Threads

In this section, you will learn:

- What a thread is
- How to spawn a thread
- How to synchronize access of shared data or critical resources
- How to create thread-safe code
- How threads communicate

Introduction

The ability of a program to concurrently execute multiple regions of code provides capabilities that are difficult or impossible to achieve with strictly sequential languages. Sequential object-oriented languages send messages (make method calls) and then block or wait for the operation to complete.

Programmers are already familiar with concurrent processes, possibly without recognizing them. For example, operating systems usually have many processes running to handle printing, updating the display, receiving mail from the network, and so on. In contrast, most programming languages do not promote the use of concurrent operations within one application. At best, programmers have access to a few library calls to launch and control multiple operations.

Java provides language-level and library support for threads--independent sequences of execution within the same program that share the same code and data address space. Each thread has its own stack to make method calls and store local variables.

Most applications that use threads are a form of simulation or have a graphical user interface, but threads in general have the following advantages over sequential programming:

- Threads support concurrent operations. For example,
    - Server applications can handle multiple clients by launching a thread to deal with each client.
    - Long computations or high-latency disk and network operations can be

handled in the background without disturbing foreground computations or screen updates.

- Threads often result in simpler programs.
    - In sequential programming, updating multiple displays normally requires a big while-loop that performs small parts of each display update. Unfortunately, this loop basically simulates an operating system scheduler. In Java, each view can be assigned a thread to provide continuous updates.
    - Programs that need to respond to user-initiated events can set up service routines to handle the events without having to insert code in the main routine to look for these events.
- Threads provide a high degree of control.
    - Imagine launching a complex computation that occasionally takes longer than is satisfactory. A "watchdog" thread can be activated that will "kill" the computation if it becomes costly, perhaps in favor of an alternate, approximate solution. Note that sequential programs must muddy the computation with termination code, whereas, a Java program can use thread control to non-intrusively supervise any operation.
- Threaded applications exploit parallelism.
    - A computer with multiple CPUs can literally execute multiple threads on different functional units without having to simulating multi-tasking ("time sharing").
    - On some computers, one CPU handles the display while another handles computations or database accesses, thus, providing extremely fast user interface response times.

Spawning a Thread

To spawn a Java thread, follow these steps:

1. Create your class, making it implement the `Runnable` interface.

```
class Animation implements Runnable {
  ...
}
```

2. Define a method within your class with this signature:

```
public void run() {
  // when this exits, thread dies
}
```

This method will be called when the thread starts up.

Elsewhere, you may:

3. Create an instance of your class.

```
Animation a = new Animation();
```

4. Create a new thread attached to your new instance.

```
Thread t = new Thread(a);
```

5. Start the thread attached to your object.

```
t.start();
```

As a simple example, consider the following runnable class that continuously prints a string to standard out.

```
class Jabber implements Runnable {

  String str;

  public Jabber(String s) { str = s; }
  public void run() {
    while (true) {
      System.out.println(str);
    }
  }
}
```

Class Jabber can be used in the following way:

```
class JabberTest {

  public static void main(String[] args) {
    Jabber j = new Jabber("MageLang");
    Jabber k = new Jabber("Institute");
    Thread t = new Thread(j);
    Thread u = new Thread(k);
    t.start();
    u.start();
  }
}
```

The output from running `JabberTest` would be the two threads intermixing, not alternating, output lines of *MageLang* and *Institute*.

Controlling Thread Execution

Thread execution may be controlled after creation with methods of Thread such as:

- `join`
  Wait for a thread to die (because it was stopped or its `run()` method terminated). For example, the following code launches a thread and prints out a message when it has completed its task:

  ```
  Computation c = new Computation(34);
  Thread t = new Thread(c);
  t.start();
  t.join();
  System.out.println("done");
  ```
- `yield`
  Allow another thread to begin execution. This is an important method for machines with cooperative scheduling of threads at the same priority level such as Macintoshes.
- `sleep(int n)`
  Put a thread to sleep for n milliseconds without losing control of any locks it holds on any objects.
- `interrupt`
  Sends signal to thread to interrupt execution.
- `suspend`
  Suspend execution of this thread (until `resume()` is called). Only threads that are alive can be suspended. Applet threads are typically suspended when the page containing that thread is exited using the applet `stop()` method.
- `resume`
  Resume a stopped thread. If the thread is alive, it is place back on the list of threads eligible for CPU time; resuming a thread does not necessarily mean that it begins execution right away. Applet threads are typically suspended when the page containing that thread is exited using the applet `stop()` method and resumed in the `start()` method.
- `stop`
  Force the thread to stop executing. Stopping a thread that has not yet started is ok as it will stop immediately when it tries to start. A stopped thread may not be restarted with method `start()`.

