Content

- **Condition Synchronization**
- **Stopping threads: Interrupt & InterruptedException**

Condition Synchronization

- **Carpark Example**
  - Cars enter and exit a car park
  - Controller is required for the car park which only permits cars to enter when the car park is not full
  - Car arrival and departure are simulated by separate threads

```
public interface CarPark {
    void enter();
    void exit();
}
```

CarPark: Cars

```
public class Car extends Thread {
   private final CarPark carpark;
   public Car(String name, CarPark carpark) {
      super(name); this.carpark = carpark;
   }
   public void run() {
      while(true) {
         spendSomeTime(10); // drive around
         carpark.enter();   // wait until place is available
         spendSomeTime(20); // park in car park
         carpark.exit();    // exit car park
      }
   }
   private void spendSomeTime(int max) {
      try { sleep((int)(Math.random() * max * 1000)); }
      catch (InterruptedException e) {  }
   }
}
```

CarPark0

```
public class CarPark0 implements CarPark {
   private int places;

   public CarPark0(int places) { this.places = places; }

   public synchronized void enter() {
      if (places == 0) {
          throw new IllegalStateException("CarPark is full!");
      }
      places --;
   }

   public synchronized void exit() {
      places ++;
   }

}
```

CarPark1

```
public class CarPark1 implements CarPark {
   private int places;

   public CarPark1(int places) { this.places = places; }

   public synchronized void enter() {
      while(places == 0) {} // busy waiting
      places --;
   }

   public synchronized void exit() {
      places ++;
   }

}
```

CarPark2

```
public class CarPark2 implements CarPark {
   private int places;

   public CarPark2(int places) { this.places = places; }

   private synchronized boolean isFull() { return places == 0; }
   private synchronized void decPlaces() { places--; }
   private synchronized void incPlaces() { places++; }

   public void enter() {
      while(isFull()) {} // busy waiting
      decPlaces();
   }

   public void exit() {
      incPlaces();
   }
}
```

CarPark3

```
public class CarPark3 implements CarPark {
   private int places;
   public CarPark3(int places){ this.places = places; }
   private boolean isFull(){ return places == 0; }

   public void enter() {
      while (true) {
         synchronized (this) {
            if (!isFull()) {
               places--; return;
            }
         }
         sleep(10);
      }
   }

   public synchronized void exit() { places++; }
}
```

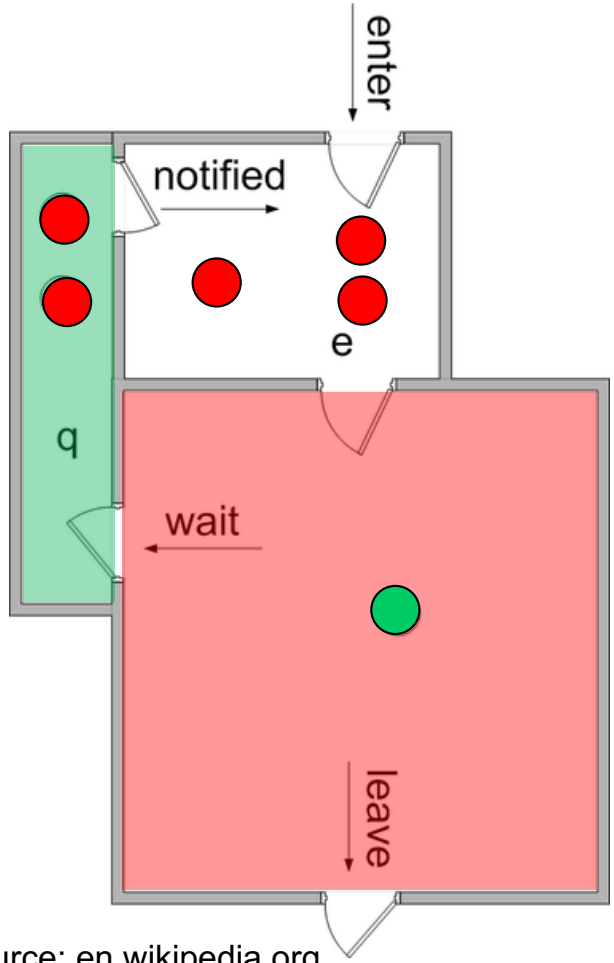
CarPark4

```
public class CarPark4 implements CarPark {
   private int places;

   public CarPark4(int places) { this.places = places; }

   public synchronized void enter() {
      while( places == 0) {
         try { wait(); }
         catch (InterruptedException e) { }
      }
      places--;
   }

   public synchronized void exit() {
      places++;
      notify();
   }
}
```

Condition Variables

Active thread
Waiting thread

- **Condition Variables**
  - Provide a means to wait until notified by another thread that some condition may now be true
  - Wait leaves critical region
    - Releases the lock
  - Notified threads
    - Must reacquire the lock
    - In competition with all other threads



