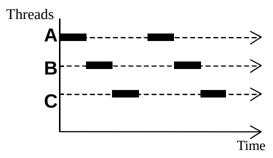
Concurrent Programming in Java

Abhik Roychoudhury and Ju Lei
CS 3211
National University of Singapore

Java Threads

Multithreaded execution is an essential feature of the Java platform. Every application has at least one thread — or several, if you count "system" threads that do things like memory management and signal handling.

From the application programmer's point of view, you start with just one thread, called the *main thread*. This thread has the ability to create additional threads.



Parallelism is not necessary, but possible.

The threads can time-share on a processor.

Managing Thread objects

Each application thread is an instance of the class Thread.

In the programming style I describe here:

the application directly controls thread creation and management by instantiating the Thread class whenever necessary.

An application that creates an instance of Thread must provide the code that will run in that thread. There are two ways to do this:

- use the Runnable interface
- create a subclass of the Thread class.

(see the next two slides for description of these two approaches).

Concurrent Thread Execution

- Each thread has a priority
  - Initial priority: inherited from its parent thread
  - **setPriority**(int newPriority)
- When multiple threads running on the same processor
  - Ready thread with highest priority get executed
  - "Randomly" select among threads with same priority

Starting a Thread (1)

```
public class HelloRunnable implements Runnable {
         public void run() {
             System.out.println("Hello from a thread!");
         }

         public static void main(String args[]) {
             (new Thread(new HelloRunnable())).start();
         }
}
```

Runnable interface contains a single method run()
--- containing code to be executed.

Re-define the run() method and pass it to the thread constructor.

Starting a Thread (2)

```
public class HelloThread extends Thread {

       public void run() {
           System.out.println("Hello from a thread!");
       }

       public static void main(String args[]) {
                 (new HelloThread()).start();
       }
}
```

Common programming mistakes

```
Thread myThread = new Thread(MyRunnable());
mythread.run();

Calling thread will execute the run() method
• Treated as normal function call
• No new thread is created
• Not desirable in most situations


Thread myThread = new Thread(MyRunnable());
mythread.start();

A new thread is created (run() is executed
in this new thread)
```

Stopping Threads

- Thread normally terminates by returning from its run() method
- Deprecated methods: **stop(), suspend(), destroy()** etc.
  - Unsafe, don't use
  - Use (shared) variables to control thread termination if necessary
- The join method allows one thread to wait for the completion of another

  t.join();

  causes the current thread to pause execution until t's thread terminates.

A sleeping thread

```
public class SleepMessages {
    public static void main(String args[]) throws InterruptedException {
         String importantInfo[] = { "Mares eat oats",
                                    "Does eat oats",
                                    "Little lambs eat ivy",
                                    "A kid will eat ivy too"
          };
          for (int i = 0; i < importantInfo.length; i++) { //Pause for 4
seconds
                          Thread.sleep(4000); //Print a message
                          System.out.println(importantInfo[i]);
          }
    }
}
```

Thread.sleep causes the current thread to suspend execution for a specified period. This is an efficient means of making processor time available to the other threads of an application or other applications that might be running on a computer system

An Example

```
public class ThreadExample {
   public static void main(String[] args){
       System.out.println(Thread.currentThread().getName());
       for(int i=0; i<10; i++){
           new Thread("" + i){
               public void run(){
                    System.out.println("Thread: " + getName() + " running");
               }
           }.start();
        }
    }
}
```

1. What does the above example do?
2. How many threads will be created ?
3. What will be the exact printout?

Thread Communication

- Two common modes of thread communication in parallel / concurrent programming
  - Shared memory
  - Message passing
- In this course, we study the Java shared memory communication style where threads read and write shared objects. This requires synchronization mechanisms to ensure safe and "correct" accesses to the objects.
- Message passing is also available in Java
  - Refer to Textbook, Chapter 10

Two problems in Concurrent Programming

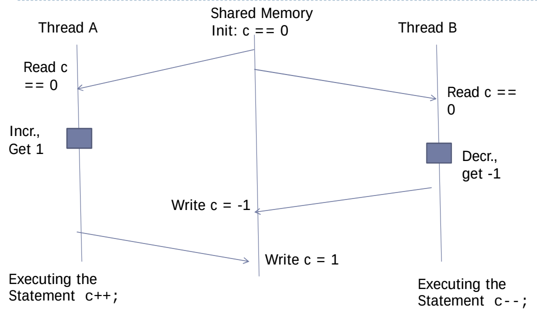
- Race conditions
  - Two or more threads access the shared data simultaneously
  - Solution: lock the data to ensure mutual exclusion of critical sections
- Deadlock
  - Two threads are waiting for each other to release a lock, or more than two processes are waiting for lock in a circular chain.

