BUILD BETTER SOFTWARE

Understanding Java Threads

Chuck Allison
Utah Valley State College
C/C++ Users Journal
www.freshsources.com

Resources

- *Thinking in Java*, 3rd Edition, Eckel
- *The Java Prog. Lang*, 3rd Ed., Arnold+
- *Java Threads*, 2nd Ed., Oaks & Wong
- *Effective Java*, Bloch
- *Taming Java Threads*, Holub
- *Concurrent Prog. in Java*, 2nd Ed., Lea

The Benefits of Threads

- Effectively use system resources
  - threads run in a single process (no IPC)
  - share same address space (caveat!)
- Concurrent programming
  - independent tasks can run independently
    - from your code's point of view
    - good design
    - enables true parallelism on MP machines
    - responsive user interfaces

The Challenges of Threads

- Tricky to program correctly!
- Must synchronize access to shared data
  - Competition Synchronization
    - Prevent simultaneous access to data
  - Cooperation Synchronization
    - Allow threads to work together
- Must prevent deadlock
  - can easily occur when sharing multiple resources

The Java Threading Model

- Simple, High-level approach
- Provides:
  - a Thread class
    - and a Runnable interface
  - simple locks for synchronizing access to code
  - inter-thread communication

What is a Thread?

- A path of execution in a program
  - shares CPU(s) with other threads
  - keeps sufficient context for swapping
- java.lang.Thread class
  - override run( ) method
  - call thread.start( )
  - terminates when your run( ) returns

A First Example

```
// Illustrates Independent Threads
class MyThread extends Thread
{
   private int count;
   public MyThread(String name, int count)
   {
      super(name); // Optional thread name
      this.count = count;
   }
   public void run()
   {
      for (int i = 0; i < count; ++i)
         System.out.println(getName());
   }
}
```

Main Program (launches 2 threads)

```
public class Independent
{
   public static void main(String[] args)
   {
      Thread t1 = new MyThread("DessertTopping", 8);
      Thread t2 = new MyThread("FloorWax", 4);
      t1.start();
      t2.start();
   }
}
```

Note: There are 3 threads in this example – the main thread launched by the JVM, and the other two launched inside main( ).

Output
(Dependent on platform and environment - YMMV)

```
DessertTopping
DessertTopping
DessertTopping
FloorWax
DessertTopping
FloorWax
DessertTopping
FloorWax
DessertTopping
FloorWax
DessertTopping
DessertTopping
```

