Concurrent Programming (Part II)
Lecture 4: Java Synchronization

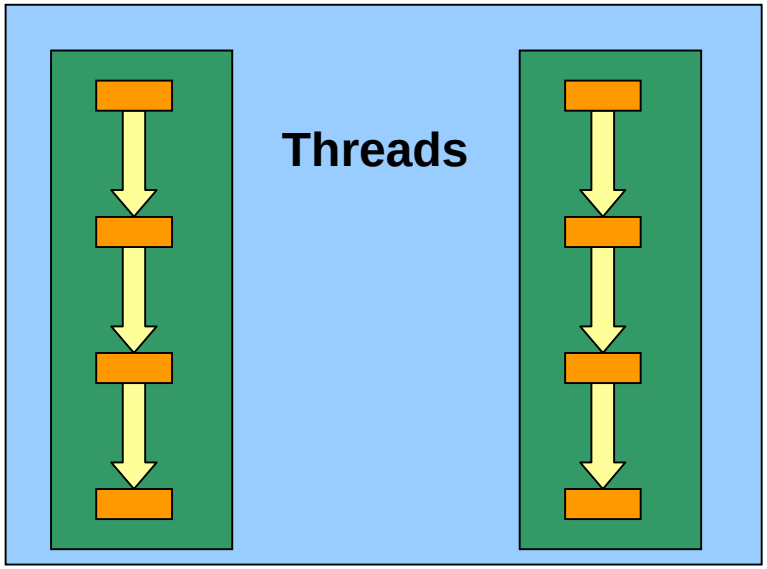
THIS IS AN IMPORTANT LECTURE

Dr Kevin Bryson
K.Bryson@cs.ucl.ac.uk
Room 8.04

Course Web Site on Moodle
http://moodle.ucl.ac.uk/course/view.php?id=753
Enrolment Key: ATOMIC

Recall from the Previous Lecture

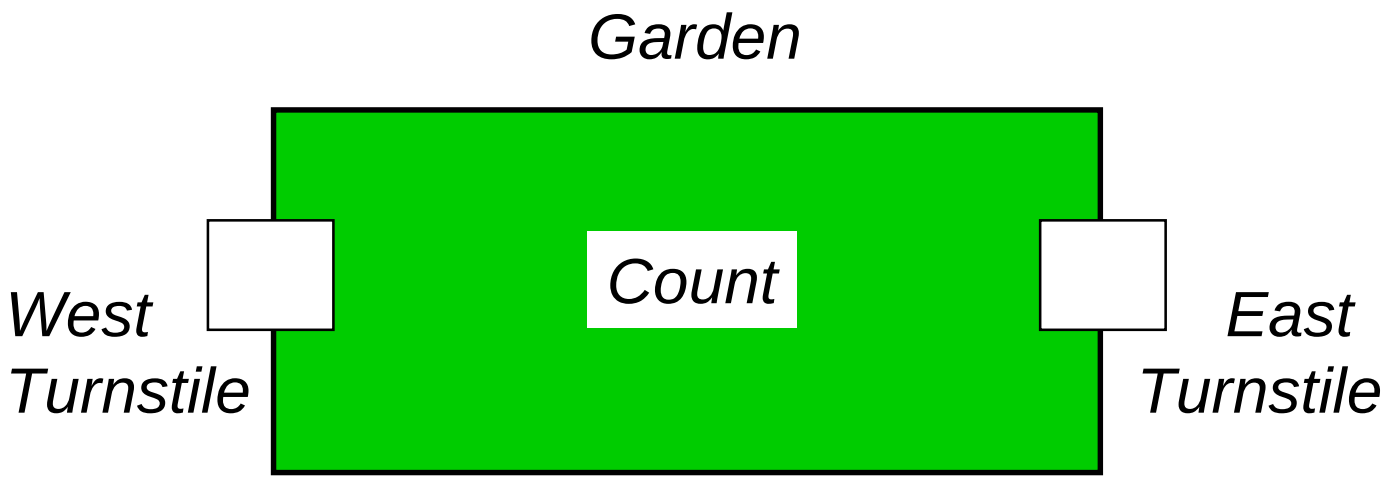
- Two ways of creating Java Threads
- Lifecycle of a Thread
- Demo containing 2 Threads



- But our Threads were largely independent …
- *What happens when we actually allow them to interact?*
- ***How can they interact?***
- *… by using shared variables.*

