Java Threads

- Resources
  - Course textbook: pp. 162-165
  - Java Threads by Scott Oaks & Henry Wong (O'Reilly)
  - API docs
    - http://download.oracle.com/javase/6/docs/api/
      - java.lang.Thread, java.lang.Runnable
      - java.lang.Object, java.util.concurrent
  - Tutorials
    - http://download.oracle.com/javase/tutorial/essential/concurrency/index.html
    - http://download.oracle.com/javase/tutorial/essential/concurrency/procthread.html
  - Introduction to Java Threads
    - http://www.javaworld.com/javaworld/jw-04-1996/jw-04-threads.html
  - Thread safety
    - http://en.wikipedia.org/wiki/Thread-safety
    - http://www.javaworld.com/jw-08-1998/jw-08-techniques.html

Coverage

- Thread class
  - run, start methods
  - yield, join
  - sleep
- Synchronization
  - synchronized methods & objects
  - wait/notify/notifyAll
  - conditions

java.lang.Thread

- Two techniques to create threads in java
- 1) implementing the Runnable interface
  - The Runnable interface should be ***implemented*** by any class whose instances are intended to be executed by a thread. The class must define a method, called *run*, with no arguments.
  - invoke Thread constructor with an instance of this Runnable class
  - See pages 162 and 164 in text for an example
- 2) extending Thread
  - Define a subclass of java.lang.Thread
    - Define a *run* method
  - In another thread (e.g., the main), create an instance of the Thread subclass
    - Then, call *start* method of that instance

Example 1

- Create 2 threads from the Main, then start them
- Threads will be instances of different thread sub-classes

```
class MyThreadA extends Thread {
    public void run() { // entry point for thread
        for (;;) {
            System.out.println("hello  world1");
        }
    }
}

class MyThreadB extends Thread {
    public void run() { // entry point for thread
        for (;;) {
            System.out.println("hello  world2");
        }
    }
}

public class Main1 {
    public static void main(String [] args) {
        MyThreadA t1 = new MyThreadA();
        MyThreadB t2 = new MyThreadB();
        t1.start();
        t2.start();
        // main terminates,  but in Java the other threads keep running
        // and hence Java program continues running
    }
}
```

hello world2
hello world2
hello world1
hello world2
hello world1
hello world2
hello world2
hello world1
hello world1
hello world1
hello world1
hello world2
hello world1
hello world1
hello world2
hello world2
hello world1
hello world1
hello world2
hello world2
hello world1
hello world1
hello world2

Example 2

- Create 2 threads from the Main, then start them
- Threads will be instances of the same thread sub-class
- Use argument of constructor of new thread class to pass text name of thread, e.g., “thread1” and “thread2”
  - Data member provides different data per thread (i.e., then name)
  - A data member can also be used to share data

```
class MyThread extends Thread {
	private String name;

	public MyThread(String name) {
		this.name = name;
	}

	public void run() {
		for (;;) {
			System.out.println(name + ": hello world");
		}
	}
}

public class Main2 {
	public static void main(String [] args) {
		MyThread t1 = new MyThread("thread1");
		MyThread t2 = new MyThread("thread2");
		t1.start(); t2.start();
	}
}
```

```
thread2: hello world
thread2: hello world
thread2: hello world
thread2: hello world
thread2: hello world
thread2: hello world
thread2: hello world
thread2: hello world
thread2: hello world
thread2: hello world
thread2: hello world
thread2: hello world
thread2: hello world
thread2: hello world
thread2: hello world
thread1: hello world
thread2: hello world
thread1: hello world
thread2: hello world
thread2: hello world
thread1: hello world
thread2: hello world
thread2: hello world
```

See the variation in output: This variation in output is called a “race condition” (often race conditions are bugs in programs)

java.lang.Thread

- public static void yield();
  - Method of java.lang.Thread
  - Thread gives up CPU for other threads ready to run

```
class MyThread extends Thread {
        private String name;

        public MyThread(String name) {
                this.name = name;
        }

        public void run() {
                for (;;) {
                        System.out.println(name + ": hello world");
                        yield();
                }
        }
}

public class Main3 {
        public static void main(String [] args) {
                MyThread t1 = new MyThread("thread1");
                MyThread t2 = new MyThread("thread2");
                t1.start(); t2.start();
        }
}
```