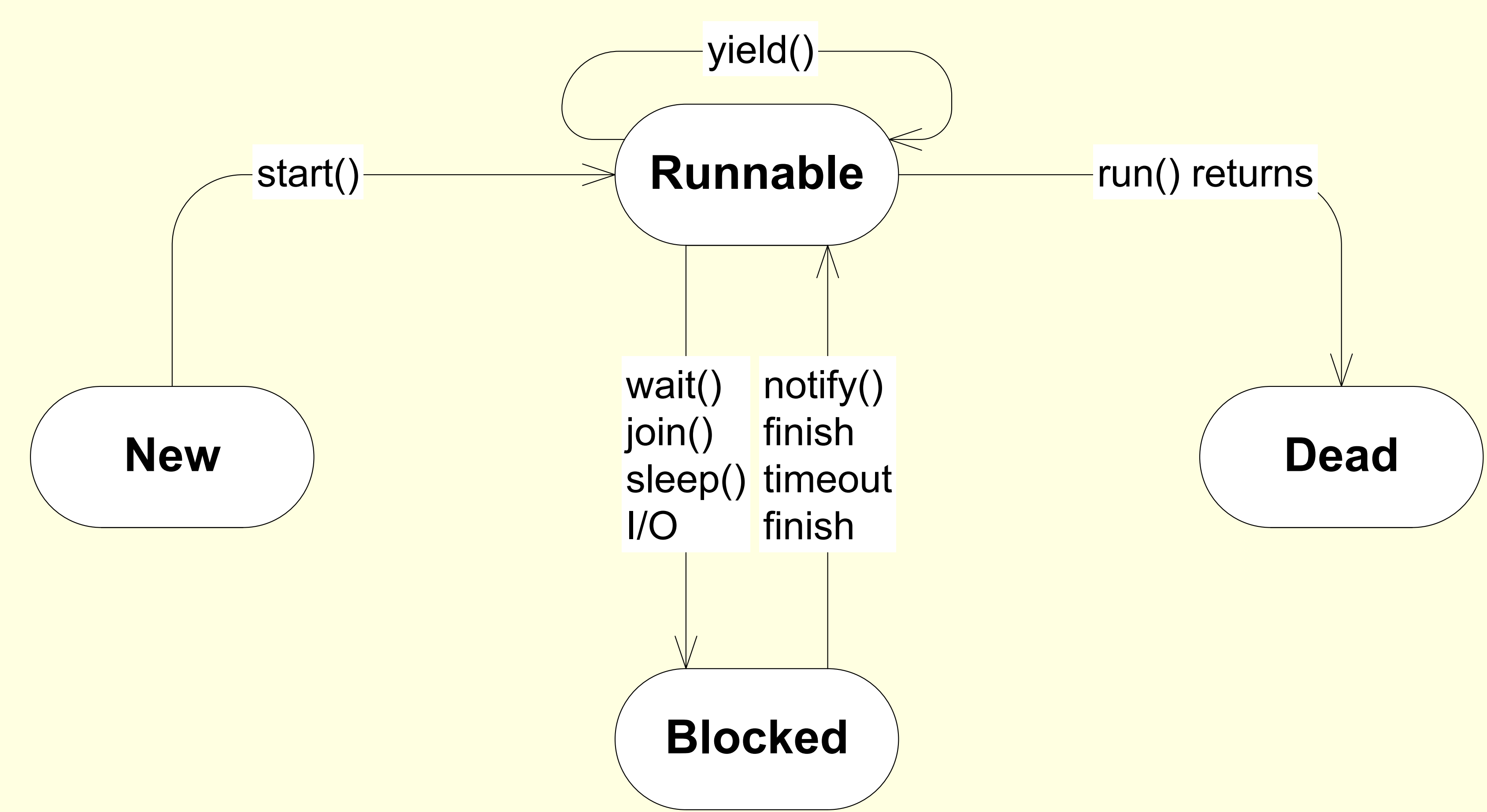
Blocking I/O

- Note that the calls to println( ) run uninterrupted
- I/O is a *blocking* operation
  - The thread waits until it completes
  - Other threads may run, but the I/O will be undisturbed
    - Reason: I/O is coarse-grained native code
- JDK 1.4 java.nio provides non-blocking I/O
  - Buffers, channels, selectors, for more fine-grained control
    - One thread can manage multiple connections

Thread State

- New
  - after creation but before start( )
- Alive
  - Runnable
    - running or swapped out by scheduler
  - Blocked
    - sleeping
    - waiting for I/O, a lock, a notify, or a join
- Dead
  - run( ) has returned, or
  - an uncaught exception occurred

Thread State



Interleaved I/O with print( )

```
class MyThread extends Thread
{
   // <snip>
   public void run()
   {
      for (int i = 0; i < count; ++i)
      {
         display();     // Replaces println()
      }
   }
   void display()
   {
      String s = getName();
      for (int i = 0; i < s.length(); ++i)
         System.out.print(s.charAt(i));
      System.out.println();
   }
}
```

Output (interleaved – oops!)

```
DessertTopping
DFloorWax
FloorWax
FloorWax
FloorWessertTopping
Desax
sertTopping
DessertTopping
DessertTopping
DessertTopping
DessertTopping
DessertTopping
```

Typical java.lang.Thread Methods

- start( )
- sleep( ) (static – for current thread)
  - for a given time; let’s other threads run
- join( )
  - waits for another thread to complete
- yield( ) (static – for current thread)
  - let’s threads of equal or higher priority run (a *hint*)
- interrupt( )
  - set’s a flag in a thread object
- isAlive( )
- isInterrupted( )
- interrupted( ) (static – for current thread)

From *Effective Java*

- “The only use that most programmers will ever have for Thread.yield is to artificially increase the concurrency of a program during testing.”

```java
// Illustrates Thread.sleep()
class MyThread extends Thread
{
   private int delay;
   private int count;
   public MyThread(String name, int delay, int count)
   {
      super(name);
      this.delay = delay;
      this.count = count;
   }
   public void run()
   {
      for (int i = 0; i < count; ++i)
      {
         try
         {
            Thread.sleep(delay);
            System.out.println(getName());
         }
         catch (InterruptedException x)
         {}
      }
   }
}
```

```
class Independent2
{
   public static void main(String[] args)
   {
      Thread t1 = new MyThread("DessertTopping",
                                        100, 8);
      Thread t2 = new MyThread("FloorWax", 50, 4);
      t1.start();
      t2.start();
    }
}
```

