Shared Objects & Mutual Exclusion

INF2140 Parallel Programming: Lecture 4

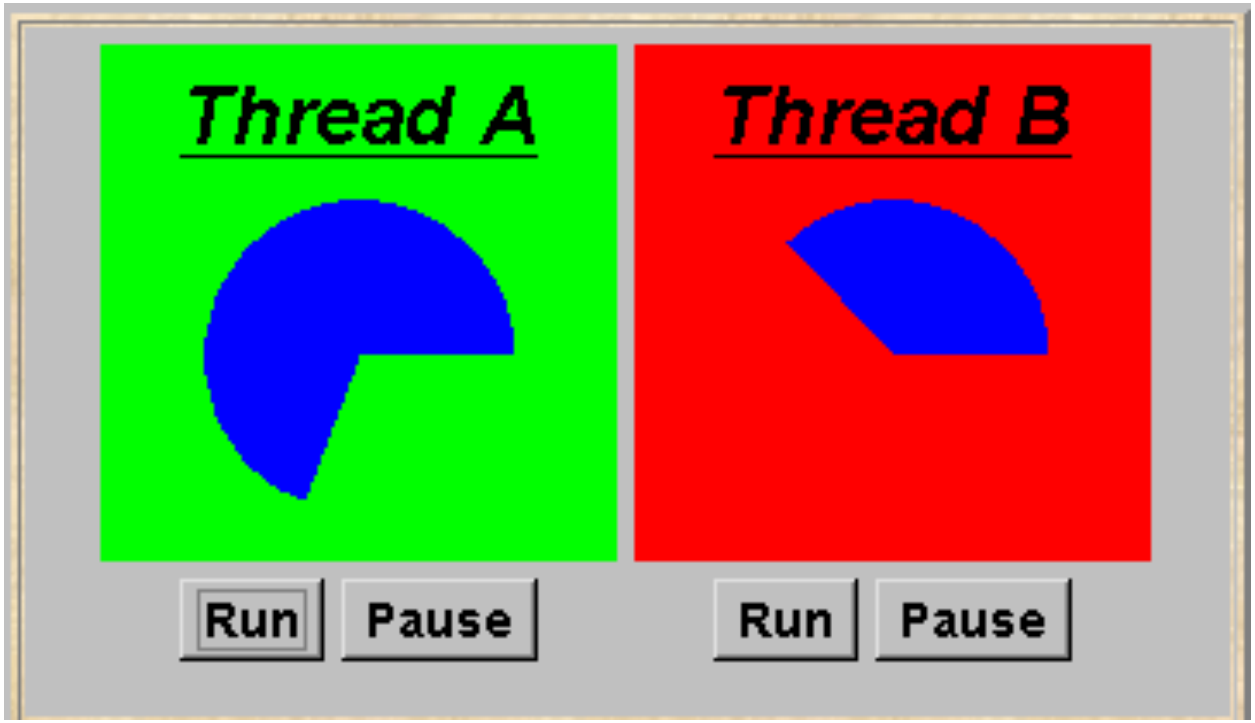
Feb. 08, 2012

Concurrent Execution

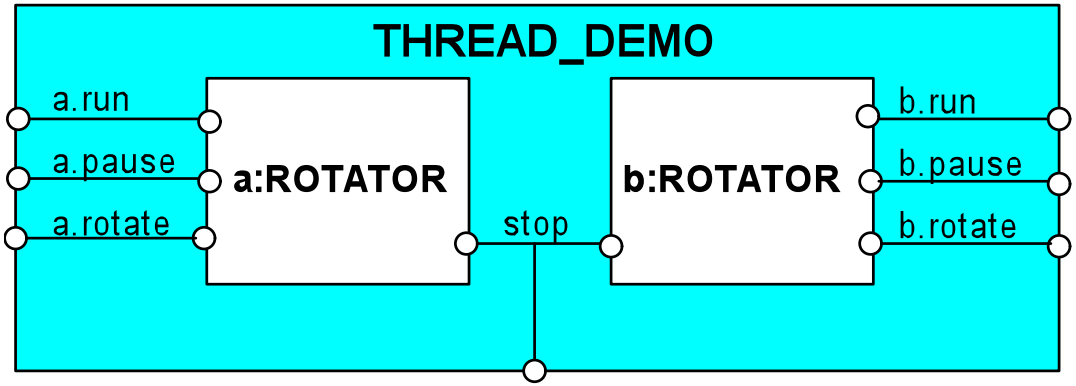
- Concepts
  - Process interference
  - Mutual exclusion
- Models
  - Model checking for interference
  - Modeling mutual exclusion
- Practice
  - Multithreaded Java programs
  - Thread interference in shared Java objects
  - Mutual exclusion in Java (synchronized objects and methods)