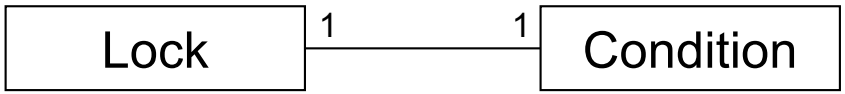
Source: en.wikipedia.org
© Theodore Norvell

Critical region (guarded using synchronized)
Waiting (waiting for a condition)
Blocked (waiting for the lock)

Condition Variables

- **Condition Variables**
  - Access to condition variable is only possible if the thread holds the synchronization lock
  - When a thread waits for a condition (enters the WAITING state), it automatically releases the synchronization lock

- **Condition Variables in Java**
  - Java: every Object
    - is a synchronization lock — synchronized(x) { … }
    - is a condition variable — x.wait() / x.notify()



Condition Variable: wait

- **wait()**

```
synchronized (q) {
   while(! condition )
      try{
         q.wait();
      } catch(InterruptedException e){ }
   }
   ...
}
```

  - Thread needs to have lock on q (otherwise: *IllegalMonitorStateException*)
    - Ensure that the lock is guarding the variables making up the condition!
  - Lock is released
    - To allow others to change the state and establish the condition
  - Thread enters the wait/notify queue of object q
  - Note use of while loop, don't use an if-statement there
    - Interrupts, spurious wakeups, notified but condition does not hold (anymore)

Condition Variable: notify

- **notify()**

```
synchronized (q) {
    ...;
    q.notify();
}
```

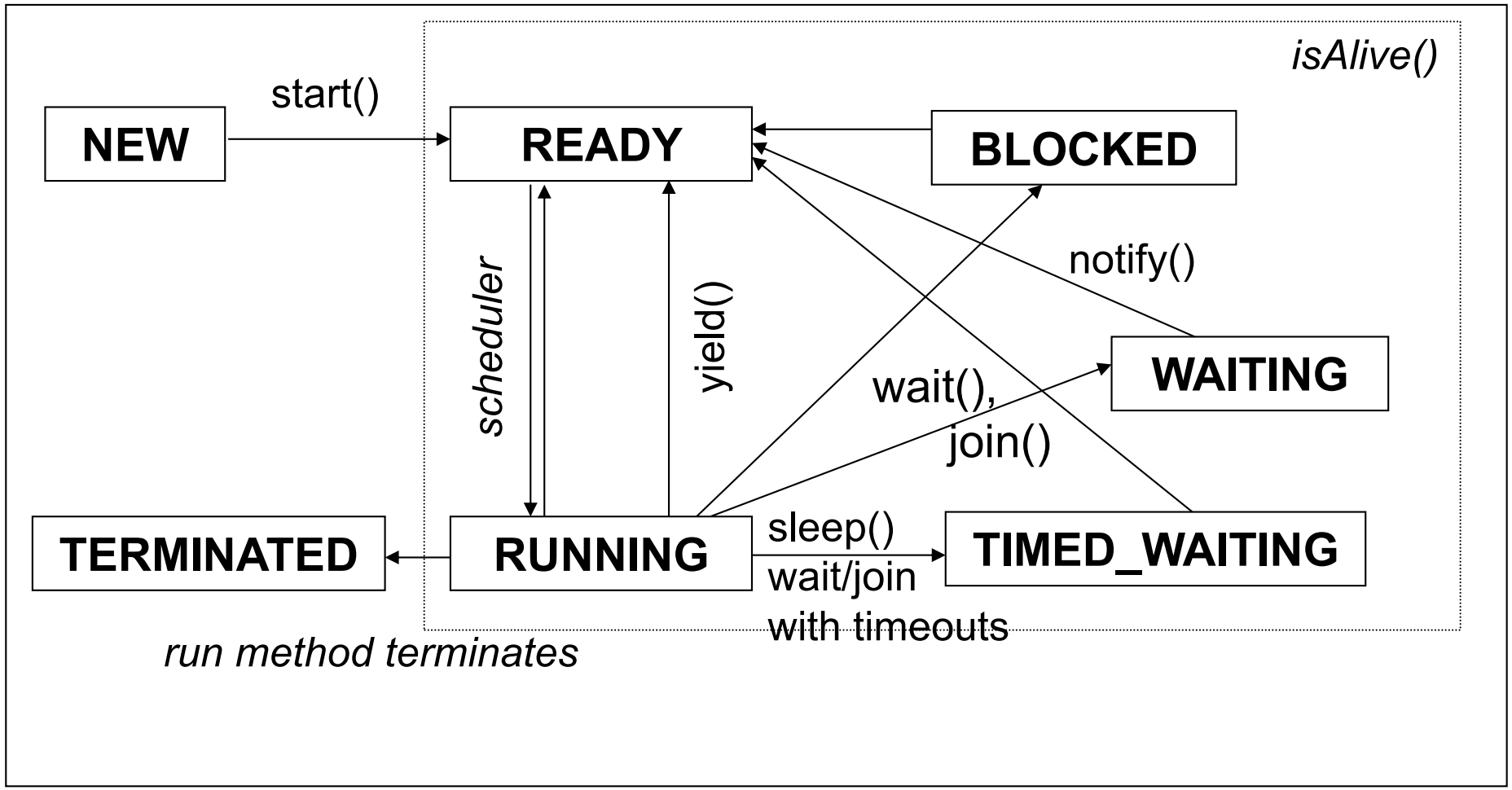
  - wakes up one waiting thread (which then must compete for the lock)
    - no control over which thread is woken up !
      - *The choice is arbitrary and occurs at the discretion of the implementation*
    - resumed thread must reacquire lock before continuing
  - if no threads are waiting: empty statement
  - thread needs to have lock on q (otherwise: *IllegalMonitorStateException*)