Output

```
FloorWax
DessertTopping
FloorWax
FloorWax
DessertTopping
FloorWax
DessertTopping
DessertTopping
DessertTopping
DessertTopping
DessertTopping
DessertTopping
```

Thread Priority

- Threads receive a priority equal to their creating thread
- Thread.setPriority( )
  - Priority levels 1 through 10
  - Use Thread.MAX_PRIORITY (10), Thread.MIN_PRIORITY (1), Thread.NORM_PRIORITY (5; the default)
  - Doesn't map well to native priorities
    - NT has 7, Solaris 2^{31}
    - Platform-dependent behavior
- Only guarantee: threads with higher priorities tend to run more often than others

Thread Priority Example

```
class Priority
{
  public static void main(String[] args)
  {
    Thread t1 = new MyThread("DessertTopping", 500);
    Thread t2 = new MyThread("FloorWax", 500);
    t2.setPriority(Thread.MAX_PRIORITY);
    t1.start();
    t2.start();
  }
}
```

Output (compressed)

```
FloorWax (126)
DessertTopping (1)
FloorWax (70)
DessertTopping (1)
FloorWax (49)
DessertTopping (1)
FloorWax (119)
DessertTopping (1)
FloorWax (5)
DessertTopping (1)
FloorWax (1)
DessertTopping (1)
FloorWax (124)
DessertTopping (1)
FloorWax (6)
DessertTopping (493)
```

The Runnable Interface

- Alternative to extending java.lang.Thread
- Declares a run( ) method
- 2 virtues:
  - Separates task from thread objects
  - Leaves you free to extend another class
    - Java only supports single inheritance
- Thread has a constructor that takes a Runnable object

```java
// Illustrates a Runnable Object
class MyTask implements Runnable
{
    private int delay;
    private int count;
    private String name;
    public MyTask(String name, int delay, int count)
    {
        this.delay = delay;
        this.count = count;
        this.name = name;
    }
    public void run()
    {
        for (int i = 0; i < count; ++i)
        {
            try
            {
                Thread.sleep(delay);
                System.out.println(name);
            }
            catch (InterruptedException x)
            {}
        }
    }
}
```

The Main Program

```
class Independent4
{
   public static void main(String[] args)
   {
      Thread t1 = new Thread(
         new MyTask("DessertTopping", 100, 8));
      Thread t2 = new Thread(
         new MyTask("FloorWax", 200, 4));
      t1.start();
      t2.start();
   }
}
```

Making any Class a Concurrent Task

- Yet another design alternative
- Use an anonymous inner class
  - Extends java.lang.Thread
  - Has a run() that calls your entry point
- Add a "start" method to create and start the thread
- (See next slide…)

```
// Illustrates an anonymous "starter" nested class
class MyTask {
   private int delay;
   private int count;
   private String name;
   public MyTask(String name, int delay, int count) {
      this.delay = delay;
      this.count = count;
      this.name = name;
   }
   public void start() {       // The new method
      new Thread() {
         public void run() {doit();}
      }.start();
   }
   private void doit() {       // The entry point
      for (int i = 0; i < count; ++i) {
         try {
            Thread.sleep(delay);
            System.out.println(name);
         }
         catch (InterruptedException x) {}
      }
   }}
```

```
public class Anonymous
{
   public static void main(String[] args)
   {
      MyTask t1 = new MyTask("DessertTopping", 100, 8);
      MyTask t2 = new MyTask("FloorWax", 200, 4);
      t1.start();
      t2.start();
   }
}
```

Responsive UIs

- User perceives multiple, simultaneous actions
- User may want to cancel a task prematurely
  - e.g., to cancel a time-consuming operation
- "Counter" example
  - Displays count until "interrupted"
    - Not necessarily interrupted "immediately"

```java
// Stops a task (not entirely safe!)
import java.io.*;

class Counter implements Runnable
{
   private int count = 0;
   private boolean cancelled = false;

   public void run()
   {
      while (!cancelled)
      {
         System.out.println(count++);
         try
         {
            Thread.sleep(1000);
         }
         catch (InterruptedException x){}
      }
      System.out.println("Counter Finished");
   }
   void cancel()
   {
      cancelled = true;
   }
}
```

Stopping a Thread

```
class Cancel
{
   public static void main(String[] args) {
      System.out.println("Press Enter to Cancel:");
      Counter c = new Counter();
      Thread t = new Thread(c);
      // Worker thread
      t.setPriority(Thread.NORM_PRIORITY-1); // typical
      t.start();

      try {
         System.in.read(); // Wait for Enter keypress
      }
      catch (IOException x) {
         System.out.println(x);
      }
      c.cancel();      // Don't forget this!
      System.out.println("Exiting main");
   }
}
```

Output

```
Press Enter to Cancel:
0
1
2

Exiting main
Counter Finished
```

Did You Notice?

