Implementing the Producer / Consumer Pattern Using Wait / Notify

José Paumard
PHD, JAVA CHAMPION, JAVA ROCK STAR

@JosePaumard http://blog.paumard.org/

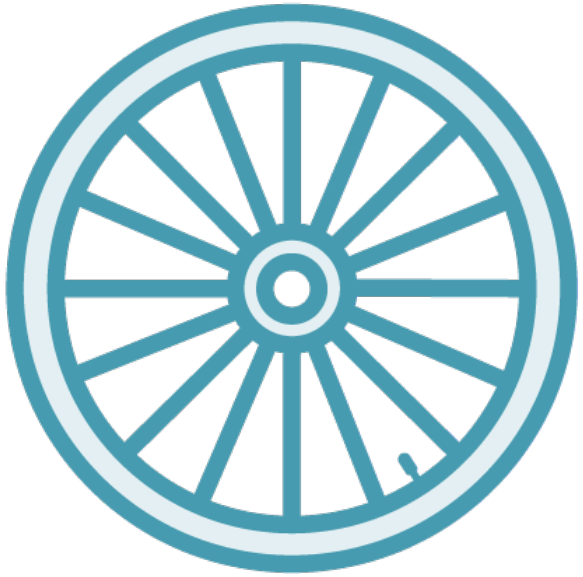
Agenda

The Runnable pattern

What is the producer / consumer pattern

How to implement it using synchronization and the wait / notify pattern

The Runnable Pattern



This is the first pattern used to launch threads in Java

Introduced in Java 1.0

Other patterns have been introduced in Java 5

How to launch a new thread?

A thread executes a task

In Java 1, the model for a task is the Runnable interface

```
public interface Runnable {
    void run();
}
```

A thread executes a task

In Java 1, the model for a task in the Runnable interface

```
@FunctionalInterface
public interface Runnable {
    void run();
}
```

A thread executes a task

In Java 1, the model for a task in the Runnable interface

It has only one method, it is thus a functional interface

```
Runnable task = () -> System.out.println(“Hello world!”);
```

A thread executes a task

1) **create an instance of Runnable**

```
Runnable task = () -> System.out.println("Hello world!");
Thread thread = new Thread(task);
```

A thread executes a task

1) **create an instance of Runnable**
2) **create an instance of Thread with the task as a parameter**

```
Runnable task = () -> System.out.println("Hello world!");
Thread thread = new Thread(task);
thread.start();
```

A thread executes a task

1) **create an instance of Runnable**
2) **create an instance of Thread with the task as a parameter**
3) **Launch the thread**

```
Runnable task = () -> System.out.println("Hello world!");
Thread thread = new Thread(task);
thread.start();
thread.run(); // NO !!!!!!
```

A common mistake

Do not call the run() method instead of the start() method!

```
Runnable task = () -> System.out.println("Hello world!");
Thread thread = new Thread(task);
thread.start();
thread.run(); // NO !!!!!!
```

A common mistake

Do not call the run() method instead of the start() method!

If you do, the task will be executed...

... but in the current thread!

```
Runnable task = () -> {
    System.out.println(Thread.currentThread().getName());
}
```

Knowing in which thread a task is executed

The `Thread.currentThread()` static method returns the current thread

How to stop a thread?

Stopping a Thread

More tricky than it seems!

There is a method in the Thread class called `stop()`

This method should not be used

It is there for legacy, backward compatibility reasons

The right pattern is to use the `interrupt()` method

```
Thread t1 = ...;
t1.interrupt();
```

Calling interrupt() on a running thread

The code of the task should call `isInterrupted()` to terminate itself

```
Runnable task = () -> {
    while(! Thread.currentThread().isInterrupted()) {
        // the task itself
    }
}
```

Calling interrupt() on a running thread

The code of the task should call `isInterrupted()` to terminate itself

Stopping a Thread

The call to `interrupt()` causes the `isInterrupted()` method to return true

If the thread is blocked, or waiting, then the corresponding method will throw an `InterruptedException`

The methods `wait()` / `notify()`, `join()` throw `InterruptedException`

Producer / Consumer

What is a producer / consumer?

Producer / Consumer

A producer produces values in a buffer

A consumer consumes the values from this buffer

Be careful: the buffer can be empty, or full

Producers and consumers are run in their own thread

```
int count = 0;
int[] buffer = new int[BUFFER_SIZE];

class Producer {
    public void produce() {
        while (isFull(buffer)) {}
        buffer[count++] = 1;
    }
}
```

```
int count = 0;
int[] buffer = new int[BUFFER_SIZE];

class Consumer {
    public void consume() {
        while (isEmpty(buffer)) {}
        buffer[--count] = 0;
    }
}
```

Producer / Consumer

What is wrong with this code?

In fact one main thing

```
class Producer {
    public void produce() {
        while (isFull(buffer)) {}
        buffer[count++] = 1;
    }
}

class Consumer {
    public void consume() {
        while (isEmpty(buffer)) {}
        buffer[--count] = 0;
    }
}
```

Race condition!

Several threads are reading and writing the buffer at the same time = race condition

This will corrupt the array

Fixing the producer / consumer

One way to fix things is to synchronize the access to the array

```
class Consumer {
    public void synchronized consume() {
        while (isEmpty(buffer)) {}
        buffer[--count] = 0;
    }
}

class Producer {
    public void synchronized produce() {
        while (isFull(buffer)) {}
        buffer[count++] = 1;
    }
}
```

Will it fix our problems?

