VIA University College

Software Development with UML and Java 2

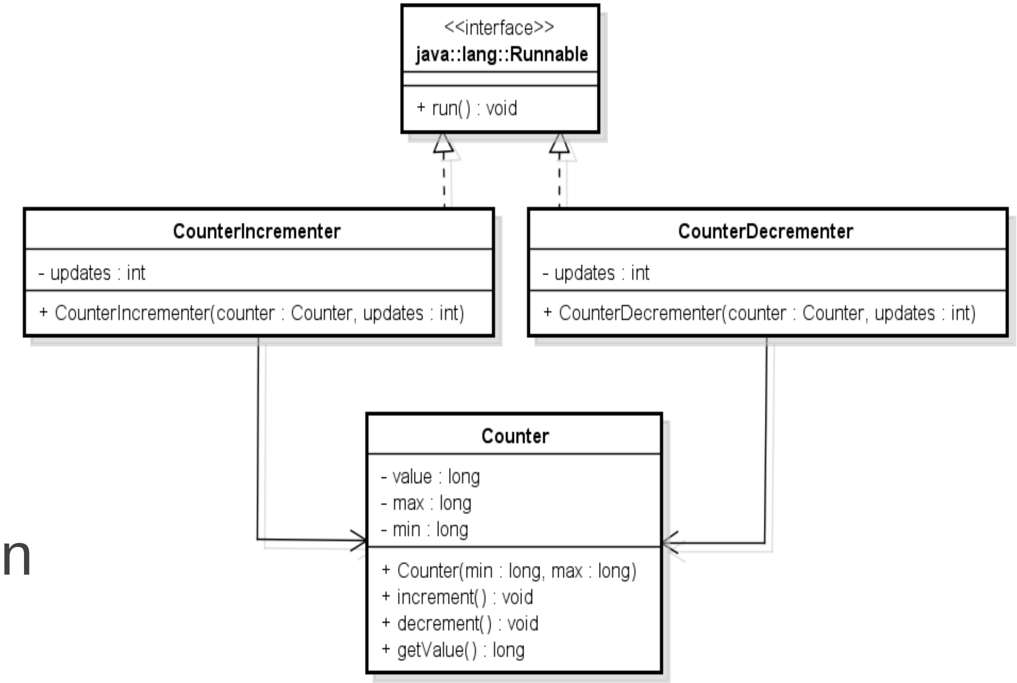
Autumn 2021

Learning Objectives

- By the end of this session, you should be able to:
  - ✓explain the concept – producer-consumer problem
  - ✓ explain blocking queue
  - ✓ Implement monitor in Java

Counter (incrementer/decrementer)

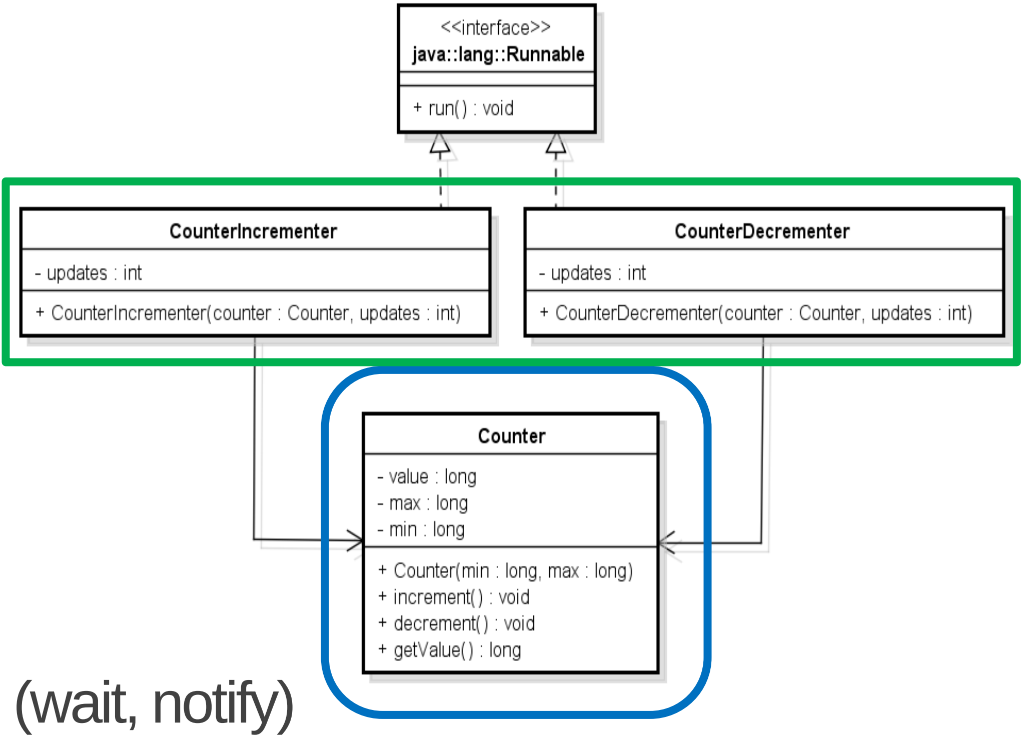
- CounterIncrementer
  - Waits while counter value >= max
  - Increment counter value
  - Notify all
- CounterDecrementer
  - Waits while counter value <= min
  - Decrement counter value
  - Notify all



