




## 4. Thread Scheduling


- Thread Priority
- The Thread Scheduler
- Thread Starvation
- Time Slicing

SKILLBUILDERS

Java Training

4.2

## Thread Priority...

 Thread priority

- A ranking that determines when thread gets CPU time
- Value is an integer from 1 (lowest) to 10 (highest)
- Thread class has an `int priority` property
  - `int getPriority( )`
  - `void setPriority( int value )`
- Static constants in the Thread class:
  - `MIN_PRIORITY (1)`
  - `NORM_PRIORITY (5)`
  - `MAX_PRIORITY (10)`
- Note: Some O/S's may not support all 10 levels!
  - For example, on Win 95 levels 3 & 4 are identical

Java Training

© 2003-2006 SkillBuilders, Inc.


Each thread has a priority setting that affects when it receives processor time. The priority value can be any integer value from 1 to 10, where the higher the number, the higher the priority.

The Java Thread class encapsulates this value in an `int priority` property, represented by a standard pair of methods:

```
public int getPriority( )
public void setPriority( int value )
```

For convenience the Thread class provides these static constants (although you can enter any literal integer in the acceptable range of values):

- `MIN_PRIORITY (1)`
- `NORM_PRIORITY (5)`
- `MAX_PRIORITY (10)`

4.3

## ...Thread Priority

- Rules for initial priority value:
  - A new thread has same priority as its creator.
  - Applications start with priority 5.
  - Applets start with priority 4.
- Value can be changed at any time
  - Value only matters while thread is in *Runnable* state
- Acceptable values:
  - Applications: 1 to 10
  - Applets: 1 to 6 only (for security)

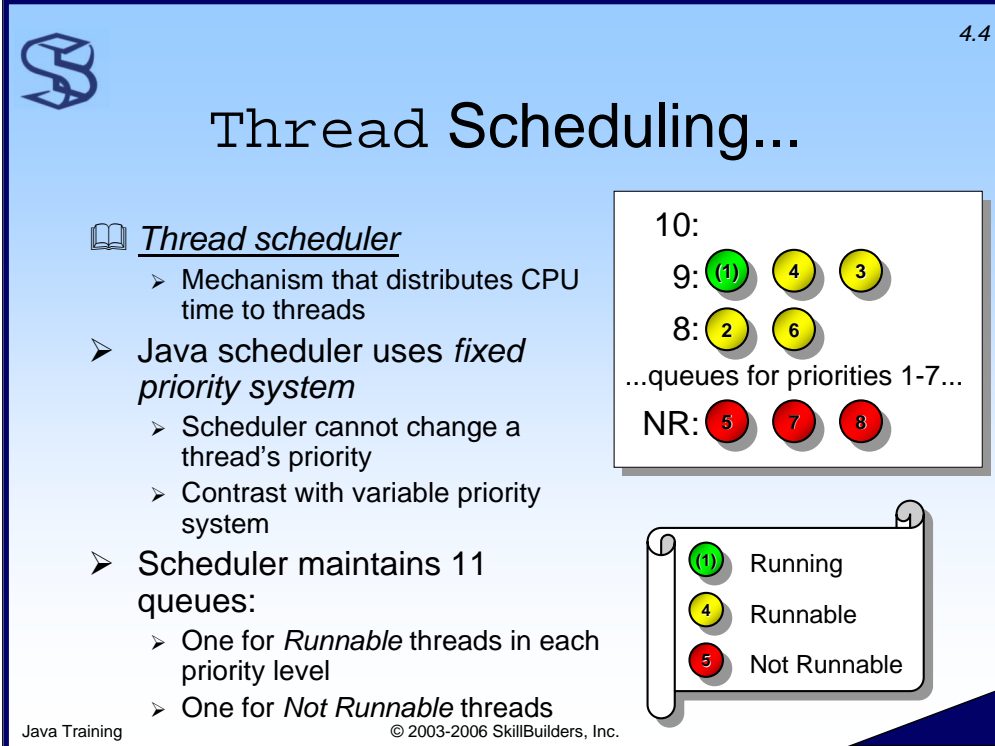
Java Training

© 2003-2006 SkillBuilders, Inc.

