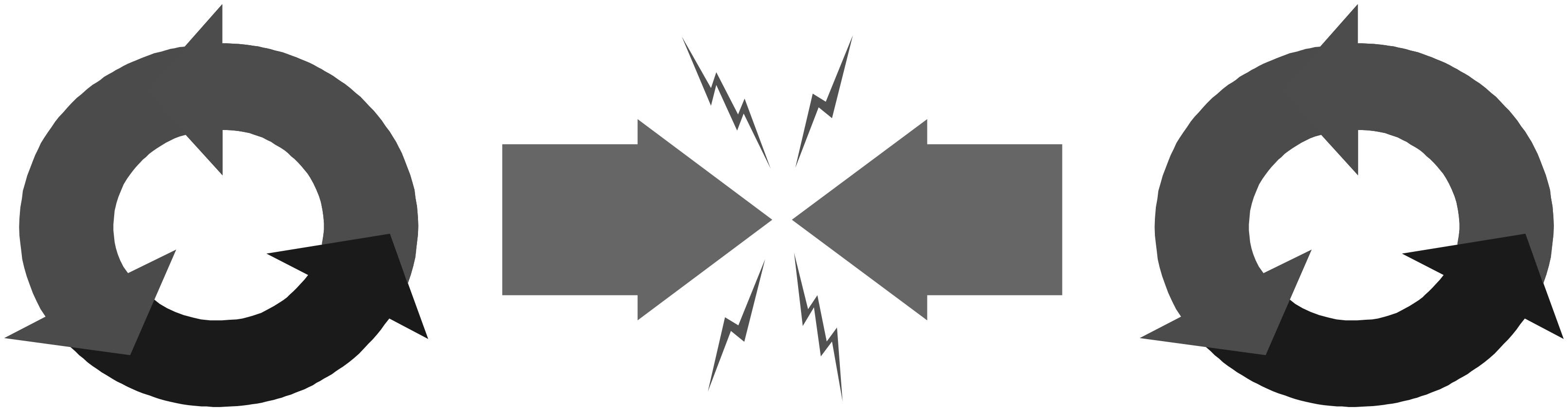
4 - Shared Objects & Mutual Exclusion



Alexandre David
adavid@cs.aau.dk

Credits for the slides:
Claus Braband
Jeff Magee & Jeff Kramer

Repetition – "Concurrent Execution"

Concepts: pseudo- vs. real concurrent execution
concurrent execution and **interleaving**
process interaction

Models: **parallel composition** of asynchronous processes
- interleaving

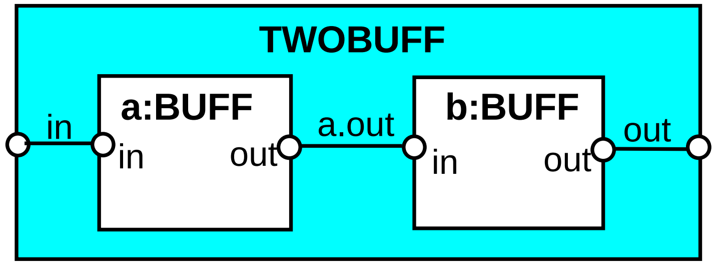
interaction - ***shared actions***
process labeling, action relabeling, and hiding
structure diagrams

Practice: Multithreaded Java programs

Repetition (week 06) - Specifically

- FSP:
  - P || Q // parallel composition
  - a:P // action prefixing
  - {...}::P // set prefixing
  - P / {x/y} // action relabling
  - P \ {...} // hiding
  - P @ {...} // keeping (hide complement)

- Structure Diagrams:



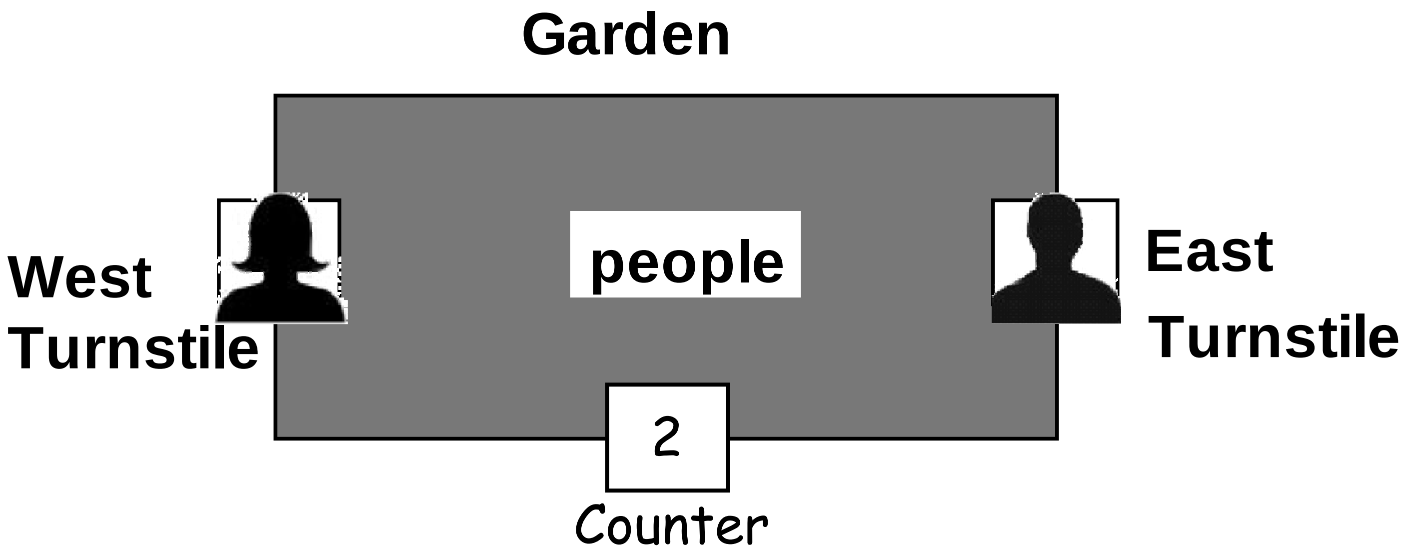
Shared Objects & Mutual Exclusion

- 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
  - synchronized objects, methods, and statements

4.1 Interference

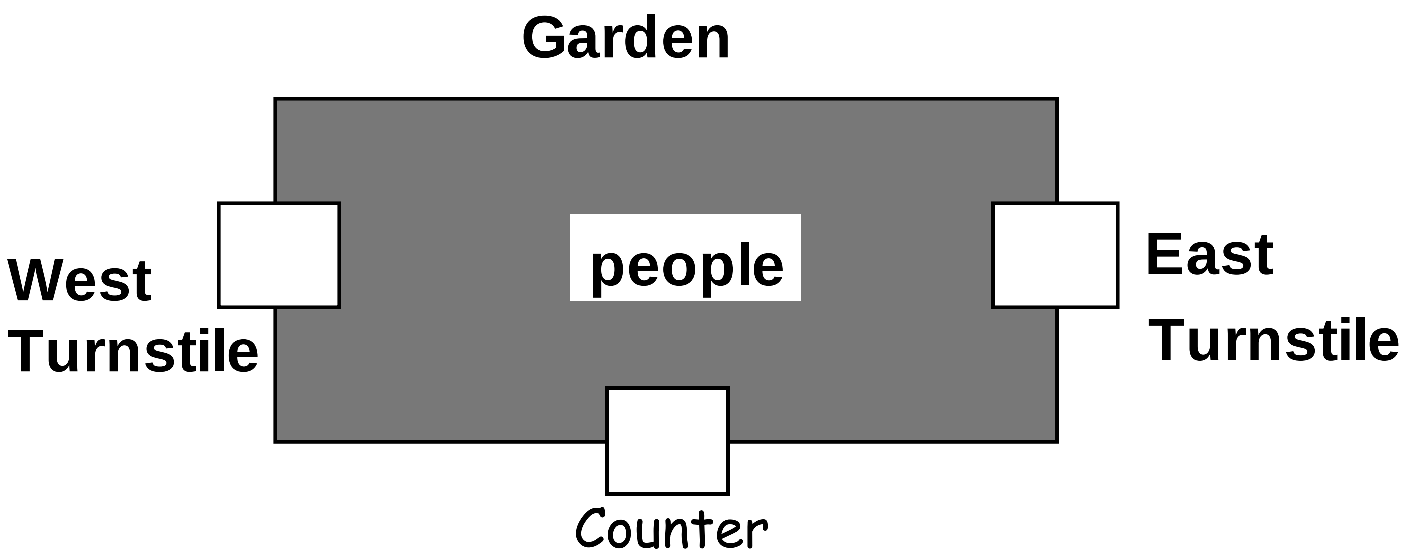
The "*Ornamental Garden Problem*":

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).



