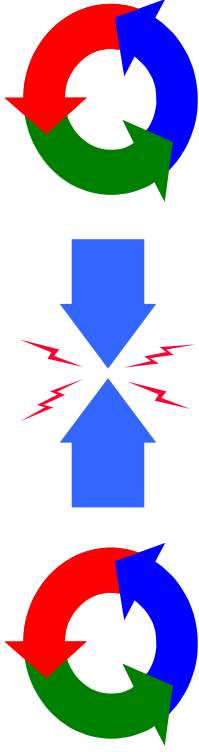


Shared Objects & Mutual Exclusion



Concepts: process interference.
mutual exclusion.

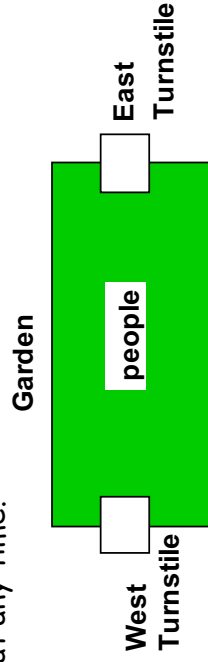
Models: model checking for interference
modeling mutual exclusion

Practice: thread interference in shared Java objects
mutual exclusion in Java
(synchronized objects/methods).

4.1 Interference

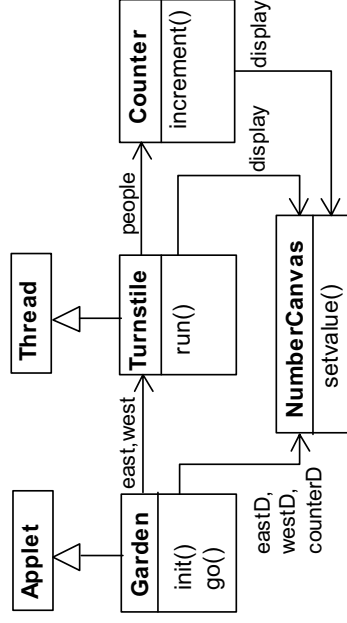
Ornamental garden problem:

People enter an ornamental garden through either of two turnstiles. Management wish to know how many are in the garden at any time.



The concurrent program consists of two concurrent threads and a shared counter object.

ornamental garden Program - class diagram



The **Turnstile** thread simulates the periodic arrival of a visitor to the garden every second by sleeping for a second and then invoking the **increment ()** method of the counter object.

ornamental garden program

The **Counter** object and **Turnstile** threads are created by the **go()** method of the Garden applet:

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

Note that **counterD**, **westD** and **eastD** are objects of **NumberCanvas** used in chapter 2.

Concurrency: shared objects & mutual exclusion

5
©Magee/Kramer 2nd Edition

Counter class

```
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.sleep()** to force a thread switch.

Concurrency: shared objects & mutual exclusion

7
©Magee/Kramer 2nd Edition

Turnstile class

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

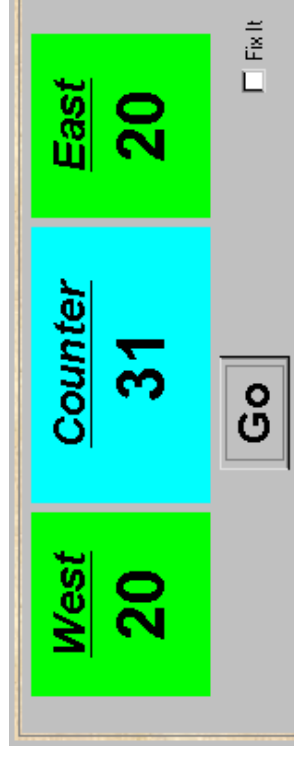
    Turnstile(NumberCanvas n,Counter c)
    { display = n; people = c; }

    public void run() {
        try{
            display.setvalue(0);
            for (int i=1;i<=Garden.MAX;i++){
                Thread.sleep(500); //0.5 second between arrivals
                display.setvalue(i);
                people.increment();
            }
        } catch (InterruptedException e) {}
    }
}
```

The **run()** method exits and the thread terminates after **Garden.MAX** visitors have entered.

©Magee/Kramer 2nd Edition

ornamental garden program - display



After the East and West turnstile threads have each incremented its counter 20 times, the garden people counter is not the sum of the counts displayed. Counter increments have been lost. **Why?**

Concurrency: shared objects & mutual exclusion

8
©Magee/Kramer 2nd Edition

concurrent method activation

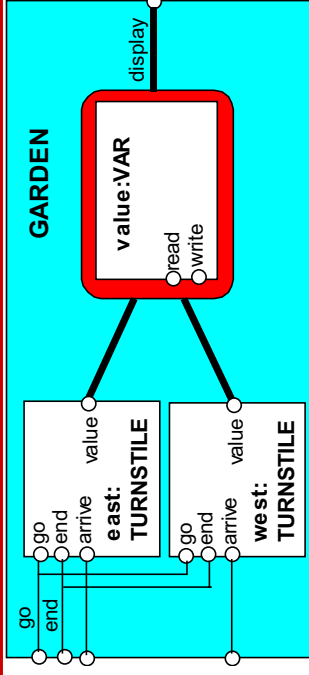
Java method activations are not atomic - thread objects **east** and **west** may be executing the code for the increment method at the same time.



