UNIVERSITY OF CALIFORNIA
Department of Electrical Engineering
and Computer Sciences
Computer Science Division

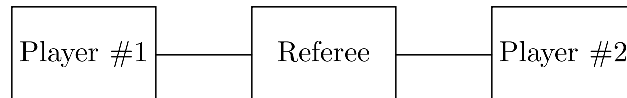
CS61B
Fall 1999

P. N. Hilfinger

Threads*

1 Multiple Threads of Control

Consider the problem of designing a multi-person game, in which two players (either or both of which might actually be a program) play against each other, with something keeping track of the play and acting as a referee. This description of the problem suggests a set of objects something like this:



I intend here that the two players use the Referee to make moves and to find out what the opponent has moved. One way to implement this sort of arrangement is to write the program (in outline) as shown in Figure 1. Player #1 and Player #2 represent modules of some kind that you (or you and some opposing team) write that either compute moves or read moves that humans make. Both of these modules, in fact, could be in the same program.

In a real game, the players can do things while waiting for the opponent to move – they can think, for example. In a program, if the "players" didn't actually think, but rather served as intermediaries to read moves from the real, human players and to display results, you could imagine that a human player might want to have the `PlayerClass` object he was talking to do various things while waiting for the opponent, such as display the moves up to this point, or display a different view of the board. Unfortunately, the design given above doesn't make that possible. The functions of each `PlayerClass` object are active only when the referee decides that it's that player's turn to move.

These considerations suggest the alternative structure shown in Figure 2, in which each `PlayerClass` instance module calls the referee both to make moves and to find out results. But that's a structure we haven't seen before. If Player #1 will not leave his **while** loop until the game is over, so how does Player #2 get a chance to do anything?

We want, in effect, to have Player #1 and Player #2 behave like two separate programs, each performing a loop in which it makes a move and then waits for the opponent to move. In

*Copyright © 1998, 1999 by Paul N. Hilfinger. All rights reserved.

```
class Board { A class representing the current game board. }

class Move { A class representing possible moves. }

class RefereeClass {
   void PlayGame (PlayerClass player1, PlayerClass player2) {
     Move m;
     while (true) {
        m = player1.move (this);
        // Check m and record it.
        if (gameOver ())
          break;
        m = player2.move (this);
        // Check m and record it.
        if (gameOver ())
          break;
     }
   }

   boolean gameOver () { ... }

   Board currentBoard () { ... }
   ...
}

class PlayerClass {
   Move move (RefereeClass r) {
     Board situation = r.currentBoard ();
     // Compute and return move based on situation.
   }
   ...
}
```

Figure 1: General outline of a simple two-player game. This version doesn't allow for thinking while the opponent moves.

```
class RefereeClass {

   void makeMove (PlayerClass mover, Move m) {
     Update current state of the board.
     Tell mover's opponent that it's now his turn
   }

   boolean gameOver () { ... }

   Board currentBoard () { ... }
   ...
}

class PlayerClass {
   void play (RefereeClass r) {
     Board situation;
     situation = r.currentBoard ();
     while (! r.gameOver ()) {
        Wait for human player or referee to do something
        if (player makes move m)
           r.makeMove (this, m);
        else if (referee reports that opponent has moved)
           situation = r.currentBoard ();
        else
           check for other requests from human player
     }
   }
   ...
}
```

Figure 2: Redesign of the game structure in Figure 1 to allow one player to continue working while the other moves.

Java, such a miniprogram-within-a-program is called a *thread*. The term is short for "thread of control." Java provides a means of creating threads and indicating what code they are to execute, and a means for threads to communicate data to each other without causing the sort of chaos that would result from multiple programs attempting to make modifications to the same variables simultaneously. The use of more than one thread in a program is often called *multi-threading;* however, we also use the terms *concurrent programming,* and *parallel programming.*

Java programs may be executed on machines with multiple processors, in which case two active threads may actually execute instructions simultaneously. However, the language makes no guarantees about this. That's a good thing, since it would be a rather hard effect to create on a machine with only one processor (as most have). The Java language specification also countenances implementations of Java that

- Allow one thread to execute until it wants to stop, and only then choose another to execute, or
- Allow one thread to execute for a certain period of time and then give another thread a turn (*time-slicing*), or
- Repeatedly execute one instruction from each thread in turn (*interleaving*).

