Università
della
Svizzera
italiana

**Facoltà
di scienze
informatiche**

Programming Fundamentals 3

Concurrency in Java, Part 2

Walter Binder

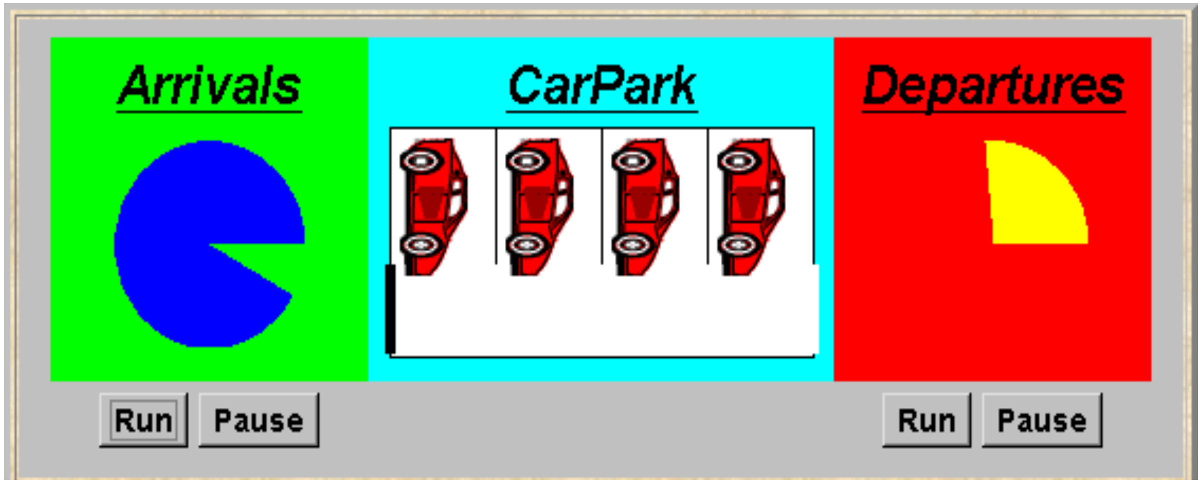
Questions (1)

- How are threads created in Java?
- What is the difference between `sleep` and `yield`?
- Are variable assignments guaranteed to be atomic and immediately visible to all threads?

Questions (2)

- Is the keyword **`synchronized`** part of the signature of a method?
- If method f1 of a class is synchronized and f2 is not, can the two execute in parallel?
- If non-static method f1 and static method f2 of a class are synchronized, can they execute in parallel?

Cooperating Threads



- A controller is required for a carpark
- Cars can enter when there is space available
- Cars cannot leave when the carpark is empty
- Car arrival and departure are simulated by separate threads

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 synch methods

block if full? (spaces==0)

block if empty? (spaces==capacity)

Object Synchronization

- Solutions to coordination
  - Enter a loop, constantly checking if the condition is true
  - Use explicit Object-based synchronization

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

Wakes up a single thread that is waiting on this object's lock

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

Wakes up all threads that are waiting on this object's lock

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

Waits to be notified by another thread. The waiting thread releases this object's lock. When notified, the thread must first re-acquire the lock before resuming execution

Using `wait()`

Thread must hold the lock

```
synchronized (obj) {
  while (! <condition>) {
    try {
      obj.wait(timeout);
    catch (InterruptedException e) {..}

  ... // Perform action
}
```

Wait either for a timeout or for a notification

When notification arrives, waiting thread is *put back into ready queue* with all other threads, and MUST check condition again before acting.

```
if (! <condition>) o.wait();
```

is not sufficient!

Car Parking Controller

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

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

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

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

Synchronized, therefore holding lock of CarParkControl instance

Wait while no spaces available

Notify a car is available to leave

Wait while no cars inside

Notify any waiting thread that a space is available

Could use notifyAll(), but only one car can possibly enter

Single Notifications

- You can reduce the context-switch overhead associated with notifications by using a single `notify` rather than `notifyAll`
- Single notifications can be used to improve performance when you are *sure that at most one thread* needs to be awoken. This applies when:
  - all possible waiting threads are necessarily waiting for conditions relying on the same notifications, usually the exact same condition
  - each notification will enable at most a single thread to continue. Thus, it would be useless to wake up others

Summary

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

- The monitor is related to the *object instance* which is used to communicate among threads

- The lock must always be acquired before wait is invoked:

```
synchronized(lock) {
    ...
    lock.wait();
}
```

- Always re-check condition, after a wait:

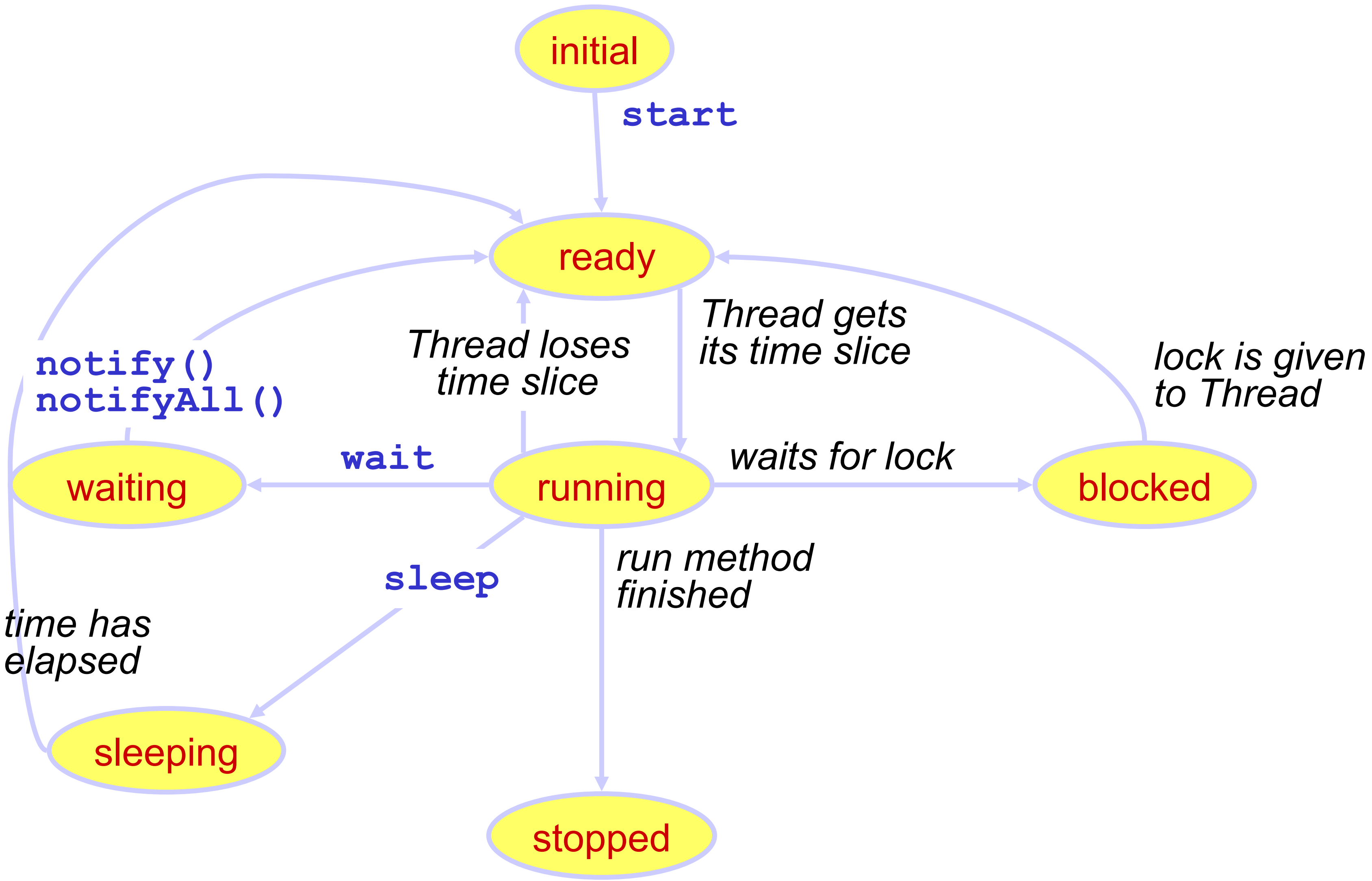
```
synchronized(lock) {
  while(!cond) {
    lock.wait();
  }
}
```

YES

```
synchronized(lock) {
  if(!cond) {
    lock.wait();
  }
}
```

NO

Thread State Transitions



Other Topics

- `t1.join()` – called by one thread to wait for another to complete before continuing
- Daemon threads
  - Must be declared with `setDaemon(true)`
  - Threads created by a daemon are also daemons
  - A program will not wait for daemon threads to terminate before exiting
- `Thread.interrupt()` breaks a thread out of a wait
- `java.util.concurrent`
  - Introduced in JDK 1.5
  - Utility classes commonly useful in concurrent programming