- **notifyAll( )**
  - wakes up all waiting threads which must then compete for the lock

CarPark5

```
public class CarPark5 implements CarPark {
   private int places;
   public CarPark5(int places) { this.places = places; }
   private final Object lock = new Object();
   public void enter() {
      synchronized(lock) {
         while( places == 0) {
            try { lock.wait(); }
            catch (InterruptedException e) { }
         }
         places--;
      }
   }
   public void exit() {
      synchronized(lock) {
         places++; lock.notify();
      }
   }
}
```

Thread State



Thread State

- **Enum Thread.State**
  - NEW
  - RUNNABLE *(READY or RUNNING)*
  - BLOCKED *waiting for a monitor lock*
  - WAITING *waiting on a condition or for termination*
  - TIMED_WAITING *waiting in sleep or in a timed wait/join*
  - TERMINATED
- **Thread class**
  - getState() *monitoring purposes*

General Structure

Guarantee stable state
- during condition checking and waiting
- between checking and acting

by guarding all variables which are part of the precondition with the **same lock!**

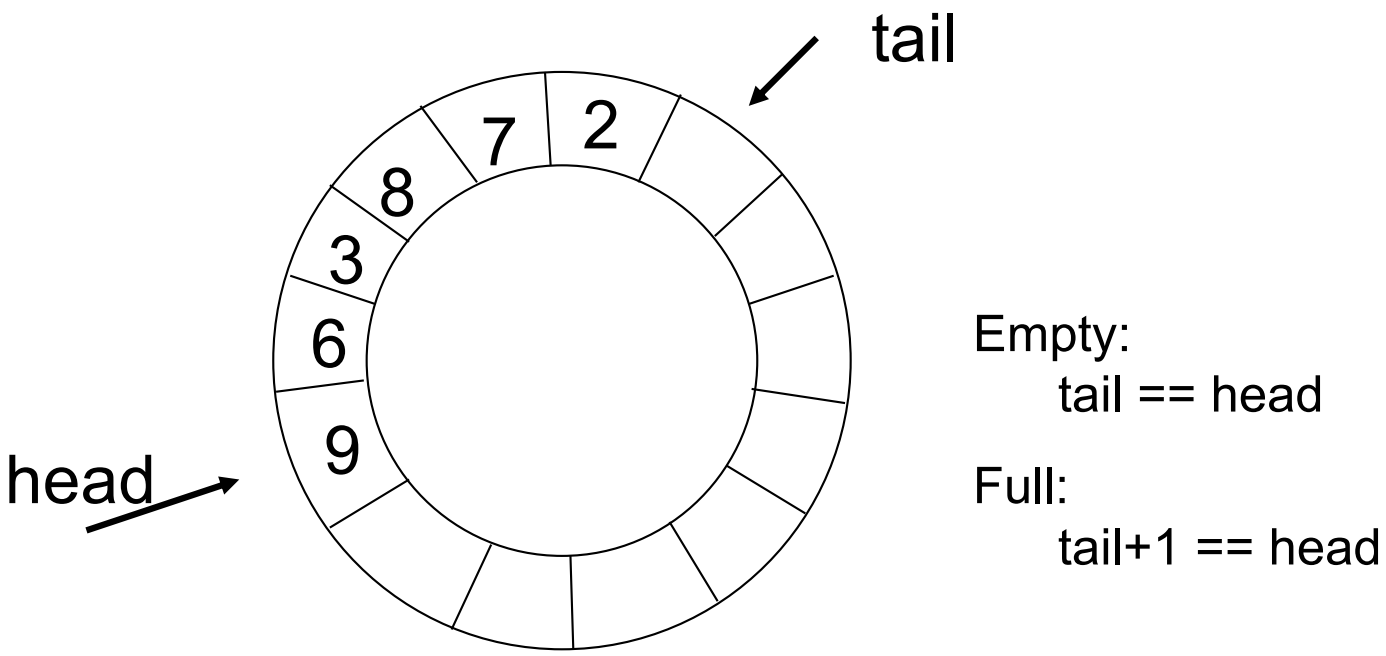
`notify` can be used instead of `notifyAll` if
- only one thread can pass and
- all threads wait on the same condition (uniform waiters)

```
//PRE for act: P (z.B. a>b)
public void m() {
   synchronized(lock) {
      while(! P) {
         try { lock.wait(); } //wait
         catch(InterruptedException i) {}
      }
      //P holds here!
      act();
      lock.notifyAll(); // only if P was changed by act
   }
}
public void setA(int a) {
   synchronized(lock) {
      //update a
      lock.notifyAll();
   }
}
```

```
public void setB(int b) {
   synchronized(lock) {
      //update b
      lock.notifyAll();
   }
}
```

