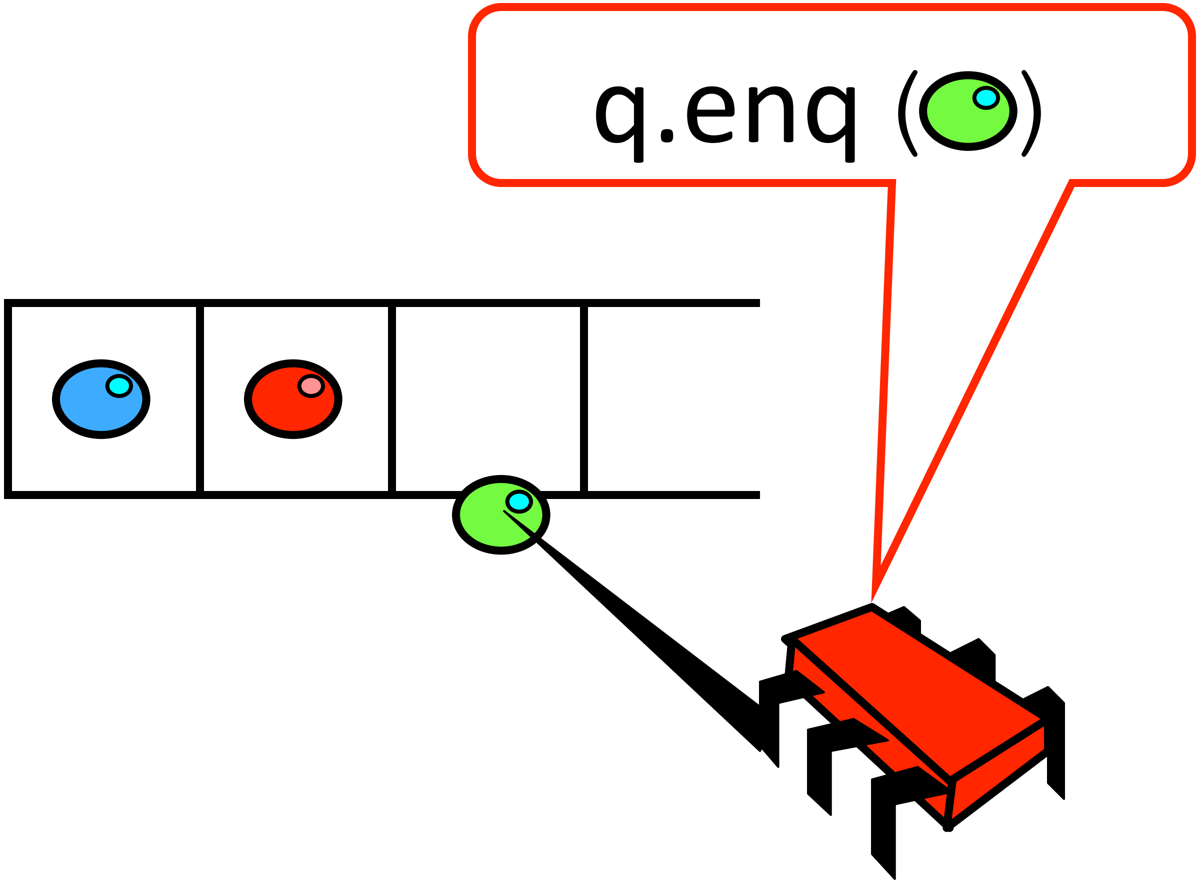
Correctness of Concurrent Objects

Luís Pina

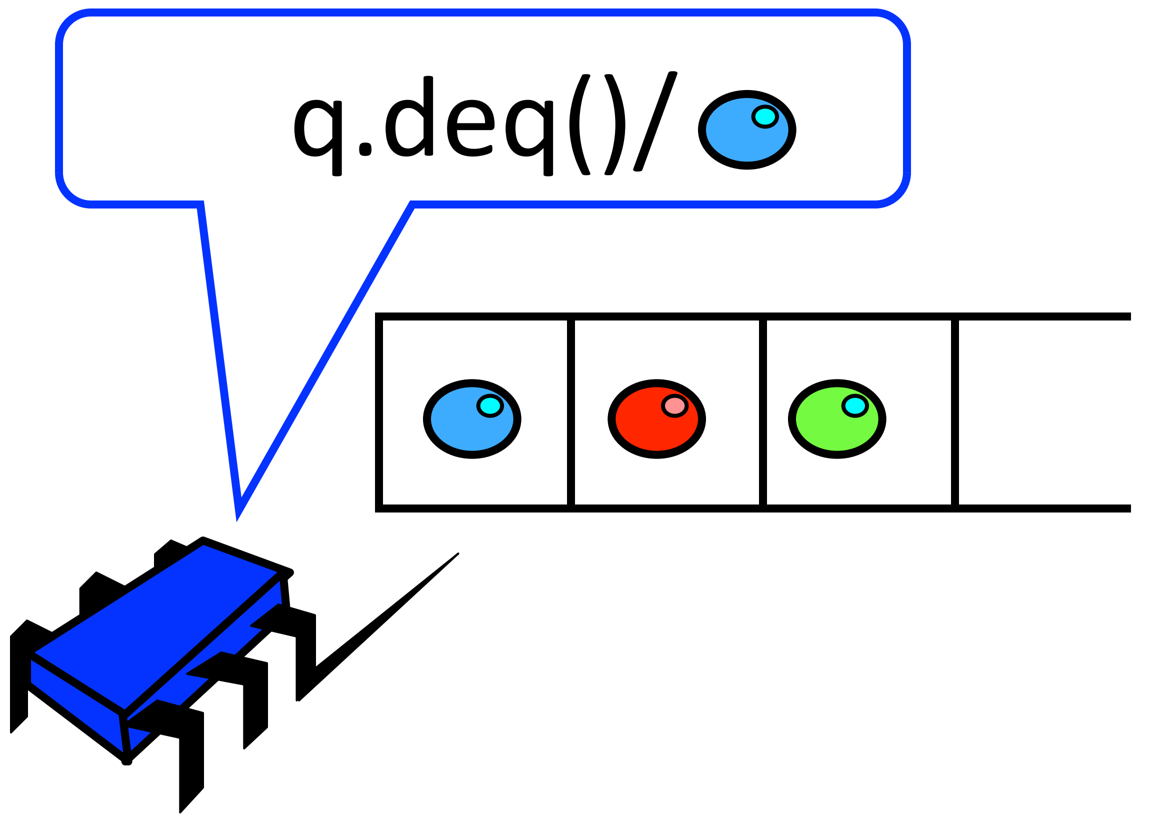
Disclaimer

- Some of these slides are adapted from the companion slides of the book "**The art of multiprocessor programming**" by Nir Shavit and Maurice Herlihy, shared with CC share-alike

Queue - `enq`



Queue - `deq`



Sequential Object Specification

- 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,

Sequential Queue Specification

Method `deq`

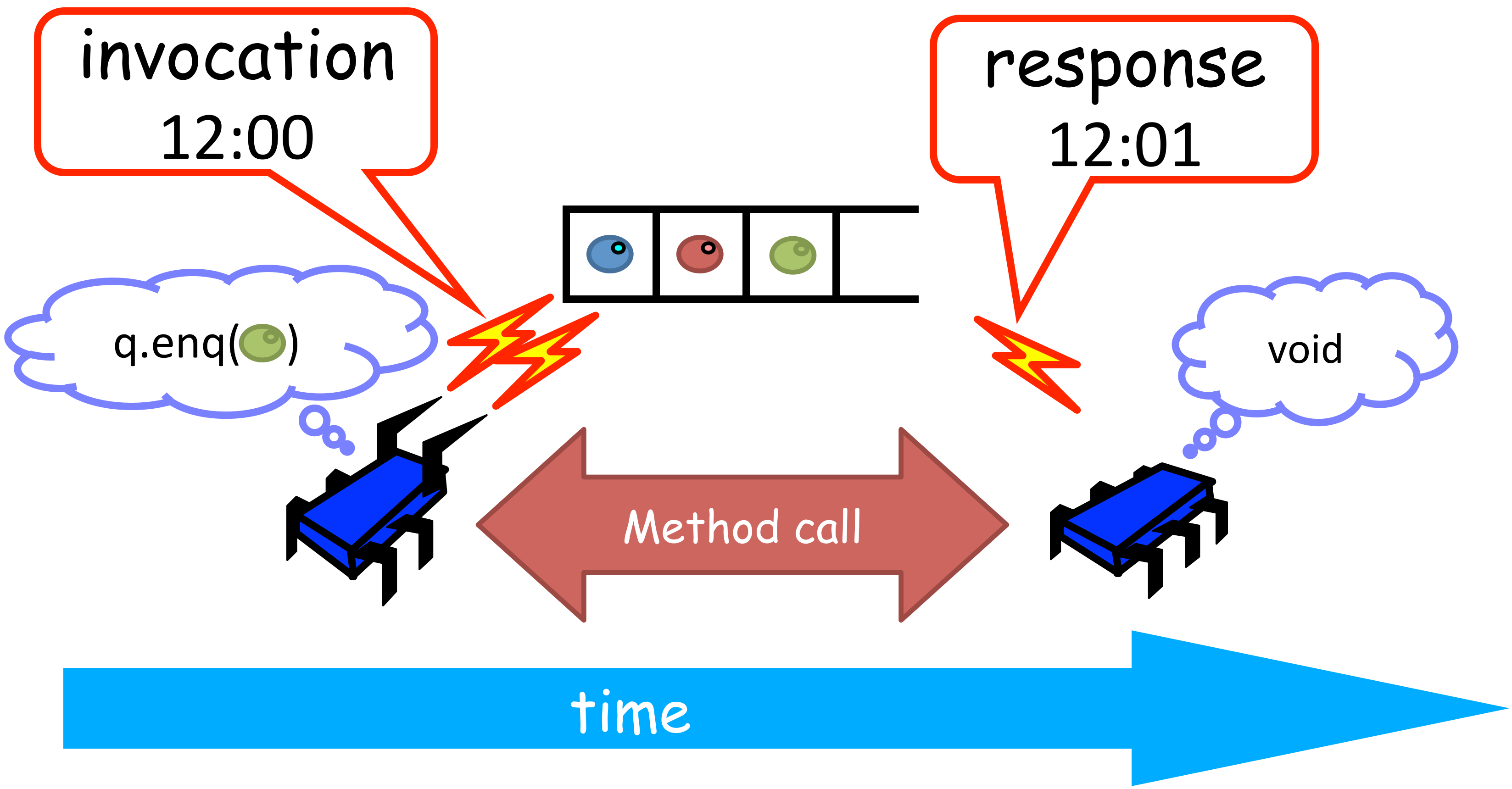
- **Precondition:**
  - First item in queue is `a`
- **Postcondition:**
  - Returns `a`
- **Postcondition:**
  - Removes first item in queue

Sequential Queue Specification

- 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 **concurrency**?

