Concurrent Programming: Threads

Contents

- What is Thread?
- Creating Threads
- Sleep, Interrupt, and Join Methods
- Synchronization
- wait, notifyAll, and notify Methods
- A Producer and Consumer Example

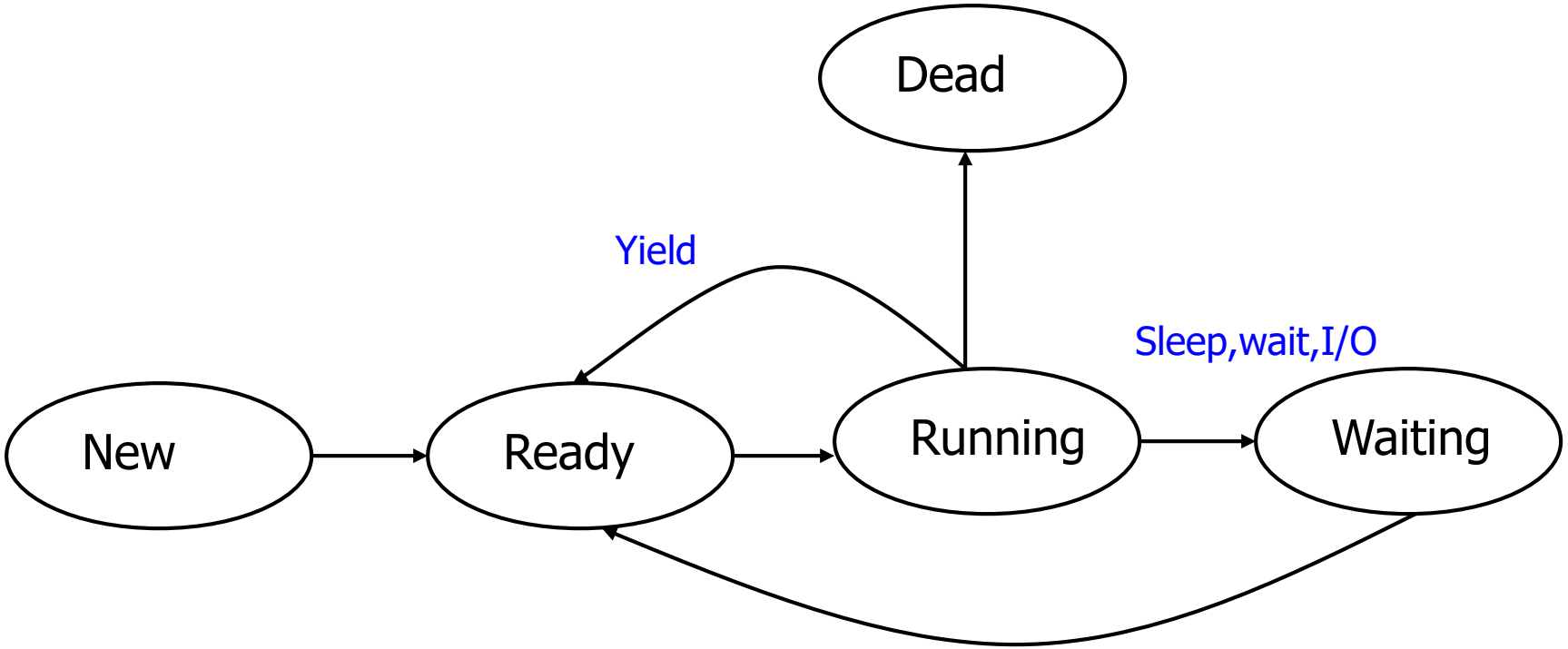
An Overview of Threads

- What is a Thread?
  - A sequence of execution within a process
  - A Lightweight process – requires fewer resources than processes
  - JVM manages and schedules threads
  - Possible States:

    (1) new (2) ready (3) running (4) waiting (5) dead

An Overview of Threads

- Thread life cycle



How to Create Threads

- **Two ways**
  - Using the Thread Class
  - Using the Runnable interface
- **Using the Thread Class**

Extend the Thread class

Implement the run method

Create Thread object

```
public class PingPong extends Thread {
 private String word;
 private int delay;
 public PingPong(String whatToSay, int
      delayTime) {
  word = whatToSay;
   delay = delayTime;
 }
 public void run() {
  try {
    for(;;) {
       System.out.print(word + " ");
       Thread.sleep(delay);
    }
  } catch (InterruptedException e) {
      return;
    }
 }
 public static void main(String[] args) {
    new PingPong("ping", 33).start();
    new PingPong("PONG",100).start();
 }
}
```

Using Runnable

- **Using Runnable Interface**
  - Create a Thread object to pass object of implementation of the Runnable interface into Thread Constructor.