Concurrency: shared objects & mutual exclusion

9 ©Magee/Kramer 2nd Edition

ornamental garden Model



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

Increment is modeled inside **TURNSTILE** since Java method activations are not atomic i.e. thread objects **east** and **west** may interleave their **read** and **write** actions.

Concurrency: shared objects & mutual exclusion

10 ©Magee/Kramer 2nd Edition

ornamental garden model

```

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} .

```

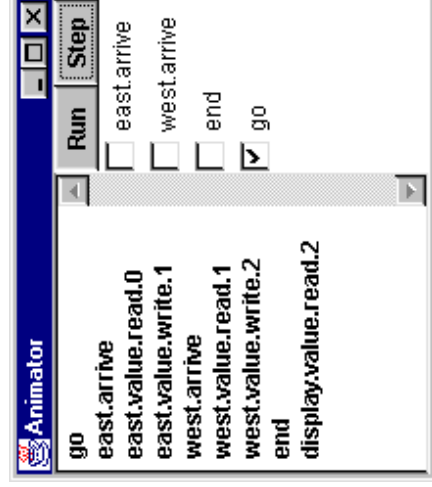
The alphabet of shared process **VAR** is declared explicitly as a **set** constant, **VarAlpha**.

The **TURNSTILE** alphabet is extended with **VarAlpha** to ensure no unintended **free (autonomous) actions** in **VAR** eg. **value.write[0]**. All actions in the shared **VAR** must be controlled (shared) by a **TURNSTILE**.

Concurrency: shared objects & mutual exclusion

©Magee/Kramer 2nd Edition

checking for errors - animation



Scenario checking - use animation to produce a trace.

Is this trace correct?

Concurrency: shared objects & mutual exclusion

12 ©Magee/Kramer 2nd Edition

checking for errors - exhaustive analysis

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.

Concurrency: shared objects & mutual exclusion

13

©Magee/Kramer 2nd Edition

ornamental garden model - checking for errors

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

Use *L TSA* 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
```

L TSA produces the shortest path to reach **ERROR**.

Concurrency: shared objects & mutual exclusion

14

©Magee/Kramer 2nd Edition

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 to give methods **mutually exclusive** access to shared objects. Mutual exclusion can be modeled as atomic actions.

Concurrency: shared objects & mutual exclusion

15

©Magee/Kramer 2nd Edition

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**, which uses a lock on the object.

We correct **COUNTER** class by deriving a class from it and making the increment method **synchronized**:

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

acquire lock

release lock

Concurrency: shared objects & mutual exclusion

16

©Magee/Kramer 2nd Edition

mutual exclusion - the ornamental garden



Java associates a *lock* with every object. The Java compiler inserts code to acquire the lock before executing the body of the synchronized method and code to release the lock before the method returns. Concurrent threads are blocked until the lock is released.

Concurrency: shared objects & mutual exclusion

17

©Magee/Kramer 2nd Edition

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 (people) { people.increment(); }
```

Why is this “less elegant”?

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

Concurrency: shared objects & mutual exclusion

18

©Magee/Kramer 2nd Edition

4.3 Modeling mutual exclusion

To add locking to our model, define a `LOCK`, compose it with the shared `VAR` in the garden, and modify the alphabet set :

```
LOCK = (acquire->release->LOCK) .  
|| LOCKVAR = (LOCK || VAR) .  
set VarAlpha = {value.{read[T],write[T],  
                  acquire, release}}
```

Modify `TURNSTILE` to acquire and release the lock:

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

Concurrency: shared objects & mutual exclusion

19

©Magee/Kramer 2nd Edition

Revised ornamental garden model - checking for errors

A sample animation execution trace

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

Use `TEST` and `L TSA` to perform an exhaustive check.

Is TEST satisfied?

Concurrency: shared objects & mutual exclusion

20

©Magee/Kramer 2nd Edition

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.

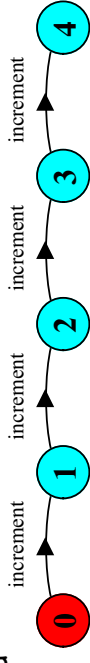
Concurrency: shared objects & mutual exclusion

21

©Magee/Kramer 2nd Edition

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.

Concurrency: shared objects & mutual exclusion

22

©Magee/Kramer 2nd Edition

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).

Concurrency: shared objects & mutual exclusion

23

©Magee/Kramer 2nd Edition