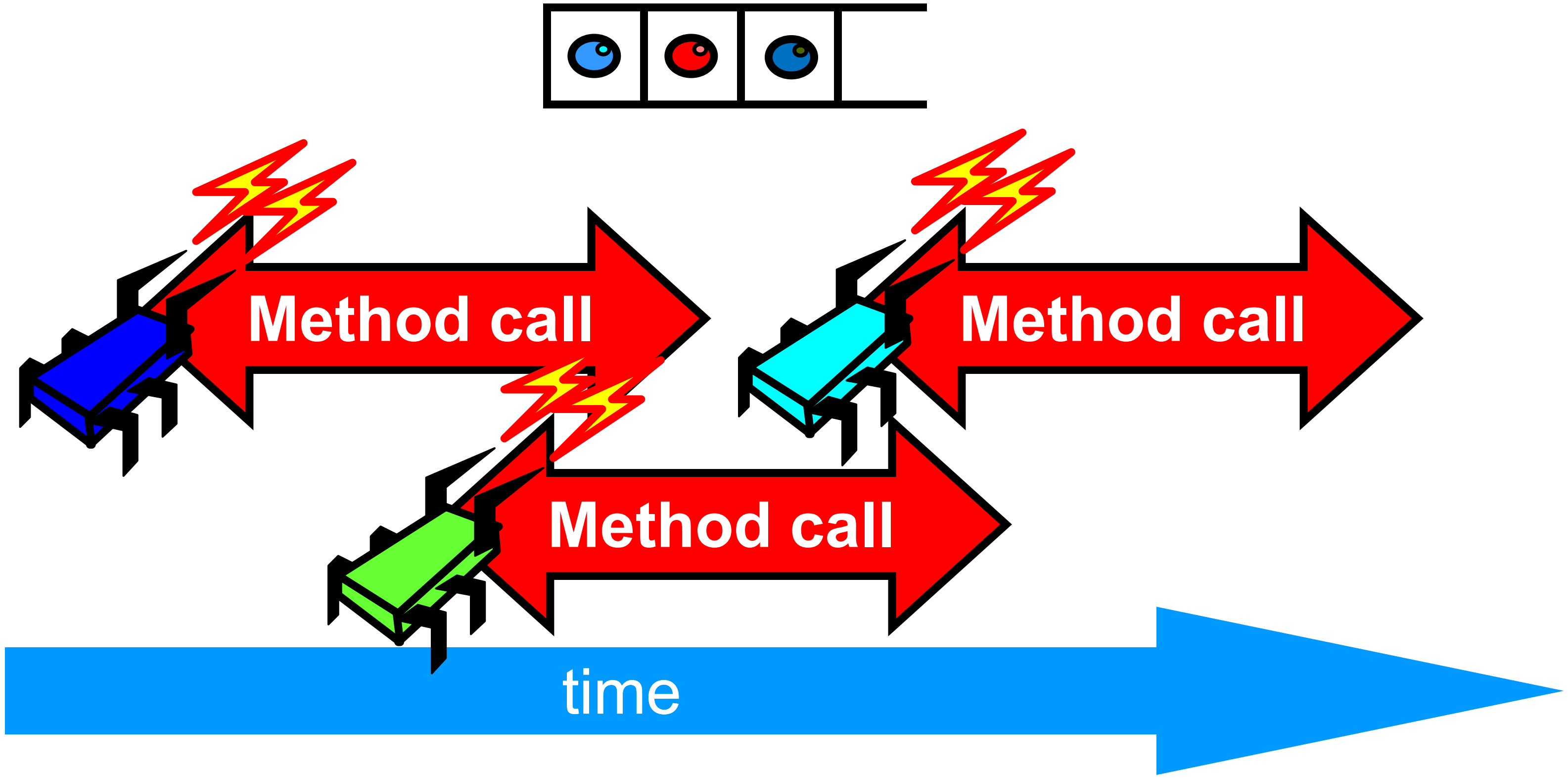
Concurrent Methods Take Overlapping Time



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

Panic!

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();
  }
}
```

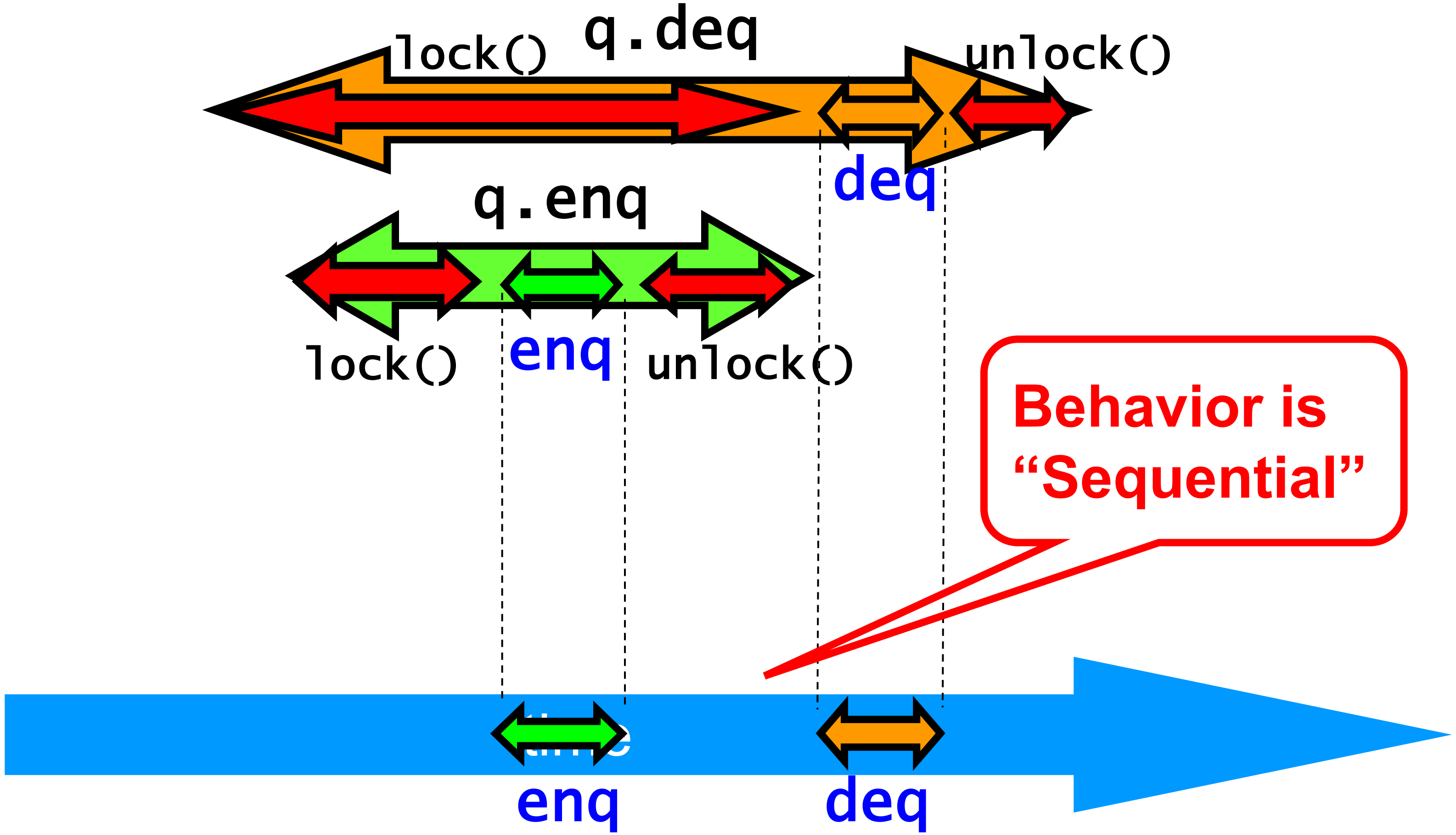
Intuitively…

