Conditional Synchronisation II and Introduction to Deadlock

Lecture 10

Alma Rahat

CS-210: Concurrency

24 Feb 2020

What did we do in the last session?

- Active and passive processes.
- Conditional access to shared resources.

In this lecture...

Swansea University
Prifysgol Abertawe

Learning outcomes.

1. To model condition synchronisation.
2. To write Java code for condition synchronisation.
3. To apply Semaphores for condition synchronisation.
4. To describe deadlocks between processes.
5. To explain the necessary and sufficient conditions for deadlock.

Outline.

1. Car park model and code.
2. `wait()`, `notify()`, and `notifyAll()`.
3. A library example with `wait()-notify()` and `Semaphore`.
4. Deadlock:
   - Dining Philosophers' Problem.
   - Necessary and sufficient conditions for deadlock.

Conditional Access to Resources

Scenario

There is a car park with one entrance and one exit, and n spaces. The controller only allows a car in *if there is at least one space left* and *lets one leave if there is at least one car in the car park*.

The resource – spaces – is going to be an attribute of the monitor class: **Controller**, and it will be accessed by the **Entrance** and **Exit** threads based on the two conditions.

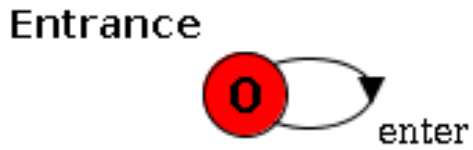
We will model the scenario first, and then see how it can be implemented in Java.

Entrance Model and Code

The entrance process simply issues an `enter` action and then remains the same process.

```
Entrance = (enter -> Entrance).
```

```
public class Entrance implements Runnable{
    Controller controller;
    private int sleepScaler = 1000;
    private String name;
    Entrance (Controller controller, String name){
        this.controller = controller;
        this.name = name;
    }
```



So, as an active process, Entrance will be a Thread by implementing the Runnable interface.

Quick recap: Why do we prefer Runnable?

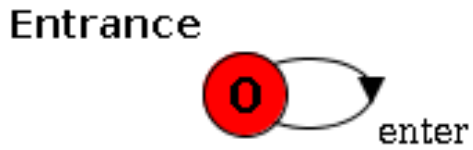
$\Rightarrow$ Prefer composition over inheritance, and it also allows you to make the active process to inherit from another class instead of `Thread`.

Entrance Model and Code

The entrance process simply issues an `enter` action and then remains the same process.

```
Entrance = (enter -> Entrance).
```

Here we call the enter method in the controller when the `Entrance` process is *run* in a thread.




```
@Override
public void run() {
    while (true){
        try {
            double random = Math.random();
            Thread.sleep((long) (random*sleepScaler));
            controller.enter();
            System.out.println(name + " arrive: " + controller.getSpaces());
        } catch (InterruptedException ex) {
            System.out.println("Interrupted Arrival Thread");
            return;
        }
    }
}
```

Exit Model and Code

The exit process simply issues a `leave` action and then remains the same process.

```
Exit = (leave -> Exit).
```

```
public class Exit implements Runnable{
    Controller controller;
    private int sleepScaler = 1000;
    private String name;
    Exit (Controller controller, String name){
        this.controller = controller;
        this.name = name;
    }
```

So, as an active process, `Exit` will be a Thread by implementing the Runnable interface.

Exit Model and Code

The exit process simply issues a `leave` action and then remains the same process.

```
Exit = (leave -> Exit).
```

Here we call the *leave* method in the controller when the `Exit` process is *run* in a thread.

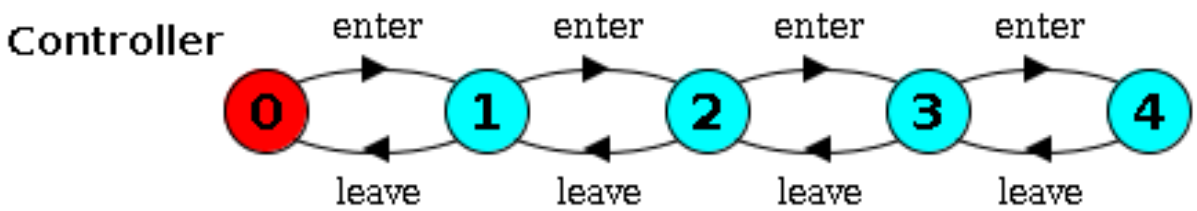

```
@Override
public void run() {
    while (true){
        try {
            double random = Math.random();
            Thread.sleep((long) (random*sleepScaler));
            controller.leave();
            System.out.println(name + " depart: " + controller.getSpaces());

        } catch (InterruptedException ex) {
            System.out.println("Interrupted Departure Thread");
            return;
        }
    }
}
```

