

# Interference III

## Lecture 8

Alma Rahat

CS-210: Concurrency

17 Feb 2021



# What did we do in the last session?

---

- Modelling an ornamental garden: an example of interference.

## Learning outcomes.

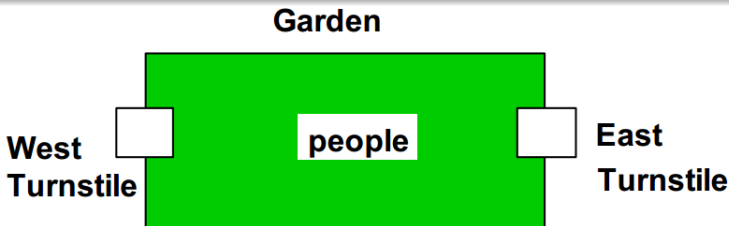
- ➊ To develop a test for an FSP model and ensure that the model does what was intended.
- ➋ To identify appropriate Java language features for guarding against interference.
- ➌ To understand the specification and the requirements of the coursework.

## Outline.

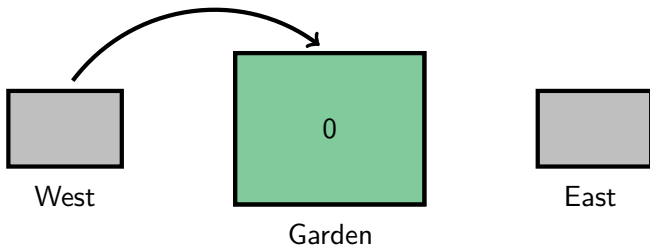
- ➊ Testing the model for correct behaviour.
- ➋ Synchronisation.
- ➌ Coursework brief.
- ➍ Mid-module feedback.

# An Ornamental Garden Problem

Consider a garden that has two turnstiles that allow people to enter the garden (no need to let them leave!). The turnstiles run parallelly and share a single counter, and each increment this counter 100 times. Ideal way to generate a race condition.



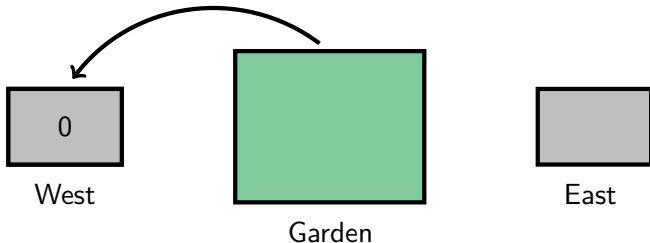
# Figuring out the test condition



A correct trace of actions:

`w.read[0]` →

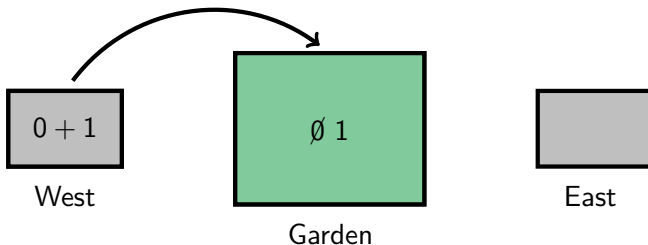
# Figuring out the test condition



A correct trace of actions:

`w.read[0] →`

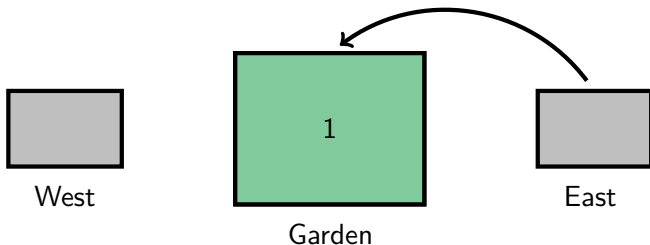
# Figuring out the test condition



A correct trace of actions:

$w.\text{read}[0] \rightarrow w.\text{write}[1] \rightarrow$

# Figuring out the test condition

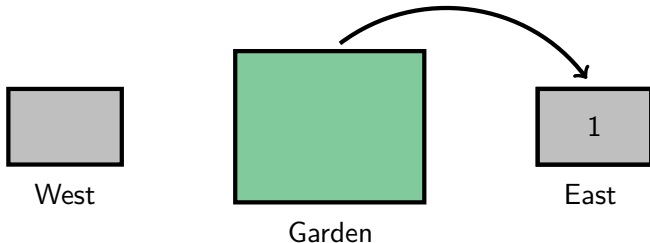


A correct trace of actions:

`w.read[0] → w.write[1] → e.read[1] →`



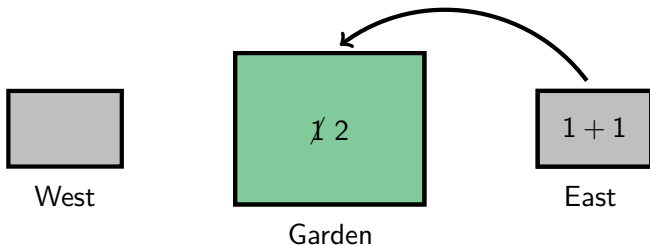
# Figuring out the test condition



A correct trace of actions:

`w.read[0] → w.write[1] → e.read[1] →`

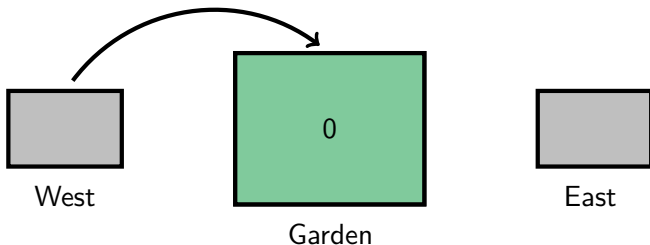
# Figuring out the test condition



A correct trace of actions:

$w.\text{read}[0] \rightarrow w.\text{write}[1] \rightarrow e.\text{read}[1] \rightarrow e.\text{write}[2] \rightarrow \text{end}.$

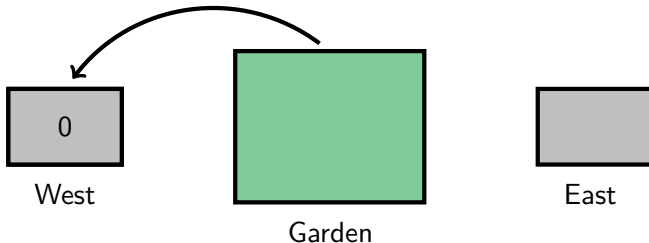
# Figuring out the test condition



A faulty trace of actions:

`w.read[0] →`

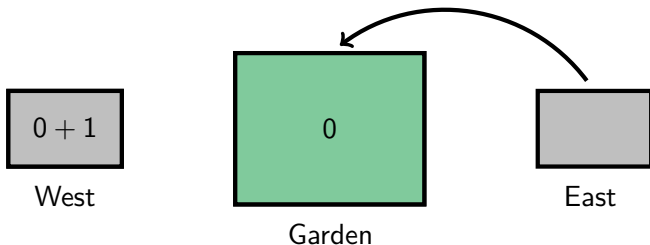
# Figuring out the test condition



A faulty trace of actions:

`w.read[0] →`

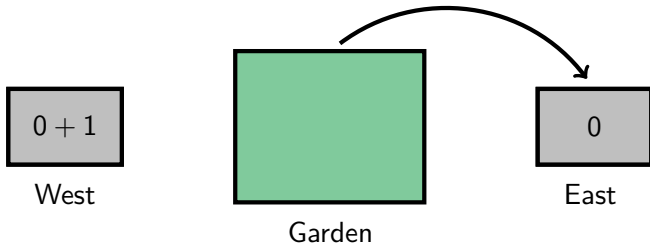
# Figuring out the test condition



A faulty trace of actions:

$w.\text{read}[0] \rightarrow e.\text{read}[0] \rightarrow$

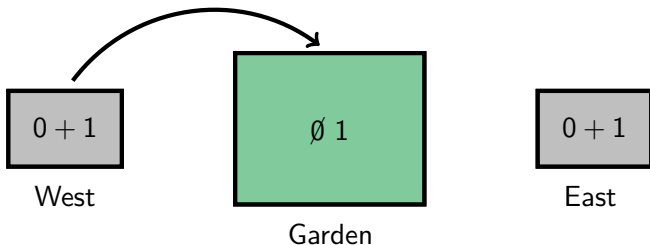
# Figuring out the test condition



A faulty trace of actions:

$w.\text{read}[0] \rightarrow e.\text{read}[0] \rightarrow$

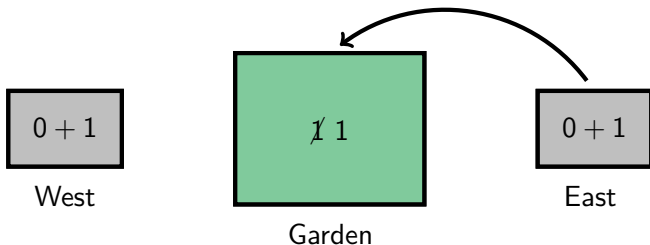
# Figuring out the test condition



A faulty trace of actions:

$w.\text{read}[0] \rightarrow e.\text{read}[0] \rightarrow w.\text{write}[1] \rightarrow$

