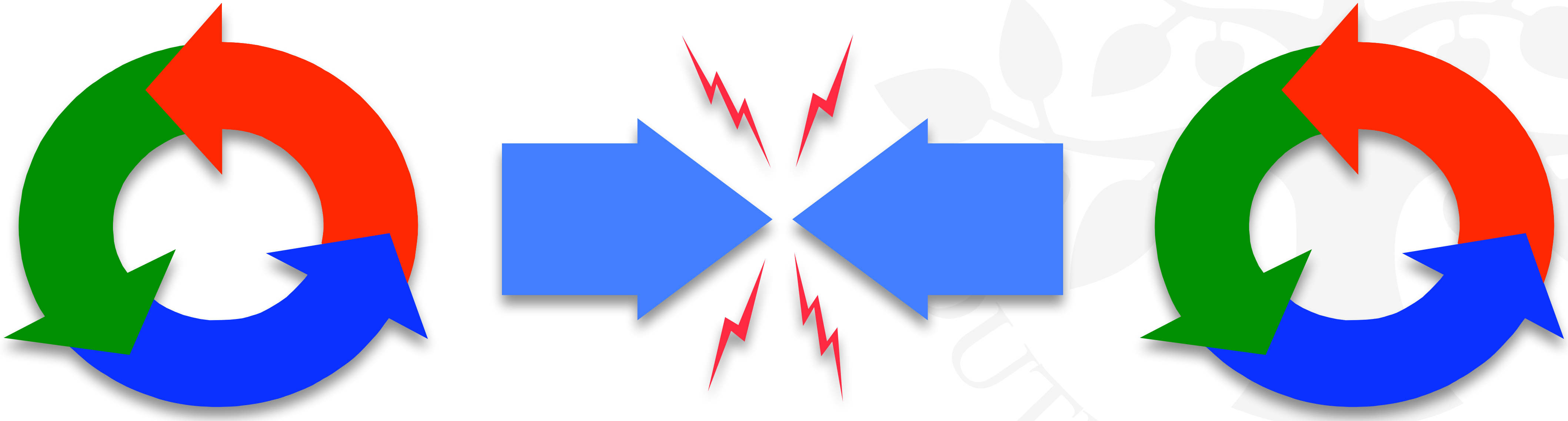
Chapter 4

University of Southern Denmark

Shared Objects & Mutual Exclusion

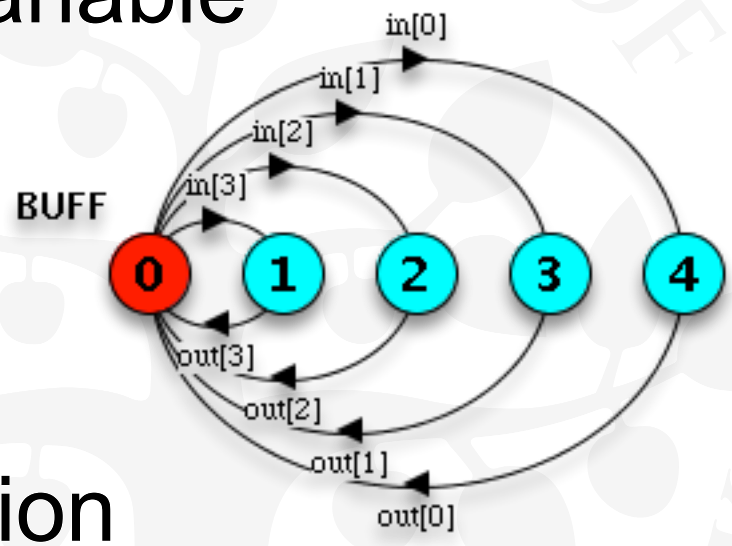


Repetition (Finite State Processes; Fsp)

Finite State Processes (FSP) can be defined using:

P =

- x -> Q // action
- Q // other process variable
- STOP // termination
- Q | R // choice
- when (...) x -> Q // guard
- ... + {write[0..3]} // alphabet extension
- X[i:0..N] =x[N-i] -> P // process & action index
- BUFF(N=3) // process parameter



```
range T = 0..3
BUFF = (in[i:T]->out[i]->BUFF).
```

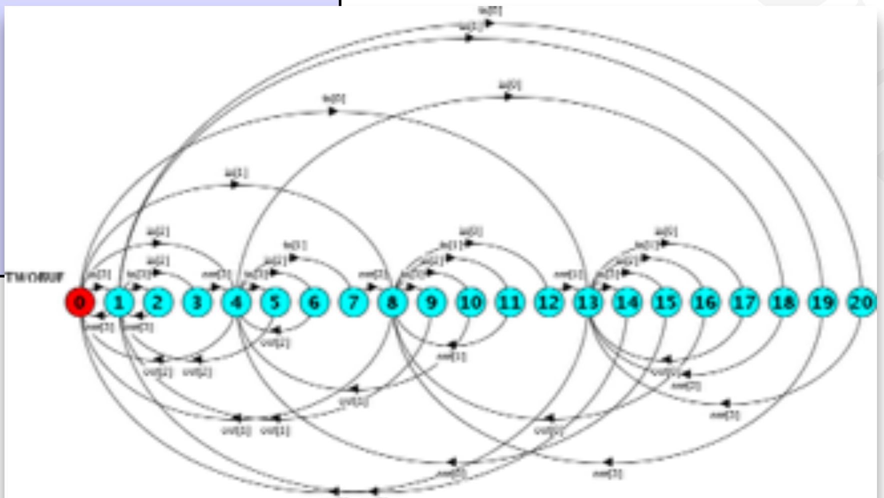
```
const N = 3          // constant definitions
range R = 0..N       // range definitions
set S = {a,b,c}      // set definitions
```

Repetition (Fsp)

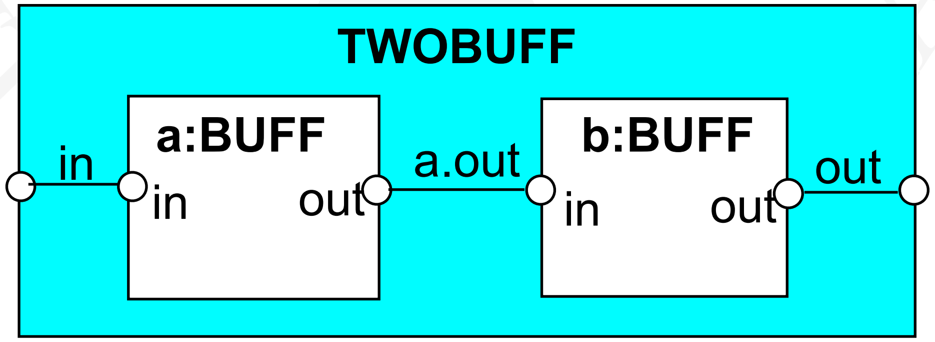
FSP:

- P || Q // parallel composition
- a:P // process labelling (1 process/prefix)
- {…}::P // process sharing (1 process w/all prefixes)
- P / {x/y} // action relabelling
- P \ {…} // hiding
- P @ {…} // keeping (hide complement)

```
||TWOBUF = (a:BUFF||b:BUFF)
              /{in/a.in,
                a.out/b.in,
                out/b.out}
              @{in,out}.
```