This ambiguity suggests that threads are not intended simply as a way to get parallelism. In fact, the thread is a useful *program structuring tool* that allows sequences of related instructions to be written together as a single, coherent unit, even when they can't actually be executed that way.

2 Creating and Starting Threads

Java provides the built-in type `Thread` (in package `java.lang`) to allow a program a way to talk about and control threads. There are essentially two ways to use it to create a new thread. Suppose that our main program discovers that it needs to have a certain task carried out – let's say washing the dishes – while it continues with its work – let's say playing chess. We can use either of the following two forms to bring this about:

```
class Dishwasher implements Runnable {
    Fields needed for dishwashing.
    public void run () { wash the dishes }
}
```

```
class MainProgram {
   public static void main (String[] args) {
      ...
      if (dishes are dirty) {
        Dishwasher washInfo = new Dishwasher ();
        Thread washer = new Thread (washInfo);
        washer.start ();
      }
      play chess
   }
}
```

Here, we first create a data object (`washInfo`) to contain all data needed for dishwashing. It implements the standard interface `java.lang.Runnable`, which requires defining a function `run` that describes a computation to be performed. We then create a `Thread` object `washer` to give us a handle by which to control a new thread of control, and tell it to use `washInfo` to tell it what this thread should execute. Calling `.start` on this `Thread` starts the thread that it represents, and causes it to call `washInfo.run ()`. The main program now continues normally to play chess, while thread `washer` does the dishes. When the `run` method called by `washer` returns, the thread is terminated and (in this case) simply vanishes quietly.

In this example, I created two variables with which to name the `Thread` and the data object from which the thread gets its marching orders. Actually, neither variable is necessary; I could have written instead

```
if (dishes are dirty)
   (new Thread (new Dishwasher ())).start ();
```

Java provides another way to express the same thing. Which you use depends on taste and circumstance. Here's the alternative formulation:

```
class Dishwasher extends Thread {
   Fields needed for dishwashing.
   public void run () { wash the dishes }
   }
}

class MainProgram {
   public static void main (String[] args) {
      ...
      if (dishes are dirty) {
        Thread washer = new Dishwasher ();
        washer.start ();  // or just (new Dishwasher ()).start ();
      }
      play chess
   }
}
```

Here, the data needed for dishwashing are folded into the `Thread` object (actually, an extension of `Thread`) that represents the dishwashing thread.

When you execute a Java application (as opposed to an embedded Java program, such as an applet), the system creates an anonymous *main thread* that simply calls the appropriate `main` procedure. When the `main` procedure terminates, this main thread terminates and the system then typically waits for any other threads you have created to terminate as well.

2.1 Applying threads to games

To handle our game-playing example from above, we might use the following structure:

```
class RefereeClass {
   ...
}

class PlayerClass extends Thread {
   RefereeClass r;

   void PlayerClass (RefereeClass theRef) {
      r = theRef;
      start ();     // start is inherited from Thread.
   }

   public void run () {
      as for play, above.
   }
}

class Driver {
   public static void main (String[] args) {
      RefereeClass ref = new RefereeClass ();
      PlayerClass player1 = new PlayerClass (ref),
                  player2 = new PlayerClass (ref);
   }
}
```

Here, the main program simply creates a referee and two players. The constructors for the players start up threads to run them. The main program then exits, and players carry on all the work. The referee here is just an object. It has no thread of its own, but relies on the players' threads to execute all its functions.