Synchronization can fix our race condition problem, but not if we write it like that

```
class Consumer {
    private Object lock;
    public void consume() {
        synchronized(lock) {
            while (isEmpty(buffer)) {}
            buffer[--count] = 0;
        }
    }
}
```

This code will work if the lock object is the same for all the producers and consumers

```
private Object lock;
```

```
class Consumer {

    public void consume() {
        synchronized(lock) {
            while (isEmpty(buffer)) {}
            buffer[--count] = 0;
        }
    }
}
```

```
class Producer {

    public void produce() {
        synchronized(lock) {
            while (isFull(buffer)) {}
            buffer[count++] = 1;
        }
    }
}
```

Fixing the producer / consumer

Is it really fixed?

```
class Consumer {

    public void consume() {
        synchronized(lock) {
            while (isEmpty(buffer)) {}
            buffer[--count] = 0;
        }
    }
}
```

```
class Producer {

    public void produce() {
        synchronized(lock) {
            while (isFull(buffer)) {}
            buffer[count++] = 1;
        }
    }
}
```

Does it fix our problem?

What happens if the buffer is empty?

```
class Consumer {

    public void consume() {
        synchronized(lock) {
            while (isEmpty(buffer)) {}
            buffer[--count] = 0;
        }
    }
}
```

Does it fix our problem?

What happens if the buffer is empty?

The thread executing this consumer is blocked in the while loop

```
class Consumer {

    public void consume() {
        synchronized(lock) {
            while (isEmpty(buffer)) {}
            buffer[--count] = 0;
        }
    }
}
```

```
class Producer {

    public void produce() {
        synchronized(lock) {
            while (isFull(buffer)) {}
            buffer[count++] = 1;
        }
    }
}
```

Does it fix our problem?

What happens if the buffer is empty?

The thread executing this consumer is blocked in the while loop

So the producer has no chance to add objects to the buffer!

Fixing the producer / consumer (again)

We need a way to “park” a thread while he is waiting for some data to be produced

Without blocking all the other threads

So the key held by this thread should be released while this thread is “parked”

This is the wait / notify pattern

Wait / Notify

wait() / notify()

`wait()` and `notify()` are two methods from the `Object` class

They are invoked on a given object

The thread executing the invocation should hold the key of that object

So: `wait()` and `notify()` cannot be invoked outside a synchronized block

Calling wait()

Calling `wait()` releases the key held by this thread

And puts that thread in a WAIT state

The only way to release a thread from a WAIT state is to notify it

Calling notify()

Calling `notify()` releases a Thread in WAIT state and puts it in RUNNABLE state

This is the only way to release a waiting thread

The released thread is chosen randomly

There is also a `notifyAll()` method

```
private Object lock;
```

```
class Producer {

    public void produce() {
        synchronized(lock) {
            if (isFull(buffer))
                lock.wait();
            buffer[count++] = 1;
            lock.notifyAll();
        }
    }
}
```

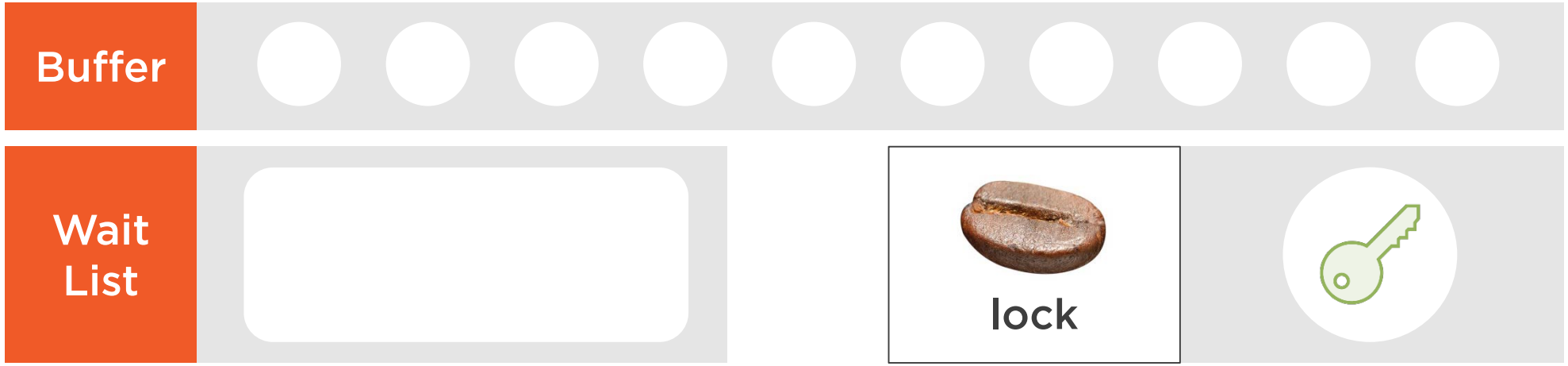
```
private Object lock;
```

```
class Producer {

    public void produce() {
        synchronized(lock) {
            if (isFull(buffer))
                lock.wait();
            buffer[count++] = 1;
            lock.notifyAll();
        }
    }
}
```

```
class Consumer {

    public void consume() {
        synchronized(lock) {
            if (isEmpty(buffer))
                lock.wait();
            buffer[--count] = 0;
            lock.notifyAll();
        }
    }
}
```