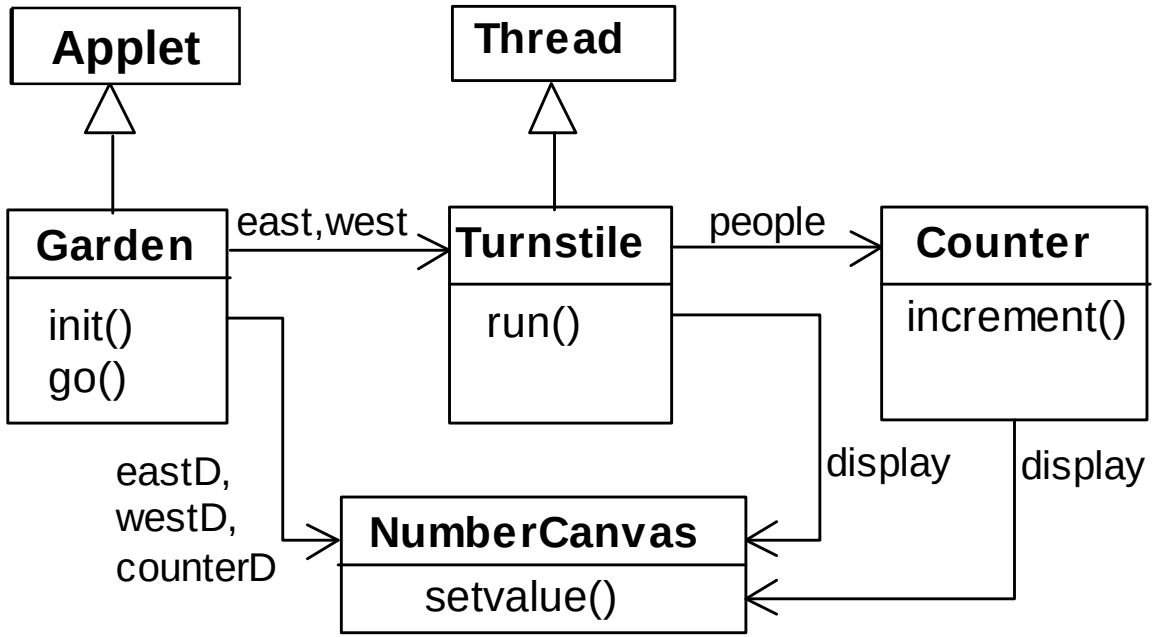
Ornamental Garden Problem (Magee & Kramer Chapter 4)

- Garden open to the public
- People enter through either one of two turnstiles
- Computer counts number of visitors



- Each turnstile implemented by a thread

Ornamental Garden Class Diagram



The **Turnstile** thread simulates the periodic arrival of a visitor to the garden every half second by sleeping for half 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();
    }
```

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) {}
  }
}
```

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.yield()** to force a thread switch.

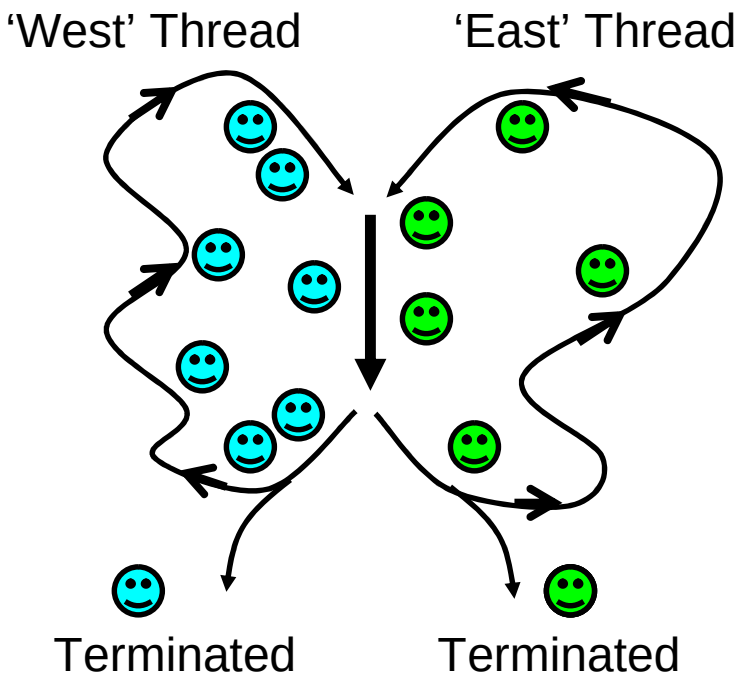
What will happen when people start to enter?

The Problem

- See the Applet Demo at:
  http://www.doc.ic.ac.uk/~jnm/book/book_applets/Garden.html

- **The counter is updated concurrently by the two turnstiles. The increment() method is not atomic!!**

What's going on ? *Interference …*



- One has to remember that the Threads executing within the program will be at different locations within their 'path of execution' …
- When they are going through the same section of code (or just using the same shared variables in different sections of code) *at roughly the same time* … they can ***interfere*** with each other.
- **They can overwrite each others updates** (in a similar manner to the assembly increment instructions shown in Lecture 2).

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 modelled as atomic actions.