Some Output

```
thread1: hello world
thread2: hello world
thread1: hello world
thread2: hello world
thread1: hello world
thread2: hello world
thread1: hello world
thread2: hello world
thread1: hello world
thread2: hello world
thread1: hello world
thread2: hello world
thread1: hello world
thread2: hello world
thread1: hello world
thread2: hello world
thread1: hello world
thread2: hello world
thread1: hello world
thread2: hello world
thread1: hello world
thread2: hello world
thread1: hello world
thread2: hello world
thread1: hello world
```

Notice the alternation of output

More Thread Members: ***join***

- public final void join();

```
MyThread t1 = new MyThread("thread1");
t1.start();
t1.join();
```

  - Wait until the thread is “not alive”
  - Threads that have completed are “not alive” as are threads that have not yet been started

- public static void sleep (long millis) throws InterruptedException;
  - Makes the currently running thread sleep (block) for a period of time
  - The thread does not lose ownership of any monitors.
  - InterruptedException - if another thread has interrupted the current thread.

Join Example

```
class MyThread extends Thread {
	public void run() {
		for (int i=0; i < 1000; i++) {
			System.out.println("hello world1");
		}
	}
}

public class Main4 {
	public static void main(String [] args) {
		MyThread t1 = new MyThread();
		t1.start();
		try {
			t1.join(); // wait for the thread to terminate
		} catch (InterruptedException e) {
			System.out.println("ERROR: Thread was interrupted");
		}

		System.out.println("Thread is done!");
	}
}
```

Some output

```
…
hello world1
hello world1
hello world1
hello world1
hello world1
hello world1
hello world1
hello world1
hello world1
hello world1
hello world1
hello world1
hello world1
hello world1
hello world1
hello world1
hello world1
hello world1
hello world1
hello world1
hello world1
Thread  is done!
```

Thread State

- public Thread.State getState()
  - Returns the state of this thread. This method is designed for use in monitoring of the system state, not for synchronization control

```
public static enum Thread.State
extends Enum<Thread.State>
```

A thread state. A thread can be in one of the following states:

- NEW
  A thread that has not yet started is in this state.
- RUNNABLE
  A thread executing in the Java virtual machine is in this state.
- BLOCKED
  A thread that is blocked waiting for a monitor lock is in this state.
- WAITING
  A thread that is waiting indefinitely for another thread to perform a particular action is in this state.
- TIMED_WAITING
  A thread that is waiting for another thread to perform an action for up to a specified waiting time is in this state.
- TERMINATED
  A thread that has exited is in this state.

A thread can be in only one state at a given point in time. These states are virtual machine states which do not reflect any operating system thread states.

http://java.sun.com/javase/6/docs/api/java/lang/Thread.State.html

Thread Scheduling in Java

- public final void setPriority(int newPriority);
- public final int getPriority();
- public static final int MAX_PRIORITY
  // on my system: 10; Mac OS X 2/21/05
- public static final int MIN_PRIORITY
  // on my system: 1; Mac OS X 2/21/05
- Scheduling
  - Priority inherited from parent, but can be changed
  - Higher priority threads generally run before lower priority threads
  - For equal priority threads, best to call yield() intermittently to handle JVM's with user-level threading (i.e., no time-slicing)

Sharing Data Across Java Threads

- Consider the situation where a parent thread wants to pass data to a child thread
  - e.g., so that child can change data and parent can have access to the changed data
- How can this be done?
- Can pass an object instance to the child thread constructor, and retain that object instance in a data member

```
class SharedData {
        public int a = 0;
        public String s = null;

        public SharedData() {
                a = 10;
                s = "Test";
        }
}

class MyThread extends Thread {
        private SharedData m_data = null;

        public MyThread(SharedData data) {
                m_data = data;
        }

        public void run() {
                for (;;) {
                        m_data.a++;
                }
        }
}
```

```
public class Main5 {
        public static void main(String [] args) {
                SharedData data = new SharedData();
                MyThread t1 = new MyThread(data);
                t1.start();

                for (;;) {
                        data.a--;
                }
        }
}
```

If we have multiple threads accessing this shared data, how do we synchronize access to ensure it remains in a consistent state?