```
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();
  }
}
```

All modifications of queue are done mutually exclusive

Intuitively

Let's capture the idea of describing the concurrent via the sequential



Behavior is "Sequential"

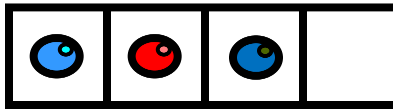
Linearizability

- 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™**

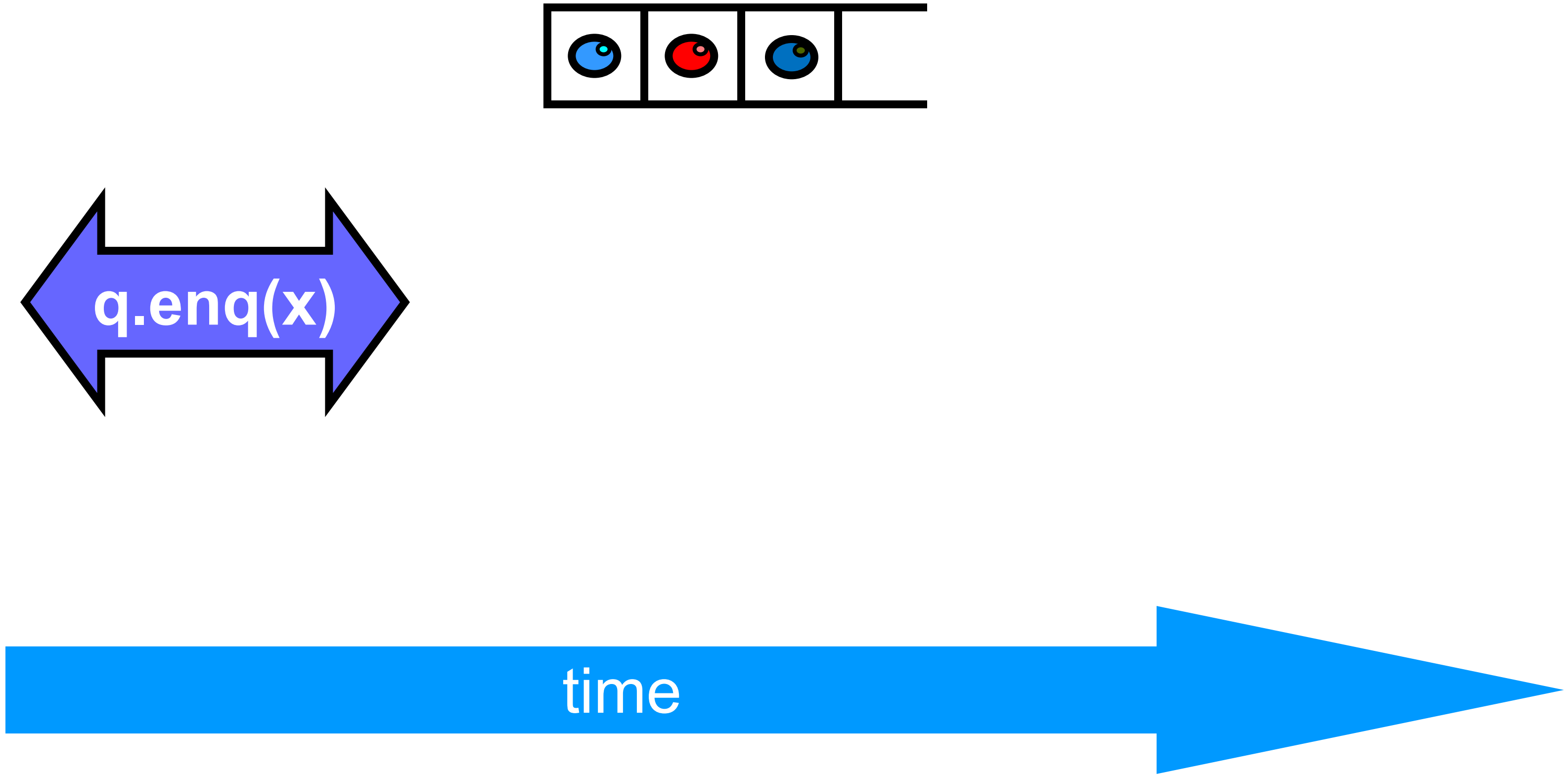
Is it really about the object?

- 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

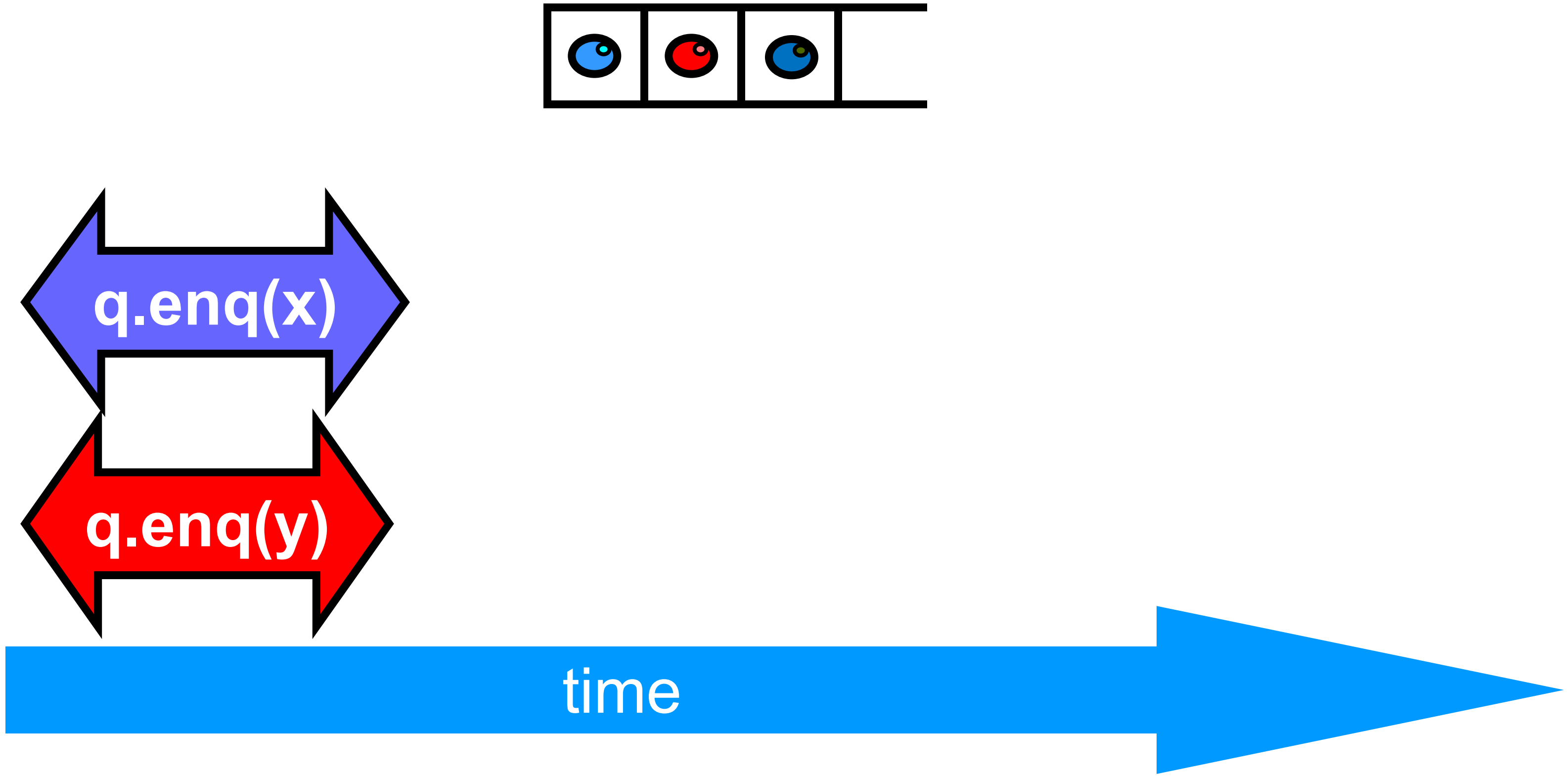
Example



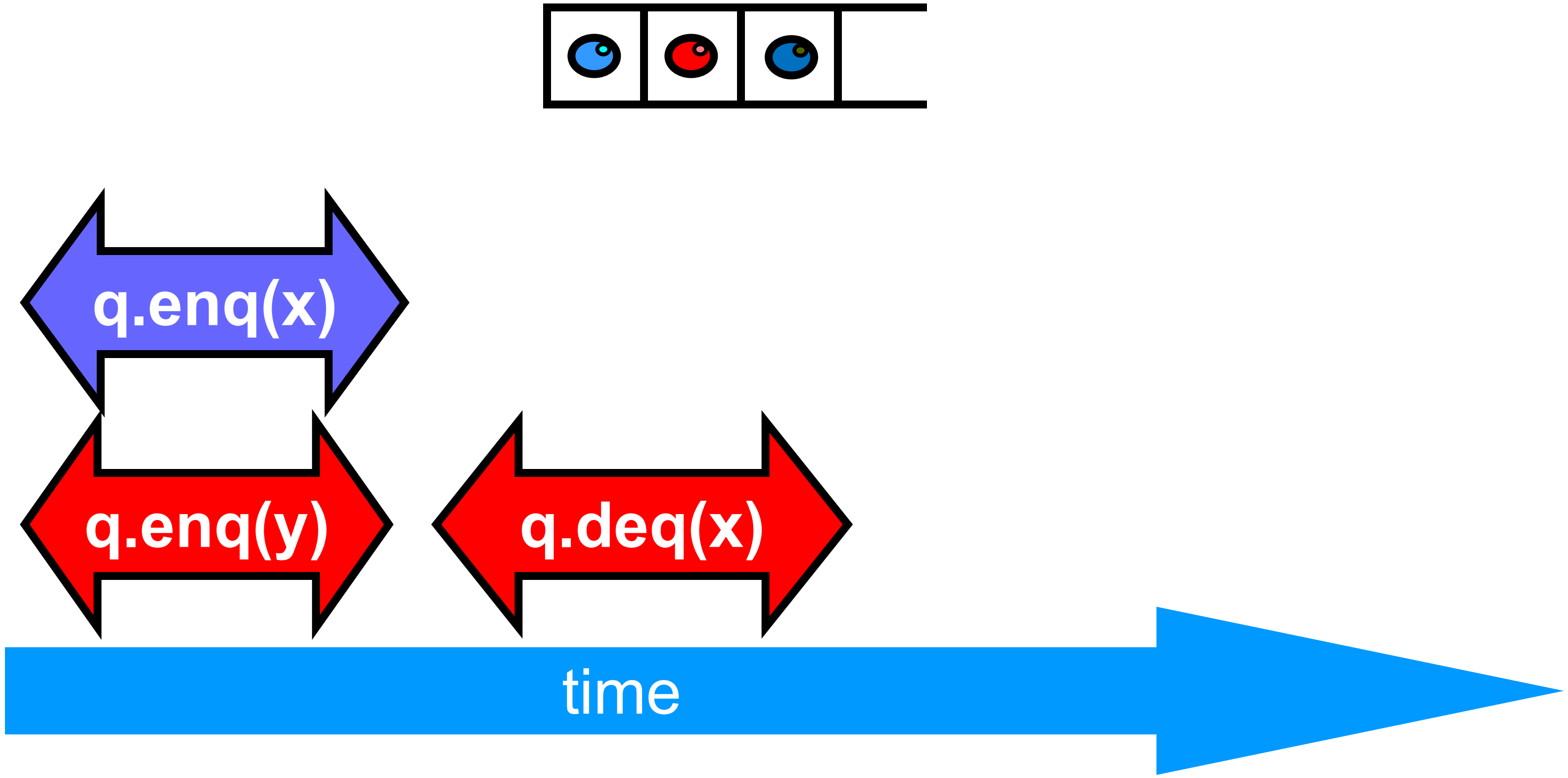
Example



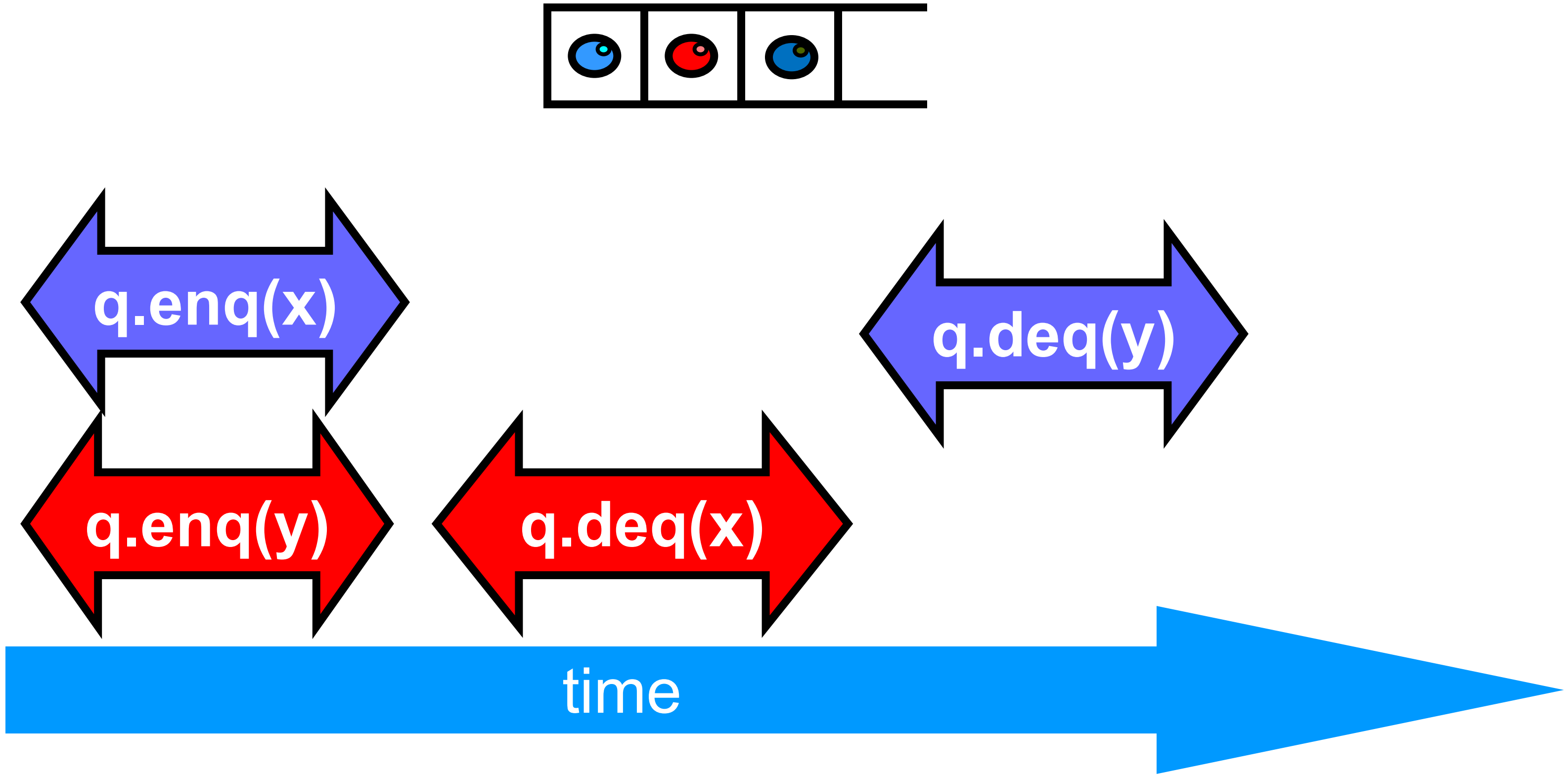
Example



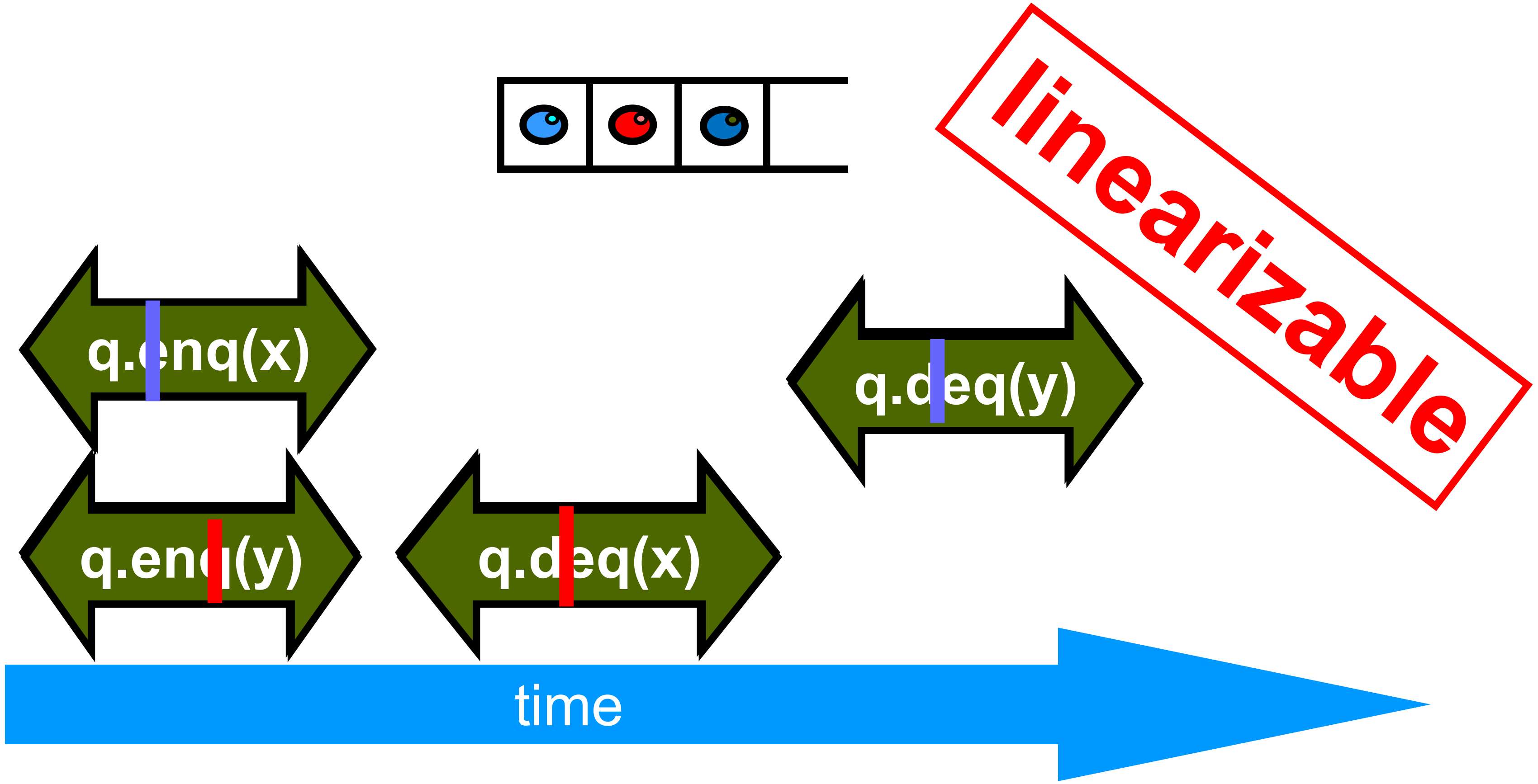
Example



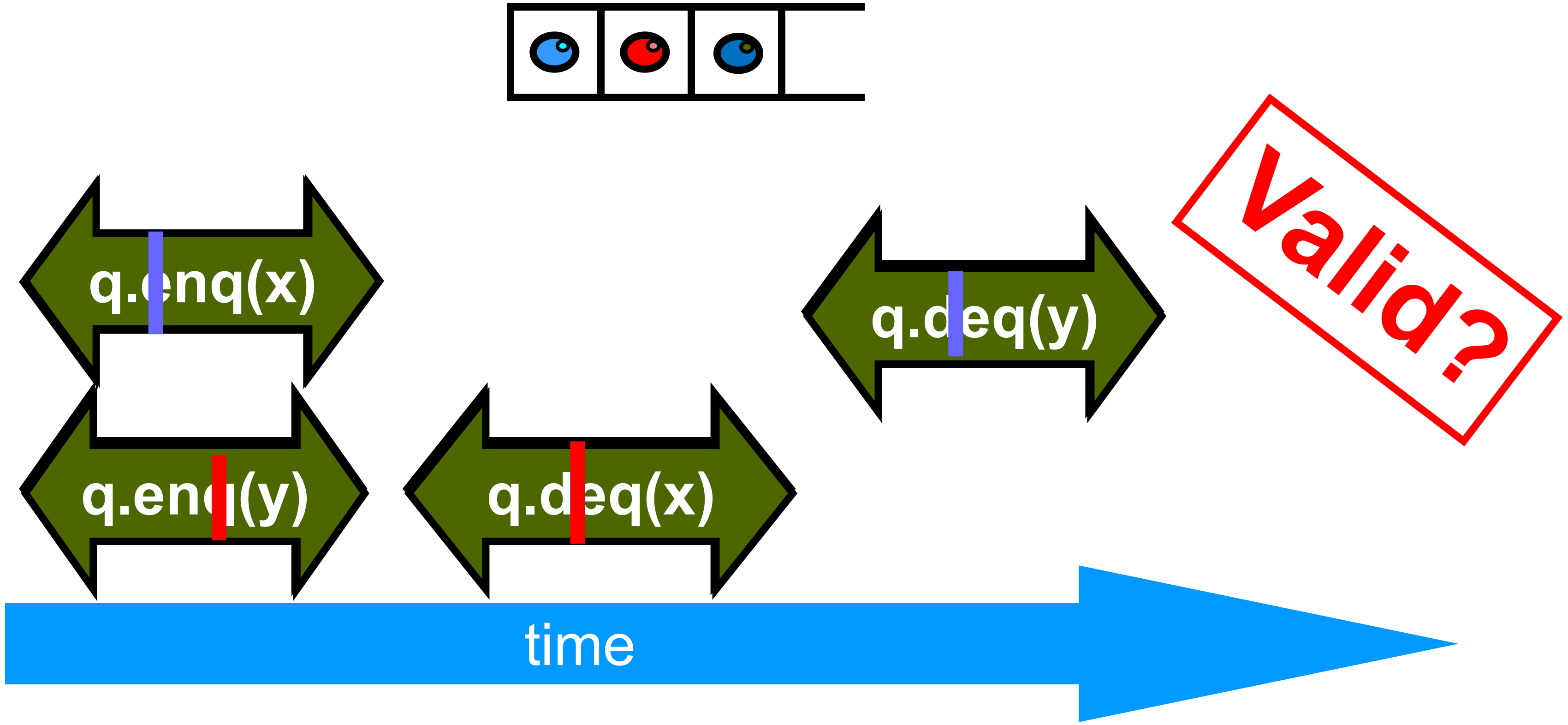
Example



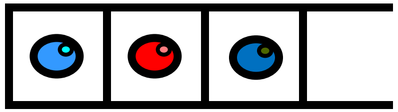
