References

The Java Programming Language, 2nd Ed. Arnold & Gosling, Addison-Wesley, 1998

Threads and Locks.

Java 1.6.0 on-line documentation http://java.sun.com/javase/6/docs/api/


Concurrent Programming in Java: Design Principles and Patterns, Doug Lea, Addison-Wesley, 1997

Concurrent Programming

Safety

Liveness

Nondeterminism

Communication
Processes verses Threads

Processes (Heavy Weight)
Child process gets a copy of parent’s variables
Relatively expensive to start
No concurrent access to variables

Thread (Light Weight Process)
Child process shares parents variables
Relatively cheap to start
Concurrent access to variables is an issue
Creating Threads by Inheritance

class ExtendingThreadExample extends Thread {
    public void run() {
        for ( int count = 0; count < 4; count++)
            System.out.println( "Message " + count + " From: Mom" );
    }

    public static void main( String[] args ) {
        ExtendingThreadExample parallel =
            new ExtendingThreadExample();
        System.out.println( "Create the thread" );
        parallel.start();
        System.out.println( "Started the thread " + parallel.getId() );
        System.out.println( "End" );
    }
}
class SecondMethod implements Runnable {
    public void run() {
        for (int count = 0; count < 4; count++)
            System.out.println("Message " + count + " From: Dad");
    }
}

public static void main(String[] args) {
    SecondMethod notAThread = new SecondMethod();
    Thread parallel = new Thread(notAThread);

    System.out.println("Create the thread");
    parallel.start();
    System.out.println("Started the thread");
    System.out.println("End");
}
public class WithNames implements Runnable {
    public void run() {
        for (int count = 0; count < 2; count++)
            System.out.println("Message "+count +
                " From: " +
                Thread.currentThread().getName());
    }

    public static void main(String[] args) {
        Thread a = new Thread(new WithNames(),
            "Mom");
        Thread b = new Thread(new WithNames(),
            "Dad");

        System.out.println("Create the thread");
        a.start();
        b.start();
        System.out.println("End");
    }
}
Ruby Threads

\[a = \text{Thread.new}\{\ 4.\text{times}\{\ k\ \rightarrow\ \text{puts}\ k\}\ \}\]
\[a\text{.join}\]

Output

0
1
2
3

x = 5
\[a = \text{Thread.new}(x)\ do\ |size|\]
size\text{.times}\{\ k\ \rightarrow\ \text{puts}\ k\}\end
\[a\text{.join}\]

Output

0
1
2
3
5
For Future Examples

```java
public class SimpleThread extends Thread {
    private int maxCount = 32;

    public SimpleThread( String name ) {
        super(name);
    }

    public SimpleThread( String name, int repetitions ) {
        super(name);
        maxCount = repetitions;
    }

    public SimpleThread( int repetitions ) {
        maxCount = repetitions;
    }

    public void run() {
        for (int count = 0; count < maxCount; count++) {
            System.out.println(count + " From: " + getName());
        }
    }
}
```
public class RunSimpleThread {
    public static void main( String[] args ) {
        SimpleThread first = new SimpleThread( 5 );
        SimpleThread second = new SimpleThread( 5 );
        first.start();
        second.start();
        System.out.println( "End" );
    }
}

Java on a Solaris machine with multiple processors can run threads on different processors
Thread Scheduling

Priorities

Time-slicing
Priorities

Each thread has a priority

If there are two or more active threads
    If one has higher priority than others
    The higher priority thread is run until it is done or not active

### Java Thread Priorities

<table>
<thead>
<tr>
<th>java.lang.Thread field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread.MAX_PRIORITY</td>
<td>10</td>
</tr>
<tr>
<td>Thread.NORM_PRIORITY</td>
<td>5</td>
</tr>
<tr>
<td>Thread.MIN_PRIORITY</td>
<td>0</td>
</tr>
</tbody>
</table>

### Ruby Thread Priorities

Any float between
- 2147483649
- 2147483648

May be machine dependent
public class PriorityExample {
    public static void main( String[] args ) {
        SimpleThread first = new SimpleThread( 5 );
        SimpleThread second = new SimpleThread( 5 );
        second.setPriority( 8 );
        first.start();
        second.start();
        System.out.println( "End" );
    }
}

On Single Processor
0 From: Thread-5
1 From: Thread-5
2 From: Thread-5
3 From: Thread-5
4 From: Thread-5
0 From: Thread-4
1 From: Thread-4
2 From: Thread-4
3 From: Thread-4
4 From: Thread-4
End
Threads Run Once

Can't restart a thread

```java
public class RunOnceExample extends Thread {
    public void run() {
        System.out.println( "I ran" );
    }

    public static void main( String args[] ) throws Exception {
        RunOnceExample onceOnly = new RunOnceExample();
        onceOnly.setPriority( 6 );
        onceOnly.start();
        System.out.println( "Try restart" );
        onceOnly.start(); /* Causes Exception */
        System.out.println( "The End" );
    }
}
```
Time-Slicing

