Lecture 9: Concurrent Programming with Java

ATSD Fall 2003
Dec. 1, 2003
Introduction to Concurrent Programming

What is concurrent programming?

A **sequential** program has a single thread of control.

A **concurrent** program has multiple threads of control allowing it perform multiple computations in parallel and to control multiple external activities which occur at the same time.
Introduction to Concurrent Programming

- **Multiple CPUs**

![Diagram of multiple CPUs](image)

- **Single CPU**

![Diagram of single CPU](image)

**Physically parallel**

**Instruction interleaving**
Programming Java Threads
Overview

- Java Threads: Basics
- Thread Scheduling
- Thread Control
- Thread Synchronization
What is a thread?

Definition:

- A thread is a single sequential flow of control within a program.

- A thread runs within the context of a program’s process and takes advantage of the resources allocated for that process and its environment.
What is a thread? Cont’d

◆ Multiple threads:
  ● Multiple threads can run at the same time in the same process performing different tasks
  ● Multiple threads can share the resources of the process they share

◆ First there was single process, then there was fork(), which gave us multi process, now there are threads

◆ Do NOT confuse with parallel processing
Multiprocesses vs. Multithreads

- Threads are different from processes created by fork().

**process**
- Child process gets a *copy* of parents variables
- Relatively expensive to start
- Don't have to worry about concurrent access to variables

**thread**
- Child process *shares* parents variables
- Relatively cheap to start
- Concurrent access to variables is an issue
A (heavyweight) process in an operating system is represented by its code, data and the state of the machine registers, given in a descriptor. In order to support multiple (lightweight) threads of control, it has multiple stacks, one for each thread.
Why is multi-threading needed?

Without multi-threading

user clicks button 1:
action method 1 starts
method 1 ...
user clicks button 2:
method 1 ...
method 1 ...
end of action method 1
action method 2 starts
method 2 ...
method 2 ...
method 2 ...
end of action method 2

With multi-threading

user clicks button 1:
action method 1 starts
method 1 ...
user clicks button 2:
action method 2 starts
method 2 ...
method 1 ...
method 1 ...
method 2 ...
end of action method 1
method 2 ...
end of action method 2
Introduction to Concurrent Programming

Why concurrent programming?

- Performance gain from multiprocessing hardware
  - parallelism.
- Increased application throughput
  - an I/O call need only block one thread.
- Increased application responsiveness
  - high priority thread for user requests.
- More appropriate structure
  - for programs which interact with the environment, control multiple activities and handle multiple events.
Java Threading Model

- Java has threads built-in (`java.lang.thread`).
- Applications consist of at least one thread:
  - Often called ‘main’
- The Java Virtual Machine creates the *initial thread* which executes the `main` method of the class passed to the JVM.
- The methods executed by the ‘main’ thread can then create other threads:
  - Either explicitly or implicitly.
Defining threads in Java, 1

A Thread class manages a single sequential thread of control. Threads may be created and deleted dynamically.

The Thread class executes instructions from its method `run()`. The actual code executed depends on the implementation provided for `run()` in a derived class.

```java
class MyThread extends Thread {
    public void run() {
        //......
    }
}

Thread x = new MyThread();
```
Defining threads in Java, 2

Since Java does not permit multiple inheritance, we often implement the `run()` method in a class not derived from Thread but from the interface Runnable.

```java
// public interface Runnable
{  
    public abstract void run();
}

class MyRun implements Runnable{
    public void run() {
        //.....
    }
}

Thread x = new Thread(new MyRun());
```
Define your own run() method
   
   ```java
   class MyThread extends Thread {
       public void run() {
           System.out.println("MyThread.run()")
       }
   }
   ```

Instantiate your own class instance, then invoke its "start" method:
   
   ```java
   MyThread mythread = new MyThread();
   mythread.start();
   ```

   *Note: "start()" is a native method. And the invocation returns immediately to the caller*
Creating & Executing Threads, 1 (Runnable)

- **Runnable interface has single method**
  - `public void run()`

- **Implement a Runnable and define run()**

```java
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("MyRunnable.run()");
    }
    //other methods and data for this class
}
```
Cont’d: Using a Runnable

Class Main {
    public static void main(String[] args) {
        MyRunnable myrun = new MyRunnable();
        Thread t1 = new Thread(myrun);
        t1.start();
        System.out.println("Inside Main()’");
    }
}