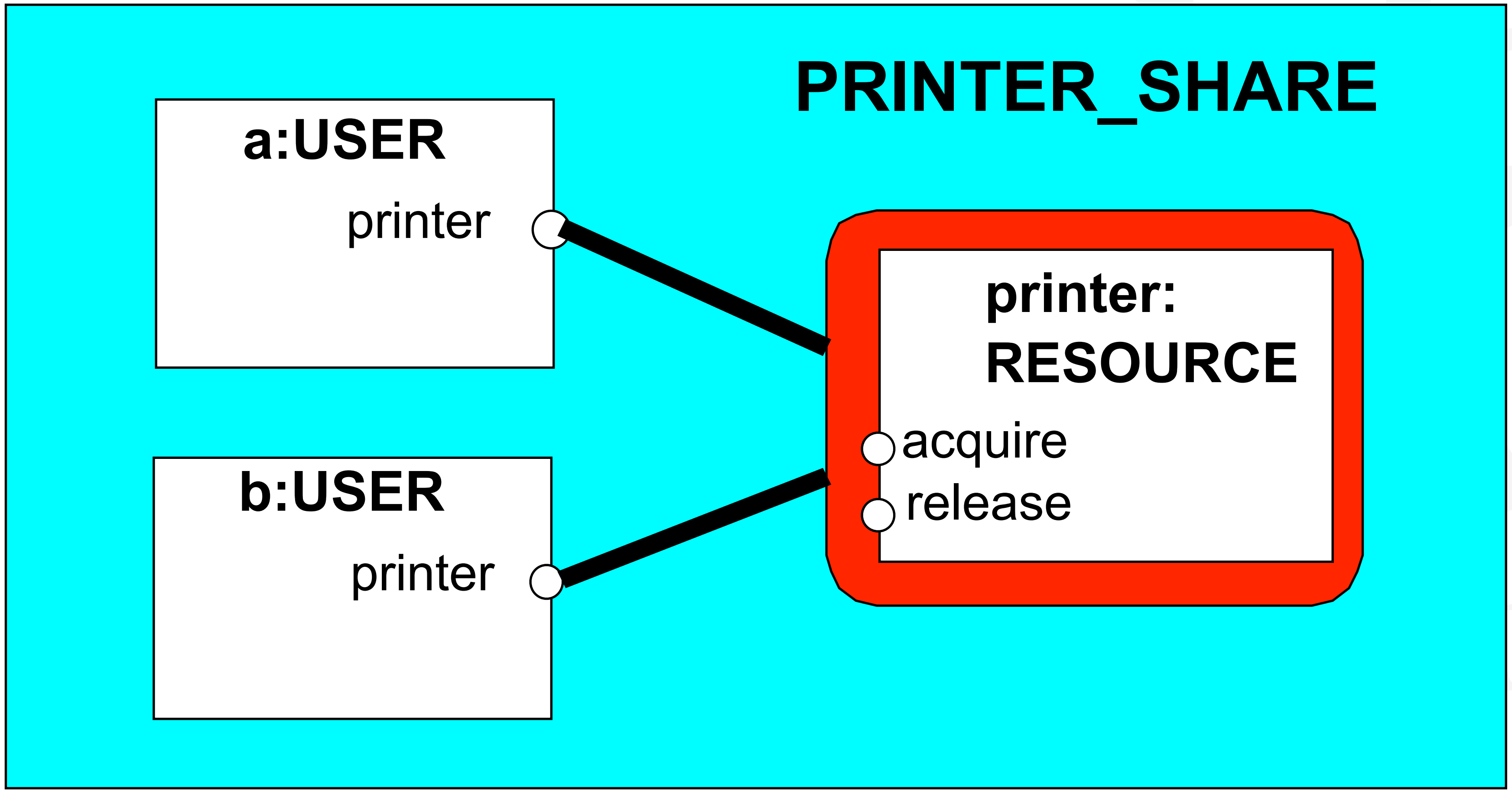
Structure Diagrams:



Structure Diagrams - Resource Sharing

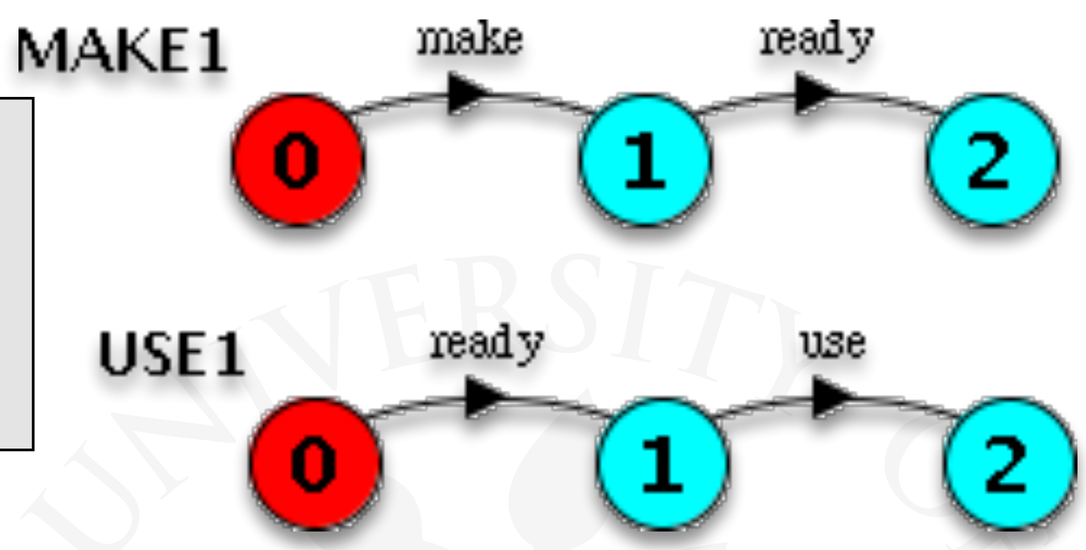
```
RESOURCE = (acquire->release->RESOURCE).
USER     = (printer.acquire->use->printer.release->USER).
```

```
||PRINTER_SHARE =
   (a:USER || b:USER || {a,b}::printer:RESOURCE).
```



How To Create The Parallel Composed Lts

```
MAKE1 = (make->ready->STOP).
USE1  = (ready->use->STOP).

||MAKE1_USE1 = (MAKE1 || USE1).
```

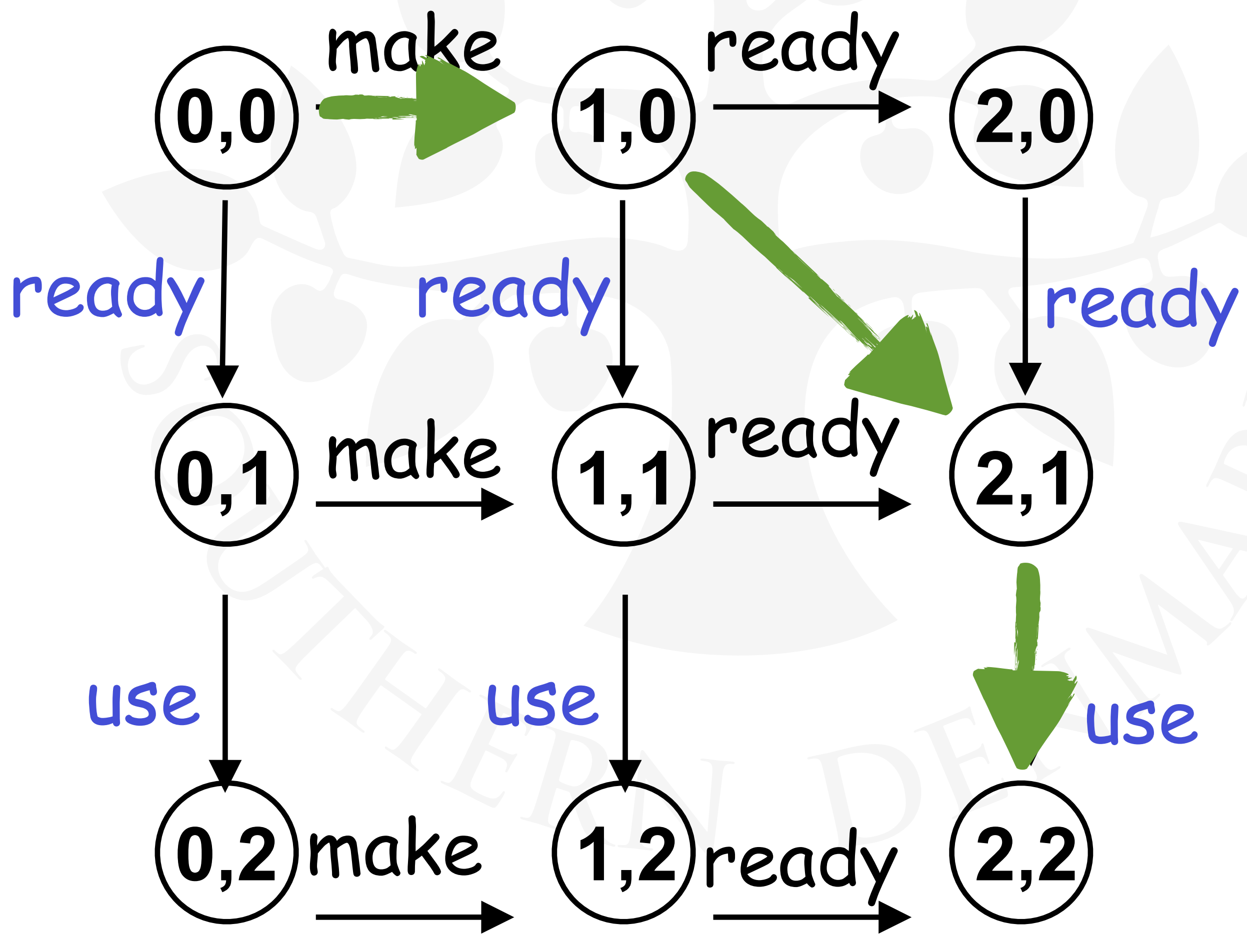


For any state reachable from the initial state (0,0), consider the possible actions and draw edges

→

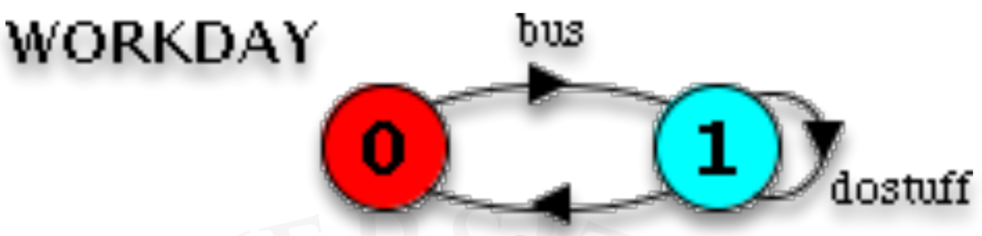
to the corresponding new states (i,j).