2.2 A question of terminology

The fact that Java's `Thread` class has the name it does may tempt you into making the equation "thread = `Thread`." Unfortunately, this is not correct. When I write 'thread'

(uncapitalized in normal text font), I mean 'thread of control' – an independently executing instruction sequence. When I write '`Thread`' (capitalized in typewriter font), I mean 'a Java object of type `java.lang.Thread`.' The relationship is that a `Thread` is an object visible to a program by which one can manipulate a thread – a sort of handle on a thread. Each `Thread` object is associated with a thread, which itself has no name, and which becomes active when some other thread executes the `start` method of the `Thread` object.

A `Thread` object is otherwise just an ordinary Java object. The thread associated with it has no special access to the associated `Thread` object. Consider the `PlayerClass` and `Driver` classes above. `PlayerClass` is an extension of `Thread`. Initially, the anonymous main thread executes the code in `Driver`. The main thread creates the `Threads` (actually `PlayerClass`) `player1`, and then executes the statements in the constructor for `player1`. That's right: the main thread executes the code in the `PlayerClass` constructor, just as it would for any object. While executing this constructor, the main thread executes `start()`, which is short for `this.start()`, with `this` being `player1`. That activates the thread (lower case) associated with `player1`, and that thread executes the `run` method for `player1`. If we were to define additional procedures in `PlayerClass`, they could be executed either by the main thread or by either of the threads associated with `player1` and `player2`. The only magic about the type `Thread` is that among other things, it allows you to start up a new thread; otherwise `Threads` are ordinary objects.

Just to cement these distinctions in your mind, I suggest that you examine the following (rather perverse) program fragment, and make sure that you understand why its effect is to print '`true`' a number of times. It makes use of the static (class) method `Thread.currentThread`, which returns the `Thread` associated with the thread that calls it. Since `Foo` extends `Thread`, this method is inherited by `Foo`.

```
// Why does the following program print nothing but 'true'?
class Foo extends Thread {
   public void run () {
      System.out.println (Main.mainThread != currentThread ());
   }

   public void printStuff () {
      System.out.println (Main.mainThread == Thread.currentThread ());
      System.out.println (Main.mainThread == Foo.currentThread ());
      System.out.println (Main.mainThread == this.currentThread ());
      System.out.println (Main.mainThread == currentThread ());
   }
}
```

```
class Main {
   static Thread mainThread;
   public static void main (String[] args) {
      mainThread = Thread.currentThread ();
      Thread aFoo = new Foo ();
      aFoo.start ();
      System.out.println (mainThread == Foo.currentThread ());
      System.out.println (mainThread == aFoo.currentThread ());
      aFoo.printStuff ();
   }
}
```

3 Synchronization

Two threads can communicate simply by setting variables that they both have access to. This sounds simple, but it is fraught with peril. Consider the following class, intended as a place for threads to leave String-valued messages for each other.

```
class Mailbox { // Version I. (Non-working)
   volatile String msg;

   Mailbox () { msg = null; }

   void send (String msg0) {
      msg = msg0;
   }

   String receive () {
      String result = msg;
      msg = null;
      return result;
   }
}
```

Here, I am assuming that there are, in general, several threads using a `Mailbox` to send messages and several using it to receive, and that we don't really care which thread receives any given message as long as each message gets received by exactly one thread.

This first solution has a number of problems:

1. If two threads call `send` with no intervening call to `receive`, one of their messages will be lost (a *write-write conflict*).

2. If two threads call `receive` simultaneously, they can both get the same message (one example of a *read-write conflict* because we wanted one of the threads to *write* null into `result` before the other thread *read* it.

3. If there is no message, then `receive` will return `null` rather than a message.

Items (1) and (2) are both known as *race conditions* – so called because two threads are racing to read or write a shared variable, and the result depends (unpredictably) on which wins. We'd like instead for any thread that calls `send` to wait for any existing message to be received first before preceding; likewise for any thread that calls `receive` to wait for a message to be present; and in all cases for multiple senders and receivers to wait their respective turns before proceeding.