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

Critical sections need to be synchronized

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

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

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

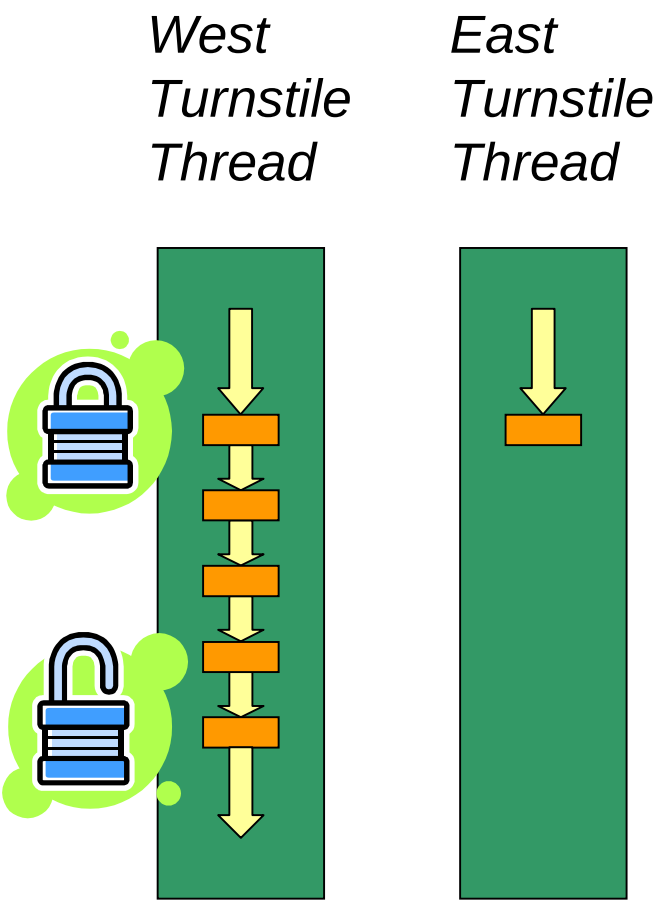
Lets see if it works!

How does synchronization work?

- *West Thread acquires object lock on method entry*
- *East Thread has to wait for the* ***mutually exclusive lock.***

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

- *West Thread releases the lock on exit and East Thread can enter*



What type of Lock are we talking about?

Front Door Lock – not exactly!

Bathroom Lock – closer!
Has mutual exclusion …

Synchronized Methods

- Whenever control enters a synchronized method, the thread that called the method **locks the object** whose method has been called.
- Other threads cannot call a synchronized method on the **same object** until the object is unlocked. *They have to wait (they are in the blocked state).*
- The acquisition and release of a lock is done automatically and atomically by the Java runtime system. This ensures that **race conditions** cannot occur in the underlying implementation of the threads, thus ensuring data integrity.

Locks are *Reentrant*

```
public class Reentrant {
   public synchronized void a() {
    b();
    System.out.println(“Here I am, in a()");
   }
   public synchronized void b() {
    System.out.println(“Here I am, in b()");
   }
 }
```

Reentrant Locks

- The Java runtime system allows a thread to re-acquire a lock that it already holds because Java locks are reentrant.
- Reentrant locks are important because they eliminate the possibility of a single thread **deadlocking** itself on a lock that it already holds.
- **Deadlock** can still occur when two or more Threads are waiting for two or more locks to be freed and the program is in a state where the locks are never freed

One Thread stopping another …

- *Two Threads access the stop attribute … hence the methods need to be synchronized*
- *But can you see what is wrong with this class?*

```
public class MyThread extends Thread {
   private boolean stop = false;

   public synchronized void run() {
      while (!stop) {
         // Do lots of work … Do lots of work …
      }
   }

   public synchronized void setStop(boolean b) {
      stop = b;
   }
}
```

One Thread stopping another … DEADLOCK

- The Thread that wants to stop this Thread cannot gain access to the ‘setStop’ method since this Thread has its lock running the ‘run’ method …
- And the ‘run’ method cannot finish and release the lock since the done flag cannot be changed to true
- Catch-22 … Deadlock!

- *How can this be resolved?*

One Thread stopping another … SOLVED

- Make the **Lock Scope** more appropriate

```
public class MyThread extends Thread {
   private boolean stop = false;

   public synchronized void run() {
      while (!getStop()) {
         // Do lots of work … Do lots of work …
      }
   }
   public synchronized boolean getStop() {
      return stop;
   }
   public synchronized void setDone(boolean b) {
      stop = b;
   }
}
```

One Thread stopping another … even better

- No need to synchronize since setting and testing booleans is Atomic
- BUT ‘stop’ attribute needs to be volatile – since the Java Memory Model allows it to be cached, e.g. in a registry, and so it may not update in one Thread when another Thread updates it in main memory
- Synchronization also flushes cached attributes …

```
public class MyThread extends Thread {
   private volatile boolean stop = false;

   public void run() {
      while (!stop) {
         // Do lots of work … Do lots of work …
      }
   }
   public void setDone(boolean b) {
      stop = b;
   }
}
```

Another way to get precise Lock Scope is to make use of the synchronization statement

- The synchronization statement can lock on any given object
- But this is less elegant than method synchronization…

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

Summary

- This lecture was about threads interacting with each other.
- Threads can interact by calling methods on a shared object (i.e. both threads have references to the same shared object). Essentially shared variables are being used by both threads.
- This interaction provides useful functionality (How else would the Gardener keep track of the number of people in his garden!)
- But these types of interaction also comes at a cost …
  the possibility of **interference** (essentially data corruption).
- **Critical sections** of code need to be **synchronized** to prevent **interference**.