Interference between threads

```
class Counter {
        private int c = 0;
        public void increment() {
            c++;
        }
        public void decrement() {
            c--;
        }
        public int value() {
            return c;
        }
}

Counter c = new Counter();
public class ThreadA extends Thread{
          public void run() { c.increment(); }
}

public class ThreadB extends Thread{
          public void run() { c.decrement(); }
}
```

Different from Promela: Java statements are not atomic!!!

Thread Interference



Avoiding thread interference – (1)

Java programming provides two basic synchronization idioms: *synchronized methods* and *synchronized statements*

```
public class SynchronizedCounter {
      private int c = 0;
      public synchronized void increment() {
                        c++;
      }
      public synchronized void decrement() {
                        c--;
      }
      public synchronized int value() {
                        return c;
      }
}
```

Why synchronized methods?

- It is not possible for two invocations of synchronized methods on the same object to overlap.
- When a synchronized method exits, it makes the object state visible to all threads accessing the object subsequently via synchronized methods

Minor point: Constructors cannot be synchronized, try it ! Why??

Synchronized statements

```
public void addName(String name) {
        synchronized(this) {
              lastName = name; nameCount++;
        }
        nameList.add(name);
}
```

Synchronized statements refer to an object --- this refers to the object whose method is being executed.

Needed to avoid generating redundant methods – see example above.

Invoking other object's methods from synch. code can be problematic?

Synchronized Method and Statements

- Synchronized methods lock this object
- Synchronized statements can lock any objects (including this)

```
public synchronized void foo() {
 …
}
```

is equivalent to

```
public void foo() {
   synchronized(this){ … }
}
```

Finer-grained concurrency

```
public class not_together {
        private long c1 = 0;
        private long c2 = 0;
        private Object lock1 = new Object();
        private Object lock2 = new Object();

        public void inc1() {
               {
                                 c1++;
              }
        }

       public void inc2() {
             synchronized(lock2) {
                                  c2++;
             }
       }
}
```

Different from using

synchronized(this)

c1 and c2 are independent, never used together.

=>
Updates can be interleaved.

Fine grained locks

```
class FineGrainLock {
       MyMemberClass x, y;
       Object xlock = new Object(),
              ylock = new Object();
       public void foo() {
           synchronized(xlock) {  //access x here
           }
           //do something - but don't use shared resources
           synchronized(ylock) {    //access y here
           }
       }
      public void bar() {
        synchronized(xlock) {
            synchronized(ylock) {
                //access both x and y here
            }
        }
      //do something - but don't use shared resources
     }
}
```

Re-emphasizing Locks

Consider a bank with 10,000 accounts, each with $1000.
Bank's asset = $10 million.

Simulate the bank's activity with two threads.

ATM thread: picks two accounts at random and moves a random amount of money from one account to another.

Audit thread: Periodically wakes up, and adds all the money in all the accounts.

We should always have $10 million in the bank (Invariant)

ATM Class

```
class ATM extends Bank implements Runnable{
     public void run(){
         int fromAcc, toAcc, amount;
         while (true){
             fromAcc = (int) random(numAcc);
             toAcc = (int) random(numAcc);
             amt = 1+ (int)random(savings[fromAcc].balance);
             savings[fromAcc].balance  -= amt;
             savings[toAcc].balance += amt;
         }
     }
}
```

Auditor Class

```
class Auditor extends Bank implements Runnable{
     public void run(){
         int total;
         while (true){
             nap(1000);  total = 0;
             for (int i =0;  i < numAcc;  i++)
                 total += savings[i].balance;
             … // print  the total
         }
}
```

```
Total is  10000000
Total is  10001090
Total is   9994800
```

Printout

Fixing the ATM

```
class ATM{
    …
    synchronized (lock) {
        savings[fromAcc]-=amt; savings[toAcc] += amt;
    }
}

class Auditor{
    …
    synchronized (lock) {
        for (i=0; i < numAcc; i++)  total += savings[i];
    }
}
```

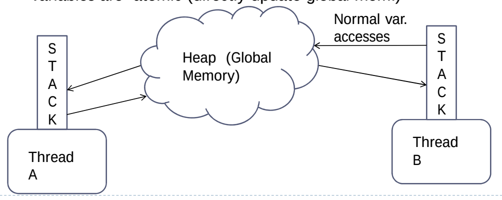
Thread safety without Synchronization

Local Variables – stored in each thread's local stack.
Accessed only by one thread, no synchronization needed.