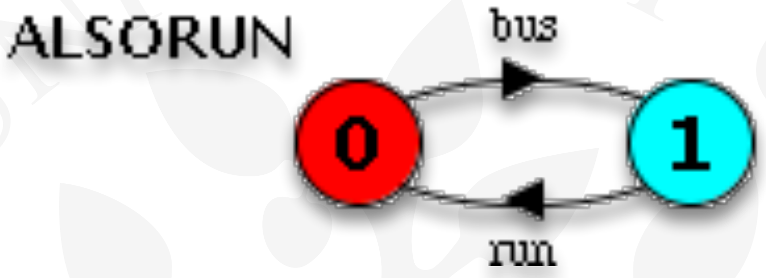
Remember to consider **shared** actions.



How To Create The Parallel Composed Lts

```
WORKDAY = HOME,
HOME = (bus -> WORK),
WORK = (dostuff-> WORK | bus -> HOME).

ALSORUN = (bus -> run -> ALSORUN).

||DAY = (WORKDAY || ALSORUN).
```

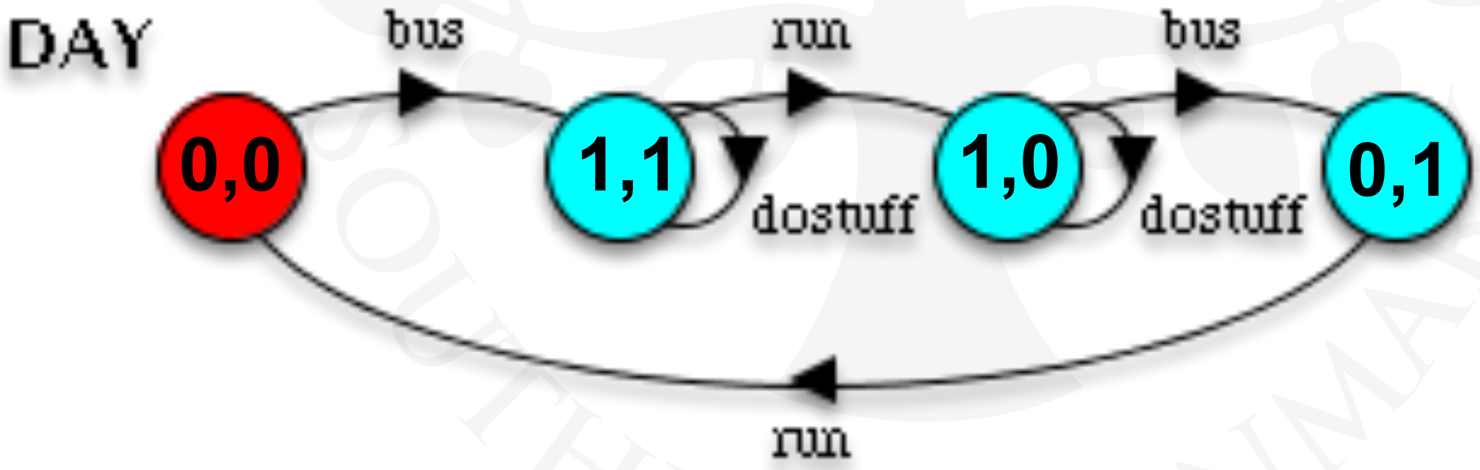




For any state reachable from the initial state (0,0), consider the possible actions and draw edges

to the corresponding new states (i,j).

Remember to consider **shared** actions.



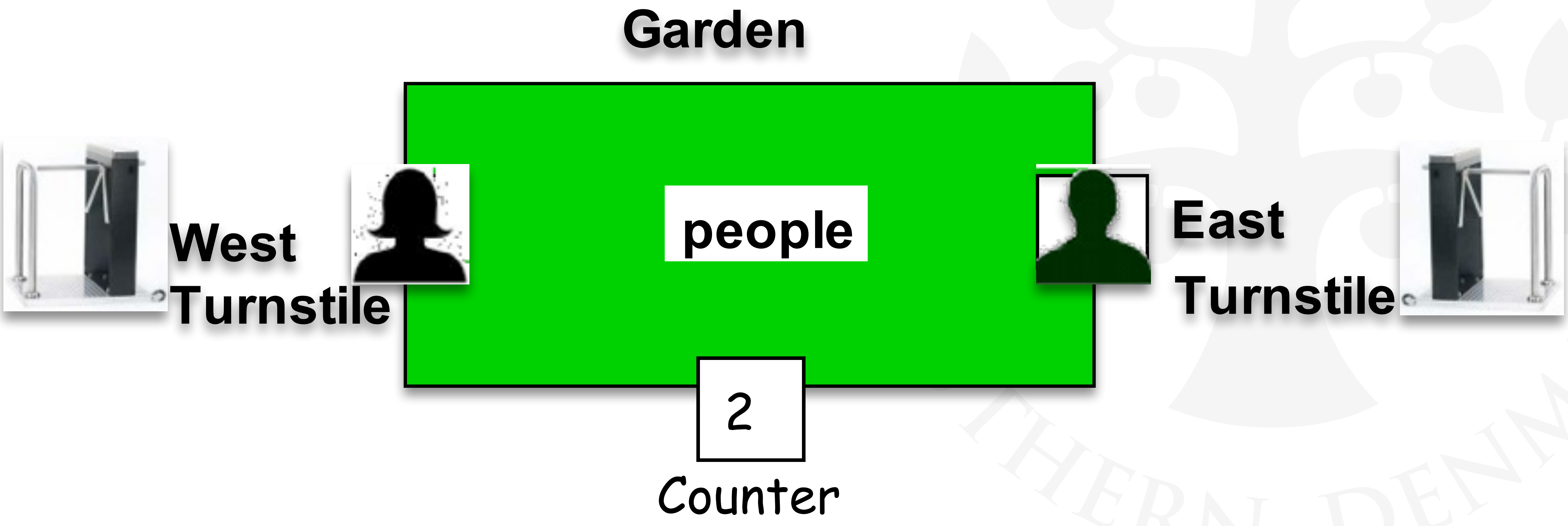
Chapter 4:
Shared Objects & Mutual Exclusion

- Concepts:
  - Process interference
  - Mutual exclusion
- Models:
  - Model-checking for interference
  - Modelling mutual exclusion
- Practice:
  - Thread interference in shared objects in Java
  - Mutual exclusion in Java
  - Synchronised objects, methods, and statements

4.1 Interference

The "Ornamental Garden Problem ":

People enter an ornamental garden through either of two turnstiles. Management wishes to know how many are in the garden at any time. (Nobody can exit).