Producer: CounterIncrementer

Consumer: CounterDecrementer

Observations

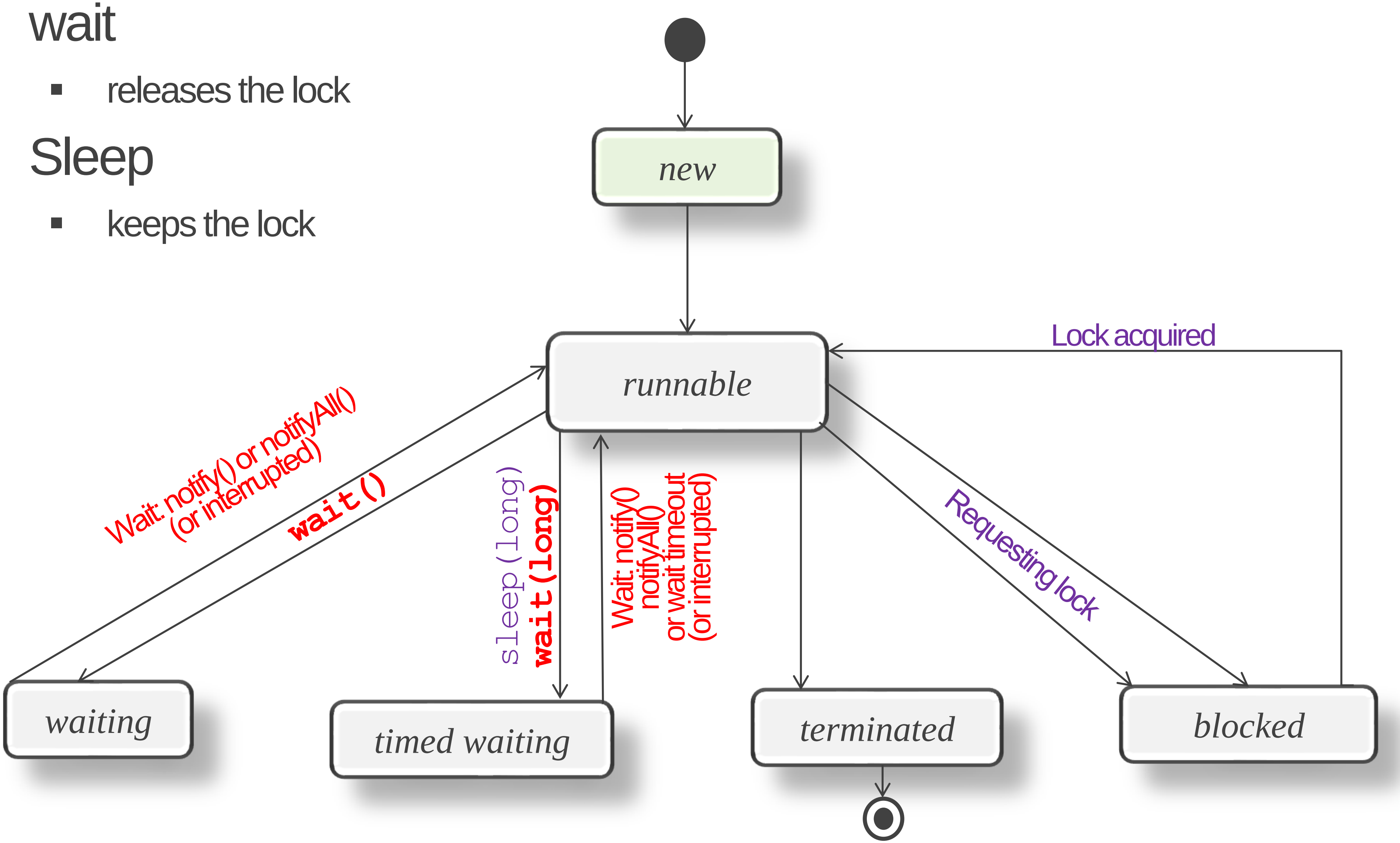


- Monitor class
  - Shared resource
  - Private instance variables
  - Synchronized methods
  - Threads to and from Wait state (wait, notify)
  - NO sleep!

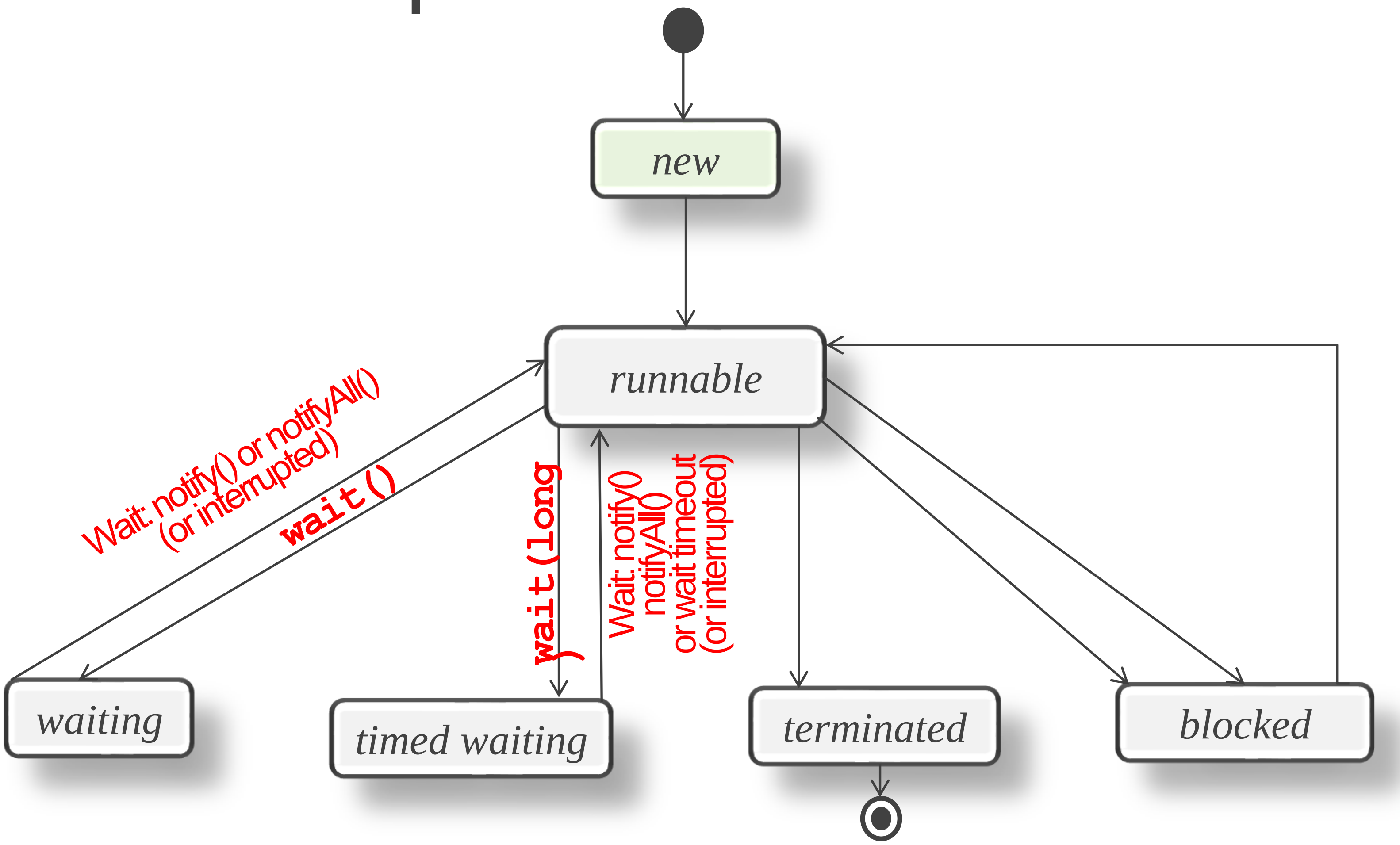
- Thread/runnable classes
  - Gets a reference to the Monitor / shared resource
  - Calling methods in Monitor class
  - Simulate operations taking time (sleep)
  - NO wait/notify!

Thread States – wait vs sleep

- wait
  - releases the lock
- Sleep
  - keeps the lock



How is a thread placed in the wait set?


new
runnable
waiting
timed waiting
terminated
blocked
Wait: notify() or notifyAll() (or interrupted)
wait()
wait(long)
Wait: notify() notifyAll() or wait timeout (or interrupted)

- N/B: thread can be placed in the wait set of an object monitor only if it once acquired the object's monitor lock
- once a thread has acquired the object's monitor lock, it must call the wait() method of the object in order to place itself into the wait set
- must notify the threads waiting in the wait set about the fulfillment of the conditions on which they are waiting (by calling the notify(), notifyAll()) .