Multi-threaded Programs in Java

Concurrency in Java occurs when more than one thread is alive.
ThreadDemo has two threads which rotate displays.



ThreadDemo Model

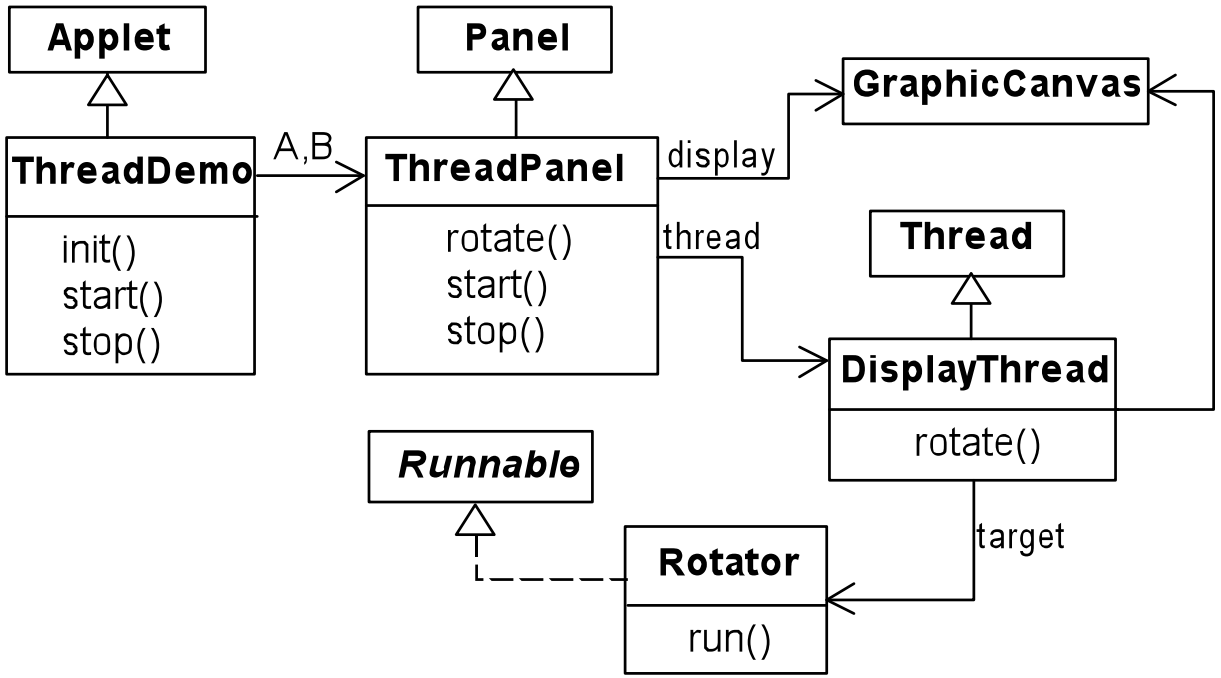


Interpret run, pause, interrupt as inputs, rotate as an output.

```
ROTATOR = PAUSED,
PAUSED  = (run->RUN | pause->PAUSED
          |interrupt->STOP),
RUN     = (pause->PAUSED |{run,rotate}->RUN
          |interrupt->STOP).
||THREAD_DEMO = (a:ROTATOR || b:ROTATOR)
                /{stop/{a,b}.interrupt}.
```

ThreadDemo: Implementation in Java

`ThreadDemo` creates two `ThreadPanel` displays when initialized.
`ThreadPanel` manages the display and control buttons, and delegates calls to `rotate()` to `DisplayThread`.
`Rotator` implements the `runnable` interface.