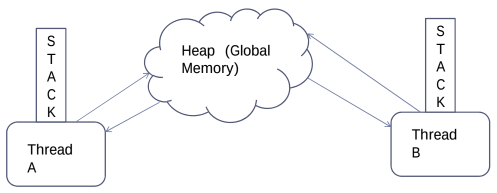
```
public void someMethod(){

        long threadSafeInt = 0;

        threadSafeInt++;
}
```

Volatile Variables

- Consider any simple Java stmt, e.g., x=0
  - Translates to a sequence of bytecodes, not atomically executed.
- One way of ensuring atomic execution in Java –
  - Mark variables as volatile
  - reads/writes (including get-and-set after Java 5) of volatile variables are atomic (directly update global mem.)



Volatile Variables



Conceptual view of volatile variable accesses – atomic reads/writes.
Ensures state visibility, not mutual exclusion.

Marking a variable as volatile tells the compiler to load/store the variable on each use, rather than optimizing away the loads and stores.

An example with volatile variables

```
public class StoppableTask extends Thread {
        private volatile boolean pleaseStop =false;

        public void run() {
            while (!pleaseStop) {
                      // do some stuff...
             }
        }

        public void tellMeToStop() {
             pleaseStop = true;
        }
}
```

A common bug with volatile

```
class Counter {
        private volatile int c = 0;
        public void increment() {
            c++;
        }
        public void decrement() {
            c--;
        }
        public int value() {
            return c;
        }
}
```

Wrong program!!!

A common bug with volatile

```
volatile int c;
...

c ++;
```

```
int temp;

synchronized (c) {

          temp = c;
}

temp++;

synchronized (c) {

     c = temp;
}
```

Volatile Vs. Synchronized

- Summarize on volatile variables
  - Value of volatile variable will **never be cached**
  - Access (read/write/get-and-set) to the variable is atomic
- Pros
  - Light-weight synchronization mechanism
  - A primitive variable may be declared volatile
  - Access to a volatile variable never block (deadlock)
- Cons
  - Correct use of volatile relies on the understanding of Java memory model (e.g., get-update-set of a volatile variable is not atomic).

Deadlock

- Example

```
threadA:                        threadB:
synchronized(lock1) {           synchronized(lock2) {
   synchronized(lock2) {           synchronized(lock1) {
       ….                              ….
   }                               }
}                               }
```

- The Java programming language does not prevent deadlock conditions
  - Programmer has to take care of possible deadlock situation (use conventional techniques/programming patterns for deadlock avoidance)
  - Formal verification (e.g., Promela&Spin)

Transfer of Thread Control

- Sometimes, a thread needs certain conditions (on shared objects) to hold before it can proceed
- Method 1: polling/spinning
  - repeatedly locking and unlocking an object to see whether some internal state has changed
  - Inefficient, possible cause of deadlock
- Method 2: wait/notify
  - a thread can suspend itself using ***wait*** until such time as another thread awakens it using notify

Wait and notify

wait()
Waits for a condition to occur. This is a method of the Object class and must be called from within a synchronized method or block.

notify()
Notifies a thread that is waiting for a condition that the condition has occurred. This a method of the Object class and must be called from within a synchronized method or block.

Every object inherits from the Object class, hence support wait /notify.