Example: Producer/Consumer - Queue

- **Queue**
  - public void enqueue(Object e); // add elements at tail of queue
  - public Object dequeue( ); // remove elements at head of queue



Example: Producer/Consumer - Queue

```
public class Queue0 {
    private final static int SIZE = 10;
    private Object[] buf = new Object[SIZE];
    private int head, tail = 0;

    public synchronized Object dequeue() {
        while (tail == head) { // while empty
            try {
                this.wait(); // wait until not empty
            } catch (InterruptedException e) { /* ignore */}
        }
        this.notify(); // notify those waiting for not full
        Object e = buf[head]; buf[head] = null; // free garbage
        head = (head + 1) % SIZE;
        return e;
    }
```

Example: Producer/Consumer - Queue

```
    public synchronized void enqueue(Object c) {
        while ((tail + 1) % SIZE == head) { // while full
            try {
                this.wait(); // wait until not full
            } catch (InterruptedException e) { /* ignore */}
        }
        this.notify(); // notify those waiting for not empty
        buf[tail] = c;
        tail = (tail + 1) % SIZE;
    }
}
```

Rules for wait / notify

- **notify() vs. notifyAll()**
  - notify()
    - Uniform waiters: All waiters are equal (wait for the same condition)!
    - One-in, one-out: A notification on the condition variable enables at most one thread to proceed
  - notifyAll() is much safer (but less efficient)

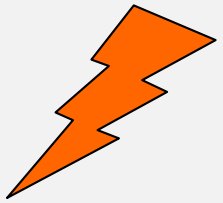
- **Solutions for Queue**
  - call notifyAll instead of notify
  - use separate objects for the two (different) conditions (according to the colors in the code)
    - notFull
    - notEmtpy

Example: Producer/Consumer - Queue

```
public class Queue1 {
    private final static int SIZE = 10;
    private Object[] buf = new Object[SIZE];
    private int tail = 0, head = 0;
    private Object notEmpty = new Object();
    private Object notFull = new Object();

    public synchronized Object dequeue() {
        while (tail == head) { // while empty
            synchronized (notEmpty) {
                try { notEmpty.wait(); } catch (Exception e) {}
            }
        }
        synchronized (notFull) { notFull.notify(); }
        Object e = buf[head]; head = (head + 1) % SIZE;
        return e;
    }
```

Example: Producer/Consumer - Queue

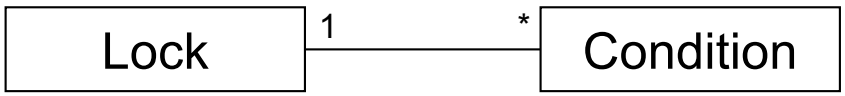
```
 public synchronized void enqueue(Object c) {
        while ((tail + 1) % SIZE == head) {
            synchronized (notFull) {
                try { notFull.wait(); } catch (Exception e) {}
            }
        }
        synchronized (notEmpty) { notEmpty.notify(); }
        buf[tail] = c;
        tail = (tail + 1) % SIZE;
    }
}
```



java.util.concurrent.locks.Lock

- **Several conditions may be associated with a lock**
- **Interface**

```
public interface Lock {
   void lock();
   void unlock();


   boolean tryLock();
   boolean tryLock(long timeout, TimeUnit unit)
                                  throws InterruptedException;

   Condition newCondition();
   ...
}
```



java.util.concurrent.locks.Condition

- **Interface Condition implements condition variables**
  - `await` waits until signaled or interrupted
  - `await(long timeout, TimeUnit unit)` waits until signaled, interrupted or time elapses
  - `signal` wakes up one waiting thread
  - `signalAll` wakes up all waiting threads
  - `...`