• Thread’s run() method invokes the Runnable’s run() Method
Example: Self-Starting Threads

```java
Class AutoRun implements Runnable {
    public AutoRun() {
        new Thread(this).start();
    }
    public void run() {
        System.out.println("AutoRun.run()");
    }
}
class Main {
    public static void main(String[] args) {
        AutoRun t1 = new AutoRun();
        System.out.println("Inside Main()");
    }
}
```
Thread Names

- All threads have a name to be printed out.
- The default name is of the format: Thread-No
  Thread-1, Thread-2, ...
- User-defined names can be given thru constructor:

  Thread myThread = new Thread("Happy Thread");

- Or using the “setName(aString)” method.
- There is a method in Thread class, called “getName()”, to obtain a thread’s name
Thread life-cycle in Java

An overview of the life-cycle of a thread as state transitions:

```
new Thread()
```

`start()` causes the thread to call its `run()` method.

The predicate `isAlive()` can be used to test if a thread has been started but not terminated. Once terminated, it cannot be restarted.
Thread *alive* states in Java

Once started, an *alive* thread has a number of substates:

- **Runnable (Ready)**
  - `start()`
  - `dispatch`
  - `yield()`
  - `sleep()`, `wait()`
  - `notify()`, `notifyAll()`
  - `wait()`, `sleep()`, `I/O blocking`

- **Non-Runnable**
  - `start()`, or `run()` returns
  - `stop()`, or `I/O completed`

- **Running**
  - `start()`, `stop()`, or `run()` returns
Life Cycle, Detailed View

◆ More detailed view

Deprecated: suspend(), resume(), stop()
Priorities and Scheduling

- All Java threads have a priority value, currently 1 and 10.
- There are three constant fields in the class Thread:
  - `Thread.MIN_PRIORITY`
  - `Thread.MAX_PRIORITY`
  - `Thread.NORM_PRIORITY (=5, default priority)`
- Priority can be changed at any time
  - `setPriority(int newPriority)`
- Initial priority is that of the creating thread or `Thread.NORM_PRIORITY (5)`
- Maximum priority can be limited by the ThreadGroup
Scheduling Threads

◆ Preemptive scheduling: JVM gives preference to higher priority threads. (Not guaranteed)

◆ The thread with highest priority will run until one of the following conditions is true:
  ● A higher priority thread becomes runnable.
  ● It yields, blocked, or its run method exits.
  ● On systems that support time-slicing, its time allotment has expired.

◆ Threads of equal priority are not necessarily scheduled in a FCFS manner.
Scheduling Threads, Cont’d

- Time-slicing is allowed but not required
  - Semantics of time-slicing (e.g. within priority group, across priorities) are not defined
- Java running Green threads on Solaris does not time-slice. (Old versions of Solaris)
- Implementations can use true multi-processing (Win 32 and Solaris 2.6 with later versions of JDK 1.1)
- There’s no interface to query the current VM; SMP model
Non time-sliced threads run until:

- they end
- they are terminated
- they are interrupted (?)
- Higher priority threads interrupts lower priority threads
- they go to sleep
- they block on some call
- reading a socket
- waiting for another thread
- they yield (Co-operative Multitasking)
Example: Testing for Time-slicing

class InfinityThread extends Thread {
    public void run() {
        while (true)
            System.out.println( "From: " + getName() );
    }
}
class TestThread {
    public static void main( String[] args ) {
        InfinityThread first = new InfinityThread();
        InfinityThread second = new InfinityThread();
        first.start();
        second.start();
    }
}

Output if Time-sliced:
"From: Thread-5" and "From: Thread-4" will intermix "forever"

Output if not Time-sliced:
"From: Thread-4" will repeat "forever"
class YieldThread extends Thread {
    public void run() {
        for (int count = 0; count < 4; count++) {
            System.out.println(count + "From: " + getName());
            yield();  // cooperative multitasking
        }
    }
}

class TestThread {
    public static void main(String[] args) {
        YieldThread first = new YieldThread();
        YieldThread second = new YieldThread();
        first.start();    second.start();
        System.out.println( "End" );
    }
}

Output
End
0 From: Thread-4
0 From: Thread-5
1 From: Thread-4
1 From: Thread-5
2 From: Thread-4
2 From: Thread-5
3 From: Thread-4
3 From: Thread-5
Setting Priorities

- Continuously running parts of the program should have lower-priority than rarer events (otherwise the GUI may not be able to refresh)