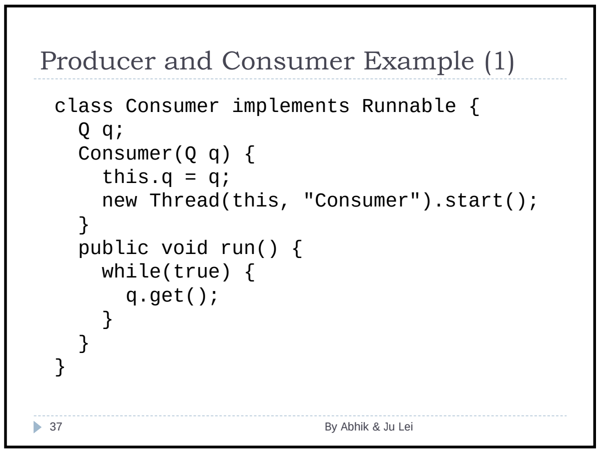
Acquiring and releasing locks
wait() releases lock prior to waiting.
Lock is re-acquired prior to returning from wait().

Producer and Consumer Example (1)

```
class Q {   //queue of size 1
    int n;
    synchronized int get() {
        System.out.println("Got: " + n);
        return n;
    }
    synchronized void put(int n) {
        this.n = n;
        System.out.println("Put: " + n);
    }
}
```

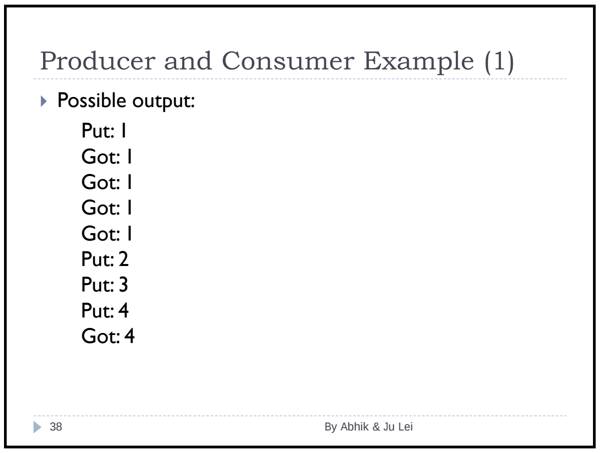
Producer and Consumer Example (1)

```
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++);
    }
  }
}
```



Producer and Consumer Example (1)

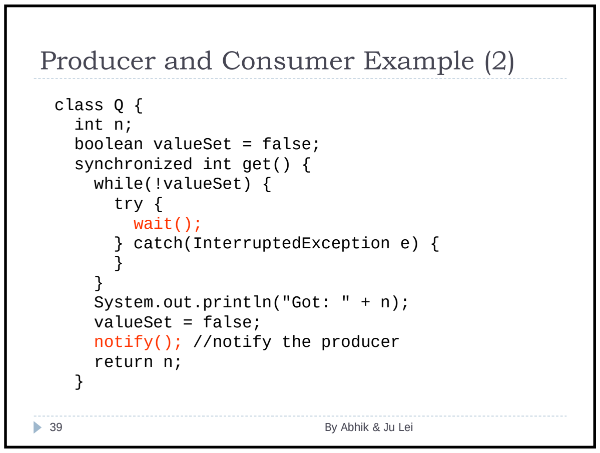
```
class Consumer implements Runnable {
  Q q;
  Consumer(Q q) {
    this.q = q;
    new Thread(this, "Consumer").start();
  }
  public void run() {
    while(true) {
      q.get();
    }
  }
}
```



Producer and Consumer Example (1)

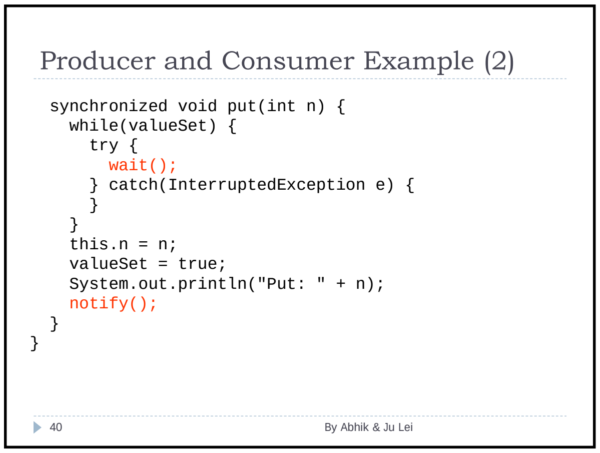
- Possible output:

```
Put: 1
Got: 1
Got: 1
Got: 1
Got: 1
Put: 2
Put: 3
Put: 4
Got: 4
```