Basic Tools for Synchronization in Java

- Synchronized methods
- Synchronized objects
- Methods
  - wait
  - notify
  - notifyAll
- Also should talk about condition variables in Java

Synchronized Methods: Monitors

- synchronized keyword used with a method
  - E.g.,

```
public synchronized void SetValue() {
    // Update instance data structure.
    // When the thread executes here, it exclusively has the monitor lock
}
```

  - Provides *instance-based* mutual exclusion
    - A lock is implicitly provided-- allows at most one thread to be executing the method at one time
  - Used on a per method basis; not all methods in a class have to have this
    - But, you'll need to design it right!!

Difference: Synchronized vs. Non-synchronized

- Class with synchronized methods
  - How many threads can access the methods of an object?
- Class with no synchronized methods
  - How many threads can access the methods of an object?

Example

- Construct a queue (FIFO) data structure that can be used by two threads to access the queue data in a synchronized manner
  - Producer thread: Adds data into queue
  - Consumer thread: Removes data from queu
- For one instance of the queue, only one thread should be able to modify the queue, i.e., we should have mutual exclusion on methods of one instance of the queue

```
// Generic synchronized queue. This can be instantiated with any Object type.
// Only 1 thread can use Add or Remove at a time
class SynchQueue<DataType> {
    public LinkedList<DataType> l;

    SynchQueue () {
        l = new LinkedList<DataType>();
    }

    public synchronized void Add(DataType elem) { // add an element to queue
        l.addLast(elem);
    }

    public synchronized DataType Remove() { // remove an element from queue
        if (l.size() > 0) {
            return l.removeFirst();
        } else {
            return null;
        }
    }
}
```


http://www.cprince.com/courses/cs5631/lectures/JavaThreads/SynchMainGeneric.java

```
class Producer extends Thread {
    SynchQueue<Integer> q;
    int curr;

    Producer (SynchQueue<Integer> q) {
        this.q = q;
        curr = 1;
    }

    public void run() {
        for (;;) {
            Integer i = new Integer(curr);
            q.Add(i);
            curr++;
        }
    }
}
```


http://www.cprince.com/courses/cs5631/lectures/JavaThreads/SynchMainGeneric.java

```
class Consumer extends Thread {
    SynchQueue<Integer> q;

    Consumer (SynchQueue<Integer> q) {
        this.q = q;
    }

    public void run() {
        for (;;) {
            Integer i = q.Remove();
            if (i != null) {
                System.out.print(i + " ");
            }
        }
    }
}
```