Exercise: variant with Entrance/Exit instead of West/East...

4.1 Ornamental Garden Problem (cont'd)

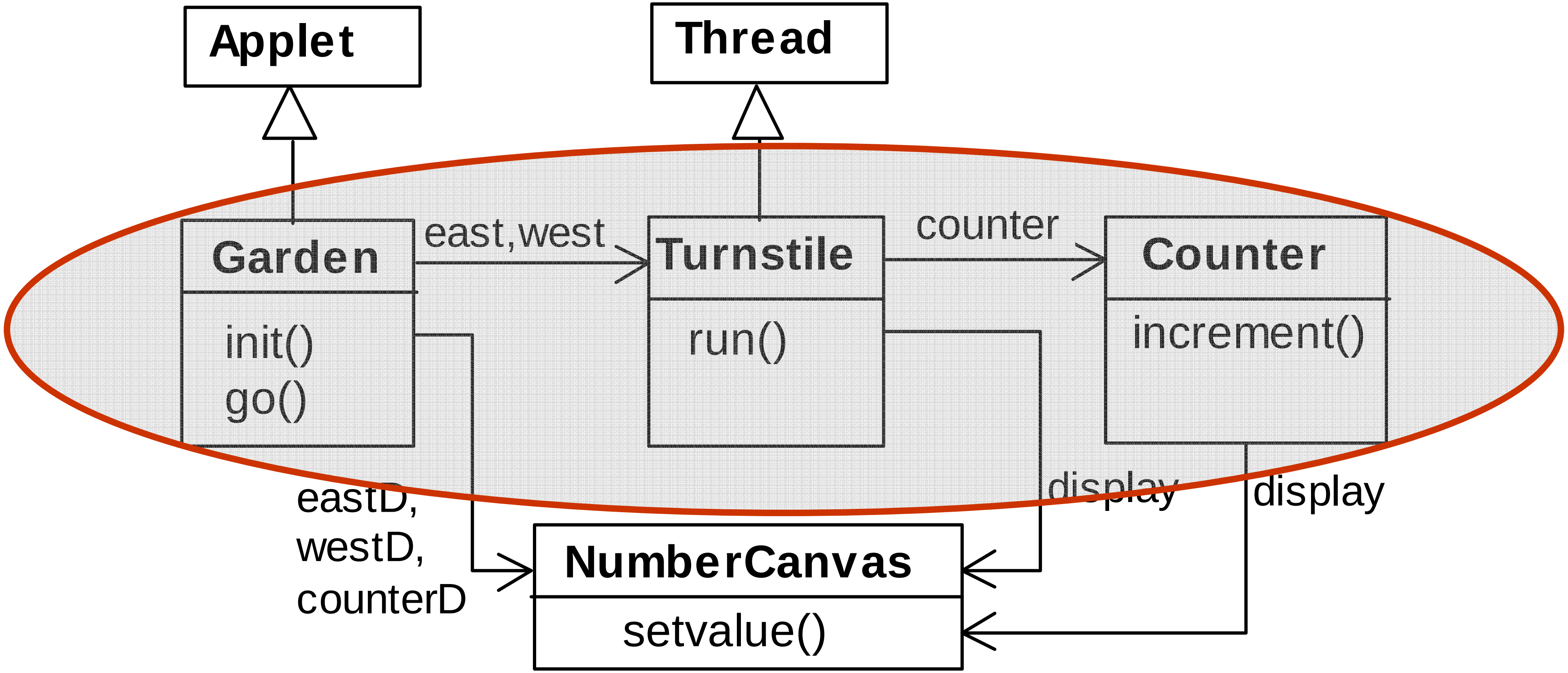


Java implementation:

The concurrent program consists of:

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

Class Diagram



Ornamental Garden Program

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

```
class Garden extends Applet {
    NumberCanvas counterD, westD, eastD;
    …
    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);
    }
}
```

Counter class – Well, Actually…

```
class Counter {
  int value=0;
  NumberCanvas display;

  Counter(NumberCanvas n) {
    display=n;
    display.setvalue(value);
  }

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

Hardware interrupts can occur at **arbitrary** times.

The **`counter`** simulates a hardware interrupt during an **`increment()`**, between reading and writing to the shared counter **`value`**. Interrupt randomly calls **`Thread.yield()`** to force a thread switch.

Running the Applet

After the East and West turnstile threads each have incremented the counter **20 times**, the garden people counter is **not 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);
    }
}
```

Thread switch!

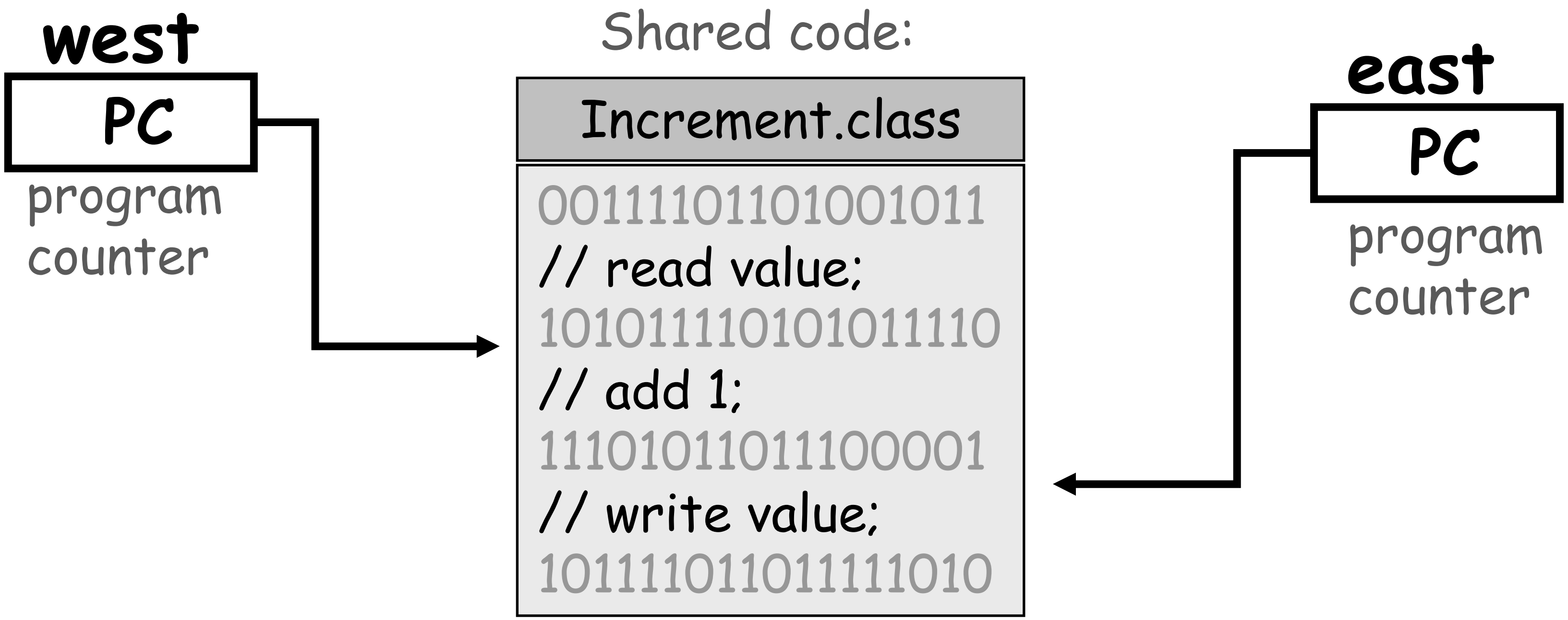
```
value = value + 1;      // 1) read value
value = value + 1;      // 2) add one
value = value + 1;      // 3) write result
```

Recall: *thread switching* (or hardware interrupts) can occur at **any** time

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.