Buffer
Wait List
lock
Consumer
Producer

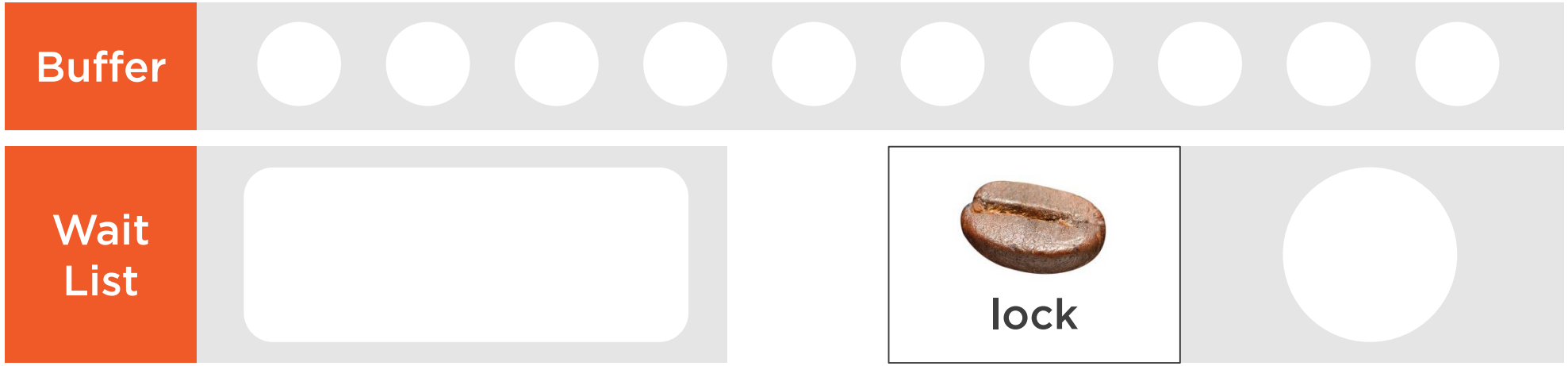
Buffer

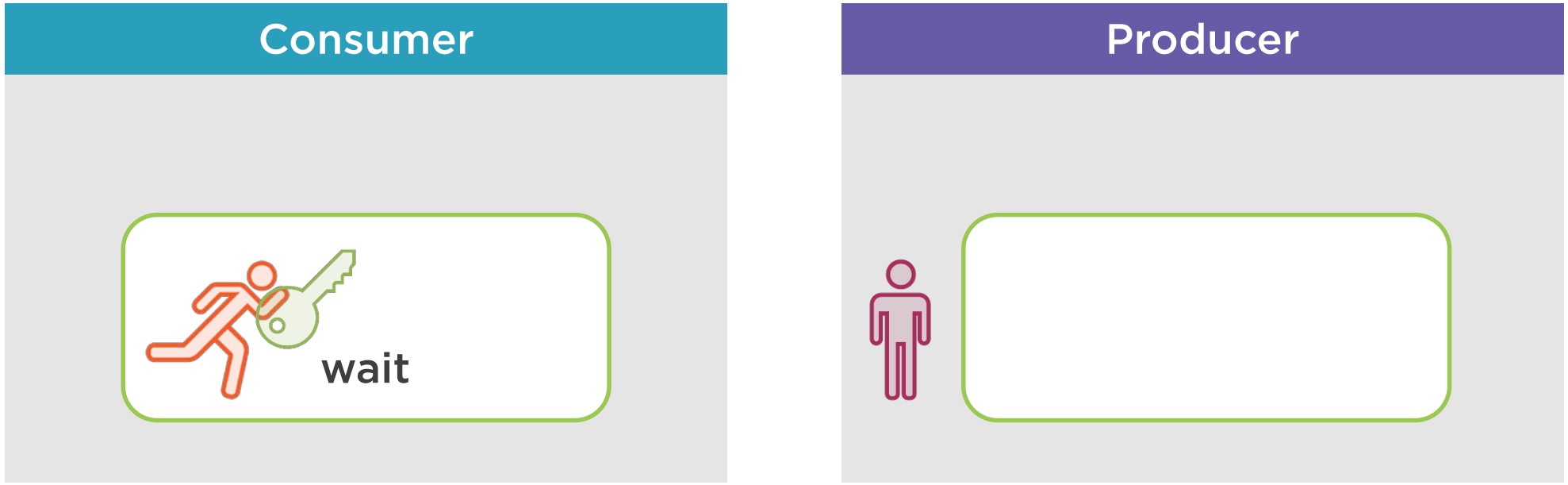
Wait List

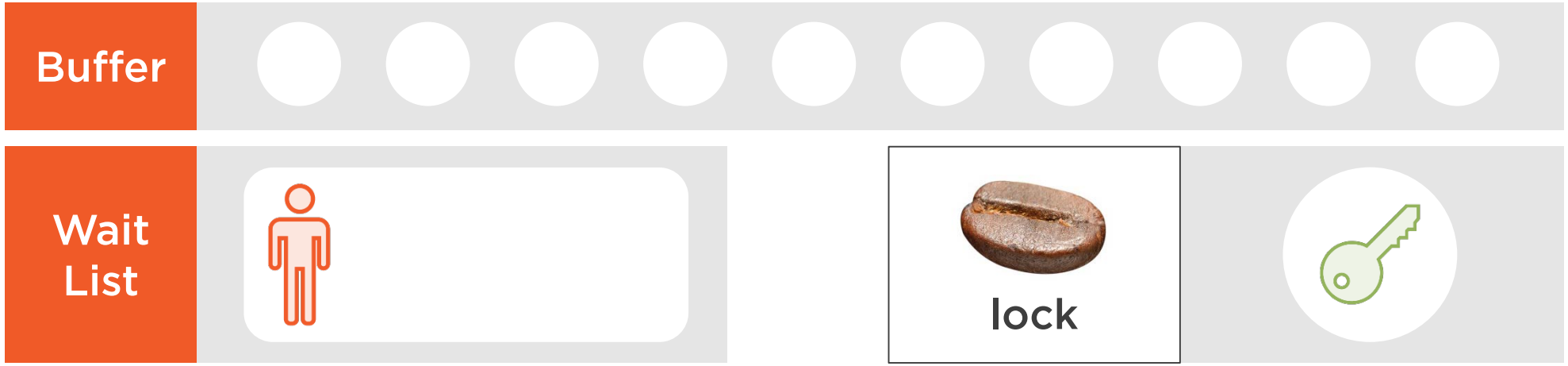
lock