Monitor / shared resource

```
class Counter
{
  private long value;
  private long max;
  private long min;

  //...
  public synchronized void increment()
  {
     while (value >= max)
     {
       try
       {
          wait();
       }
       catch (InterruptedException e)
       {
          //...
       }
     }
     value++;
     notifyAll();
   }
```

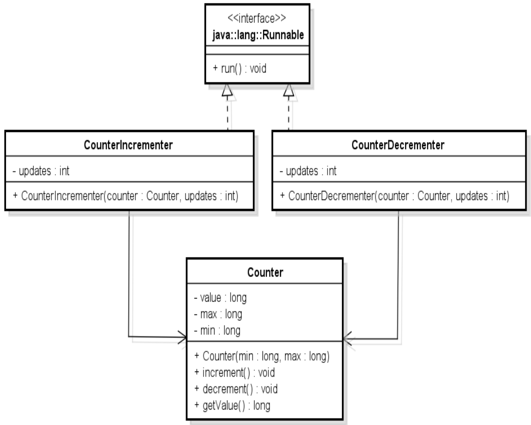
```
  public synchronized void decrement()
  {
     while (value <= min)
     {
       try
       {
          wait();
       }
       catch (InterruptedException e)
       {
          //...
       }
     }
     value--;
     notifyAll();
   }
}
```

Monitor / shared resource

```
class Counter {
  private long value;
  private long max;
  private long min;

  //...
  public synchronized void increment()
  {
     while (value >= max)
     {
       try
       {
          wait();
       }
       catch (InterruptedException e)
       {
          //...
       }
     }
     value++;
     if (value == min+1)
     {
       notify();
     }
   }
```

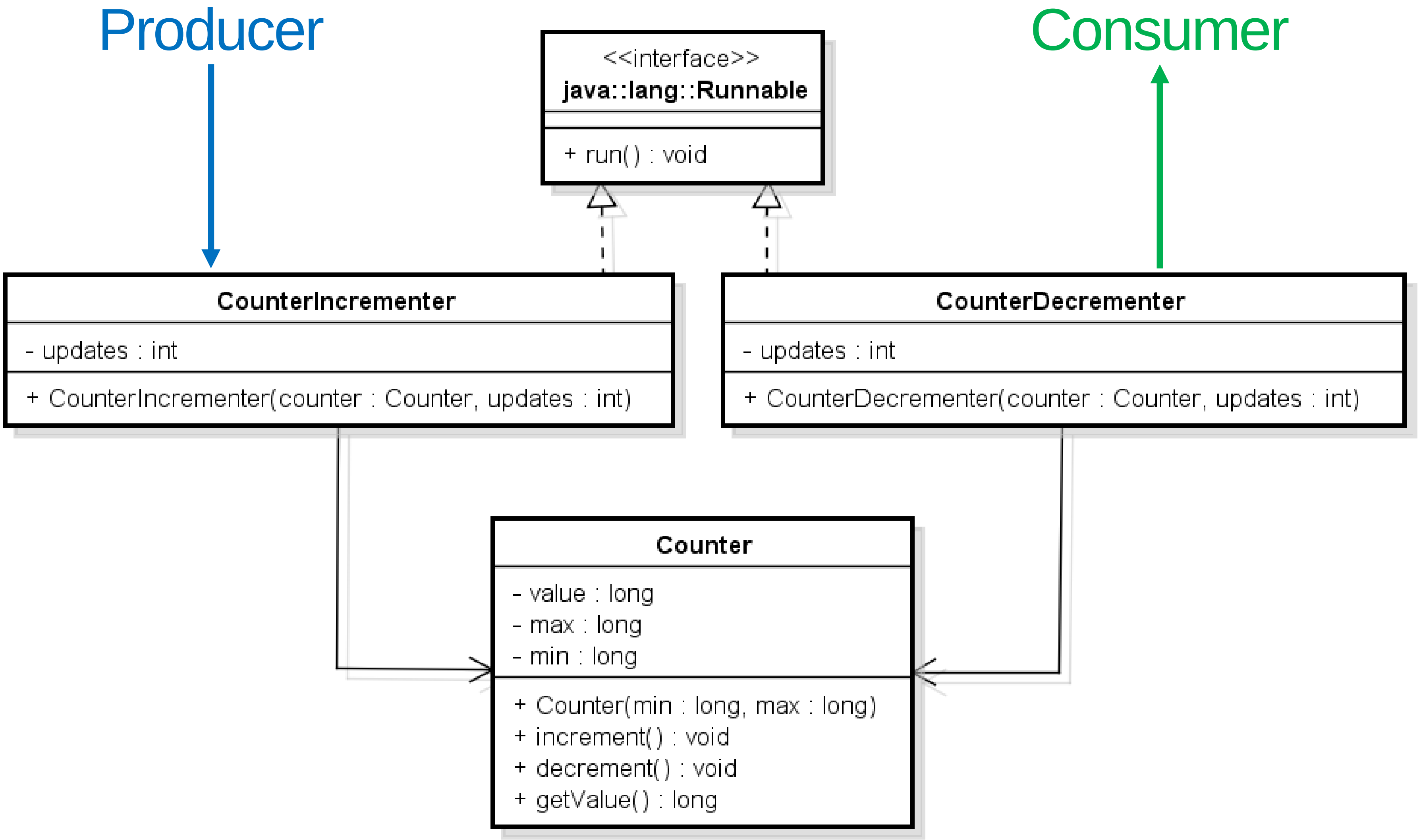
```
  public synchronized void decrement()
  {
     while (value <= min)
     {
       try
       {
          wait();
       }
       catch (InterruptedException e)
       {
          //...
       }
     }
     value--;
     if (value == max-1)
     {
       notify();
     }
   }
}
```

Counter simulator

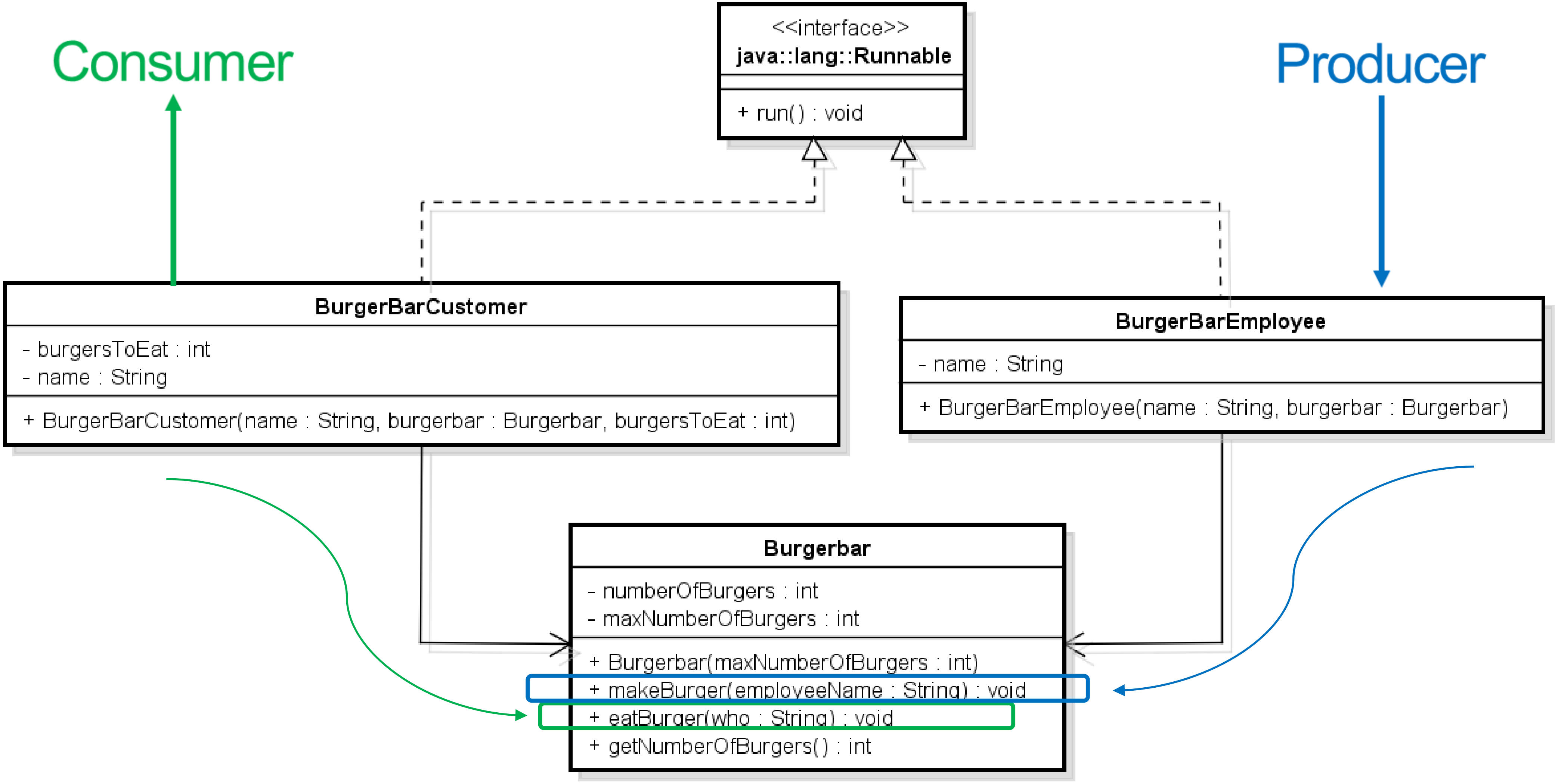


```java
public class CounterTest {
   public static void main(String[] args) {
      Counter counter = new Counter(0, 100); // monitor/shared resource
      System.out.println("Starting Counter: " + counter.getValue());

      CounterIncrementer ci1 = new CounterIncrementer(counter, 300);
      CounterDecrementer cd1 = new CounterDecrementer(counter, 300);
      CounterIncrementer ci2 = new CounterIncrementer(counter, 300);
      CounterDecrementer cd2 = new CounterDecrementer(counter, 300);

      Thread t1 = new Thread(ci1, "Incrementer1");
      Thread t2 = new Thread(ci2, "Incrementer2");
      Thread t3 = new Thread(cd1, "Decrementer1");
      Thread t4 = new Thread(cd2, "Decrementer2");

      t1.start();
      t2.start();
      t3.start();
      t4.start();
   }
}
```