Rotator Class

```
class Rotator implements Runnable {

  public void run() {
    try {
      while(true) ThreadPanel.rotate();
    } catch(InterruptedException e) { }
  }
}
```

`Rotator` implements the `runnable` interface, calling `ThreadPanel.rotate()` to move the display.

`run()` finishes if an exception is raised by `Thread.interrupt()`.

ThreadPanel Class

```
public class ThreadPanel extends Panel {
  // construct display with title and segment color c
  public ThreadPanel(String title, Color c) {...}
  // rotate display of current thread 6 degrees
  public static boolean rotate()
    throws InterruptedException {...}
  // create and start a new thread with target r
  public void start(Runnable r) {
  thread = new DisplayThread(canvas,r,...);
  thread.start();}
  // stop the thread using Thread.interrupt()
  public void stop() {thread.interrupt();}}
```

ThreadPanel manages the display and control buttons for a thread.
Calls to rotate() are delegated to DisplayThread.
Threads are created by start(), and terminated by stop().

ThreadDemo Class

```
public class ThreadDemo extends Applet {
  ThreadPanel A; ThreadPanel B;
  public void init() {
    A = new ThreadPanel("Thread A",Color.blue);
    B = new ThreadPanel("Thread B",Color.blue);
    add(A); add(B);}
  public void start() {
    A.start(new Rotator());
    B.start(new Rotator());}
  public void stop() {
    A.stop(); B.stop(); }}
```

ThreadDemo creates two ThreadPanel displays when initialized and two threads when started.

ThreadPanel is used extensively in later demonstration programs.

Interference

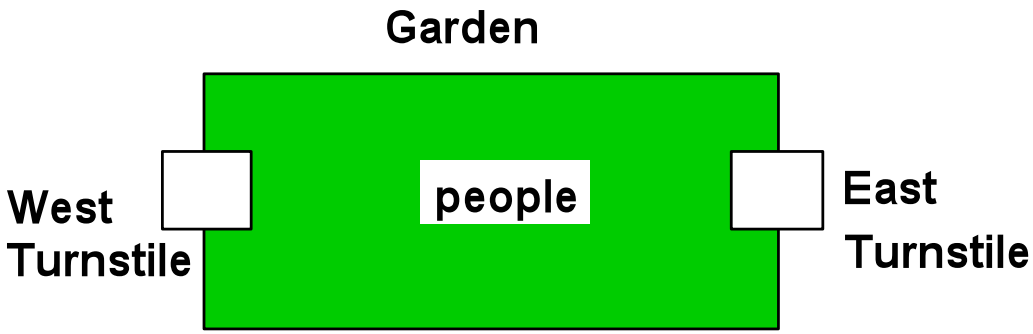
- So far, we have discussed the execution of multiple processes, modeling concurrent execution by interleaving and executing multiple threads in Java
- Process interaction modeled by **shared atomic actions**
- How do real processes or threads interact?

The simplest way for Java threads to interact, is via a shared object: an object whose methods can be invoked by a set of threads