Controller Model

The `Controller` *monitor* allows access to the *spaces* based on the two conditions:

- There is at least one space left, allow `enter` action.
- There is at least one car, allow `leave` action.

```
Controller(Capacity=4) = Spaces[Capacity],
Spaces[spacesLeft:0..Capacity] =
(when spacesLeft>0 enter -> Spaces[spacesLeft-1]
| when spacesLeft<Capacity leave ->
Spaces[spacesLeft+1]).
```



Implementing `Controller` in Java

Swansea University
Prifysgol Abertawe

Attributes `spacesLeft`, and `capacity`.

Methods `enter` and `leave`.

The methods will manipulate the attribute `spacesLeft`.
The constructor clearly shows that at the beginning the spaces left is equal to the capacity.

```
public class Controller {
    int spacesLeft;
    int capacity;
    public int getSpaces() {
        return spacesLeft;
    }
    Controller (int n){
        capacity = spacesLeft = n;
    }
```

Implementing Controller in Java

Attributes `spacesLeft`, and `capacity`.

Methods `enter` and `leave`.

The methods will manipulate the attribute `spacesLeft`.

Let's have a look at the `enter` method. We used synchronized to ensure that only one thread gets access to this method and the shared resource `spacesLeft`. Is this sufficient given the scenario?

Please go the `www.menti.com` and use the code 29 49 08 3.

```
public synchronized void enter(){
    spacesLeft++;
}
```

Implementing Controller in Java

Attributes `spacesLeft`, and `capacity`.

Methods `enter` and `leave`.

The methods will manipulate the attribute `spacesLeft`.

Let's have a look at the `enter` method. We used synchronized to ensure that only one thread gets access to this method and the shared resource `spacesLeft`. Is this sufficient given the scenario?

Please go the `www.menti.com` and use the code 29 49 08 3.

Where is the condition that ensures that we cannot push another car in when it is full?

```
public synchronized void enter(){
    spacesLeft++;
}
```

Implementing `Controller` in Java

Attributes `spacesLeft`, and `capacity`.

Methods `enter` and `leave`.

The methods will manipulate the attribute `spacesLeft`.

So, now we have a condition check that ensures that we only increment when there is at least one spaces. This is condition synchronisation in practice: access is dependent on meeting a condition.

Please go the `www.menti.com` and use the code 29 49 08 3.

```
public synchronized void enter(){
    if (spacesLeft<capacity)
        spacesLeft++;
}
```

Implementing `Controller` in Java

Attributes `spacesLeft`, and `capacity`.

Methods `enter` and `leave`.

The methods will manipulate the attribute `spacesLeft`.

So, now we have a condition check that ensures that we only increment when there is at least one spaces. This is condition synchronisation in practice: access is dependent on meeting a condition. Any further issues here?

Please go the `www.menti.com` and use the code 29 49 08 3.

```
public synchronized void enter(){
    if (spacesLeft<capacity)
        spacesLeft++;
}
```

Implementing `Controller` in Java

Swansea University
Prifysgol Abertawe

Attributes `spacesLeft`, and `capacity`.

Methods `enter` and `leave`.

The methods will manipulate the attribute `spacesLeft`.

So, now we have a condition check that ensures that we only increment when there is at least one spaces. This is condition synchronisation in practice: access is dependent on meeting a condition. Any further issues here?