Example



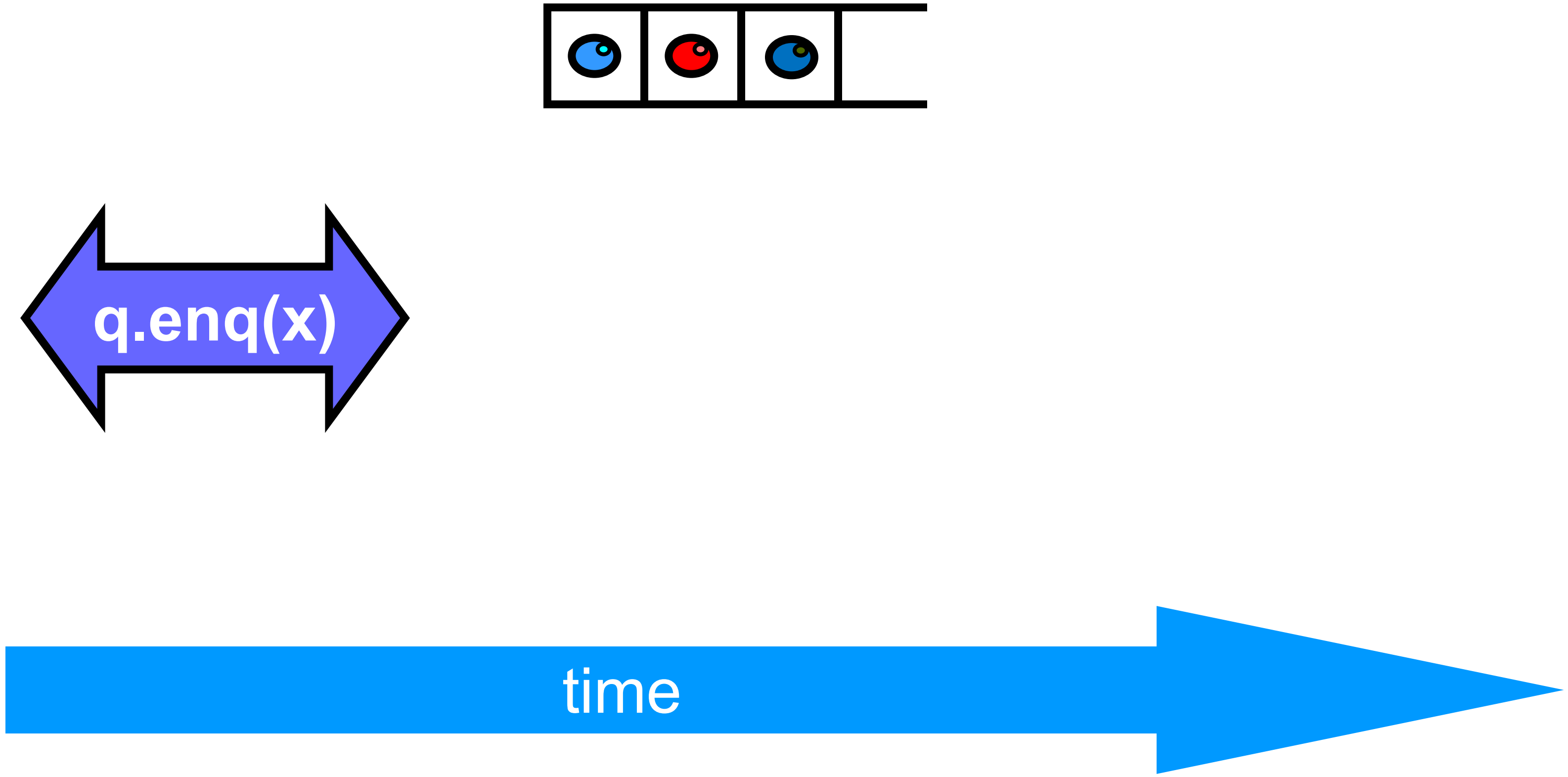
Example



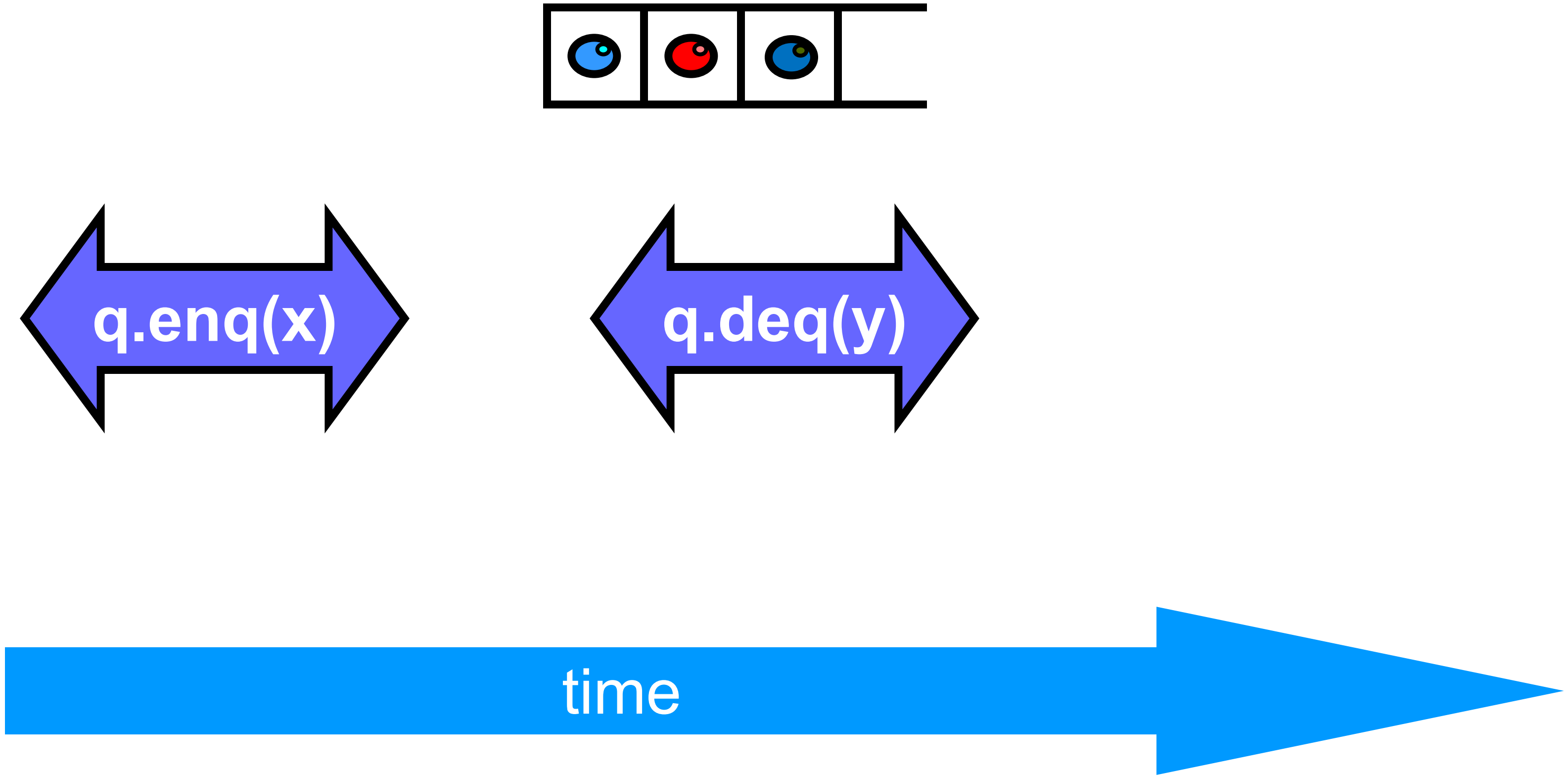
Example



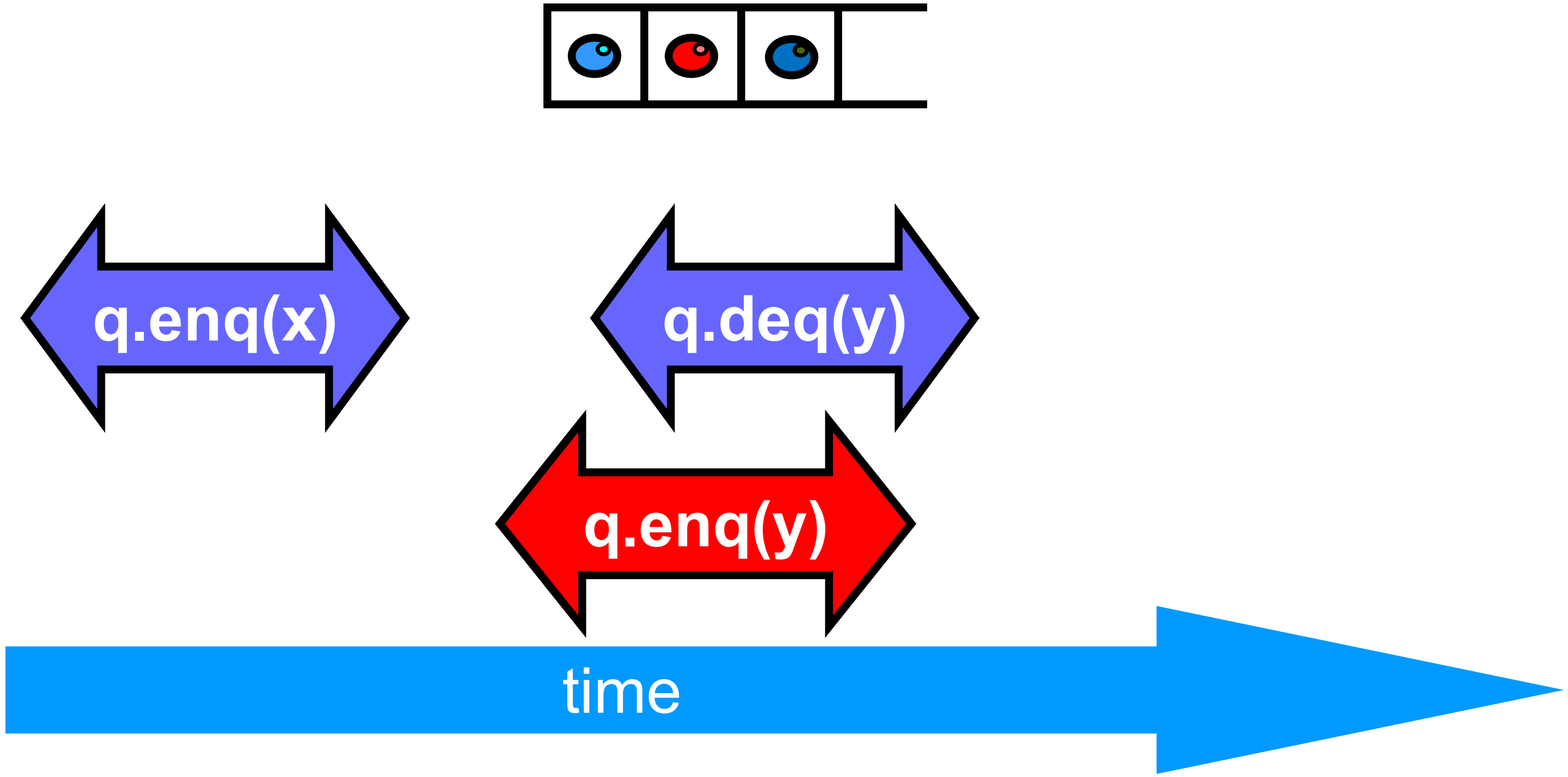
Example



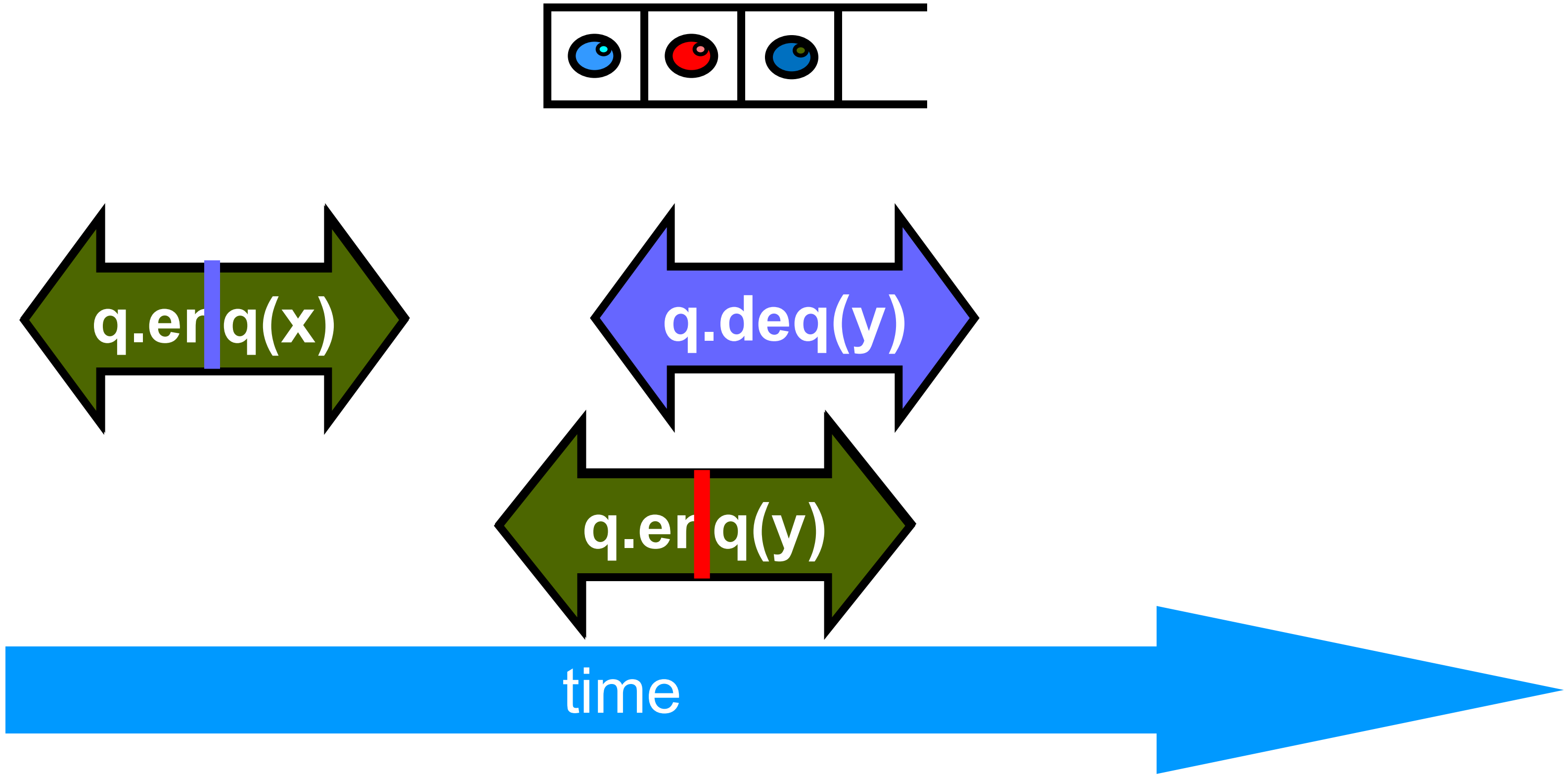
Example



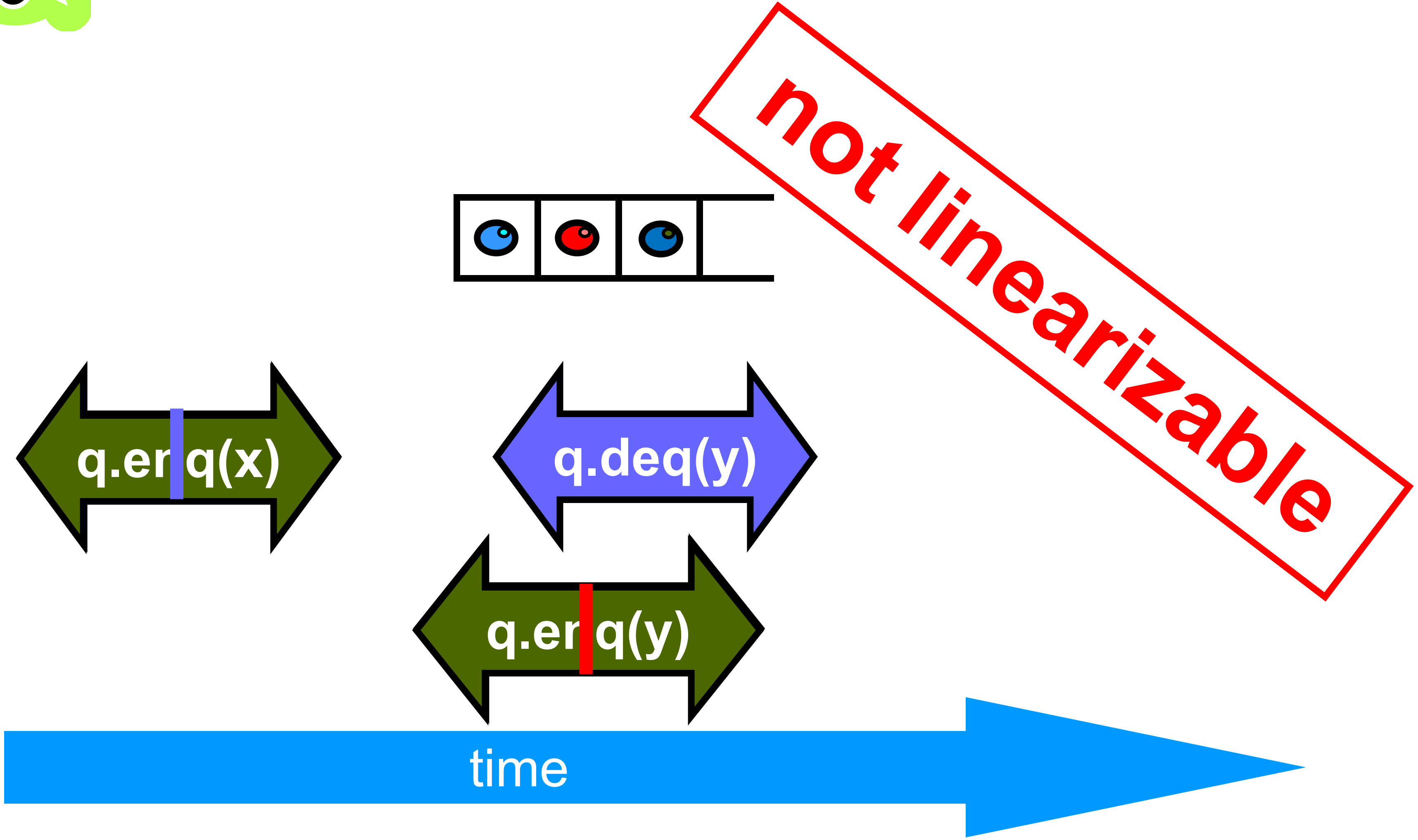
Example



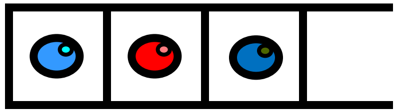
Example



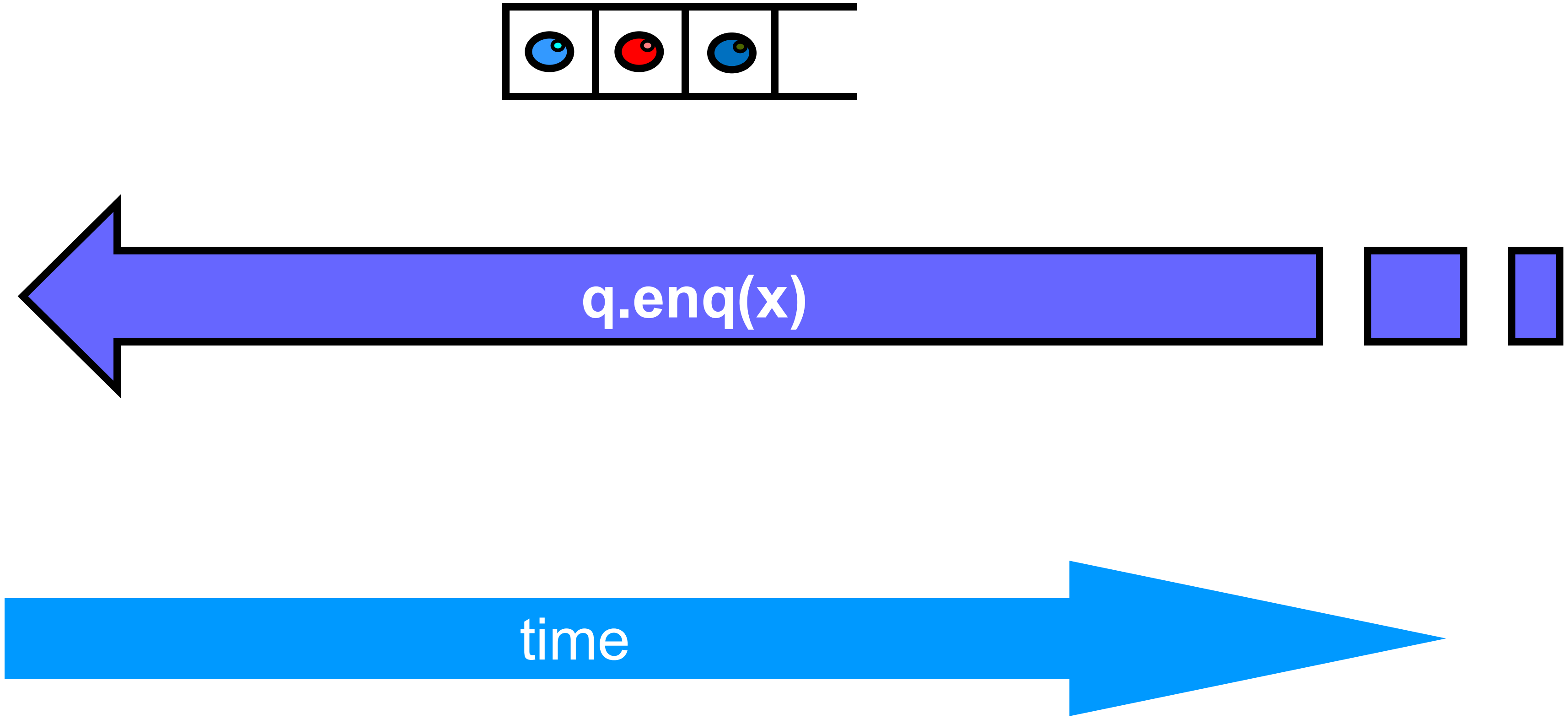
Example



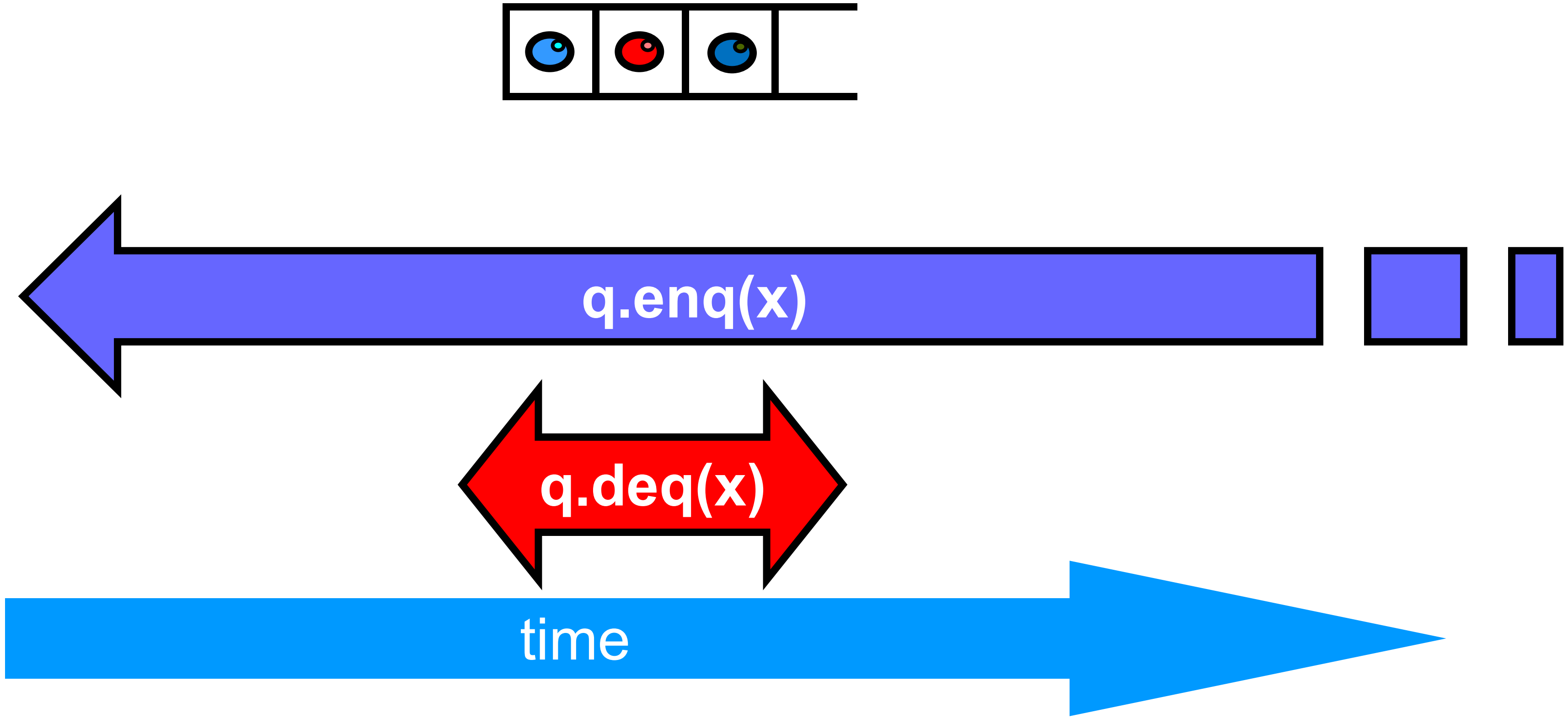
Example



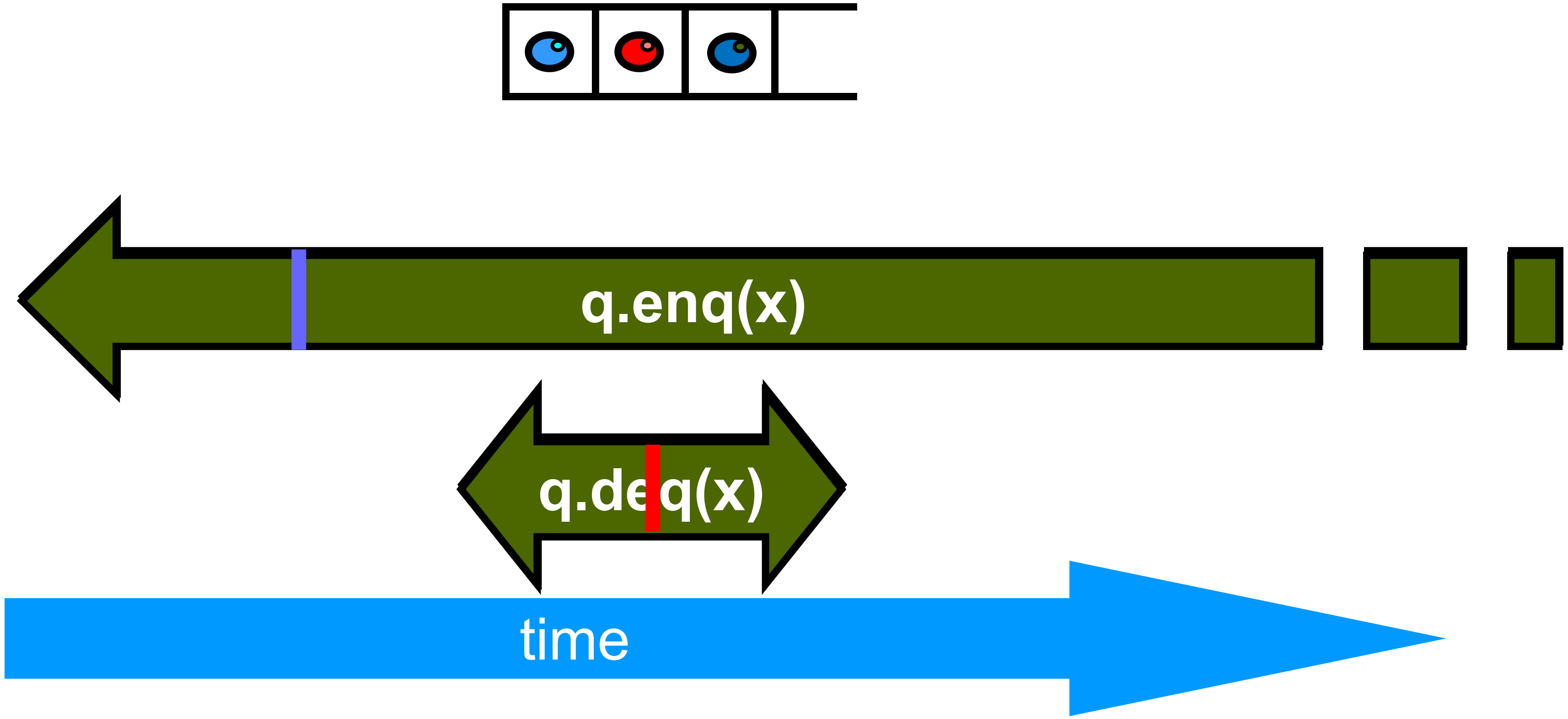
Example



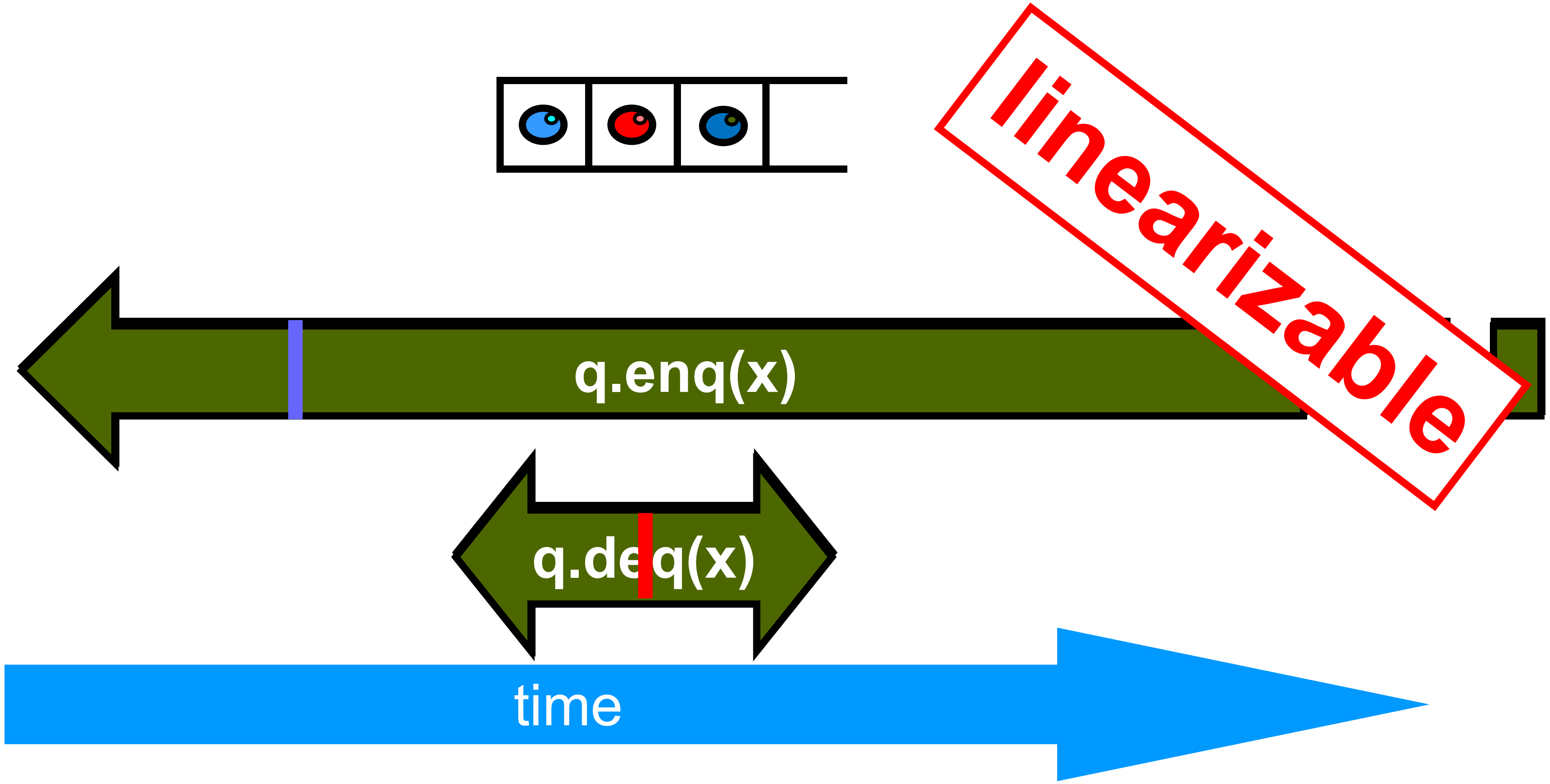
Example



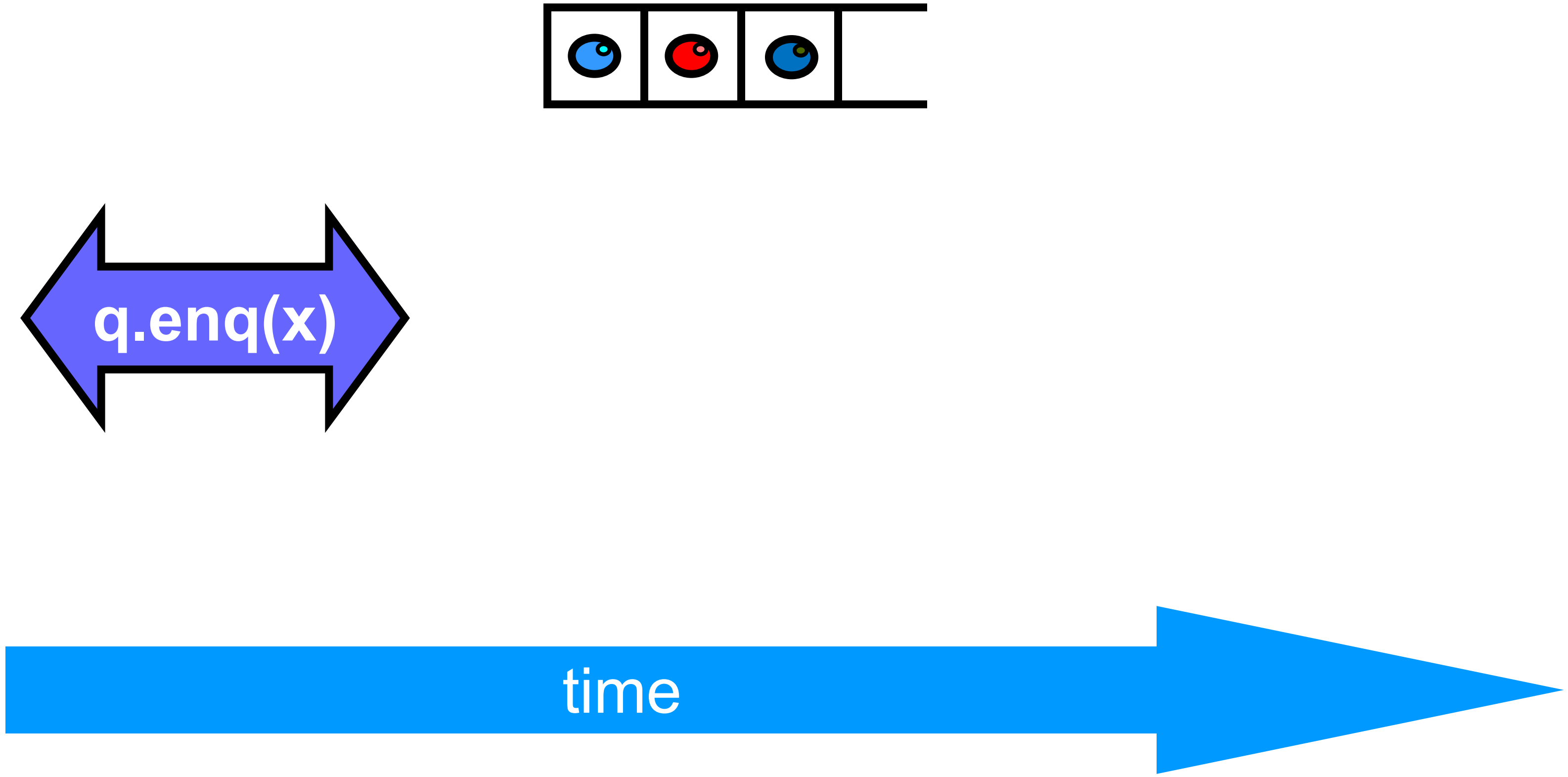
Example



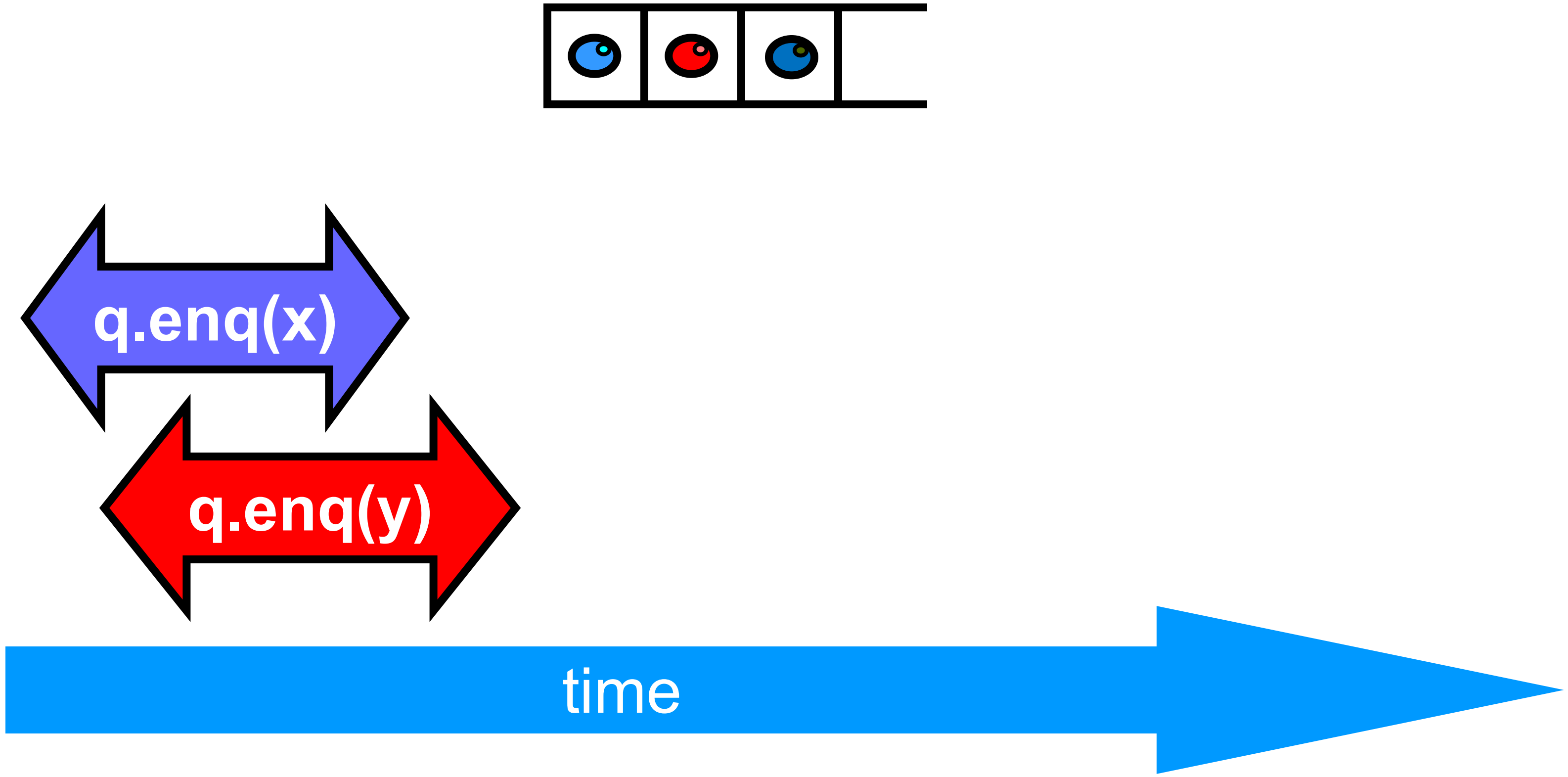
Example



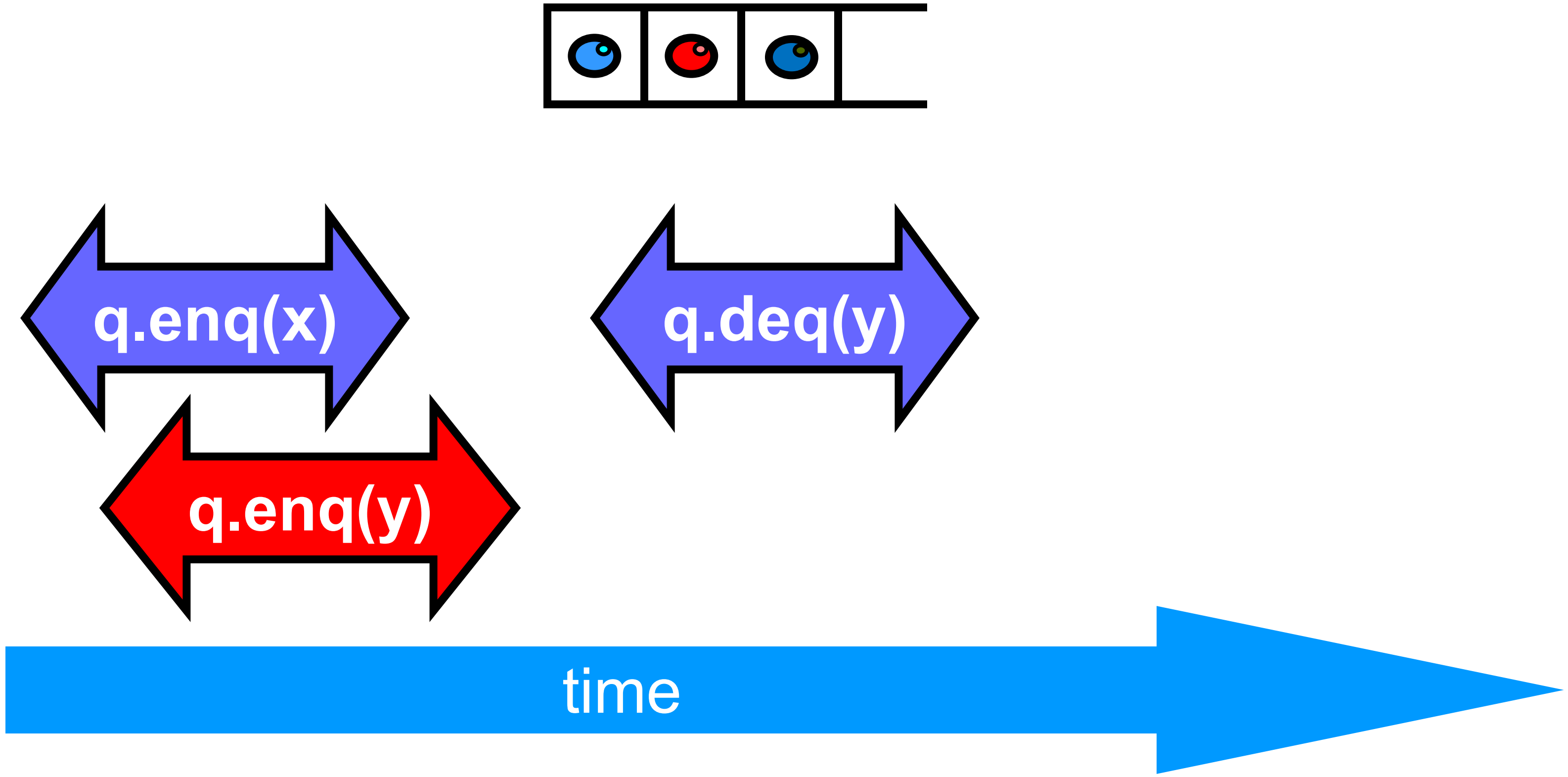
Example



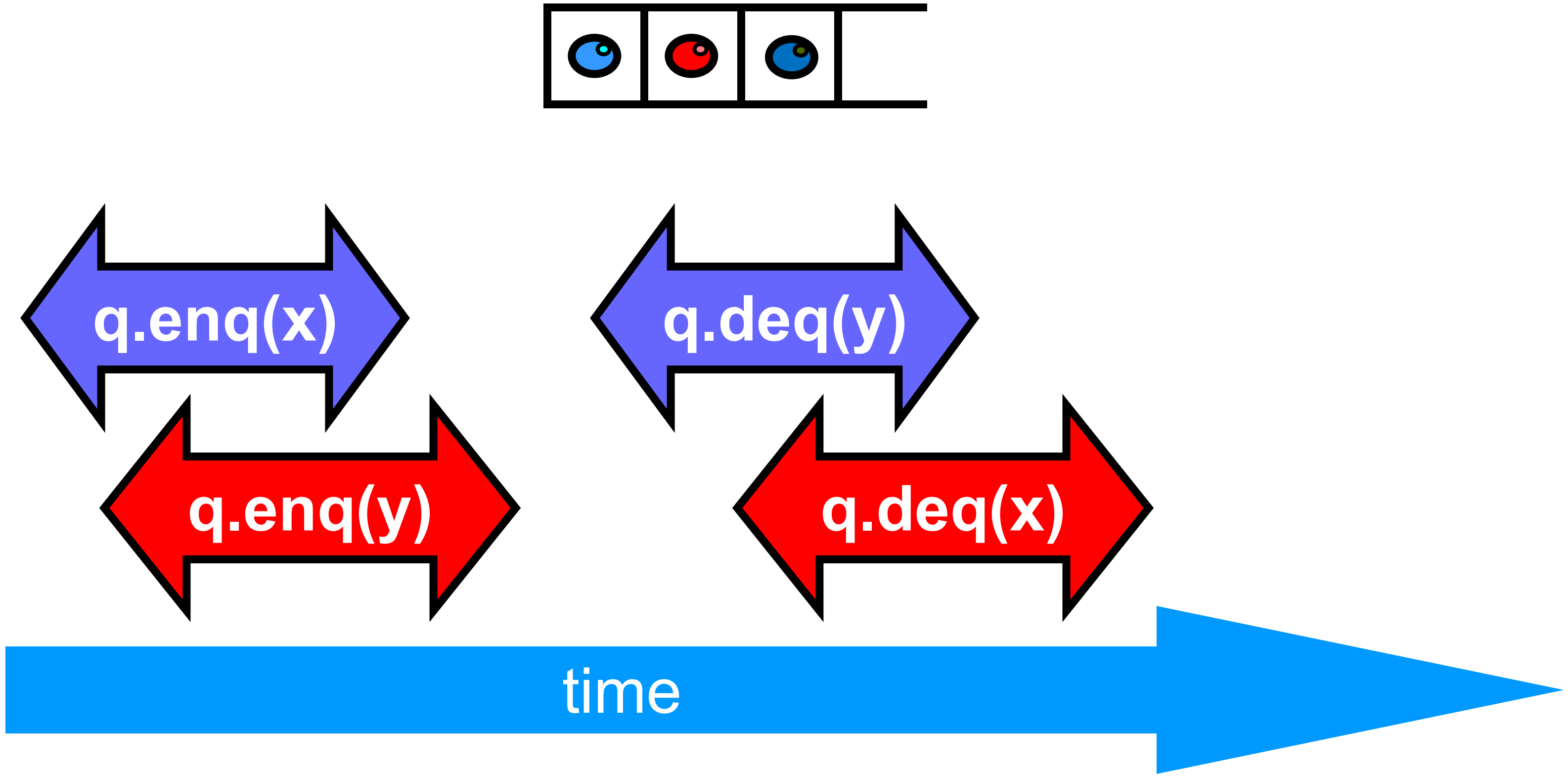
Example



Example