What priority does a new thread have (before you change it)? It depends on where and when it is created. The basic rule is this: *A new thread has same priority as the thread that created it.* As for top-level threads, the main line of a console application (the `main( )` method) has priority 5; the main line of an applet starts with priority 4.

You are free to change the priority of a thread at any time during its life (although obviously this is pointless once the thread dies). However the only state in which a thread's priority is evaluated is the *Runnable* state.

Generally, the value of the priority setting can be any value from 1 to 10. This is true of all threads in console applications. However applets operate under a restriction: the applet's main line and all threads it creates cannot have a priority higher than 6. This restriction protects against a malicious or careless applet shutting out system threads.



The diagram illustrates the Java thread scheduling mechanism. It features a title 'Thread Scheduling...' with a Java logo. A list of bullet points describes the thread scheduler and the fixed priority system. To the right, a diagram shows 11 queues for priorities 1-7, with threads 1-7 in the Running state and threads 8-10 in the Not Runnable state. A legend identifies the states: Running (green), Runnable (yellow), and Not Runnable (red).

**Thread scheduler**

- Mechanism that distributes CPU time to threads
- Java scheduler uses *fixed priority system*
  - Scheduler cannot change a thread's priority
  - Contrast with variable priority system
- Scheduler maintains 11 queues:
  - One for *Runnable* threads in each priority level
  - One for *Not Runnable* threads

10: 9: (1) 4 3  
8: 2 6  
...queues for priorities 1-7...  
NR: 5 7 8

(1) Running  
4 Runnable  
5 Not Runnable

Java Training © 2003-2006 SkillBuilders, Inc.


The Java *thread scheduler* is in charge of allocating a single CPU among the multiple threads of a Java application. In general, we only need to be concerned with the scheduling when the application has one or more CPU-intensive threads. In this case, it helps to know what is going on behind the scenes in the Java virtual machine so that we can optimize the execution of our application and avoid some potentially nasty situations.

The scheduler uses a *fixed priority* system. That means it cannot change the priority of a thread (which would change when and how much CPU time it gets). Contrast this to a *variable priority* system, where the scheduler can dynamically adjust the priority of processes in order to avoid starvation.

The scheduler maintains 11 thread queues:

- One queue for each of the ten priority levels of *Runnable* threads.
- One queue for threads in the *Not Runnable* state.

Any given queue may contain a number of threads, waiting their turn to be run. When a thread leaves the queue or moves to the end, all the other threads move forward.

4.5

## ...Thread Scheduling

- How the Java scheduler decides who runs:
  1. Highest priority *Runnable* thread at head of its queue runs
  2. If a higher priority thread becomes *Runnable*, scheduler preempts running thread
  3. A preempted thread goes to end of its queue
  4. If running thread leaves *Runnable* state, scheduler chooses again using rule 1
- Any thread moves to end of its queue when:
  - It goes from *Not Runnable* to *Runnable*
  - It is *Runnable* and its priority changes
- A running thread moves to end of its queue when:
  - The scheduler preempts it
  - It yields its place

Java Training

© 2003-2006 SkillBuilders, Inc.

### The Scheduler's Logic


Here is how the scheduler chooses which thread to run:

1. The highest priority *Runnable* thread at the head of its queue runs. If at a given moment the highest priority *Runnable* threads have priority 9, the first thread in that queue runs.
2. If a higher priority thread becomes *Runnable*, scheduler preempts running thread. If, while the priority 9 thread is running, a priority 10 thread becomes *Runnable*, it preempts the previous running thread.
3. A preempted thread goes to end of its queue. The preempted 9 thread goes to the end of its queue.
4. If the running thread leaves the *Runnable* state (sleeps or blocks), the scheduler chooses again using rule 1.

### Moving to the End of the Queue

When a thread goes from *Not Runnable* to *Runnable*, it moves to the end of its priority queue. Also, if a thread is *Runnable* and its priority changes, it moves to the end of its new priority queue.

A running thread moves to the end of its queue when the scheduler preempts it or it yields its place.