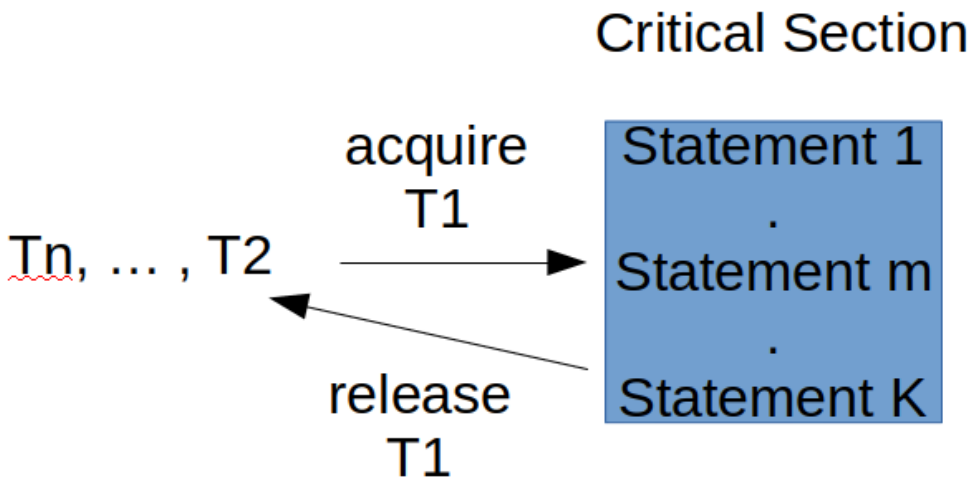
Please go the `www.menti.com` and use the code 29 49 08 3.

Here, a car (thread) comes to the entrance, and when the car park is full it cannot enter, so potentially it turns away. Okay, but in reality we may want the car to wait until a space becomes available.

```
public synchronized void enter(){
    if (spacesLeft<capacity)
        spacesLeft++;
}
```

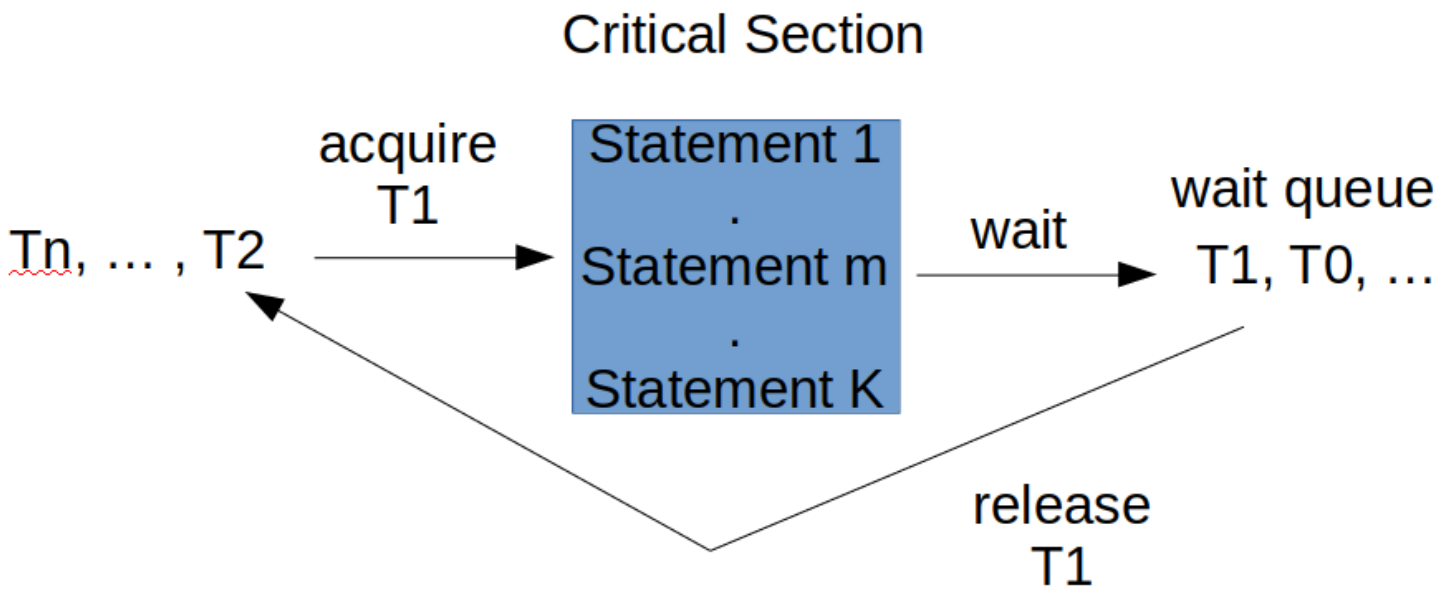
Waiting and Notifying in Java

Typically, a thread $T1$ enters the critical section, manipulates the data in the monitor, and releases the lock for others to capture.