- Notice that main( ) finished before Counter.run( ) did
- The JVM runs until all *user* threads terminate

Thread Types

- User Threads
  - The default
  - Run until completion
- Daemon Threads (“DEE-mun”)
  - JVM doesn’t care about them
  - They die after the last user thread dies
    - Then the JVM halts

A Daemon Counter

```
// cancel() not needed
import java.io.*;

class Counter implements Runnable
{
   private int count = 0;

   public void run()
   {
      for (;;)
      {
         System.out.println(count++);
         try
         {
            Thread.sleep(1000);
         }
         catch (InterruptedException x){}
      }
   }
}
```

Setting a Daemon Thread

```
class Cancel2
{
   public static void main(String[] args)
   {
      System.out.println("Press Enter to Cancel:");
      Counter c = new Counter();
      Thread t = new Thread(c);
      t.setDaemon(true);                // ←
      t.start();

      try {
         System.in.read();
      }
      catch (IOException x) {
         System.out.println(x);
      }
      System.out.println("Exiting main");
   }
}
```

Output
(Daemon dies an unnatural death)

```
Press Enter to Cancel:
0
1
2

Exiting main
```

Interrupting a Thread

- Thread.interrupt( )
- Gets a thread's attention
  - Sets a flag in the thread
  - Thread.interrupted( ) (static, current thread)
    - Clears interrupted status
  - Thread.isInterrupted( ) (non-static)
    - Doesn't clear interrupted status
  - InterruptedException may occur:
    - If interrupted during a sleep(), wait(), or join() (and flag is cleared)
- Often used for asking a thread to quit

```java
// Interrupts a thread
import java.io.*;

class Counter extends Thread
{
   private int count = 0;

   public void run()
   {
      while (!interrupted())
      {
         System.out.println(count++);
         try
         {
            Thread.sleep(1000);
         }
         catch (InterruptedException x)
         {
            interrupt();  // recover interruption
         }
      }
      System.out.println("Counter Finished");
   }
}
```

```java
class Interrupt
{
   public static void main(String[] args)
   {
      System.out.println("Press Enter to Cancel:");
      Counter c = new Counter();
      c.start();

      try
      {
         System.in.read();
      }
      catch (IOException x)
      {
         System.out.println(x);
      }
      c.interrupt();
      System.out.println("Exiting main");
   }
}
```

Output

```
Press Enter to Cancel:
0
1
2

Exiting main
Counter Finished
```

Sharing Resources

- Two threads could contend for the same resource
  - *Race condition*
- Access could be interleaved, leaving data in an inconsistent state
- Solution: *critical sections*
  - Only one thread allowed in at a time
  - Allowed to "finish" before another thread gets a chance at the same code

Locks and Monitors

- Every object has a hidden lock object
  - Used to protect code blocks
- Monitor concept
  - Only allows one thread in at a time
  - Thread acquires a lock via some object
  - Other related threads wait until lock is released
    - Applies to all guarded methods for that object only
- Achieved with the **`synchronized`** keyword
  - Protects code (not data directly)
  - Make data private!

How `synchronized` Works (conceptually)

```
synchronized void f()
{
    <protected code>
}
```

is the same as the following pseudocode…

```
void f() {
   this.lock.acquire();
   try
   {
       <protected code>
   }
   finally
   {
       this.lock.release();
   }
}
```

Library Example

- Check-out system
  - Usually solved by database locks, but humor me
- Book class
- Must only allow one thread access to check-out check-in code
- Synchronized methods

```java
// Illustrates synchronized methods

class Book
{
   private final String title;
   private final String author;
   private String borrower;

   public Book(String title, String author)
   {
      this.title = title;
      this.author = author;
      borrower = null;
   }

   public synchronized boolean
   checkOut(String borrower)
   {
      if (isAvailable())
      {
         this.borrower = borrower;
         return true;
      }
      else
         return false;
   }
```

```
public synchronized boolean checkIn()
{
   if (!isAvailable())
   {
      borrower = null;
      return true;
   }
   else
      return false;
}

public String getTitle()
{
   return title;
}

public String getAuthor()
{
   return author;
}
```

```
    public synchronized boolean isAvailable()
    {
        return borrower == null;
    }

    public synchronized String getBorrower()
    {
        return borrower;
    }
}
```

