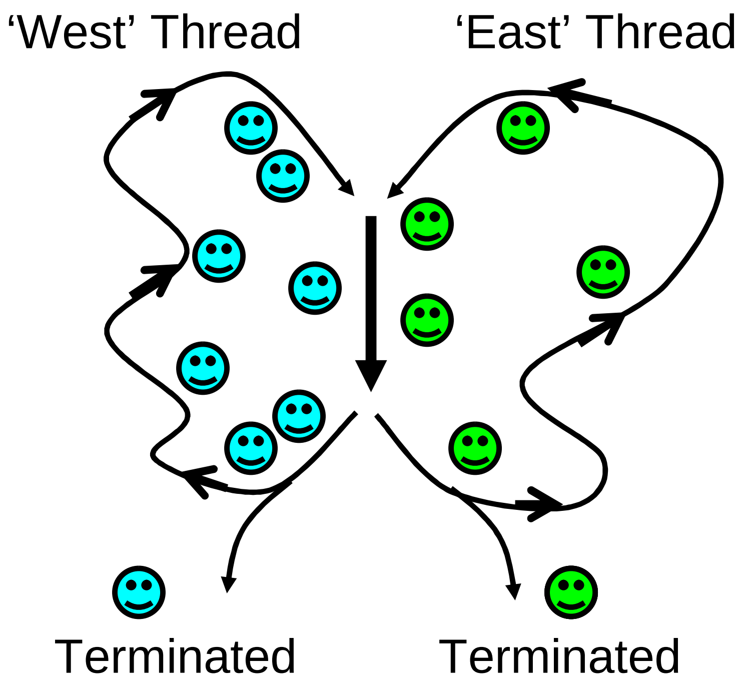
Concurrent Programming (Part II) Lecture 5: Monitors & Conditional Synchronization

Dr Kevin Bryson
K.Bryson@cs.ucl.ac.uk
Room 8.04

Course Web Site on Moodle
http://moodle.ucl.ac.uk/course/view.php?id=753
Enrolment Key: ATOMIC

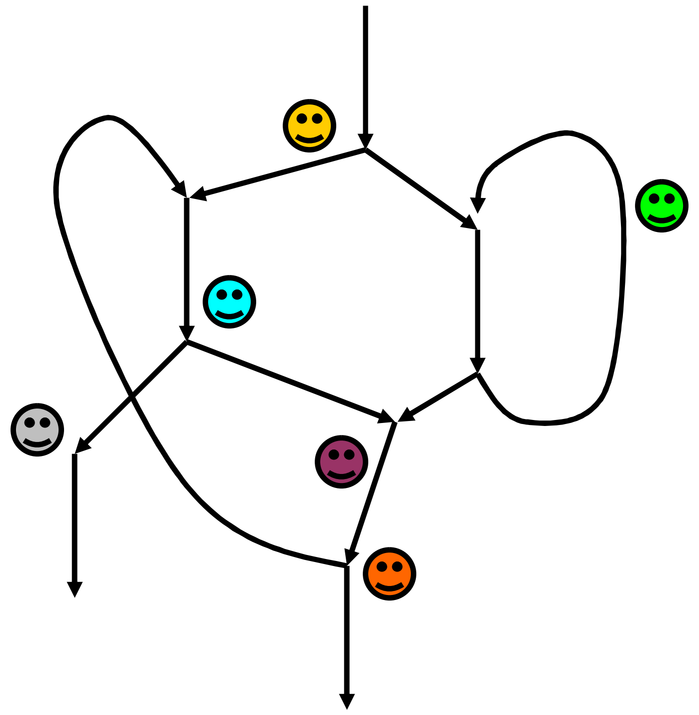
Recall from the last lecture …

- A lot of the examples we examine are on the ‘small scale’ – for instance the last lecture examined the Ornamental Garden with two Threads ‘interfering’ within the increment() method of the Counter object.



- Recall the threads will be at different places on their ‘paths of execution’ …
- They could be ‘timed’ not to interfere – but this is hardly a robust solution !!!
- A more robust solution is to ‘**synchronize**’ **critical sections** of code so that only one Thread is allowed to traverse them at a time.