Waiting and Notifying in Java

With waiting mechanism, a thread in a particular critical section may not progress further without satisfying a condition. Hence, it goes to the waiting arbitrary queue for its chance to come back and make progress, and releases the monitor lock. Other threads can now acquire the monitor lock to use this or another critical section, that may change the state of the monitor. A state change may generate a notification, which will wake up one of the threads in the queue.



Waiting and Notifying in Java

Swansea University
Prifysgol Abertawe

Java methods.

- wait() Causes the thread to exit the monitor, permitting other to enter the monitor. Use it to wait in a queue for condition synchronisation.
- notify() Notifies a single thread. Use it when there is only one condition upon which a queue of threads are potentially waiting.
- notifyAll() Notifies all threads that the state of the monitor has changed; a safer option to use.

Back to Implementing Controller

We used `wait()`, which can throw an `InterruptedException`. We also used `notifyAll()` to make sure that all other threads are given a nudge.

```
when condition==1 action -> State
```

```
public synchronized void enter()
    throws InterruptedException {
    while (spacesLeft == capacity){
        wait();
    }
    spacesLeft++;
    notifyAll();
}
```

Back to Implementing Controller

`leave` implementation is similar, except now the condition is different.

```
public synchronized void leave()
    throws InterruptedException {
    while (spacesLeft == 0){
        wait();
    }
    spacesLeft--;
    notifyAll();
}
```