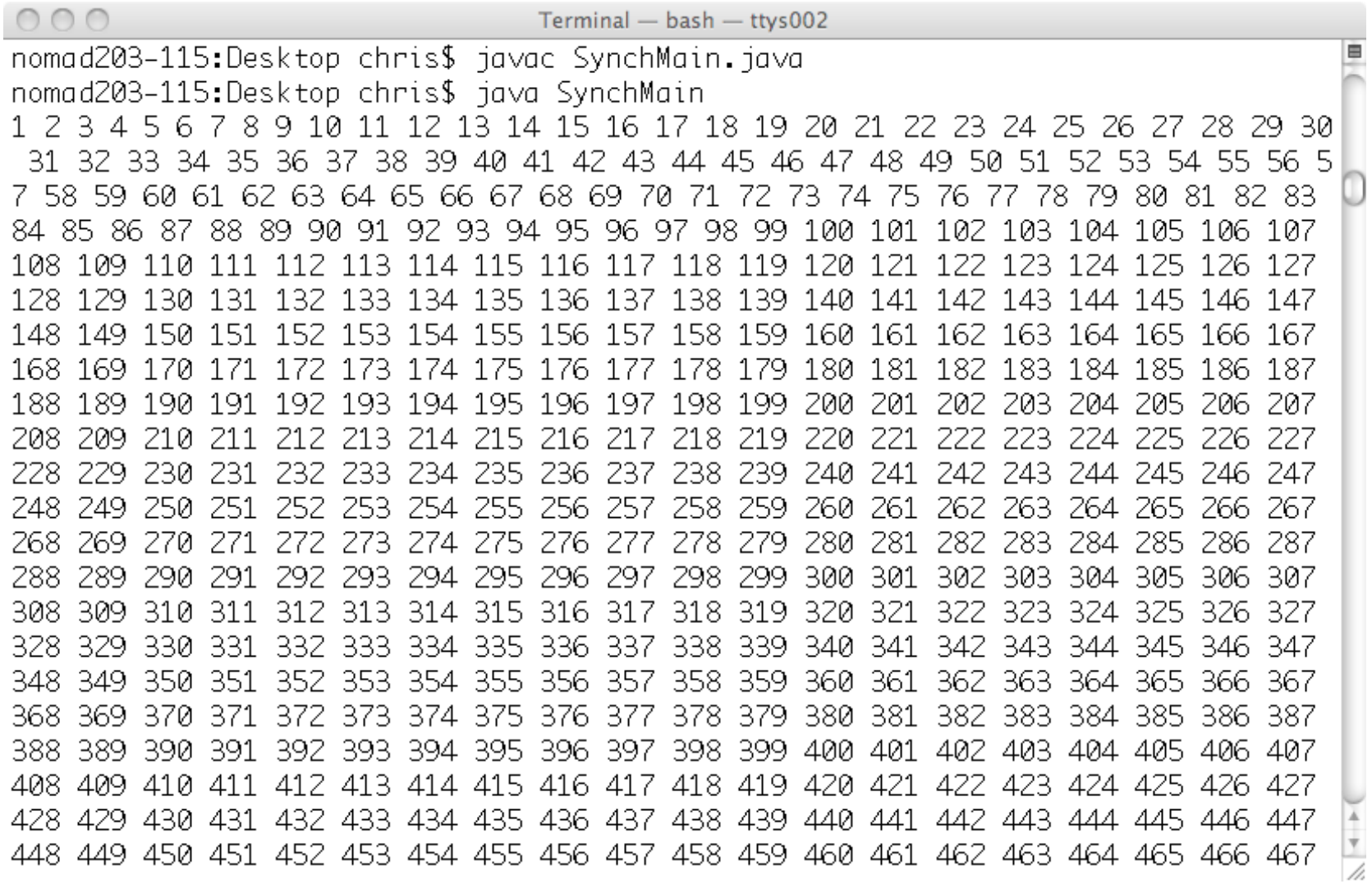

http://www.cprince.com/courses/cs5631/lectures/JavaThreads/SynchMainGeneric.java

```
import java.util.LinkedList; // not synchronized

public class SynchMainGeneric {
    public static void main(String args[]) {
        SynchQueue<Integer> q = new SynchQueue<Integer>();
        Producer p = new Producer(q);
        Consumer c = new Consumer(q);
        p.start();
        c.start();
    }
}
```