  Note: The use of `suspend`, `resume`, and `stop` is strongly discouraged because it may leave the Java environment in an unstable state. Their usage is so discouraged that starting with the 1.2 JDK, the methods are flagged as deprecated, indicating that there will be a time when they will be dropped. For additional information, see Sun's Why JavaSoft is Deprecating Thread.stop, Thread.suspend and Thread.resume (http://java.sun.com/products/jdk/1.2/docs/guide/misc/threadPrimitiveDeprecation.html) resource.

To illustrate the `stop()` method and to provide a useful mechanism, consider the following WatchDog class that launches a thread, but stops it if the thread lives too long (deadlocked perhaps?).

```
class WatchDog {
  public WatchDog(Runnable r, int ms) {
    Thread t = new Thread(r);
    t.start();
    try {Thread.sleep(ms);}
    catch (InterruptedException e) {}
    t.stop();
  }
}
```

The following main program shows how to use the watchdog:

```
public class KillThread {
  public static void main(String[] args) {
    Analysis a = new Analysis();
    WatchDog w = new WatchDog(a, 1000);
  }
}
```

Here is the simple runnable class that will be killed.

```
class Analysis implements Runnable {
  public void run() {
    while ( true ) {
      System.out.println("analyze...");
    }
  }
}
```

Without the watchdog, a thread would run forever running on this object.

Of course, since `stop()` is becoming deprecated, an alternative needs to be used. The following demonstrates the same concept, without using the `stop()` method. The key difference is that without `stop` the thread can ignore your request to terminate. All you can do is request that it terminates, usually through a change in a status flag. When the thread decides to cooperate (check the status flag and react to the change), it should relinquish all resources and then stop.

```
public class KillThread {
  public static void main(String[] args) {
    Analysis a = new Analysis();
    WatchDog w = new WatchDog(a, a, 1000);
  }
}
class Analysis implements Runnable, ThreadStopper {
  private boolean stopFlag = false;
  Thread thisThread = Thread.currentThread();
  public void run() {
    while (!stopFlag) {
       System.out.println("analyze...");
    }
  }
  public void stopIt() {
    stopFlag = true;
  }
}
interface ThreadStopper {
  public void stopIt();
}
class WatchDog {
  public WatchDog(Runnable r, ThreadStopper stopper, int ms) {
    Thread t = new Thread(r);
    t.start();
    try {Thread.sleep(ms);}
    catch (InterruptedException e) {}
    stopper.stopIt();
  }
}
```

Multithreading in Applets

Working with threads in applets requires a little extra care. Normally, you do not want threads to continue running after the user has left the web page. To ensure this happens properly, careful use of the `start()` and `stop()` methods will both ensure threads stop and resources are not locked when the webpage is not visible. The following demonstrates a reusable pattern for multithreading within applets.

```
public class ThreadApplet
    extends java.applet.Applet
    implements Runnable {
  private Thread runner;
  private int sleepInterval = 500; // milliseconds
  public void start() {
    runner = new Thread (this);
    runner.start();
  }
  public void stop() {
    // runner.stop() // Old/unsafe way
    runner = null;
  }
  public void run() {
    Thread thisThread = Thread.currentThread();
   // instead of while (true)
    while (runner == thisThread) {
      try {
        thisThread.sleep (sleepInterval);
        // Do something
        System.out.println ("Do something");
      } catch (InterruptedException e){
        e.printStackTrace();
      }
    }
  }
}
```

Thread Safety

Threads are an extremely useful programming tool, but can render programs difficult to follow or even nonfunctional. Consider that threads share the same data address space and, hence, can interfere with each other by interrupting critical sections (so-called "atomic operations" that must execute to completion). For example, consider that a write operation to a database must not be interrupted by a read operation because the read operation would find the database in an incorrect state. Or, perhaps more graphically, consider the analogy of two trains that need to share the same resource (run on the same stretch of track). Without synchronization, a disaster would surely occur the next time both trains arrived at nearly the same time.

The state of a running program is essentially the value of all data at a given instant. The history of a program's execution then is the sequence of states entered by a program. For single-threaded application there is exactly one history, but multi-threaded applications have many possible histories-- some of which may contain a number of incorrect states due to interference of threads.