 Thread Scheduling... 4.6

## A Scheduling Example...

- A GUI console application has these threads:
  - I** ➤ Reads intermittent input from a socket (priority 9)
    - Infinite execution
    - Random blocks
  - C** ➤ Displays a status bar clock (priority 8)
    - Infinite execution
    - Sleeps 1 second per cycle
  - S** ➤ Recalculates a spreadsheet (priority 8)
    - Finite execution
    - No blocks

🕒 Time 1:  
**I** and **C** blocked,  
**S** calculating:

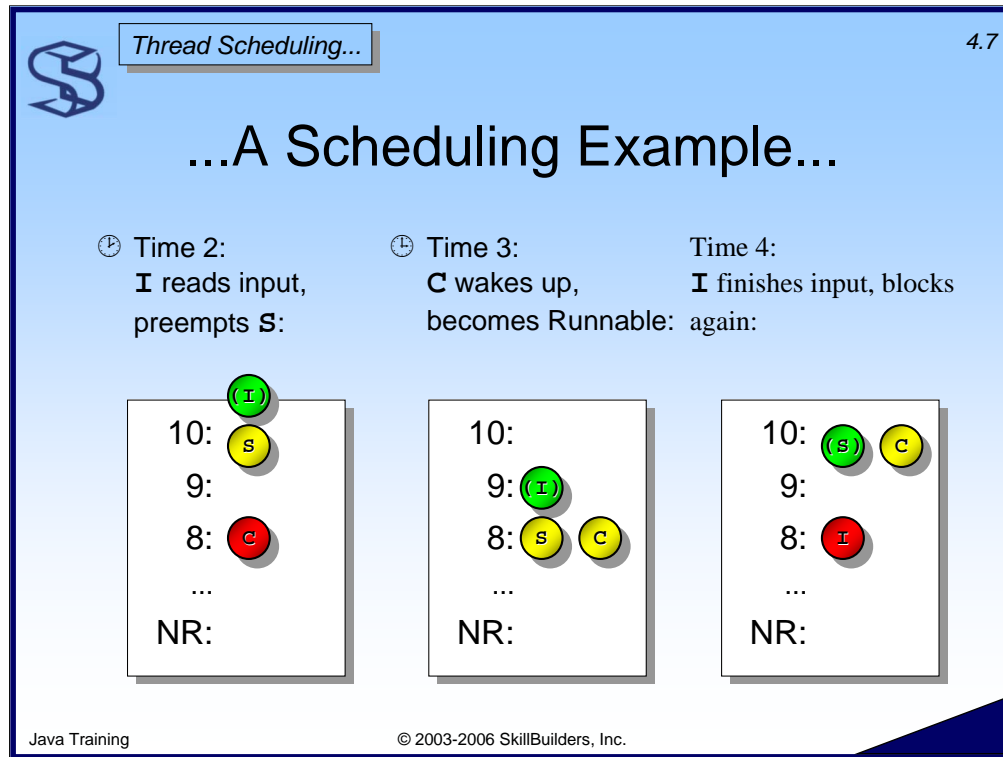
10:  
9:  
8: **(S)**  
...  
NR: **I** **C**

Java Training © 2003-2006 SkillBuilders, Inc.

To illustrate how thread scheduling works, let's look at an example involving a console application. This application has three threads in particular:

- Thread **I** reads intermittent input from a socket. It executes infinitely, with random blocks while it waits for input from the socket. It has priority 9.
- Thread **C** (priority 8) displays a status bar clock. It also executes infinitely, but sleeps for 1 second per iteration of its loop. It has priority 8.
- Thread **S** (priority 8) recalculates a spreadsheet. It executes a finite line of code with no blocks. It also has priority 8.