Getting a feeling for Multi-Threaded Programs

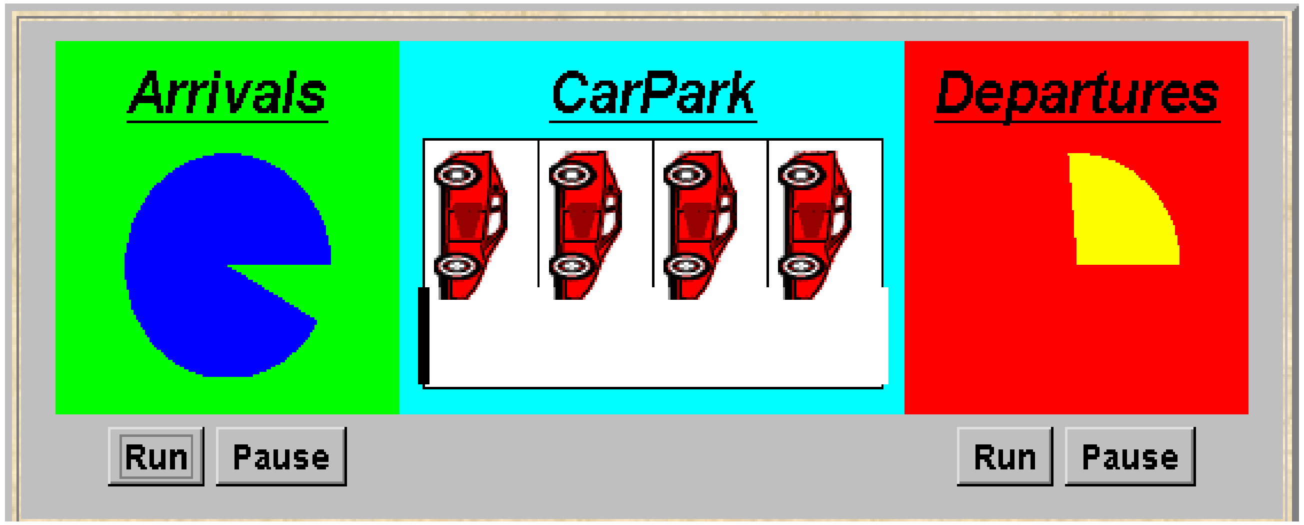


- But real concurrent programs tend to be much more complex than this
    - More complex paths
    - Many more Threads
    - *Shared data* accessed at different locations on these paths.
- We need strict discipline and **design patterns** to design efficient and robust multi-threaded applications.
- **In this lecture we will examine some very common design patterns within Java concurrent applications:**
    - **Monitors**
    - **Conditional Synchronization**

What do we mean by a Monitor and Conditional Synchronization?

- The term Monitor is used inconsistently within different threading contexts & languages …
- We will employ the Magee & Kramer definition - a Monitor is first and foremost a **data abstraction mechanism**.
- A Monitor encapsulates the representation of an abstract object and provides a set of operations that are the only means to manipulate the representation.
- A Monitor only allows one Thread to modify the representation at a time.
- Monitors can support Conditional Synchronization by only allowing methods to be run when the encapsulated representation is in a particular state (**guarded actions**).

Car Park Example
(Magee & Kramer Chapter 5)