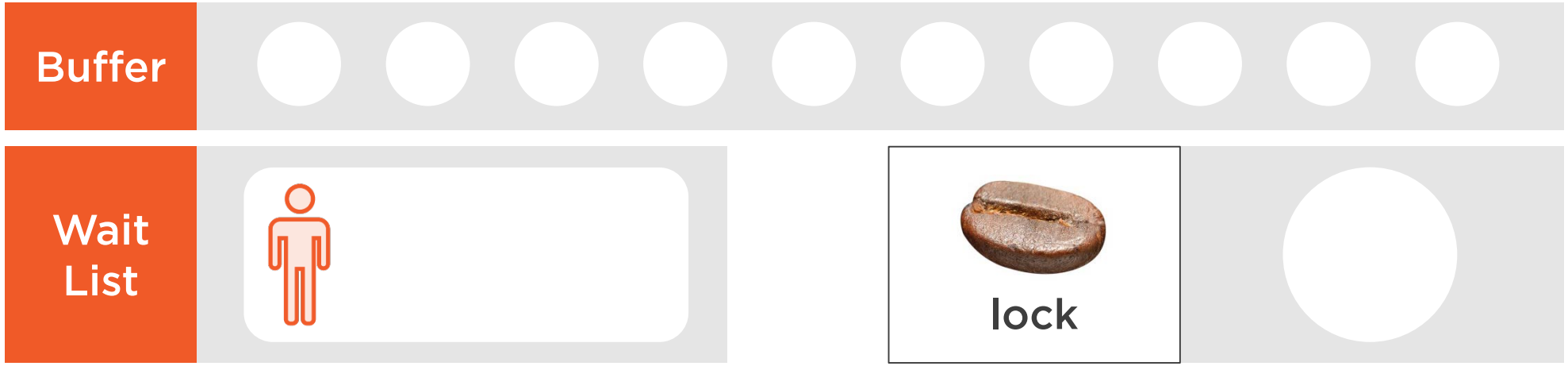
Consumer

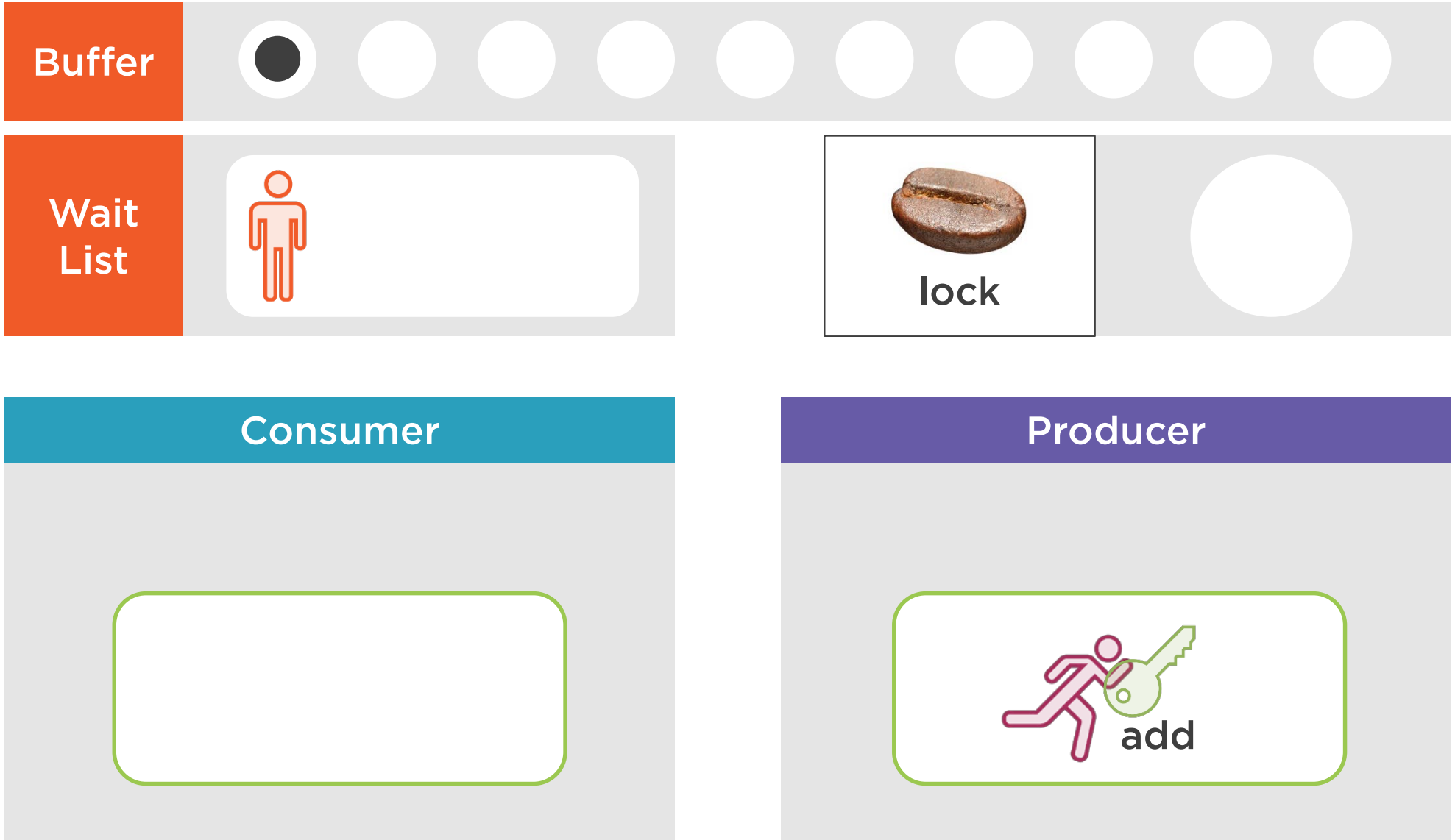
Producer

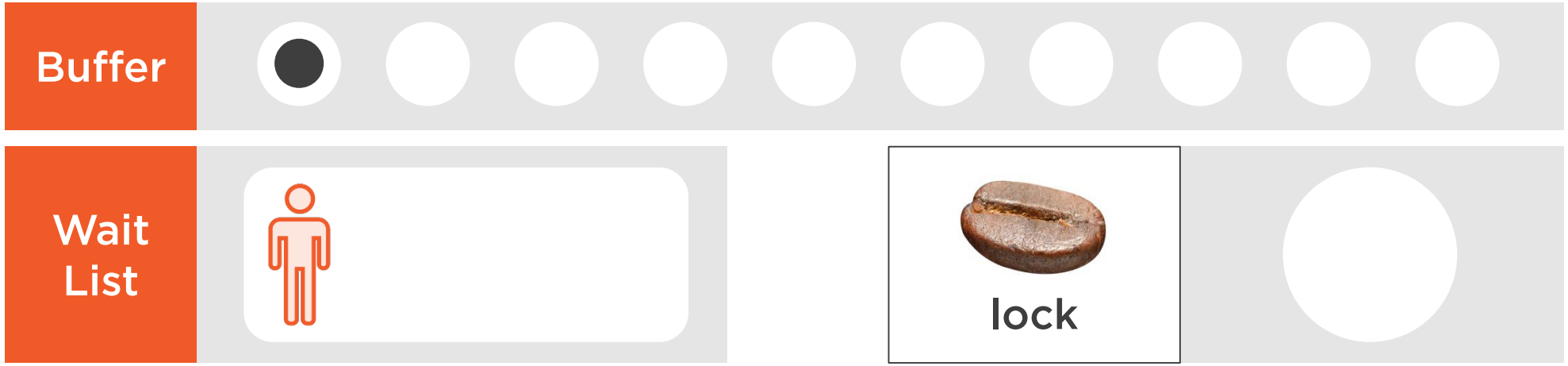

Buffer
Wait List
lock

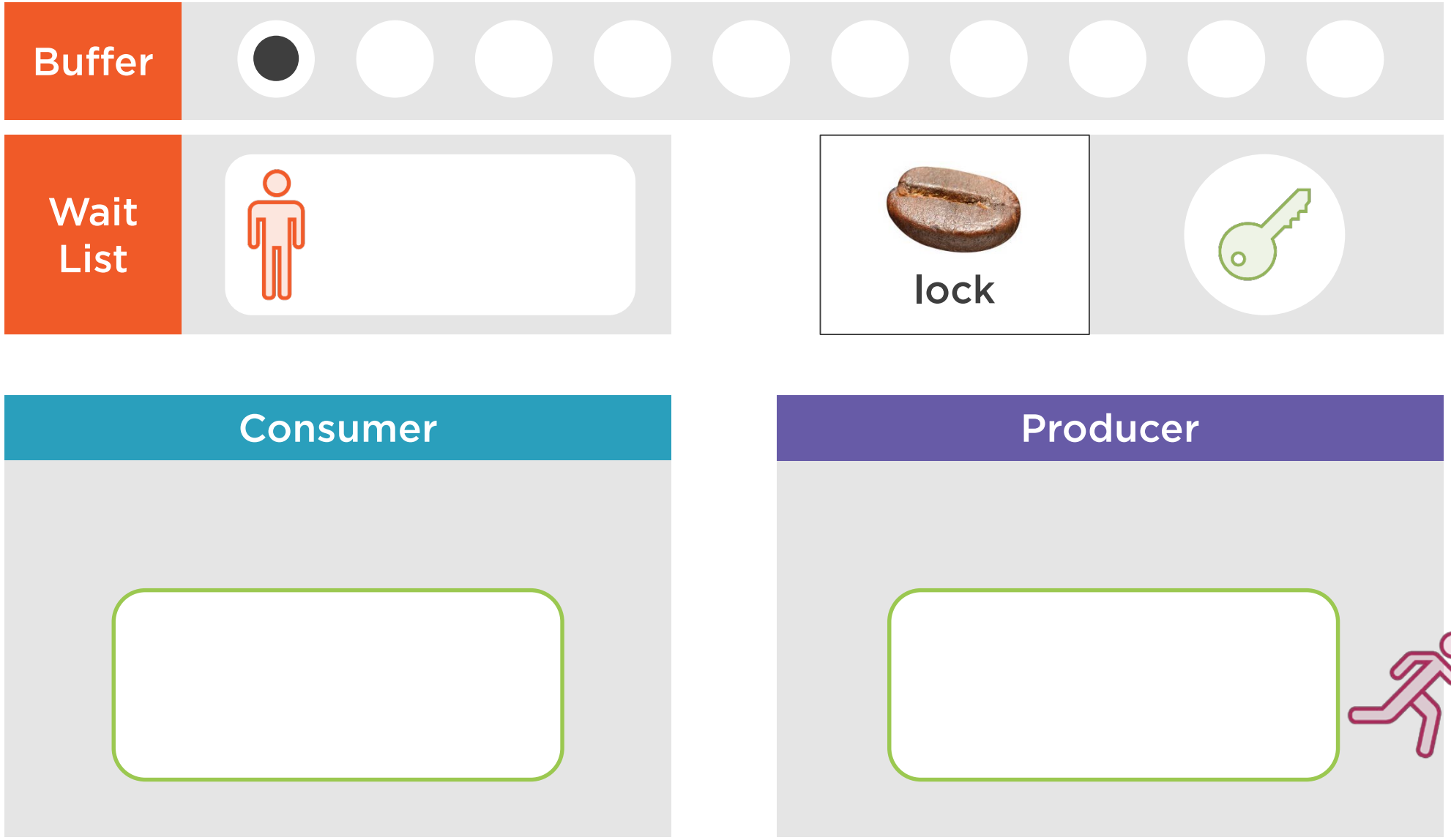

Consumer