A thread is run for a short time slice and suspended,
It resumes only when it gets its next "turn"

Threads of the same priority share turns

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

Java spec allows time-sliced or non-time-sliced threads

Ruby docs don't talk about this
Testing for Time-slicing

If time-sliced output will be mixed

```java
public class InfinityThread extends Thread {
    public void run() {
        while (true)
            System.out.println("From: "+getName());
    }

    public static void main(String[] args) {
        InfinityThread first = new InfinityThread();
        InfinityThread second = new InfinityThread();
        first.start();
        second.start();
    }
}
```

```ruby
a = Thread.new do
  10.times { |k| puts "a #{k}" }
end

b = Thread.new do
  10.times { |k| puts "b #{k}" }
end

a.join
b.join
```

If time-sliced output will be mixed
Java user & daemon Threads

Daemon thread
Expendable
When all user threads are done
  the program ends
  all daemon threads are stopped

User thread
Not expendable
Execute until
  Their run method ends or
  An exception propagates beyond the run method.
When a Java Program Ends

Runtime.exit(int) has been called and the security manager permits the exit operation to take place.

or

Only daemon threads are running
public class DaemonExample extends Thread {
    public static void main(String[] args) {
        DaemonExample shortLived = new DaemonExample();
        shortLived.setDaemon(true);
        shortLived.start();
        System.out.println("Bye");
    }

    public void run() {
        while (true) {
            System.out.println("From: " + getName());
            System.out.flush();
        }
    }
}

Output
From: Thread-0 (Repeated many times)
Bye
From: Thread-0 (Repeated some more, then the program ends)
Thread States

Executing

Only one thread per processor can be running at a time

Runnable

A thread is ready to run but is not currently running

Not Runnable

A thread that is suspended or waiting for a resource
Yield

Allow another thread of the same priority to run
Thread is still runnable

```java
public class YieldThread extends Thread {
    public void run() {
        for (int count = 0; count < 4; count++) {
            System.out.println(count + " From: " + getName());
            yield();
        }
    }
}

public static void main(String[] args) {
    YieldThread first = new YieldThread();
    YieldThread second = new YieldThread();
    first.setPriority(1);
    second.setPriority(1);
    first.start();
    second.start();
    System.out.println("End");
}
```

Output (Explain this)
0 From: Thread-0
0 From: Thread-1
1 From: Thread-0
1 From: Thread-1
2 From: Thread-0
2 From: Thread-1
3 From: Thread-0
3 From: Thread-1
Java sleep

Put calling thread in not-runnable state for specified milliseconds

```java
public class NiceThread extends Thread {
    public void run() {
        try {
            System.out.println("Thread started");
            sleep(5);
            System.out.println("From: "+getName());
            System.out.println("Clean up operations");
        }
        catch (InterruptedException interrupted) {
            System.out.println("In catch");
        }
    }
}

class Main after start
From: Thread-0
Clean up operations
```
Java sleep

Put **calling** thread in not-runnable state for specified milliseconds

```java
public class NiceThread extends Thread {
    public void run() {
        System.out.println( "Thread started");
        System.out.println( "From: " + getName() );
        System.out.println( "Clean up operations" );
    }

    public static void main( String args[] ) throws InterruptedException {
        NiceThread missManners = new NiceThread();
        missManners.start();
        missManners.sleep(50); //Who is sleeping
        System.out.println( "Main after start" );
    }
}
```

**Output**
Thread started
From: Thread-0
Clean up operations
Main after start
Java deprecated Thread methods

The following Thread methods are not thread safe

suspend
resume
stop
destroy
The following program does not end
The interrupt just sets the interrupt flag!

```java
public class NoInterruptThread extends Thread {
    public void run() {
        while (true) {
            System.out.println("From: " + getName());
        }
    }
}

public static void main(String args[]) throws InterruptedException{
    NoInterruptThread focused = new NoInterruptThread();
    focused.setPriority(2);
    focused.start();
    Thread.currentThread().sleep(5); // Let other thread run
    focused.interrupt();
    System.out.println("End of main");
}
```

**Output**

From: Thread-0 (repeated many times)
End of main
From: Thread-0 (repeated until program is killed)
Using Thread.interrupted

public class RepeatableNiceThread extends Thread {
    public void run() {
        while (true) {
            while (!Thread.interrupted()) {
                System.out.println("From: " + getName());
            
                System.out.println("Clean up operations");
            }
        }
    }
}

public static void main(String args[]) throws InterruptedException{
    RepeatableNiceThread missManners =
        new RepeatableNiceThread();
    missManners.setPriority(2);
    missManners.start();
    Thread.currentThread().sleep(5);
    missManners.interrupt();
}