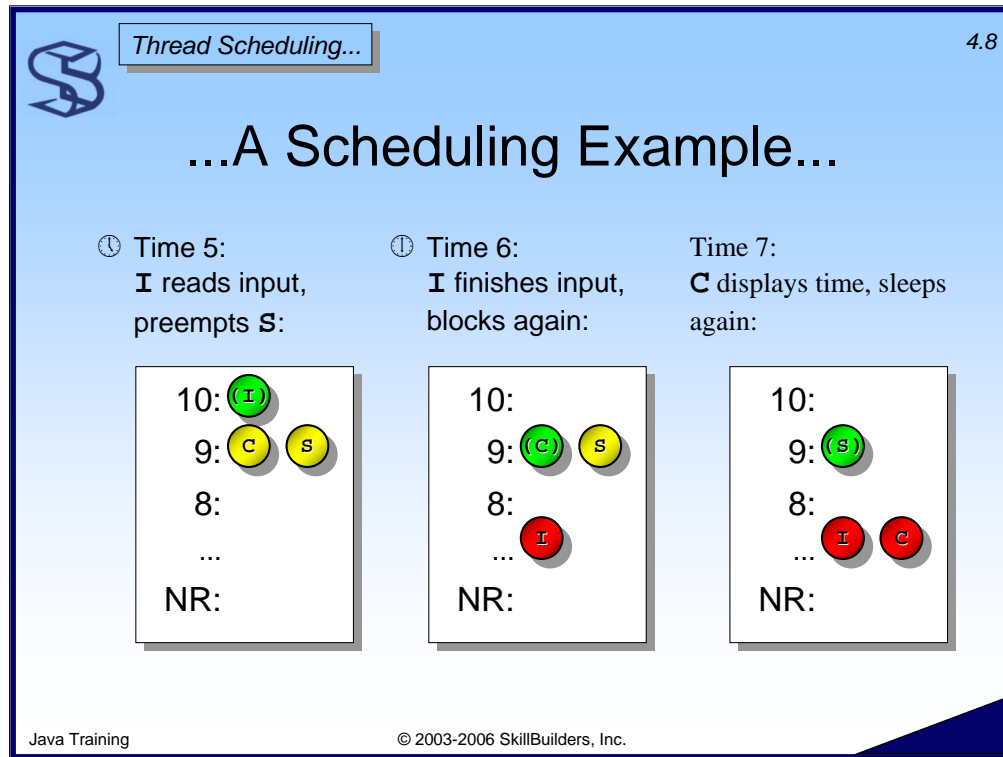
At Time 1, **S** is the highest-priority *Runnable* thread, so it is running. While **S** is calculating the spreadsheet, **I** and **C** are in the *Not Runnable* queue: **I** is blocked waiting for input; **C** is sleeping for 1 second.



At Time 2, **I** receives input and goes to the *Runnable* state. Since it has a higher priority than **S**, it preempts the running thread, which then goes to the back of its queue. As the only thread in its queue, **S** is back at the head of the line.

At Time 3, **C** wakes up from its sleep and becomes *Runnable*. It moves to the end of the priority 8 queue, behind **S**. But since **I** has higher priority, it continues to run.

At Time 4: **I** finishes reading input and blocks again, waiting for the next chunk of data from the socket. It moves to the *Not Runnable* queue. **S** is now once again the first of the highest-priority threads, so it begins running.