http://www.cprince.com/courses/cs5631/lectures/JavaThreads/SynchMainGeneric.java

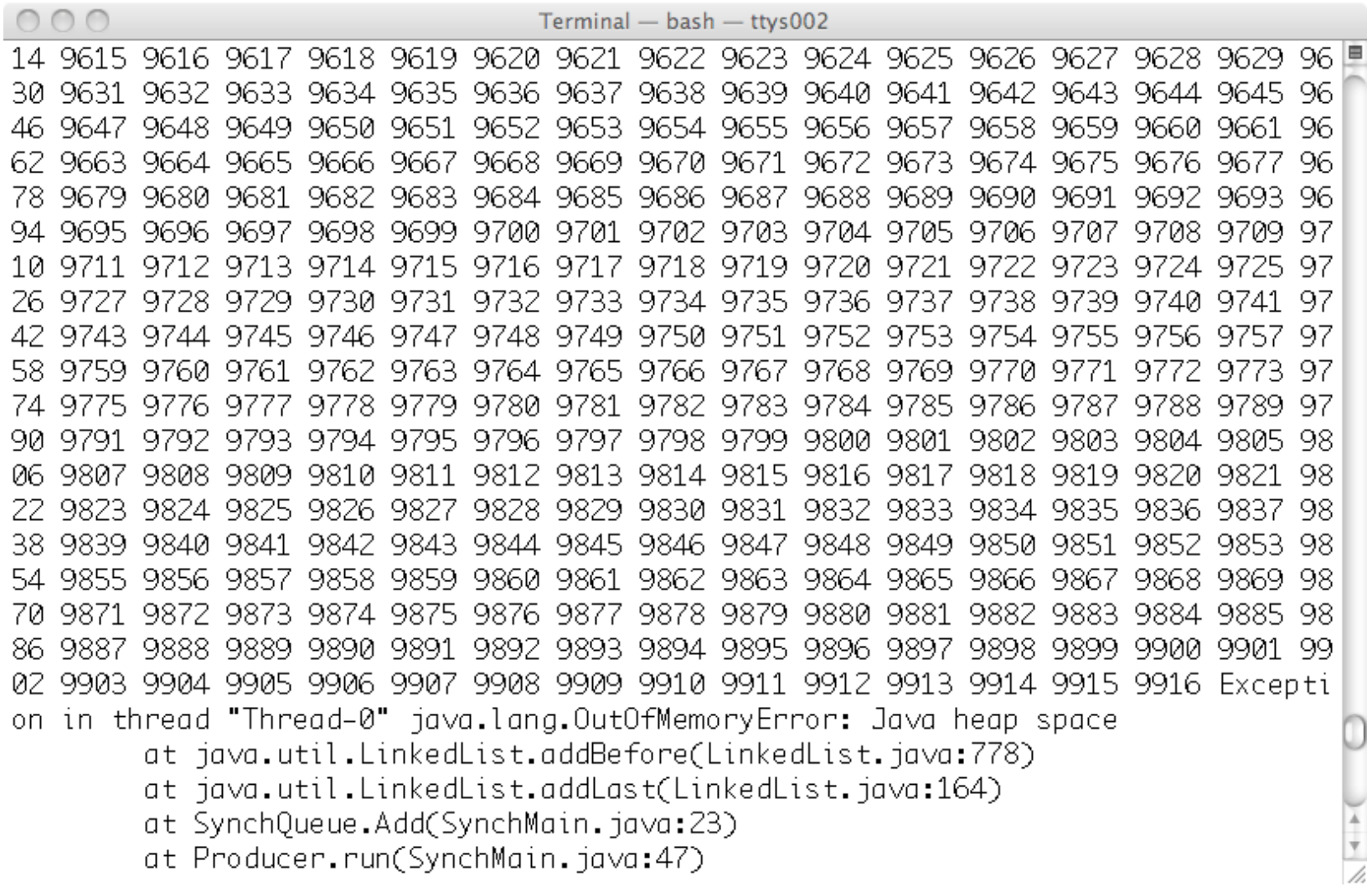
Let's run this and see what happens



Terminal — bash — ttys002

```
26 5327 5328 5329 5330 5331 5332 5333 5334 5335 5336 5337 5338 5339 5340 5341 53
42 5343 5344 5345 5346 5347 5348 5349 5350 5351 5352 5353 5354 5355 5356 5357 53
58 5359 5360 5361 5362 5363 5364 5365 5366 5367 5368 5369 5370 5371 5372 5373 53
74 5375 5376 5377 5378 5379 5380 5381 5382 5383 5384 5385 5386 5387 5388 5389 53
90 5391 5392 5393 5394 5395 5396 5397 5398 5399 5400 5401 5402 5403 5404 5405 54
06 5407 5408 5409 5410 5411 5412 5413 5414 5415 5416 5417 5418 5419 5420 5421 54
22 5423 5424 5425 5426 5427 5428 5429 5430 5431 5432 5433 5434 5435 5436 5437 54
38 5439 5440 5441 5442 5443 5444 5445 5446 5447 5448 5449 5450 5451 5452 5453 54
54 5455 5456 5457 5458 5459 5460 5461 5462 5463 5464 5465 5466 5467 5468 5469 54
70 5471 5472 5473 5474 5475 5476 5477 5478 5479 5480 5481 5482 5483 5484 5485 54
86 5487 5488 5489 5490 5491 5492 5493 5494 5495 5496 5497 5498 5499 5500 5501 55
02 5503 5504 5505 5506 5507 5508 5509 5510 5511 5512 5513 5514 5515 5516 5517 55
18 5519 5520 5521 5522 5523 5524 5525 5526 5527 5528 5529 5530 5531 5532 5533 55
34 5535 5536 5537 5538 5539 5540 5541 5542 5543 5544 5545 5546 5547 5548 5549 55
50 5551 5552 5553 5554 5555 5556 5557 5558 5559 5560 5561 5562 5563 5564 5565 55
66 5567 5568 5569 5570 5571 5572 5573 5574 5575 5576 5577 5578 5579 5580 5581 55
82 5583 5584 5585 5586 5587 5588 5589 5590 5591 5592 5593 5594 5595 5596 5597 55
98 5599 5600 5601 5602 5603 5604 5605 5606 5607 5608 5609 5610 5611 5612 5613 56
14 5615 5616 5617 5618 5619 5620 5621 5622 5623 5624 5625 5626 5627 5628 5629 56
30 5631 5632 5633 5634 5635 5636 5637 5638 5639 5640 5641 5642 5643 5644 5645 56
46 5647 5648 5649 5650 5651 5652 5653 5654 5655 5656 5657 5658 5659 5660 5661 56