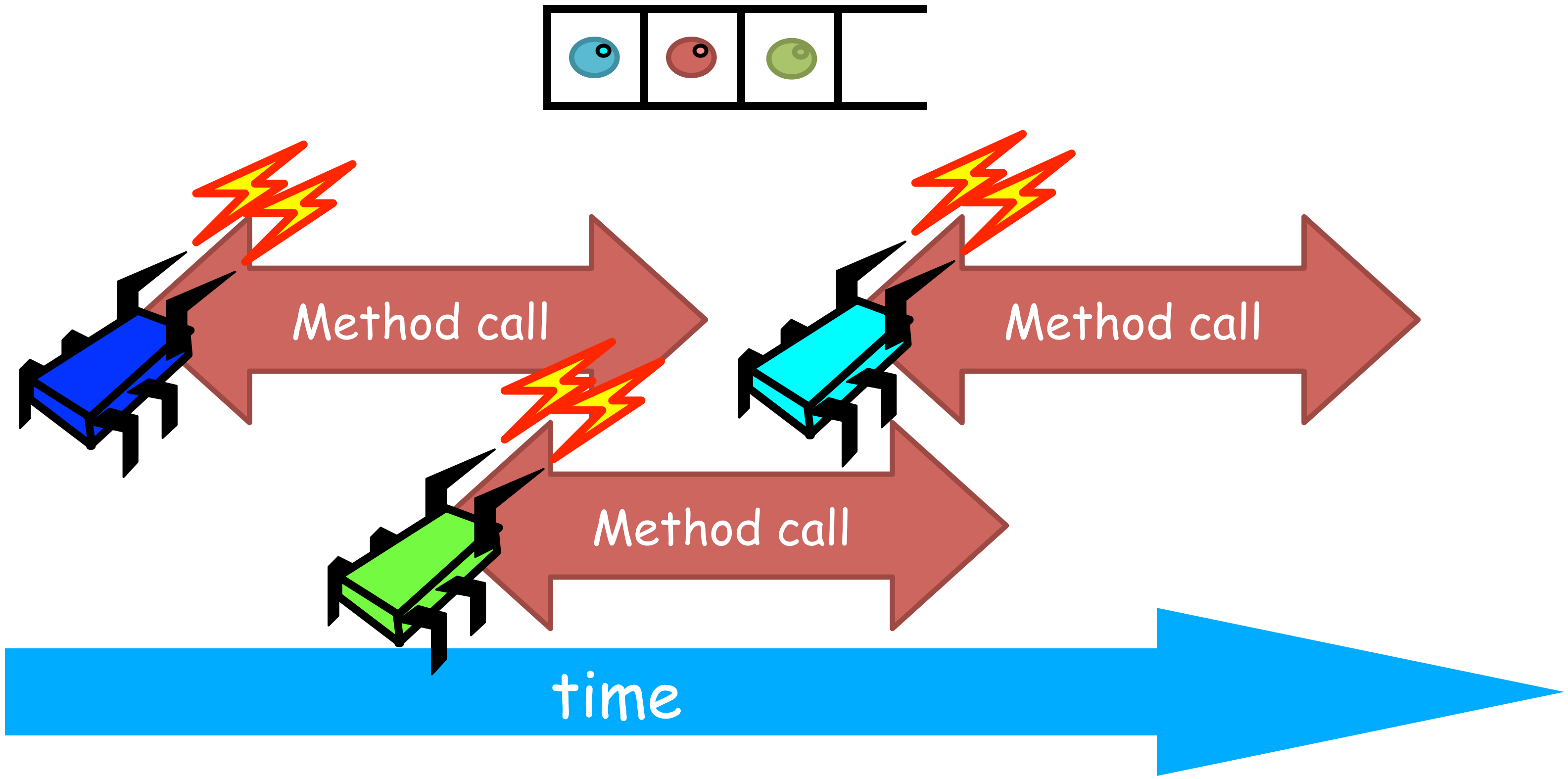
Methods take time



Sequential vs Concurrent

- Sequential
  - Methods take time? Really?
- Concurrent
  - Method call is not an event
  - Method call is an **interval**
    - Between two events
      - **Method call**
      - **Method return**

Concurrent methods take overlapping time



Sequential vs Concurrent

- Sequential
  - Object state only changes between method calls
- Concurrent
  - Due to overlap, object **might never** be between method calls

Sequential vs Concurrent

- Sequential
  - Each method described in isolation
- Concurrent
  - Every method interacts with every other method
    - Even with itself!
    - Total interactions = Cartesian product of all instructions in all methods with the number of threads

Sequential vs Concurrent

- Sequential
  - Each method described in isolation
- Concurrent
  - Every method interacts with every other method
    - Even with itself!
    - Total interactions = Cartesian product of all instructions in all methods with the number of threads

Panic!

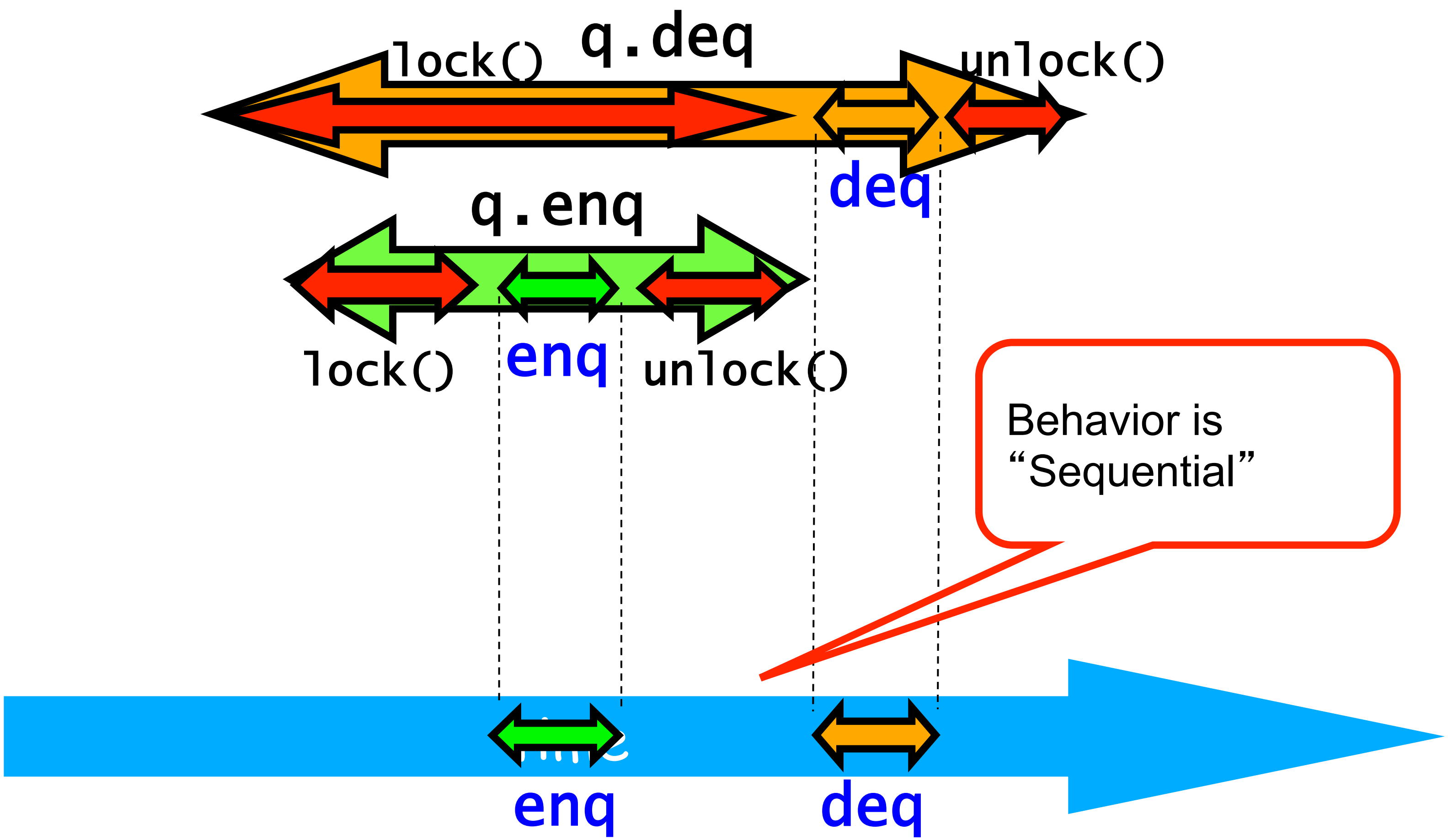
Mutual Exclusion

```
public class ConcurrentQueue extends Queue
  private Lock lock = new Lock();

  public Object dequeue()
    lock.lock();
    try
      return super.dequeue();
    finally
      lock.unlock();
```

Mutual Exclusion

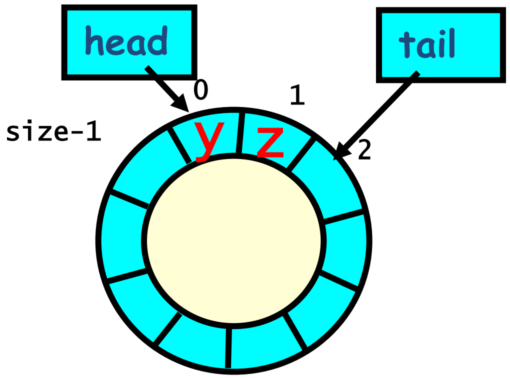
Can we increase the concurrency?



Simple Concurrent Queue

- Only 2 threads
  - One thread **enq only**
  - One thread **deq only**
  - Similar to producer/consumer
- Performance
  - Two threads should make progress in parallel
    - Implementation with locks does not allow this!

Simple Concurrent Queue




```
public class SimpleConcurrentQueue
  int head = 0, tail = 0;
  int size = 1<<16;
  Object[] items = new Object[size];

  public void enqueue(Object i)
    while (tail-head == size); // busy-wait
    items[tail % size] = i; tail++;

  public Object dequeue()
    while (tail == head); // busy-wait
    Item i = items[head % size];
    head++;
    return i;
```

Intuition

- Each operation has a single point in which it takes effect
  - **enq**: `tail++`
  - **deq**: `head++`
- From the point of view of the concurrent object, correct behavior is "sequential"
- Any such object is **linearizable**!

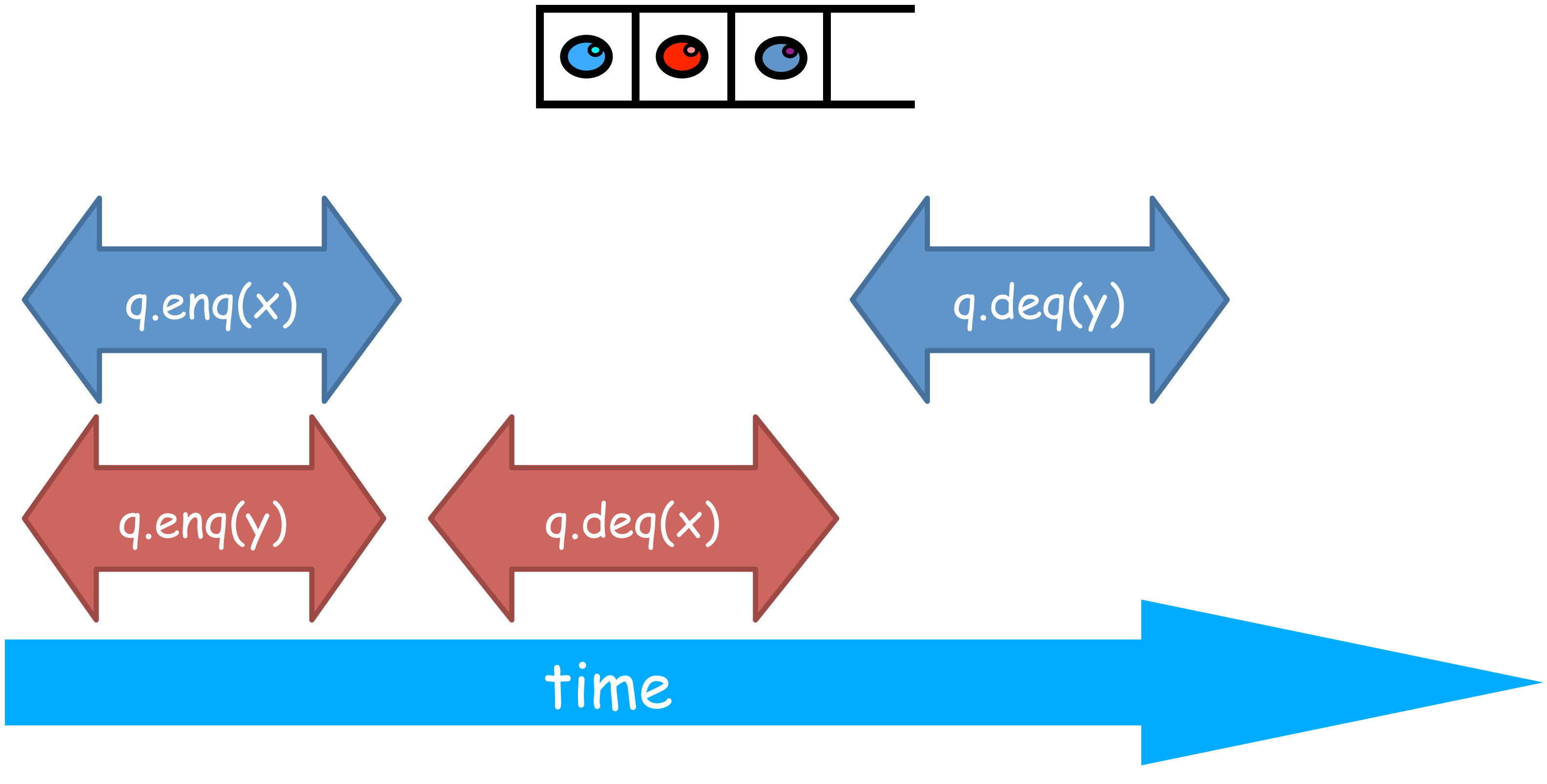
Linearizability

- Each method
  - Takes effect instantaneously at a **linearization point**
  - Between invocation and return
- Object is correct if this “sequential” behavior is correct
  - The sequential behavior is actually a property of the execution

