**Multi-Threading Notes**

Thread priorities are integers that specify the relative priority

of one thread to another. As an absolute value, a priority is meaningless; a higher-priority

thread doesn’t run any faster than a lower-priority thread if it is the only thread running.

Instead, a thread’s priority is used to decide when to switch from one running thread to the

next. This is called a context switch. The rules that determine when a context switch takes

place is simple:

• A thread can voluntarily relinquish control. This is done by explicitly yielding, sleeping,

or blocking on pending I/O. In this scenario, all other threads are examined, and

the highest-priority thread that is ready to run is given the CPU.

• A thread can be preempted by a higher-priority thread. In this case, a lower-priority thread

that does not yield the processor is simply preempted—no matter what it is doing—

by a higher-priority thread. Basically, as soon as a higher-priority thread wants to

run, it does. This is called preemptive multitasking.

When a Java program starts up, one thread begins running immediately. This is usually

called the main thread of your program, because it is the one that is executed when your

program begins. The main thread is important for two reasons:

• It is the thread from which other “child” threads will be spawned.

• Often, it must be the last thread to finish execution because it performs various

shutdown actions.

Although the main thread is created automatically when your program is started, it

can be controlled through a Thread object. To do so, you must obtain a reference to it by

calling the method currentThread( ), which is a public static member of Thread

**static Thread currentThread( )**

Two ways exist to determine whether a thread has finished. First, you can call **isAlive( )**

on the thread. This method is defined by Thread, and its general form is shown here:

**final boolean isAlive( )**

The **isAlive( )** method returns true if the thread upon which it is called is still running. It

returns false otherwise.

While **isAlive( )** is occasionally useful, the method that you will more commonly use to

wait for a thread to finish is called **join( )**, shown here:

final void **join( )** throws InterruptedException

This method waits until the thread on which it is called terminates. Its name comes from

the concept of the calling thread waiting until the specified thread joins it. Additional forms

of **join( )** allow you to specify a maximum amount of time that you want to wait for the

specified thread to terminate.

**Synchronization**

When two or more threads need access to a shared resource, they need some way to ensure

that the resource will be used by only one thread at a time. The process by which this is

achieved is called synchronization.

Synchronization can be achieved using 2 ways

**Using Synchronized Methods**

Synchronization is easy in Java, because all objects have their own implicit monitor associated

with them. To enter an object’s monitor, just call a method that has been modified with the

**synchronized** keyword. While a thread is inside a synchronized method, all other threads that

try to call it (or any other synchronized method) on the same instance have to wait. To exit

the monitor and relinquish control of the object to the next waiting thread, the owner of the

monitor simply returns from the synchronized method.

**The synchronized Statement**

**synchronized** block.

This is the general form of the **synchronized** statement:

synchronized(*objRef*) {

// statements to be synchronized

}

Here, *objRef* is a reference to the object being synchronized. A synchronized block ensures

that a call to a synchronized method that is a member of *objRef*’s class occurs only after the

current thread has successfully entered *objRef*’s monitor.

**Interthread Communication**

To avoid polling, Java includes an elegant interprocess communication mechanism via

the wait( ), notify( ), and notifyAll( ) methods. These methods are implemented as final

methods in Object, so all classes have them. All three methods can be called only from

within a synchronized context. Although conceptually advanced from a computer science

perspective, the rules for using these methods are actually quite simple:

• wait( ) tells the calling thread to give up the monitor and go to sleep until some

other thread enters the same monitor and calls notify( ) or notifyAll( ).

• notify( ) wakes up a thread that called wait( ) on the same object.

• notifyAll( ) wakes up all the threads that called wait( ) on the same object. One of

the threads will be granted access.

These methods are declared within Object, as shown here:

final void wait( ) throws InterruptedException

final void notify( )

final void notify All( )

Although wait( ) normally waits until notify( ) or

notifyAll( ) is called, there is a possibility that in very rare cases the waiting thread could be

awakened due to a spurious wakeup. In this case, a waiting thread resumes without notify( )

or notifyAll( ) having been called. (In essence, the thread resumes for no apparent reason.)

Because of this remote possibility, Oracle recommends that calls to wait( ) should take place

within a loop that checks the condition on which the thread is waiting.