The prevention of interference is called mutual exclusion and results primarily from synchronization, which constrains all possible histories to only those containing exclusively good states. Condition synchronization refers to threads waiting for a condition to become true before continuing; in other words, waiting for the state to become valid or correct before proceeding. In Java language terms, synchronized methods or code blocks have exclusive access to the member variables of an object resulting in atomic code blocks.

A safe program is one that has been synchronized to prevent interference, but at the same time avoids deadlock, the discussion of which is deferred to a future section.

In this section, we dive into more detail concerning synchronization and condition synchronization, but begin with a short background discussion of the thread synchronization model used by Java.

Monitors

In the mid-1960's, E. Dijkstra invented the notion of a semaphore for implementing mutual exclusion and signaling events and defined P and V operations to specify critical sections. Unfortunately, semaphores are very low-level elements and programs written with them are hard to read as both condition synchronization and mutual exclusion are specified with the same mechanism. In the early 1970's, Tony Hoare defined monitors, which provided a structured approach to exclusion and condition synchronization. Monitors were eventually included in Concurrent Pascal, Modula, and Mesa (at Xerox PARC) [And91 (#thAND91)]. A number of the PARC researchers that worked on the Cedar/Mesa project now work at JavaSoft. Naturally, the Java thread synchronization mechanism, which is monitor-based, evolved from Cedar/Mesa.

A monitor is a chunk of data that can only be accessed through a set of routines. This type of encapsulation is expressed as an object in today's terminology, although monitors were designed like modules rather than instances of a class. A monitor's access routines are guaranteed to execute in a mutually exclusive manner. Java relaxes this constraint slightly to allow a class' methods to be explicitly specified as synchronized, which always execute to completion.

Monitors use condition variables plus wait and signal statements to provide condition synchronization. An accessor routine waits on a condition variable until awakened by another thread executing a signal statement on that variable. Java has a simpler, more efficient, condition synchronization scheme. A thread executes a wait() in a method of some object and blocks until awakened. A call to notifyAll() awakens all threads waiting on a signal for that object--there is no way to specify that only certain threads are to be awakened (a call to notify() wakes up a single waiting thread).

One can say that a monitor access routine acquires a lock on that monitor, which it releases upon returning from that method. In this way, mutual exclusion is implicitly achieved. Only synchronized Java methods acquire a lock on an object. A call to wait() releases the lock, but will reacquire it as it is awakened by a notifyAll(). notifyAll() does not yield execution nor release its hold on an object's lock. The only way to release a lock is by waiting or returning from a synchronized method.

As a final detail, we note that some operations are considered atomic and, hence, do not require synchronization of any kind. In Java, only assignments to primitives except long and double are considered atomic.

Mutual Exclusion

Mutual exclusion is the simple idea that certain code blocks cannot be interrupted once begun; i.e., they execute to completion. Java allows instance methods, class methods, and blocks of code to be synchronized. All may be considered to be synchronized using the lock of some object. For example, consider the following two synchronized instance methods.

```
class DataBase {
  public synchronized void
                write(Object o, String key) {...}
  public synchronized void read(String key){...}
  public String getVersion() {...}
}
```

Java ensures that once a thread enters either method, another thread may not interrupt it to execute the other method. Notice that method `getVersion()` is not synchronized and, therefore, may be called by any thread even if another thread has the lock on the same object. Synchronized methods may call other synchronized methods (even those of other objects) and may call non-synchronized methods.

Non-method code blocks may be synchronized via an object also. For example, let's rewrite the DataBase class to be nonsynchronized:

```
class DataBase {
   ...
   public void write(Object o, String key) {...}
   public void read(String key){...}
   public String getVersion() {...}
}
```

Code may still access a database in a safe manner by locking the call site rather than the method definition. Assume the following definition:

```
DataBase db = new DataBase();
```

If one thread executes:

```
synchronized (db) {
   db.write(new Employee(), "Jim");
}
```

another thread may execute the following safely:

```
synchronized (db) {
   db.read("Jim");
}
```

Once a write operation has been started, a read cannot proceed and once a read has been started, a write may not proceed.

Class methods may also be synchronized even though they do not operate on an object, however, each object has a reference to its class definition object (of type `Class`). Class methods gain the lock of their class definition object for mutual exclusion. Thinking of class method locks in this manner allows us to generalize, saying that all methods and all code blocks synchronize on a per-object basis.