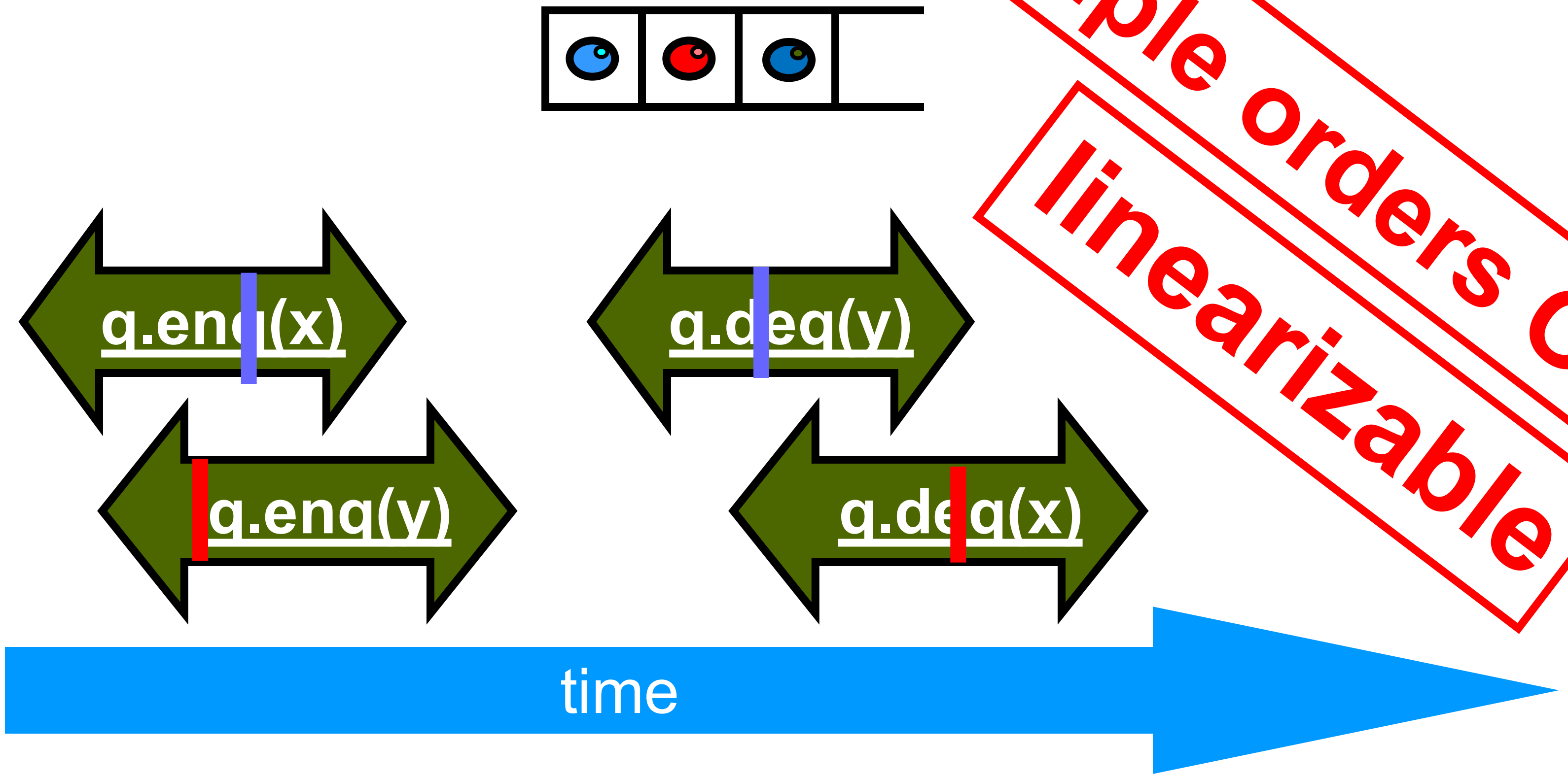


Example



Example

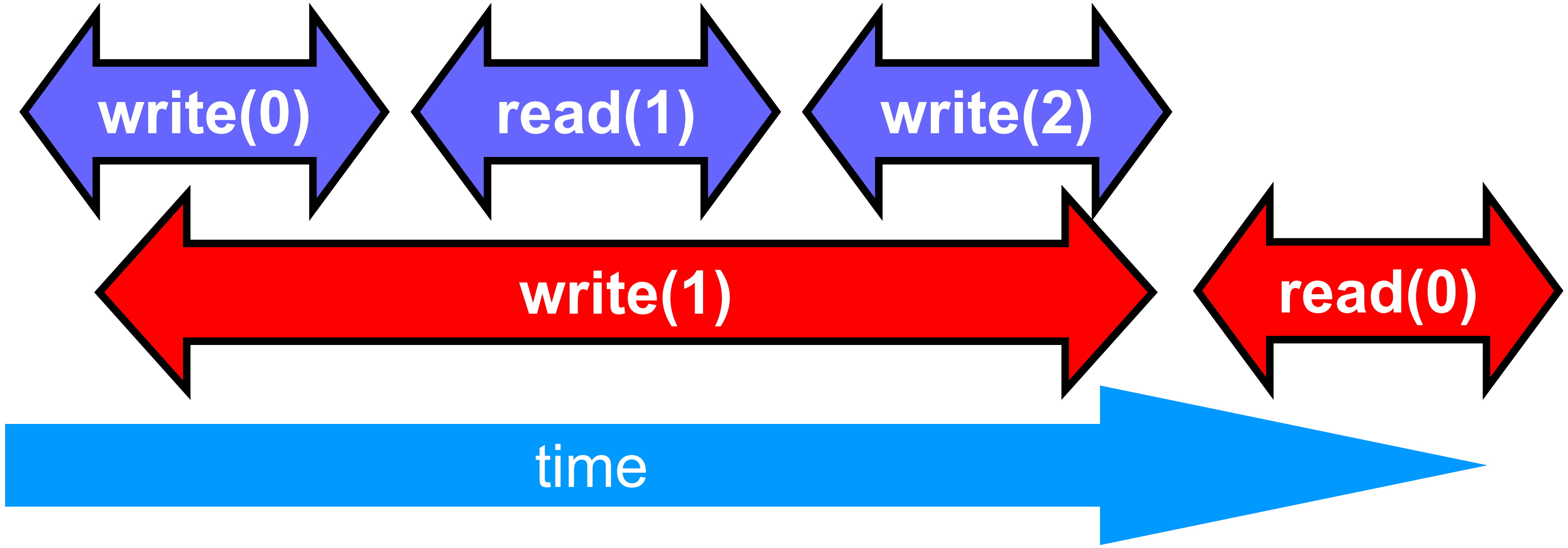
Comme ci
Comme ça



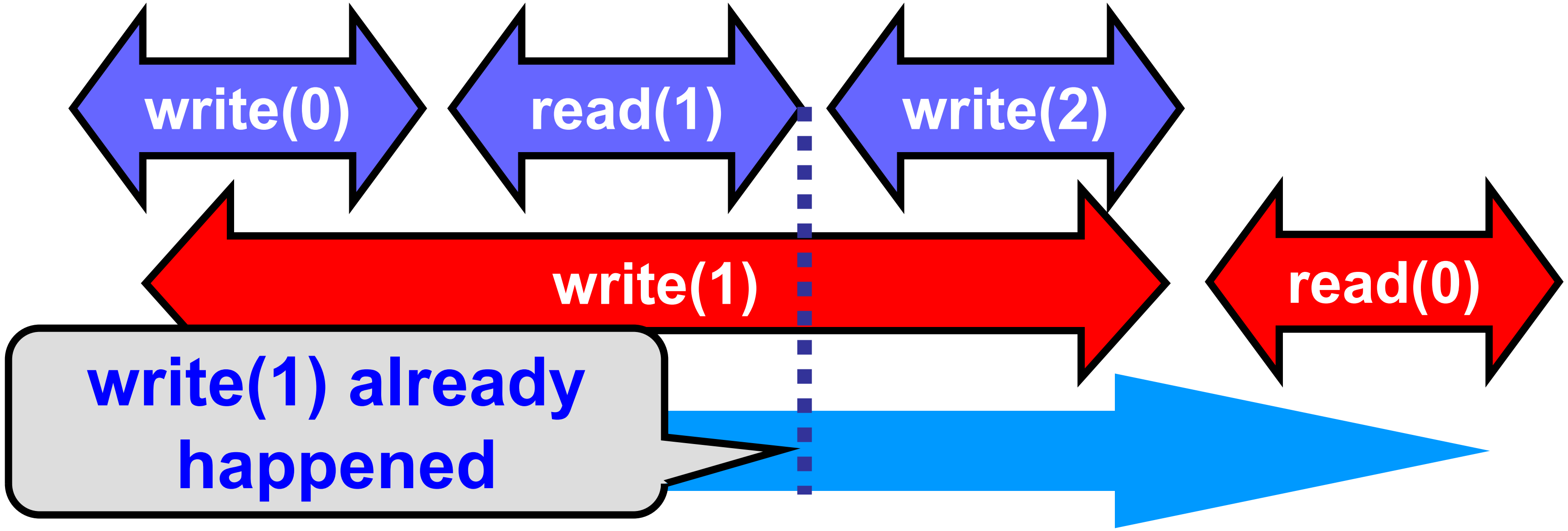
multiple orders OK

linearizable

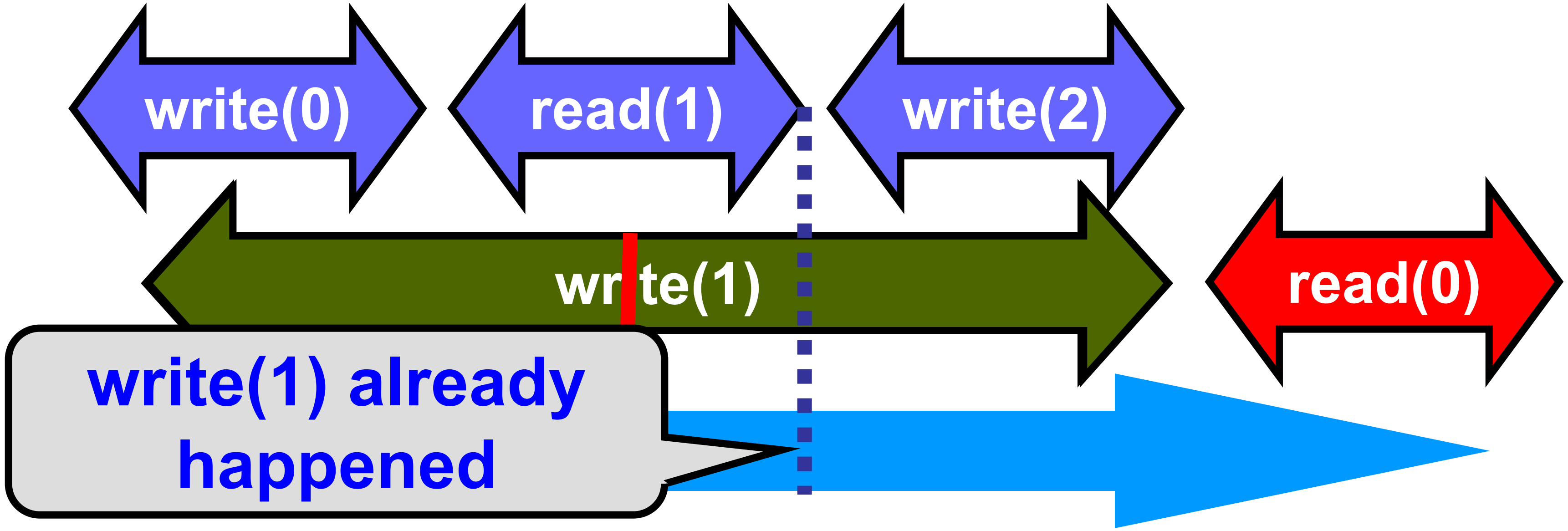
Read/Write Register Example



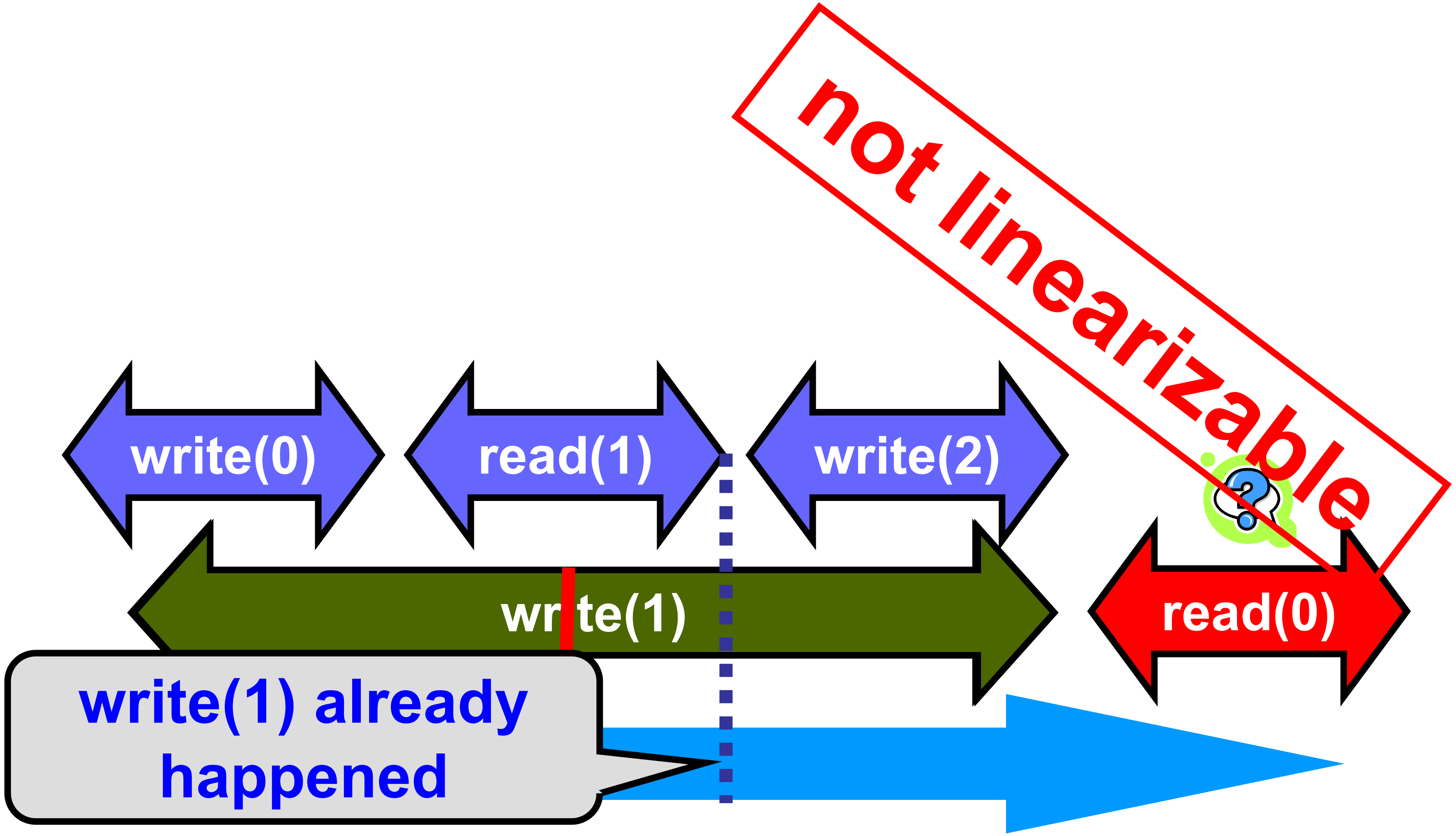
Read/Write Register Example



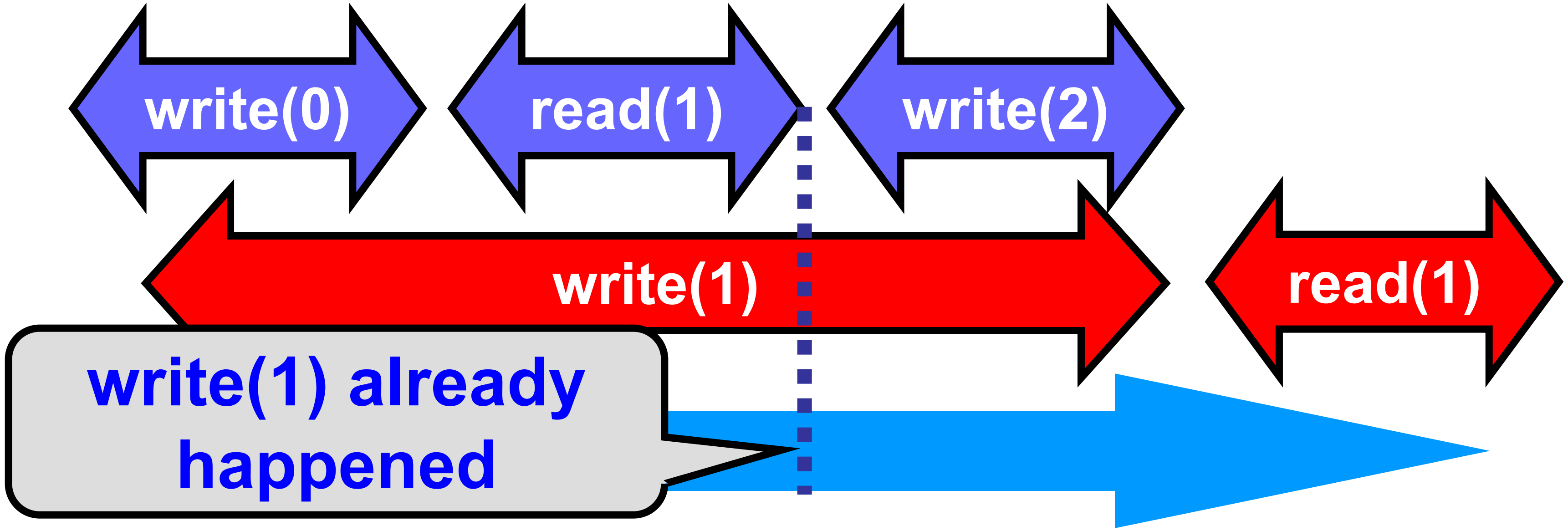
Read/Write Register Example



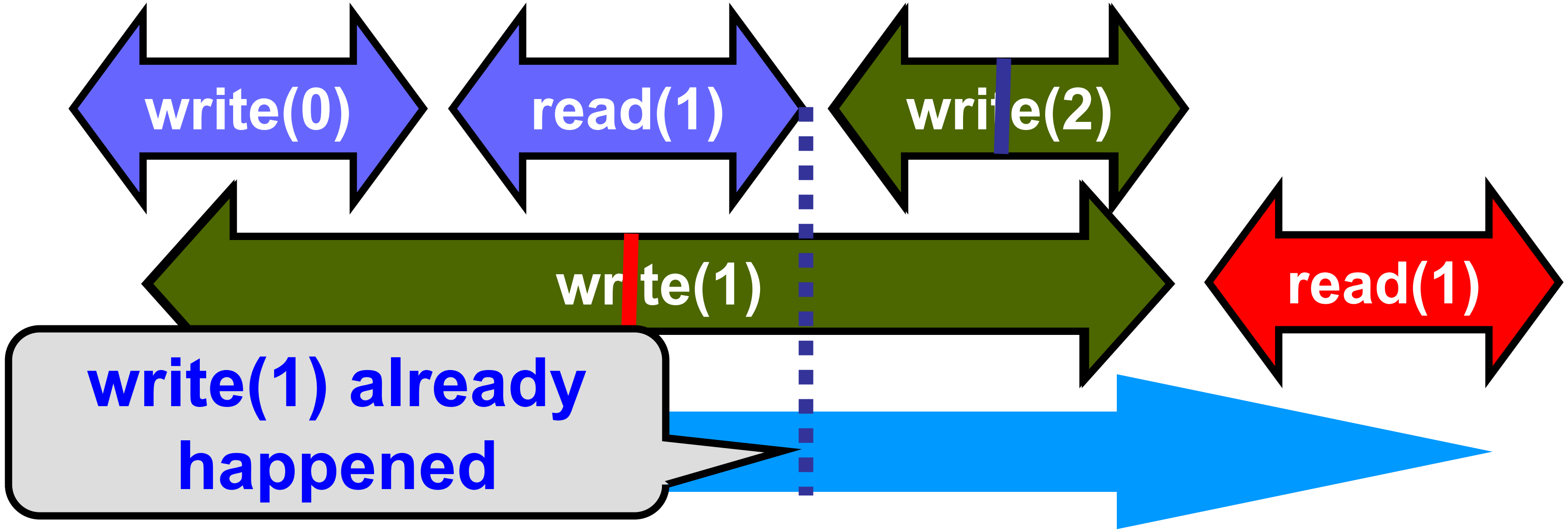
Read/Write Register Example



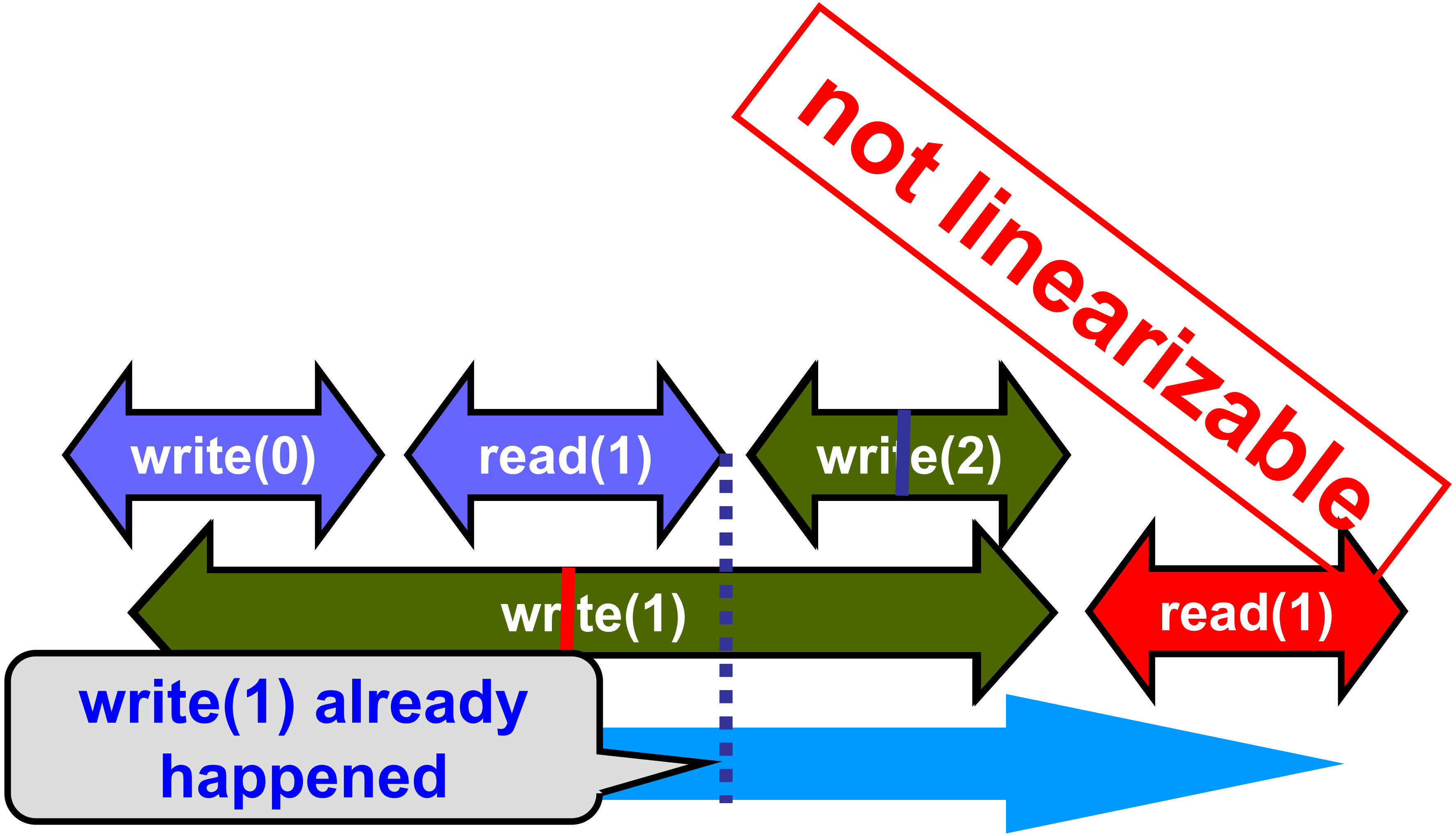
Read/Write Register Example



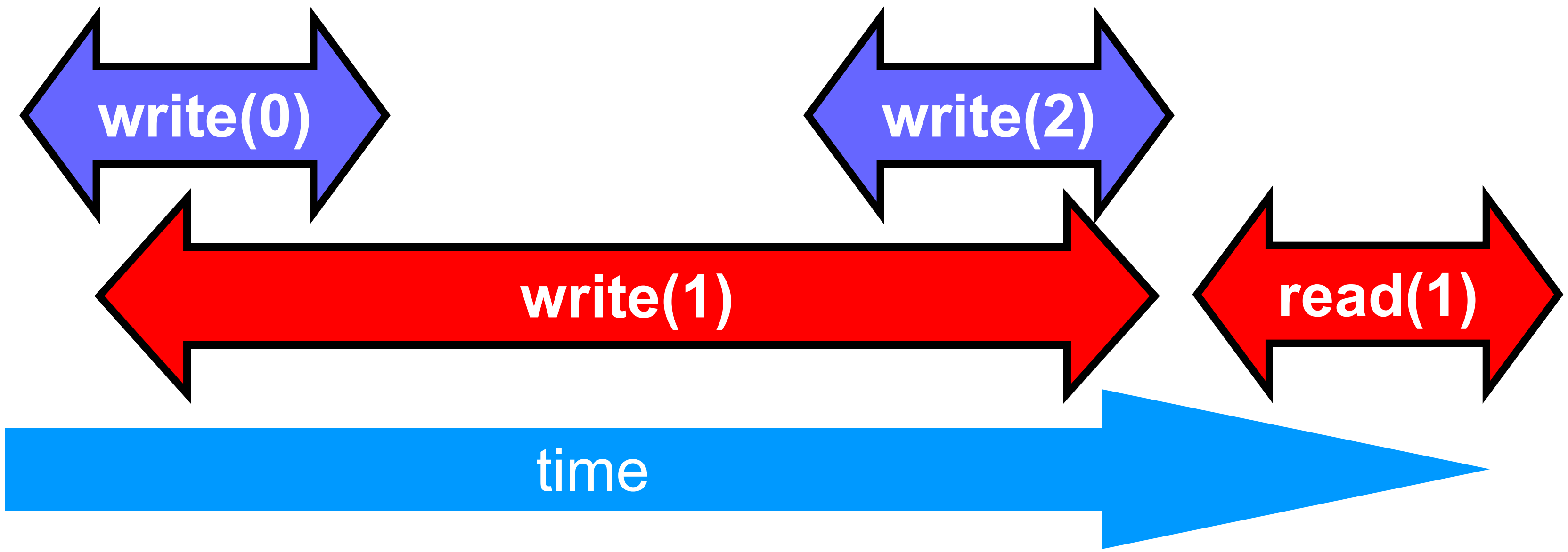
Read/Write Register Example



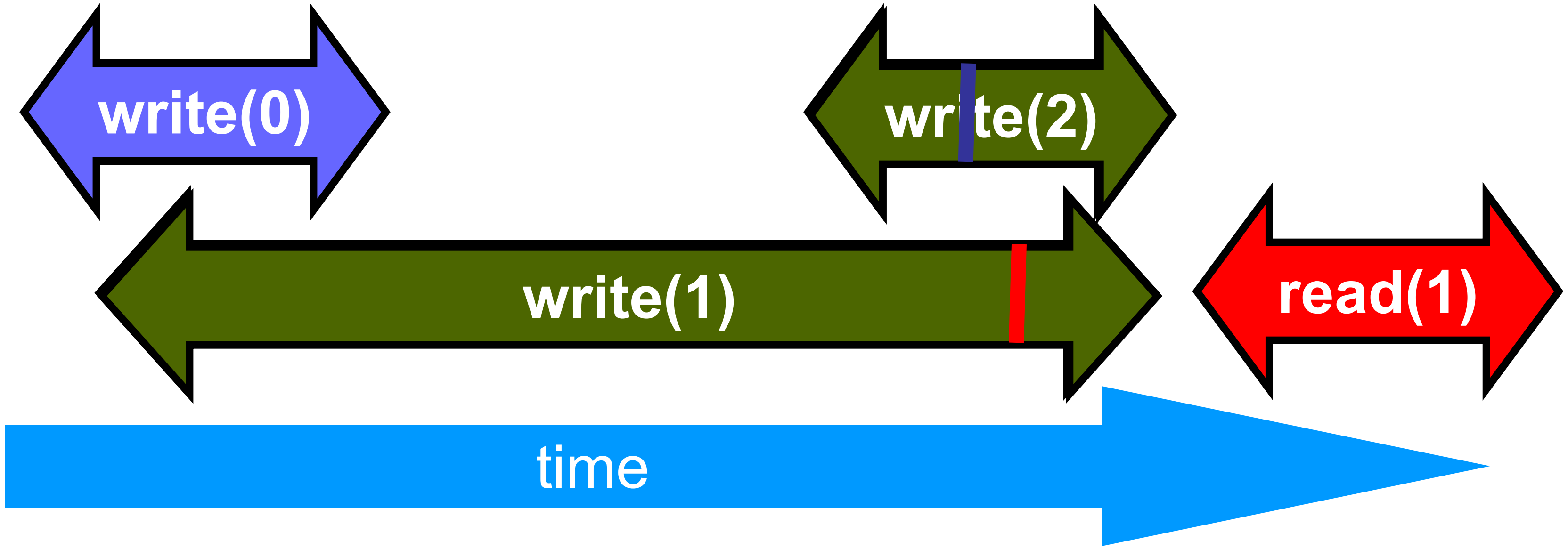
Read/Write Register Example



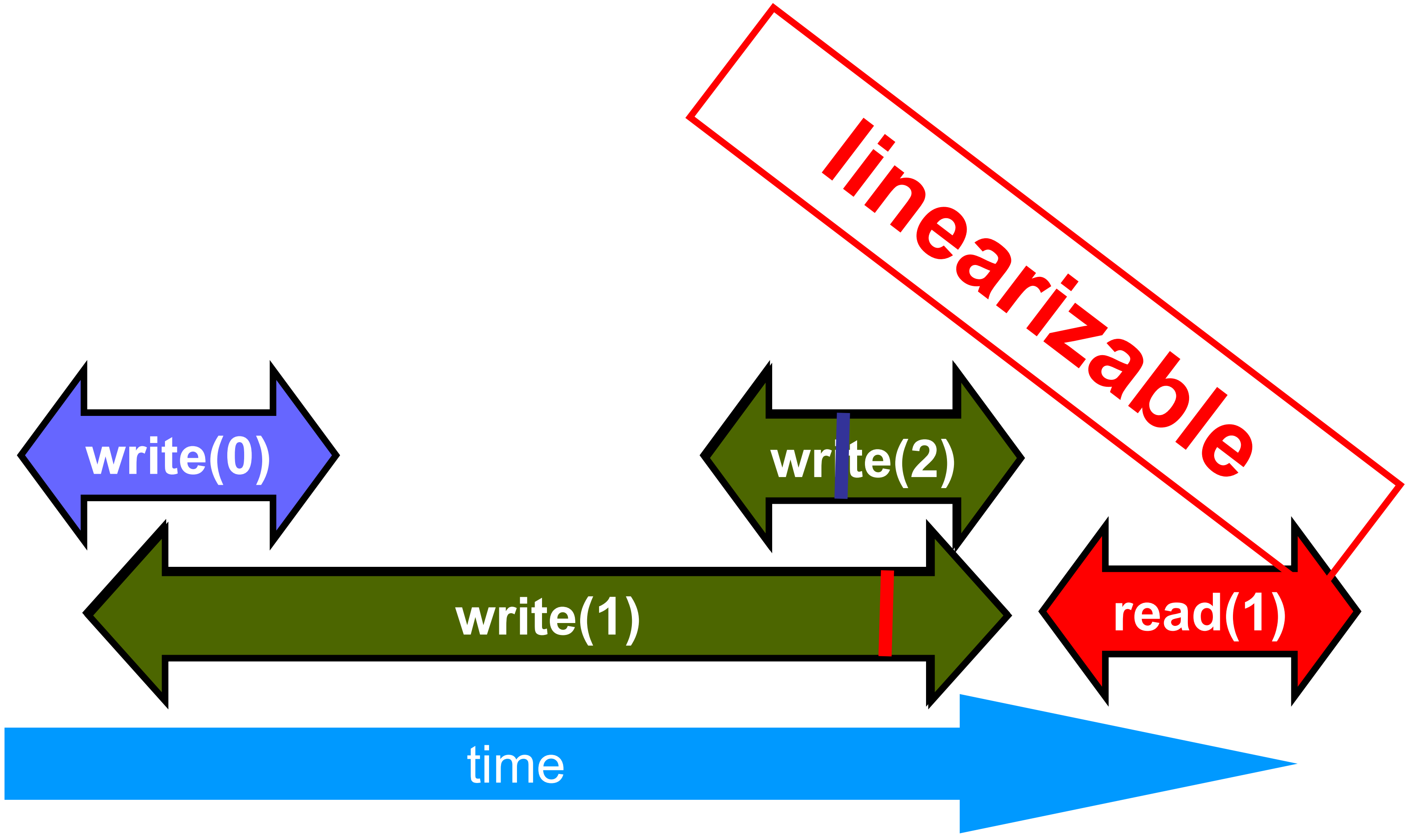
Read/Write Register Example