Garden

West Turnstile

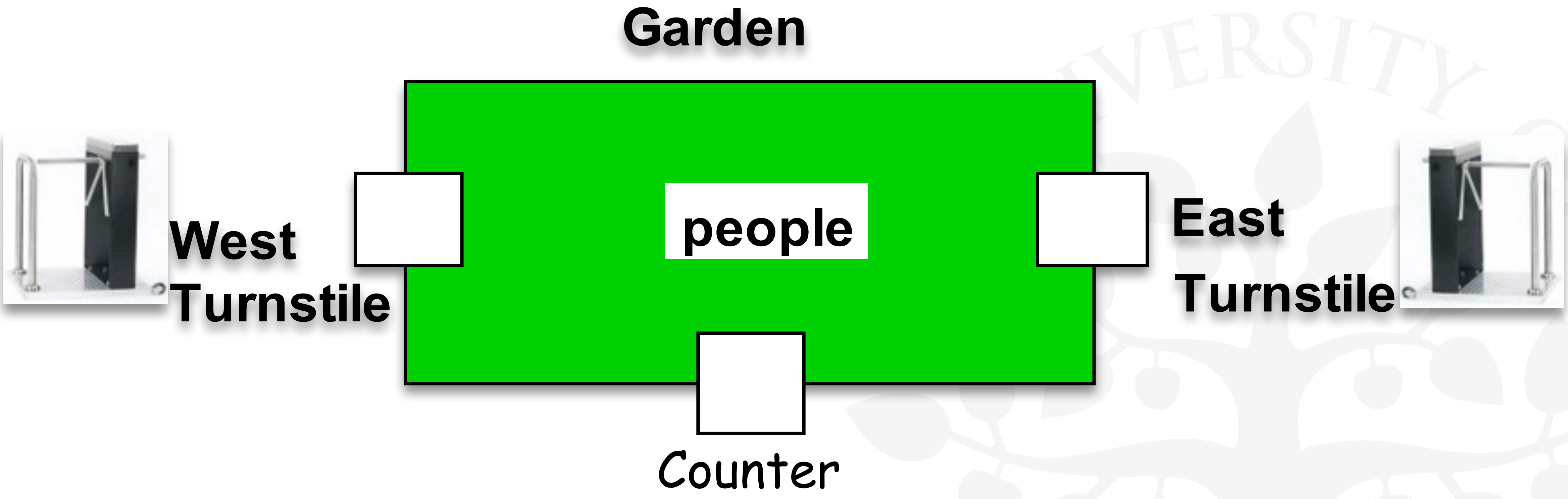
people

East Turnstile

2

Counter

4.1 Ornamental Garden Problem (Cont'd)



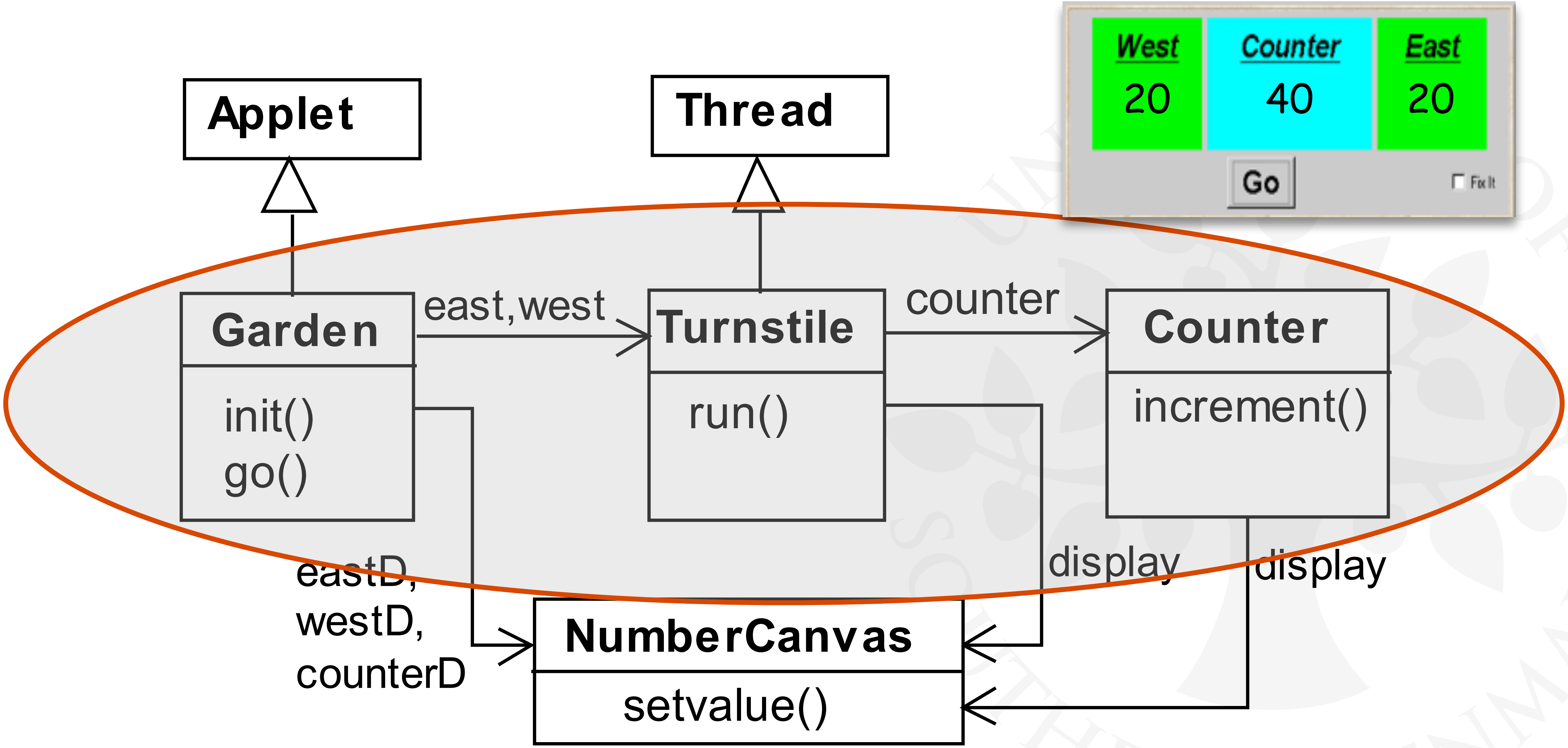
Java implementation:

The concurrent program consists of:

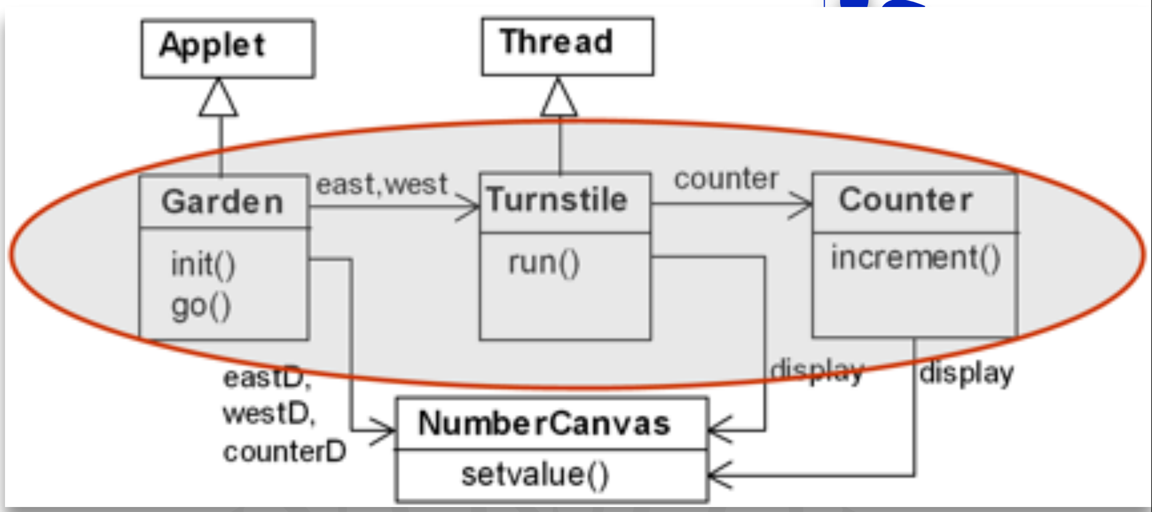
- two concurrent threads (west & east); and
- a shared counter object

2

Class Diagram



Ornamental Garden Program

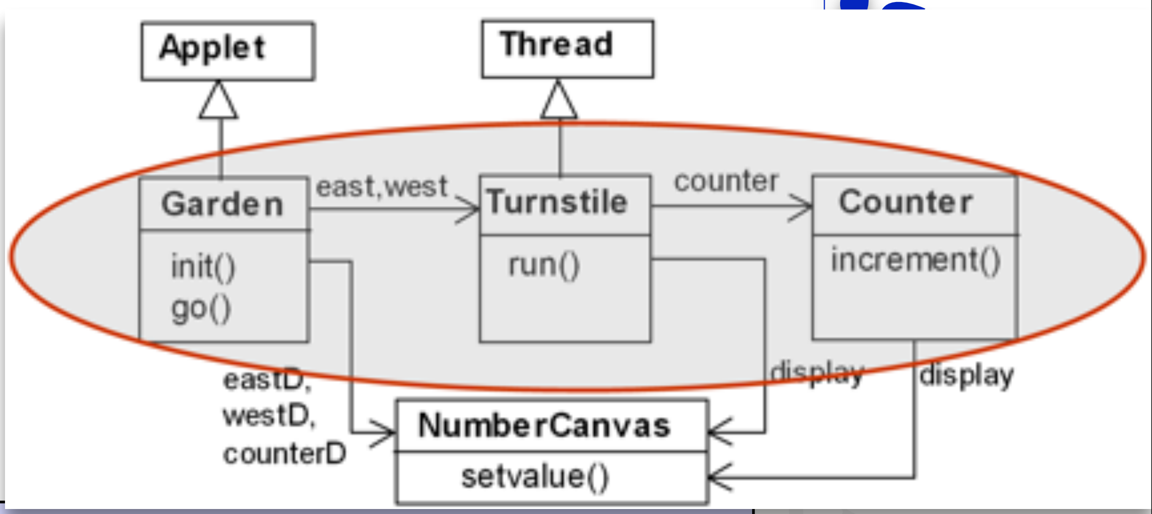


The **go()** method of the Garden applet…

```
class Garden extends Applet {
    NumberCanvas counterD, westD, eastD;
    Turnstile east, west;
    . . .
    private void go() {
        counter = new Counter(counterD);
        west = new Turnstile(westD,counter);
        east = new Turnstile(eastD,counter);
        west.start();
        east.start();
    }
}
```

…creates the shared **Counter** object & the **Turnstile** threads.