62 5663 5664 5665 5666 5667 5668 5669 5670 5671 5672 5673 5674 5675 5676 5677 56
78 5679 5680 5681 5682 5683 5684 5685 5686 5687 5688 5689 5690 5691 5692 5693 56
94 5695 5696 5697 5698 5699 5700 5701 5702 5703 5704 5705 5706 5707 5708 5709 57
```

Ooops! What happened?



- This implementation has a problem! The Consumer prints, which slows it down a LOT, and thus the producer is faster, and thus the producer fills up the queue, and causes heap space to run out!!
- This is a kind of ***race condition***
  - The results depend on the speed of execution of the two processes
- Would like to alter this program to limit the maximum number of items that are stored in the queue.
- ***Goal***: have the producer ***block*** (wait) when the queue reaches some fixed size limit

Also

- Better to have the Remove block (wait) when the queue is empty
- I.e., presently we are doing a “busy wait” (also called polling)
- We are repeatedly checking the queue to see if it has data, and using up too much CPU time doing this

wait method (see also java.lang.Object)

- Does a blocking (not busy) wait
- Relative to an Object
  - E.g., Used within a synchronized method
- Releases lock on Object and waits until a condition is true
  - Blocks calling process until notify() or notifyAll() is called on same object instance (or exception occurs)
- Typically used within a loop to re-check a condition
- wait(long millis); // bounded wait

notify and notifyAll methods (see also java.lang.Object)

- Stop a process from waiting– wakes it up
- Relative to an Object
  - E.g., Used within a synchronized method
- Wakes up a blocked thread (notify) or all blocked threads (notifyAll)
  - One woken thread reacquires lock; The awakened thread will not be able to proceed until the current thread relinquishes the lock on this object.
- For notify, if more than one thread available to be woken, then one is picked