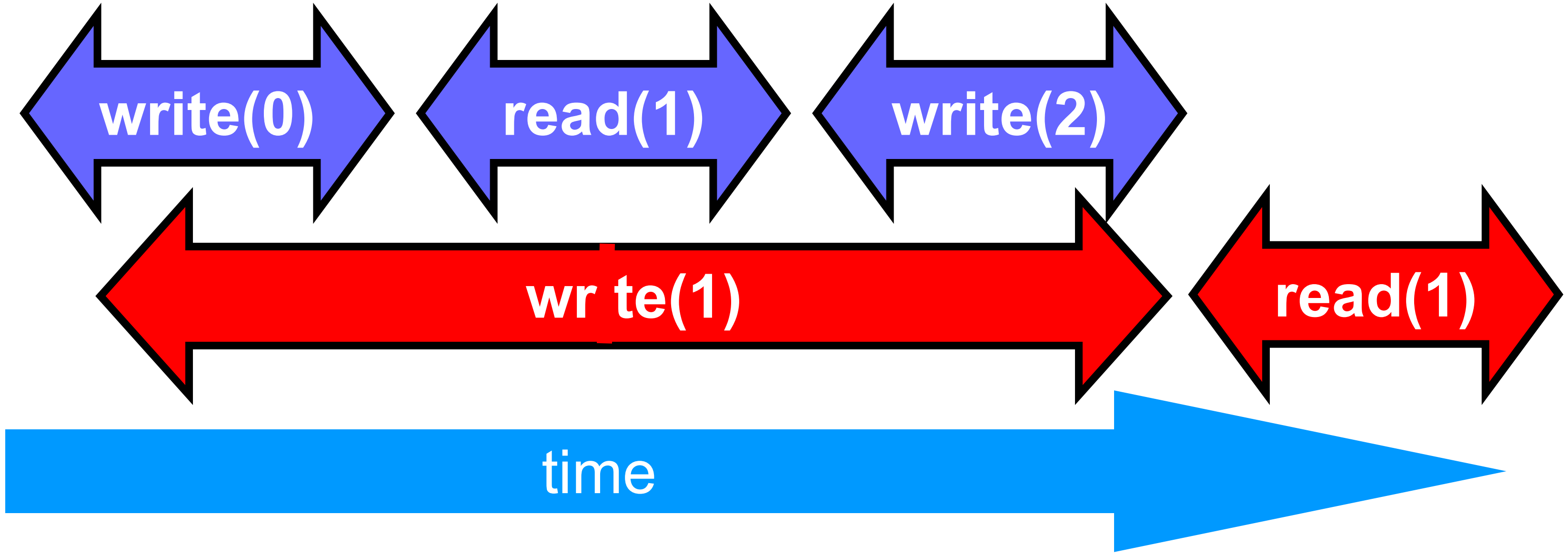
Read/Write Register Example



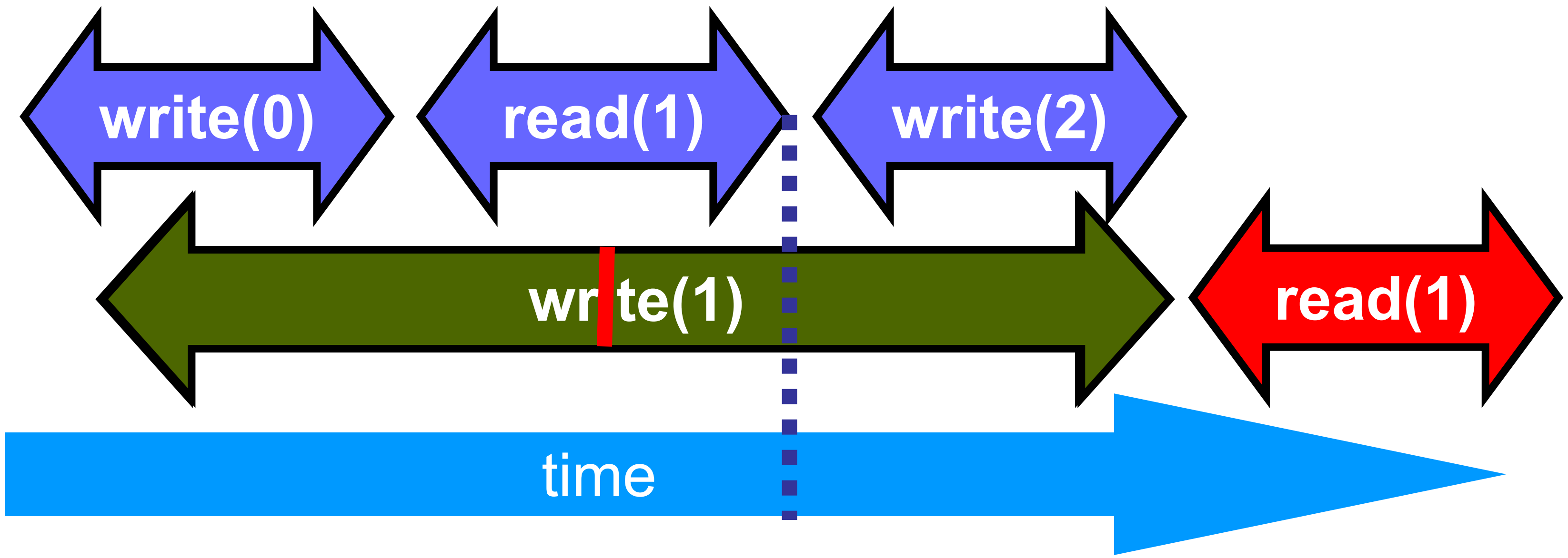
Read/Write Register Example



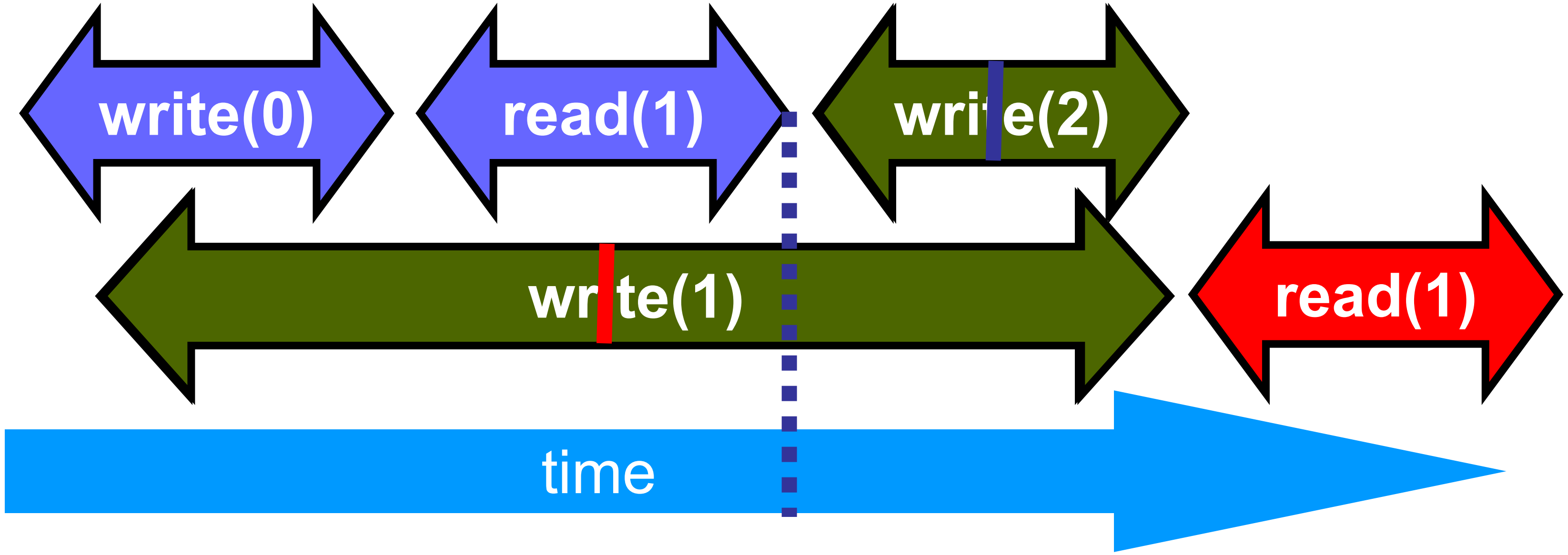
Read/Write Register Example



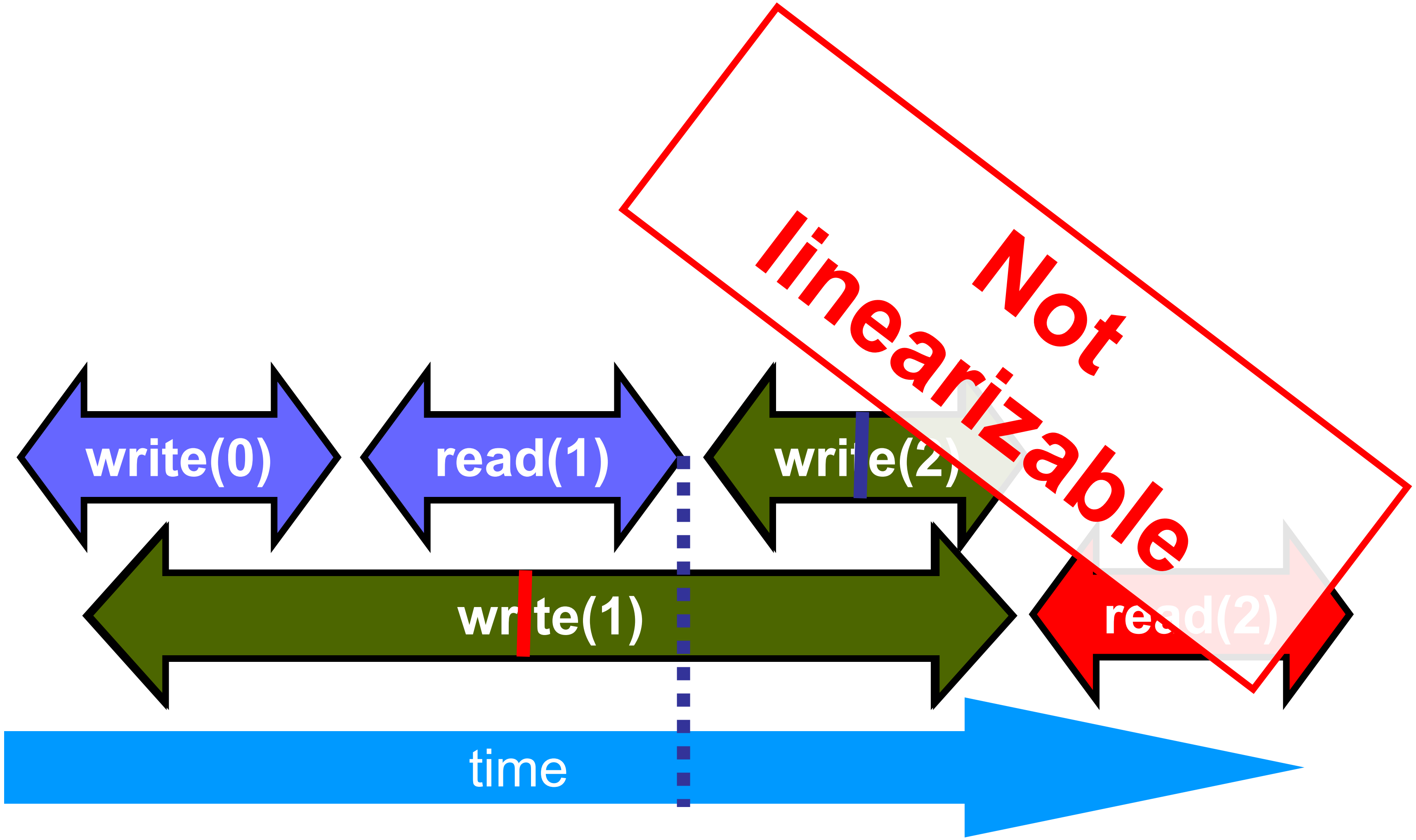
Read/Write Register Example



Read/Write Register Example



Read/Write Register Example



Talking About Executions

- Why?
  – Can’t we specify the linearization point of each operation without describing an execution?
- Not Always
  – In some cases, linearization point depends on the execution

Formal Model of Executions

- Define precisely what we mean
  - Ambiguity is bad when intuition is weak
- Allow reasoning
  - Formal
  - But mostly informal
    - In the long run, actually more important

Split Method Calls into Two Events