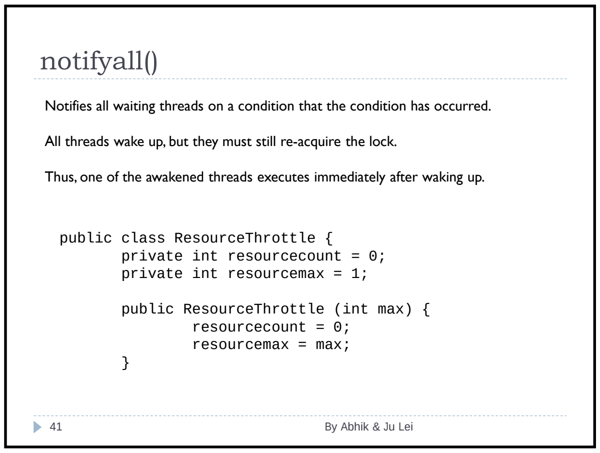
Producer and Consumer Example (2)

```
class Q {
  int n;
  boolean valueSet = false;
  synchronized int get() {
    while(!valueSet) {
      try {
        wait();
      } catch(InterruptedException e) {
      }
    }
    System.out.println("Got: " + n);
    valueSet = false;
    notify(); //notify the producer
    return n;
  }
```



Producer and Consumer Example (2)

```
  synchronized void put(int n) {
    while(valueSet) {
      try {
        wait();
      } catch(InterruptedException e) {
      }
    }
    this.n = n;
    valueSet = true;
    System.out.println("Put: " + n);
    notify();
  }
}
```



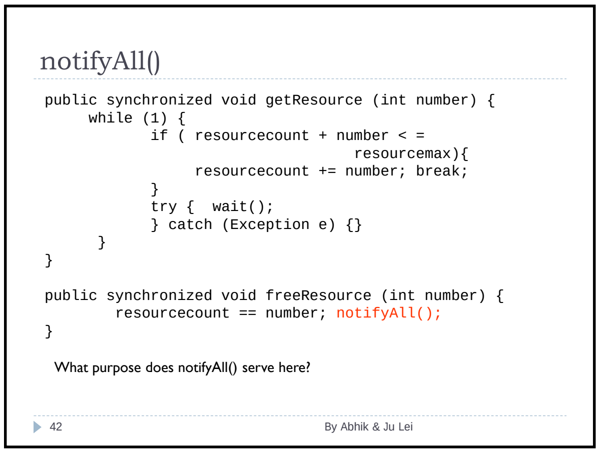
notifyall()

Notifies all waiting threads on a condition that the condition has occurred.

All threads wake up, but they must still re-acquire the lock.

Thus, one of the awakened threads executes immediately after waking up.

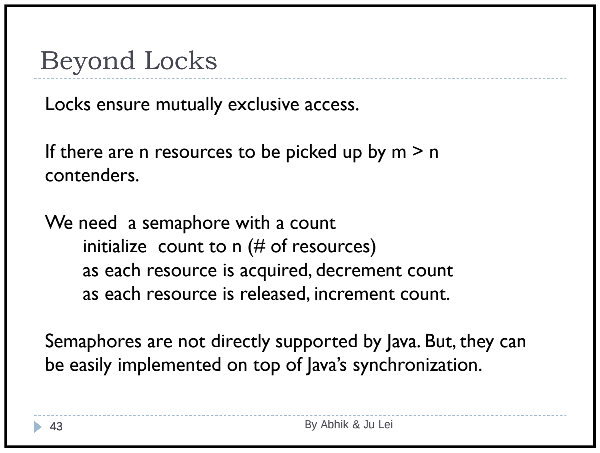
```
public class ResourceThrottle {
        private int resourcecount = 0;
        private int resourcemax = 1;

        public ResourceThrottle (int max) {
                resourcecount = 0;
                resourcemax = max;
        }
```



notifyAll()

```
public synchronized void getResource (int number) {
     while (1) {
            if ( resourcecount + number < =
                                  resourcemax){
                 resourcecount += number; break;
            }
            try {  wait();
            } catch (Exception e) {}
      }
}

public synchronized void freeResource (int number) {
        resourcecount == number; notifyAll();
}
```

What purpose does notifyAll() serve here?