Implement Runnable Interface

Implement the run method

Create Thread object

```
public class RunPingPong implements Runnable {
 private String word;
 private int delay;
 public RunPingPong(String whatToSay, int delayTime) {
   word = whatToSay;
   delay = delayTime;
 }
 public void run() {
   try {
     for(;;) {
       System.out.print(word + " ");
       Thread.sleep(delay);
     }
   } catch (InterruptedException e) {
       return;
     }
 }
 public static void main(String[] args) {
     Runnable ping = new RunPingPong("ping", 33);
     Runnable pong = new RunPingPong("PONG", 100);
     new Thread(ping).start();
     new Thread(pong).start();
 }
}
```

Pausing Execution with Sleep

- Thread.sleep method causes the current thread to suspend execution for a specified period.
- 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.
- Sleep Methods
  - static void sleep(long millis)
  - static void sleep(long millis, int nanos)

```
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]);
        }
    }
}
```

It throws the InterruptedException.

Yield

- **Ending Thread Execution**
  - **The run method returns normally**
  - **public static void sleep(long millis)**
  - **public static void sleep(long millis, int nanos)**
  - **public static void yield()**

```
class Babble extends Thread {
  static boolean doYield;
  static int howOften;
  private String word;
  Babble(String whatToSay) {
   word = whatToSay;
  }
  public void run() {
    for(int i=0; i<howOften; i++) {
     System.out.println(word);
      if (doYield)
        Thread.yield(); // let other threads run
    }
  }
public static void main(String[] args) {
  doYield = new Boolean(args[0]).booleanValue();
  howOften = Integer.parseInt(args[1]);

  // create a thread for each world
  for (int i=2; i < args.length; i++)
    new Babble(args[i]).start();
 }
}
```

Run:

% java Babble false 2 Did DidNot

Result:

Did

Did

DidNot

DidNot

Join

- 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.

- Overloaded Methods
  - void join() : Waits for this thread to die.
  - void join(long millis)
  - void join(long millis, int nanos)

```
class ThreadM extends Thread {
   public void run() {
   try {
     for (int i = 0; i < 10; i++) {
       Thread.sleep(1000);
       System.out.println("ThreadM");       }
   }
   catch (InterruptedException ex) {
     ex.printStackTrace();     }
 }   }
```

```
class ThreadN extends Thread {
   public void run() {
   try {
     for (int i = 0; i < 20; i++) {
       Thread.sleep(2000);
       System.out.println("ThreadN");
     }
   }
   catch(InterruptedException ex) {
     ex.printStackTrace();
   }
  }
}
```

join() method:

Waits for this thread to die.

```
class JoinDemo1 {
  public static void main(String args[]) {
    ThreadM tm = new ThreadM();
    tm.start();
    ThreadN tn = new ThreadN();
    tn.start();
    try {
      tm.join();
      tn.join();
      System.out.println("Both threads have finished");
    }
    catch (Exception e) {       e.printStackTrace();     }
  }
}
```

Interrupts

- An interrupt is an indication to a thread that it should stop what it is doing and do something else.
- A thread sends an interrupt by invoking the “interrupt()” method on the Thread object for the thread to be interrupted.
- Supporting Interruption
  - If the thread is frequently invoking methods that throw InterruptedException, it simply returns from the run method after it catches that exception.
  - Tests for the interrupt and exits the thread if one has been received.
  - In more complex applications, to throw an InterruptedException

```
for (int i = 0; i < importantInfo.length; i++) {
    try {
        Thread.sleep(4000);
    } catch (InterruptedException e) {
        //We've been interrupted: no more
      messages.
        return;
    }
    System.out.println(importantInfo[i]);
}
```

```
  for (int i = 0; i < inputs.length; i++) {
      heavyCrunch(inputs[i]);
      if (Thread.interrupted()) {
          //We've been interrupted: no more
        crunching.
          return;
      }
  }
```

```
if (Thread.interrupted()) {
    throw new InterruptedException();   }
```