- Invocation
  - method name & args
  - `q.enq(x)`
- Response
  - result or exception
  - `q.enq(x)` returns `void`
  - `q.deq()` returns `x`
  - `q.deq()` throws `empty`

Invocation Notation

A q.enq(x)

Invocation Notation

A q.enq(x)

thread

Invocation Notation

A q.enq(x)

thread

method

Invocation Notation

A q.enq(x)

thread

method

object

Invocation Notation

A q.enq(x)

thread

method

object

arguments

Response Notation

A q: void

Response Notation

A q: void

thread

Response Notation

A q: void

thread

result

Response Notation

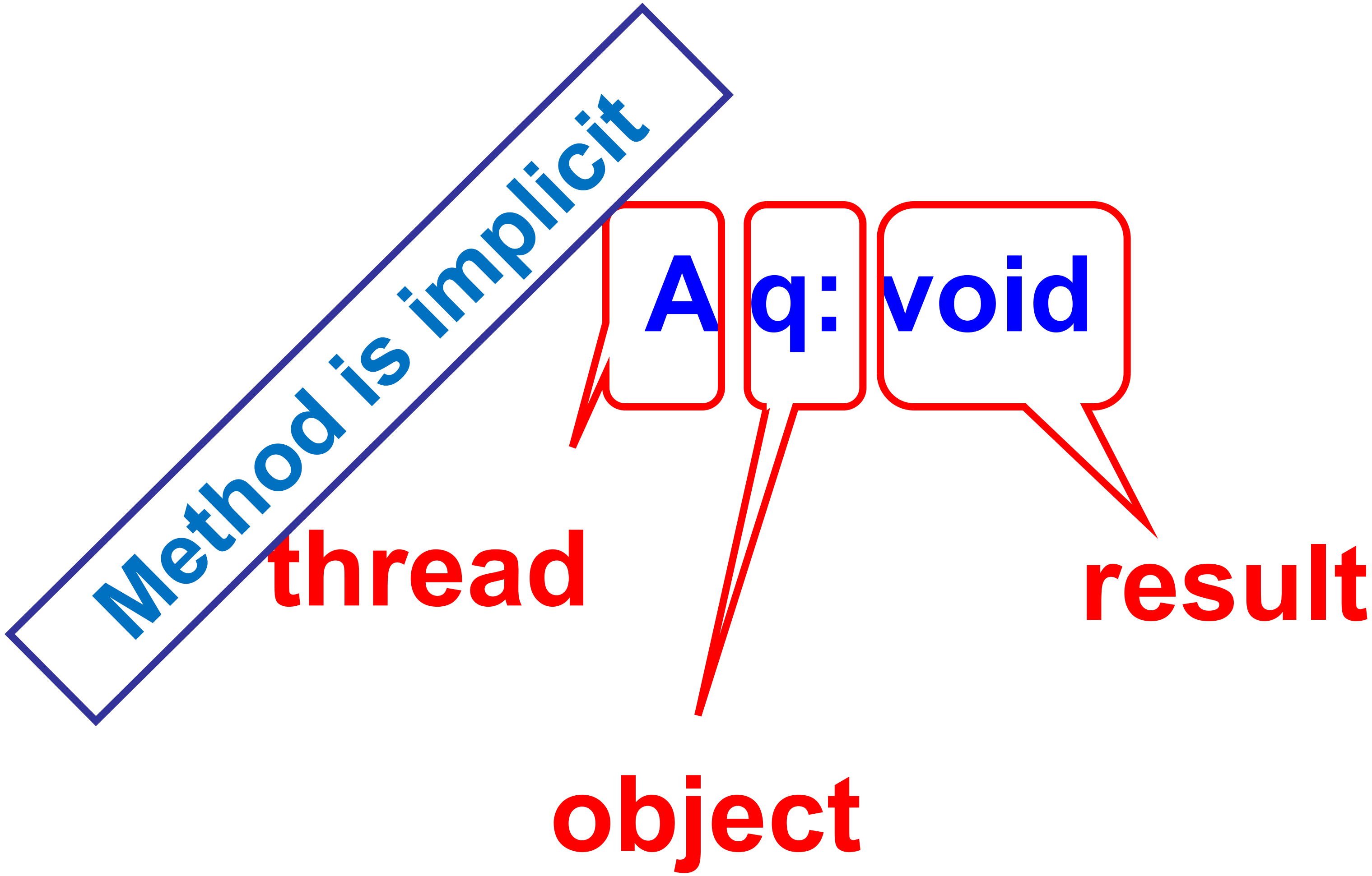
A q: void

thread

result

object

Response Notation



Response Notation

Method is implicit

A q: empty()

thread

object

exception

History - Describing an Execution