Output

From: Thread-0
Clean up operations
From: Thread-0
From: Thread-0 (repeated)
public class NiceThread extends Thread {
    public void run() {
        try {
            System.out.println( "Thread started" );
            while ( !isInterrupted() ) {
                sleep( 5 );
                System.out.println( "From: " + getName() );
            }
            System.out.println( "Clean up operations" );
        } catch ( InterruptedException interrupted ) {
            System.out.println( "In catch" );
        }
    }
}

public static void main( String args[] ) {
    NiceThread missManners = new NiceThread( );
    missManners.setPriority( 6 );
    missManners.start();
    missManners.interrupt();
}

Output
Thread started
From: Thread-0
In catch
Java interrupt ()

Sent to a thread to interrupt it

If thread is blocked on a call to wait, join or sleep
   InterruptedException is thrown &
   The interrupted status flag is cleared

if the thread is blocked on I/O operation on an interruptible channel (NIO)
   ClosedByInterruptException is thrown
   The interrupted status flag is set

If the thread is blocked by a selector (NIO)
   Interrupt status is set
   The thread returns from the selector call as normal

If none of the other conditions hold then the thread’s interrupt status is set
Details

If thread is blocked on a call to wait, join or sleep
   InterruptedException is thrown &
   The interrupted status flag is cleared

if the thread is blocked on I/O operation on an interruptible channel (NIO)
   ClosedByInterruptException is thrown
   The interrupted status flag is set

If the thread is blocked by a selector (NIO)
   Interrupt status is set
   The thread returns from the selector call as normal

If none of the other conditions hold then the thread’s interrupt status is set
Interrupt and Pre JDK 1.4 NIO operations

If a thread is blocked on a read/write to a:
  Stream
  Reader/Writer
  Pre-JDK 1.4 style socket read/write

The interrupt does not interrupt the read/write operation!

The threads interrupt flag is set

Until the IO is complete the interrupt has no effect

This is one motivation for the NIO package
Safety - Mutual Access

What happens when one thread reads a value while another is modifying it?
Java Safety - Synchronize

A call to a synchronized method locks the object
    Object remains locked until synchronized method is done

Any other thread's call to any synchronized method on the same object
    will block until the object is unlocked
class SynchronizeExample {
    int[] data;

    public String toString() {
        return "array length " + data.length + " array values " + data[0];
    }

    public synchronized void initialize(int size, int startValue) {
        data = new int[size];
        for (int index = 0; index < size; index++)
            data[index] = (int) Math.sin(index * startValue);
    }

    public void unSafeSetValue(int newValue) {
        for (int index = 0; index < data.length; index++)
            data[index] = (int) Math.sin(index * newValue);
    }

    public synchronized void safeSetValue(int newValue) {
        for (int index = 0; index < data.length; index++)
            data[index] = (int) Math.sin(index * newValue);
    }
}
class SynchronizeExample {
    int[] data;

    public String toString() {
        return "array length " + data.length + " array values " + data[0];
    }

    public synchronized void initialize( int size, int startValue) {
        data = new int[size];
        for (int index = 0; index < size; index++)
            data[index] = (int)Math.sin(index * startValue);
    }

    public void unSafeSetValue( int newValue) {
        for (int index = 0; index < data.length; index++)
            data[index] = (int)Math.sin(index * newValue);
    }

    public synchronized void safeSetValue( int newValue) {
        for (int index = 0; index < data.length; index++)
            data[index] = (int)Math.sin(index * newValue);
    }
}

Locks class
Blocks other synchronized class methods
Synchronized Statements

synchronized ( expression ) {
    statements
}

expression must evaluate to an object

That object is locked

class LockTest {
    public synchronized void enter() {
        System.out.println( "In enter" );
    }
}

class LockTest {
    public void enter() {
        synchronized ( this ) {
            System.out.println( "In enter" );
        }
    }
}
public class LockExample extends Thread {
    private Lock myLock;

    public LockExample( Lock aLock ) {
        myLock = aLock;
    }

    public void run() {
        System.out.println( "Start run" );
        myLock.enter();
        System.out.println( "End run" );
    }

    public static void main( String args[] ) throws Exception {
        Lock aLock = new Lock();
        LockExample tester = new LockExample( aLock );

        synchronized ( aLock ) {
            System.out.println( "In Block" );
            tester.start();
            System.out.println( "Before sleep" );
            Thread.currentThread().sleep( 5000 );
            System.out.println( "End Block" );
        }
    }
}

class Lock {
    public synchronized void enter() {
        System.out.println( "In enter" );
    }
}

Output
In Block
Start run
Before sleep
End Block
In enter
End run  (why is this at the end?)
class Top {
    public void synchronized left() {
        // do stuff
    }

    public void synchronized right() {
        // do stuff
    }
}

class Bottom extends Top {
    public void left() {
        // not synchronized
    }

    public void right() {
        // do stuff not synchronized
        super.right(); // synchronized here
        // do stuff not synchronized
    }
}
**wait and notify**

public final void wait(timeout) throws InterruptedException

public final void wait(timeout, nanos) throws InterruptedException

public final void wait() throws InterruptedException

Causes a thread to wait until it is notified or the specified timeout expires.