Counter Class: How to Exhibit this Behaviour?

```
class Counter {
    void increment() {

        value = value + 1;

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

Counter Class: How to Exhibit this Behaviour?

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

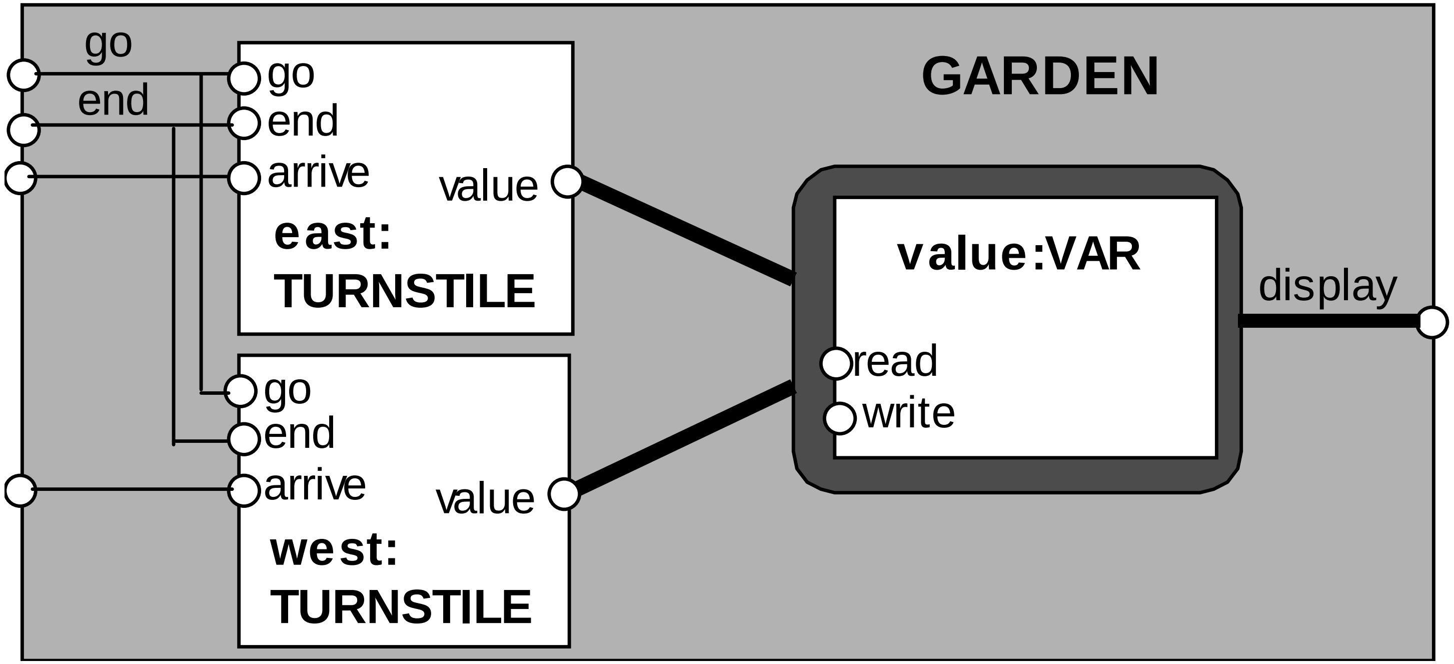
The **counter** simulates a *hardware interrupt* during an **increment()**, between reading and writing to the shared counter **value**.

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

Running the Applet

The erroneous behaviour occurs all the time!

Ornamental 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)

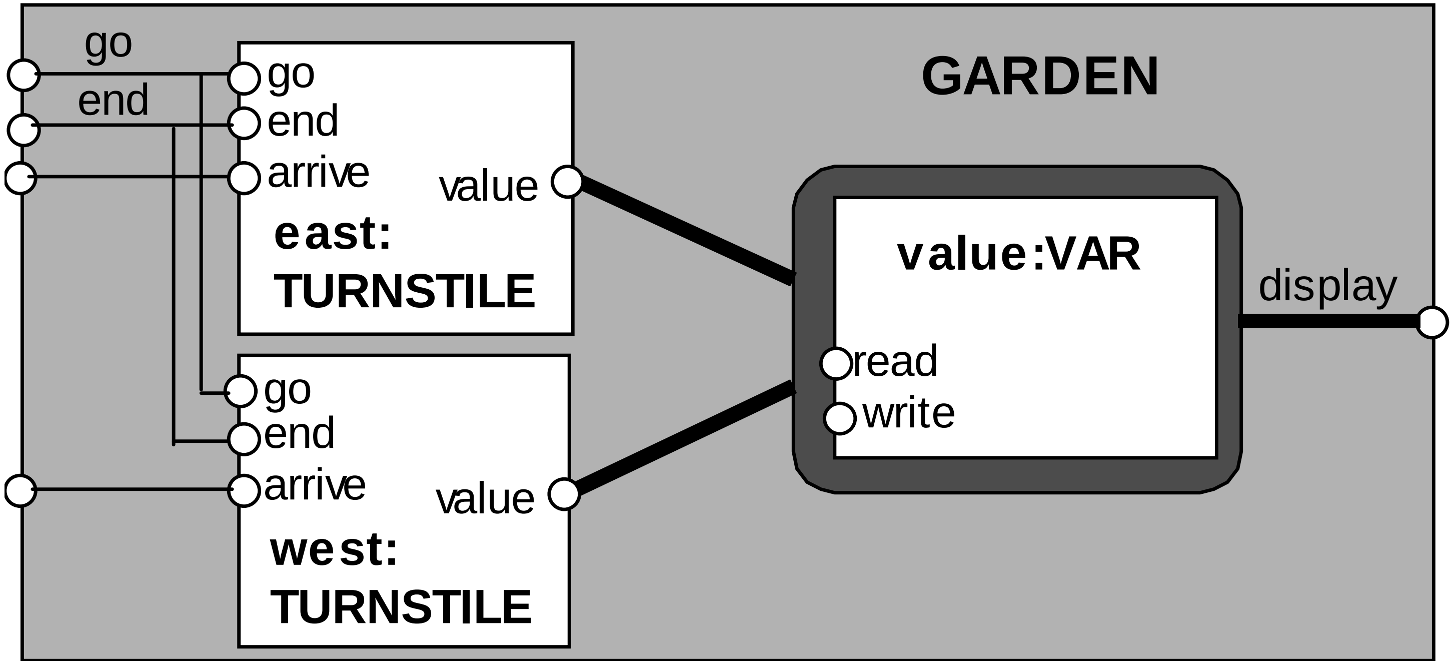
```
const N = 4
range T = 0..N
set VarAlpha = { value.{read[T],write[T]} }