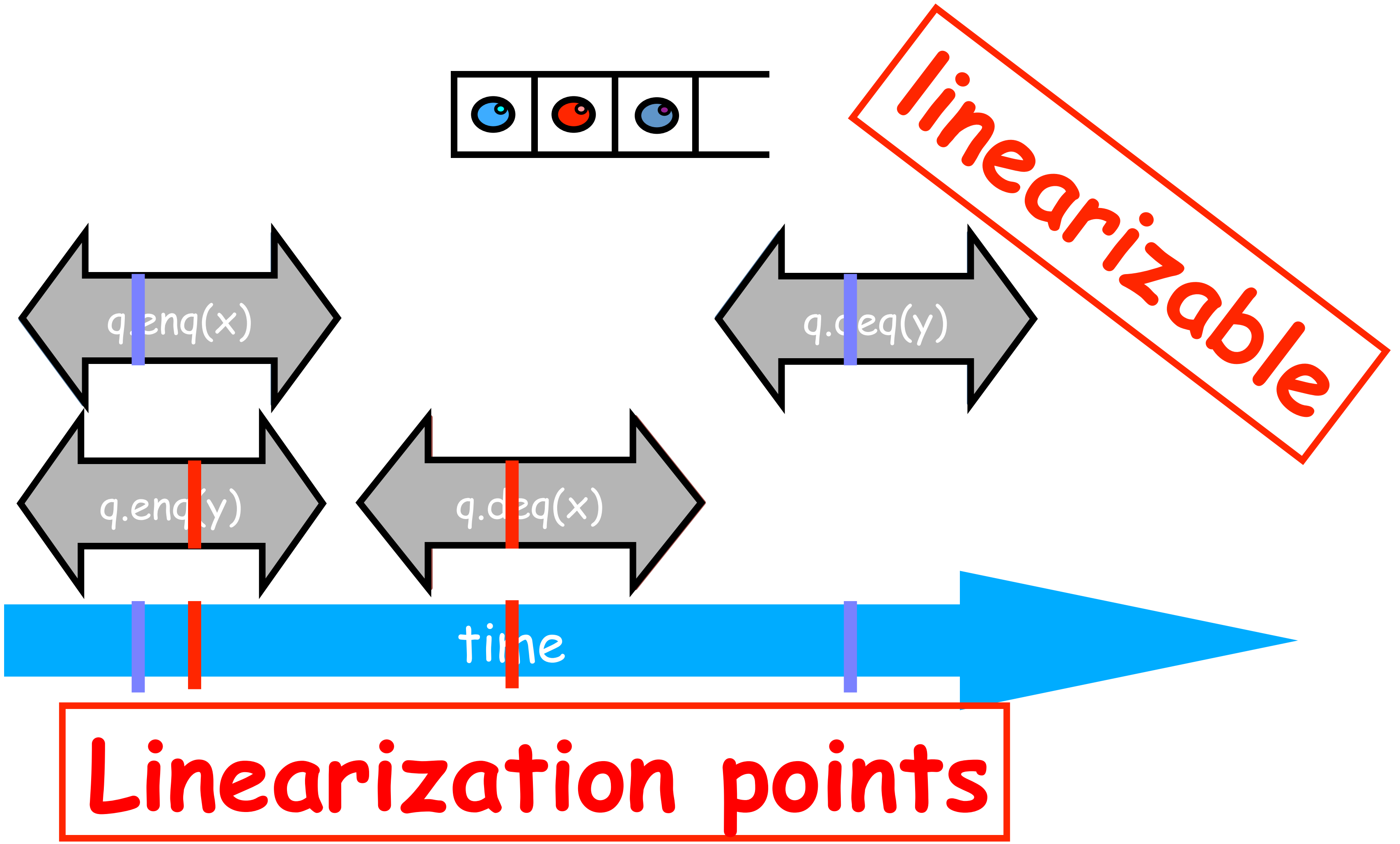
Linearizability

Object is linearizable if all of the possible executions are linearizable

Example 1

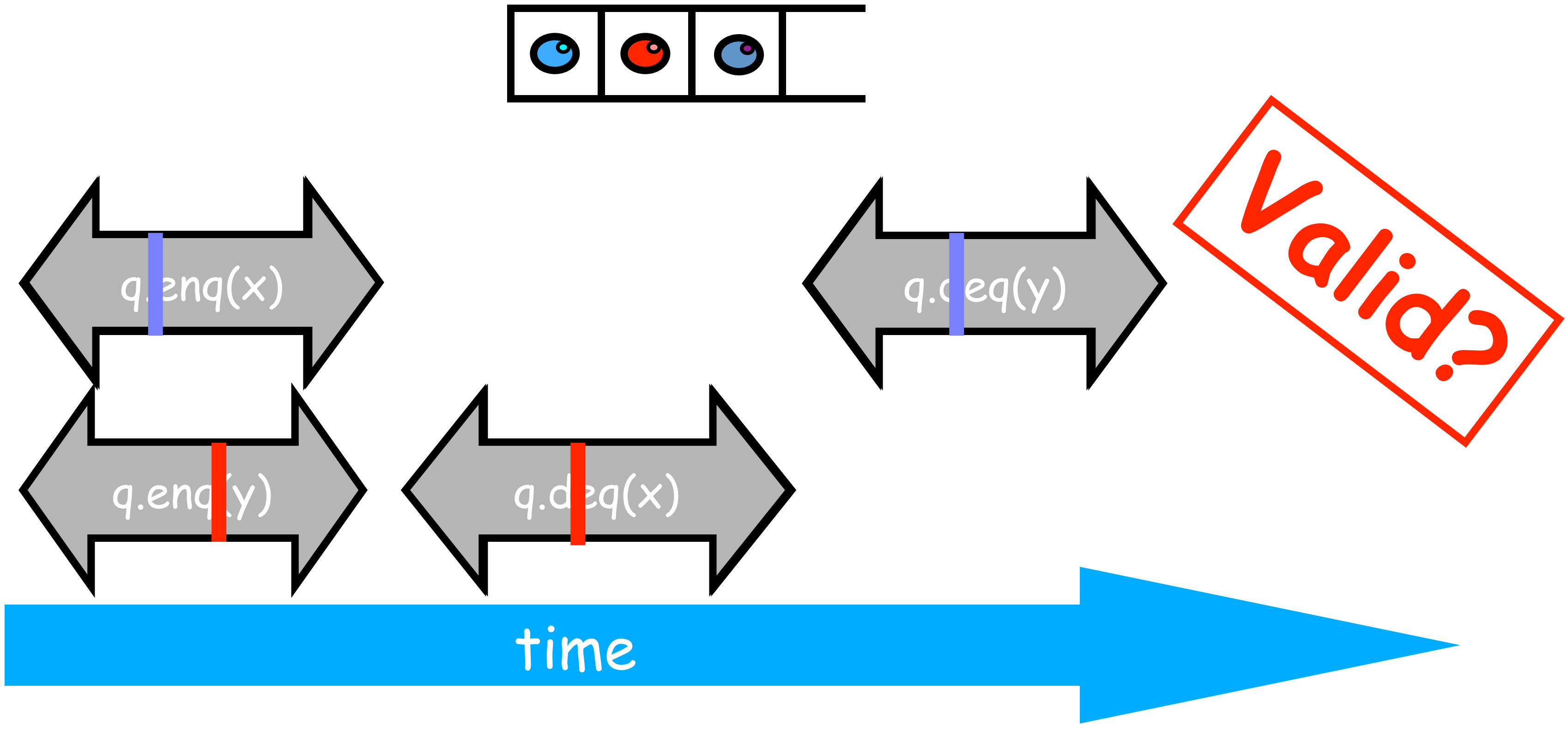


Example 1

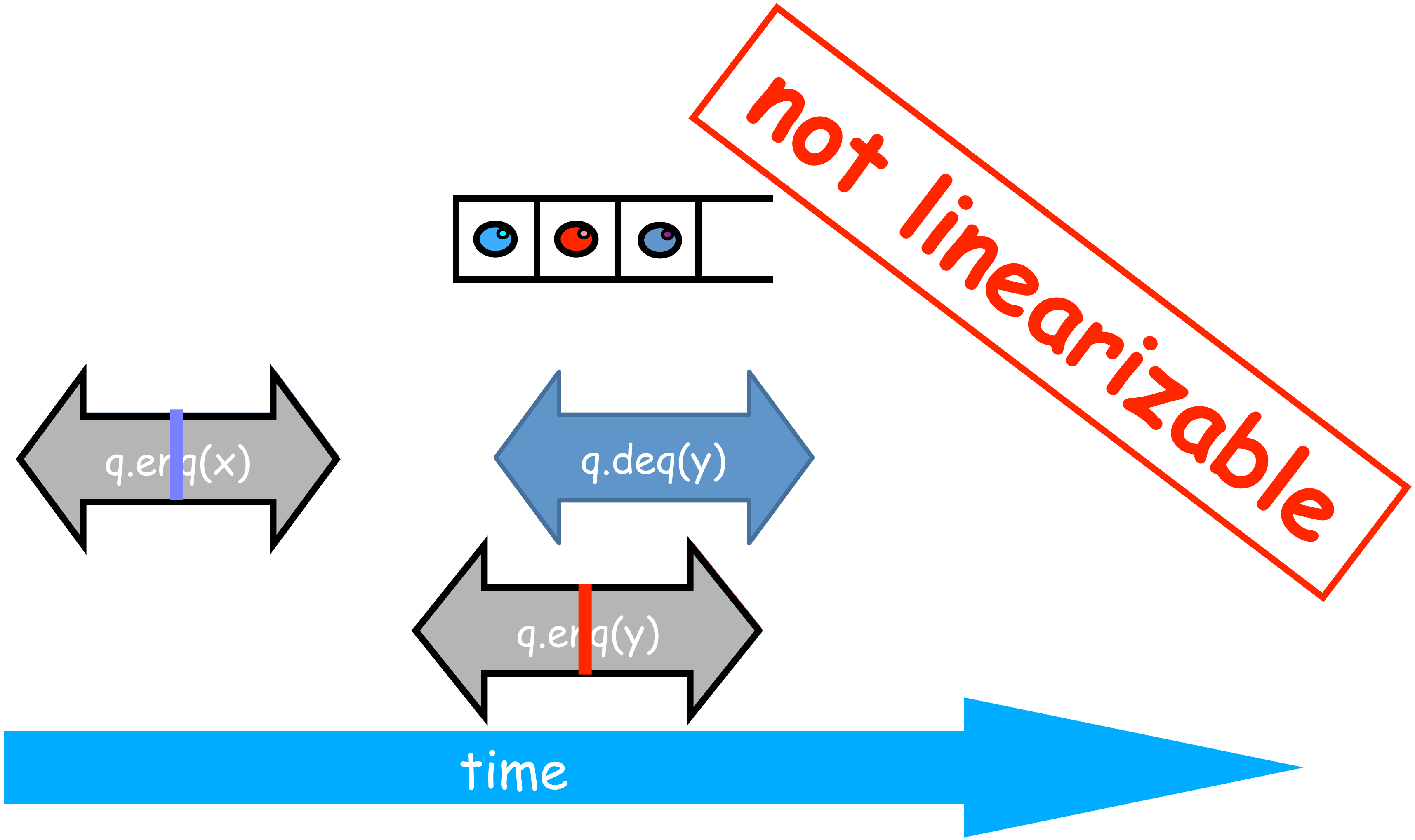


linearizable
q.enq(x)
q.deq(y)
q.enq(y)
q.deq(x)
time
Linearization points

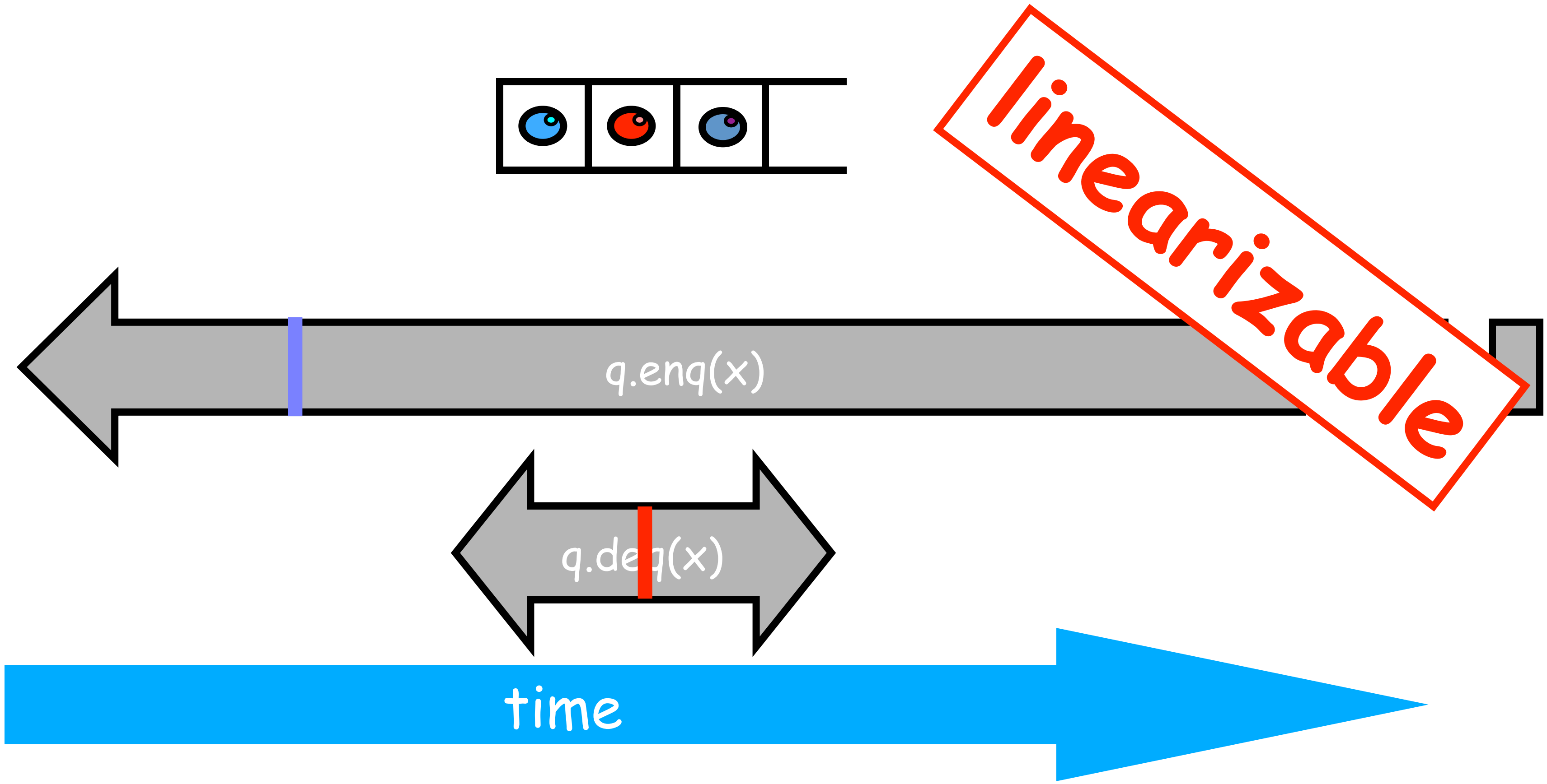