Example: SimpleThreads.java

```
public class SimpleThreads {

    //Display a message, preceded by the
      name of the current thread

    static void threadMessage(String
      message) {

        String threadName =
      Thread.currentThread().getName();

        System.out.format("%s: %s%n",
      threadName, message);

    }

    private static class MessageLoop
      implements Runnable {

        public void run() {

            String importantInfo[] = {

             "Mares eat oats", "Does eat oats",

                "Little lambs eat ivy",

                "A kid will eat ivy too"

            };
```

When this thread receives an interrupt, it happens.

```
try {

    for (int i = 0; i < importantInfo.length; i++)
     {

            //Pause for 4 seconds

              Thread.sleep(4000);

            //Print a message

      threadMessage(importantInfo[i]);        }

    } catch (InterruptedException e) {

                threadMessage("I wasn't done!"); }

    } // end of run

    } // end of


public static void main(String args[]) throws
    InterruptedException {

      //Delay, in milliseconds before we
    interrupt MessageLoop

      //thread (default one hour).

      long patience = 1000 * 60 * 60;
```

Example: SimpleThreads.java

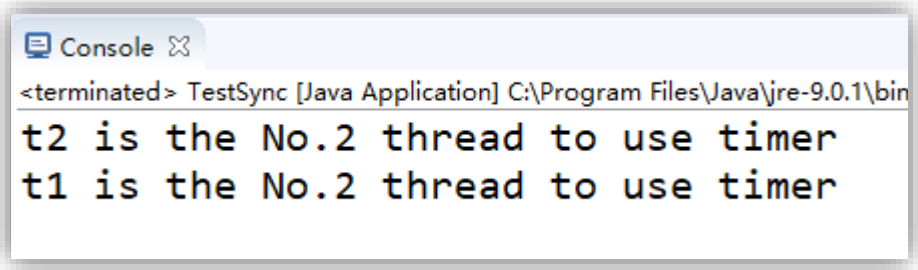
```
//If command line argument present, gives
      patience in seconds.
  if (args.length > 0) {
     try {
        patience = Long.parseLong(args[0]) * 1000;
     } catch (NumberFormatException e) {
              System.err.println("Argument must be
      an integer.");
              System.exit(1);
     }
  }

threadMessage("Starting MessageLoop thread");
 long startTime = System.currentTimeMillis();
       Thread t = new Thread(new MessageLoop());
       t.start();
```

When elapsed time is larger than the patience, it send interrupt to the thread “t”.

```
 threadMessage("Waiting for MessageLoop thread
to finish");
       //loop until MessageLoop thread exits
    while (t.isAlive()) {
         threadMessage("Still waiting...");
            //Wait maximum of 1 second for
MessageLoop thread to finish.
         t.join(1000);
         if (((System.currentTimeMillis() - startTime) >
patience) && t.isAlive()) {
              threadMessage("Tired of waiting!");
              t.interrupt();
         //Shouldn't be long now -- wait indefinitely
              t.join();
           }
       }
       threadMessage("Finally!");
    }
}
```