- User input should have very high priority (GUI)

- A thread that continually updates some data is often set to run at MIN_PRIORITY
Types of Threads

◆ When does a non-main thread end?
◆ **User threads:**
  continue to execute until their run method ends or an exception propagates beyond the run method.
◆ **Daemon threads:**
  When all user threads are done, the program ends and all daemon threads are stopped by JVM.
◆ Use `aThread.setDaemon(true)` to make a thread daemon. Another method “isDaemon()” to query.
◆ The type a thread can **only** be changed before its `start()` method is called.
◆ The “checkAccess()” method determines if the currently running thread has permission to modify this thread.
Threads Control

How to control a thread’s execution? e.g., cancel a thread’s execution *asynchronously*, graceful shutdown of a program.

Methods of interest:
- `run()`, `start()`, `yield()`, `destroy()`, `stop()`
- `sleep(long)`, `suspend()`, `resume()`, `join()`
- `isAlive()`, `interrupt()`, `Thread.interrupted()`, `isInterrupted()`

“InterruptedException”
Invoking run()

run() is just a method

- So it can be invoked directly, but only executed sequentially

```java
MyRunnable t1 = new MyRunnable();
t1.run();
```

run() is public

- So we can't hide it

run() may need protection. How?

Thread identity is the key
Thread Identity

- `Thread.currentThread()`
  - Static method
  - Returns reference to the running thread

- **Compare running thread with created thread**
  ```java
class AutoRun implements Runnable {
    private Thread _me;
    public AutoRun() {
      me_ = new Thread(this);
      me_.start();
    }
  }
```
Thread Identity (cont...)

```java
Public void run() {
    if (Thread.currentThread() == _me)
        System.out.println("AutoRun.run()" newX);
}

class Main {
    public static void main(String[] args) {
        AutoRun t1 = new AutoRun();
        t1.run();  //no printout
        System.out.println("Inside Main()" newX);
    }
}
```
Creation and Termination

- **void start()** (a *native* method)
  - Creates a new thread of control to execute the `run()` method of the `Thread` object
  - Can only be invoked once per `Thread` object
    - `else InvalidThreadStateException`

- **void stop()**
  - *Deprecated!*
  - Stops execution in a `safe` manner by causing a `ThreadDeath` exception
  - Details are complex

- **void destroy()**
  - Abrupt termination? [Not implemented]
Scheduling the Current Thread

- `static void sleep(long millis)`
  - Blocks this thread for at least the time specified
  - `sleep(0)` is not defined as a special case
  - Variant that takes a nanosecond argument
    `sleep(millisecons, nanoseconds)`

- `static void yield()`
  - Allows the scheduler to select another runnable thread (of the same priority)
  - No guarantees as to which thread
Scheduling Threads

- `void suspend()`
  - `Deprecated!`
  - Temporarily pauses a thread's execution
  - All resources retained
  - Be careful suspending other threads
    - what was the thread doing?
    - what resources (eg. locks) did it hold?
  - Self-suspension can be safe

- `void resume()`
  - `Deprecated!`
  - Allows a suspended thread to continue
Thread Waiting & Status Check

- `void join()`, `void join(long)`, `void join(long, int)`
  - One thread (A) can wait for another thread (B) to end.
    
    ```java
    // in thread A
    threadB.join()
    ```

- `boolean isAlive()`
  - Returns true if the thread has been started and not stopped.
  - Cannot differentiate between a Runnable thread and a Not Runnable thread.
News Flash!

The following Thread methods have been listed as deprecated for Java 1.2

- `stop()`, `stop(Thrower t)`
- `suspend()`
- `resume()`

Why?

- Too dangerous, lead to inconsistent state

Aligns API closer to POSIX
Why Deprecating stop()?

- Not safe, inconsistent state, deadlock!

A deadlock example

```java
class some_class {
    //...

    synchronized void f() {
        Thread.currentThread().stop();
    }
    // the lock is not released by stop().
}
```
What Replaces Stop?