Ornamental Garden Problem

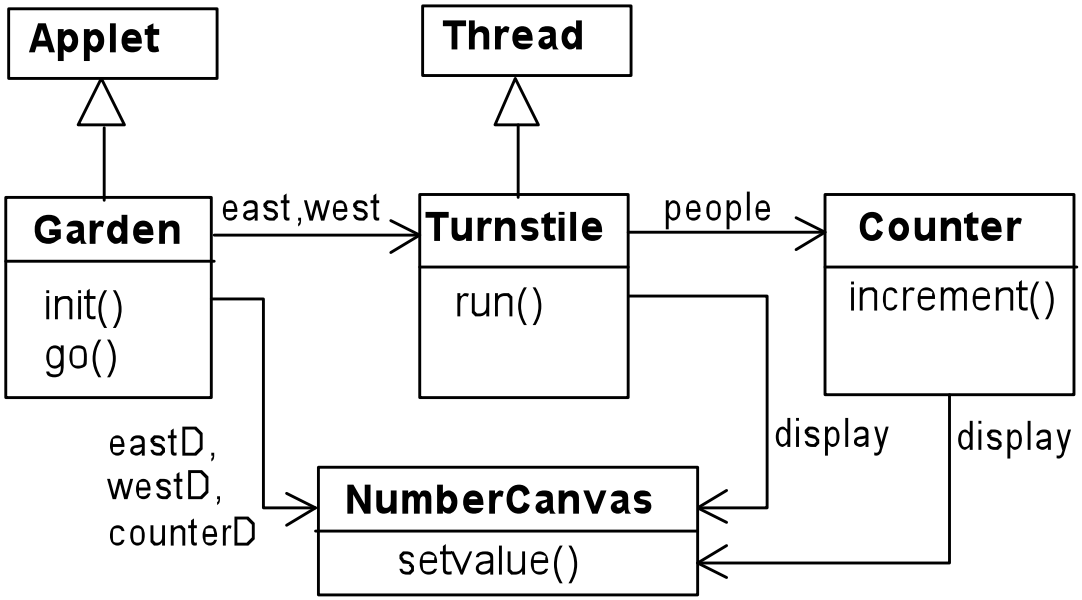
People enter an ornamental garden through either of two turnstiles.

Management wants to know how many people are in the garden at any time.



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

Ornamental Garden Problem: 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 class `NumberCanvas` (used in Chapter 2).