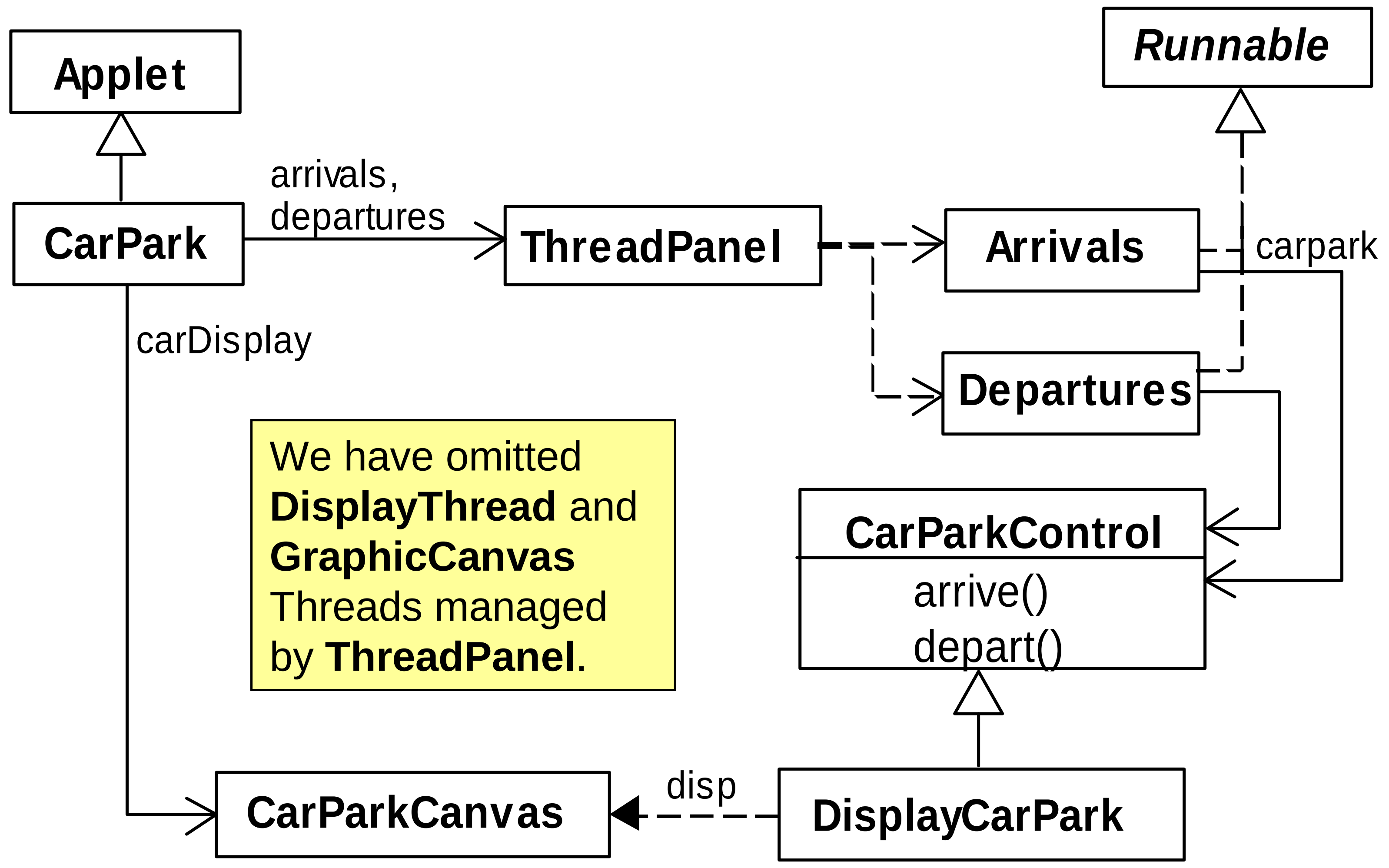
A controller is required for a car park, which only permits cars to enter when the car park is not full and does not permit cars to leave when there are no cars in the car park. Car arrival and departure are simulated by separate threads.

Modelling a Concurrent System in Terms of Threads and Monitors

- A concurrent system can be modelled in terms of:
    - Active Entities
    - Passive Entities that the active entities operate on.
- In a similar manner to breaking a system into objects, there are many different ways of expressing a concurrent system as active and passive entities – however some are more natural than others.
- Java Threads can be used to model the Active Entities.
- Monitors can be used to model the Passive Entities.

How do you think the Car Park Demo should be modelled ?

Car Park Example – Class Diagram



We have omitted **DisplayThread** and **GraphicCanvas** Threads managed by **ThreadPanel**.

Car Park Example – `starting` the Threads

`Arrivals` and **`Departures`** implement **`Runnable`**, **`CarParkControl`** provides the control

Instances of these are created by the **`start()`** method of the **`CarPark`** **object:**

```
public void start() {
  CarParkControl c =
     new DisplayCarPark(carDisplay,Places);
  arrivals.start(new Arrivals(c));
  departures.start(new Departures(c));
}
```

The Arrivals and Departures Threads

```
class Arrivals implements Runnable {
  CarParkControl carpark;

  Arrivals(CarParkControl c) {carpark = c;}

  public void run() {
    try {
      while(true) {
        ThreadPanel.rotate(330);
        carpark.arrive();
        ThreadPanel.rotate(30);
      }
    } catch (InterruptedException e){}
  }
}
```

Similarly **`Departures`** which calls **`carpark.depart()`**.