As an example, consider how you might restrict access to a system resource such as a printer, by using per-class synchronization. You might define the following class:

```
class HPLaser {
   private static Device dev = ...;
   static synchronized print(String s) {...}
}
```

Then, if a thread is executing this:

```
HPLaser.print("Help! I'm trapped in this PC!");
```

another thread may not execute this:

```
HPLaser.print("We're not in Kansas anymore");
```

Condition Synchronization

Condition synchronization delays the execution of a thread until a condition is satisfied whereas mutual exclusion delays the execution of a thread until it can acquire a lock. Normally, this condition signifies the completion of another task or that the object state is now valid again and it is safe to proceed.

Conditional delays would be easy to specify, but inefficient to implement as:

```
await(condition) statement;
```

Java chooses a simpler mechanism where the programmer must loop according to the condition around a wait:

```
while ( !condition ) do wait();
```

We use a while-loop instead of an if-statement because there is no way to restrict a `notifyAll()` to a particular condition. This thread may wake up even though a different condition has changed. Also, after waking, a thread still may find the condition unsatisfied because another thread had awakened ahead of it.

To awaken a waiting thread, have another thread call `notifyAll()`. For example, consider the simple problem of reading information from a blocking queue where you want read operations to block waiting for information.

```
class BlockingQueue {
   int n = 0;
   ...
   public synchronized Object read() {
      // wait until there is something to read
      while (n==0) wait();
      // we have the lock and state we're seeking
      n--;
      ...
    }
    public synchronized void write(Object o) {
      ...
      n++;
      // tell threads to wake up--one more object
      notifyAll();
    }
}
```

Notice that only one read can occur simultaneously because `read()` is synchronized.

Because the read operation is destructive (removes an element), it is proper that only one simultaneous read occurs. Returning to the database read/write problem from above, we note that reads do not have side effects and there is no theoretical reason to restrict access to a single simultaneous read. The "readers and writers" problem is well known and is solved by having an object control access to the database so that the read/write methods do not have to be synchronized.

Thread Liveness

The notion that your program will not lock up and eventually do something useful is called the liveness property. In sequential programming, this property is equivalent to termination. In a multi-threaded environment, this property implies that your program will stop responding or processing before a desired result is achieved.

There are a number of reasons why a multi-threaded program will fail liveness:

- **Deadlock.** Two threads lock, waiting for the other to perform some task. Deadlock is by far the biggest liveness issue.
- **Unsatisfied wait conditions.** A thread is waiting on a condition (via `wait()`), but another thread cannot or does not satisfy the condition and notify the waiting thread.
- **Suspended thread is not resumed.** A thread is suspended, but another thread cannot or does not resume that thread.
- **Starvation.** A thread with higher priority preempts your thread, never allowing it any CPU time. In Java, the thread with the highest priority is running, implying that any thread at a lower priority is starved unless the higher priority thread blocks or waits. Another case of starvation is platform dependent. On some machines such as the Macintosh, threads at the same priority level are cooperatively scheduled; i.e., you must yield, sleep, wait, or call some method that does in order for a task switch to occur. On most systems, however, threads at the same level are preemptively scheduled by the operating system; e.g., compute bound tasks cannot starve other threads at the same priority.
- **Premature death.** A thread was killed via `stop()` before it was appropriate to terminate.

Thread safety often conflicts with the goal of liveness. To produce thread-safe code, programmers are tempted to simply put `synchronized` on every publicly visible method. Unfortunately, a poorly out thread-safety scheme could induce deadlock.

Deadlock

Deadlock can occur even for simple problems. For example, consider a class that computes prime numbers in a given range--a multi-processor computer could then run a thread on each object to compute primes in parallel. Each object knows of the other prime computation peer objects, hence, any object can return a composite list of primes from the peers. For simplicity, assume there are only two such prime number computation objects and that both have references to each other in order to report the primes:

```
class ComputePrime {
  private int[] primes;
  private int lower, upper;
  private ComputePrime peer;
  public ComputePrime(int low, int high) {
    lower = low;
    upper = high;
    primes = new int[high-low+1];
  }

  public void setPeer(ComputePrime p) {
    peer = p; // who am I associated with?
  }

  public synchronized int[] getPrimes() {
    return primes;
  }

  public synchronized String getAllPrimes() {
    int[] peerPrimes = peer.getPrimes();
    // return merge of peerPrimes with primes
  }
}
```

To see the potential for deadlock, assume that we have the following two peer clients:

```
ComputePrime c1 = new ComputePrime(2,100);
ComputePrime c2 = new ComputePrime(101,200);
c1.setPeer(c2);
c2.setPeer(c1);
```

If one thread executes `c1.getAllPrimes()`, but is interrupted by another thread that executes `c2.getAllPrimes()`, deadlock will occur. The following sequence highlights the problem:

1. Thread 1 executes

   ```
   c1.getAllPrimes();
   ```

   thus, locking c1.