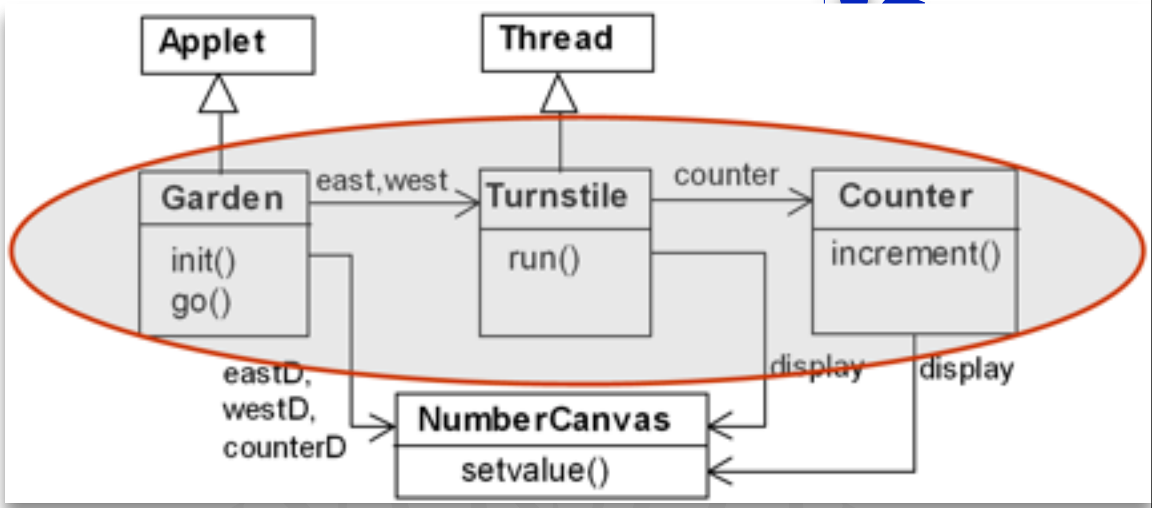
The Turnstile Class



```
class Turnstile extends Thread {
    NumberCanvas display;
    Counter counter;

    public void run() {
        try {
            display.setvalue(0);
            for (int i=1; i<=Garden.MAX; i++) {
                Thread.sleep(1000);
                display.setvalue(i);
                counter.increment();
            }
        } catch (InterruptedException _) {}
    }
}
```

The **Turnstile** thread simulates periodic arrival of visitors by invoking the counter object's **increment()** method every second

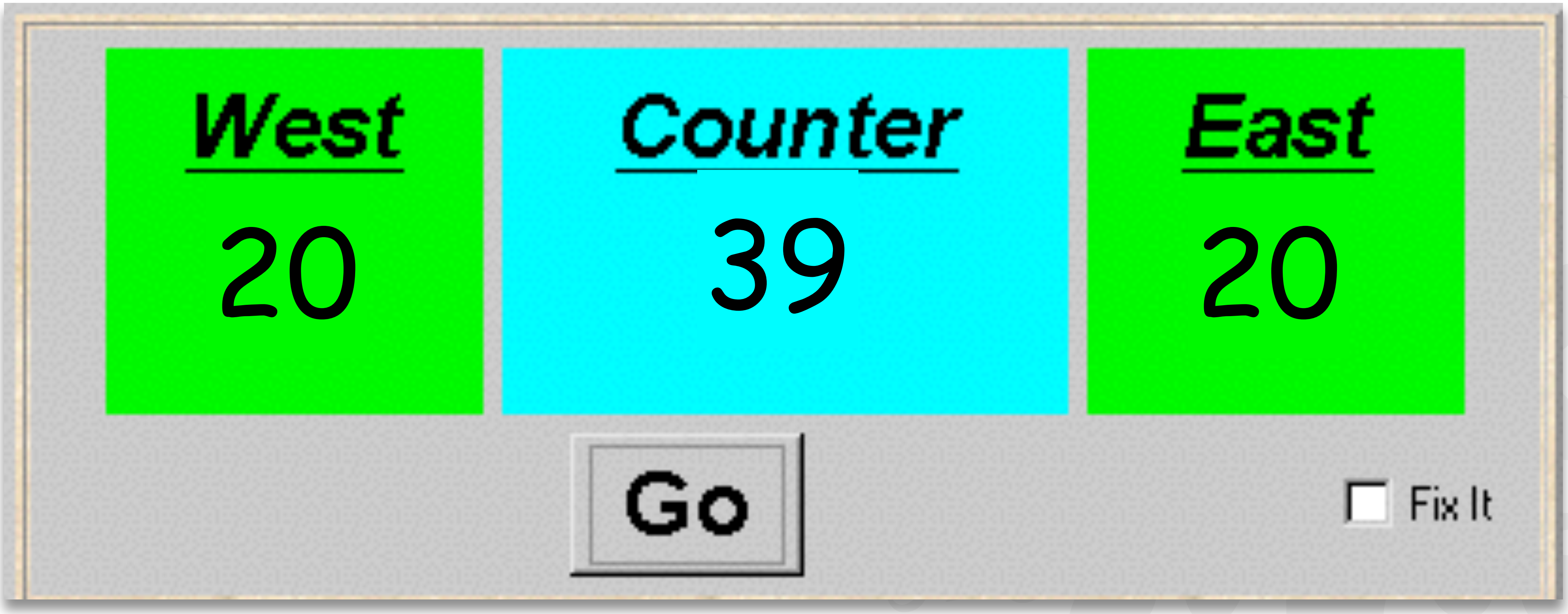
The *Shared* Counter Class



The **increment()** method of the Counter class increments its internal value and updates the display.

```
class Counter {
    int value;
    NumberCanvas display;

    void increment() {
        value = value + 1;
        display.setvalue(value);
    }
}
```

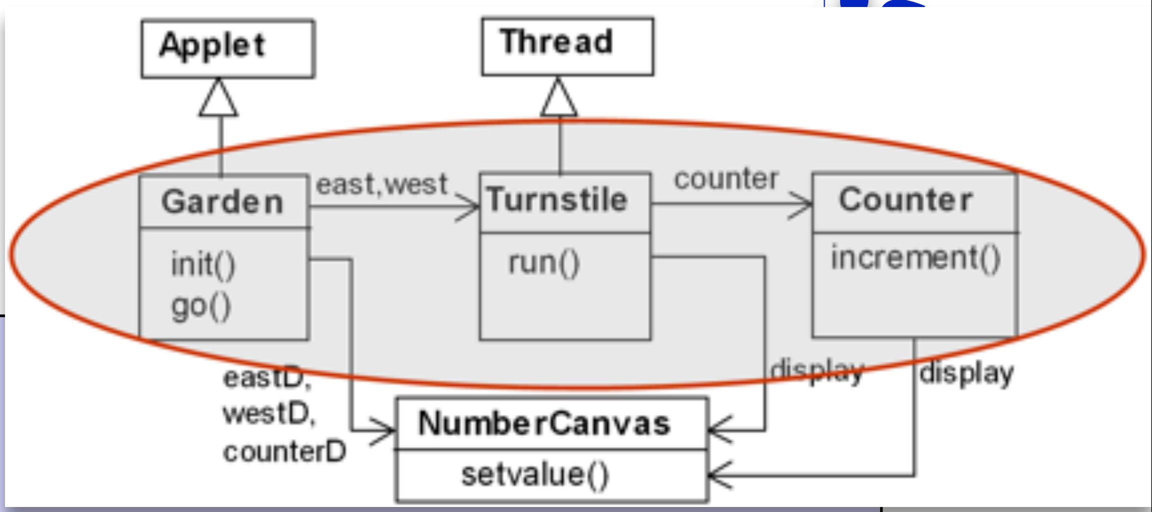
Running The Applet



After the East and West turnstile threads each have incremented the counter **20 times**, the garden people counter is **not always the sum of the counts** displayed.

Why?