But how do we implement the control of **`CarParkControl`**?

The shared `CarParkControl` Monitor

```
class CarParkControl {
  protected int spaces;
  protected int capacity;

  CarParkControl(int n)
    {capacity = spaces = n;}

  synchronized void arrive() {
    …  --spaces; …
    }

  synchronized void depart() {
    … ++spaces; …
    }
}
```

mutual exclusion by synchronized methods

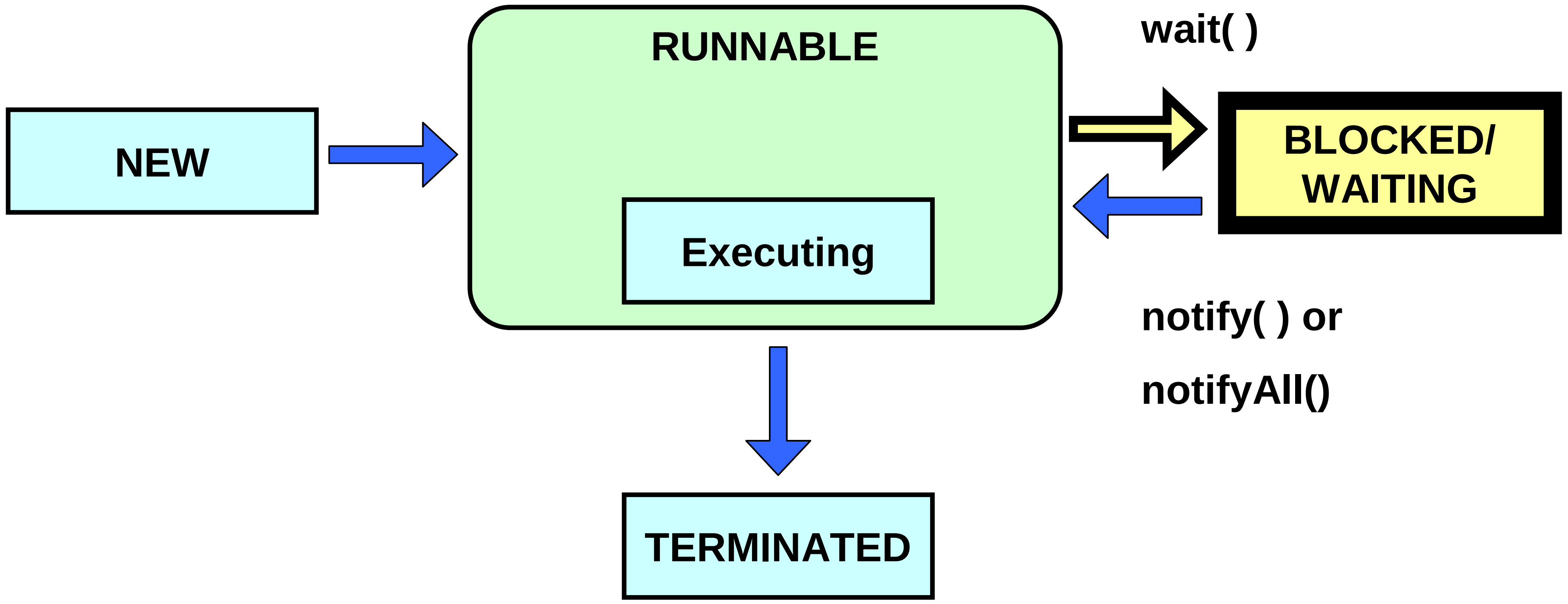
block if full? (spaces==0)

block if empty? (spaces==N)

This is called ***condition synchronization***

Recall the Thread Lifecycle from Lecture 3 …

The wait() method puts a Thread into a ‘WAITING’ state until another Thread calls notify() or notifyAll() to activate it again



How is Condition Synchronization implemented in Java?