Counter Example



Burger bar (another counter)


Consumer
Producer
<<interface>>
java::lang::Runnable
+ run() : void
BurgerBarCustomer
- burgersToEat : int
- name : String
+ BurgerBarCustomer(name : String, burgerbar : Burgerbar, burgersToEat : int)
BurgerBarEmployee
- name : String
+ BurgerBarEmployee(name : String, burgerbar : Burgerbar)
Burgerbar
- numberOfBurgers : int
- maxNumberOfBurgers : int
+ Burgerbar(maxNumberOfBurgers : int)
+ makeBurger(employeeName : String) : void
+ eatBurger(who : String) : void
+ getNumberOfBurgers() : int

Queues (limited)

- A queue
  - Customers in a waiting room
  - Washing halls for cars
  - Parking places

Producer Consumer Problem

- Characterized by programs that use a buffer (queue)
- Two processes: producers and consumers share a buffer with a fixed size
  - producer puts an item to the buffer
  - consumer takes an item from the buffer
- Observations
  - ❖ What happens when the producer wants to put an item to the buffer that is already full?
  - ❖ What about when the consumer wants to take an item from the buffer when the buffer is empty?

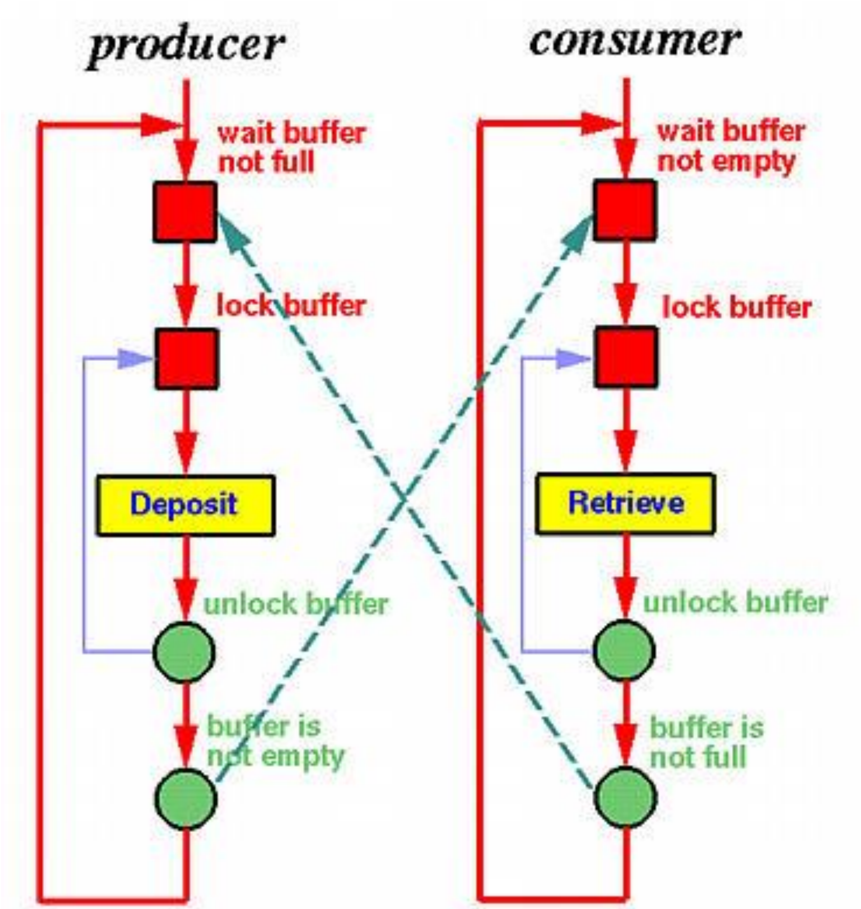
Producer-Consumer

- Everyone operating in/on the shared resource
    - Synchronous or buffered communication
        - Blocking others while operating (synchronization)
- Producers
    - produce items that are sent to consumer(s)
    - waiting for a condition to produce (wait)
    - updating values or adding objects to a queue (notify)
- Consumers
    - receive items and process them independently
    - waiting for a positive value or an object to consume (wait)
    - decreasing values or removing objects from a queue (notify)

The producer-consumer is a typical thread synchronization problem that uses the wait() and notify() methods.