The *Shared* Counter Class (Cont'd)



```
class Counter {
    int value;
    NumberCanvas display;

    void increment() {
        value = value + 1;
        display.setvalue(value);
    }
}
```

javac Counter.java
javap -c Counter > Counter.bc

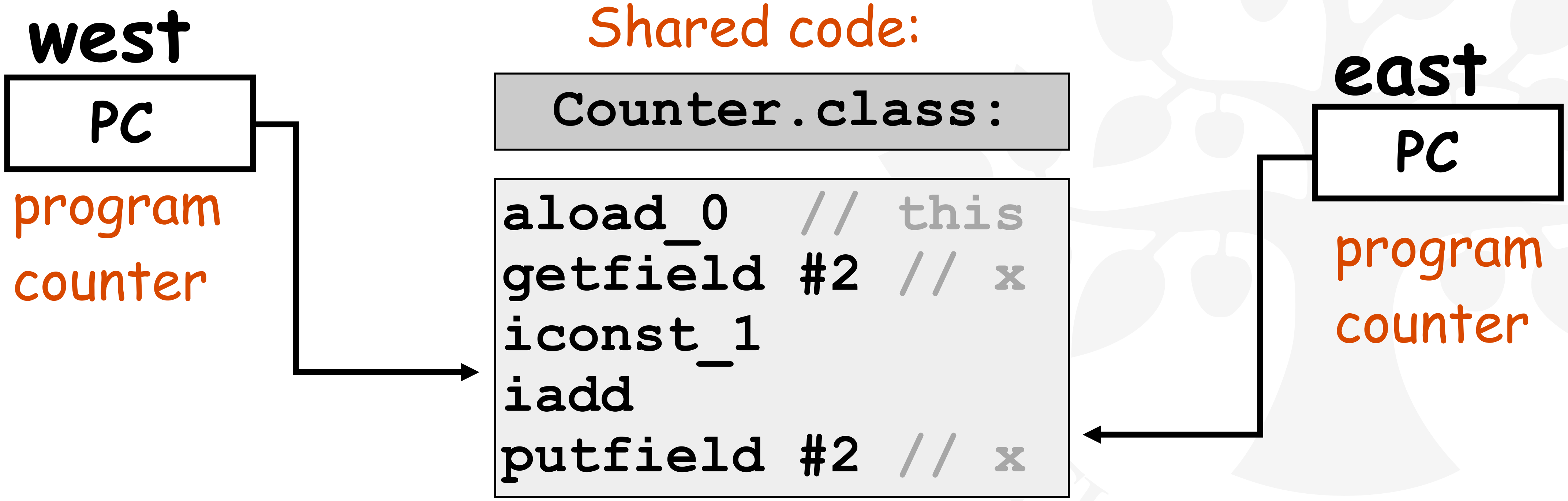
Thread switch?

```
aload_0         // push "this" onto stack
getfield #2     // get value of "this.value"
iconst_1        // push 1 onto stack
iadd            // add two top stack elements
putfield #2     // put result into "this.value"
```

Concurrent Method Activation

Java method activation is **not atomic**!

Thus, threads **`east`** and **`west`** may be executing the code for the increment method at the same time.



west

PC

program counter

Shared code:

`Counter.class:`

```
aload_0      // this
getfield #2  // x
iconst_1
iadd
putfield #2  // x
```

east

PC

program counter

Pedagogification; The Counter Class (Cont'd)

```
class Counter {
    void increment() {

        value = value + 1;

        display.setvalue(value);
    }
}
```

Pedagogification; The Counter Class (Cont'd)

```
class Counter {
    void increment() {
        int temp = value;   // read
        Simulate.HWinterrupt();
        value = temp + 1;   // write
        display.setvalue(value);
    }
}
```

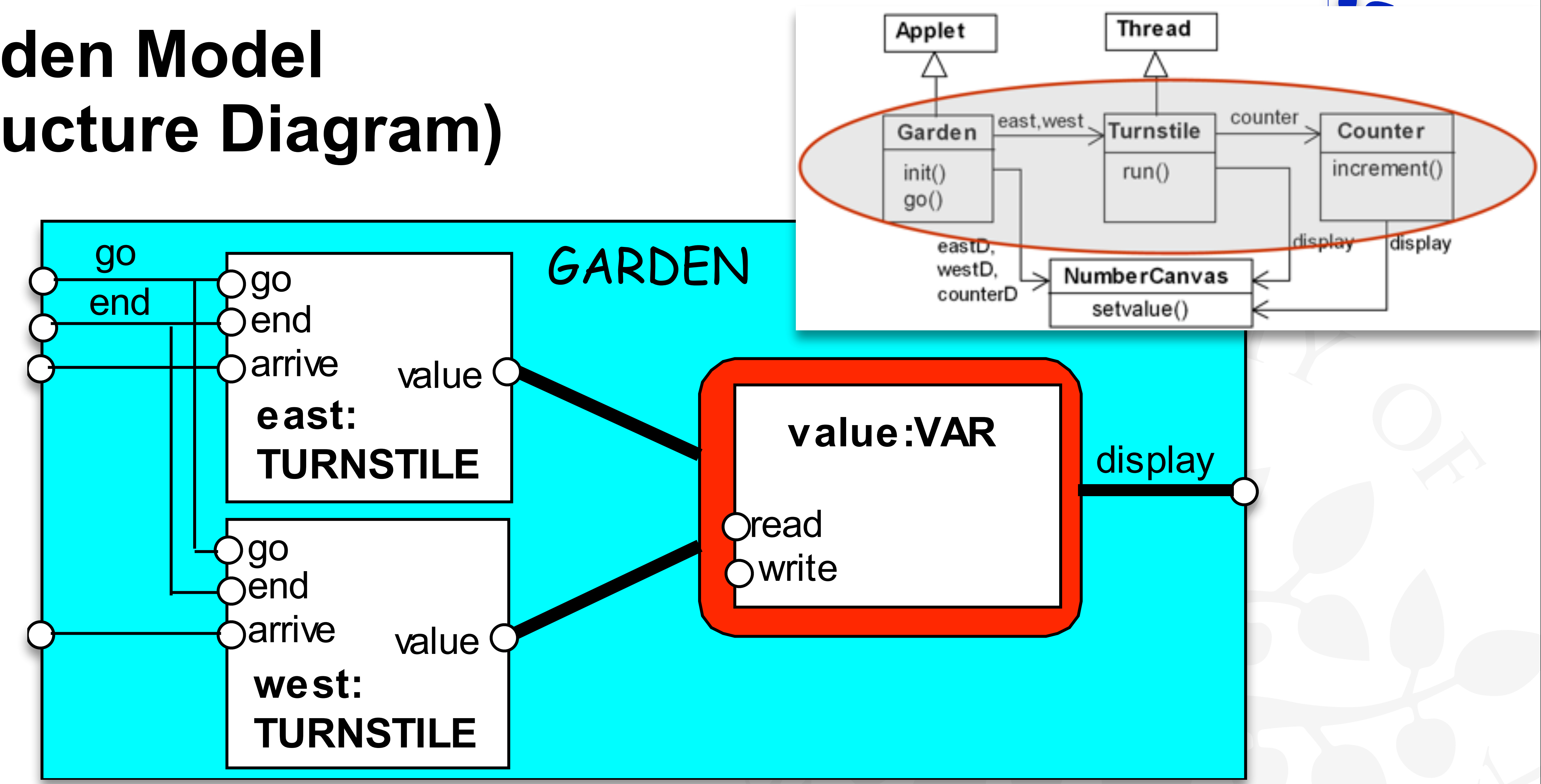
The **counter** simulates a hardware interrupt during an **increment()**, between reading and writing to the shared counter **value**.

```
class Simulate {    // randomly force thread switch!
    public static void HWinterrupt() {
        if (random()<0.5) Thread.yield();
    }
}
```

Running The Applet

Now the erroneous behaviour occurs almost all the time!

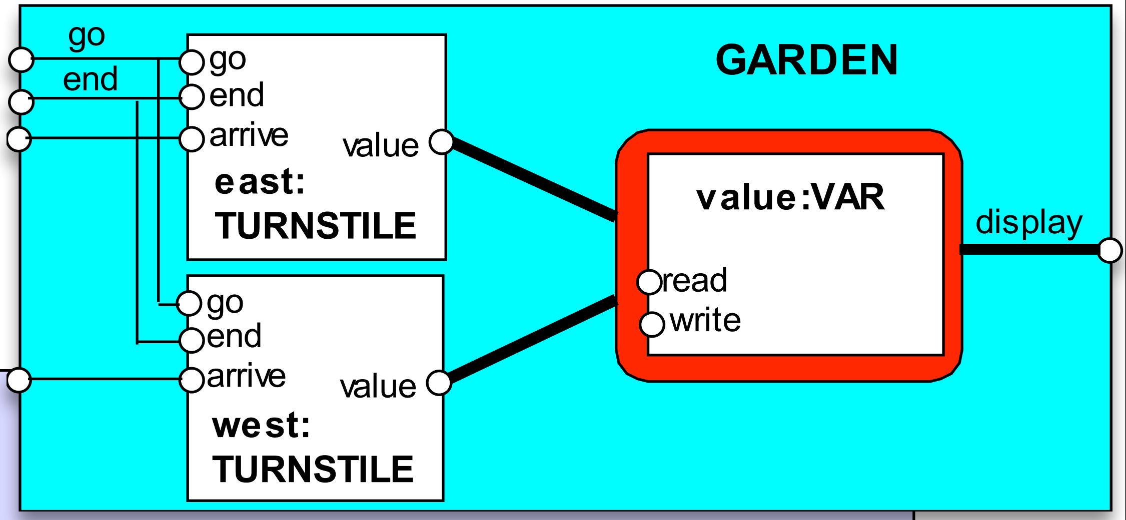
Garden Model
(Structure Diagram)



VAR:
models read and write access to the shared counter `value`.