Principles

- Always make data private
- Always protect access to shared data with a monitor (i.e., using **`synchronized`**)
- Synchronize as little code as possible
  - Blocks instead of entire methods:
    - **`{… synchronized (obj) {…} … }`**

Synchronizing Static Methods

- Same syntax
- Thread obtains lock from the *class* object

```
// Prevents interleaving via a monitor
class MyThread extends Thread
{
    private int delay;
    private int count;
    public MyThread(String name, int delay, int count)
    {
        super(name); // Optional thread name
        this.delay = delay;
        this.count = count;
    }
    public void run()
    {
        for (int i = 0; i < count; ++i)
        {
            try
            {
                Thread.sleep(delay);
                display(getName());
            }
            catch (InterruptedException x)
            {
                // Won't happen in this example
            }
        }
    }
```

```java
    synchronized static void display(String s) // <-
    {
        for (int i = 0; i < s.length(); ++i)
            System.out.print(s.charAt(i));
        System.out.println();
    }
}

class StaticLock
{
    public static void main(String[] args)
    {
        Thread t1 = new MyThread("DessertTopping",
                                 100, 8);
        Thread t2 = new MyThread("FloorWax", 200, 4);
        t1.start();
        t2.start();
    }
}
```

Output (not interleaved)

```
DessertTopping
FloorWax
DessertTopping
DessertTopping
FloorWax
DessertTopping
DessertTopping
FloorWax
DessertTopping
DessertTopping
FloorWax
DessertTopping
```

From *Effective Java*

- "Whenever multiple threads share mutable data, each thread that reads or writes the data must obtain a lock."
- "Do not let the guarantee of atomic reads and writes deter you from performing proper synchronization."
- "The use of the `volatile` modifier … is an advanced technique… the extent of its applicability will not be known until the ongoing work on the memory model is complete."
- "As a rule … do as little work as possible inside synchronized regions."

New in JDK 1.4

- The holdsLock( ) Method
- Used with assertions (also new in 1.4)
- For safety
  - asserting that a lock is or isn't held
- For efficiency
  - Calling a synchronized method that you already have a lock on still has a cost
  - If a method will only be called in a monitor, don't synchronize it, just assert that the thread holds the lock

holdsLock( ) Example
(courtesy Allen Holub)

```
public synchronized void f()
{   //...
    workhorse();
    //...
}

private /* not synchronized */ void workhorse()
{   assert Thread.holdsLock(this) : "Improper call";
    //...
}

// Fires assertion:
protected void g(){ workhorse(); }
```

Deadlock

- When multiple threads wait on each other forever
- Can easily occur when multiple resources are shared
  - Suppose threads X and Y both need resources A and B
  - If X has A and is waiting for B, and Y has B and is waiting for A, Big Trouble!

Managing Multiple Resources

- Techniques to prevent deadlock:
  - Fixed locking order
  - Try-and-back-off

Fixed Locking Order

- All tasks obtain locks for the resources in the *same order*, and release them LIFO
- Not always possible
  - Only works if all threads must share the same, fixed set of resources
  - Dining Philosophers can’t be solved this way

Try-and-back-off

- See if *all* the locks are available, locking each one as you go
  - Order not as crucial with this technique
- If a lock is unavailable, back out all locks and start all over
- More expensive
- Requires a "try-lock" method
  - Tells whether an object's lock is in use
  - Java doesn't have one
    - But it will!

```
class Resources {
   static Integer A = new Integer(1);
   static Integer B = new Integer(2);
}

class X extends Thread {
   public void run()
   {
      for (int i = 0; i < 10; ++i) {
         synchronized(Resources.A) {
            synchronized(Resources.B) {
               System.out.println("X in process");
               try {
                  Thread.sleep(1000);
               }
               catch (InterruptedException x) {
               }
            }
         }
      }
   }
}
```

```
class Y extends Thread
{
   public void run()
   {
      for (int i = 0; i < 10; ++i) {
         synchronized(Resources.B) {      // oops!
            synchronized(Resources.A) {   // oops!
               System.out.println("Y in process");
               try {
                  Thread.sleep(100);
               }
               catch (InterruptedException x) {
               }
            }
         }
      }
   }
}
class Deadlock {
   public static void main(String[] args) {
      (new X()).start();
      (new Y()).start();
   }
}
```

The Dining Philosophers

- n philosophers seated at a round table
- 1 chopstick between each (n total)
- Spend their time eating or thinking
- To eat, must obtain both right and left chopstick
- Try-and-back-off works
  - Using Win32 *WaitForMultipleObjects*
- How to solve in Java?