Example 1



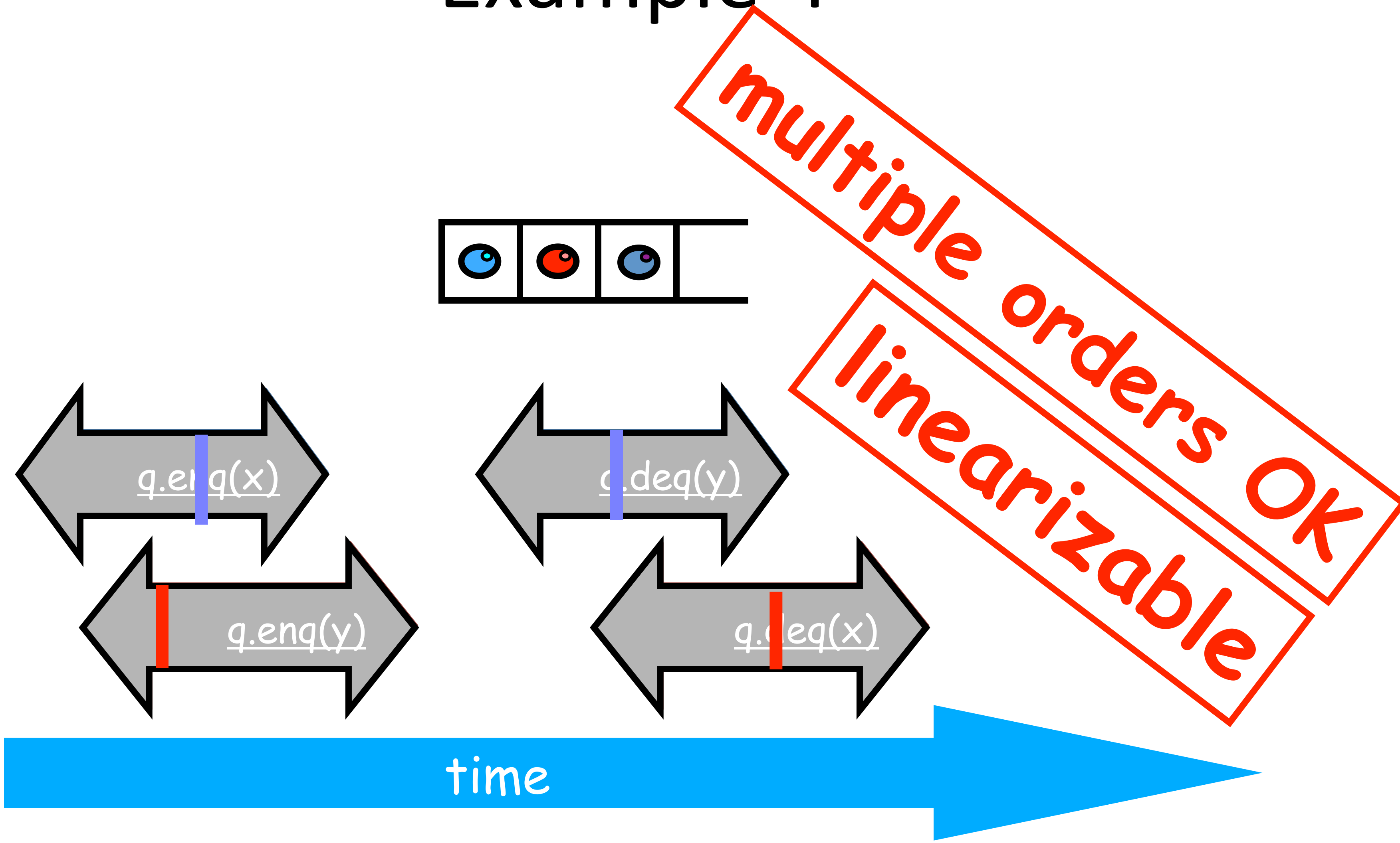
Example 2



Example 3



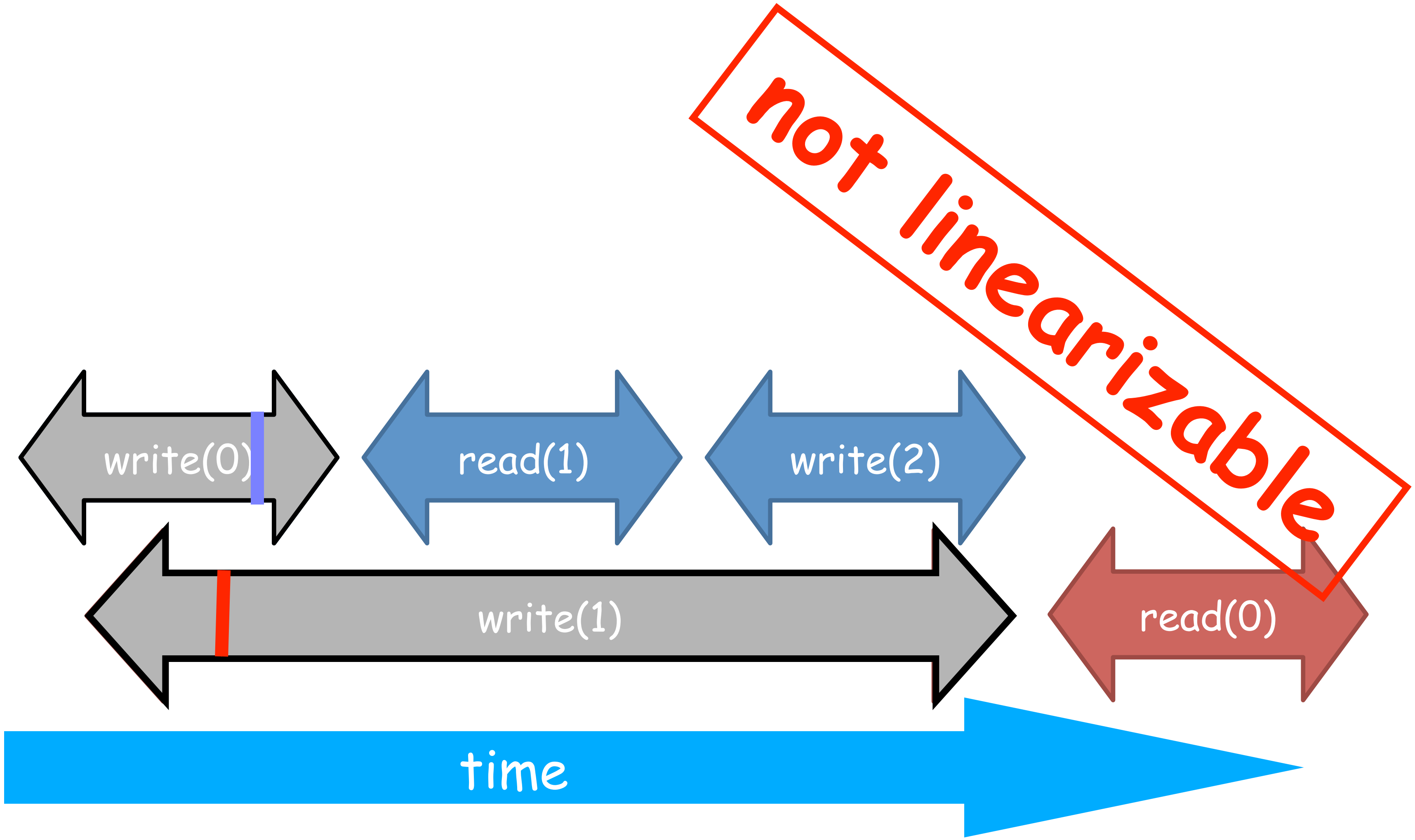
Example 4


multiple orders OK
linearizable
q.enq(x)
q.enq(y)
q.deq(y)
q.deq(x)
time

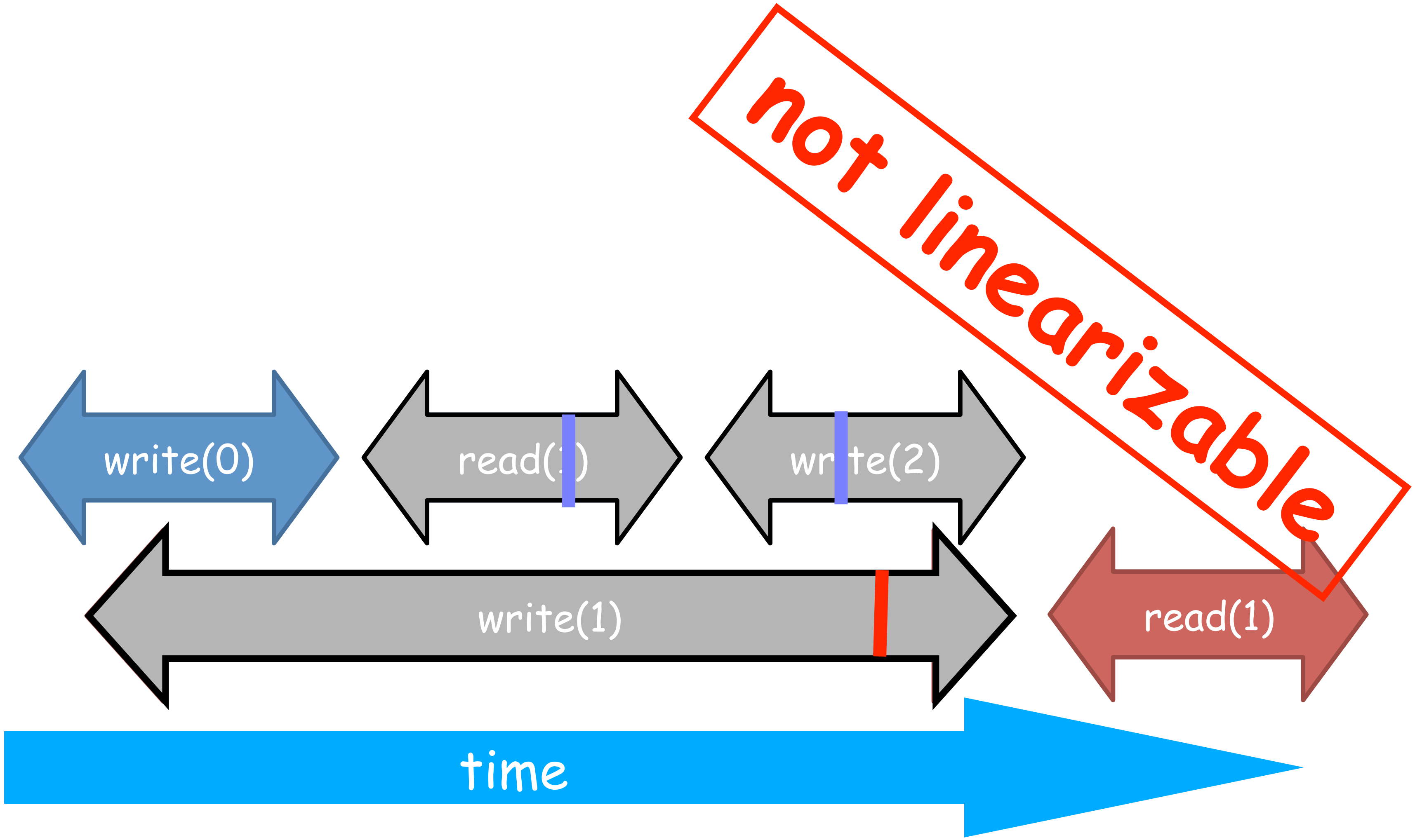
Concurrent FIFO Queue

- FIFO Queue
  - **Strict** temporal order
- Concurrent Queue
  - **Ambiguous** temporal order