wait
Producer

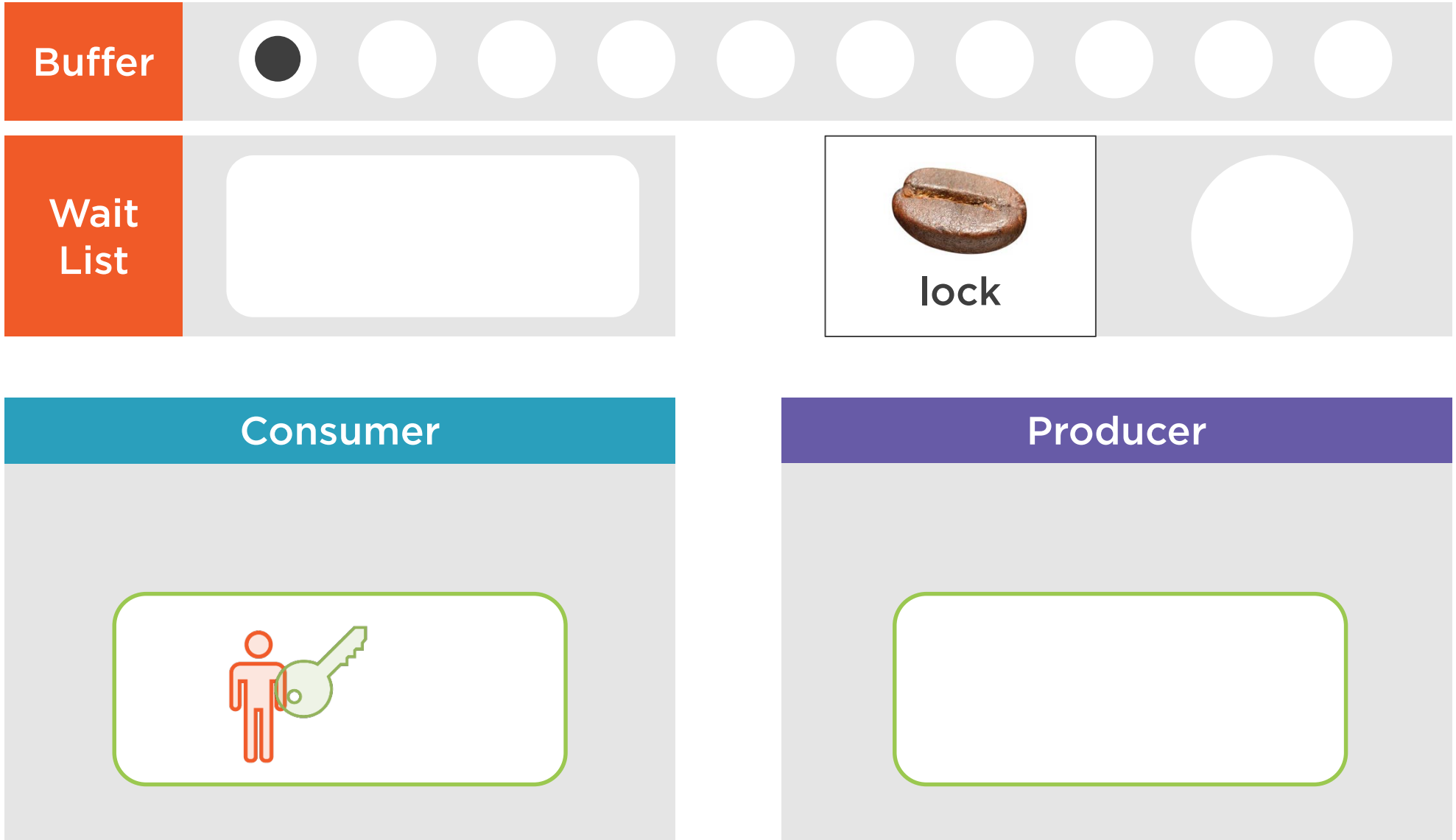

Buffer
Wait List
lock
Consumer
Producer

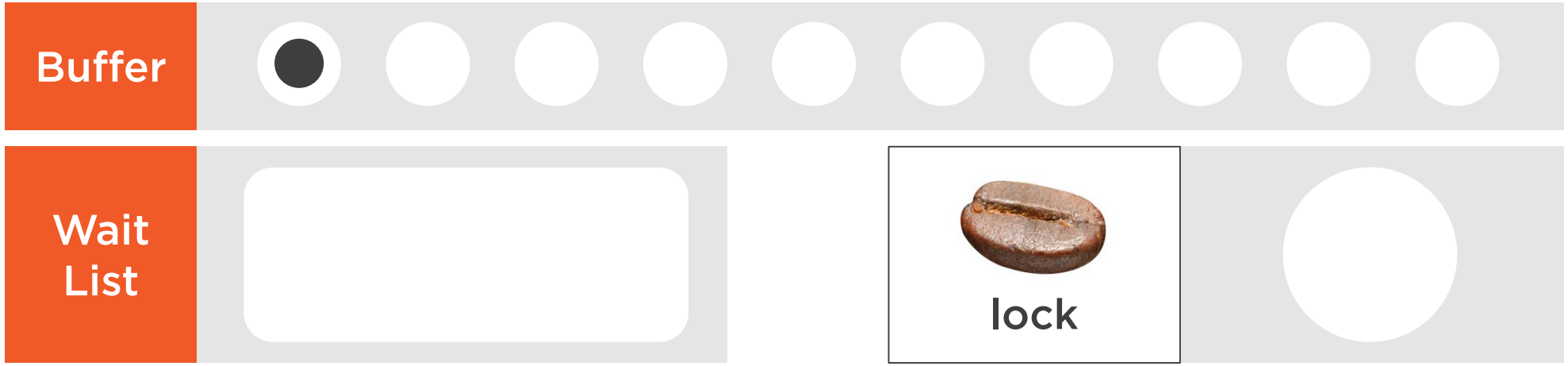

Buffer
Wait List
lock
Consumer
Producer


Buffer
Wait List