2. Thread 2 executes

   ```
   c2.getAllPrimes()
   ```

   locking c2.

3. Thread 1 calls

   ```
   c2.getPrimes()
   ```

but blocks waiting for thread 2 to release the lock on c2.

4. Thread 2 calls

```
c1.getPrimes()
```

but blocks waiting for the thread 1 to release the lock on c1.

Thread Priorities

Every thread runs at a particular priority level. Altering thread priorities allows you to fine tune your program's behavior to better utilize system resources. Whenever a higher priority thread is runnable, the priority-based scheduler within Java favors the running of the higher priority thread. So, if a particular task takes longer to run, and it is important that it does run, then you should probably raise its priority so that it is scheduled to run more frequently. The methods related to priorities are `getPriority()` and `setPriority()`, where they work with a range of `int` priority values from `MIN_PRIORITY` to `MAX_PRIORITY`, with a default value of `NORM_PRIORITY`.

Thread Groups

Related threads can be grouped together in what is called a `ThreadGroup`. Each and every thread is in exactly one group. The advantage of working with thread groups is you can signal them all collectively with methods like `resume`, `stop`, `suspend`, and `interrupt`. Also, you can limit the maximum priority of the thread group, via `setMaxPriority` to ensure a particular group of threads do not consume too much CPU time. (Keep in mind that `resume`, `stop`, and `suspend` are becoming deprecated in the 1.2 JDK, so the class' purpose seems more limited to just plain grouping.) And, there are security-related restrictions within thread groups. A thread within a group can only affect other threads within the group or subgroups.

The following program demonstrates the usage of `ThreadGroup` by traversing the complete thread tree.

```
public class PrintGroupInfo {
  public static void main (String args[]) {
    ThreadGroup group = new ThreadGroup
      ("The Thread Group");
    group.setMaxPriority (3);
    for (int i = 0; i < 5; i++)
```

```
      new Thread (group,
        "TheThread-" + i).setPriority(i+1);
    PrintGroupInfo.startPrinting();
  }

  public static synchronized void startPrinting() {
    ThreadGroup root =
      Thread.currentThread().getThreadGroup();
    ThreadGroup parent = root;
    while ((parent = parent.getParent()) != null)
      root = parent;
    printGroupInfo ("", root);
  }

  public static void printGroupInfo
      (String indent, ThreadGroup group) {
    if (group == null) return;
    System.out.println (indent +
      "Group: " + group.getName() +
      "-> MaxPriority: " + group.getMaxPriority());
    int numThreads = group.activeCount();
    Thread threads[]  = new Thread[numThreads];
    numThreads = group.enumerate (threads, false);
    for (int i = 0; i < numThreads; i++)
      printThreadInfo(indent + "   ", threads[i]);
    int numGroups  = group.activeGroupCount();
    ThreadGroup groups[] = new ThreadGroup[numGroups];
    numGroups = group.enumerate (groups, false);
    for (int i = 0; i < numGroups; i++)
      printGroupInfo (indent + "   ", groups[i]);
  }

  public static void printThreadInfo
      (String indent, Thread thread) {
    if (thread == null) return;
    System.out.println (indent +
      "Thread: " + thread.getName() +
      "-> Priority: " + thread.getPriority() +
      (thread.isAlive() ?
        " <Alive>" : " <NotAlive>") +
      ((Thread.currentThread() == thread) ?
        " <-- current" : ""));
  }
}
```

Sample output follows:

```
Group: system-> MaxPriority: 10
   Thread: Finalizer thread-> Priority: 1 <Alive>
   Group: main-> MaxPriority: 10
      Thread: main-> Priority: 5 <Alive> <-- current
      Group: The Thread Group-> MaxPriority: 3
         Thread: TheThread-0-> Priority: 1 <NotAlive>
         Thread: TheThread-1-> Priority: 2 <NotAlive>
```

```
Thread: TheThread-2-> Priority: 3 <NotAlive>
Thread: TheThread-3-> Priority: 3 <NotAlive>
Thread: TheThread-4-> Priority: 3 <NotAlive>
```