All java objects implement the following methods – they can only be used within a synchronized section when the Thread is holding the object lock:

```
public final void wait()
                        throws InterruptedException
```

Waits to be notified by another thread. The waiting thread releases the synchronization lock associated with the monitor. When notified, the thread must wait to reacquire the lock before resuming execution.

```
public final void notify()
```

Wakes up a single thread that is waiting for the condition

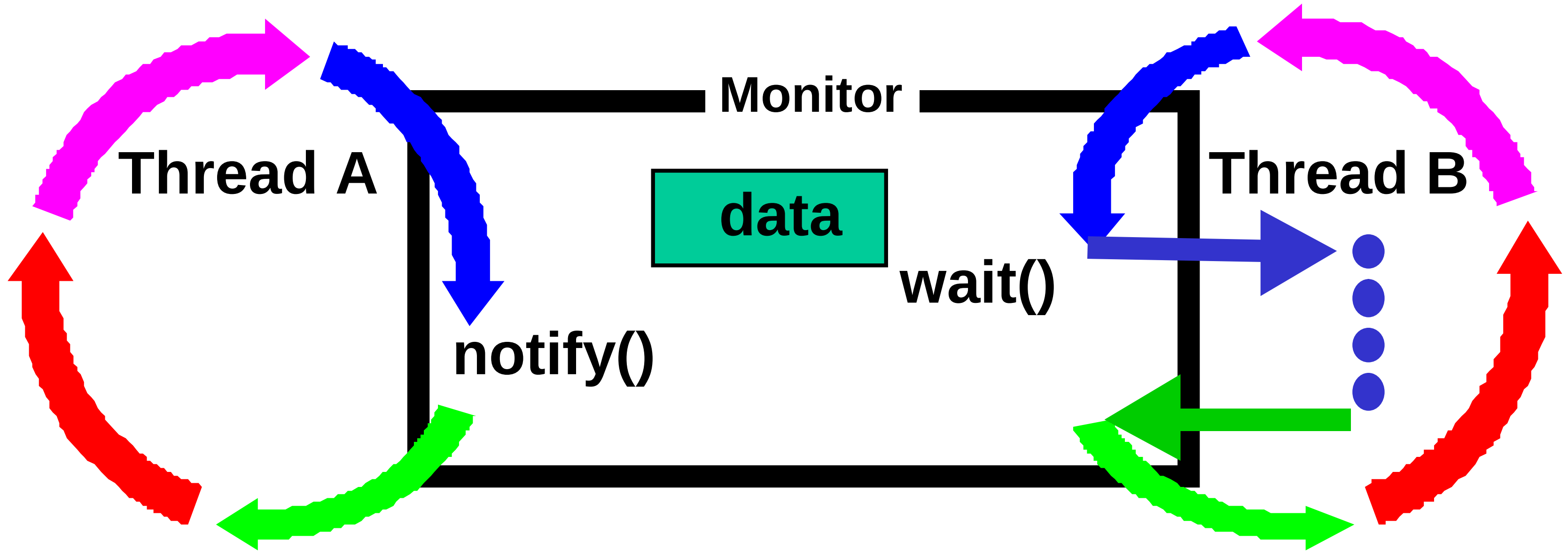
```
public final void notifyAll()
```

Wakes up all threads that are waiting for the condition

Condition Synchronization – graphical view

We refer to a thread ***entering*** a monitor when it acquires the mutual exclusion lock associated with the monitor and ***exiting*** the monitor when it releases the lock.

wait() - causes the thread to exit the monitor, permitting other threads to enter the monitor.



Condition Synchronization in Java

```
public synchronized void do_lots_work()
                     throws InterruptedException
{
   while (!cond) wait();
   // modify monitor data
   notifyAll();
}
```

The **while** loop is necessary to retest the condition *cond* to ensure that *cond* is indeed satisfied when the Thread re-enters the monitor.