Producer-Consumer - Monitor

- Producer
  - Waits while the buffer is full
  - Deposit its data
  - Notify the consumers that the buffer is not empty.
- Consumer
  - Waits while the buffer is empty
  - Retrieve a data item
  - Notify the producers that the buffer is not full.



https://www.cs.mtu.edu/~shene/NSF-3/e-Book/SEMA/TM-example-buffer.html

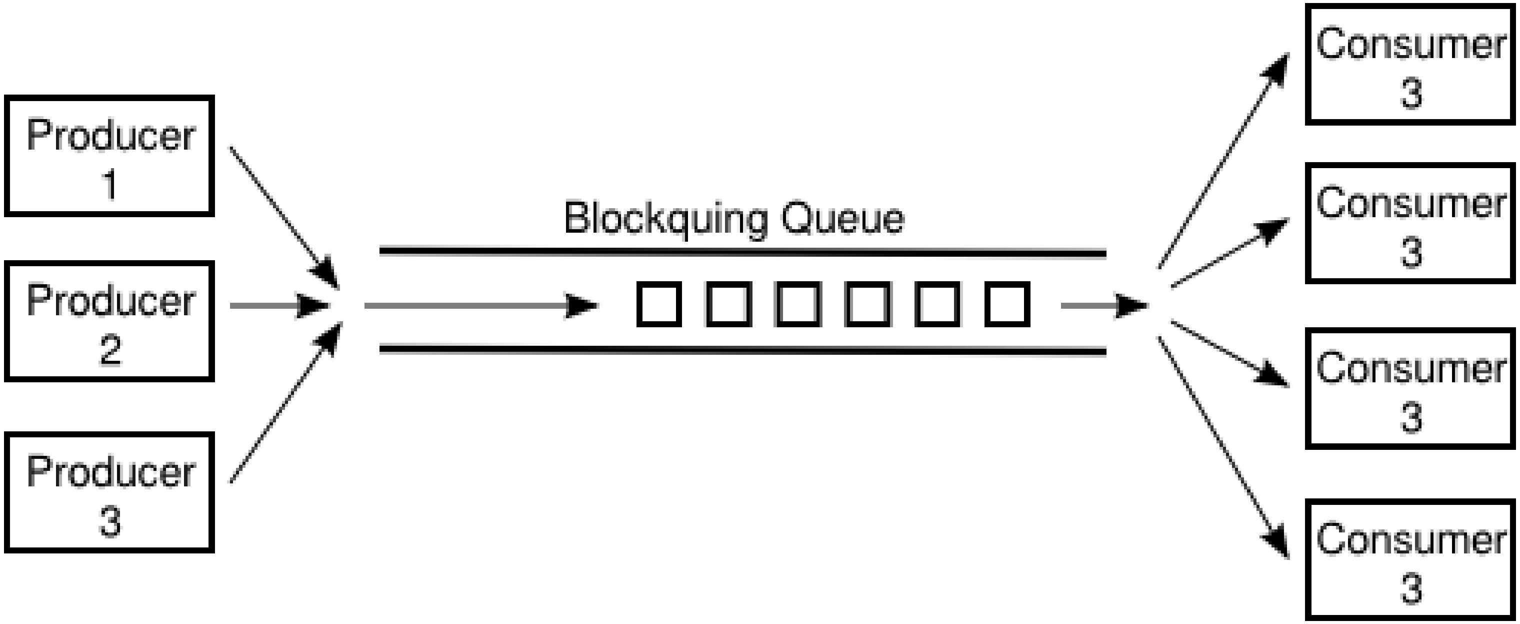
Monitor - Example

```
class Counter {
  private long value;
  private long max;
  private long min;

  //...
  public synchronized void increment()
  {
     while (value >= max)
     {
       try
       {
          wait();
       }
       catch (InterruptedException e)
       {
          //...
       }
     }
     value++;
     if (value == min+1)
     {
       notify();
     }
   }
```

```
  public synchronized void decrement()
  {
     while (value <= min)
     {
       try
       {
          wait();
       }
       catch (InterruptedException e)
       {
          //...
       }
     }
     value--;
     if (value == max-1)
     {
       notify();
     }
   }
}
```

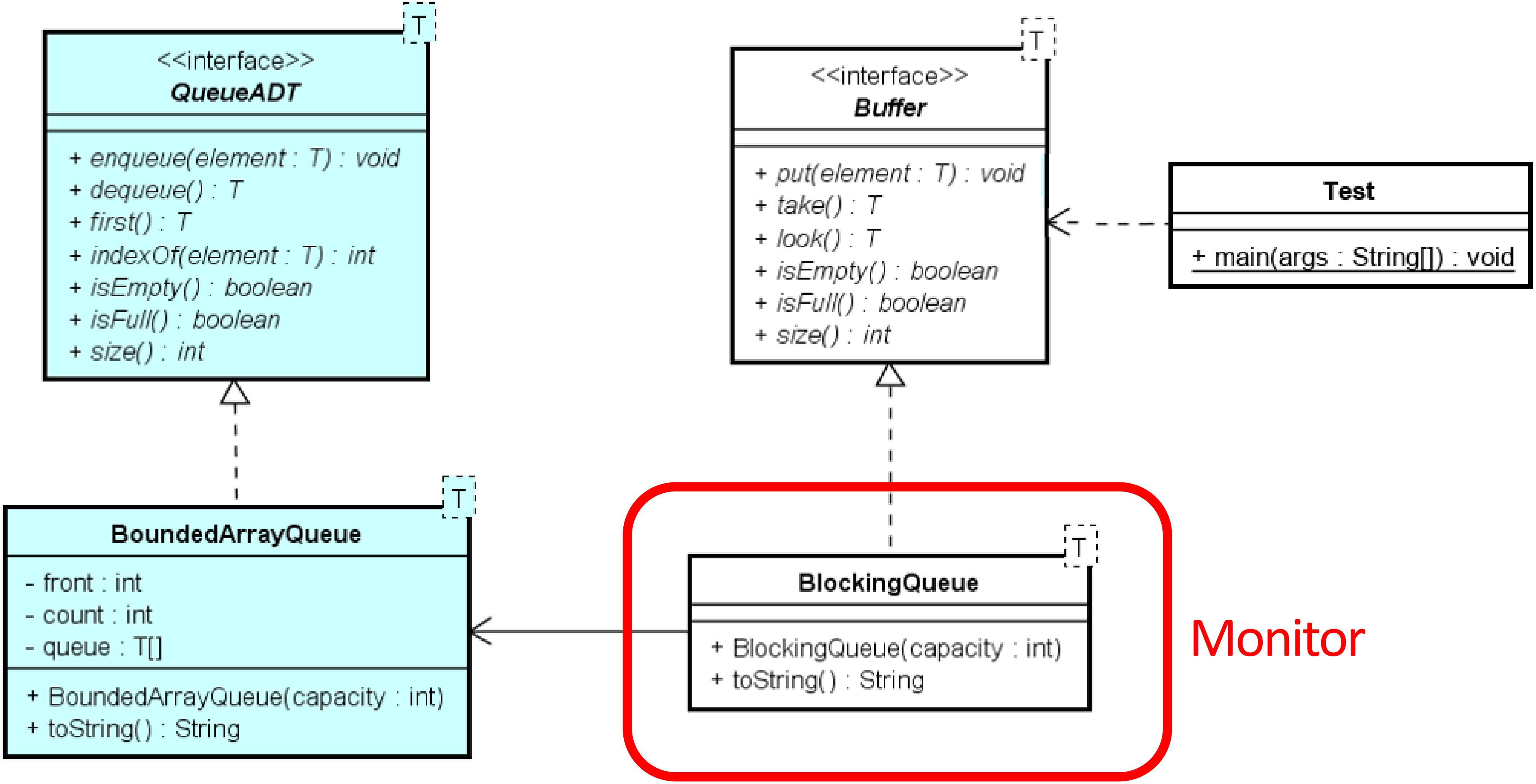
Producer-Consumer Monitor/Blocking Queue

- Producer
  - Waits while the buffer is full
  - Deposit its data
  - Notify the consumers that the buffer is not empty.
- Consumer
  - Waits while the buffer is empty
  - Retrieve a data item
  - Notify the producers that the buffer is not full.