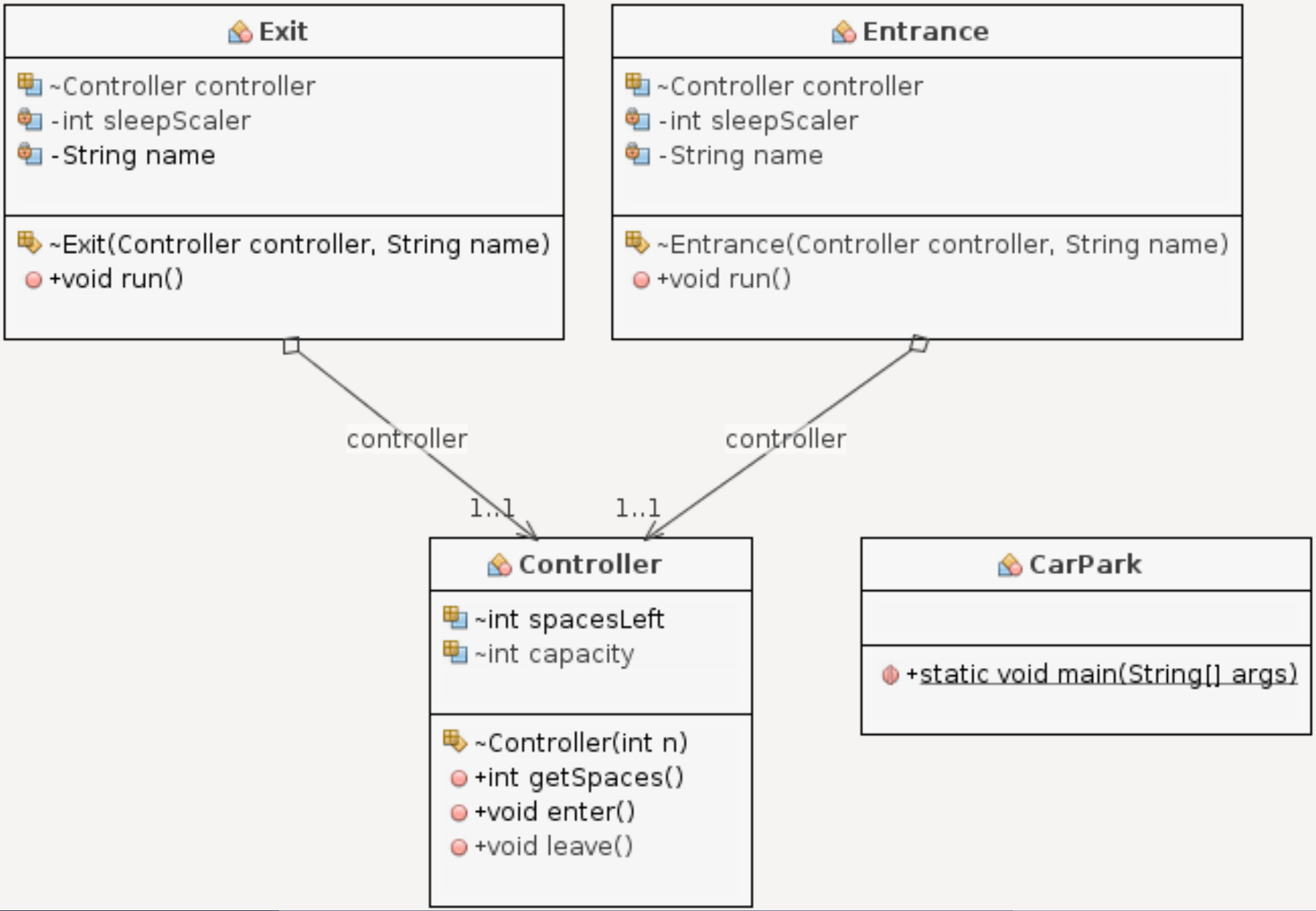
Back to Implementing Controller

The complete model and code for the car park brings everything together.

```
||CarPark = (Entrance || Controller || Exit).
```

```
public static void main(String[] args) throws InterruptedException {
    Controller controller = new Controller(4);
    Entrance entrance = new Entrance(controller, "In");
    Exit exit = new Exit(controller, "Out");
    Thread arrThread = new Thread(entrance);
    Thread depThread = new Thread(exit);
    arrThread.start();
    depThread.start();
    Thread.sleep(5*10000);
    arrThread.interrupt();
    arrThread.join();
    depThread.interrupt();
    depThread.join();
}
```

Back to Implementing Controller



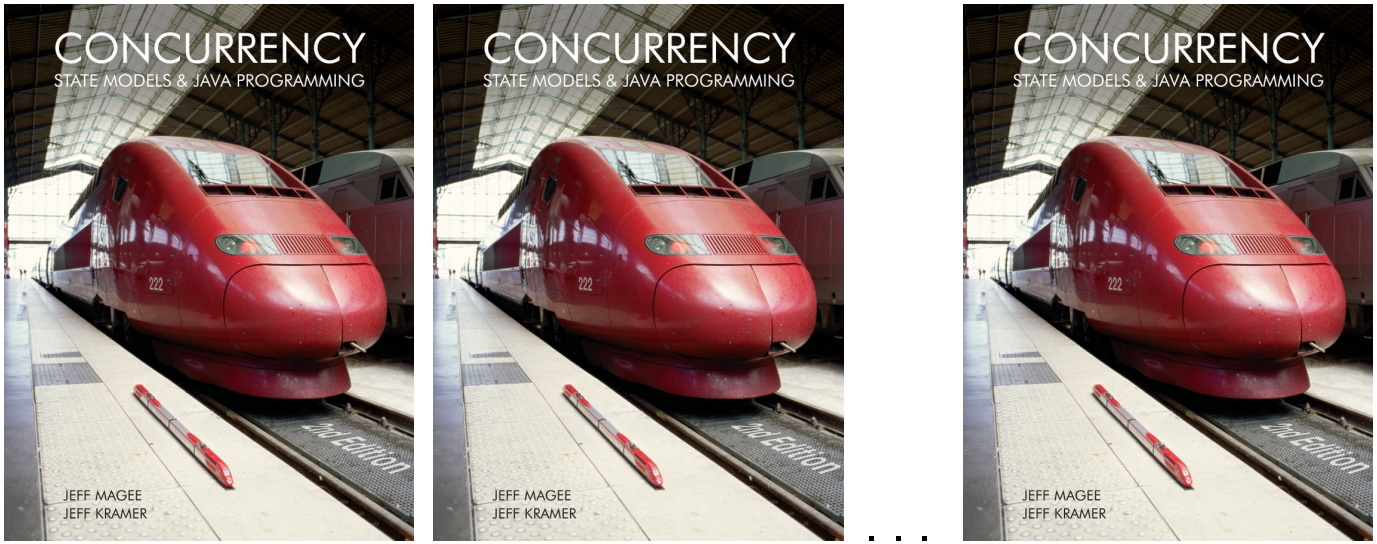
Any questions?