```
      A q.enq(3)
      A q:void
      A q.enq(5)
H =   B p.enq(4)
      B p:void
      B q.deq()
      B q:3
```

Sequence of invocations and responses

Definition

- Invocation & response *match* if

Thread names agree

Object names agree

```
A q.enq(3)
A q:void
```

Method call

Object Projections

```
      A q.enq(3)
      A q:void
H =   B p.enq(4)
      B p:void
      B q.deq()
      B q:3
```

Object Projections

```
A q.enq(3)
A q:void
```

H|q =

```
B q.deq()
B q:3
```

Thread Projections

H =
```
A q.enq(3)
A q:void
B p.enq(4)
B p:void
B q.deq()
B q:3
```

Thread Projections

$H|B =$

```
B p.enq(4)
B p:void
B q.deq()
B q:3
```

Complete Subhistory

```
      A q.enq(3)
      A q:void
      A q.enq(5)
H =   B p.enq(4)
      B p:void
      B q.deq()
      B q:3
```

An invocation is *pending* if it has no matching respnse

Complete Subhistory

```
      A q.enq(3)
      A q:void
      A q.enq(5)
H =   B p.enq(4)
      B p:void
      B q.deq()
      B q:3
```

May or may not have taken effect

Complete Subhistory

```
      A q.enq(3)
      A q:void
      A q.enq(5)
H =   B p.enq(4)
      B p:void
      B q.deq()
      B q:3
```

discard pending invocations

Complete Subhistory

```
A q.enq(3)
A q:void
```

Complete(H) =

```
B p.enq(4)
B p:void
B q.deq()
B q:3
```

Sequential Histories

```
A q.enq(3)
A q:void
B p.enq(4)
B p:void
B q.deq()
B q:3
A q:enq(5)
```

Sequential Histories

```
A q.enq(3)
A q:void
B p.enq(4)
B p:void
B q.deq()
B q:3
A q:enq(5)
```

match

(4)

Sequential Histories

```
A q.enq(3)
A q:void
B p.enq(4)
B p:void
B q.deq()
B q:3
A q:enq(5)
```

match

match

Sequential Histories

```
A q.enq(3)
A q:void
B p.enq(4)
B p:void
B q.deq()
B q:3
A q:enq(5)
```

match

match

match

Sequential Histories

```
A q.enq(3)
A q:void
B p.enq(4)
B p:void
B q.deq()
B q:3
A q:enq(5)
```

match

match

match

Final pending invocation OK

Sequential Histories

```
A q.enq(3)
A q:void
B p.enq(4)
B p:void
B q.deq()
B q:3
A q:enq(5)
```

match

match

match

Final pending invocation OK

Method calls of different threads do not interleave

Well-Formed Histories

```
    A q.enq(3)
    B p.enq(4)
    B p:void
H=  B q.deq()
    A q:void
    B q:3
```

Well-Formed Histories

Per-thread projections sequential

$H|B$=

```
B p.enq(4)
B p:void
B q.deq()
B q:3
```

H=

```
A q.enq(3)
B p.enq(4)
B p:void
B q.deq()
A q:void
B q:3
```

Well-Formed Histories

Per-thread projections sequential

```
H=   A q.enq(3)
     B p.enq(4)
     B p:void
     B q.deq()
     A q:void
     B q:3
```

```
H|B= B p.enq(4)
     B p:void
     B q.deq()
     B q:3
```

```
H|A= A q.enq(3)
     A q:void
```

Equivalent Histories

Threads see the same thing in both

$$\begin{cases} H|A = G|A \\ H|B = G|B \end{cases}$$

H=

```
A q.enq(3)
B p.enq(4)
B p:void
B q.deq()
A q:void
B q:3
```

G=

```
A q.enq(3)
A q:void
B p.enq(4)
B p:void
B q.deq()
B q:3
```

Sequential Specifications

- A sequential specification is some way of telling whether a
  - Single-thread, single-object history
  - Is legal
- For example:
  - Pre and post-conditions
  - But plenty of other techniques exist …

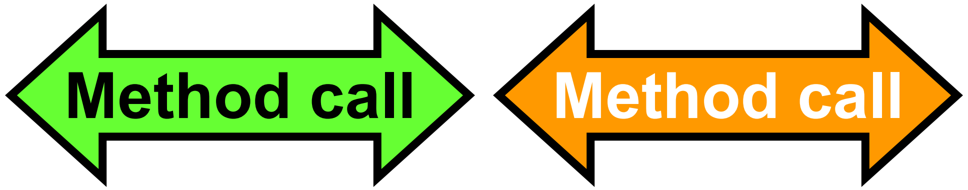
Legal Histories

- A sequential (multi-object) history H is legal if
  - For every object **x**
  - **H|x** is in the sequential spec for **x**

Precedence

```
A q.enq(3)
B p.enq(4)
B p.void
A q:void
B q.deq()
B q:3
```

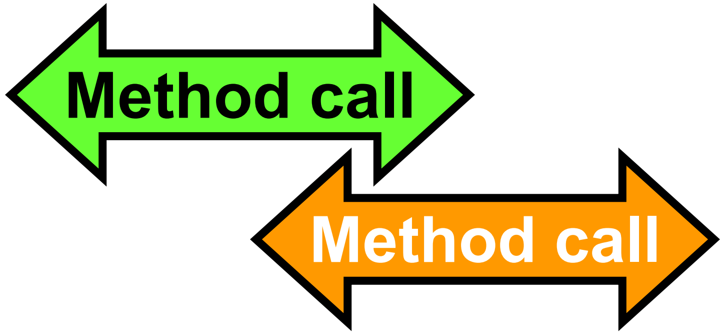
A method call precedes another if response event precedes invocation event



Non-Precedence

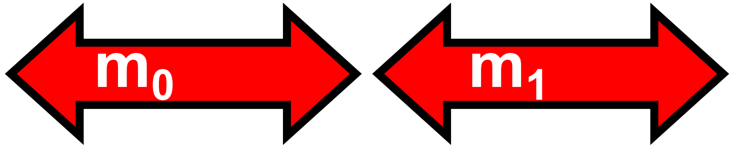
```
A q.enq(3)
B p.enq(4)
B p.void
B q.deq()
A q:void
B q:3
```

Some method calls overlap one another



Notation

- Given
  - History **H**
  - method executions $\mathbf{m_0}$ and $\mathbf{m_1}$ in **H**
- We say $m_0 \rightarrow_{\mathbf{H}} m_1$, if
  - $\mathbf{m_0}$ precedes $\mathbf{m_1}$
- Relation $\mathbf{m_0} \rightarrow_{\mathbf{H}} \mathbf{m_1}$ is a
  - Partial order
  - Total order if **H** is sequential



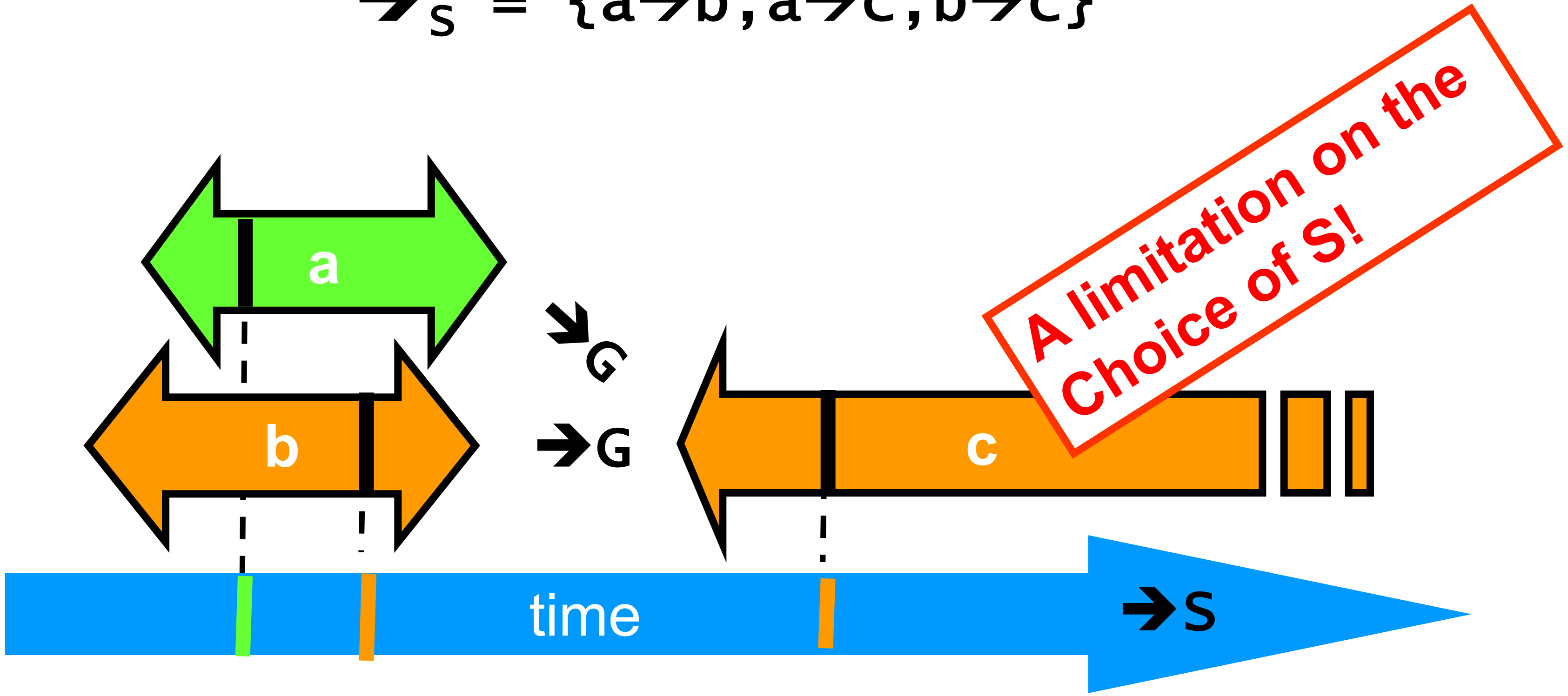
Linearizability

- History H is ***linearizable*** if it can be extended to **G** by
  - Appending zero or more responses to pending invocations
  - Discarding other pending invocations
- So that G is equivalent to
  - Legal sequential history **S**
  - where $\rightarrow_{\mathbf{G}} \subset \rightarrow_{\mathbf{S}}$

Ensuring $\rightarrow_G \subset \rightarrow_S$

$\rightarrow_G$ = {a→c,b→c}

$\rightarrow_S$ = {a→b,a→c,b→c}

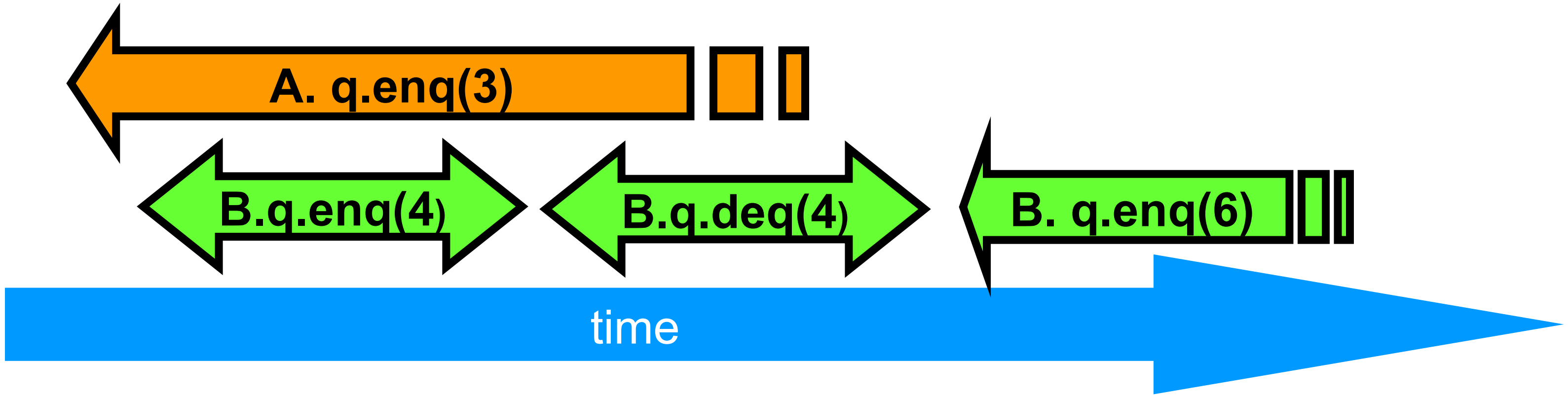


Remarks

- Some pending invocations
  - Took effect, so keep them
  - Discard the rest
- Condition $\rightarrow_{\mathbf{G}} \subset \rightarrow_{\mathbf{S}}$
  - Means that **S** respects "real-time order" of **G**

Example

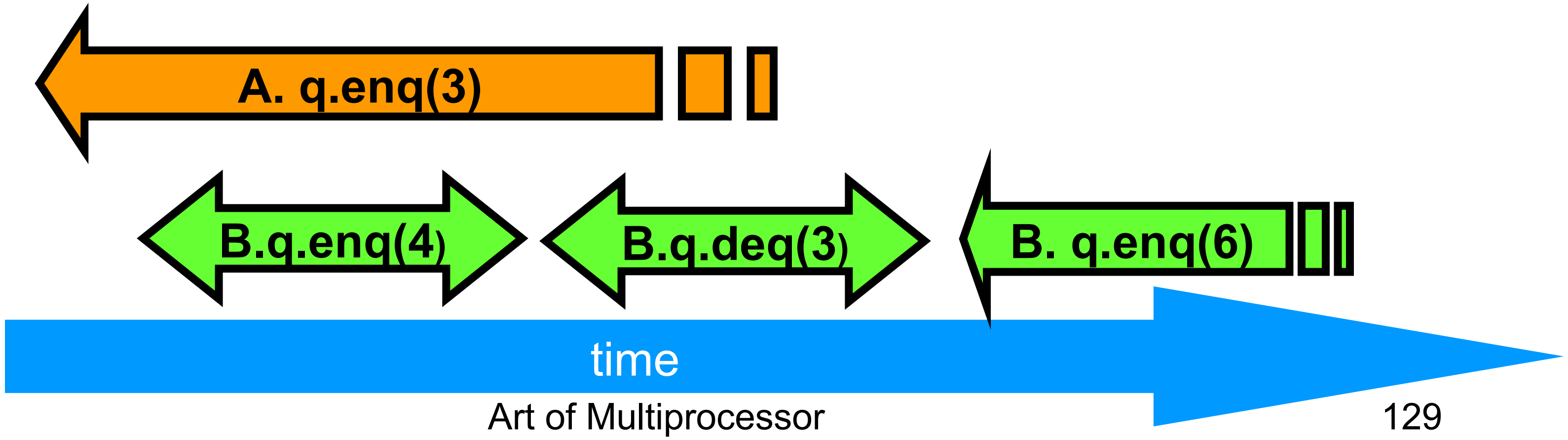
```
A q.enq(3)
B q.enq(4)
B q:void
B q.deq()
B q:4
B q:enq(6)
```



Example