- **Condition and lock**
  - await* / signal* can only be called if thread owns the corresponding lock
  - await releases the lock, and puts the thread into the signal's wait queue
  - Spurious wakeups are possible => while loop
  - Implementations may provide *fair* wait queues

Queue with j.u.c.l.Condition

```
public class Queue2 {
    private final static int SIZE = 10;
    private final Object[] buf = new Object[SIZE];
    private int tail = 0, head = 0;
    private final Lock lock = new ReentrantLock();
    private final Condition notEmpty = lock.newCondition();
    private final Condition notFull = lock.newCondition();

    public Object dequeue() {
        lock.lock();
        try {
            while (tail == head) { // while empty
                try { notEmpty.await(); } catch (Exception e) {}
            }
            Object e = buf[head]; head = (head + 1) % SIZE;
            notFull.signal(); return e;
        } finally { lock.unlock(); }
    }
```

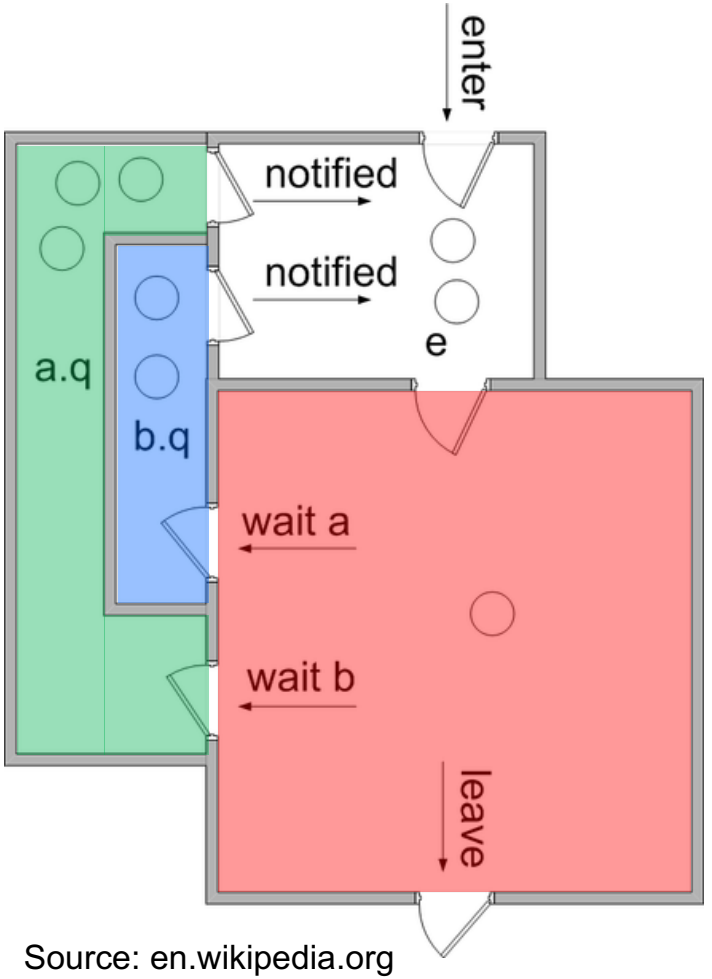
Queue with j.u.c.l.Condition

```
    public void enqueue(Object c) {
        lock.lock();
        try {
            while ((tail + 1) % SIZE == head) {
                try { notFull.await(); } catch (Exception e) {}
            }
            buf[tail] = c; tail = (tail + 1) % SIZE;
            notEmpty.signal();
        } finally {
            lock.unlock();
        }
    }
}
```

j.u.c.l.Conditions and Locks Illustrated

- **Multiple conditions per lock**
  - Each with a separate wait queue
  - Each wait queue can be signaled separately