Ornamental Garden: The 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++){
        // sleep 0.5 sec. between arrivals
        Thread.sleep(500);
        display.setvalue(i); people.increment();}
    } catch (InterruptedException e){ }}
}
```

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

Ornamental Garden: The 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.

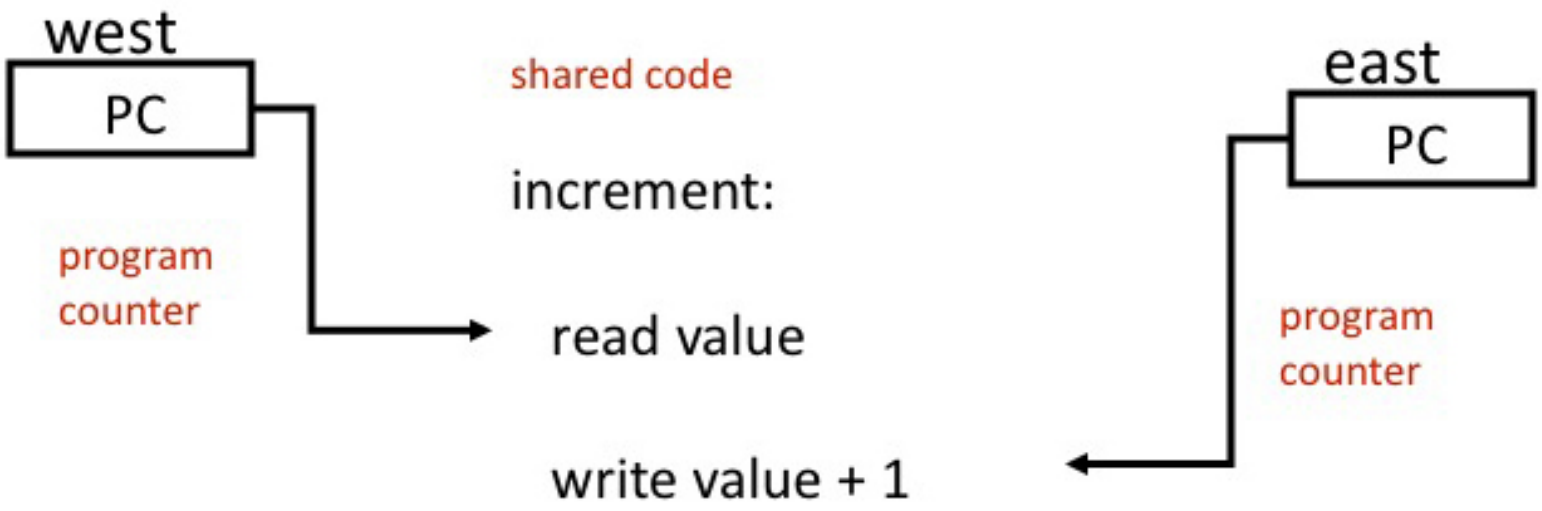
Ornamental Garden Program: Display

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?

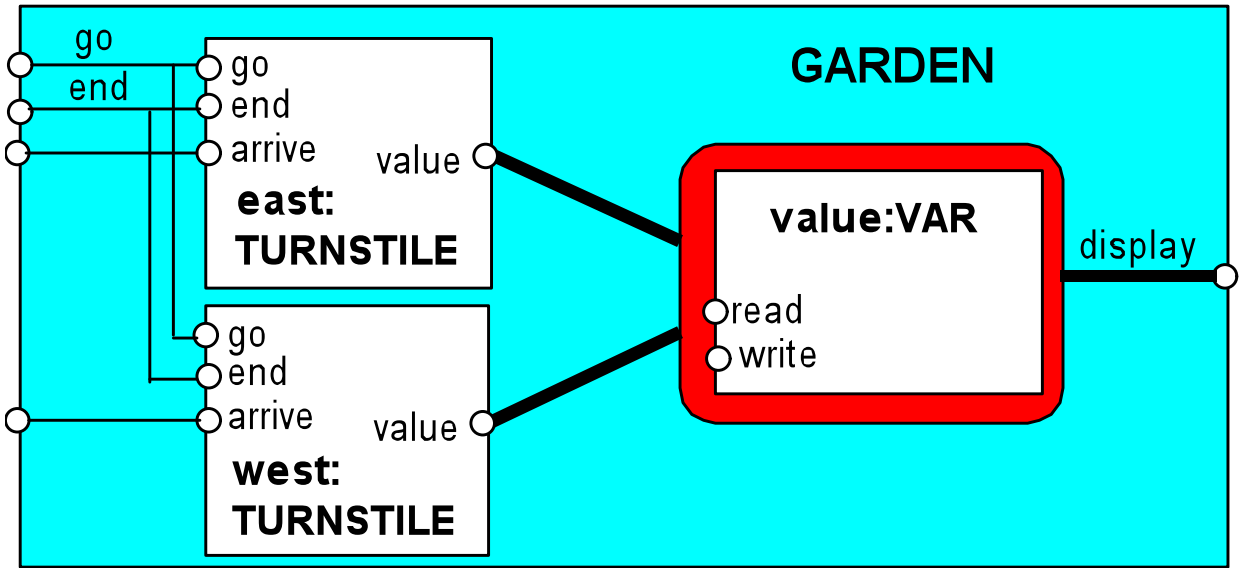
Concurrent Method Activations

Java method activations are not atomic:

The thread objects `east` and `west` may be executing the code for the `increment` method *at the same time.*



Ornamental Garden Model (1)



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

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.

Ornamental Garden Model (2)