# Figuring out the test condition

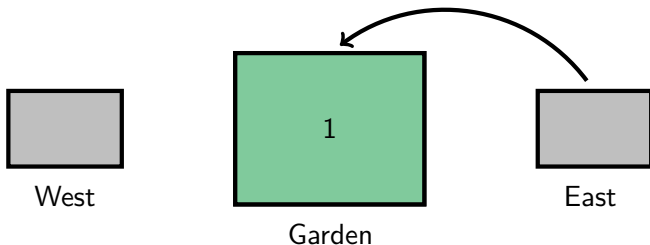


A faulty trace of actions:

$w.\text{read}[0] \rightarrow e.\text{read}[0] \rightarrow w.\text{write}[1] \rightarrow e.\text{write}[1] \rightarrow$



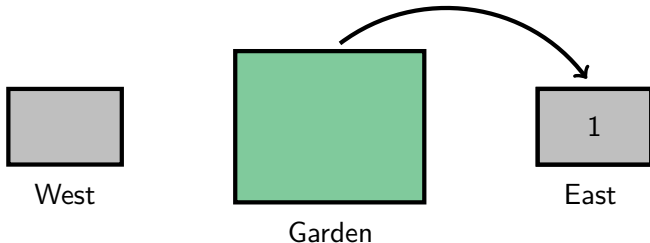
# Figuring out the test condition



A faulty trace of actions:

$w.\text{read}[0] \rightarrow e.\text{read}[0] \rightarrow w.\text{write}[1] \rightarrow e.\text{write}[1] \rightarrow e.\text{read}[1] \rightarrow$

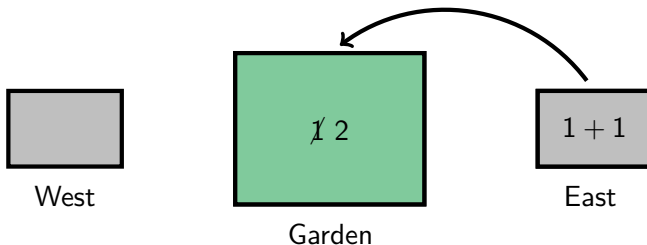
# Figuring out the test condition



A faulty trace of actions:

$w.\text{read}[0] \rightarrow e.\text{read}[0] \rightarrow w.\text{write}[1] \rightarrow e.\text{write}[1] \rightarrow e.\text{read}[1] \rightarrow$

# Figuring out the test condition



A faulty trace of actions:

We needed more writes (3) here to achieve two increments. So, we can count these for checking correctness.

$w.\text{read}[0] \rightarrow e.\text{read}[0] \rightarrow w.\text{write}[1] \rightarrow e.\text{write}[1] \rightarrow e.\text{read}[1] \rightarrow e.\text{write}[2] \rightarrow \text{end}.$

ERROR is a keyword in FSP: it represents a state with a label  $-1$ .

Here, if we consider that there must be  $N$  write actions for the program to successfully complete, we can count it in the following manner:

```
const A = 2*N // if both processes performed all writes then at
most we get A writes
range D = 0..A
TEST = TEST[0],
TEST[v:D] = ({east,west}.write[u:1..N] -> TEST[v+1] |
reset -> TEST[0] | when (v>N) wrong -> ERROR).
||TESTGARDEN = (GARDEN || TEST).
```

Check > Supertrace

Trace to property violation in TEST:

```
go
reset
west.arrive
west.read.0
west.write.1
west.arrive
west.read.1
east.arrive
east.read.1
west.write.2
west.arrive
west.return
east.write.2
wrong
```

Analysed using Supertrace in: 0ms

Auto-generated faulty trace with three write actions.

# Any questions?

---



An atomic action is defined as a statement (or set of statements) that is executed all at once (physically or logically). Such actions cannot be interleaved (for correct operation).

Here are some simple atomic actions (at the machine code level) in Java.

- Independent reads and writes are atomic for reference variables and for most primitive variables (excluding `long` and `double`).
- Independent reads and writes are atomic for all variables declared as `volatile` (including `long` and `double`).

Very simple expressions can encapsulate many complex interactions.

Increment action (e.g. `variable++`) is not atomic. Hence we observe the inconsistent results.

Even if independent reading and writing are atomic, we may face problems when there are shared objects.

synchronized methods embodying critical sections of code.

```
public synchronized void increment() {}
```

You can use `synchronised` with an instance of the `Counter` class, e.g. `synchronised(people) {people.increment();}`;, but then the user objects must ensure that they are using it responsibly.

## Effects of `synchronized`

- It is not possible for two invocations of `synchronised` methods on the same object to interleave. If one is using it, others suspend execution until the first one is done.
- Automatically establishes hierarchical (happens-before) relationship with any subsequent invocation of the method for the same object.





```
@Override  
public synchronized void increment(){  
    int temp = value; //read value  
    Simulate.HWInterrupt();  
    value = temp + 1; // set value  
}
```

Synchronisation uses locks with acquire and release actions to manage access. Once a Thread object has access to it, no other thread can access it until the critical section is completed by the key owner.