Homework

A bounded buffer has a fixed number of slots. There is a *producer* process, e.g. a keyboard, that puts integer numbers into the buffer array, and a *consumer* process, e.g. a display, that gets numbers form the buffer.

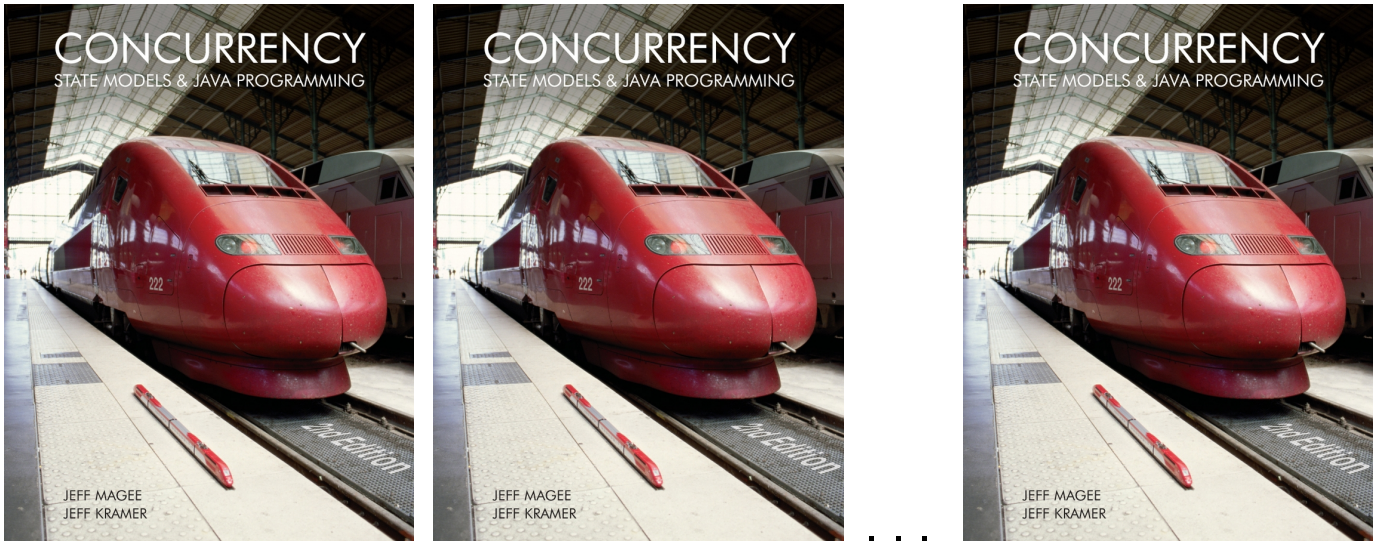
Model this system using FSP and write down the Java code for the monitor in this scenario.

A Library Example



Our library has n copies of the concurrency book for the course. A student can borrow a copy, read it, and then return it. A record is kept of which slot in the shelf has been emptied. Condition is that no one can take a book when there are none, and hence must wait for one to be returned.