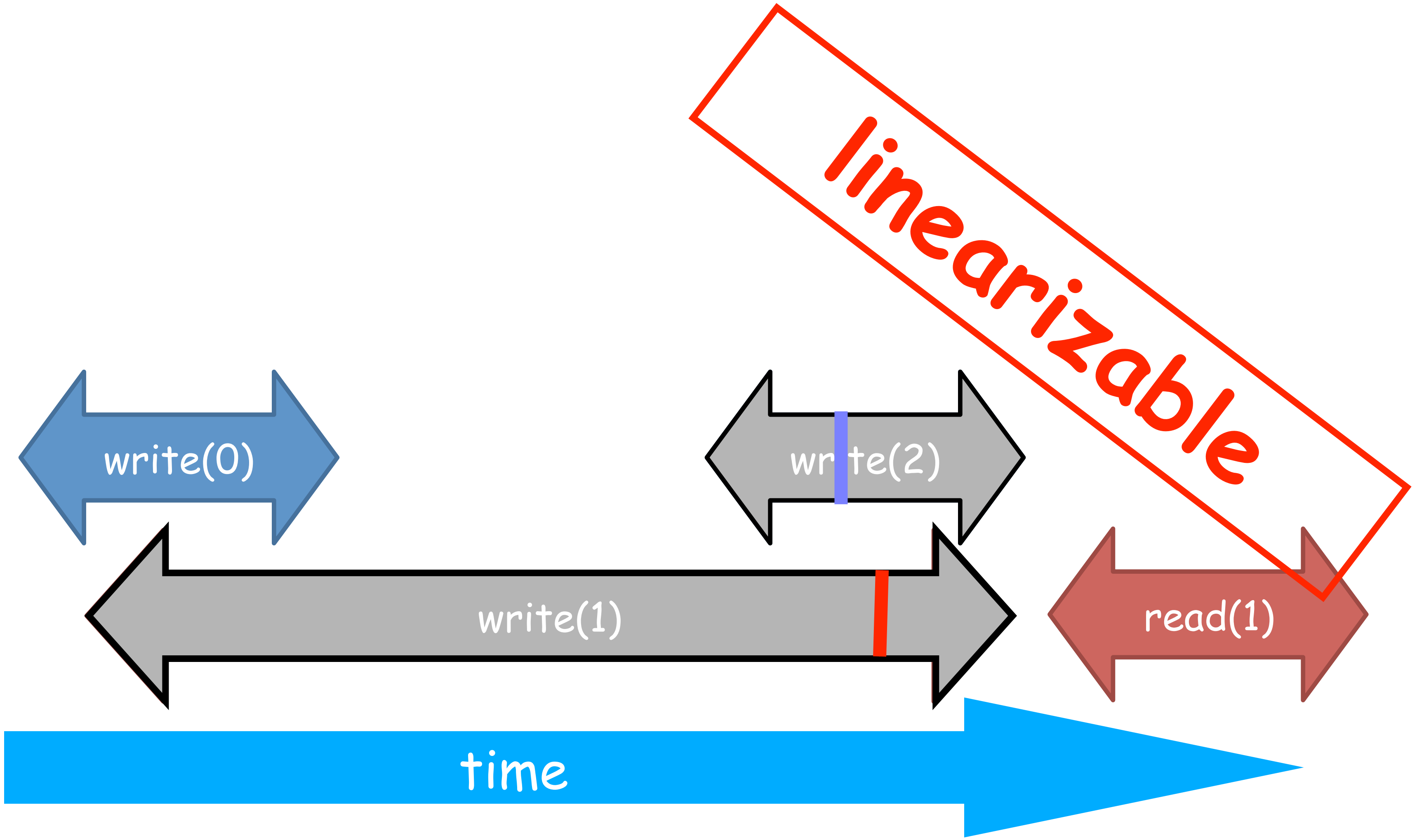
Read/Write Register Example



Read/Write Register Example



Read/Write Register Example



Reason about executions, not operations

- Can't we specify the linearization point of each operation without describing an execution?
- Not always
  - Linearization point might depend on the execution
    - E.g. queue throws error when deq on empty
      - Linearization point for normal case: **head++**
      - Linearization point for empty case: **head == tail**

Linearizability vs Atomicity

- **Atomicity** states that a method either takes effect all at once or it does not take effect at all
- What about visible intermediate inconsistent values?
  - **Isolation**

Linearizability vs Atomicity

"Operations A and B are **atomic** with respect to each other if, from the perspective of a thread executing A, when another thread executes B, either all of B has executed or none of it has. An **atomic operation** is one that is atomic with respect to all operations, including itself, that operate on the same state."

Java Concurrent in Practice, Goetz et al.

Linearizability vs Atomicity

"**Linearizability** defined **atomicity of individual objects** by requiring that each method call of a given object appear to take effect instantaneously between its invocation and response, **Serializability**, on the other hand, defines **atomicity for entire transactions**, that is, blocks of code that may include calls to multiple objects. It ensures that a transaction appears to take effect between the invocation of its first call and the response to its last call."

The Art of Multiprocessor Programming, Shavit and Herlihy

Atomicity

- Broad concept that must be carefully defined for each domain
- Concurrent objects: **Linearizability**
- Transactions: **Serializability**
- Hardware: **Atomic Instructions**

Formal Model of Executions

Define precisely what we mean

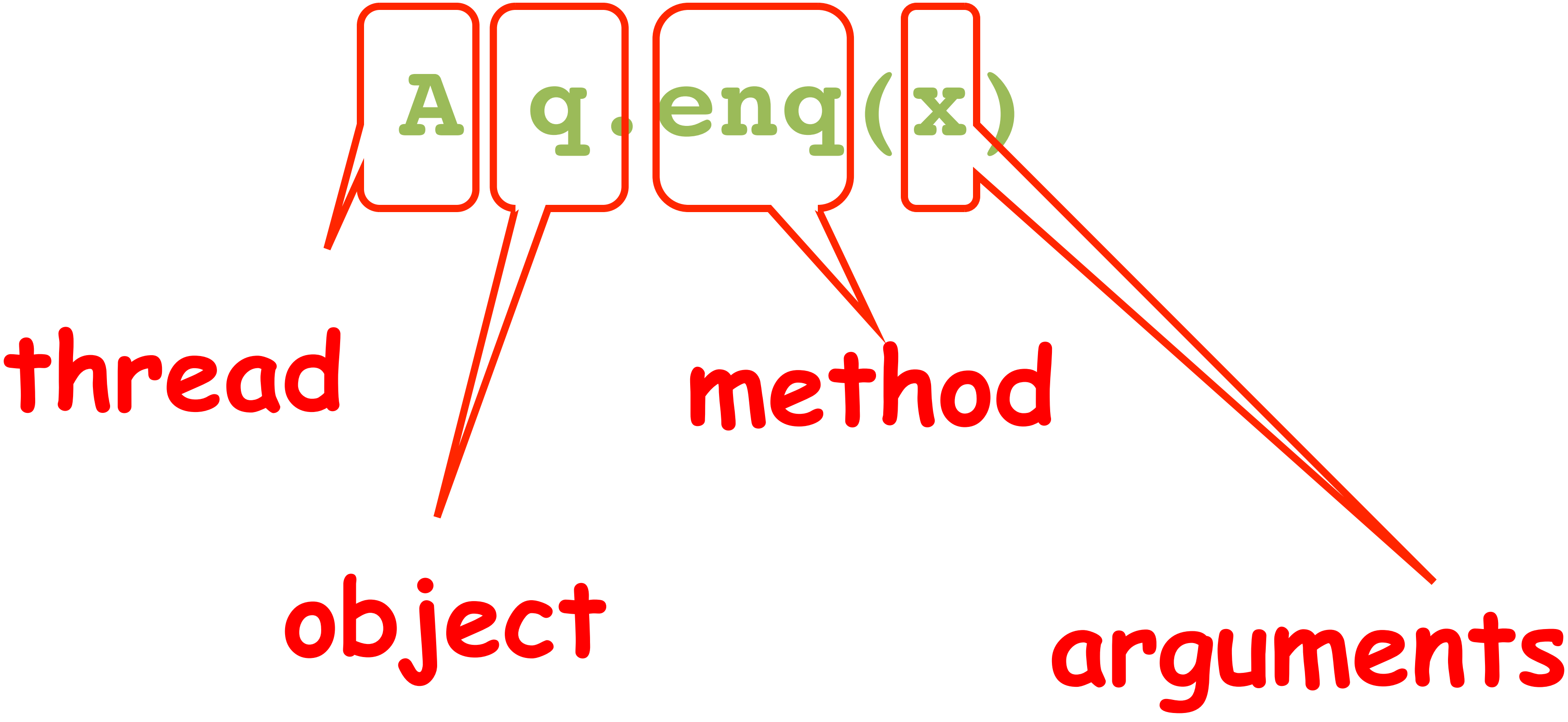
Formal Model

- Notation
- Histories
- Projections
  - Object/Thread
- Complete history
  - Matching/Pending invocations
- Sequential history
- Well-formed history
- Equivalent history
- Precedence/Concurrent calls
  - Partial order

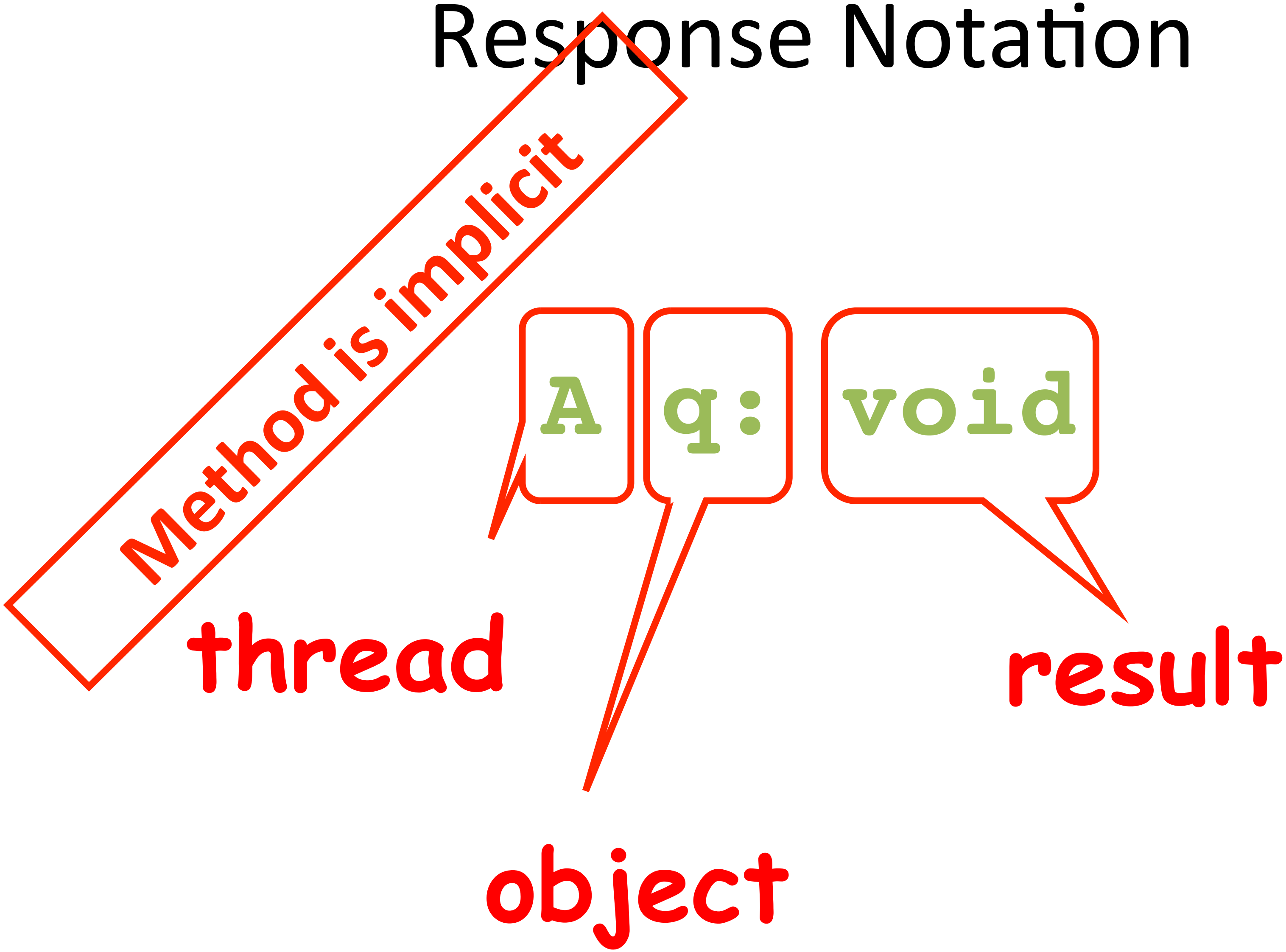
Method call events

- Split method calls into two events
  - Invocation
    - Object, method name and arguments
    - `q.enq(x)`
  - Response
    - Object, result or exception
    - `q.enq(x)` returns `void`
    - `q.deq()` returns x
    - `q.deq()` throws EmptyException