Avoiding Chopstick Wars

- Must prevent circular waiting
  - If every philosopher tried for Left chopstick, then Right, deadlock would occur quickly
  - Instead, have *all but 1* philosopher use the Left-Right sequence
  - Have 1 use Right-left to break the cycle
- Example from Thinking in Java, 3rd Ed.
  - DiningPhilosophers.java

Inter-Thread Communication

- Allows threads to cooperatively solve a problem
- One threads waits for a condition to occur
  - To be "notified" by another thread
- The thread that causes the condition notifies other threads, so they can proceed
  - The notify-ees still have to compete for CPU time
- Typical in producer-consumer cases

Methods for Inter-Thread Communication

- Object.wait( )
  - Thread blocks until it is notified
    - And releases its lock
  - Optional timeout argument
- Object.notify( )
  - Another thread is awakened arbitrarily
    - The notifying thread must release the lock
- Object.notifyAll( )
  - All threads waiting on the lock awaken
- Must run in a monitor!

Object.wait( )

- Equivalent (sort of) to:

```
this.lock.release();
this.condition.wait_for_true();
this.lock.acquire();
```

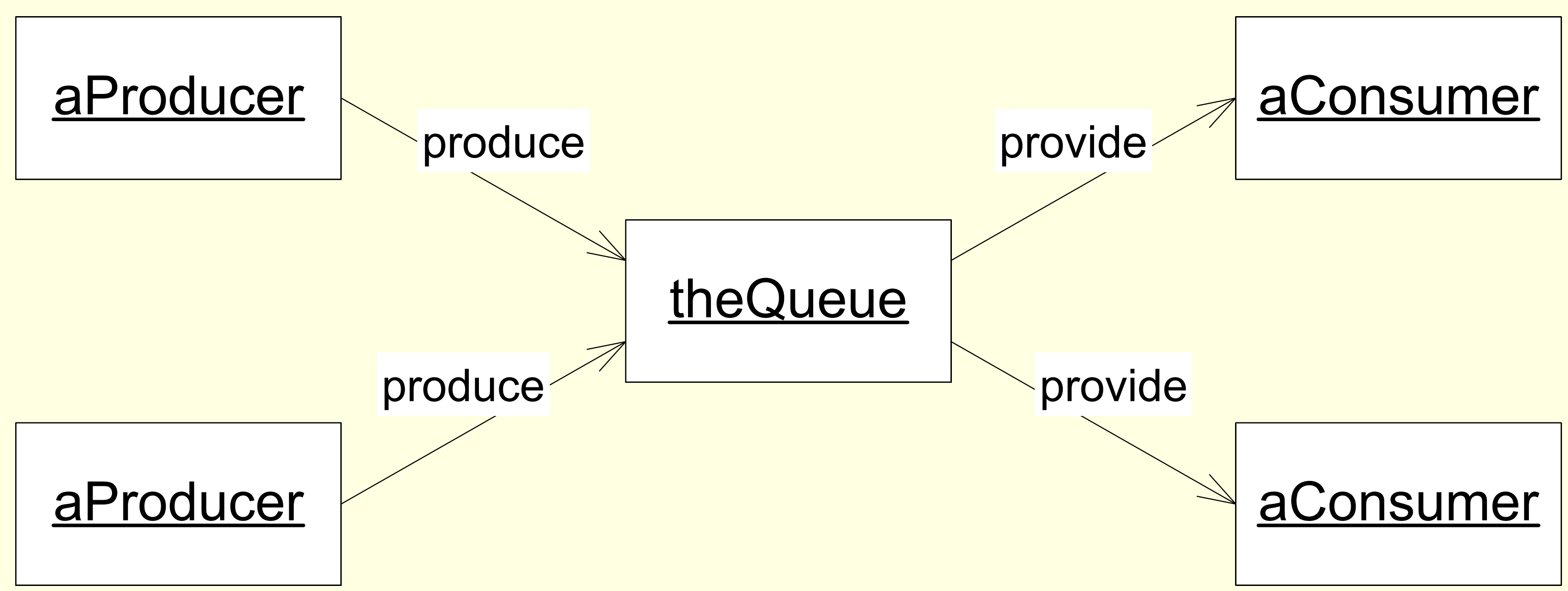
- Must check condition in a *loop*
  - A loop that calls wait( )
  - You could get notified on a different condition
  - Or another thread might immediately invalidate the condition