TURNSTILE:
Increment is modelled inside TURNSTILE, since Java method activation is not atomic (i.e., thread objects `east` and `west` may interleave their `read` and `write` actions).

Ornamental Garden Model (Fsp)



```
const N = 4
range T = 0..N

VAR      = VAR[0],
VAR[u:T] = (read[u] -> VAR[u] | write[v:T] -> VAR[v]).

TURNSTILE = (go -> RUN),
RUN       = (arrive -> INCREMENT | end -> TURNSTILE),
INCREMENT = (value.read[x:T] -> value.write[x+1] -> RUN)
                                       +{value.write[0]}.

DISPLAY =(value.read[T]->DISPLAY)+{value.write[T]}.

||GARDEN = (east:TURNSTILE || west:TURNSTILE || display:DISPLAY
                             || {east,west,display}::value:VAR)
      /{ go / {east,west}.go , end / {east,west}.end}.
```

α(VAR) ?

α(value:VAR) ?

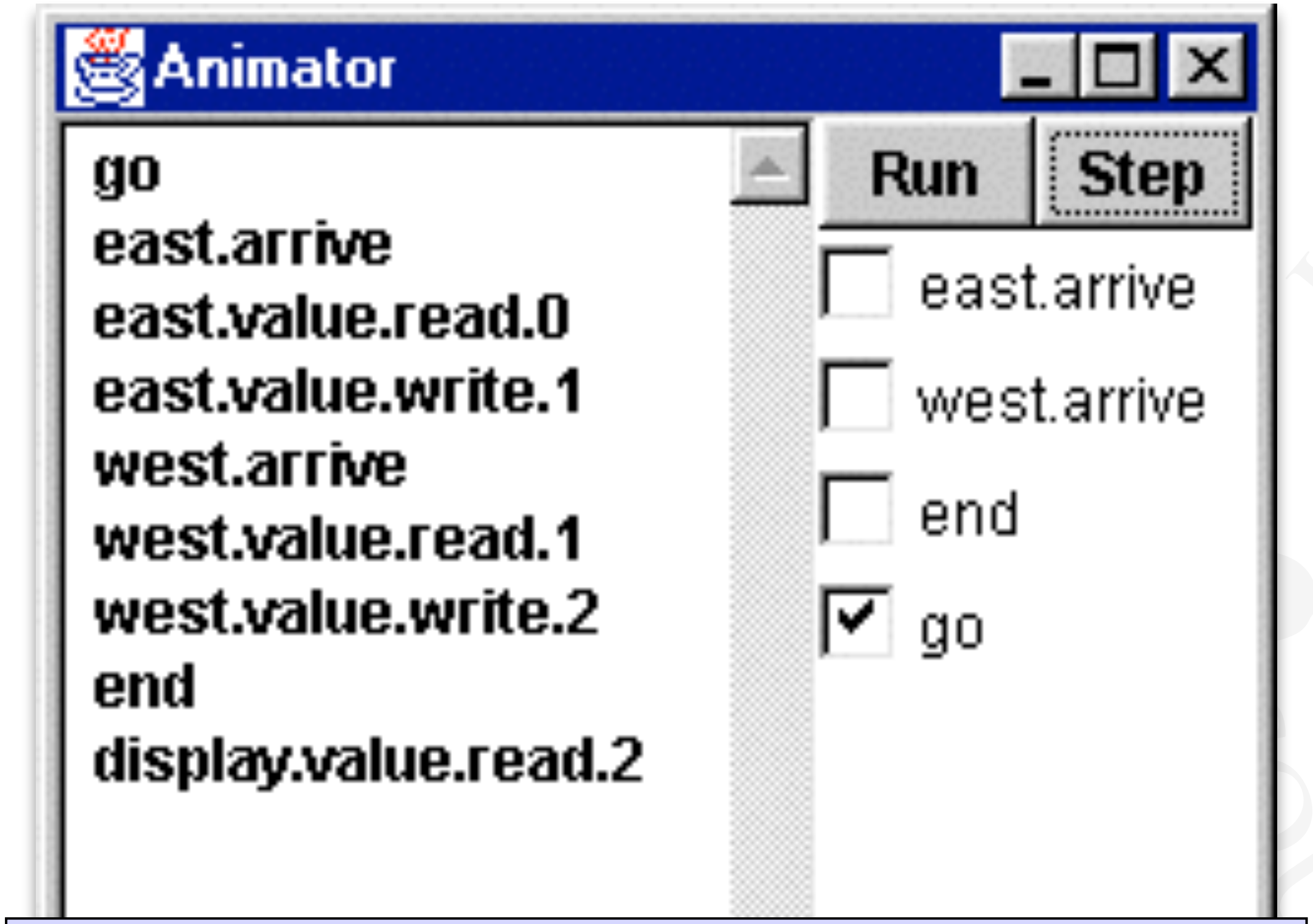
α({east,west,display}::value:VAR) ?

α(TURNSTILE) ?

α(east:TURNSTILE) ?

α(display:DISPLAY) ?

Checking For Errors - Animation



```
go
east.arrive
east.value.read.0
east.value.write.1
west.arrive
west.value.read.1
west.value.write.2
end
display.value.read.2
```

☐ east.arrive
☐ west.arrive
☐ end
☑ go

Scenario checking - use animation to produce a trace.

Is the model correct?

> "Never send a human to do a machine's job"
>
> \- *Agent Smith (1999)*

Checking For Errors - Compose With Error Detector

Exhaustive checking - compose the model with a TEST process which sums the arrivals and checks against the display value:

```
TEST        = TEST[0],

TEST[v:T]   = (when  (v<N) west.arrive->TEST[v+1]
              |when  (v<N) east.arrive->TEST[v+1]
              |end -> CHECK[v]),

CHECK[v:T] = (display.value.read[u:T] ->
                  (when  (u==v) right -> TEST[v]
                  |when  (u!=v) wrong -> ERROR)).
```

Checking For Errors - Exhaustive Analysis

```
||TESTGARDEN = (GARDEN || TEST).
```

Use **LTSA** to perform an exhaustive search for **ERROR**:

```
        Trace to property violation in TEST:
            go
            east.arrive
            east.value.read.0
            west.arrive
            west.value.read.0
            east.value.write.1
            west.value.write.1
            end
            display.value.read.1
      wrong
```

LTSA produces the shortest path to reach the **ERROR** state.

Interference And Mutual Exclusion

Destructive update, caused by the arbitrary interleaving of read and write actions, is termed **interference**.

Interference bugs are **extremely difficult** to locate.

The general solution is:

- Give methods mutually exclusive access to shared objects.

Mutual exclusion can be modelled as atomic actions.

4.2 Mutual Exclusion In Java

Concurrent activations of a method in Java can be made mutually exclusive by prefixing the method with the keyword `synchronized`.

We correct the `Counter` class by deriving a class from it and making its increment method synchronized:

```
class SynchronizedCounter extends Counter {
    SynchronizedCounter(NumberCanvas n) {
        super(n);
    }
    synchronized void increment() {
        super.increment();
    }
}
```

The Garden Class (Revisited)

If the `fixit` checkbox is ticked, the **go()** method creates a **SynchronizedCounter**:

```
class Garden extends Applet {
    private void go() {
        if (!fixit.getState())
            counter = new Counter(counterD);
        else
            counter = new SynchCounter(counterD);
        west = new Turnstile(westD,counter);
        east = new Turnstile(eastD,counter);
        west.start();
        east.start();
    }
}
```

Mutual Exclusion - The Ornamental Garden

Java associates a lock with every object.

The Java compiler inserts code to:

- acquire the lock before executing a synchronized method
- release the lock after the synchronized method returns.

Concurrent threads are blocked until the lock is released.

Java `synchronized` Statement

Synchronized methods:

```
synchronized void increment() {
    super.increment();
}
synchronized void decrement() {
    super.decrement();
}
```

Variant - the synchronized statement :

object reference

```
class Turnstile{
    ...
    public void run() {
    ...
        synchronized(counter) {
            counter.increment();
        }
    ...
```

Use synch methods whenever possible.