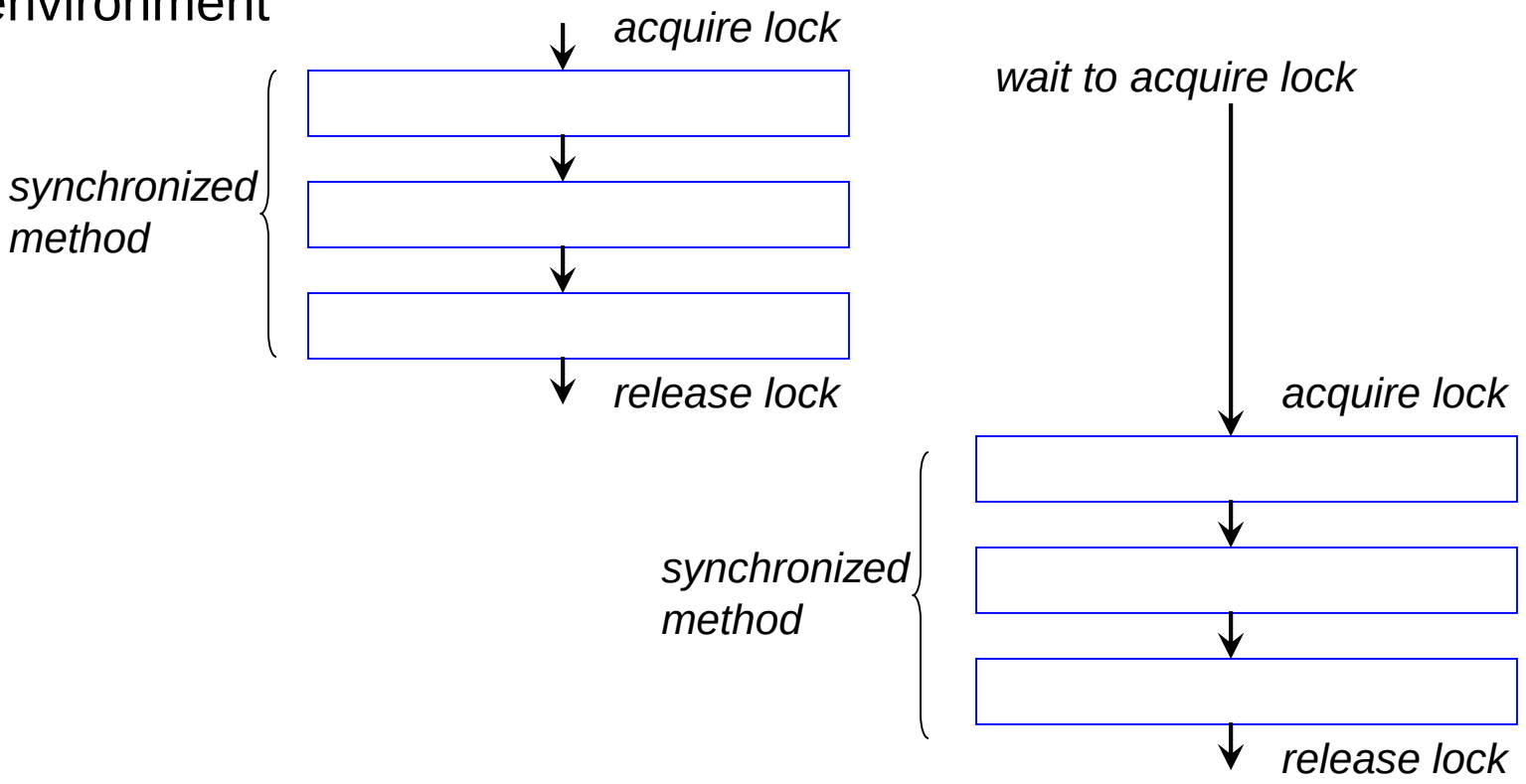
Synchronization

```
public class TestSync implements Runnable {
  Timer timer = new Timer();
  public static void main(String args[]) {
    TestSync test = new TestSync();
    Thread t1 = new Thread(test);
    Thread t2 = new Thread(test);
    t1.setName("t1");
    t2.setName("t2");
    t1.start();
    t2.start();
  }
  public void run() {
    timer.add(Thread.currentThread().getName());
  }
}
class Timer {
  private static int num = 0;
  public void add(String name) {
  num++;
  try {
    Thread.sleep(1);
  } catch (InterruptedException e) {
    e.printStackTrace();
  }
  System.out.println(name + "is the  No." + num + " thread to use timer");
  }
}
```



Synchronized Methods

- Synchronized Methods : protection from interference in a multithreaded environment



If one thread invokes a synchronized method on an object, the lock of that object is first acquired, the method body executed, and then the lock released. Another thread invoking a synchronized method on that same object will block until the lock is released

Synchronized Methods

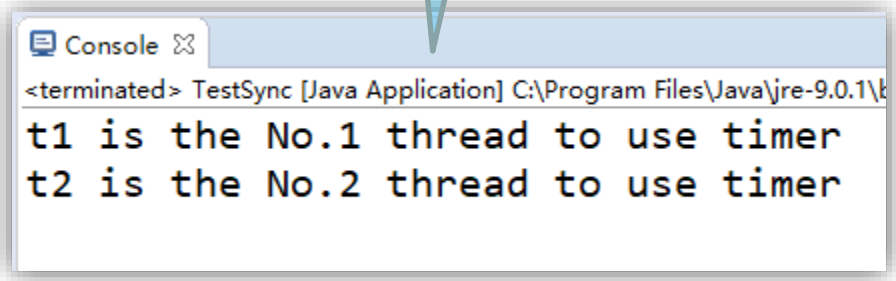
- **Example Code**

```
public class BankAccount {
  private long number;  // account number
  private long balance;  // current balance (in cents)
  public BankAccount(long initialDeposit) {
    balance = initialDeposit;
  }
  synchronized public long getBalance() {
    return balance;
  }
  private final void setBalance(double amount) {
    balance = amount;
  }
  synchronized public void deposit(double amount) {
    double bal = getBalance();
    bal += amount;
    setBalance(bal);
  }
 // … rest of methods
}
```