Object.notify vs. Object.notifyAll

- You usually use notifyAll( )
  - Wakes up all waiting threads in the wait queue for the lock
  - They compete for CPU, and check the condition
  - Always use if there are multiple conditions
  - Analogous to the Observer Design Pattern
- Use notify( ) only if
  - Threads are waiting on the same condition
  - Only one thread should respond to the condition

A Producer-Consumer Example

- Producer threads populate a shared queue
  - Call notify after inserting an element
- Consumer threads remove queue elements
  - Wait until queue has something in it
- Access to queue must be synchronized
  - Threads synchronize on queue object


aProducer
produce
aProducer
produce
theQueue
provide
aConsumer
provide
aConsumer

```java
import java.util.*;

class Queue
{
   // The shared buffer:
   static Queue store = new Queue();

   private LinkedList data = new LinkedList();

   public void put(Object o)
   {
      data.addFirst(o);
   }
   public Object get()
   {
      return data.removeLast();
   }
   public int size()
   {
      return data.size();
   }
}
```

```
class Producer extends Thread
{
   static Random numbers = new Random();
   Producer(String id) {
      super(id);
   }
   public void run() {
      // Generate some elements for the Queue
      for (;;) {
         int number = numbers.nextInt(1000);
         System.out.println(getName() + " producing "
                                   + number);
         synchronized(Queue.store) {
            Queue.store.put(new Integer(number));
            Queue.store.notify();
         } // Give up lock!
      }
   }
}
```

```java
class Consumer extends Thread
{
   Consumer(String id) {
      super(id);
   }
   public void run() {
      for (;;) {
         synchronized(Queue.store) {
            while (Queue.store.size() == 0)
            {
                try {
                    Queue.store.wait();
                }
                catch (InterruptedException x) {
                   interrupt();
                }
            }
            System.out.println(getName() +
                " consuming " + Queue.store.get());
         }
      }
   }
}
```

```java
class CommTest
{
   public static void main(String[] args)
      throws InterruptedException
   {
      // Start Producers
      new Producer("Producer1").start();
      new Producer("Producer2").start();

      // Start Consumers
      new Consumer("Consumer1").start();
      new Consumer("Consumer2").start();
   }
}
```

```
/* Output:
Producer1 producing 289
Producer1 producing 34
Producer1 producing 975
Producer1 producing 804
Producer1 producing 913
Producer1 producing 514
Producer1 producing 94
Producer1 producing 425
Producer2 producing 863
Consumer1 consuming 289
Consumer2 consuming 34
Producer2 producing 758
Consumer1 consuming 975
Consumer2 consuming 804
Producer2 producing 274
Consumer1 consuming 913
Consumer2 consuming 514
Producer2 producing 311
Consumer1 consuming 94
Consumer2 consuming 863
Producer2 producing 997
(etc...)
*/
```