Use a state variable and check it periodically. For example, in an applet:

Thread blinker; //= new Thread(this); in start()

class blinker;
public void stop() { blinker = null; }
public void run() {
    Thread thisThread = Thread.currentThread();
    while (blinker == thisThread) {
        try { ... } catch (InterruptedException e) {
        }
        repaint();
    }
}
Interrupting Threads (JDK 1.1)

- Public void `interrupt()` throws `SecurityException`
  - Signal a thread object that it should stop running
  - Asynchronous notification
    - Does not stop the thread right away
    - Sets an “interrupted” boolean to true
  - This flag can be tested and then an `InterruptedException` thrown

- What can be interrupted? Methods that throw an `InterruptedException`
  - `sleep()`, `wait()`, `join()`
  - I/O operations that throw `IOException`
    - `InterruptedIOException` is derived from it
Interrupting Threads (JDK 1.4)

- Interrupts this thread.
- First the `checkAccess` method of this thread is invoked, which may cause a `SecurityException` to be thrown.
- If this thread is blocked in an invocation of the `wait()`, `wait(long)`, or `wait(long, int)` methods of the `Object` class, or of the `join()`, `join(long)`, `join(long, int)`, `sleep(long)`, or `sleep(long, int)` methods of this class, then its interrupt status will be cleared and it will receive an `InterruptedException`.
- If this thread is blocked in an I/O operation upon an `interruptible channel` then the channel will be closed, the thread's interrupt status will be set, and the thread will receive a `ClosedByInterruptException`.
- If this thread is blocked in a `Selector` then the thread's interrupt status will be set and it will return immediately from the selection operation, possibly with a non-zero value, just as if the selector's `wakeup` method were invoked.
- If none of the previous conditions hold then this thread's `interrupt status` will be set.
Interruptions (cont ‘d)

◆ **static** boolean interrupted()
  - Returns true if the **current thread** has been interrupted
  - **Clears** the interrupt state

◆ boolean isInterrupted()
  - Returns true if the **specified thread** has been interrupted
  - Does not change the interrupt state
Using Interruptions to Cancel a Thread

◆ The cancellable threads must check the interruption status.

```java
if (Thread.currentThread().isInterrupted())
    ... cancellation code ...
// normal operation

OR

try {

    Thread.sleep(...); // wait(); join()

} catch (InterruptedException ex) {
    ... cancellation code ...
}
```
When to Check Interruption

- **Method entry**
- **Before a `synchronized` block.**

  ```java
  if (Thread.currentThread().isInterrupted()) return;
  synchronized(this) { . . . }
  ```

- **Outer loops surrounding computation-intensive processing.**

  ```java
  class Daemon implements Runnable {
      public void run() {
          while (!Thread.interrupted())
              oneStep();
      }
      void oneStep() { ... }
  . . . }
  ```
When to Check, cont’d

◆ Sections of methods that perform checks that lead to the throwing of any other kind of failure exception.

◆ Before any action that is difficult or impossible to later undo. (*point of no return*)
class StopWorker extends Thread {
    public void run() {
        long sum = 0;
        for (int i=0; i<5000000; i++) {
            sum = sum + i; // do some work
            // every n iterators... check isInterrupted()
            if (i%100000 == 0) {
                if (isInterrupted()) {
                    // clean up, exit when interrupted
                    System.out.println(getName() + " interrupted");
                    return;
                }
                System.out.println(getName() + " " + i);
            }
            Thread.yield();
        }
    }
}
public static void main(String[] args) {
    StopWorker a = new StopWorker();
    StopWorker b = new StopWorker();

    System.out.println("Starting...");
a.start();
b.start();
try {
    // sleep a little, so they make some progress
    Thread.sleep(100);
} catch (InterruptedException ignored) {}  
a.interrupt();
b.interrupt();
System.out.println("Interruption sent");
try {
    a.join();
b.join();
} catch (Exception ignored) {}  
System.out.println("All done");
}
Interruption() example output

- /*
- Starting...
- Thread-0 0
- Thread-1 0
- Thread-1 100000
- Thread-0 100000
- Thread-1 200000
- ...
- Thread-0 900000
- Interruption sent
- Thread-0 interrupted
- Thread-1 interrupted
- All done
- */
Interruptions and IO

- There is an InterruptedIOException class, derived from IOException. But it doesn’t quite work through JDK 1.1.6.

- To avoid indefinite IO waiting,
  - close the operating stream to force an IOException.
  - implement one’s own non-blocking IO streams.

- When a socket times out, it will throw an InterruptedIOException.
  
