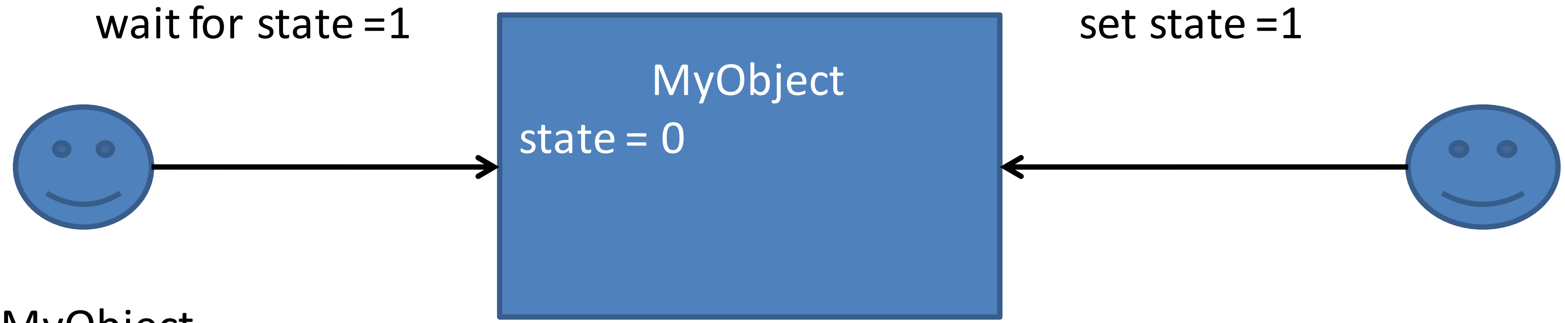
Threads Coordination: How **Not** To Do It



wait for state =1

MyObject

state = 0

set state =1

```
class MyObject
{
        private int state = 0;
        public synchronized void setState(int state) {
                this.state=state;
        }

        public synchronized void waitForState(int state) {
                while (this.state !=state) {
                        sleep(100);
                }
                System.out.println(“IT IS  ” + state);
        }
}
```

If **wateForState**() began to run, monitor is locked and **setState**() can not start

How To: Polling

```
Thread 1
lock monitor
while (check condition  != true) {
    unlock monitor
    sleep (period);
    lock monitor;
}
do something
unlock monitor
```

if event occurs here,
it will be accepted only
after sleep period

```
Thread 2
lock monitor
set state
unlock monitor
```

Polling:

+ simple

- event accepted with delay
- min delay → max CPU load

How To: Blocking

Thread 1

```
lock monitor
while (check condition  != true) {
    unlock monitor
    wait until notify        atomically  cond_var.wait(monitor)
    lock monitor
}
do something
unlock monitor
```

Thread 2

```
lock monitor
set state
cond_var.notify()
unlock monitor
```

Polling:

+ simple
- event arrives with delay
- min delay ➔ max CPU load

Blocking:

- not so simple
+ minimal delay for arrived event
+ minimal CPU load

Condition Variable: The Concept

Condition Variable is a synchronization device that allows threads to suspend execution and release the processors until shared data will be changed to have a desired state.

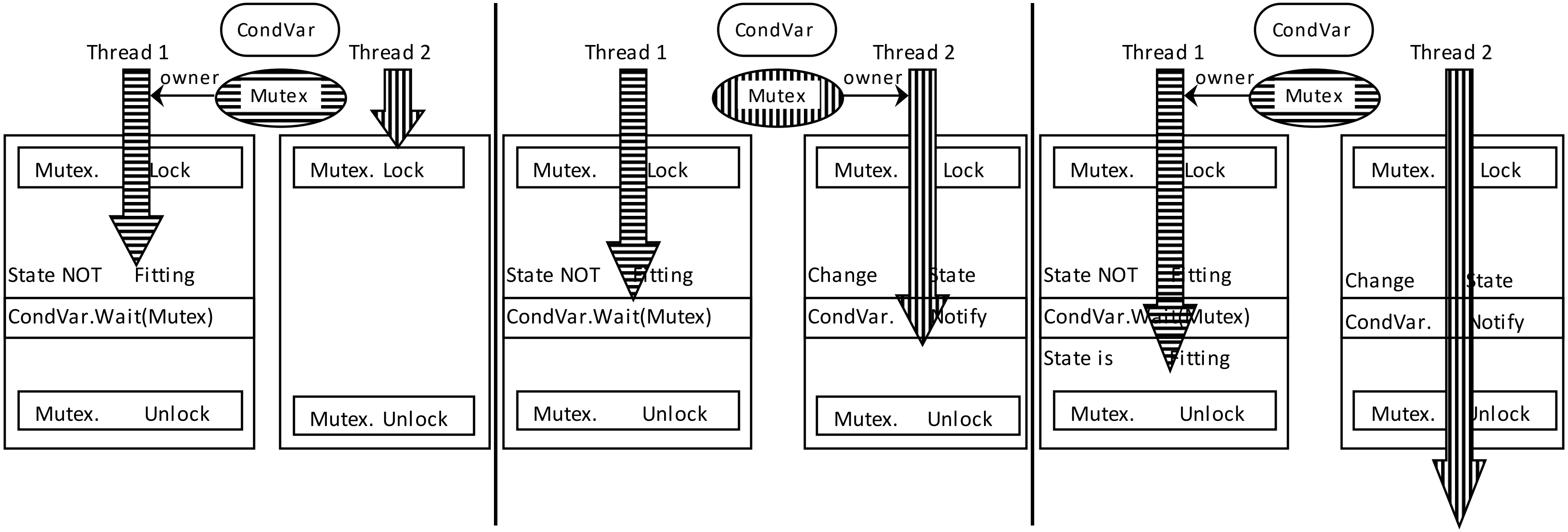
The basic operations on Condition Variables are:

- Wait for the specific state of shared data, suspending the thread execution until another thread changes the shared data and notifies (signals) the Condition Variable, that state is changed.
- Notify one (signal) or all (broadcast) threads, waiting for specific condition, that shared data state is changed.

A Condition Variable is stateless signaling device. Notification (signal) does not change the state of device. It affects only the thread(s), that are waiting on this Condition Variable in the moment of notification (signal).

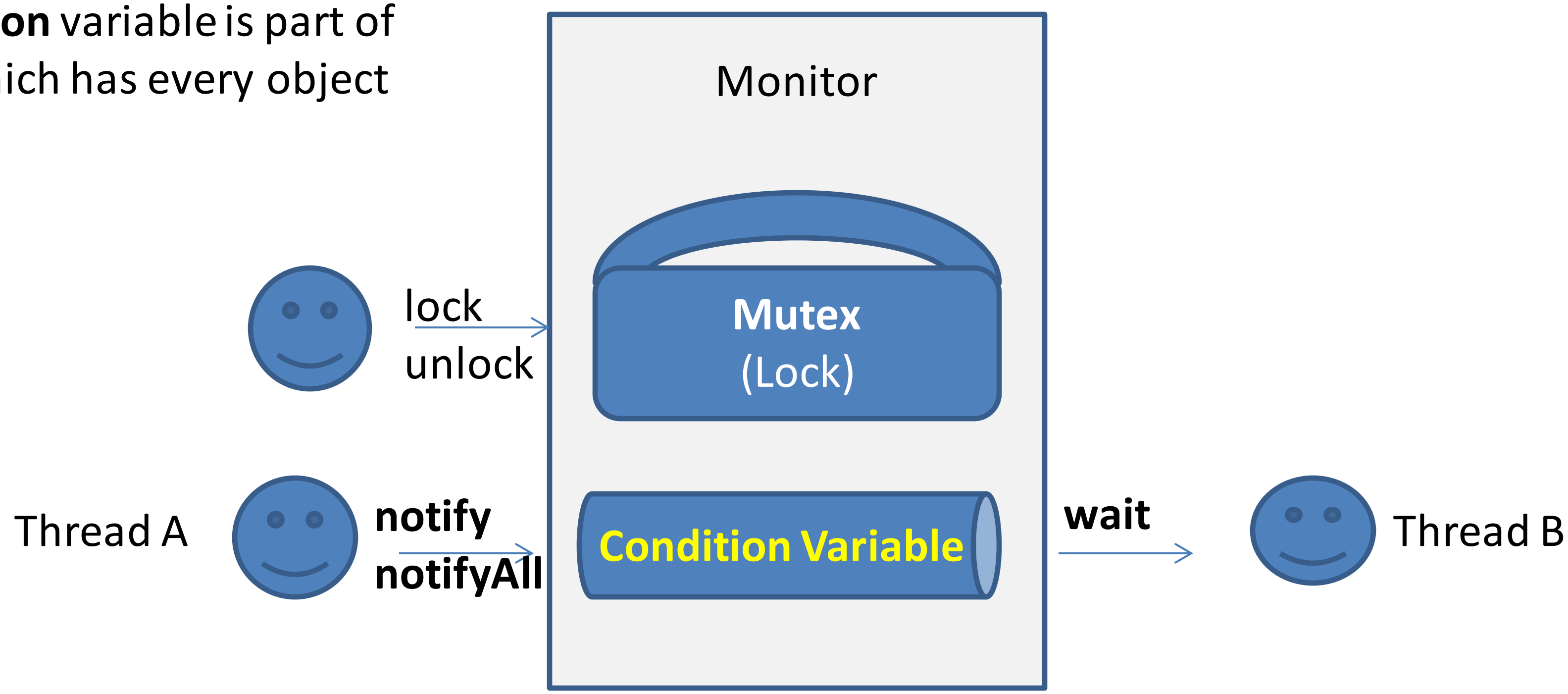
A Condition Variable must always be associated with a Mutex, to avoid the race condition where a thread prepares to wait on a Condition Variable and another thread notifies (signals) the condition just before the first thread actually waits on it.