Java -> Java Bytecode

```
class X {
  int x;
  void m() {
    synchronized(this) {
      x++;
    }
  }
}
```

compile

```
Method void m()
>> max_stack=3, max_locals=3 <<


    0 aload_0
    1 dup
    2 astore_1
    3 monitorenter
    4 aload_0
    5 dup
    6 getfield #2 <Field X.x:int>
    9 iconst_1
   10 iadd
   11 putfield #2 <Field X.x:int>
   14 aload_1
   15 monitorexit
   16 goto 24
   19 astore_2
   20 aload_1
   21 monitorexit
   22 aload_2
   23 athrow
   24 return


Exception table:
   from    to  target type
     4     16    19   any
    19     22    19   any
```

4.3 Modelling Mutual Exclusion

```
||GARDEN = (east:TURNSTILE || west:TURNSTILE || {east,west,display}::value:LOCKVAR)
```

Define a mutual exclusion LOCK process:

```
LOCK = (acq -> rel -> LOCK).
```

...and compose it with the shared VAR in the Garden:

```
||LOCKVAR = (LOCK || VAR).
```

Modify **TURNSTILE** to acquire and release the lock:

```
TURNSTILE = (go -> RUN),
RUN       = (arrive -> INCREMENT | end -> TURNSTILE),
INCREMENT = (value.acq
               -> value.read[x:T]
                -> value.write[x+1]
                 -> value.rel->RUN )+{value.write[0]}.
```

Revised Ornamental Garden Model - Checking For Errors

A sample trace:

```
go
east.arrive
east.value.acq
east.value.read.0
east.value.write.1
east.value.rel
west.arrive
west.value.acq
west.value.read.1
west.value.write.2
west.value.rel
end
display.value.read.2
right
```

Use **LTSA** to perform an exhaustive check:
"is TEST satisfied"?

Counter: Abstraction Using Action Hiding

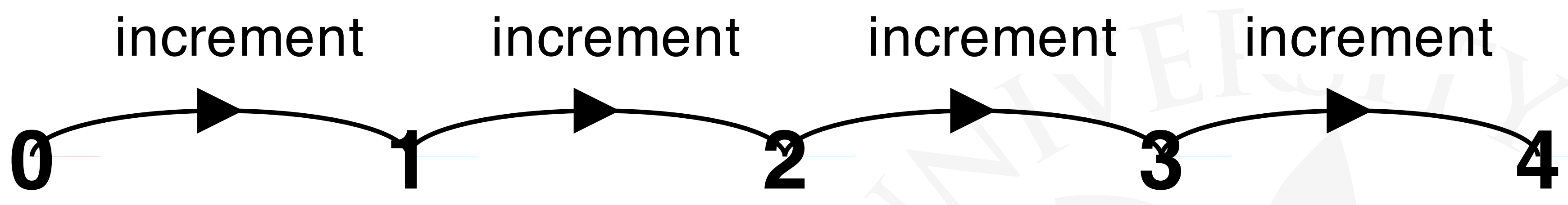
We can abstract the details by hiding.

For `SynchronizedCounter` we hide `read`, `write`, `acquire`, `release` actions.

```
const N = 4
range T = 0..N

VAR = VAR[0],
VAR[u:T] = ( read[u]->VAR[u]
           | write[v:T]->VAR[v]).

LOCK = (acquire->release->LOCK).

INCREMENT = (acquire->read[x:T]
               -> write[x+1]
               -> release->increment->INCREMENT)
                    +{read[T],write[T]}.

||COUNTER = (INCREMENT||LOCK||VAR)@{increment}.
```

Counter: Abstraction Using Action Hiding

Minimised LTS:



We can give a more abstract, simpler description of a **COUNTER** which generates the same LTS:

```
COUNTER = COUNTER[0]
COUNTER[v:T] = (when (v<N) increment -> COUNTER[v+1]).
```

This therefore exhibits “equivalent” behaviour, i.e., has the same observable behaviour.

Active & Passive Processes

Comparing FSP and Java

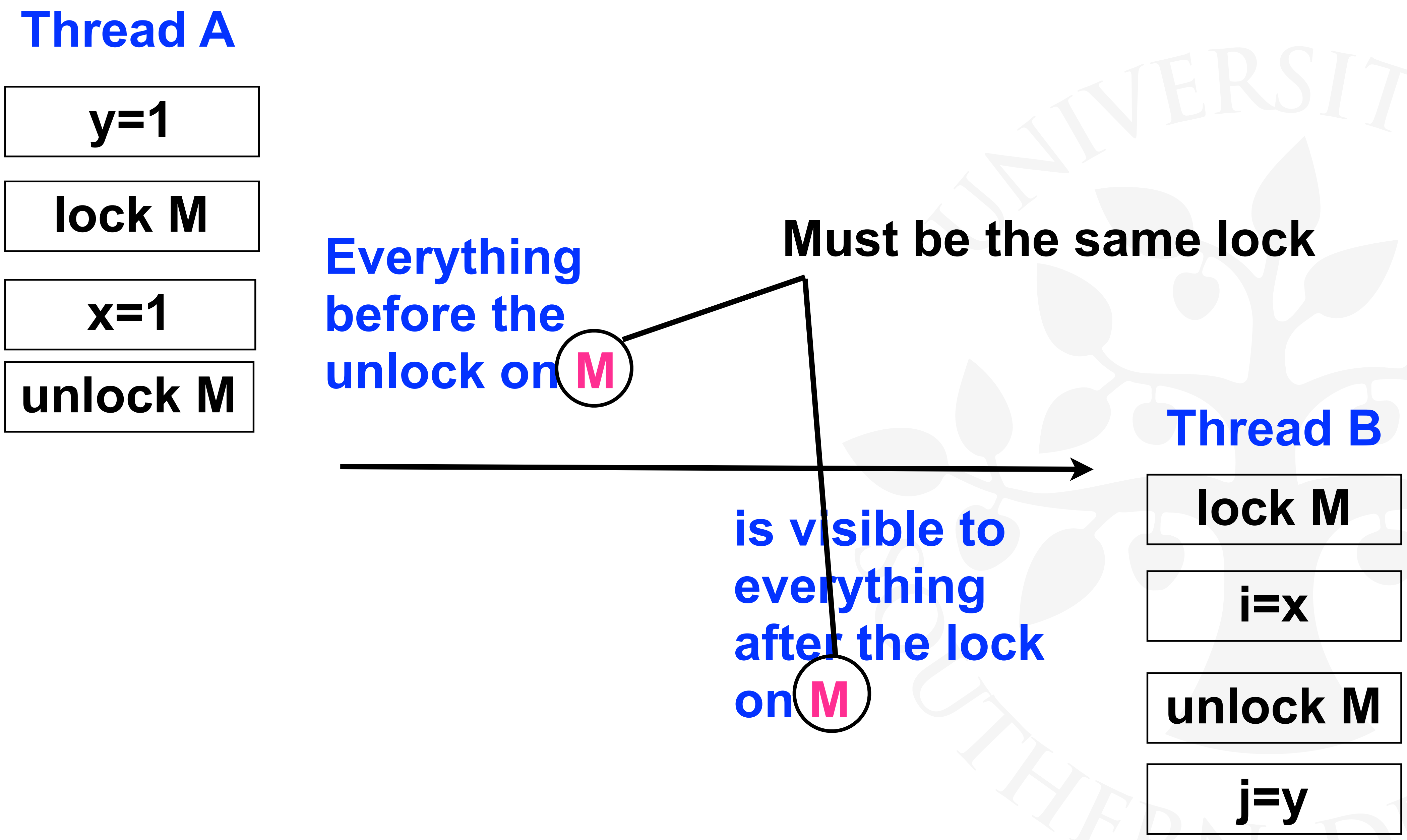
- active processes : threads, e.g., TURNSTILE
- passive processes: shared objects, e.g., COUNTER

```
const N = 4
range T = 0..N
set VarAlpha = {value.{read[T],write[T],acquire,release}}

VAR = VAR[0], VAR[u:T] = (read[u]->VAR[u] | write[v:T]->VAR[v]).
LOCK = (acquire->release->LOCK).
||LOCKVAR = (LOCK || VAR).

TURNSTILE = (go -> RUN),
RUN       = (arrive-> INCREMENT | end -> TURNSTILE),
INCREMENT = (value.acquire
             -> value.read[x:T]->value.write[x+1]
             ->value.release->RUN)+VarAlpha.

DISPLAY =(value.read[T]->DISPLAY)+{value.{write[T],acquire,release}}.

||GARDEN = (east:TURNSTILE || west:TURNSTILE || display:DISPLAY
            || {east,west,display}::value:LOCKVAR)
            /{go /{east,west}.go,
              end/{east,west}.end}.
```

Java Memory Model

```
public class NoVisibility {
    private static boolean ready;
    private static int number;

    private static class ReaderThread extends Thread {
        public void run() {
            while (!ready) {
                yield();
            }
            System.out.println(number);
        }
    }

    public static void main(String[] args) {
        new ReaderThread().start();
        number = 42;
        ready = true;
    }
}
```

Synchronisation In Java Is Not Just Mutual Exclusion; It's Also About Memory Visibility



Thread A

y=1

lock M

x=1

unlock M

Everything before the unlock on M

Must be the same lock

Thread B

lock M

i=x

unlock M

j=y

is visible to everything after the lock on M

Without synchronisation, there is no such guarantee.

Summary

- Concepts
  - process interference
  - mutual exclusion
- Models
  - model checking for interference
  - modelling mutual exclusion
- Practice
  - thread interference in shared Java objects
  - mutual exclusion in Java (synchronized objects/methods).