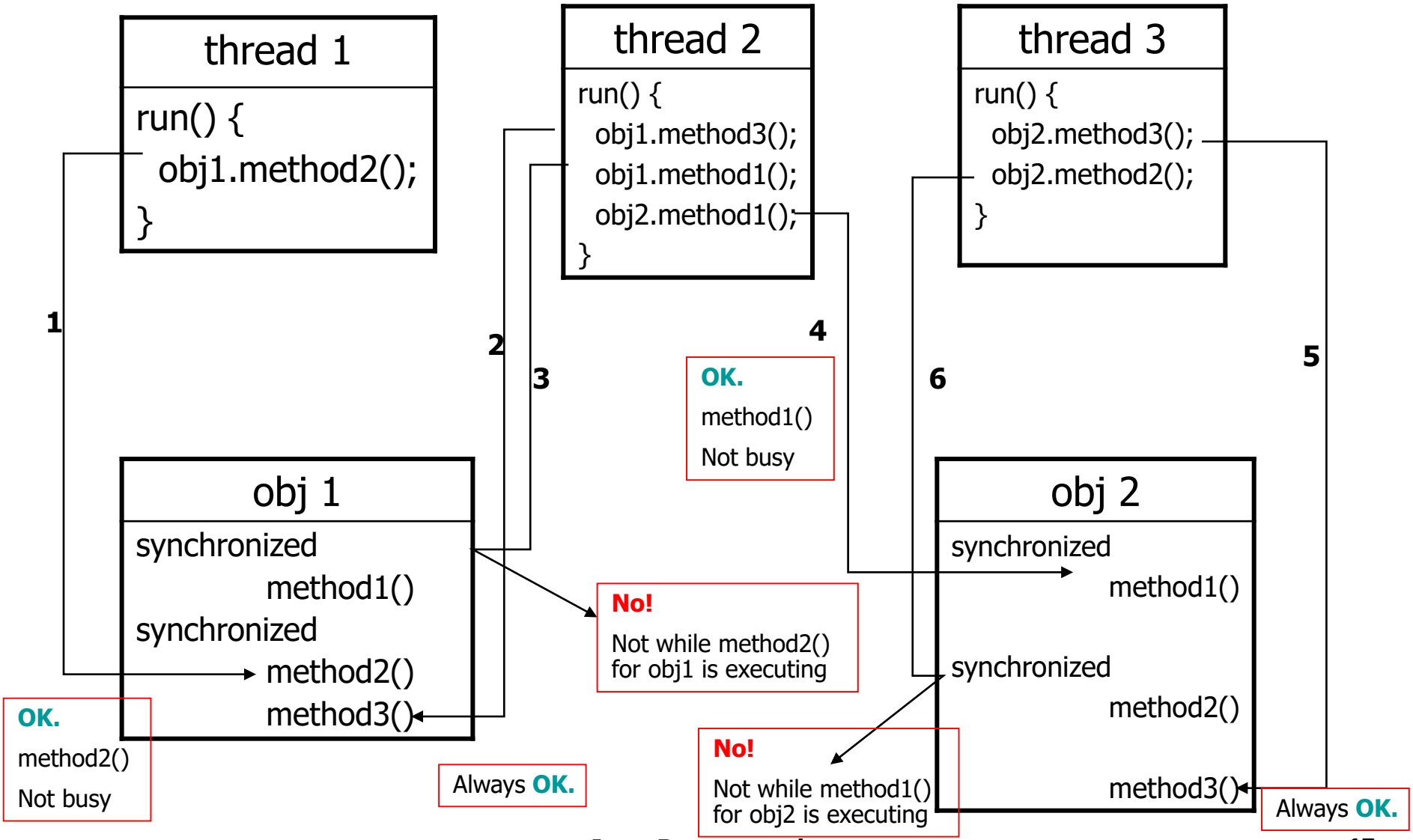
Synchronized Methods

```
public class TestSync implements Runnable {
  Timer timer = new Timer();
  public static void main(String args[]) {
    TestSync test = new TestSync();
    Thread t1 = new Thread(test);
    Thread t2 = new Thread(test);
    t1.setName("t1");
    t2.setName("t2");
    t1.start();
    t2.start();
  }
  public void run() {
    timer.add(Thread.currentThread().getName());
  }
}
class Timer {
  private static int num = 0;
  public synchronized void add(String name) {
  num++;
  try {
    Thread.sleep(1);
  } catch (InterruptedException e) {
    e.printStackTrace();
  }
  System.out.println(name + " is the  No." + num + " thread to use timer");
  }
}
```