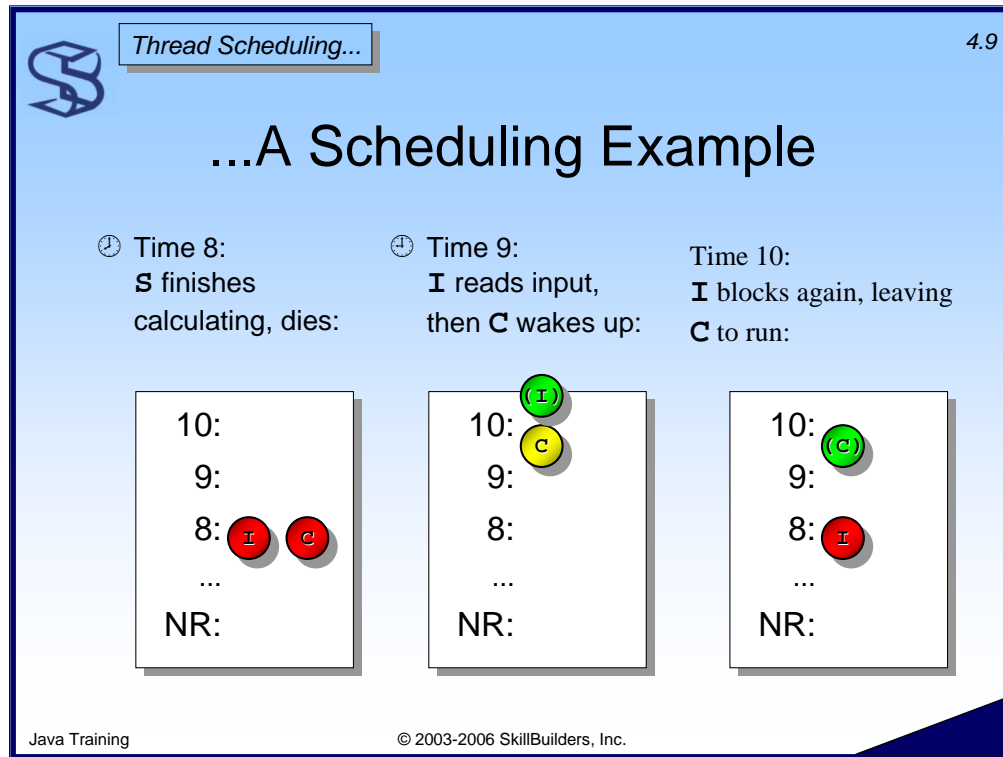


At Time 5: input again comes from the socket monitored by **I**. The thread moves from *Not Runnable* back to the *Runnable* state and preempts **S**. The latter moves to the end of its queue.

At Time 6: **I** finishes reading input and blocks again. It moves back to the *Not Runnable* queue, allowing a priority 8 thread to take control. Now, since the clock thread **C** is at the head of that queue, it gets CPU attention.

At Time 7: **C** has finished displaying the current time and sleeps again. As it moves to the *Not Runnable* queue, **S** moves to the head of its queue and becomes the current thread.







At Time 8: **S** finishes calculating the spreadsheet. Its `run( )` method terminates and the thread dies, leaving the queues altogether. Since the other two threads are still *Not Runnable*, neither of them runs. A lower-priority thread (perhaps garbage collection) takes over at this point.

At Time 9: **I** again reads input and becomes *Runnable*. It immediately becomes the current thread. A moment later **C** wakes up and moves into the priority 8 queue, waiting its turn.

At Time 10: **I** finishes reading input and blocks again, leaving **C** to execute.

4.10

# Thread Starvation

 Thread starvation

- When a thread receives no CPU time
- Caused by threads that are CPU-intensive (do not block)
- Two variations:
  1. Variation 1:
    - A single CPU-intensive thread has highest priority
    - Starves all lower-priority threads
  2. Variation 2:
    - Several CPU-intensive threads with equal priority
    - Not taking turns fairly

➤ Let's look at each in turn...

Java Training


© 2003-2006 SkillBuilders, Inc.

If a high-priority thread runs continuously without relinquishing control of the CPU (by either sleeping or yielding), then equal and lower priority threads might never have a chance to execute. They will receive little or no processor time because other threads are constantly shutting them out. This is commonly known as *thread starvation*.

We can envision two variations on this problem. In the first variation, a single CPU-intensive thread has highest priority and starves all lower-priority threads.

In the second variation, several CPU-intensive threads have equal priority. Although they may all get a turn at the processor, it will not happen in a fair distribution.

On the following pages we look at each variation in more detail.



Thread Starvation...

4.11

## Starvation 1

➤ Problem:

- 1 doesn't block; others do
- 1 starves all other *Runnable* threads

➤ Solution: lower 1's priority

- It runs when no higher-priority threads are *Runnable*

10:

9: (1)

8: (2) (3) (4)

...

NR: (5) (6) (7) (8)

10:

9:

8: (2) (3) (4)

...

3: (1)

NR: (5) (6) (7) (8)


Java Training

© 2003-2006 SkillBuilders, Inc.

The first variation of *thread starvation* involves a single high-priority thread shutting out all lower-priority threads from the processor.

In the example shown above, thread **T1** is the culprit. It performs some PU-intensive task (such as calculating prime numbers) without pausing. The other threads in the system spend most of their time in the *Not Runnable* state. So **T1** hogs the processor, never giving threads **T2**, **T3** and **T4** (not to mention all the other threads with lower priority) a chance. These latter threads are starved.

The simple solution to this problem is to lower **T1's** priority considerably. As shown above, it is lowered to 3. It runs when no higher-priority threads are *Runnable*; because the other threads are *Not Runnable* most of the time, **T1** still gets its chance.