Ensuring Safety and Liveness

The following techniques can be used to promote safety and liveness (in order of increasing probability of success):

- **Testing.** Test the program in many different situations or with lots of different input trying to catch incorrect behavior and deadlock.
- **Case analysis or "What would happen in this situation?"** Analyzing all possible program states is impossible, but trying to examine worst case scenarios often highlights potential problems. For example, "What happens when I interrupt a write operation to do a read operation in this object?"
- **Design patterns.** Concurrent programming problems are fit into known patterns whose behaviors are well understood. This has the advantage that can reuse code or algorithms that are known to work, but your programming problem may not fit any one pattern exactly.
- **Formal proofs.** Using predicates and axioms to describe program state and state transformations, proofs are constructed to establish correctness.

For the average programmer, the common approach will be to use case analysis to guide algorithm design and use known patterns where possible, thus, trying to build in thread safety and liveness. Testing can then be used to (hopefully) catch unforeseen problems and to gain confidence that you have correctly implemented a task.

The class library supplied by Sun is thread safe. Unfortunately, this does not guarantee that the code you write using the library is thread safe as deadlock is easy to achieve if you are not careful.

In this section, we provide a few tips for building in thread safety.

Restricted Universes

One of the most common techniques for providing thread safety while promoting efficiency and without endangering liveness is to create restricted universes within which you know exactly which threads are executing.

Consider a recursive-descent parser for the Java language. The parser object will have a series of mutually-recursive methods, one for every rule in the grammar;

e.g.,

```
class JavaParser {
   public synchronized void compilationUnit() {}
   public synchronized void expression() {}
   public synchronized void statement() {}
   public synchronized void method() {}
   public synchronized void declaration() {}
   ...
}
```

The methods are all synchronized because you can begin parsing at any one of the rule-methods--having multiple threads walking the same parser object would cause serious interference with input pointers and so on.

While synchronizing all the methods provides safety, it unnecessarily slows down the overall parser because synchronized methods are slower than unsynchronized methods. The solution is to restrict the universe in which the parser object can exist, allowing the parser object to assume that at most one thread can ever run on it. A containment object (sometimes called a facade, sentinel, or broker) can provide a synchronized interface to the parser, which would then not need to be synchronized.

If you needed to access the main starting symbol `compilationUnit()` and also rule-method `expression()`, the following containment object would provide thread safe access without slowing down the parser.

```
class JavaParserSentinel {
  JavaParser parser;
  public JavaParserSentinel(JavaParser p) {
    parser = p;
  }

  public synchronized void compilationUnit() {
    parser.compilationUnit();
  }

  public synchronized void expression() {
    parser.expression() {
  }
}
```

"Red Flags" And Hints

There are a number of common situations in which thread safety or liveness is in jeopardy. Here are a few:

- Any time a reference cycle (object x refers to y and y refers to x) exists in a multithreaded application, interference can occur. Synchronizing the access methods can induce deadlock, however. Generally, an arbitrator is required to prevent deadlock.

- Be very careful with the conditions you use for while-wait condition synchronization. If another thread does not satisfy that condition or if it is itself blocked, the waiting thread is effectively dead.
- Any time multiple threads compete for limited resources, the possibility of deadlock occurs.
- Threads managed by the AWT such as the repaint or event thread should not be used to perform large parts of your application or applet. For example, having the thread that executes `Applet.init()` block on some I/O generally prevents the window from being refreshed. Event handling code should merely set conditions or launch other threads to perform large pieces of work.

Recall that from our Java parser example, one can be sure that objects contained within and solely accessed by thread safe code do not need to be synchronized.

As a final suggestion, remember that if one thread doesn't read the member variables written to by another thread, and vice-versa, no interference can occur. Further, local variables cannot be accessed from a different thread and, hence, are by definition disjoint.

Advanced Magercises

For those looking for a little something extra, or for those who want to see a detailed example of thread use in an application, these Advanced Magercies are also available.

Further Reading and References

Students will find *Concurrent Programming in Java* useful. (Written by Doug Lea published in the Java Series from Addison-Wesley, Menlo Park, CA, 1996).

Some of the fundamental concepts summarized here are explored in great detail in the following textbook:

[And91] (null) Concurrent Programming: Principles and Practice by Gregory Andrews, Addison-Wesley, Menlo Park, CA, 1991.

[*MML: 0.995a*]
[*Version: $Id: //depot/main/src/edu/modules/Threads/threads.mml#3 $*]