A Better Solution

- The previous example doesn't halt!
- Suppose we terminate Producers
  - After a certain number of inserts, say
- How can the Consumers detect that all the Producers are finished?
- Need an independent object
  - Tracks the number of active Producers
  - Consumers halt when reaches zero

```java
import java.util.*;

class Counter
{
   private int count;

   public Counter()
   {
      count = 0;
   }
   public synchronized void increment()
   {
      ++count;
   }
   public synchronized void decrement()
   {
      --count;
   }
   public synchronized int get()
   {
      return count;
   }
}
```

```
class Producer extends Thread
{
   private static Random numbers = new Random();
   private Counter counter;

   Producer(String id, Counter counter) {
      super(id);
      this.counter = counter;
   }
   public void run() {
      counter.increment();   // ←

      // Generate some elements for the Queue
      for (int i = 0; i < 8; ++i) {
         int number = numbers.nextInt(1000);
         System.out.println(getName() +
                           " producing " + number);
         synchronized(Queue.store) {
            Queue.store.put(new Integer(number));
            Queue.store.notify();
         }
      }
```

```
      synchronized(Queue.store) {
         // Prevent infinite loop
         // (because there are multiple consumers)
         counter.decrement();
         Queue.store.notifyAll(); // ←
      }
      System.out.println("\t" + getName() +
                         " finished");
   }
}

class Consumer extends Thread
{
   private Counter counter;

   Consumer(String id, Counter counter)
   {
      super(id);
      this.counter = counter;
   }
```

```java
    public void run()
    {
        for (;;) {
            synchronized(Queue.store) {
                while (Queue.store.size() == 0) {
                    if (counter.get() == 0) {
                        // Producers done and queue is empty
                        System.out.println("\t" + getName()
                                           + " terminating");
                        return;
                    }
                    try {
                        Queue.store.wait();
                    }
                    catch (InterruptedException x) {}
                }
                System.out.println(getName() + " consuming "
                                   + Queue.store.get());
            }
        }
    }
}
```

```
class CommTest2
{
   public static void main(String[] args)
   {
      // Start Producers
      Counter counter = new Counter();
      new Producer("Producer1", counter).start();
      new Producer("Producer2", counter).start();

      // Start Consumers
      new Consumer("Consumer1", counter).start();
      new Consumer("Consumer2", counter).start();
   }
}

/* Output:
Producer1 producing 386
Producer1 producing 240
Producer1 producing 982
Producer1 producing 453
Producer1 producing 878
Producer1 producing 228
Producer1 producing 245
…
```

```
…
Consumer1 consuming 189
Consumer2 consuming 740
Producer2 producing 761
Consumer1 consuming 264
Consumer2 consuming 686
Producer2 producing 586
Consumer1 consuming 761
Producer2 producing 847
Consumer1 consuming 586
Producer2 producing 161
Consumer1 consuming 847
Consumer1 consuming 161
        Producer2 finished
Consumer1 consuming 245
Producer1 producing 329
Consumer2 consuming 329
        Consumer1 terminating
        Consumer2 terminating
        Producer1 finished
*/
```

Things to Remember

- Every wait( ) needs a notify
- Call notifyAll( ) if more than one thread needs to be notified
- Always use condition loops

Thread Groups and Exceptions

- Exceptions belong to a Thread
  - Both are stack-based
- When an exception occurs in a monitor, the lock is released
  - Duh!
- When an exception is not caught:
  - It falls off the top of the stack
  - It has no where to go

Uncaught Exceptions

- The current thread dies
- ThreadGroup.uncaughtException( ) is called
  - Default behavior is to print stack trace to System.err
  - You can override it
    - That’s why you would create a new ThreadGroup
    - Otherwise thread groups are useless!!

An Uncaught Exception

```
class MyThread extends Thread
{
    public void run()
    {
        System.out.println("Throwing in " +
                           "MyThread");
        throw new RuntimeException();
    }
}
```

```
class Uncaught
{
    public static void main(String[] args)
    {
        MyThread t = new MyThread();
        try
        {
            t.start();
            Thread.sleep(1000);
        }
        catch (Exception x)
        {
            System.out.println("This won't happen");
        }
        System.out.println("Exiting main");
    }
}

/* Output:
Throwing in MyThread
java.lang.RuntimeException
        at MyThread.run(Uncaught.java:7)
Exiting main
*/
```

Using Thread Groups

- Extend ThreadGroup
- Override uncaughtException( )
- Place thread in group
  - By using appropriate constructor

```java
// Join a thread group
class MyThread extends Thread
{
    public MyThread(ThreadGroup g, String name)
    {
       super(g, name);
    }
    public void run()
    {
       System.out.println("Throwing in " + "MyThread");
       throw new RuntimeException();
    }
}
```

Overriding uncaughtException( )

```
class MyGroup extends ThreadGroup
{
   public MyGroup(String name)
   {
      super(name);
   }
   public void uncaughtException(Thread t,
                                 Throwable x)
   {
      System.out.println(t + " aborted with a " + x);
   }
}
```

```
class Uncaught2
{
   public static void main(String[] args)
   {
      MyGroup g = new MyGroup("My Thread Group");
      MyThread t = new MyThread(g, "My Thread");
      try {
         t.start();
         Thread.sleep(1000);
      }
      catch (Exception x) {
         System.out.println("This won't happen");
      }
      System.out.println("Exiting main");
   }
}

/* Output:
Throwing in MyThread
Thread[My Thread,5,My Thread Group] aborted with a
java.lang.RuntimeException
Exiting main
*/
```

Finis, El Fin, O Fim, The End