Critical region
Wait queue a
Wait queue b



Source: en.wikipedia.org
© Theodore Norvell

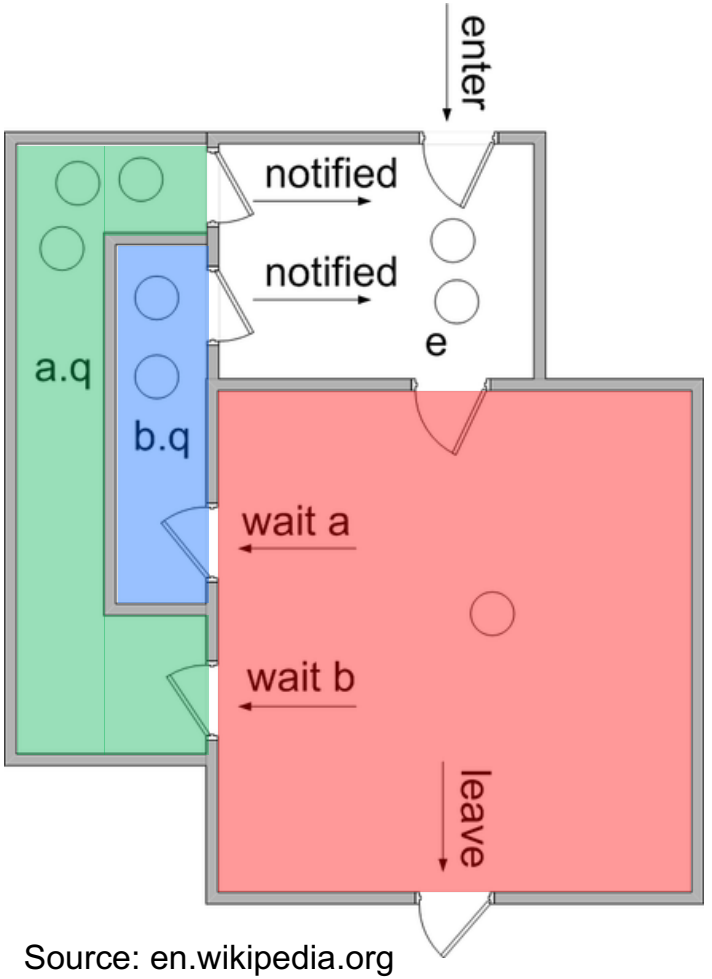
Monitor

- **History**
  - Monitors were invented 1974 by C.A.R. Hoare and Per Brinch Hansen

- **Monitor characteristics**
  - Methods of a monitor are executed with mutual exclusion, i.e. at each point in time at most one thread may be executing any of its methods
    - *Java's Insecure Parallelism*, P.B. Hansen, ACM SIGPLAN Notices 4/99
      - In Java methods have to be declared as synchronized
  - Monitors provide a mechanism for threads to temporarily give up exclusive access in order to wait for some condition to be met, before regaining exclusive access and resuming their task.
    Monitors also have a mechanism for signaling other threads that such conditions have been met
    - This characteristic is available for every Java object

Monitor

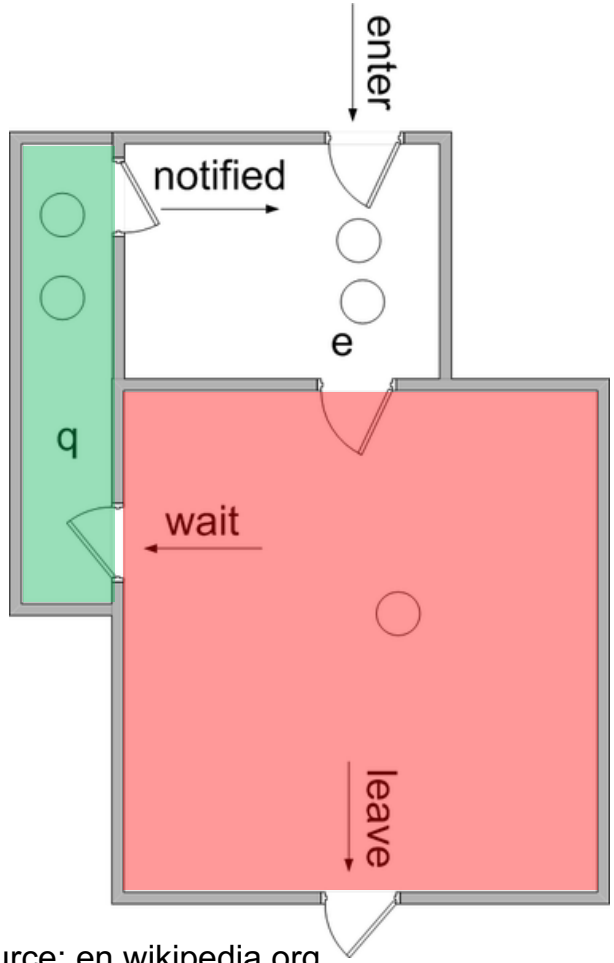
- **Nonblocking condition variables**
  - Mesa style monitor
    (Mesa is a programming language)
  - Signaling thread does not lose
    the monitor (i.e. is not blocked)
  - Typically a notifyAll is provided