Being awaken after Wait, the thread always must re-evaluate its condition

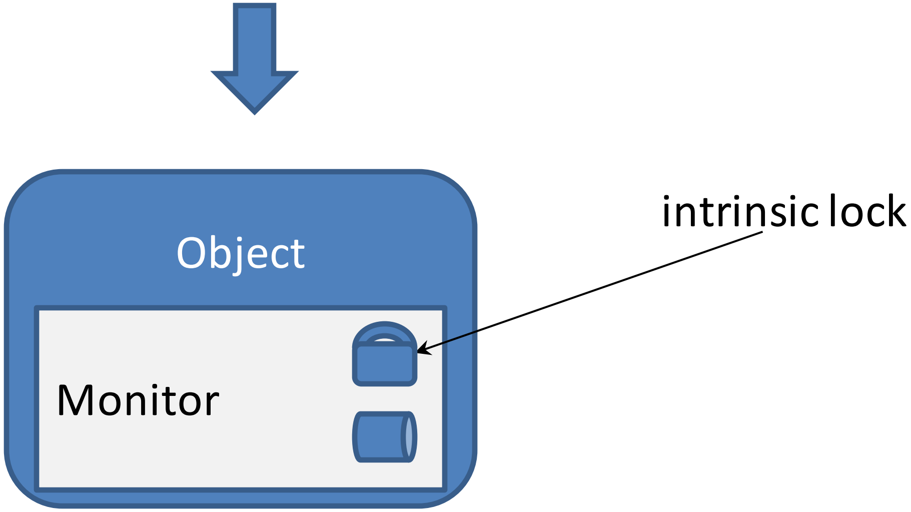


Java Implementation

The **Condition** variable is part of Monitor which has every object



The methods ***wait()***, ***notify()***, ***notifyAll*** are delegated to Object class and could be called only from synchronized methods / statements, taking the lock of the same object monitor



Waiting For Multiple Different States

Approach A: Using Monitor (lock and condition) of Single Object

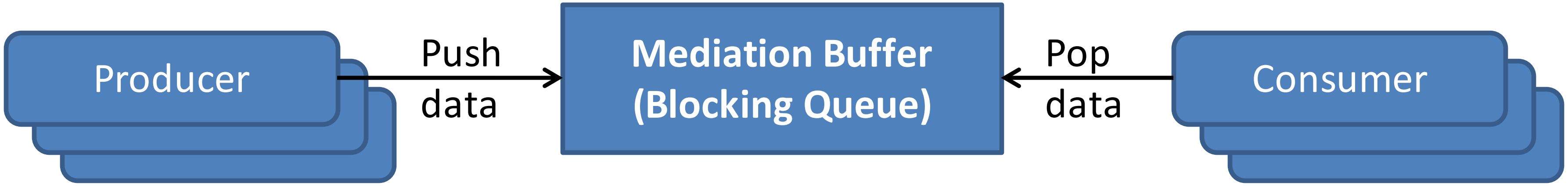
- All synchronization is performed by means of ***synchronized*** methods of target object
- Multiple Threads call ***wait()*** on the same object, actually waiting for its different states
- Thread, changing the state of target object, calls ***notifyAll()*** to wake-up all waiting threads
- Each waked-up Thread:
    - Re-acquires the monitor lock
    - Re-evalutes the state
    - If state not fits, repeats ***wait*** (once more releasing the lock)
- As result:
    - only Thread(s) waiting for specific state will continue the work
    - rest of Threads ***performs extra idle work ☹***

Approach B: (since 1.5) Using Multiple Condition-s associated with single Reentrant Lock

- All synchronization is performed by means of ***ReentrantLock*** instance
- For each state of interest the separate ***Condition*** instance is created by means of call to ***ReentrantLock.newCondition() .***Multiple Threads waiting for specific state call method ***await()*** of specific state-related ***Condition*** instance
- Thread, changing the state of target object calls ***signal() / signalAll()*** methods of new state-related Condition instance only
- As result:
    - only Thread(s) waiting for specific state will be awaken
    - rest of Threads ***never perform idle work ☺***

Producer-Consumer Pattern

- The Producer repeatedly generates a piece of data and puts it into the Mediation Buffer
- The Consumer repeatedly extracts the piece of data from Mediation Buffer and provides its processing.
- The main idea of the pattern is to make sure that the Producer won't try to add data into the Buffer if it's full, and that the Consumer won't try to remove data from the Buffer if it’s empty.



Java (since 1.5) provides multiple ready to use synchronized containers for implementation of Producer-Consumer pattern.

See ***java.util.concurrent.BlockingQueue*** interface and derived classes