```
A q.enq(3)
B q.enq(4)
B q:void
B q.deq()
B q:4
B q:enq(6)
```

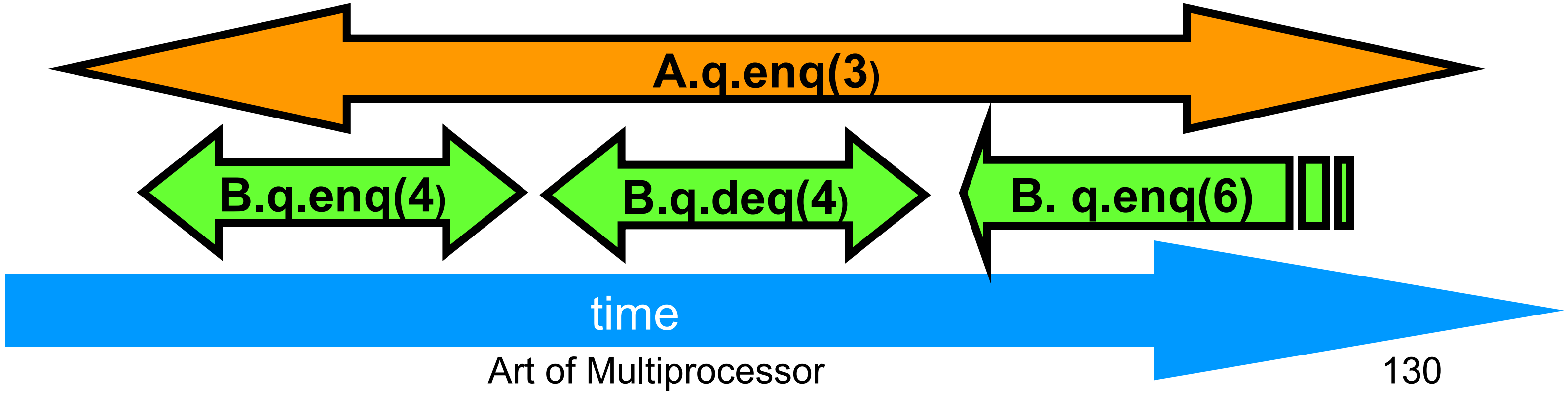
Complete this pending invocation



Example

```
A q.enq(3)
B q.enq(4)
B q:void
B q.deq()
B q:4
B q:enq(6)
A q:void
```

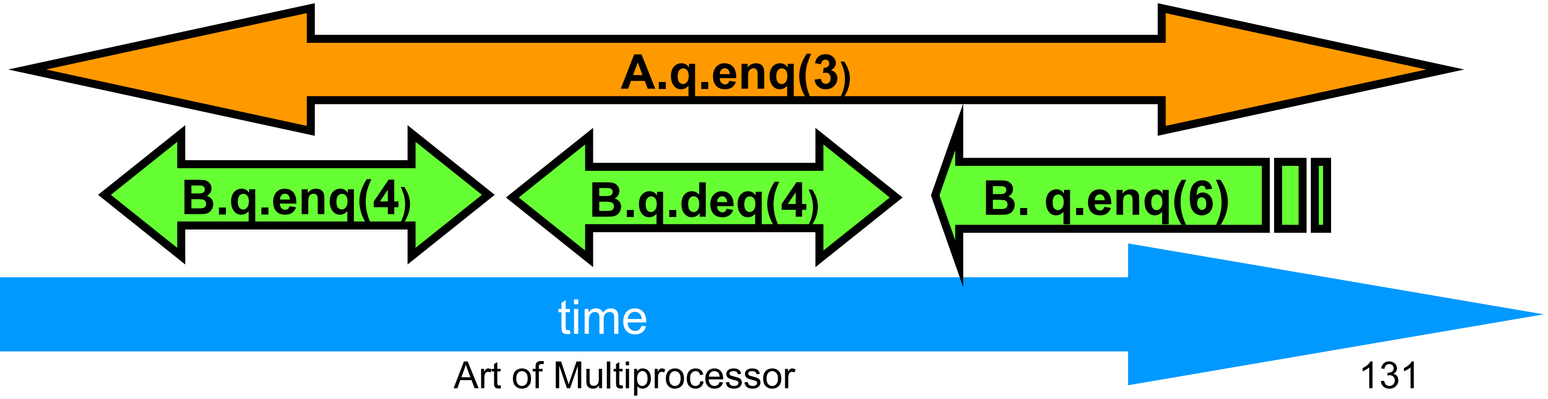
Complete this pending invocation



Example

discard this one

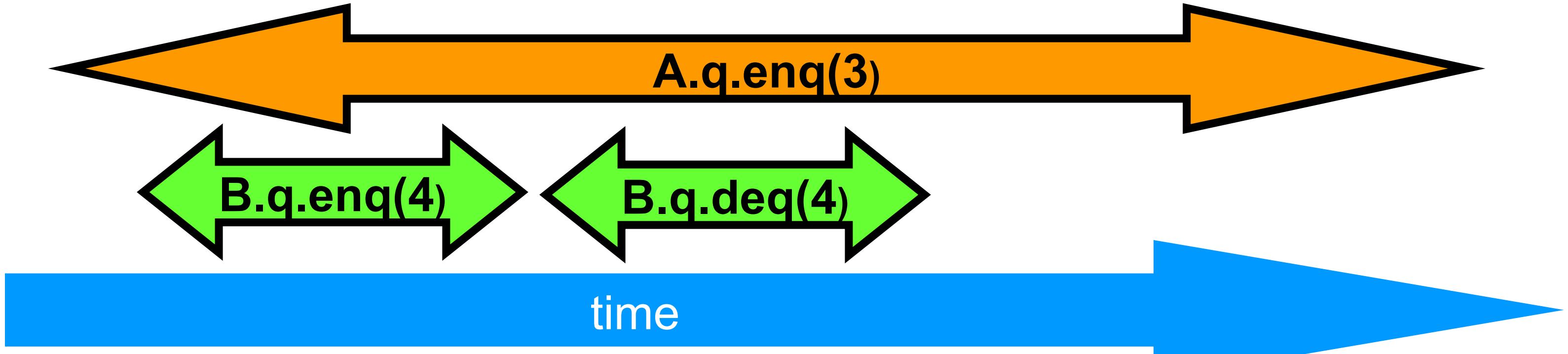
```
A q.enq(3)
B q.enq(4)
B q:void
B q.deq()
B q:4
B q:enq(6)
A q:void
```



Example

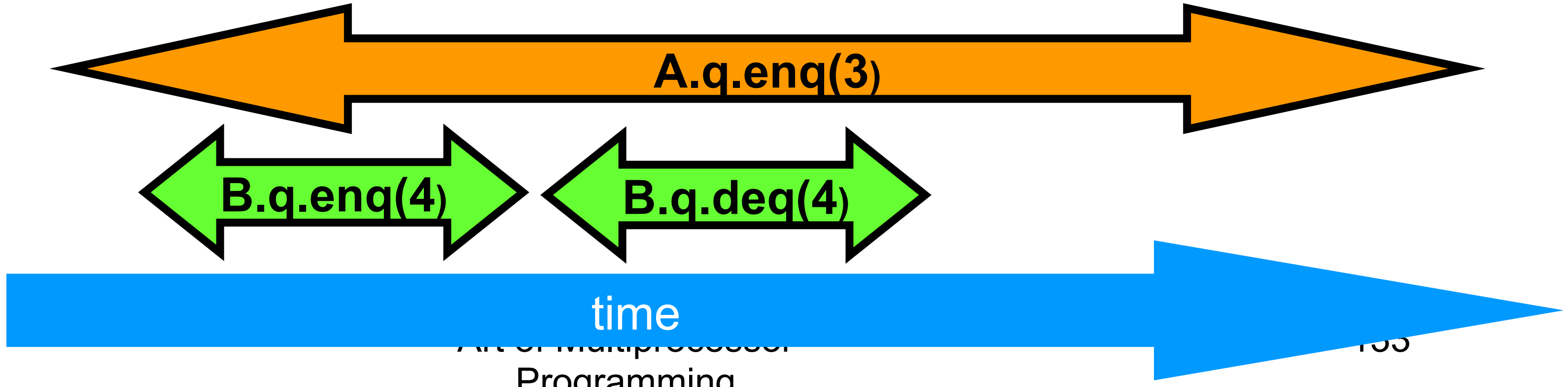
```
A q.enq(3)
B q.enq(4)
B q:void
B q.deq()
B q:4

A q:void
```

discard this one



Example

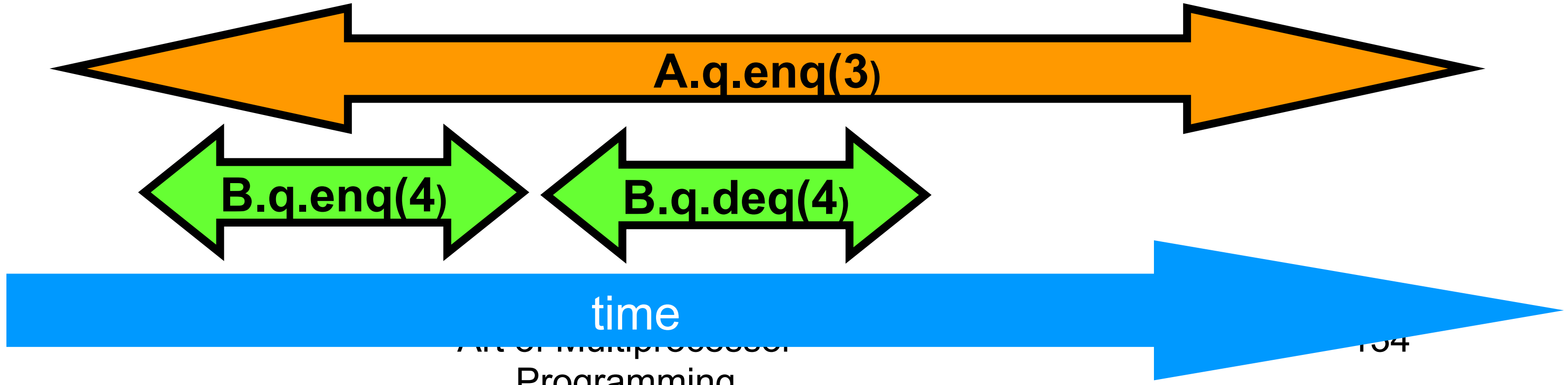
```
A q.enq(3)
B q.enq(4)
B q:void
B q.deq()
B q:4
A q:void
```



Example

```
A q.enq(3)
B q.enq(4)
B q:void
B q.deq()
B q:4
A q:void
```

```
B q.enq(4)
B q:void
A q.enq(3)
A q:void
B q.deq()
B q:4
```

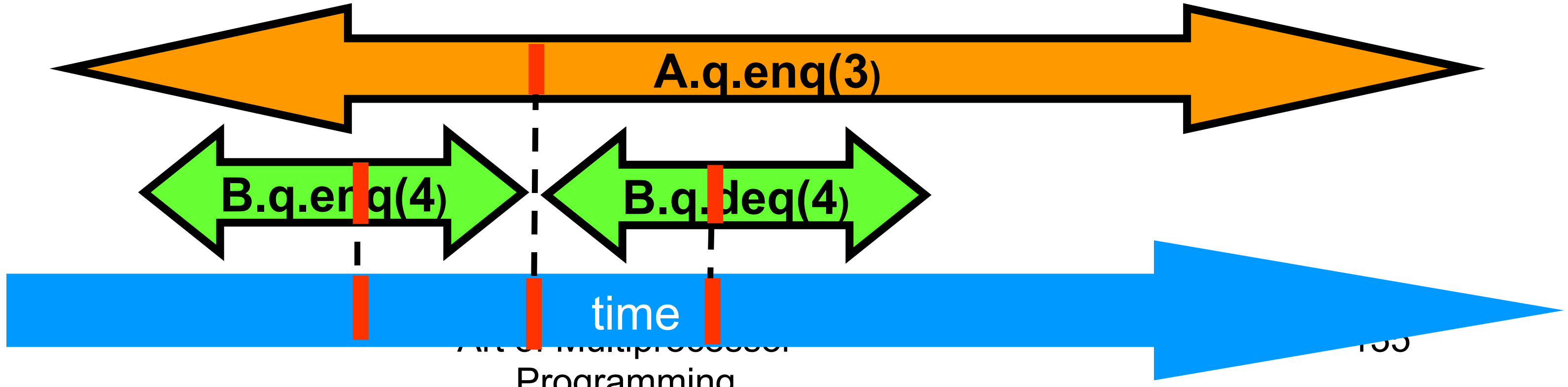


Example

```
A q.enq(3)
B q.enq(4)
B q:void
B q.deq()
B q:4
A q:void
```

Equivalent sequential history

```
B q.enq(4)
B q:void
A q.enq(3)
A q:void
B q.deq()
B q:4
```



Composability Theorem

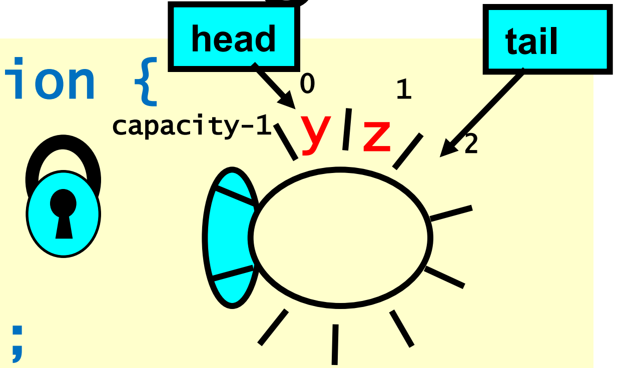
- History H is linearizable if and only if
  - For every object x
  - H|x is linearizable
- We care about objects only!
  - (Materialism?)

Why Does Composability Matter?

- Modularity
- Can prove linearizability of objects in isolation
- Can compose independently-implemented objects

Reasoning About Linearizability: 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();
  }
}
```



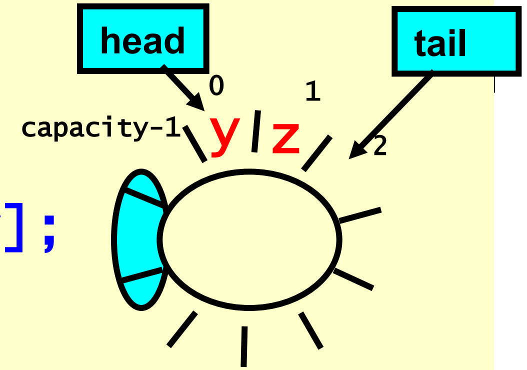
Reasoning About Linearizability: Locking

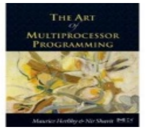
```
public T deq() throws EmptyException {
  lock.lock();
  try {
    if (tail == head)
       throw new EmptyException();
    T x = items[head % items.length];
    head++;
    return x;
  } finally {
    lock.unlock();
  }
}
```

Linearization points are when locks are released

More Reasoning: Wait-free

```
public class WaitFreeQueue {

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

  public void enq(Item x) {
    if (tail-head == capacity) throw
         new FullException();
    items[tail % capacity] = x; tail++;
  }
  public Item deq() {
     if (tail == head) throw
         new EmptyException();
     Item item = items[head % capacity]; head++;
     return item;
}}
```



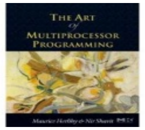


More Reasoning: Wait-free

```
public clas   itFreeQueue {

  int        l =
  it       new Ob

       id enq(Item x) {
       ail-head == capacity) throw
       new FullException();
     items[tail % capacity] = x; tail++;
  }
  public Item deq() {
     if (tail == head) throw
         new EmptyException();
     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



Strategy

- Identify one atomic step where method “happens”
  - Critical section
  - Machine instruction
- Doesn’t always work
  - Might need to define several different steps for a given method

Linearizability: Summary

- Powerful specification tool for shared objects
- Allows us to capture the notion of objects being “atomic”
- Don’t leave home without it