Typical use of *wait* within a synchronized method

```
while (condition not true) {
    try {
        wait(); // this.wait();
    } catch {
        System.out.println(“Interrupted!”);
    }
}
// After loop, condition now true & thread
// has monitor lock for this object instance
```

Example

- Extend the example from before:
  - a queue (FIFO) data structure that can be used by two threads to access the queue data in a synchronized manner
- This time, use wait & notify to block the Producer thread if the queue is full, and block Consumer thread if the queue is empty

Re-checking Monitor Conditions

- wait/notify
  - After receiving a notify, a process waiting on a condition may not be next to gain access to monitor (to the data)
    - E.g., occurs if notifyAll used
  - Process may need to re-check the conditions upon which it was waiting
- An "awakened thread will compete in the usual manner with any other threads that might be actively competing to synchronize on this object; for example, the awakened thread enjoys no reliable privilege or disadvantage in being the next thread to lock this object." (http://java.sun.com/j2se/1.5.0/docs/api/)

InterruptedException

- Wait can be woken by the exception, I.e., for reasons other than notify
- Sometimes this can be handled as part of the process of re-checking conditions
- There is another way to handle it too

Exception in Wait

```
// In a synchronized method

// check your condition, e.g., with a semaphore
// operation, test "value" member variable

if /* or while */ (/* condition */) {
    boolean interrupted;
    do {
        interrupted = false;
        try {
            wait();
        } catch (InterruptedException e) {
            interrupted = true;
        }
     } while (interrupted);
}
```

Only allows release from wait caused by notify or notifyAll

- ```
  class Q {
  int n;
  boolean valueSet = false;
  synchronized int get() {
  if(!valueSet)
  try {
  wait();
  } catch(InterruptedException e) {
  System.out.println("InterruptedException caught");
  }
  System.out.println("Got: " + n);
  valueSet = false;
  notify();
  return n;
  }
  synchronized void put(int n) {
  if(valueSet)
  try {
  wait();
  } catch(InterruptedException e) {
  System.out.println("InterruptedException caught");
  }
  this.n = n;
  valueSet = true;
  System.out.println("Put: " + n);
  notify();
  }
  }
  ```

- ```
  class Producer implements Runnable {
  Q q;
  Producer(Q q) {
  this.q = q;
  new Thread(this, "Producer").start();
  }
  public void run() {
  int i = 0;
  while(true) {
  q.put(i++);
  }
  }
  }
  ```

- ```
  class Consumer implements Runnable {
  Q q;
  Consumer(Q q) {
  this.q = q;
  new Thread(this, "Consumer").start();
  }
  public void run() {
  while(true) {
  q.get();
  }
  }
  }
  ```

- ```
  class PCFixed {
  public static void main(String args[]) {
  Q q = new Q();
  new Producer(q);
  new Consumer(q);
  System.out.println("Press Control-C to stop.");
  }
  }
  ```

Synchronized Blocks

- Synchronized methods
  - Implicitly lock is on *this* object
- Synchronized *blocks*
  - lock on an arbitrary, specified object
  - similar to condition variables in monitors
  - but need to have a synchronized block around an object before wait/notify used
  - use wait/notify on the object itself

Syntax

```
synchronized (object) {
   // object.wait()
   // object.notify()
   // object.notifyAll()
}
```

- For example, this allows you to synchronize just a few lines of code, or to synchronize on the basis of an arbitrary object

Another Example

- Suppose in a Global File Table, suppose that per open file you keep an

Object Lock;

- you can then use a synchronized block to make sure that some operations only get done in a mutually exclusive manner on the file

```
synchronized (file[i].Lock) {
        // if we get to here we're the only one
        // accessing file i
}
```

Conditions

- Java interface:
  - http://download.oracle.com/javase/6/docs/api/java/util/concurrent/locks/Condition.html
- Let's you have multiple independent wait events for a monitor
- So, have one monitor lock, but can wait within the monitor for more than one reason
- Use await/signal (not wait/notify)
- Also: Must have explicit lock (don't use *synchronized* keyword)
- Lock monitor as ***very first*** thing you do
- Unlock monitor as ***very last*** thing you do

END!