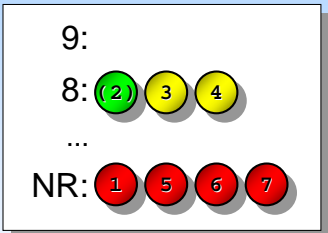


Thread Starvation...

4.12

## Starvation 2

- Problem: **T2, T3** and **T4** do not block
  - Running thread gives up CPU only when preempted by higher-priority thread
  - Time allocation depends on unrelated threads
- Solution: Have each thread yield periodically
  - Use `yield( )` method
  - Causes current thread to yield, move to end of queue
  - Thread remains *Runnable*
- See notes for another example...



```
public static void yield( )
```

Java Training

© 2003-2006 SkillBuilders, Inc.

The second variation of *thread starvation* involves several CPU-intensive threads with equal priority. The problem is not necessarily that one hogs the processor, but that the distribution will not be fair.

In the example shown above, **T2, T3** and **T4** are all CPU-intensive. **T2** starts out as the current thread and remains so until preempted by a higher-priority thread. When that happens, **T2** moves to the back of its queue. When the higher-priority thread becomes *Not Runnable*, **T3** gets to run. If a higher-priority again becomes thread *Runnable*, the rotation continues.

The problem is that time allocation among these three threads depends on unrelated threads preempting the current thread and moving it to the back of its queue.

Lowering the priority of these threads will improve the situation, since more threads will have higher priority than these three, so more frequent rotation will occur. But they still rely on the kindness of strangers.

Another solution is to have each thread periodically yield its hold on the CPU. One does this with the `yield( )` method of the `Thread` class:

```
public static void yield( )
```

This method causes the current thread to yield CPU control. The thread remains *Runnable* but moves to the end of its queue.

Let's look at a variation of our Counter application from earlier. Here is the Counter class again, extending Thread and counting from 1 to some maximum value:

```
// Counter class
class Counter extends Thread {
    private int iMax = 0;

    public Counter( String name, int max ) {
        super( name );
        iMax = max;
    }

    public void run() {
        for( int i = 1; i <= iMax; i++ )
            System.out.println( getName() + ": " + i );
        System.out.println( getName() + " is done" );
    }
}
```

Here is our application. It creates 3 Counter instances, lowers their priority to 4 (so main( ) is not starved), and starts them all.

```
class CounterApp {
    public static void main( String[] args ) {
        Counter[] counters = new Counter[3];
        for( int i = 0; i < counters.length; i++ ) {
            counters[i] = new Counter( "Counter " + (i + 1), 6 );
            counters[i].setPriority(4);
            counters[i].start();
        }
        System.out.println( "main() is done" );
    }
}
```

Continued...


Below in the left column is the output we might see. Lowering the priority of the Counter threads allowed main( ) to finish first, but the counters don't take turns. Counter 1 still starves 2 and 3.

By modifying the Counter class slightly, we can change the outcome dramatically:


```
// Counter class
class Counter extends Thread {
    // Instance vars and c'tors...
    public void run() {
        for( int i = 1; i <= iMax; i++ ) {
            System.out.println( getName() + ": " + i );
            Thread.yield();
        }
        System.out.println( getName() + " is done" );
    }
}
```

And the result is in the right column below:

Without yield( )	With yield( )
main() is done	main() is done
Counter 1: 1	Counter 1: 1
Counter 1: 2	Counter 2: 1
Counter 1: 3	Counter 3: 1
Counter 1: 4	Counter 1: 2
Counter 1: 5	Counter 2: 2
Counter 1: 6	Counter 3: 2
Counter 1 is done	Counter 1: 3
Counter 2: 1	Counter 2: 3
Counter 2: 2	Counter 3: 3
Counter 2: 3	Counter 1: 4
Counter 2: 4	Counter 2: 4
Counter 2: 5	Counter 3: 4
Counter 2: 6	Counter 1: 5
Counter 2 is done	Counter 2: 5
Counter 3: 1	Counter 3: 5
Counter 3: 2	Counter 1: 6
Counter 3: 3	Counter 2: 6
Counter 3: 4	Counter 3: 6
Counter 3: 5	Counter 1 is done
Counter 3: 6	Counter 2 is done
Counter 3 is done	Counter 3 is done

4.15

## Time Slicing

 Time slicing

- Scheduler gives running thread a maximum time
- Effectively an automatic yield
- Insures a fairer distribution of CPU time
- Reduces chance of starvation
- Not all schedulers perform time slicing
  - The VM spec does not mention it
  - MS Windows implementations do so
  - Sun Solaris implementation does not
- Tip: Do not rely on time-slicing!
  - It makes your program less portable

Java Training

© 2003-2006 SkillBuilders, Inc.