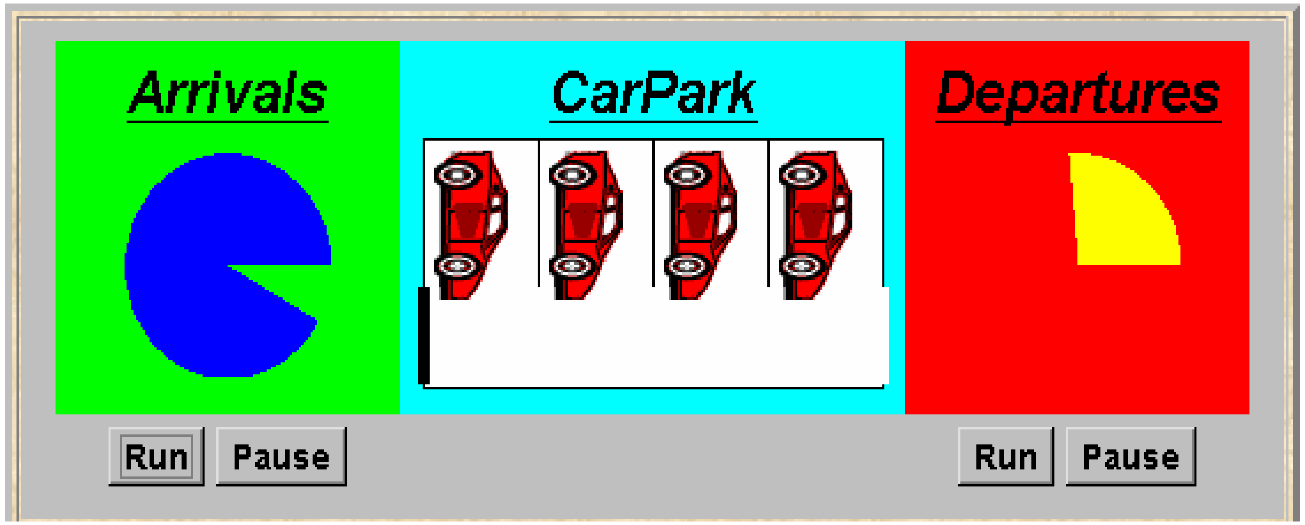
notifyAll() is necessary to awaken other thread(s) that may be waiting to enter the monitor now that the monitor data has been changed.

CarParkControl - condition synchronization

```
class CarParkControl {
  protected int spaces;
  protected int capacity;

  CarParkControl(int n)
    {capacity = spaces = n;}

  synchronized void arrive() throws InterruptedException {
    while (spaces==0) wait();
    --spaces;
    notifyAll();
  }

  synchronized void depart() throws InterruptedException {
    while (spaces==capacity) wait();
    ++spaces;
    notifyAll();
  }
}
```

Why would it be safe **IN THIS SPECIFIC CASE** to use **`notify()`** rather than **`notifyAll()`**?

How does the Applet behave?



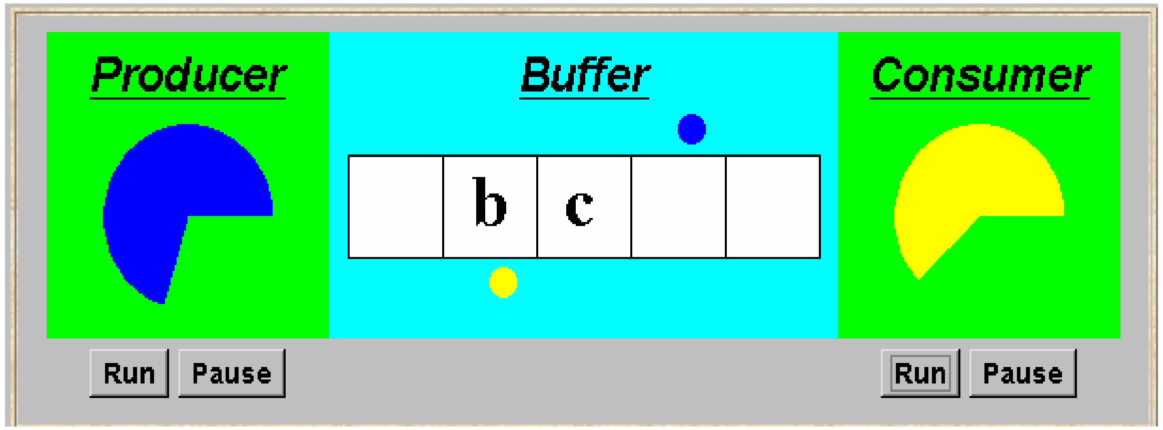
Well let’s see …

But how often will you be asked to write a Car Park Controller?

It should be clear than this is a general concurrent design pattern. Any situation which may need to wait for data to be in a particular state can use Condition Synchronization, often within a Monitor.