  ```java
  socket.setTimeout(6000);
  ```

- JDK 1.4 NewIO introduces Non-Blocking I/O
public class NonBlockingInputStream extends FilterInputStream {
    private int pollDuration;
    private long timeout;
    public NonBlockingInputStream( InputStream input, int pollDelay,
                                    long timeout )
    {
        super( input );
        pollDuration = pollDelay;
        this.timeout = timeout;
    }

    public int read() throws IOException {
        pollInput();
        return in.read();
    }

    public int read(byte inputBuffer[], int offset, int length) throws IOException {
        pollInput();
        return in.read( inputBuffer, offset, length);
    }

    private void pollInput() throws IOException {
        long startTime = System.currentTimeMillis();
        while ( in.available() <= 0 )
        {
            try {
                if ( System.currentTimeMillis() - startTime >= timeout )
                    throw new IOException( "Timeout on read");
                Thread.sleep( pollDuration );
            } catch ( InterruptedException stopReading ) {
                throw new InterruptedIOException( "read() interrupted" );
            }
        }
    }
}
Thread Termination

Five ways to terminate a thread:
- The run() method returns normally
- Uncaught exception thrown
- stop() invoked
- destroy() invoked (Not implemented)
- exit() invoked (System or Runtime)
Suppose you want to implement a Timer class in Java. You want to be able to program your class to:

- start the timer
- have the timer measure some time interval

then carry out some action when the correct time has elapsed

Use a callback function to link the timer and the client.

interface Timed {
    public void doAction();
}
class Timer extends Thread {
    long interval;
    Vector timedObjects;  Vector wakeUpTimes;
    Timer(long t) { interval = t; }
    public void register(Timed obj, long sleepTime) {
        timedObjects.addElement(obj);
        wakeUpTimes.addElement(new Long(sleepTime+System.getCurrentTimemills() ));
    }
    public void run() {
        while (true) {
            Thread.sleep(interval);
            for (int i=0; i<timeObjects.size();  i++) {
                long t = System.getCurrentTimemills();
                if (t >= ((Long) wakeUpTime.elementAt(i) ).longValue()) {
                    ( (Timed) timedObjects.elementAt(i)).doAction();
                }
            }
        }
    }
}
Inter-Threads Communication

- Shared static variables (the same class)
- PipeInputStream and PipeOutputStream
- PipeReader and PipeWriter
class TestIO {
    public static void main(String[] args) throws IOException {
        PipedReader inPipe = new PipedReader();
        PipedWriter outPipe = new PipedWriter(inPipe);
        PrintWriter cout = new PrintWriter(outPipe);
        ReadThread reader = new ReadThread("Read", inPipe);
        reader.setPriority(6); // 5 is normal priority
        reader.start();
        System.out.println("In Main");
        cout.println("Hello");
        System.out.println("End");
    }
}
Concurrency Issues: Thread Synchronization
The advantage of threads is that they allow many things to happen at the same time.

The problem with threads is that they allow many things to happen at the same time.

- **Safety**
  - "Nothing bad ever happens", no race condition

- **Liveness**
  - "Something eventually happens", no deadlock
Mutual Exclusive Data Access

Assume we have two threads in a bank accessing the same account:

Account acct = new Account();

Athread:                         Bthread:
|                               B = acct.getBalance();
A = acct.getBalance();          | |
|                               B = B - 1000 (withdraw)
A = A + 1000 (deposit)          | |
|                               balance = B
balance = A                     | |

mistake!!!
class Account
{
    private float balance = 0;

    public synchronized void deposit(float f)
    {
        balance += f;
    }

    public synchronized void withdraw(float f)
    {
        balance -= f;
    }
}
Objects and Locks

- Every Java object has an associated lock acquired via:
  - synchronized statements
    ```java
    synchronized( foo ){
        // execute code while holding foo's
    }
    ```
  - synchronized methods
    ```java
    public synchronized void op1(){
        // execute op1 while holding `this' lock
    }
    ```

- Only one thread can hold a lock at a time
Objects and Locks (cont ...)