```
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]).
```

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

Ornamental Garden Model (3)

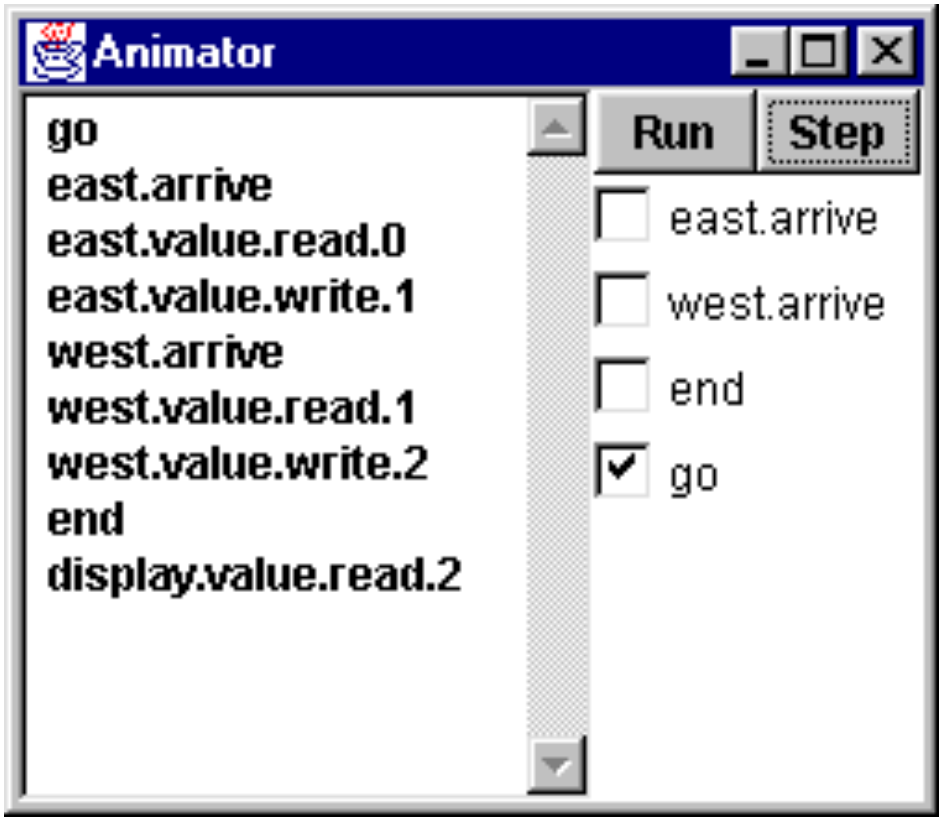
```
TURNSTILE = (go -> RUN),
RUN       = (arrive-> INCREMENT
            |end -> TURNSTILE),
INCREMENT = (value.read[x:T]
             -> value.write[x+1]->RUN
            )+VarAlpha.
```

The TURNSTILE alphabet is *extended* with VarAlpha to ensure no unintended free (autonomous) actions in VAR; e.g., value.write[0].

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

All actions in the shared VAR must be controlled (shared) by a TURNSTILE.

Checking for Errors: Animation



Scenario checking: use animation to produce a trace.

Is this trace correct?

Checking for Errors: Exhaustive Analysis (1)

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 (2)

```
||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 ERROR.

Interference and Mutual Exclusion

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

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 the `COUNTER` class by deriving a class from it, where we make the `increment` method synchronized:

```
class SynchronizedCounter extends Counter {

  SynchronizedCounter(NumberCanvas n)
     {super(n);}

   synchronized void increment() { // acquire lock
         super.increment();
   } // release lock
}
```

Mutual Exclusion: The Ornamental Garden

Java associates a lock with every object. The 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.

Java's 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*

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.
```

Revised Ornamental Garden Model: Checking for Errors

A sample animation execution trace

Use TEST and LTSA to perform an exhaustive check.

Is TEST satisfied?

```
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
```

COUNTER: Abstraction Using Action Hiding (1)

```
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}.
```

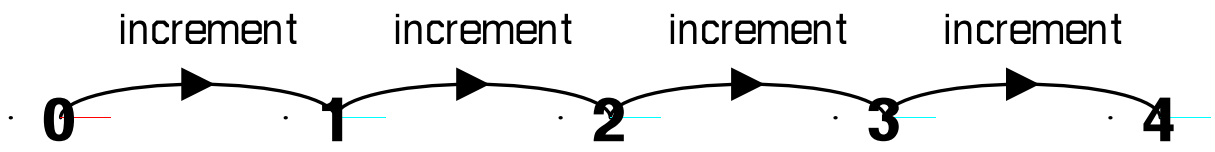
We have seen that shared actions can be made atomic by using locks.

We can abstract the details of locks by hiding, and model shared objects directly in terms of their `synchronized` methods,

`SynchronizedCounter`: we hide `read`, `write`, `acquire`, `release` actions

COUNTER: Abstraction Using Action Hiding (2)

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 model therefore exhibits "equivalent" behavior; i.e., it has the same observable behavior.

Summary

- Concepts
  - Process interference
  - Mutual exclusion
- Models
  - Model checking for interference
  - Modeling mutual exclusion
- Practice
  - Multithreaded Java programs
  - Thread interference in shared Java objects
  - Mutual exclusion in Java (`synchronized` objects and methods)