Lab 5: Agent Simulation

- Could use synchronized blocks to accomplish synchronization on the environment cells

```
synchronized (cell) {
    // check to see if agent can consume food
    // or socialize depending on what the goal
    // of the agent is
}
```

Example

- Implement Semaphore class with Java synchronization
  - Provide constructor, and P (wait) and V (signal) methods
  - Use synchronized methods
    - and Java wait/notify
- Note
  - Java implements Semaphores—
    - http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/Semaphore.html

java.lang.Runnable Interface

- The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. The class must define a method of no arguments called *run*.
- Known Implementing Classes:
  - AsyncBoxView.ChildState, FutureTask, RenderableImageProducer, Thread, TimerTask
- From http://java.sun.com/j2se/1.5.0/docs/api/
- Runnable can be used to create threads
  - See p.136-137 of text

END!

Example

- Two producer threads (A & B), and one consumer thread
- Consumer needs one type of item from thread A and one type of item from thread B before it can proceed
- Use a loop and a wait and recheck conditions in the consumer

```
// only 1 thread can use Add or Remove at a time
class AB {
       public boolean aReady = false;
       public boolean bReady = false;

       AB () {
       }

       public synchronized void PutA() {
            aReady = true;
            notify();
       }

       public synchronized void PutB() {
            bReady = true;
            notify();
       }

       public synchronized void GetAB() {
         while ((! aReady) || (! bReady)) {
             try {
                 wait();
             } catch (InterruptedException e) {
                 System.out.println(
                  "ERROR: Interrupted Exception!\n");
             }
         }

            // It must be the case that
            // aReady == true and bReady == true
            // AND we have exclusive access
            // to this object instance,
            // so, go ahead and change
            // the ready state back to false.
            aReady = false;
            bReady = false;
       }
}
```

```
class ProducerA extends Thread {
       AB ab;
       ProducerA (AB ab) {
          this.ab = ab;
       }

       public void run() {
            for (;;) {
              ab.PutA();
            }
       }
}

class ProducerB extends Thread {
       AB ab;
       ProducerB (AB ab) {
          this.ab = ab;
       }

       public void run() {
            for (;;) {
              ab.PutB();
            }
       }
}

class Consumer extends Thread {
       AB ab;

       Consumer (AB ab) {
            this.ab = ab;
       }

       public void run() {
            for (;;) {
             ab.GetAB();
            }
       }
}
```

```
public class Recheck {
        public static void main(String args[]) {
                AB ab = new AB();
           ProducerA a = new ProducerA(ab);
           ProducerB b = new ProducerB(ab);
           Consumer c = new Consumer(ab);
           a.start();
           b.start();
           c.start();
        }
}
```

Blocking Remove

```
// only 1 thread can use Add or Remove at a time
class SynchQueue {
        public LinkedList<Integer>  l;

        SynchQueue () {
                l = new LinkedList<Integer>();
        }

        public synchronized void Add(Integer elem) {
                l.addLast(elem);
                notify();
        }

        public synchronized Integer Remove() {
                while (l.size() == 0) {
                        try {
                                wait();
                        } catch (InterruptedException e) {
                                System.out.println("ERROR: Thread interrupted!");
                        }
                }

                return l.removeFirst();
        }
}
```

```
// Generic synchronized queue. This can be instantiated with any Object type.
// Only 1 thread can use Add or Remove at a time.
class SynchQueue<DataType> {
    public LinkedList<DataType> l;

    SynchQueue () {
        l = new LinkedList<DataType>();
    }

    public synchronized void Add(DataType elem) { // add an element to queue
        l.addLast(elem);
        notify();
    }

    public synchronized DataType Remove() { // remove an element from queue
        while (l.size() == 0) {
            try {
                wait();
            } catch (InterruptedException e) {
                System.out.println("Interrupted!");
            }
        }

        return l.removeFirst();
    }
}
```