When a synchronized method is invoking, **other synchronized methods** in the class cannot be invoked, but **non-synchronized methods** can be invoked.



Locking Objects with Synchronized Methods



Synchronized Statements

- Synchronized Statements
  - The synchronized statement enables to execute synchronized code that acquires the lock of any object, not just the current object, or for durations less than the entire invocation of a method.

```
/** make all elements in the array
    non-negative */
pubic static void abs(int[] values) {
   synchronized (values) {
     for (int i=0; i < values.length; i++)
     {
       if (values[i] < 0)
        values[i] = -values[i];
     }
   }
}
```

The array is not changed during execution by any other code that is similarly synchronized on the values array

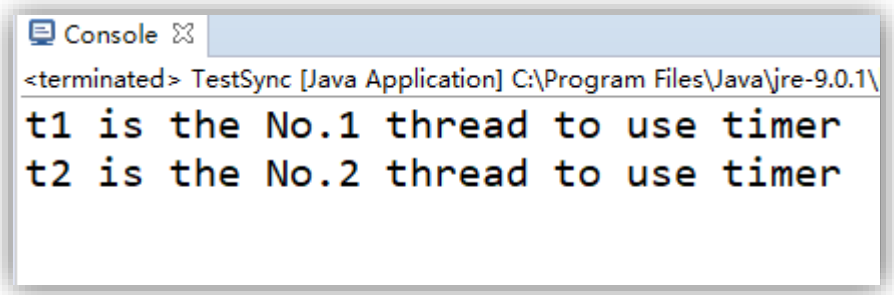
```
synchronized (syncObject) {
   statements
}
```

To execute when the lock is obtained.

An object whose lock is to be acquired

Synchronized Statements

```
public class TestSync implements Runnable {
  Timer timer = new Timer();
  public static void main(String args[]) {
    TestSync test = new TestSync();
    Thread t1 = new Thread(test);
    Thread t2 = new Thread(test);
    t1.setName("t1");
    t2.setName("t2");
    t1.start();
    t2.start();
  }
  public void run() {
    timer.add(Thread.currentThread().getName());
  }
}

class Timer {
  private static int num = 0;
  public void add(String name) {
    synchronized (this) {
      num++;
      try {
        Thread.sleep(1);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
        System.out.println(name + " is the No." + num + " thread to use timer");
    }
  }
}
```



Synchronized Statements

- A necessity of synchronize statement

needs to synchronize changes to lastName and nameCount

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

- In MsLunch, the c1 and c2, that are never used together. All updates of these fields must be synchronized, but there's no reason to prevent an update of c1 from being interleaved with an update of c2 — and doing so reduces concurrency by creating unnecessary blocking.

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

    public void inc1() {
        synchronized(lock1) {
            c1++;
        }
    }

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

Synchronized Statements

- Advantages of the synchronized statement
  - *Can define a synchronized region of code that is smaller than a method.*
  - *Allow to synchronize on objects other than* **this**, *allowing a number of different synchronization designs to be implemented. A finer granularity of locking.*

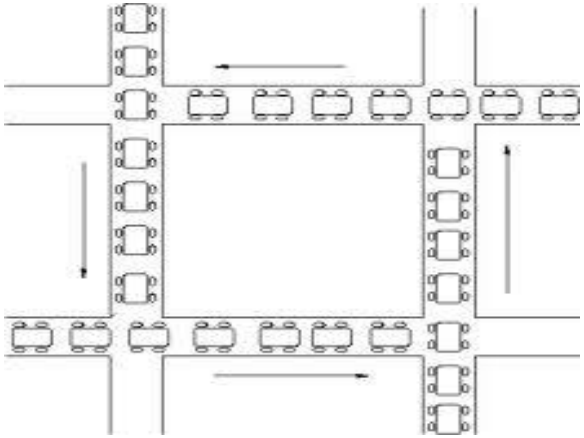
You can define separate objects to be used as locks for each such group using synchronized statements

```
class SeparateGroups {
  private double aVal = 0.0;
  private double bVal = 1.1;
  protected final Object lockA = new Object();
  protected final Object lockB = new Object();
  public double getA() {
    synchronized(lockA) {      return aVal;     }
  }
  public void setA(double val) {
    synchronized (lockA) {      aVal = val;    }
  }
  public double getB() {
    synchronized(lockB) {      return bVal;    }
  }
  public void setB(double val) {
    synchronized (lockB) {      bVal = val;    }
  }
  public void reset() {
    synchronized (lockA) {
      synchronized (lockB) {        aVal = bVal =
       0.0;      }
    }
  }
}
```