- If the lock is unavailable the thread is blocked until it is allocated the lock
- Locking and unlocking are automatic
- Java locks are reentrant:
  - A thread hitting synchronized passes if the lock is free or it already possesses the lock, else waits
  - Recursive methods don't deadlock
- Un-synchronized code ignores the lock
- A lock on an object also locks all access to the object via synchronized methods.
Synchronization and Static Members

◆ Synchronization on class objects
  ● synchronized static methods
    ```java
class SomeClass {
    static int[] classData;
    public synchronized static void initialize(int size, int startValue) {
        ...
    }
}
```
  ● synchronized statements
    ```java
    synchronized(this.getClass())
    {
        . . .
    }
    ```
public void fred( int value )
{
    synchronized( lock_1 )
    {
        synchronized( lock_2 )
        {
            . . .
        }
    }
}

public void barney( int value )
{
    synchronized( lock_2 )
    {
        synchronized( lock_1 )
        {
            . . .
        }
    }
}
class Notifying_queue
{
    private static final queue_size = 10;
    private Object[] queue = new Object[queue_size];
    private int head = -1;         private int tail = -1;
    public void synchronized enqueue( Object item )
    {
        queue[++head %= queue_size] = item;
        this.notify();                  // The "this" is there only to
                                            // improve readability.
    }
    public Object synchronized dequeue( )
    {
        try
        {
            if ( head == tail )   // This is a bug
                this.wait();
        }
        catch( InterruptedException e )
        {
            return null;  // abandon waiting
        }
        return queue[++tail %= queue_size];
    }
}
Nested-Monitor Lockout Example, Cont'd

class Notifying_queue
{
    // blocks on dequeue from
    // an empty queue. ...
}
class Black_hole
{
    private Notifying_queue queue = new Notifying_queue(5):
    public synchronized void put( Object thing )
    {
        queue.enqueue( thing );
    }
    public synchronized Object get()  // if executed first, get bolted
    {
        return queue.dequeue();
    }
}
Avoiding Nested-Monitor Lockout

- Don't make blocking calls in synchronized methods

- Make sure there's a way to talk to the blocking object via another class or a nonsynchronized method (e.g., not making “put” synchronized)
Another Nested-Monitor Lockout

class Friendly extends Thread {
    private Friendly aFriend;
    public Friendly( String name ) { super( name ); }
    public void setFriend( Friendly myFriend )
    { aFriend = myFriend; }
    public synchronized void hug()
    {
        try {
            System.out.println( "I, " + getName() + ", am hugged ");
            sleep(5);
            aFriend.rehug();
        } catch (InterruptedException e) {} 
    }
    public synchronized void rehug()
    {
        System.out.println( "I " + getName() + " am rehugged ");
    }
    public void run() { aFriend.hug(); }
}
public class NestedMonitor {
    public static void main( String args[] ) {
        Friendly paul = new Friendly("Paul");
        Friendly john = new Friendly("John");
        john.setFriend( paul );
        paul.setFriend( john );
        paul.start();
        john.start();
        System.out.println("main Ends");
    }
}

Output-->
Main Ends
I, John, am hugged
I, Paul, am hugged
Create a Shared Lock Object to Save Them

class Friendly extends Thread {
    private Friendly aFriend;
    private Object lock;
    public Friendly( String name, Object lock ) {
        super( name );
        this.lock = lock;
    } // same as before
    public void run() {
        synchronized (lock) { // sequentialize the calls
            aFriend.hug();
        }
    }
}
public static void main( String args[] ) {
    Object aLock = "saver";
    Friendly paul = new Friendly("John", aLock);
    Friendly john = new Friendly( "Paul", aLock); ...}
Coordination Among Threads

- Two kinds of synchronization: mutual exclusion and co-operation
- Mutual exclusion: synchronized lock
- Coordination
  - Waitset
  - Wait
  - Notify
- Bounded Buffer, Reader/Writer, Dining Philosophers, etc.
Wait-sets and Notification

- Every Java Object has a wait-set
  - Threads enter the wait-set by invoking `wait()`
  - `notify()` releases a single thread
  - `notifyAll()` releases all threads

- Wait-sets can only be manipulated while the object lock is held
  - `IllegalMonitorStateException` is thrown
  - `wait()` releases the lock while the thread waits
Waiting for a Condition

- **Generic form of a guarded condition:**

  ```java
  synchronized (foo) {
      while (!condition) { // if is not good enough
          try {
              foo.wait();
          } finally {
              catch (InterruptedException ex) {
                  // exception handling code
              }
          } // main body of code
  }
  ```
Waiting for a Condition (cont ...)

- `wait()` releases this object's lock
  - Including a lock held multiple times
- `wait()` *does not* release locks on other objects held by the thread
- Optional time-out: `wait(long millis)`
  - No direct indication that a time-out occurred
- The object's lock must be reacquired before `wait()` can return, no matter how
**Notification**

- **Generic form of a notification:**