A very common example in computer science is …

Bounded Buffer



A bounded buffer consists of a fixed number of slots. Items are put into the buffer by a *producer* process and removed by a *consumer* process. It can be used to smooth out transfer rates between the *producer* and *consumer* (compare with car park example)

For example – a LINUX pipe

Let's see it …

Bounded buffer program - Buffer Monitor

```
class BufferImpl<E> implements Buffer<E> {
     …

 public synchronized void put(E o)
            throws InterruptedException {
    while (count==size) wait();
    buf[in] = o; ++count; in=(in+1)%size;
    notifyAll();
  }

  public synchronized E get()
            throws InterruptedException {
    while (count==0) wait();
    E o =buf[out];
    buf[out]=null; --count; out=(out+1)%size;
    notifyAll();
    return (o);
    }

}
```

UCL

Bounded buffer program - Producer Process

```
class Producer implements Runnable {
  Buffer buf;
  String alphabet= "abcdefghijklmnopqrstuvwxyz";

  Producer(Buffer b) {buf = b;}

  public void run() {
    try {
      int ai = 0;
      while(true) {
        ThreadPanel.rotate(12);
        buf.put(alphabet.charAt(ai));
        ai=(ai+1) % alphabet.length();
        ThreadPanel.rotate(348);
      }
    } catch (InterruptedException e){}
  }
}
```

Similarly `Consumer` which calls `buf.get()`.

Monitor Summary

Active entities (that initiate actions) are implemented as **Threads**.
Passive entities (that respond to actions) are implemented as **Monitors**.

Each guarded action in the model of a monitor is implemented as a `synchronized` method which uses a while loop and `wait()` to implement the guard. The while loop condition is the negation of the model guard condition.

Changes in the state of the monitor are signalled to waiting threads using `notify()` or `notifyAll()`.

So far we have described the “traditional” Java concurrency mechanisms …

- With Java 5, a whole new sophisticated package dedicated to concurrency was introduced (**java.util.concurrent**).
- This package duplicates all the functionality of the traditional concurrency mechanisms … but also adds a lot of new concurrency features.
- However, this package **is more dangerous** to use compared to the traditional methods.
- Thus you should only use this package when you need to use functionality which is not available within the traditional mechanism.

Java 5 Explicit Lock Interface

```
package java.util.concurrent.lock;

public Interface Lock {

    // Lock – with blocking.
    void      lock(); // Lock.
    void      lockInterruptibly()
                        throws InterruptedException;

    // Lock – no blocking or timed blocking.
    boolean   tryLock();
    boolean   tryLock(long time, TimeUnit unit)
                        throws InterruptedException;

    void      unlock();       // Unlock
    Condition newCondition(); // Covered later ...
}
```

How to do locking with new package - Example from Ornamental Garden Counter

```
class Counter {
    ...
    private Lock counterLock = new ReentrantLock();

    void increment() {
        try {
            counterLock.lock();
            int temp = value;    //read value
            Simulate.HWinterrupt();
            value=temp+1;        //write value
        } finally {
            counterLock.unlock();
        }
        display.setvalue(value);
    }
}
```

What are the disadvantages ?

What are the benefits ?

How to do Condition Synchronization with new package - Example for CarParkControl

```
class CarParkControl {
  protected int       spaces;
  protected int       capacity;
  private   Lock      park_lock        = new ReentrantLock();
  private   Condition spaces_condition = park_lock.newCondition();
  ...
  void arrive() throws InterruptedException {
    try {
       park_lock.lock();

       while (spaces==0) spaces_condition.await();
       --spaces;
       spaces_condition.signal();
    }
    finally {
       park_lock.unlock();
    }
  }
    ...
}
```

Looks more complex – *Why bother?*

1. Part of a much larger coherent framework with much more sophisticated concurrency facilities
2. Locks are explicitly represented:
   a) Can have different locks within the same object
   b) Can share locks between different objects
3. New lock interface allows a Thread to do ‘other things’ while waiting for a lock to be released
4. New lock interface allows a Thread to wait for only a certain amount of time before continuing
5. “Conditional variables” are explicitly represented
   a) Thus can have multiple conditions for one particular lock to signal different types of changes
   b) For example, two conditional variables seem suitable for the Car Park example: car park empty and car park full.

Summary

- We looked at modelling concurrent systems as active objects (Threads) and passive objects (**Monitors**).
- Monitors provide a higher level concurrent design pattern that can **guard shared variables** from **interference** within a multi-threaded system.
- Monitors can support **Conditional Synchronization**.
- We introduced the new Java 5 concurrency package. This has explicit support for:
  - Multiple locks
  - Multiple conditional variables