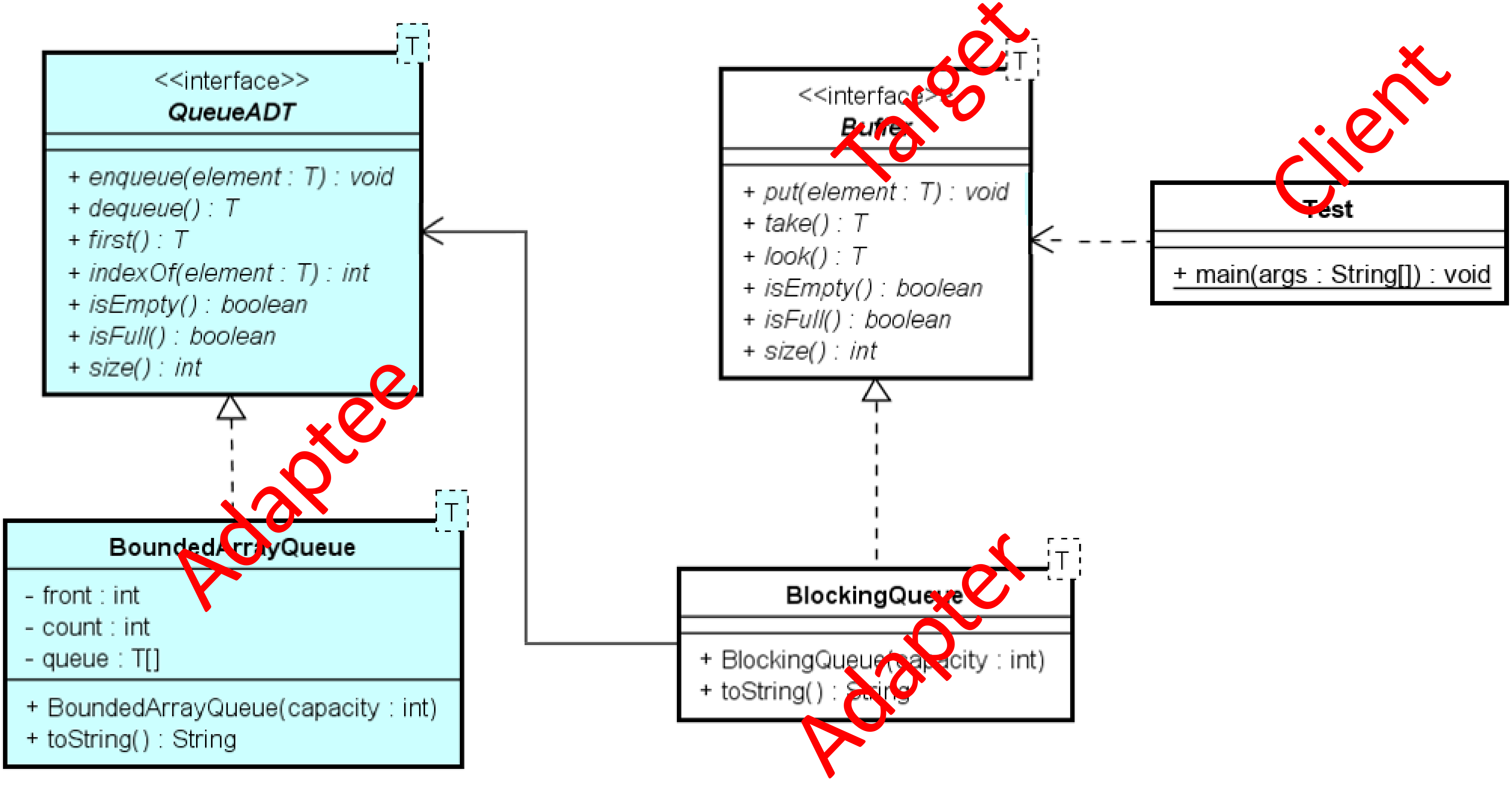


Reference: http://math.hws.edu/javanotes/c12/s3.html

Implementing a Blocking Queue

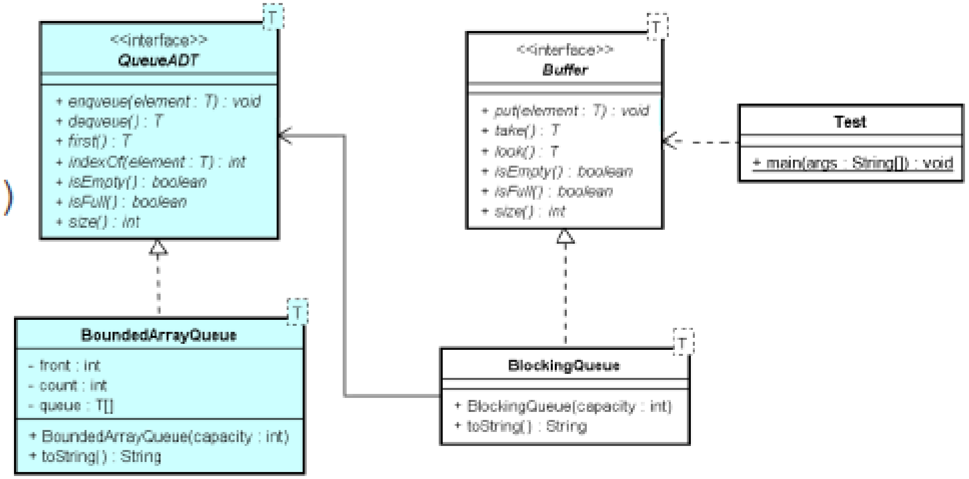


Implementing a Blocking Queue



Implementing a Blocking Queue (Adapter pattern)

```java
public class BlockingQueue<T> implement Buffer<T>
{
  private QueueADT<T> queue;
  public BlockingQueue(int capacity)
  {
    this.queue = new BoundedArrayQueue<>(capacity);
  }

  @Override public synchronized void put(T element)
  {
    while (queue.isFull())
    {
      try
      {
        wait();
      }
      catch (InterruptedException e)
      {
        //...
      }
    }
    queue.enqueue(element);
    notifyAll();
  }
  // ...
}
```