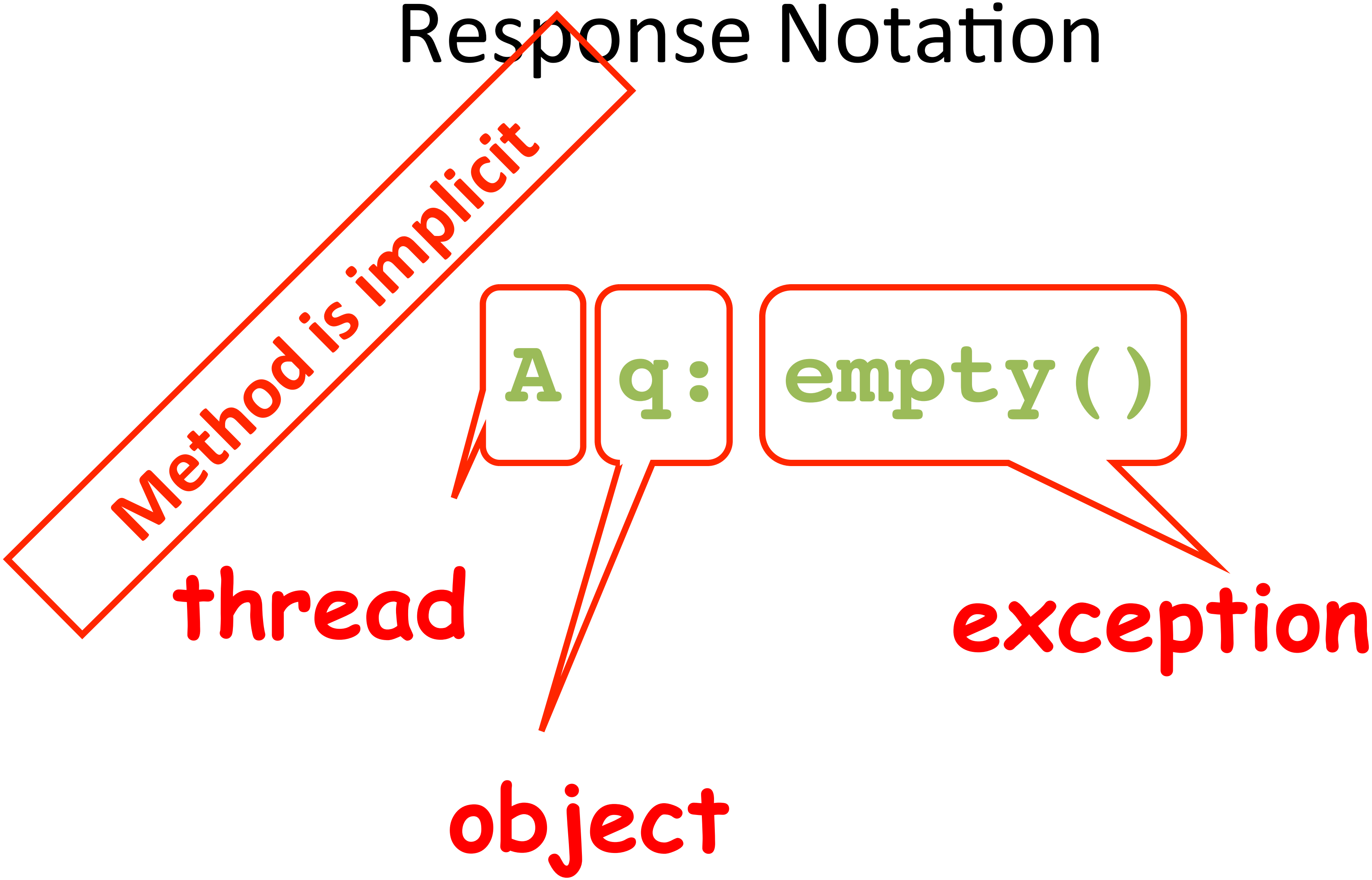
Invocation Notation



Response Notation



Response Notation



Histories

- Describe executions with a sequence of invocations and responses

$$H = \left\{ \begin{array}{l} \texttt{A q.enq(3)} \\ \texttt{A q:void} \\ \texttt{A q.enq(5)} \\ \texttt{B p.enq(4)} \\ \texttt{B p:void} \\ \texttt{B q.deq()} \\ \texttt{B q:3} \end{array} \right.$$

Matching invocation and response

Thread names agree

Object names agree

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

Same method

Object Projections

$H =$

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

q

$H|q =$

Thread Projections

H =

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

B

H|B =

Complete Subhistory

Complete(H) =

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

Pending

- May have taken effect
- Or not

Sequential History

```
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? OK!

Well-Formed Histories

H =

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

Well Formed

H | B =

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

Sequential

H | A =

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

Sequential

Equivalent Histories

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
```

$H|A = G|A$

$H|B = G|B$

H is equivalent to G

Legal Histories

- A **sequential specification** defines if a single-threaded, single-object history is **correct**
  - Example: Pre and post-conditions

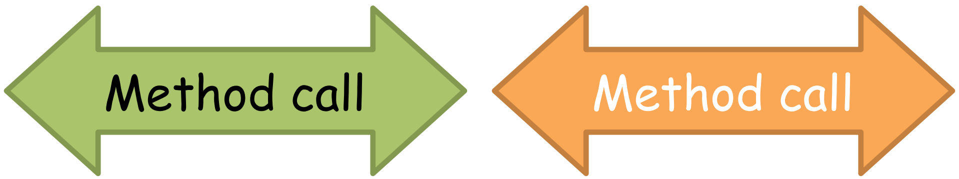
- A history **H** is **legal** if
  - For every object **x** in the history
  - **H|x** is in the sequential spec for **x**

Precedence

H =

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

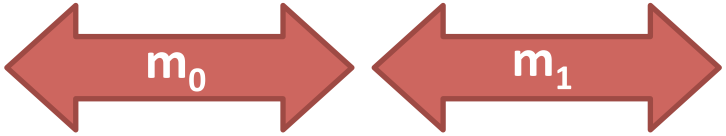
A method call **precedes** another if response event precedes invocation event



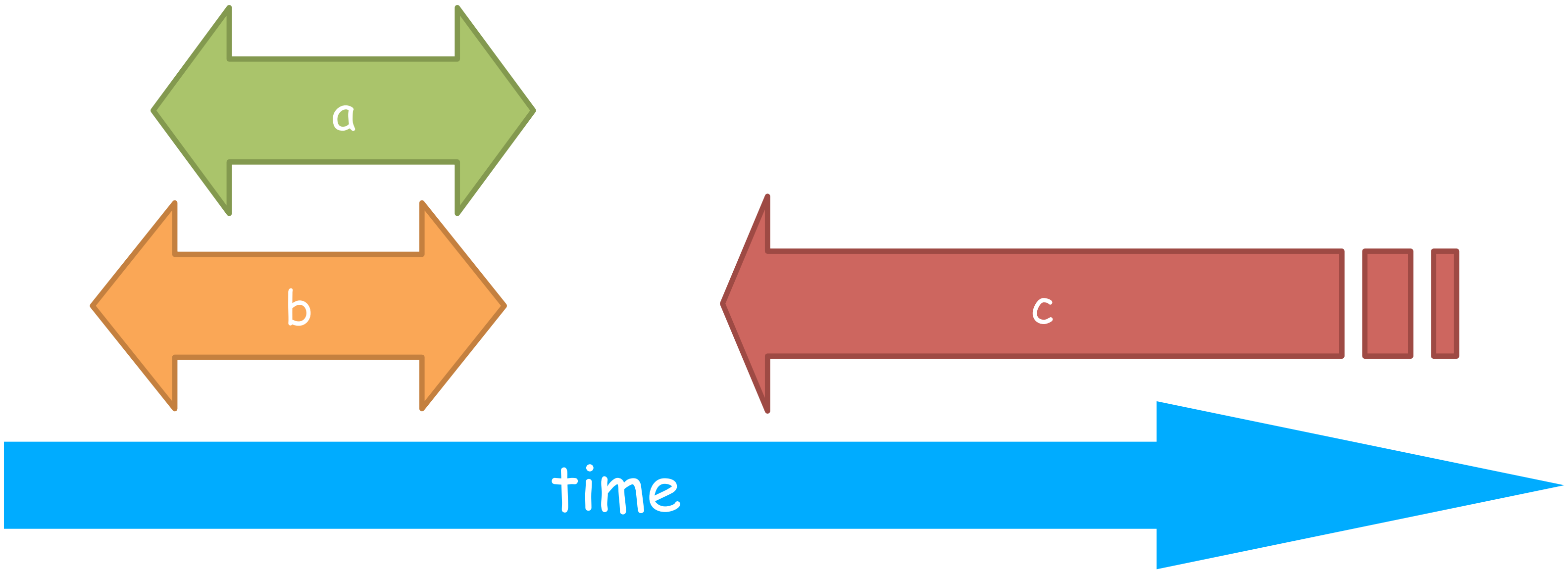
Precedence on executions

- Given
  - History **H**
  - Method executions $\mathbf{m_0}$ and $\mathbf{m_1}$ in **H**

- It is true that $\mathbf{m_0} \rightarrow \mathbf{m_1}$ if
  - $\mathbf{m_0}$ precedes $\mathbf{m_1}$



Precedence on concurrent executions defines a partial order



- a → c
- b → c
- What about a → b? Or b → a?

Linearizability

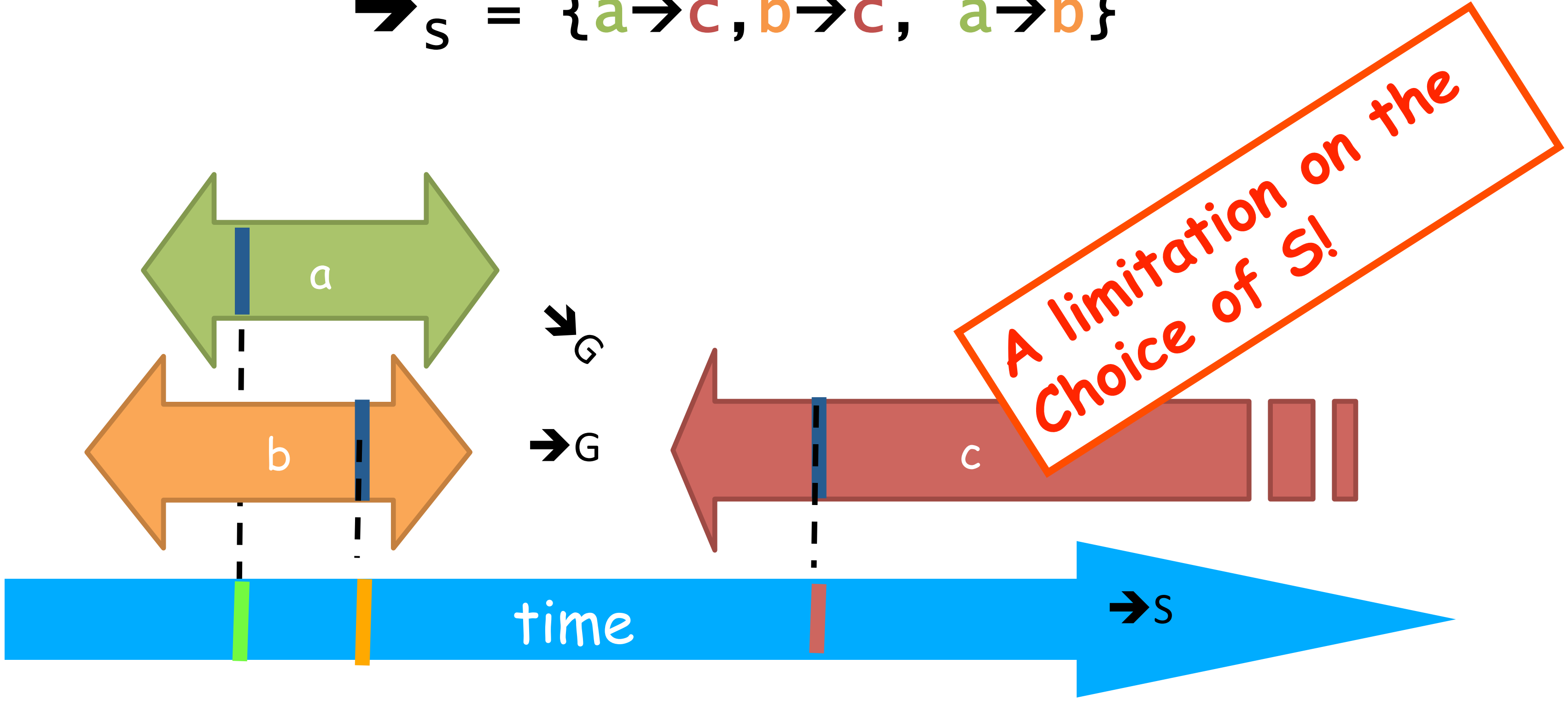
- History **H** is **linearizable** if it can be exte… d to **G** by
  - Appending zer… …ses to pending invo…
  - …ding all the other pending invocations
- So that **G** is equivalent to
  - Legal sequential hi…
  - Such th… …g

Deal with pending invocations

Remove ambiguity

What is $\rightarrow_G \subset \rightarrow_S$?