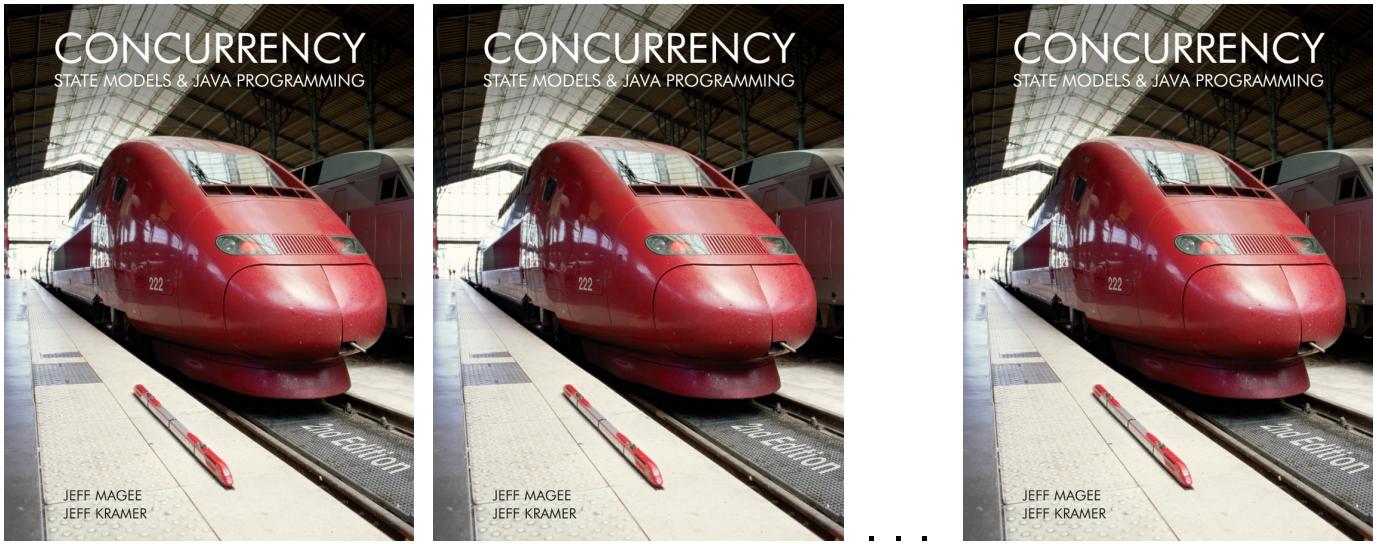
A Library Example



Our library has n copies of the concurrency book for the course. A student can borrow a copy, read it, and then return it. A record is kept of which slot in the shelf has been emptied. Condition is that no one can take a book when there are none, and hence must wait for one to be returned.

Similar to the Car Park and Buffer problems. We can use locks in these cases, but it is much better to use `wait()`→`notifyAll()` structure, so that we are not blocking access. Why?