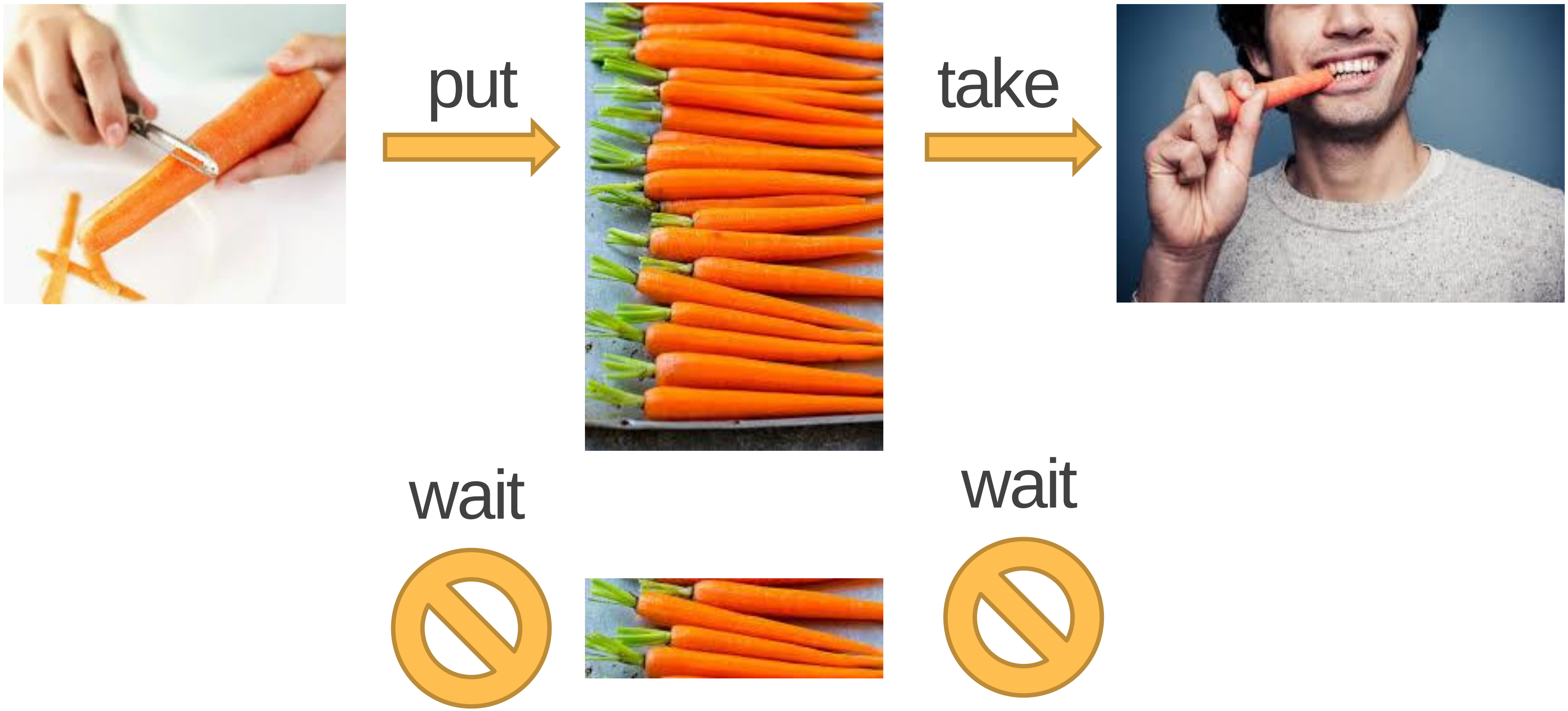


Queues in monitor classes

1. BlockingQueue (synchronized with wait/notify)
   - the monitor *is* the queue (the queue *is* the monitor)
   - a general reusable class

2. Specific designed monitor class
   - the monitor class *has* a queue (or more or other collections)
   - the class has synchronized methods and therefore, the queue/collection don’t have to be thread safe
   - a specific class designed for one system only

Peeling and eating carrots



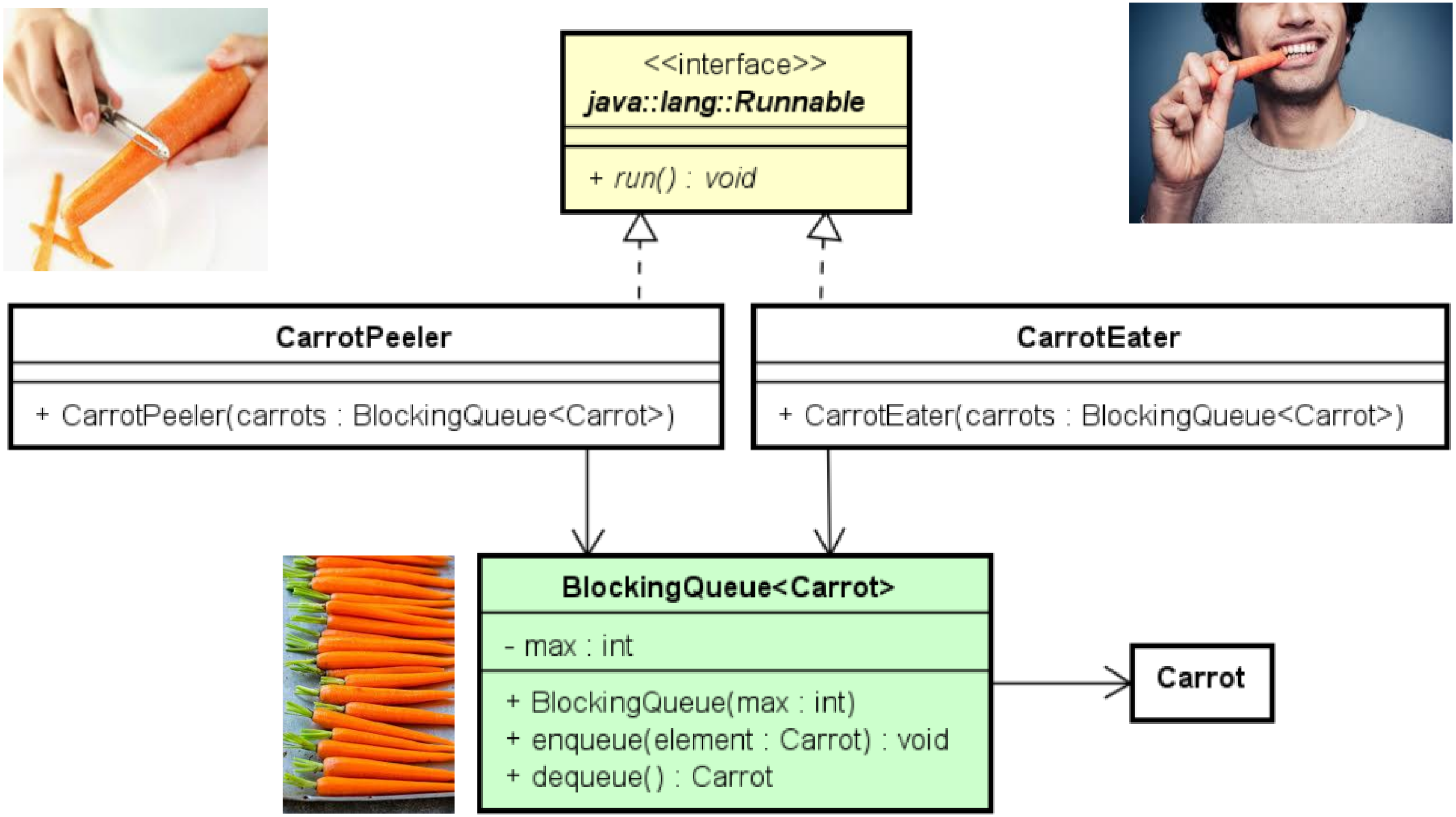
put

take

wait

wait

1: Using a general Blocking Queue (Monitor)



<<interface>>
java::lang::Runnable

+ *run() : void*

CarrotPeeler

+ CarrotPeeler(carrots : BlockingQueue<Carrot>)

CarrotEater

+ CarrotEater(carrots : BlockingQueue<Carrot>)