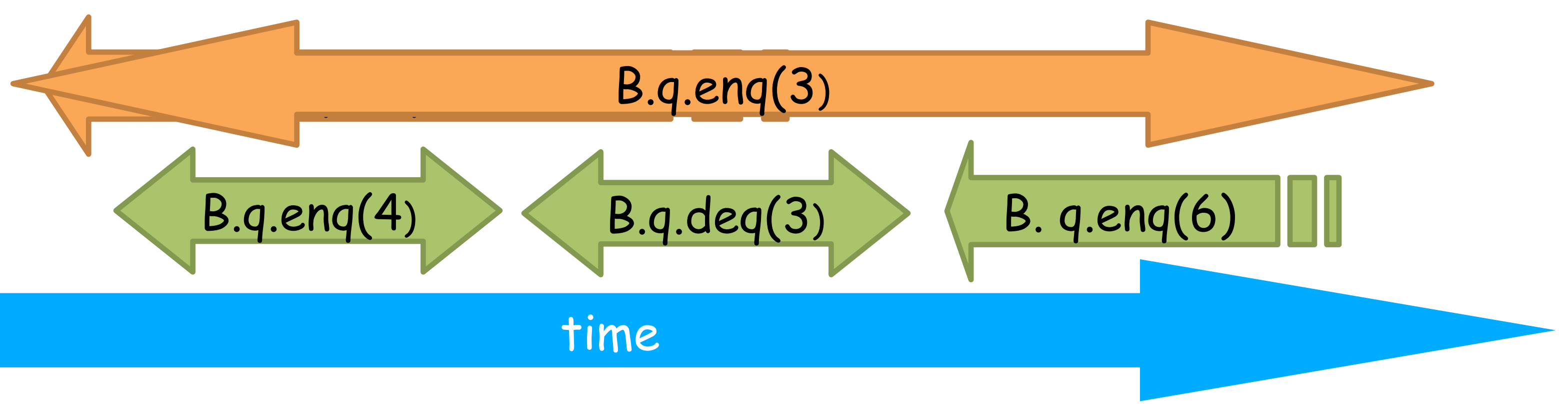
$\rightarrow_G$ = {a→c, b→c}
$\rightarrow_S$ = {a→c, b→c, a→b}



Example

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

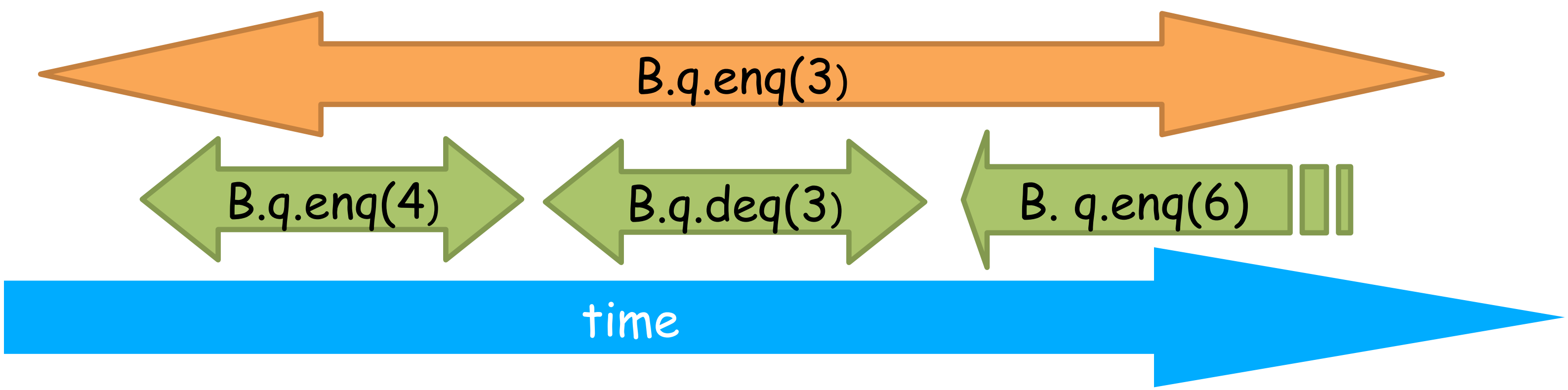
Complete pending invocation!