Deadlock

- Deadlock describes a situation where two or more threads are blocked forever, waiting for each other.
- Alphonse and Gaston are friends, and great believers in courtesy.
- Bowing Rule: When you bow to a friend, you must remain bowed until your friend has a chance to return the bow.
- Unfortunately, this rule does not account for the possibility that two friends might bow to each other at the same time.

```
public class Deadlock {
    static class Friend {
        private final String name;
        public Friend(String name) {
            this.name = name;
        }
        public String getName() {
            return this.name;
        }

     public synchronized void bow(Friend bower) {
        System.out.format("%s: %s has bowed to me!%n",
                this.name, bower.getName());
         bower.bowBack(this);
     }

     public synchronized void bowBack(Friend bower) {
        System.out.format("%s: %s has bowed back to
        me!%n“,  this.name, bower.getName());
        }
    }


public static void main(String[] args) {
        final Friend alphonse = new Friend("Alphonse");
        final Friend gaston = new Friend("Gaston");
        new Thread(new Runnable() {
            public void run() { alphonse.bow(gaston); }
        }).start();
        new Thread(new Runnable() {
            public void run() { gaston.bow(alphonse); }
        }).start();
    }
}
```



Wait, notifyAll, and notify

- ***The wait() method***
  - The wait() method allows a thread that is executing a synchronized method or statement block on that object to release the lock and wait for a notification from another thread.

- ***The notify() method***
  - The notify() method allows a thread that is executing a synchronized method or statement block to notify another thread that is waiting for a lock on this object.

- Standard Pattern of Wait

```
synchronized void doWhenCondition()
    {
  while(!condition)    wait();
   … Do what must be done when the
     condition is true…
}
```