3.1 Mutual exclusion

We could try the following re-writes of `send` and `receive`:

```
class Mailbox { Version II. (Still has problems)
   volatile String msg;

   Mailbox () { msg = null; }

   void send (String msg0) {
     while (msg != null)
        ; /* Do nothing */
     msg = msg0;
   }

   String receive () {
     while (msg == null)
       ; /* Do nothing */
     String result = msg;
     msg = null;
     return result;
   }
}
```

The **while** loops with empty bodies indulge in what is known as *busy waiting.* A thread will not get by the loop in `send` until `this.msg` becomes null (as a result of action by some other thread), indicating that no message is present. A thread will not get by the loop in `receive` until `this.msg` becomes non-null, indicating that a message is present. (For an explanation of the keyword **volatile**, see §3.4.)

Unfortunately, we still have a serious problem: it is still possible for two threads to call `send`, simultaneously find `this.msg` to be null, and then both set it, losing one message (similarly for `receive`). We want to make the section of code in each procedure from testing `this.msg` for null through the assignment to `this.msg` to be effectively *atomic* – that is, to be executed as an indivisible unit by each thread.

Java provides a construct that allows us to *lock* an object, and to set up regions of code in which threads *mutually exclude* each other if operating on the same object. We can use this construct in the code above as follows:

```
class Mailbox { // Version III.  (Starvation-prone)
   String msg;

   Mailbox () { msg = null; }

   void send (String msg0) {
     while (true)
        synchronized (this) {
           if (msg == null) {
              msg = msg0;
              return;
           }
        }
   }

   String receive () {
     while (true)
        synchronized (this) {
           if (msg != null) {
              String result = msg;
              msg = null;
              return result;
           }
        }
   }
}
```

The two **synchronized** blocks above are said to *lock* the object referenced by `this`.

When a given thread, call it t, executes the statement

synchronized (X) `{` S `}`

the effect is as follows:

- If some thread other than t has one or more *locks* on the object pointed to by X, t stops executing until these locks are removed. t then places a lock on the object pointed to by X (the usual terminology is that it *acquires* a lock on X). It's OK if t already has such a lock; it then gets another. These operations, which together are known as *acquiring a lock on (the object pointed to by)* X are carried out in such a way that only one thread can lock an object at a time.
- S is executed.
- No matter how S exits – whether by **return** or **break** or exception or reaching its last statement or whatever – the lock that was placed on the object is then removed.

As a result, if every function that touches the object pointed to by X is careful to synchronize on it first, only one thread at a time will modify that object, and many of the problems alluded to earlier go away. We call S a *critical region* for the object referenced by X. In our Mailbox example, putting accesses to the `msg` field in critical regions guarantees that two threads can't both read the `msg` field of a Mailbox, find it null, and then both stick messages into to it (one overwriting the other).

We're still not done, unfortunately. Java makes no guarantees about which of several threads that are waiting to acquire a lock will 'win'. In particular, one cannot assume a first-come-first-served policy. For example, if one thread is looping around in `receive` waiting for the `msg` field to become non-null, it can permanently prevent another thread that is executing `send` from ever acquiring the lock! The 'receiving' thread can start executing its critical region, causing the 'sending' thread to wait. The receiving thread can then release its lock, loop around, and re-acquire its lock even though the sending thread was waiting for it. We say that the sender can *starve.* There are various technical reasons for Java's being so loose and chaotic about who gets locks, which we won't go into here. We need a way for a thread to wait for something to happen without starving the very threads that could make it happen, and without monopolizing the processor with useless thumb-twiddling.