Source: en.wikipedia.org
© Theodore Norvell

Monitor

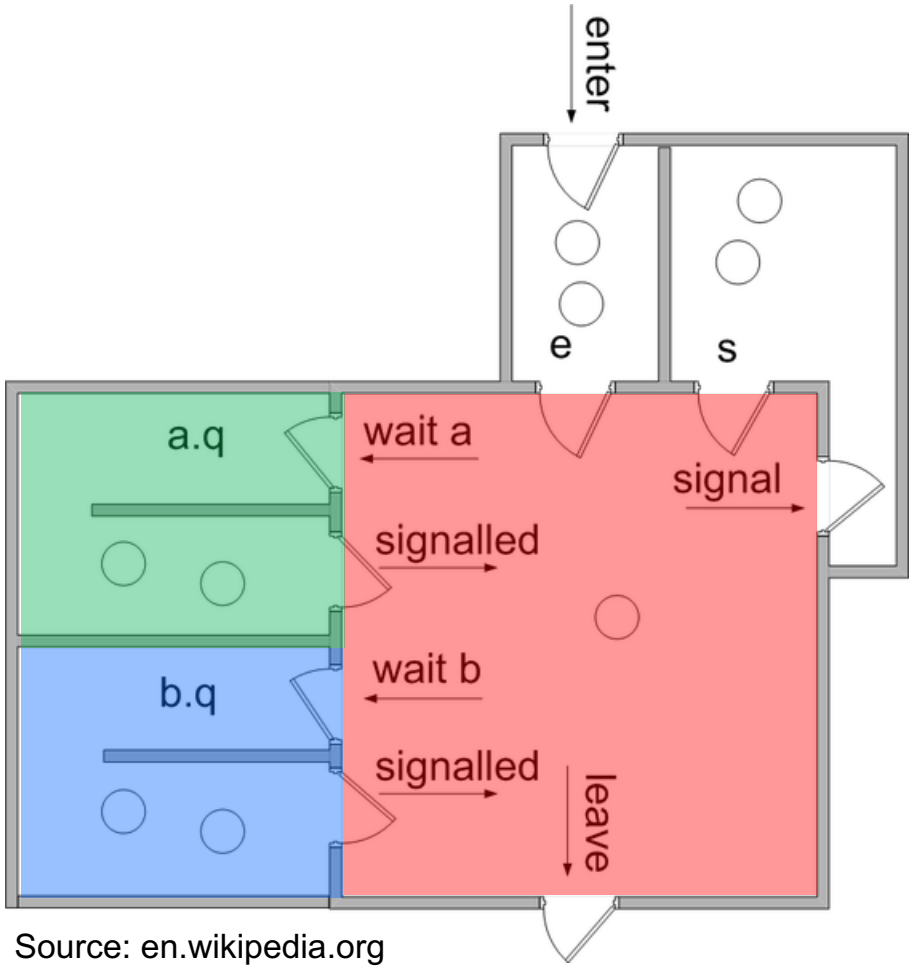
- **Implicit condition monitor**
  - Java style monitor
  - Every object (implicitly) contains
    - One built-in lock
    - One wait queue
  - Adopted in C#



Source: en.wikipedia.org
© Theodore Norvell

Monitor

- **Blocking condition variables**
  - Hoare style monitor
  - Signaling thread must wait outside the monitor (at least) until the signaled thread relinquishes occupancy of the monitor by either returning or by again waiting on a condition
  - Wait queue s has priority over wait queue e
  - A notifyAll cannot be provided



Source: en.wikipedia.org
© Theodore Norvell

Content

- **Condition Synchronisation**
- **Stopping threads: Interrupt & InterruptedException**

Blocking Methods

- **Blocking methods can potentially take forever if the event they are waiting for never occurs**
  - Blocking operations should be cancelable
    - E.g. waiting for an object to become available (blocking queue)
  - Long-running non-blocking methods should be cancelable as well
    - E.g. Mandelbrot computation

- **Cancel Operations (on thread objects)**
  - stop()
  - interrupt()

Stop Method

- **Method stop is declared deprecated as it is inherently unsafe**
  - Thread.stop causes to unlock all the monitors that the thread has locked
  - If any of the objects previously protected by these monitors were in an inconsistent state, the damaged objects become visible to other threads, potentially resulting in arbitrary behavior
  - Threads should be stopped in a cooperative fashion => interrupt

  - Reference
    - http://java.sun.com/javase/6/docs/technotes/guides/concurrency/threadPrimitiveDeprecation.html

Interrupt Flag

- **Interrupt Flag**
  - Every thread has a boolean interrupted status property
- **Setting the flag: *interrupt()***
  - If the thread is blocked in an invocation of wait, sleep or join statement, an *InterruptedException* is thrown
  - Otherwise the thread's interrupt flag is set
  - Interrupt flag is cleared before the InterruptedException is thrown
  - If interrupt flag is set, a subsequent wait / sleep / join call immediately throws an InterruptedException
- **Reading the flag: *isInterrupted()***
  - flag can be read with the *isInterrupted* method
    - instance method of class Thread
  - flag can be read **and cleared** with *Thread.interrupted* (poor name!)
    - static method of class Thread