  ```java
  synchronized(foo) {
    condition = true;
    foo.notify(); // or foo.notifyAll();
  }
  ```

- **Order in which `notify()` selects a thread from the wait-set is not defined**

- **Which method to use?**
  - Only use `notify()` if you can guarantee the right thread will always get selected!
The Producer/Consumer Problem

- *Classical synchronization problem*

**Bounded Buffer**

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

- **put(value)**
- **get()**

To make it simpler, assume the buffer size is 1 and only 1 producer/consumer pair.
class Buffer {
    private int data;
    private boolean available = false;
    public synchronized int get() {
        while (available == false) {
            try {
                wait();
            } catch (InterruptedException e) {
            }
        }
        available = false;
        notify();
        return data;
    }
    public synchronized void put(int value) {
        while (available == true) {
            try {
                wait();
            } catch (InterruptedException e) {
            }
        }
        data = value;
        available = true;
        notify();
    }
}
class **Producer** extends Thread {
    private Buffer buf;
    private int number;

    public Producer(**Buffer b**, int number) {
        buf = b;
        this.number = number;
    }

    public void **run**() {
        for (int i = 0; i < 10; i++) {
            buf.put(i);
            System.out.println("Producer #" + this.number + " put: i);
        }
    }
}
class Consumer extends Thread {
    private Buffer buf;
    private int number;
    public Consumer(Buffer b, int number) {
        buff = b;
        this.number = number;
    }
    public void run() {
        int value = 0;
        for (int i = 0; i < 10; i++) {
            value = buf.get();
            System.out.println
                ("Consumer "+ this.number + " got: "+value);
        }
    }
}

The Main Program
class ProducerConsumerTest {
    public static void main(String args[]) {
        Buffer b = new Buffer();
        Producer p1 = new Producer(b, 1);
        Consumer c1 = new Consumer(b, 1);

        p1.start();
        c1.start();
    }
}
Why notifyAll? Java Monitor

[以bounded buffer problem(buffer size = 1)为例]

class BufferJavaMonitor{
    private int count = 0;
    private Object buffer;
    public synchronized void put(Object x){
        while (count == 1) wait();
        buffer=x; count=1;
        notify();
    }
    public synchronized Object get() {
        Object x;
        while (count == 0) wait();
        count=0;
        notify(); // change to notifyAll();
        return x;
    }
}
Hoare Monitor: A Synchronization Mechanism

- Private data
- Condition variables and their queues
- Operations
- Initialization code
Condition Variables

- Condition variables allow an object inside a monitor to suspend itself based on some condition, and therefore allow other threads to enter the monitor.
- Java does not provide full scale condition variables. There is only a single queue for object locks and condition variables.
- Have to implement one's own Condition class.

- December issue of Java World online: http://www.javaworld.com/
- Or an exercise by Yu-Lung Chen
Monitor Class (to be subclassed)

class Monitor {
    private boolean lock = false;
    private int counter = 0;
    public boolean signalFlag = false;
    public synchronized void setLock() {
        if (lock) {
            counter++;
            wait();
        } else lock = true;
    }
    public synchronized void releaseLock() {
        if (!signalFlag) {
            if (counter != 0) {
                counter--;
                notify();
            } else lock = false;
        } else signalFlag = false;
    }
}
class Condition {
    private int condCounter=0;
    public synchronized void condWait(Monitor m) {
        condCounter++;
        m.releaseLock();
        wait();
    }

    public synchronized void condSignal(Monitor m) {
        if (condCounter > 0) {
            condWaiter--;
            notify();
            m.signalFlag = true;
        }
    }

    public synchronized boolean queue() {
        return (condCounter > 0) ? true : false;
    }
}
class **BoundedBufferMonitor** extends Monitor{
    private Object buffer[];
    private int count=0,lastPointer=0;
    *private Condition nonempty, nonfull;*
    public BoundedBufferMonitor() {
        buffer = new Object[N];
        nonempty = new Condition();
        nonfull = new Condition();
    }
    public void **append**(Object x) {
        *setLock();*
        if (count == N)
            nonfull.condWait(this);
        buffer[lastPointer] = x;
        lastPointer=(lastPointer+1)mod N;
        count++;
        nonempty.condSignal(this);
        *releaseLock();*}
public Object remove()
{
    Object x;
    setLock();
    if (count == 0)
        nonempty.condWait(this);
    x = buffer[lastPointer];
    lastPointer = (lastPointer - count) mod N;
    count--;
    nonfull.Signal(this);
    releaseLock();
    return x;
}
There are three forms of `wait` method:

```java
public final void wait ()
    throws InterruptedException;
public final void wait (long millisecTimeout)
    throws InterruptedException;
public final void wait (long millisecTimeout,
    int nanosecTimeout)
    throws InterruptedException;
```

Unfortunately, Java doesn't directly inform applications whether the wait call returned because the object was interrupted or because a timeout occurred.
Common Misconceptions

The following are all false:

- Java guarantees the highest priority thread will always run
- Threads of equal priority are scheduled FIFO
- Object locks are granted FIFO
- Threads in a wait-set are notified FIFO
- yield( ) only yields to threads of the same priority

Most early books documented the behavior of the Sun JVM rather than the behaviour guaranteed by the language specification.
Semantic Trade-offs

- Non-portable scheduling semantics
  - Notionally priority but not guaranteed
- Haphazard notification
  - There is no way to specify a particular thread or a group of threads that should be notified. API
- Priority inversion
  - Thread priority has no bearing on Java’s notification mechanism
Queuing

- Order in which threads are granted locks is not defined
  - Not necessarily priority based
  - No preference to threads released from a wait-set
- Order in which a notified thread is selected is not defined
  - Not necessarily priority based
  - Not necessarily FIFO
A state polling approach:

public class TT extends Thread {
    static final int RUN = 0;
    static final int SUSPEND = 1;
    static final int STOP = 2;
    private int state = RUN;

    public synchronized void setState(int s) {
        state = s;
        if (s == RUN) notify();
    }
}
private boolean checkState() {  
    while (state == SUSPEND) {  
        try { wait(); } 
        catch (InterruptedException e) {}  
    }  
    if (state == STOP) return false;  
    return true;  
}  
public void run() {  
    while (true) {  
        doSomething();  
        if (!checkState()) // state polling  
            break;  
    }  
}
Grouping of Threads

- Thread groups provide a mechanism for collecting multiple threads into a single object and manipulating those threads all at once, rather than individually.

- Java thread groups are implemented by the java.lang.ThreadGroup class.

- When a Java application first starts up, the Java runtime system creates a ThreadGroup named main.

- Unless specified otherwise, all new threads that you create become members of the main thread group.
Grouping of Threads (cont.)

To put a new thread in a thread group the group must be explicitly specified when the thread is created:

- public `Thread(ThreadGroup group, Runnable runnable)`
- public `Thread(ThreadGroup group, String name)`
- public `Thread(ThreadGroup group, Runnable runnable, String name)`

A thread can not be moved to a new group after the thread has been created.
A ThreadGroup may also contain other ThreadGroups allowing the creation of an hierarchy of threads and thread groups:

- public \texttt{ThreadGroup}(ThreadGroup parent, String name)

To get the thread group of a thread the \texttt{getThreadGroup} of the Thread class should be called:

\begin{verbatim}
ThreadGroup group = myThread.getThreadGroup();
\end{verbatim}
Thread Groups (cont.)

- system
  - garbage collector
  - clock

- MAIN
  - main
    - your group
      - Thd 1
      - Thd 2

= ThreadGroup

= Thread
ThreadGroup Method Categories

- **Collection management modules**--
  - methods that manage the collection of threads and subgroups contained in the thread group

- **Methods that operate on a group**--
  - these methods set or get attributes of ThrGrp

- **Methods that operate on all threads in a grp**--
  - performs action on all threads and subgroups

- **Access restriction methods**--
  - allow SecurityManager to restrict on mbrshp
The Thread API (cont.)

- Miscellaneous Thread Methods (cont.)
  
  ```java
  public static int enumerate(Thread tarray[]);
  ```

  Returns a list of all threads in the current thread’s ThreadGroup class (including its subgroups).

  ```java
  public static void dumpStack(); // for debugging
  ```

  Prints a method-by-method list of the stack trace for the current thread to the System.err output stream.

  ```java
  public String toString();
  ```

  Returns a string that describes this thread, including the thread's name, priority, and thread group.
Further Readings

◆ D. Lea, Concurrent Programming with Java, 2nd. Ed., Addison-Wesley
◆ A. Holub, Programming Java threads in the real world: part 1 ~ 9, JavaWorld
◆ D. Lea, Package java.util.concurrent
◆ D. Lea’s web site