lock
Consumer
Producer
add


Buffer
Wait List
lock
Consumer
Producer
notify


Buffer
Wait List
lock
Consumer
Producer


Buffer
Wait List
lock
Consumer
Producer


Buffer
Wait List
lock
Consumer
take
Producer

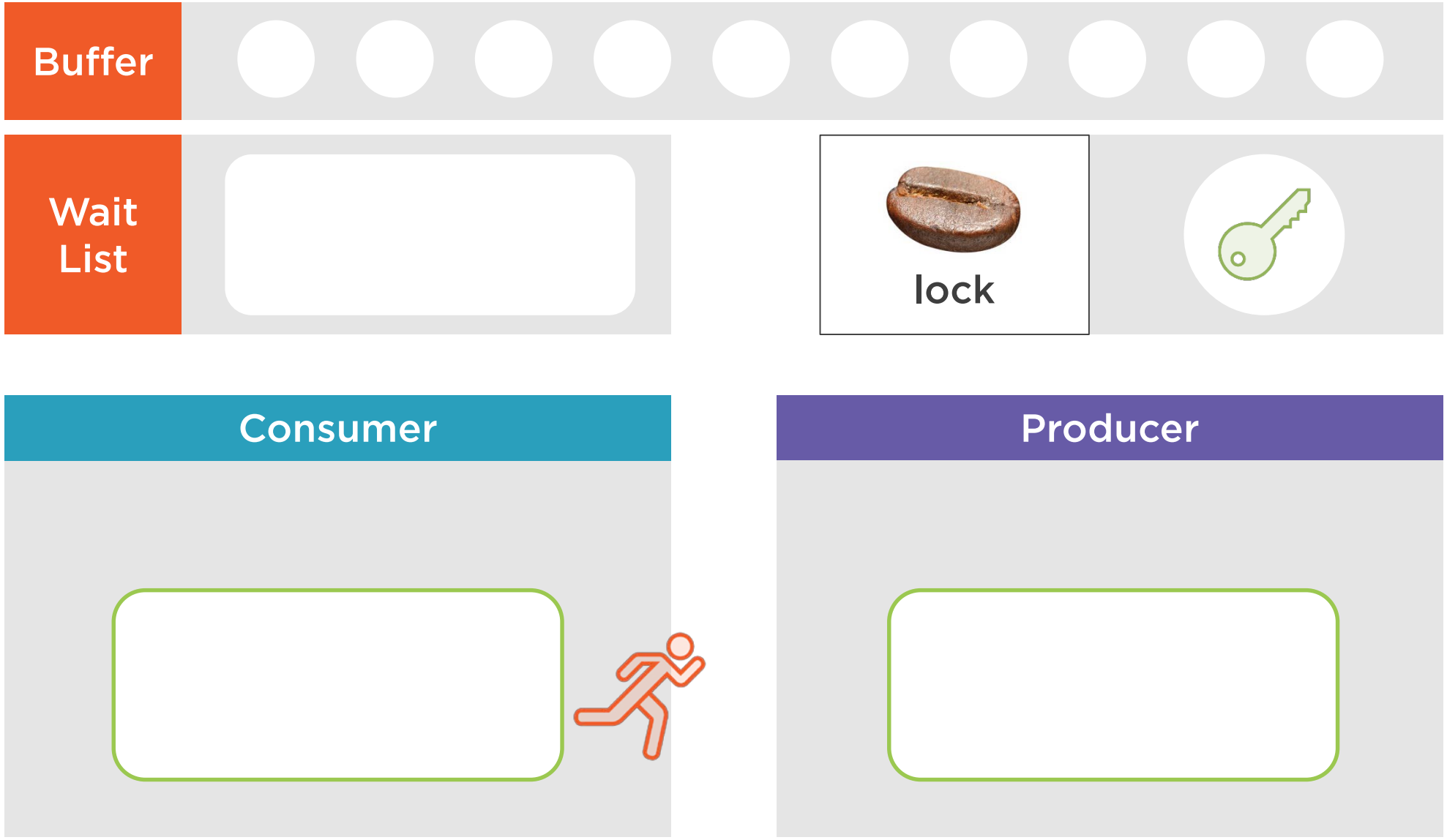
Buffer

Wait List

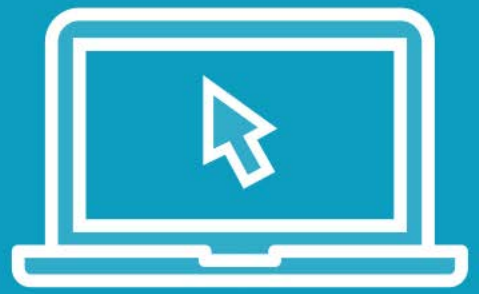
lock

Consumer

Producer


Buffer
Wait List
lock
Consumer
Producer

Demo



Let us see some code!