Interrupt Flag Example

- **Quiz: What is the output?**

```
Thread.currentThread().interrupt();
System.out.println(Thread.interrupted());
try {
   Thread.sleep(1000); System.out.println("ok1");
} catch (InterruptedException e) {
   System.out.println("IE: " +
                Thread.currentThread().isInterrupted());
}
Thread.currentThread().interrupt();
System.out.println(Thread.currentThread().isInterrupted());
try {
   Thread.sleep(1000); System.out.println("ok2");
} catch (InterruptedException e) {
   System.out.println("IE: " +
                Thread.currentThread().isInterrupted());
}
```

Handling the InterruptedException

- **InterruptedException: What should we do?**

```
try{
   …
} catch(InterruptedException e) {
   // ??? /* help! */
}
```

- **Possible reactions:**
  1. Ignore the exception
  2. Propagate the exception
  3. Defer the exception

Handling the InterruptedException

- **Ignore the exception**
  - Possible, if it can be asserted that interrupt is never called on a thread
    - E.g. a local non-accessible thread class
  - Possible if thread must not be interrupted
    - E.g. essential services
  - Example: Semaphore.acquire

```
public synchronized void acquire() {
   while(value <= 0) {
      try {
         wait();
      } catch(InterruptedException e) {
      }
   }
   value--;
}
```

Handling the InterruptedException

- **Propagate the exception**
  - Declare your method throwing an InterruptedException as it is a blocking method as well
  - Simple cleanup possible in the exception handler before rethrowing

```
public synchronized void enqueue(Object c)
                                    throws InterruptedException {
   while ((first + 1) % SIZE == last) { // while full
      try {
         this.wait(); // wait until not full
      } catch ( InterruptedException e ) {
         /* cleanup */ throw e;
      }
   }
   buf[first] = c;
   first = (first + 1) % SIZE;
   this.notifyAll(); // notify those waiting for not empty
}
```

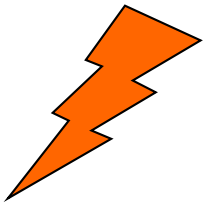
Handling the InterruptedException

- **Defer the exception**
  - Throwing InterruptedException is not always an option (perhaps your task is defined by a `Runnable`)
  - Restore the interruption status:
    - Fact that InterruptedException was thrown should be preserved by setting the interrupt flag
    - => code higher up on the call stack can handle interrupt
  - Interruption (in a blocking or non-blocking operation) can then be detected (and handled) with isInterrupted

```
try {
   /* ... */
} catch (InterruptedException e) {
   // Restore the interrupted status
   Thread.currentThread().interrupt();
}
```

Interrupt & notify

- **Lost Signals**
  - When a thread is notified with notify simultaneously to an interrupt, the signal may be lost!
    - Threads t1 and t2 are waiting in a wait()
    - Thread t3 performs a notify => t1 is selected
    - Thread t4 interrupts t1
    - => wait called by t1 throws InterruptedException
      => t1 does not process notification
      => t2 does not wake up



=> Handler for the InterruptedException must consider this possibility

- **Solution 1:**
  - Use notifyAll instead

Interrupt & notify

- **Solution 2**
  - Invoke `notify()` from within the exception handler in order to wake up another waiting thread

```
synchronized(this) {
   try {
      while( !cond ) {
         wait();
      }
      assert(cond);
      // perform task
   } catch( InterruptedException e ) {
      notify();
      throw e;
   }
}
```

Example 1: Stopping a Thread

```
class Worker extends Thread {
   public void run( ) {
      while( !isInterrupted() ) { // test only
         // do work
         try {
            ...
         }
         catch(InterruptedException e){
            Thread.currentThread().interrupt();
         }
      }
      // clean up
   }
}
```

Example 2: CarPark4 (2nd version)

```
public class CarPark4 implements CarPark {
   private int places;

   public CarPark4(int places){ this.places = places; }

   public synchronized void enter() throws InterruptedException {
      while( places == 0) {
         try { wait(); }
         catch (InterruptedException e) { notify(); throw e; }
      }
      places--;
   }

   public synchronized void exit() {
      places++; notify();
   }
}
```

Cars may be interrupted while waiting (i.e. may turn away from the entry queue)

Example 3: java.util.concurrent.locks.Lock

- **Interruptible lock acquisition**
  - Allow to use locking in cancellable activities (the default lock operation is non-interruptibly)
  - Caution: InterruptedException has to be handled, i.e. *two try blocks are necessary* (except when Exception can be propagated)

```
try {
   lock.lockInterruptibly();
   try {
      /* access resources protected by this lock */
   } finally {
      lock.unlock();
   }
} catch( InterruptedException e) {
   // handle interrupt
}
```

Content

- **Condition Synchronization**
  - JCIP: Chapter 14 (available on AD)
- **Stopping threads: Interrupt & InterruptedException**
  - JCIP: Chapter 7 (available on AD)

- **Assignment 3**
  - Implementation of a fair Semaphore

```
public interface Semaphore {
   int available();
   void acquire();     // P()
   void release();     // V()
}
```