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
            )+VarAlpha.

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

The alphabet of process **VAR** is declared explicitly as a **set** constant, **VarAlpha**.

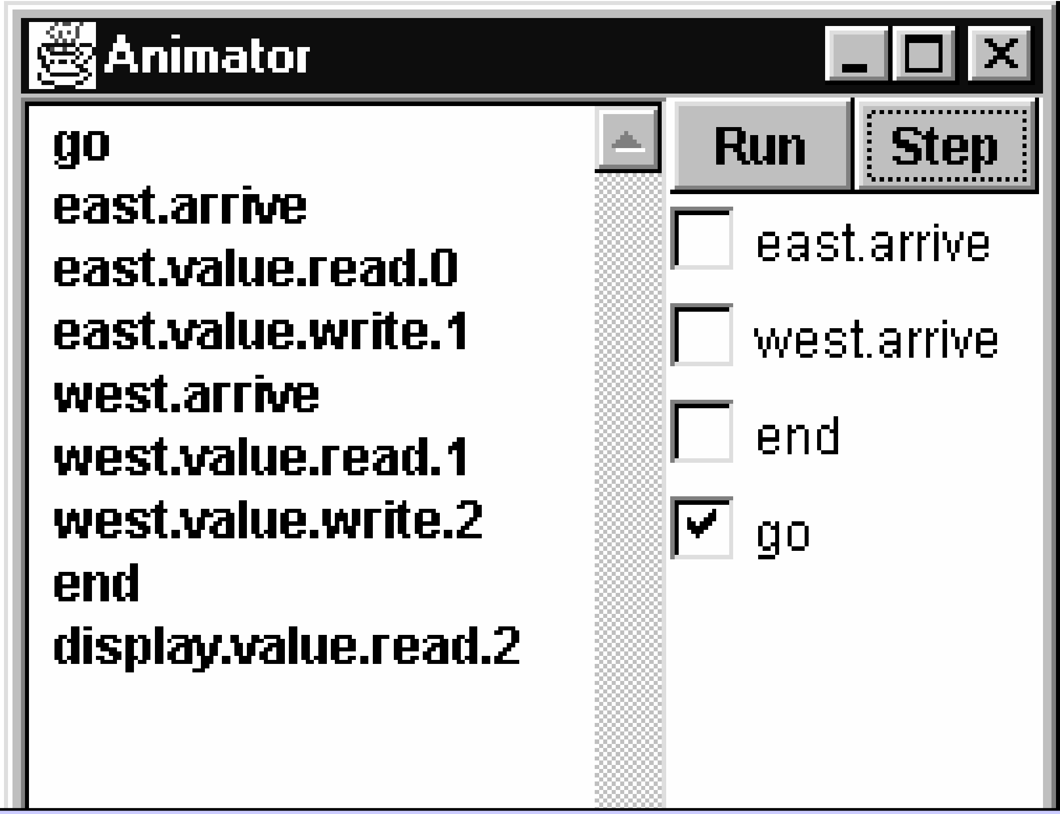
The alphabet of **TURNSTILE** is extended with **VarAlpha** to ensure no unintended free actions in **VAR** ie. all actions in **VAR** must be controlled by a **TURNSTILE**.

Ornamental Garden Model (Structure Diagram)

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



Checking for Errors - Animation



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){east.arrive,west.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
       )
    )+{display.VarAlpha}.
```

Like **STOP**, **ERROR** is a predefined FSP local process (state), numbered -1 in the equivalent LTS.

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 before the method returns.

Concurrent threads are blocked until the lock is released.

Java `synchronized` Statement

Access to an object may also be made mutually exclusive by using the **`synchronized`** statement:

```
synchronized (object) { statements }
```

A less elegant way to correct the example would be to modify the **`Turnstile.run()`** method:

```
synchronized(counter) {counter.increment();}
```

Why is this "less elegant"?

To ensure mutually exclusive access to an object, **all object methods** should be synchronized.

Java synchronized Statement

Synchronized methods:

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

Variant - the synchronized *statement* :

object reference

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

Use synch *methods* whenever possible.

4.3 Modeling Mutual Exclusion

Define a mutual exclusion LOCK process:

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

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

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

Update the alphabet set:

```
set VarAlpha = {value.{read[T],write[T], acq, rel}}.
```

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 )+VarAlpha.
```

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"?

Yes! No error found!

COUNTER: Abstraction Using Action Hiding

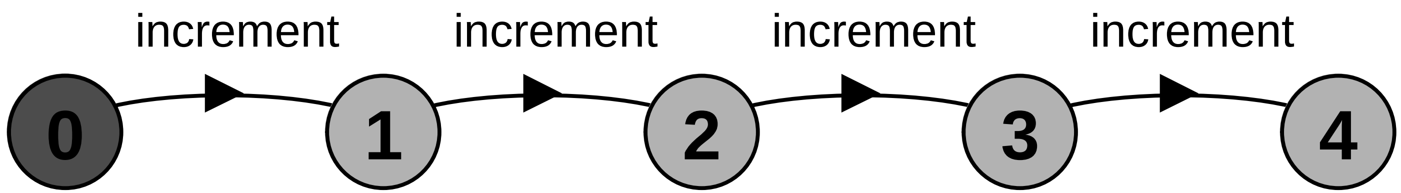
```
const N = 4
range T = 0..N

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

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

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

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

To model shared objects directly in terms of their synchronized methods, we can abstract the details by hiding.

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

COUNTER: Abstraction Using Action Hiding

Minimized 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" behavior i.e. has the same observable behavior.

Summary

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