Example

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

Discard



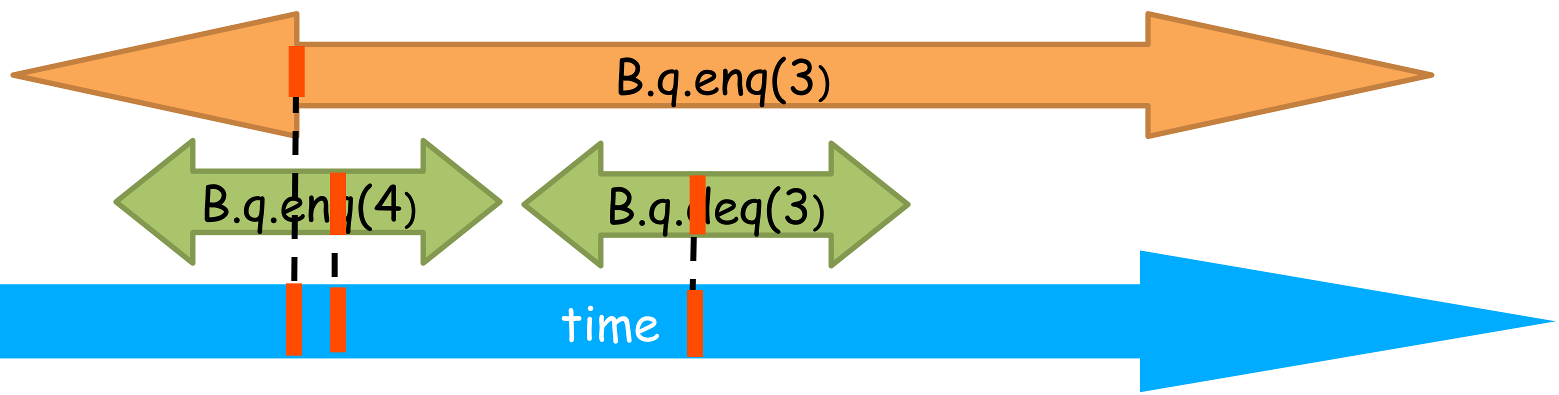
Example

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

A q:void
```

Equivalent Sequential History

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



Linearizability is **composable**

- History **H** is linearizable iff
  - For every object **x**
  - **H|x** is linearizable

- Only using **local reasoning** about each object in isolation to assert linearizability as a **global property**
  - Therefore, composable

Implementing linearizability

- Identify atomic steps in your algorithm where method “happens”
  - Critical section
  - Machine instruction
- Methods might have more than one linearization point
  - Depending on the control flow

Linearizability

- Powerful specification for shared objects
- Captures the notion of objects being atomic
- Don't leave home without it!

What about the hardware?

- **Linearizability** is way too strong for the hardware to implement efficiently

- Next: **Sequential Consistency**
  - Relax linearizability
  - Still too strong to allow efficient hardware

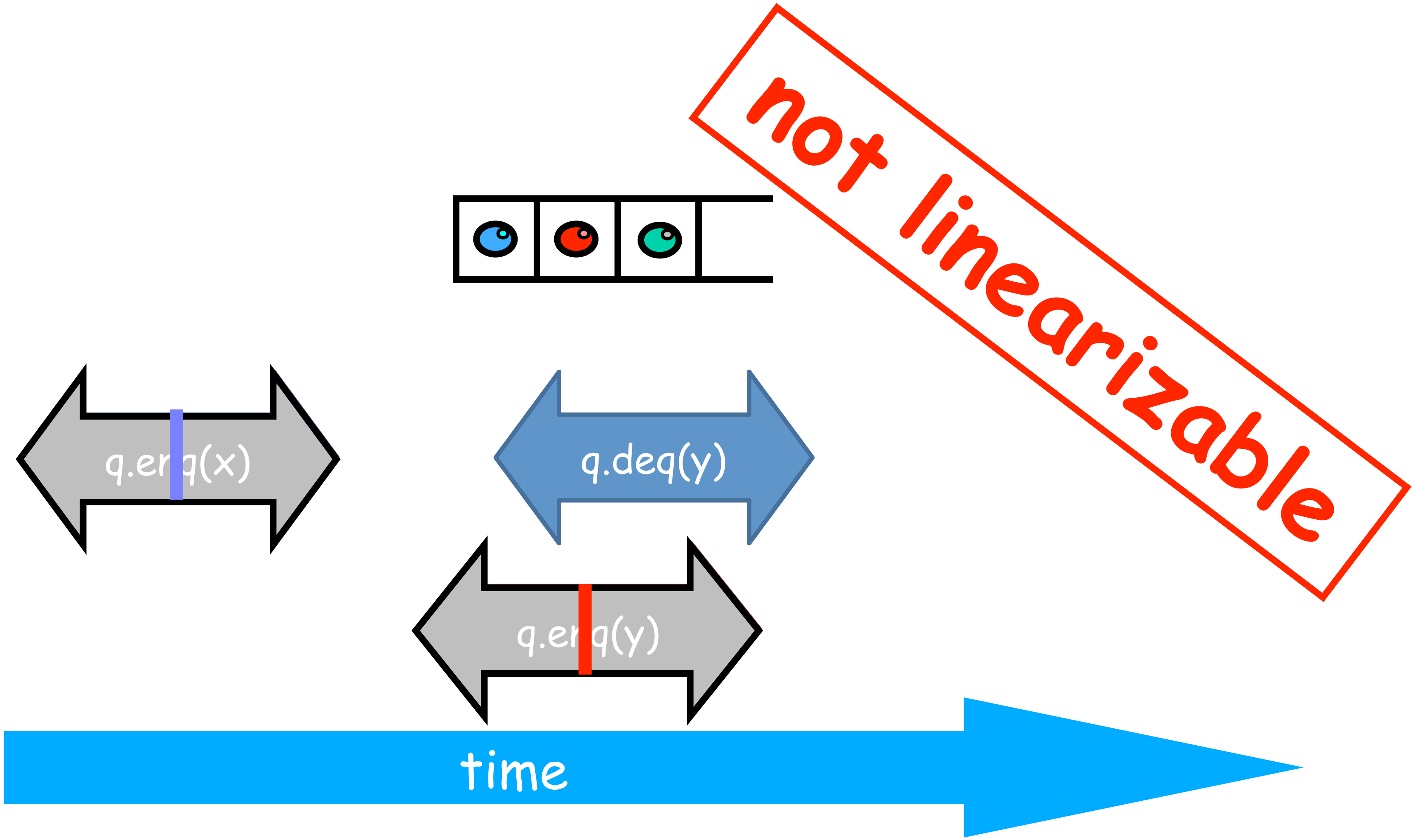
- Hardware uses **happens-before relationship**
  - Sequential consistency just when you need it
  - Fast/"inconsistent" otherwise

Sequential Consistency

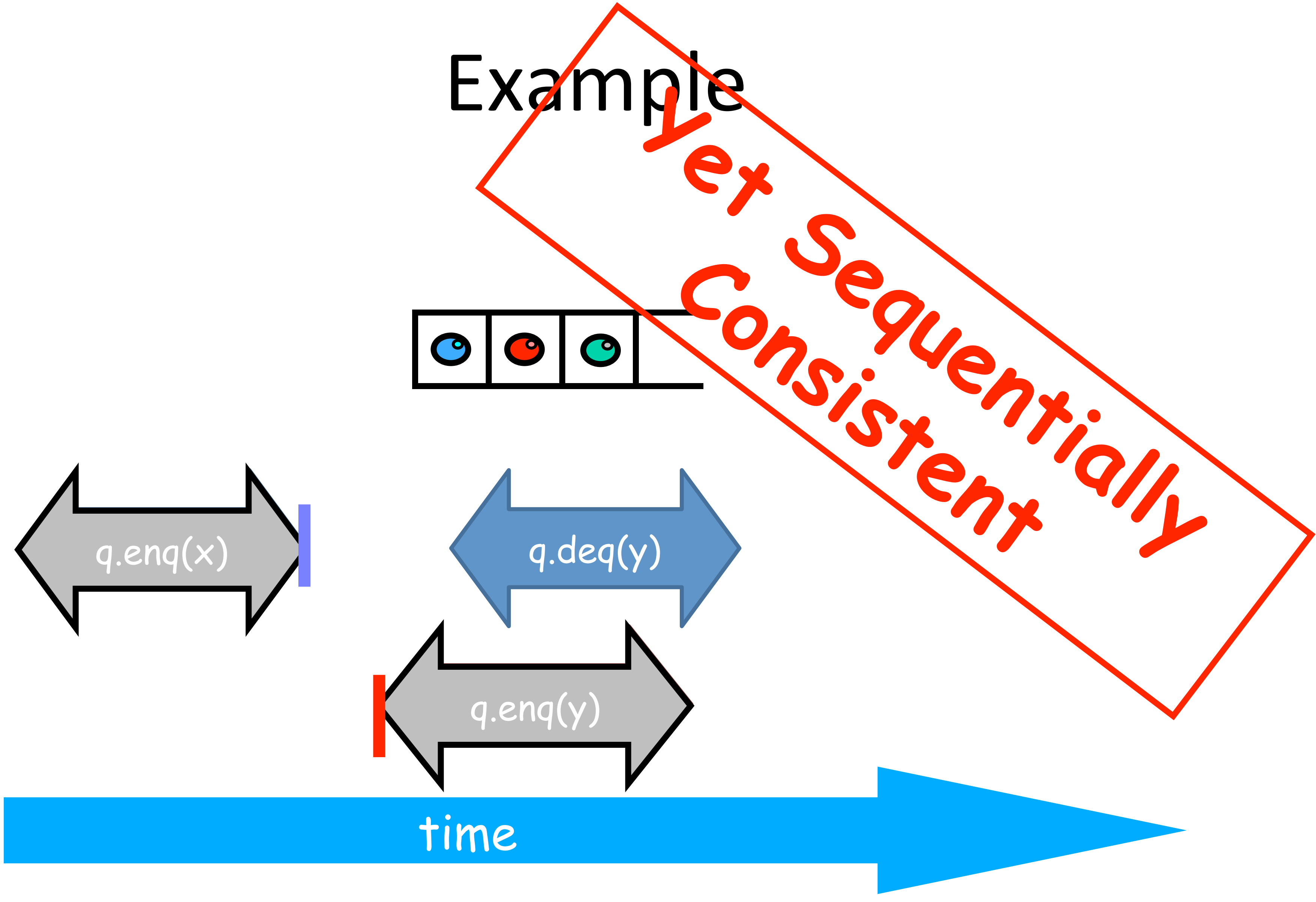
- History **H** is **sequentially consistent** if it can be extended to **G** by
  - Appending zero or more responses to pending invocations
  - Discarding all the other pending invocations
- So that **G** is equivalent to
  - Legal sequential history **S**
  - ~~Such that $\rightarrow_G \subset \rightarrow_S$~~

Differs from linearizability

Example



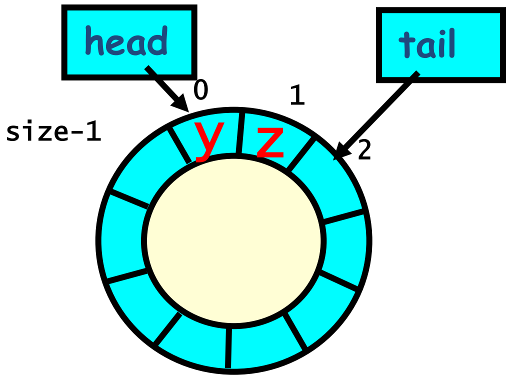
Example



Sequential Consistency

- More relaxed than **linearizability**
- What did we lose?
  - **Composability**
- Why?
  - Sequential Consistency is not a local property
- Relaxed enough for the hardware?
  - No…

Simple Concurrent Queue

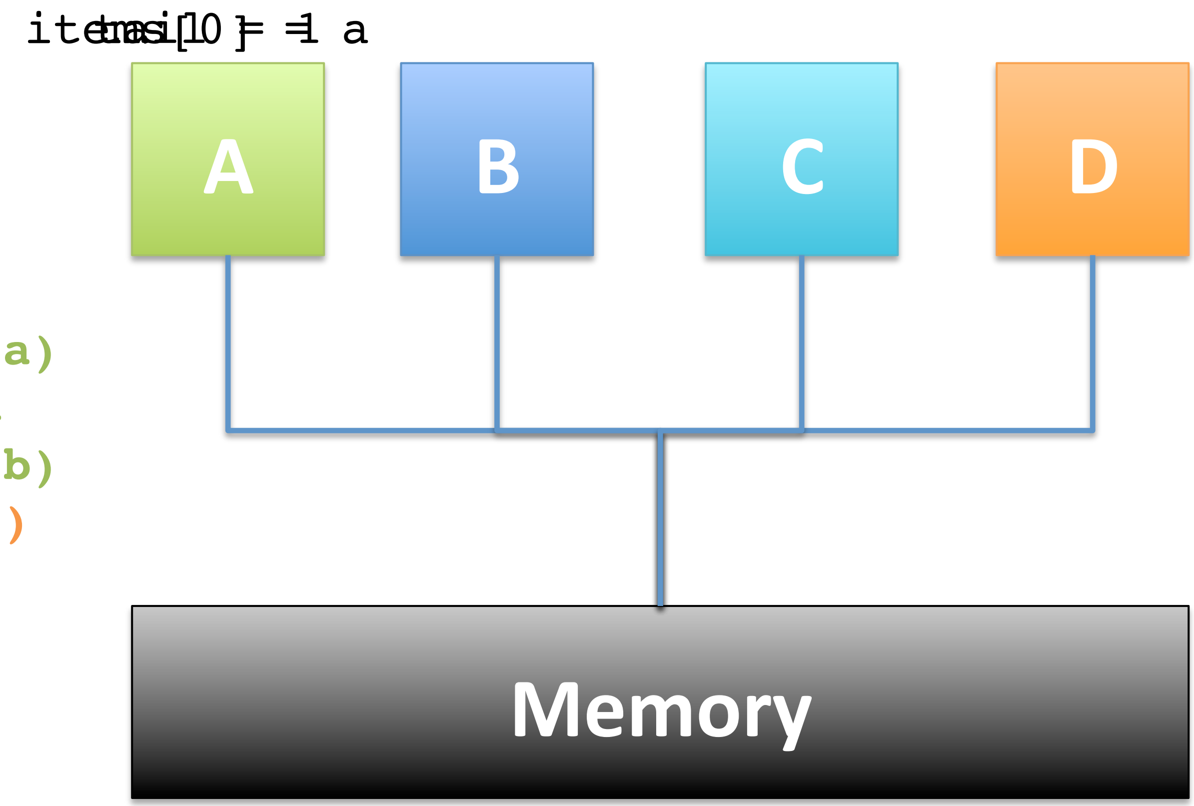



```
public class SimpleConcurrentQueue
  int head = 0, tail = 0;
  int size = 1<<16;
  Object[] items = new Object[size];

  public void enqueue(Object i)
    while (tail-head == size); // busy-wait
    items[tail % size] = i; tail++;

  public Object dequeue()
    while (tail == head); // busy-wait
    Item i = items[head % size];
    head++;
    return i;
```

Ideal hardware

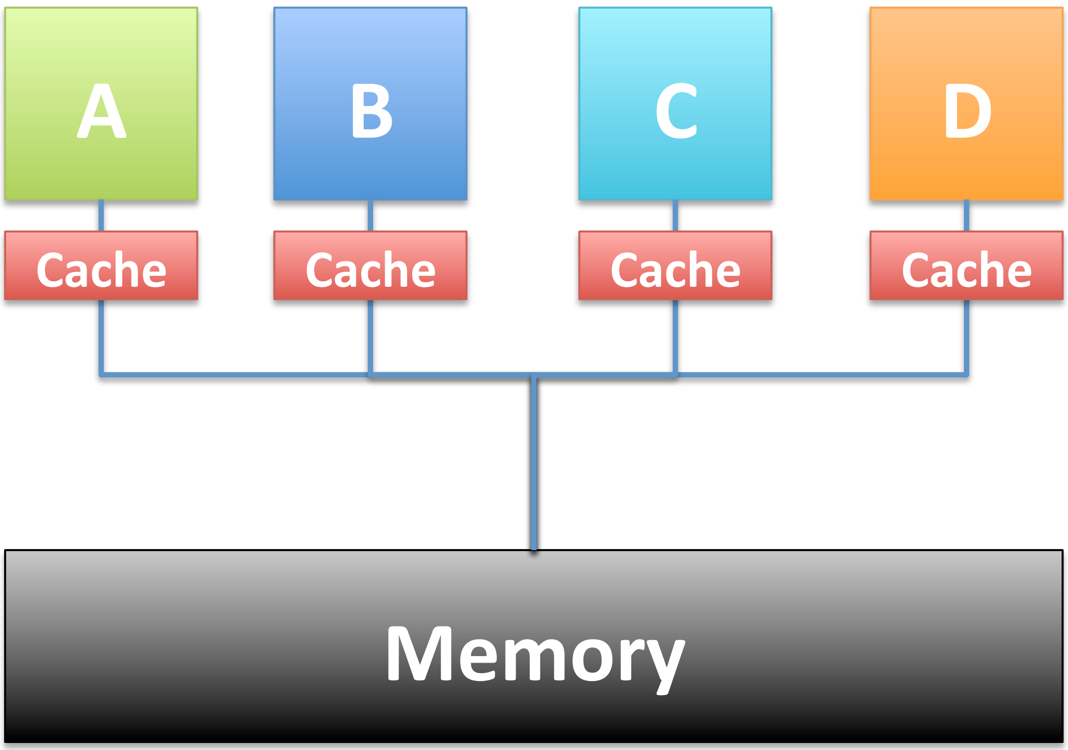


A: enq(a)
A: void
A: enq(b)
D: deq()
D: a

items[0] = a ; head = 0 ; tail = 1

Real hardware

items[0] = a
tail = 1




```
A: enq(a)
A: void
A: enq(b)
D: deq()
D: NULL
```

items[0] = null; head = 0 ; tail = 1

Java memory model

- Does not even provide sequential consistency between threads
  - Strange memory effects
- Require sequential consistency with **happens-before relationship**
  - If memory operation **a** happens-before **b**
  - Any thread that sees **b** must see **a**
  - Thread does not see **b**, may see **a** or not

Happens-before in Java

- Volatile operations
  - A **volatile write** happens-before any **volatile read** that sees the written value
- Locks
  - **Releasing** a lock happens-before **re-acquiring** the same lock
- Thread start
  - Starting a thread happens-before the new thread starts executing

Happens-before in Java

- Concurrent API
  - E.g. java.util.concurrent.ConcurrentLinkedQueue

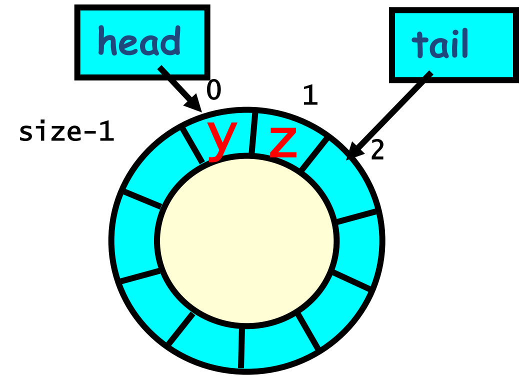
> “Memory consistency effects: As with other concurrent collections, actions in a thread prior to placing an object into a ConcurrentLinkedQueue happen-before actions subsequent to the access or removal of that element from the ConcurrentLinkedQueue in another thread.”

Happens-before in Java

- x86 CAS
  - Native-code
  - sun.misc.Unsafe

- Does not really happen-before anything
  - Implemented with memory fences
  - Just like volatile operations
  - In fact, behaves like happens-before

Fixed Simple Concurrent Queue




```
public class SimpleConcurrentQueue
  volatile int head = 0, tail = 0;
  int size = 1<<16;
  Object[] items = new Object[size];

  public void enqueue(Object i)
    while (tail-head == size); // busy-wait
    items[tail % size] = i; tail++;

  public Object dequeue()
    while (tail == head); // busy-wait
    Item i = items[head % size];
    head++;
    return i;
```