Throws: IllegalMonitorStateException
If the current thread is not the owner of the Object's monitor.

Throws: InterruptedException
Another thread has interrupted this thread.

public final void notify()
public final void notifyAll()

Notifies threads waiting for a condition to change.
wait - How to use

The thread waiting for a condition should look like:

```java
synchronized void waitingMethod()
{
    while ( ! condition )
        wait();

    Now do what you need to do when condition is true
}
```

Everything is executed in a synchronized method

The test condition is in loop not in an if statement

The wait suspends the thread it atomically releases the lock on the object
synchronized void changeMethod()
{
    Change some value used in a condition test

    notify();
}

notify - How to Use
wait and notify Example

When can Consumer read from queue?
public class Producer extends Thread {
    BlockingQueue<String> factory;
    int workSpeed;

    public Producer( String name, BlockingQueue<String> output, int speed ) {
        setName(name);
        factory = output;
        workSpeed = speed;
    }

    public void run() {
        try {
            int product = 0;
            while (true) {
                System.out.println( getName() + " produced " + product);
                factory.add( getName() + String.valueOf( product) );
                product++;
                sleep( workSpeed);
            }
        } catch ( InterruptedException workedToDeath ) {
            return;
        }
    }
}
import java.util.concurrent. *;

class Consumer extends Thread {
    BlockingQueue<String> localMall;
    int sleepDuration;

    public Consumer( String name, BlockingQueue<String> input, int speed ) {
        setName(name);
        localMall = input;
        sleepDuration = speed;
    }

    public void run() {
        try {
            while (true) {
                System.out.println( getName() + " got " + localMall.take());
                sleep( sleepDuration );
            }
        }
        catch ( InterruptedException endOfCreditCard ) {
            return;
        }
    }
}
import java.util.concurrent.*;
public class ProducerConsumerExample {
    public static void main( String args[] ) throws Exception {
        BlockingQueue<String> wallmart = new ArrayBlockingQueue(100, true);
        Producer nike = new Producer( "Nike", wallmart, 500 );
        Producer honda = new Producer( "Honda", wallmart, 1200 );
        Consumer valleyGirl = new Consumer( "Sue", wallmart, 400);
        Consumer valleyBoy = new Consumer( "Bob", wallmart, 900);
        Consumer dink = new Consumer( "Sam", wallmart, 2200);
        nike.start();
        honda.start();
        valleyGirl.start();
        valleyBoy.start();
        dink.start();
    }
}

<table>
<thead>
<tr>
<th></th>
<th>Nike produced 0</th>
<th>Nike produced 2</th>
<th>Nike produced 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Honda produced 0</td>
<td>Sue got Nike2</td>
<td>Sue got Nike4</td>
<td></td>
</tr>
<tr>
<td>Sue got Nike0</td>
<td>Honda produced 1</td>
<td>Honda produced</td>
<td></td>
</tr>
<tr>
<td>Bob got Honda0</td>
<td>Bob got Honda1</td>
<td>Bob got Honda2</td>
<td></td>
</tr>
<tr>
<td>Nike produced 1</td>
<td>Nike produced 3</td>
<td>Nike produced 5</td>
<td></td>
</tr>
<tr>
<td>Sam got Nike1</td>
<td>Sue got Nike3</td>
<td>Sue got Nike5</td>
<td></td>
</tr>
</tbody>
</table>
Java Blocking Queues

ArrayBlockingQueue
DelayQueue
LinkedBlockingQueue
PriorityBlockingQueue
SynchronousQueue
import java.util.concurrent.*;

public class ThreadPoolExample extends Object
{
    public static void main(String[] args)
    {
        int corePoolSize = 2;
        int maximumPoolSize = 5;
        long keepAliveTime = 60 * 10;
        TimeUnit keepAliveUnit = TimeUnit.SECONDS;
        BlockingQueue<Runnable> surplusJobs = new LinkedBlockingQueue<Runnable>();
        ThreadPoolExecutor workers = new ThreadPoolExecutor(corePoolSize,
                                                      maximumPoolSize, keepAliveTime, keepAliveUnit, surplusJobs);

        for (int k = 0; k < 5; k++)
            workers.execute(new SimpleThread(k + 5));
    }
}