You may want to define the value variable as volatile. In a multithreading environment, volatile makes all threads use the main memory of the value, rather than a cached local copy. This is useful when you are expected to access a variable outside synchronized blocks.



```
@Override  
public synchronized void increment(){  
    int temp = value; //read value  
    Simulate.HWInterrupt();  
    value = temp + 1; // set value  
}
```

Synchronisation uses locks with acquire and release actions to manage access. Once a Thread object has access to it, no other thread can access it until the critical section is completed by the key owner.

You may want to define the value variable as volatile. In a multithreading environment, volatile makes all threads use the main memory of the value, rather than a cached local copy. This is useful when you are expected to access a variable outside synchronized blocks.

- Two instances of the same class will not be synchronized: each object will have its own lock.
- Static methods working with static variables are using the lock at the Class level.

## Performance issues

- Threads compete against each-other to capture the lock, and once held by one, others are suspended or blocked (this is computationally expensive).
- If there are multiple synchronized methods in an object, a thread working on one of these methods, does not allow others to access the other independent methods.

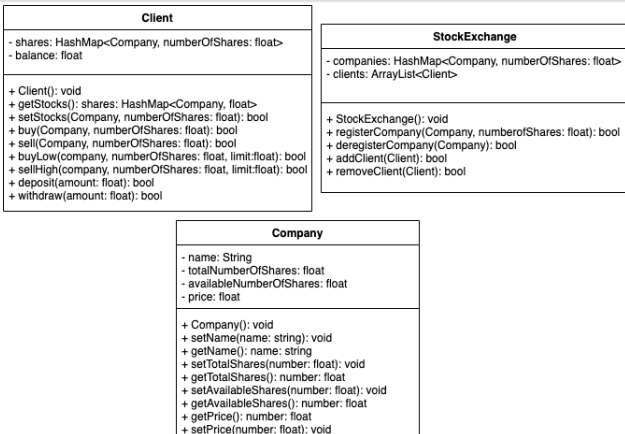
# Any questions?

---



## Scenario

A start-up company wants to develop a platform for trading shares.



Due date: 11:00 on the 26 March.

## Submission details.

**Code** Submit your Java code file.

**Documentation** You are expected to write a short document (no more than a page) where you detail:

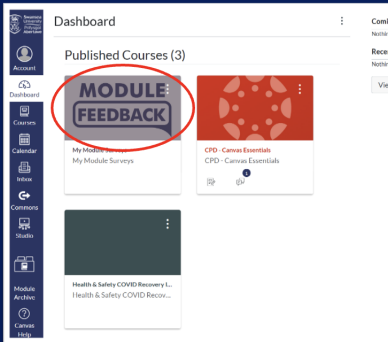
- A LTS graph for the component processes; no need to use FSP code or LTSA tool.
- What measures have been taken to protect against race conditions.
- What measures have been taken to avoid deadlock.

# Any questions?

---



# Could you please give us some feedback?



Dashboard

Published Courses (3)

**MODULE FEEDBACK**

My Module Surveys  
My Module Surveys

CPD - Canvas Essentials  
CPD - Canvas Essentials

Health & Safety COVID Recovery L...  
Health & Safety COVID Recov...

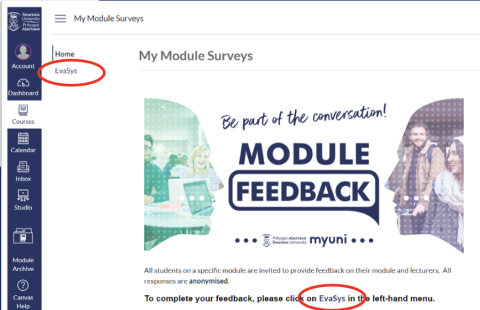
Coming up  
Nothing for the next week

Recent feedback  
Nothing for now

View Grades

You will see 'My Module Surveys' on your Canvas dashboard. All surveys for your modules will appear here.

Click on the course.



My Module Surveys

Home  
**EvaSys**

My Module Surveys

Be part of the conversation!

**MODULE FEEDBACK**

... myuni ...

All students on a specific module are invited to provide feedback on their module and lecturers. All responses are **anonymous**.

To complete your feedback, please click on **EvaSys** in the left-hand menu.

You can complete your survey(s) by clicking on EvaSys either in the left hand menu or in the main text.

EvaSys is the software used to create the surveys

Your input is extremely valuable for improving the module. Please let us know what we are doing well, and how we can improve in your comments.



- Arbitrary interleaving – on which we have no control – can cause interference.
- We model the behaviours and identify where the problem is through FSP and testing.