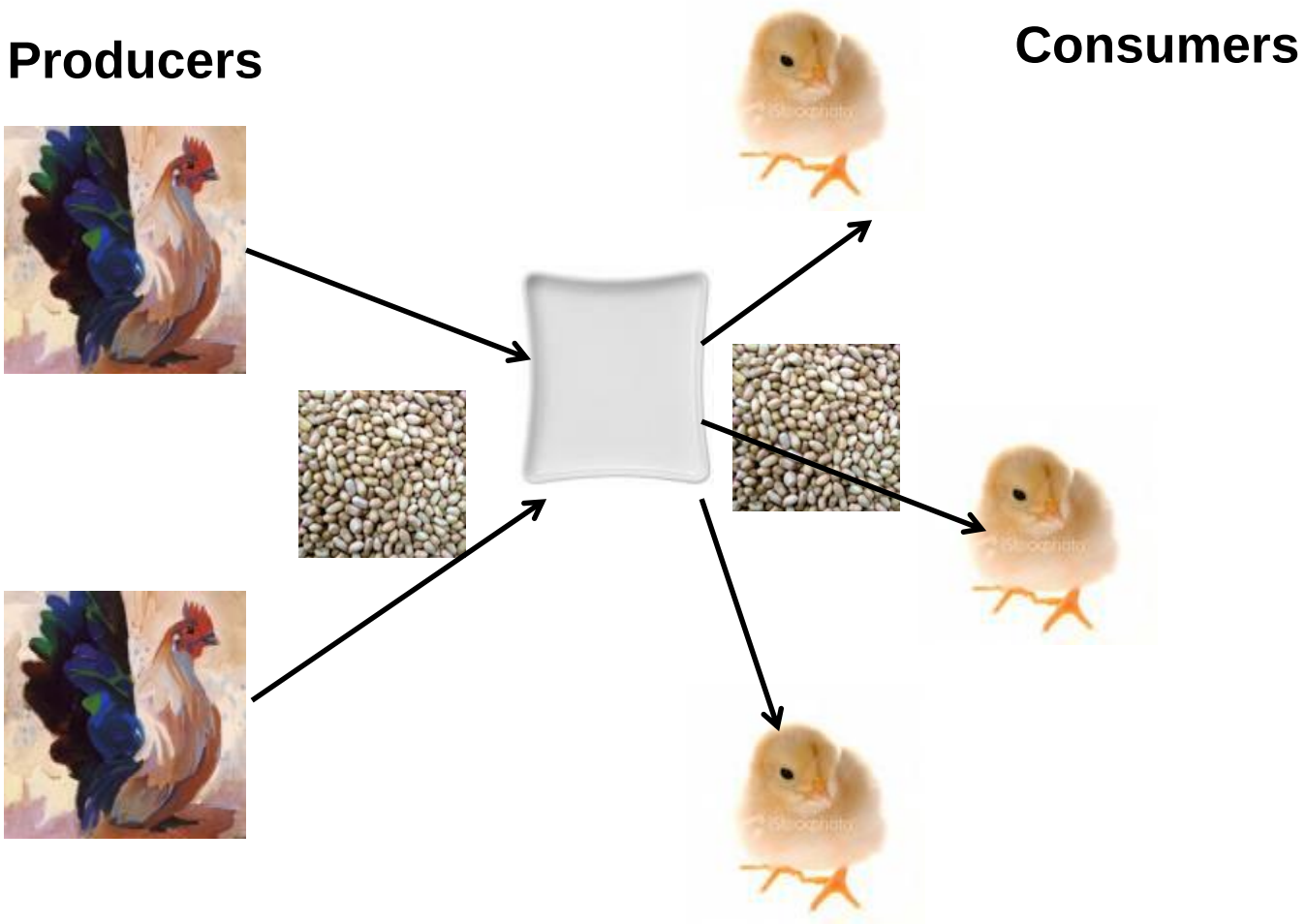
- Notification

```
synchronized void changeCondition() {
  … change some value used in a
     condition test….
  notifyAll();  // or notify()
}
```

Wait, notifyAll, and notify

```
Class PrintQueue {
  private SinglLinkQueue<PrintJob> queue = new
     SingleLinkQueue<PrintJob>();
  public synchronized void add(PrintJob j) {
    queue.add(j);
    notifyAll();   // Tell waiters: print job added
  }
  public synchronized PrintJob remove() throws InterruptedException {
    while (queue.size() == 0)
      wait();   // Wait for a print job
    return queue.remove();
   }
}
```

Producer & Consumer Example



Producer & Consumer Example

```
class Producer extends Thread {
  Queue queue;

  Producer(Queue queue) {
    this.queue = queue;
  }

  public void run() {
    int i = 0;
    while(true) {
      queue.add(i++);
    }
  }
}

class Consumer extends Thread {
  String str;
  Queue queue;

  Consumer(String str, Queue queue) {
    this.str = str;
    this.queue = queue;
  }

  public void run() {
    while(true) {
```

Now, Queue is full, wait until a consumer use a element, so the queue has a space.

```
      System.out.println(str + ": " + queue.remove());
    }
  }
}

class Queue {
  private final static int SIZE = 10;
  int array[] = new int[SIZE];
  int r = 0;
  int w = 0;
  int count = 0;
  synchronized void add(int i) {
    while(count == SIZE) {
      try {
        wait();
      }
      catch(InterruptedException ie) {
        ie.printStackTrace();
        System.exit(0);
      }
    }
    array[w++] = i;
    if (w >= SIZE)
      w = 0;
    ++count;
    notifyAll();
  }
```

Notification to some consumers waiting for element(s) the Producer provides

Producer & Consumer Example

```
synchronized int remove() {
  while(count == 0) {
      try {
        wait();
      }
      catch(InterruptedException ie) {
        ie.printStackTrace();
        System.exit(0);
      }
    }
 int element = array[r++];
    if (r >= SIZE)
      r = 0;
     --count;
     notifyAll();
     return element;
  }
}
```

Now, there is no element to remove, and wait until some element(s) come in to the queue.

```
class ProducerConsumers {
  public static void main(String args[]) {
    Queue queue = new Queue();
    new Producer(queue).start();
    new Consumer("ConsumerA", queue).start();
    new Consumer("ConsumerB", queue).start();
    new Consumer("ConsumerC", queue).start();
  }
}
```