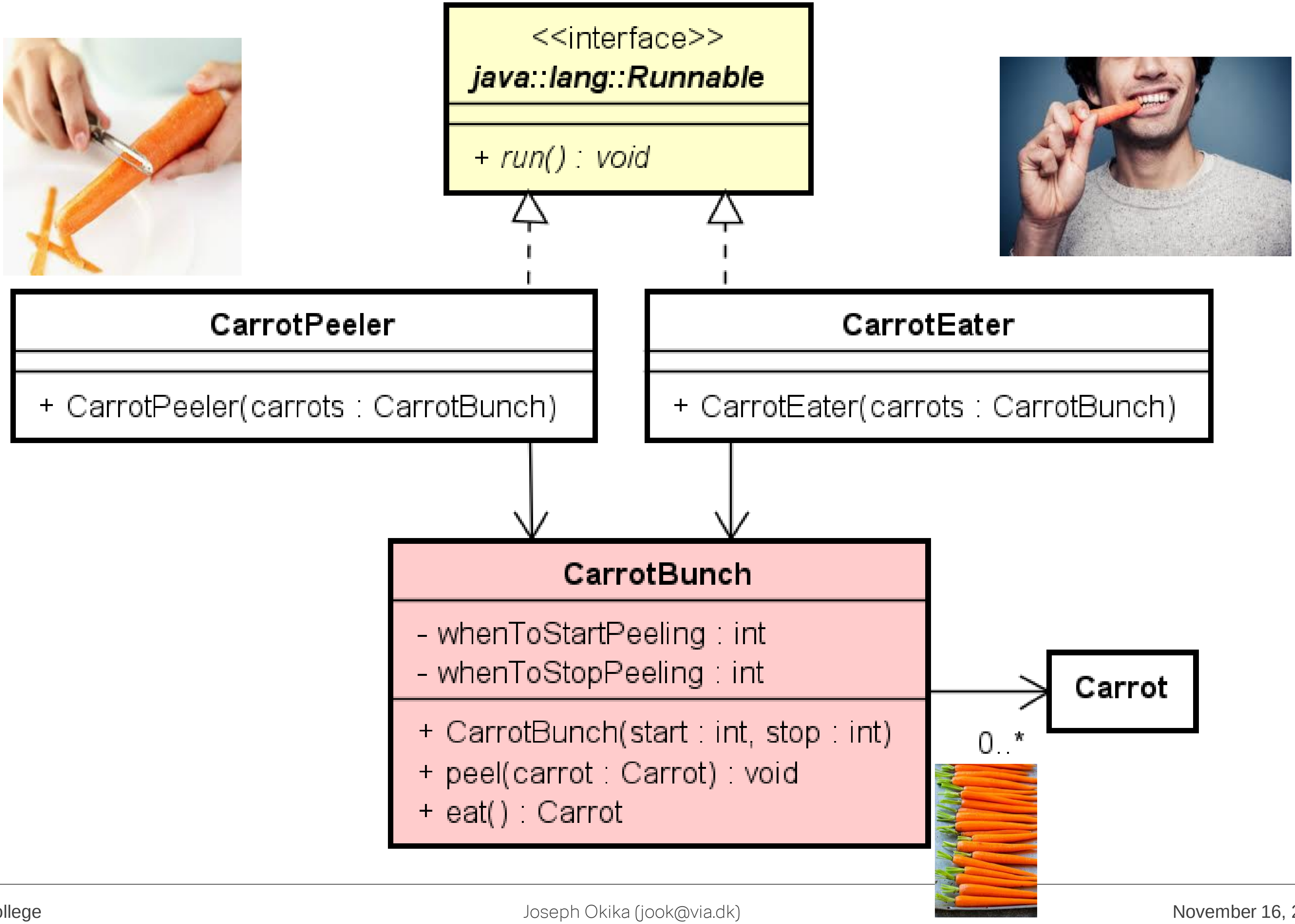
BlockingQueue<Carrot>

- max : int

+ BlockingQueue(max : int)
+ enqueue(element : Carrot) : void
+ dequeue() : Carrot

Carrot

2: Make a specialized Monitor (with a Queue)



<<interface>>
java::lang::Runnable

+ *run() : void*

CarrotPeeler

+ CarrotPeeler(carrots : CarrotBunch)

CarrotEater

+ CarrotEater(carrots : CarrotBunch)

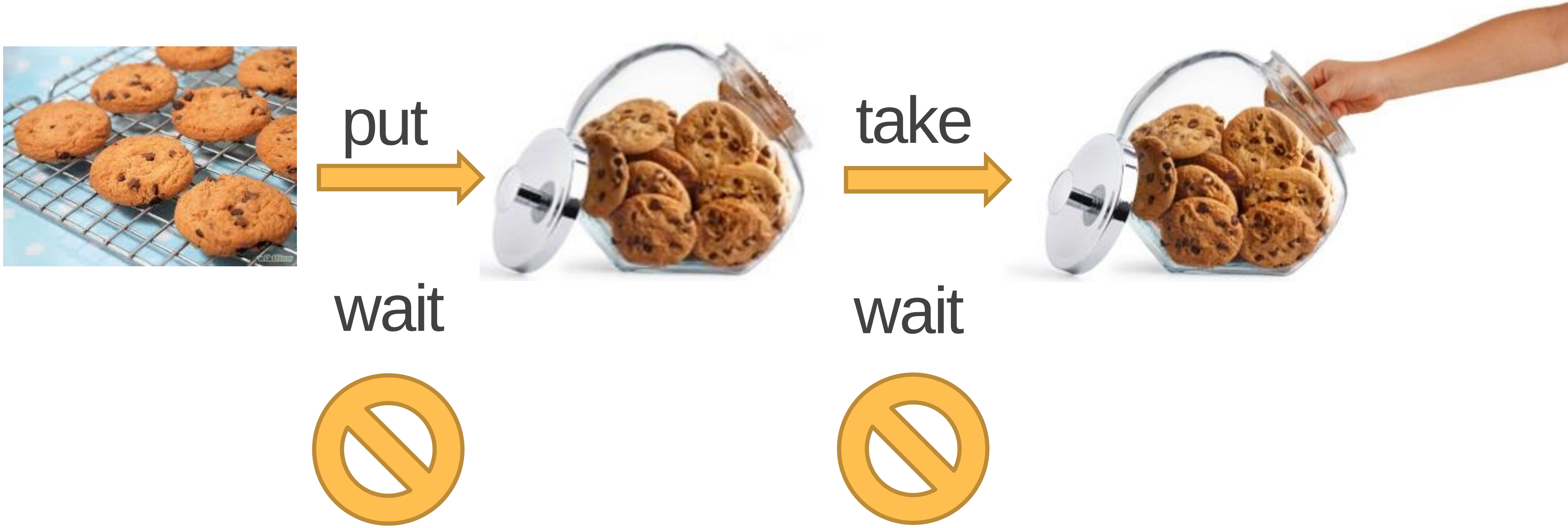
CarrotBunch

- whenToStartPeeling : int
- whenToStopPeeling : int

+ CarrotBunch(start : int, stop : int)
+ peel(carrot : Carrot) : void
+ eat() : Carrot

Carrot

Cookie Jar – Baking and Eating Cookies



put

take

wait

wait

- Baker
  - It takes time to bake cookies
  - Bake only when there is a minimum
  - Put in the jar, when baking is finished
- Eater
  - Keep eating

Using BlockingQueue
from package java.util.concurrent

- Encapsulates the synchronization for you
- ArrayBlockingQueue: a bounded implementation class for BlockingQueue.
  - thread-safe buffer class that implements interface BlockingQueue
  - declares put that places element at the end of the BlockingQueue
    - waiting if the queue is full
  - declares a take that removes an element from the head of the queue
    - waiting if the queue is empty

Example – Hands-on

- Restaurant
  - Managing a queue of customers at the reception before getting a table and being served by a wait person(waiter/waitress)