Let us see this producer / consumer pattern in action!

States of a Thread

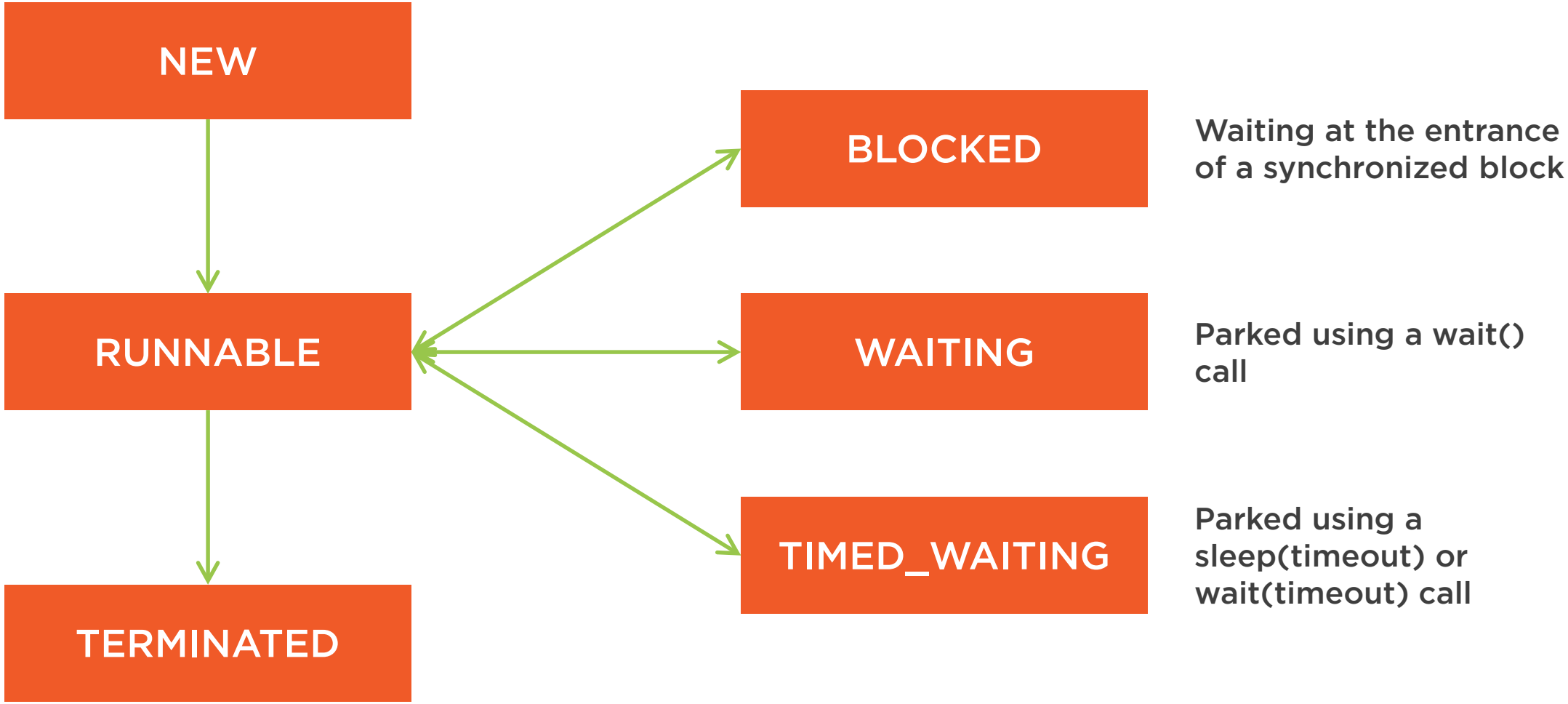
A Thread Has a State

A thread can be running or not

If it is not running, can the thread scheduler give it a hand?

If it is in the WAIT list, the answer is no

Let us see the different states a thread can have


NEW
RUNNABLE
TERMINATED
BLOCKED
WAITING
TIMED_WAITING
Waiting at the entrance of a synchronized block
Parked using a wait() call
Parked using a sleep(timeout) or wait(timeout) call

A Thread Has a State

The thread scheduler can run the threads in the state RUNNABLE

A BLOCKED thread can only run again when the key is released

A WAITING thread can only run again when the notify() method is called

```
Thread t = ...;

Thread.State state = t.getState();
```

Getting the state of a thread

The getState() method returns a enumerated value of type Thread.State

```
public enum State {
    NEW, RUNNABLE, TERMINATED,
    BLOCKED, WAITING, TIME_WAITING;
}
```

Getting the state of a thread

The State enumeration is a member enumeration of the Thread class

Wrapup

What did we learn?

The Runnable pattern works with the Runnable interface and the Thread class

How to set up a producer / consumer pattern

What can go wrong with it

How to fix it using synchronization and the wait / notify pattern