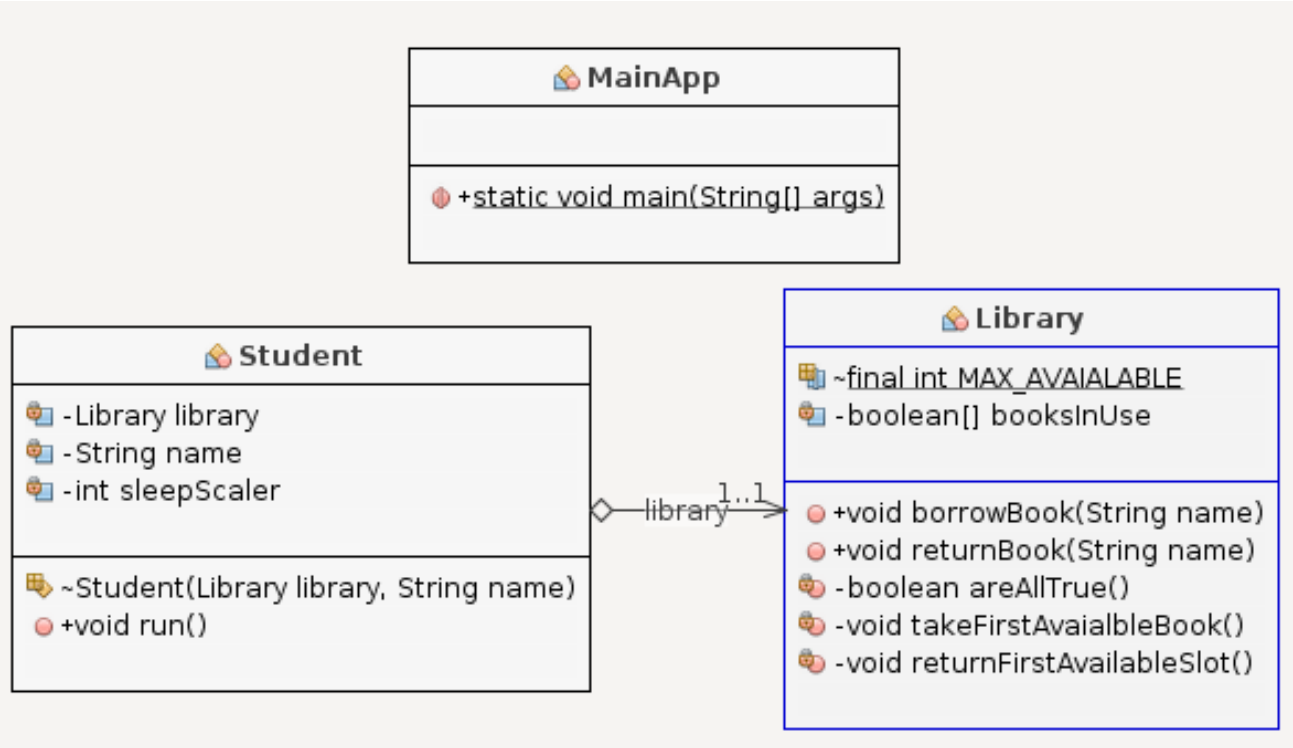
A Library Example



Our library has n copies of the concurrency book for the course. A student can borrow a copy, read it, and then return it. A record is kept of which slot in the shelf has been emptied. Condition is that no one can take a book when there are none, and hence must wait for one to be returned.

Similar to the Car Park and Buffer problems. We can use locks in these cases, but it is much better to use `wait()`→`notifyAll()` structure, so that we are not blocking access. Why? Better access control.

A Library Example



Library is the passive process here, and therefore the monitor class.
Student is implementing the *Runnable* interface as the active process.

A Library Example

Swansea University
Prifysgol Abertawe

```
public class Library{
    static final int MAX_AVAIALABLE = 2;
    private boolean[] booksInUse = new boolean[MAX_AVAIALABLE];
    public synchronized void borrowBook(String name)
            throws InterruptedException{
        while(areAllTrue()){
            System.out.println(name + " is waiting!");
            wait();
        }
        System.out.println(name + " is trying to get a book!");
        takeFirstAvaialbleBook();
        notifyAll();
    }
    private boolean areAllTrue(){
        for(boolean b : booksInUse) if(!b) return false;
        return true;
    }
    private void takeFirstAvaialbleBook() {
        for(int i=0; i<MAX_AVAIALABLE; i++)
            if (booksInUse[i] == false){
                booksInUse[i] = true;
                break;
            }
    }
```

A Library Example

```
    public synchronized void returnBook(String name)
            throws InterruptedException{
        System.out.println(name + " is trying to return a book!");
        returnFirstAvailableSlot();
        notifyAll();
    }
    private void returnFirstAvailableSlot() {
        for(int i=0; i<MAX_AVAIALABLE; i++)
            if (booksInUse[i] == true){
                booksInUse[i] = false;
                break;
            }
    }
}
```

A Library Example

```
public class Student implements Runnable{
    private Library library;
    private String name;
    private int sleepScaler = 10000;
    Student(Library library, String name){
        this.library = library;
        this.name = name;
    }
    @Override
    public void run() {
        try {
            library.borrowBook(name);
            double random = Math.random();
            System.out.println(name + " is starting to read.");
            Thread.sleep((long) (random*sleepScaler));
            System.out.println(name + " has finisehd reading.");
            library.returnBook(name);
        } catch (InterruptedException ex) {
            System.out.println("Interrupted Arrival Thread");
            return;
        }
    }
}
```

A Library Example

```
public class MainApp {
    public static void main(String[] args)
            throws InterruptedException {
        Library library = new Library();
        Student s1 = new Student(library, "Tom");
        Student s2 = new Student(library, "Jenny");
        Student s3 = new Student(library, "Plato");
        Thread t1 = new Thread(s1);
        Thread t2 = new Thread(s2);
        Thread t3 = new Thread(s3);
        t1.start();
        t2.start();
        t3.start();
        t1.join();
        t2.join();
        t3.join();
    }
}
```

A Library Example

```
run:
Tom is trying to get a book!
Plato is trying to get a book!
Jenny is waiting!
Plato is starting to read.
Tom is starting to read.
Plato has finisehd reading.
Plato is trying to return a book!
Jenny is trying to get a book!
Jenny is starting to read.
Tom has finisehd reading.
Tom is trying to return a book!
Jenny has finisehd reading.
Jenny is trying to return a book!
BUILD SUCCESSFUL (total time: 12 seconds)
```

Tom and Plato gets a book from the shelf, but Jenny keep waiting until Plato returns his book.

Any questions?

Summary

- In Java `wait` and `notify` allow us to implement condition synchronisation.