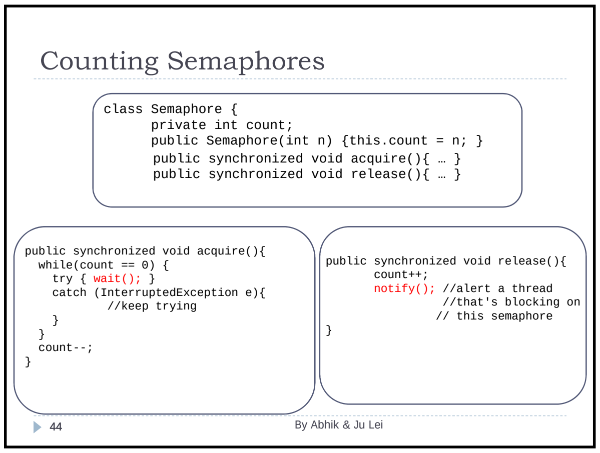
Beyond Locks

Locks ensure mutually exclusive access.

If there are n resources to be picked up by m > n contenders.

We need a semaphore with a count
- initialize count to n (# of resources)
- as each resource is acquired, decrement count
- as each resource is released, increment count.

Semaphores are not directly supported by Java. But, they can be easily implemented on top of Java's synchronization.

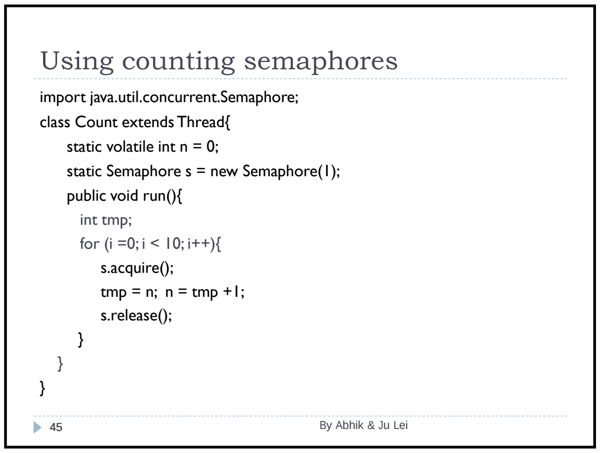


Counting Semaphores

```
class Semaphore {
       private int count;
       public Semaphore(int n) {this.count = n; }
       public synchronized void acquire(){ … }
       public synchronized void release(){ … }
```

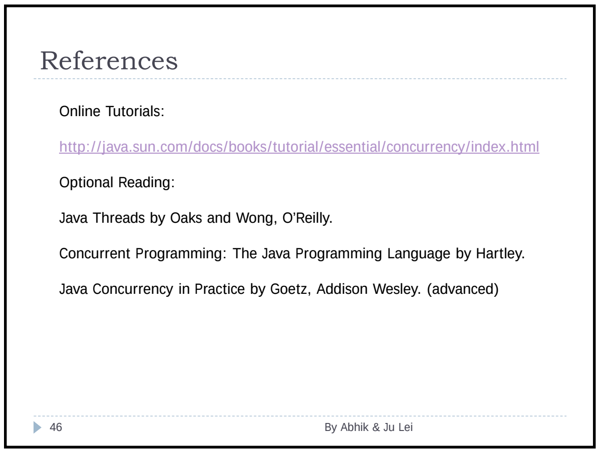
```
public synchronized void acquire(){
  while(count == 0) {
    try { wait(); }
    catch (InterruptedException e){
            //keep trying
    }
  }
  count--;
}
```

```
public synchronized void release(){
       count++;
       notify(); //alert a thread
                 //that's blocking on
                // this semaphore
}
```



Using counting semaphores

```
import java.util.concurrent.Semaphore;
class Count extends Thread{
    static volatile int n = 0;
    static Semaphore s = new Semaphore(1);
    public void run(){
      int tmp;
      for (i =0; i < 10; i++){
         s.acquire();
         tmp = n;  n = tmp +1;
         s.release();
      }
    }
}
```



References

Online Tutorials:

http://java.sun.com/docs/books/tutorial/essential/concurrency/index.html

Optional Reading:

Java Threads by Oaks and Wong, O'Reilly.

Concurrent Programming: The Java Programming Language by Hartley.

Java Concurrency in Practice by Goetz, Addison Wesley. (advanced)