*Time slicing* is a variation on the scheduling rules in which the scheduler gives the running thread a maximum running time (usually measured in nanoseconds). When the running thread uses up its allotted time, the scheduler bumps it to the end of its queue and reevaluates which thread is next. In effect, time slicing is like an automatic yield built into every thread. It generally insures a fairer distribution of CPU time among competing threads and reduces the chance of a thread starving, because a CPU intensive thread must periodically yield.

Since the Java Virtual Machine specification neither requires nor prohibits time slicing, the implementation of threads largely depends on the threading mechanism of the underlying operating system that the virtual machine is running on. So not all schedulers perform it. Microsoft Windows implementations of the VM support time slicing, so Java threads running on Windows will be allocated a certain time-slice and will be preempted when that period of time expires. On the other hand, the Solaris version of Java uses a simple threading package that does no time slicing at all. Therefore, threads running under the Solaris version of Java will not be interrupted preemptively. Under Solaris, threads must be preempted by the Java application.

The difference in the underlying threading mechanism introduces some system-dependent behavior into Java. Therefore, in order to keep your Java code as portable as possible, ***do not rely on time-slicing!***



4.16

## Where We've Been

- Threads have a priority setting
- The Java thread scheduler allocates CPU time
  - 11 queues are used
- Thread starvation is a serious problem
  - Lower the priority of CPU-intensive threads
  - Use `yield( )` to distribute CPU time fairly
- Time slicing can change the scheduling outcome



Java Training

© 2003-2006 SkillBuilders, Inc.





## Your Objective

In this workshop you will test the effects of setting thread priority and yielding on the behavior of threads.

This workshop uses a non-GUI console application that displays a counter to the screen, along the lines of the first examples we saw in the course.

### A. Examine the Application

The application has been started for you; it resides in `CounterApp.java` in the directory indicated by the instructor. The application consists of two classes:

- `Counter` encapsulates `name` and `max` properties. Its `run( )` method iterates from 1 to `max`, displaying a message to standard output on each pass.
- `CounterApp` has a `main( )` method that instantiates 3 counters and runs them.

*Continues...*

➤ **Processing:**

1. Copy the file to your working directory.
2. Compile and run the application. Examine the output. (If necessary, redirect it to a file.) Do the counters execute sequentially? When does `main( )` finish?

**B. Convert Counters to Threads**

1. Modify the `Counter` class to extend `Thread`. Remove the name property and use that of the `Thread` class instead.
2. Modify `CounterApp` to start the threads instead of calling their `run( )` method.
3. Compile and run the application. Do you see any difference in the sequence of output?

**C. Have the Counters Yield**

Modify the counters so that they optionally will yield control.

1. Give the `Counter` class a boolean field to indicate yielding. Add a third constructor argument to assign a value to the field.
2. In the `run( )` method, inside the loop, have the counter yield if the value is `true`.
3. In `CounterApp`, have the presence of at least two command-line arguments (i.e. `args.length >= 2`) turn on yielding for all the threads.
4. Recompile. Run the application again, passing in two command-line arguments to trigger yielding. Any change in the sequence of output?

*Continues...*

## D. Change Thread Priorities

1. Modify the `main( )` method to display the priority of the main thread. Use the `Thread.currentThread( )` method to get the `Thread` reference.
2. Compile and run the application. What is the main thread's priority?
3. Now modify `main( )` to give the three counters different priorities before they are started if a command-line argument is passed in. Use the first command-line argument as the base priority value, converting the `String` to an `int` like so:

```
int intBasePriority = -1;
if( args.length > 0 )
    intBasePriority = Integer.parseInt( args[0] );
```

4. Compile and run. Try several possibilities:
  - Give all three threads a higher priority than `main( )`.
  - Give all three threads a lower priority than `main( )`.
  - Give a combination of the two.
5. Do the threads behave as you expected?