3.2 Wait and notify

The solution is to use the routines `wait`, `notify`, and `notifyAll` to temporarily remove locks until something interesting happens. These methods are defined on all `Objects`. If thread T calls X`.wait(),` then T must have a lock on the object pointed to by X (there is an exception thrown otherwise). The effect is that T temporarily removes all locks it has on that object and then *waits on* X. (Confusingly, this is *not* the same as waiting to acquire a lock on the object; that's why I used the phrase "stops executing" above instead.) Other threads can now acquire locks on the object. If a thread that has a lock on X executes `notifyAll`, then all threads waiting on X at that time stop waiting and each tries to re-acquire all its locks on X and continue (as usual, only one succeeds; the rest continue to contend for a lock on the object). The `notify` method is the same, but wakes up only one thread (choosing arbitrarily). With these methods, we can re-code `Mailbox` as follows:

```
// Version IV. (Mostly working)
void send (String msg0) {
  synchronized (this) {
    while (msg != null)
      try {
        this.wait ();
      } catch (InterruptedException e) { }

    msg = msg0;
    this.notifyAll ();
  }
}
```

```
String receive () {
  synchronized (this) {
    while (this.msg == null)
      try {
        this.wait ();
      } catch (InterruptedException e) { }

    String result = msg;
    this.notifyAll ();   // Unorthodox placement.  See below.
    msg = null;
    return result;
  }
}
```

In the code above, the `send` method first locks the `Mailbox` and then checks to see if all messages that were previously sent have been received yet. If there is still an unreceived message, the thread releases its locks and waits. Some other thread, calling `receive`, will pick up the message, and notify the would-be senders, causing them to wake up. The senders cannot immediately acquire the lock since the receiving thread has not left `receive` yet. Hence, the placement of `notifyAll` in the `receive` method is not critical, and to make this point, I have put it in a rather non-intuitive place (normally, I'd put it just before the return as a stylistic matter). When the receiving thread leaves, the senders may again acquire their locks. Because several senders may be waiting, all but one of them will typically lose out; that's why the `wait` statements are in a loop.

This pattern – a function whose body synchronizes on `this` – is so common that there is a shorthand (whose meaning is nevertheless identical):

```
synchronized void send (String msg0) {
  while (msg != null)
    try {
      this.wait ();
    } catch (InterruptedException e) { }

  msg = msg0;
  this.notifyAll ();
}

synchronized String receive () {
  while (this.msg == null)
    try {
      this.wait ();
    } catch (InterruptedException e) { }

  String result = msg;
```

```
    this.notifyAll ();
    msg = null;
    return result;
  }
```

Nevertheless, there are times when one wants to lock some arbitrary existing object (such as a shared `Vector` of information), and the **synchronized** statement I showed first is still useful.

It may have occurred to you that waking up all threads that are waiting only to have most of them go back to waiting is a bit inefficient. This is true, and in fact, so loose are Java's rules about the order in which threads get control, that it is still possible with our solution for one thread to be starved of any chance to get work done, due to a constant stream of fellow sending or receiving threads. At least in this case, *some* thread gets useful work done, so we have indeed improved matters. A more elaborate solution, involving more objects, will fix the remaining starvation possibility, but for our modest purposes, it is not likely to be a problem.

3.3 Read locks

While it is desirable to prevent one thread from modifying objects while another is also fiddling with them, there's nothing wrong with having as many threads as want to *look* at objects without modifying them. This leads to the idea of a *read lock,* a kind of synchronization device that allows access to an object either to an arbitrary number of *readers* (threads that do not modify an object) to access an object, or to one *writer* (a thread that does modify the object). There is no such specific construct in Java, but we can get the desired effect, as shown in the following example.

```
class ReadOrWrite {
   public void readMe (...) throws InterruptedException {
      lock.getReadLock ();
      try {
         // Perform actions that don't modify this.
      } finally { lock.releaseReadLock (); }
   }

   public void writeMe (...) throws InterruptedException {
      lock.getWriteLock ();
      try {
         // Perform any actions on this.
      } finally { lock.releaseWriteLock (); }
   }

   private Lock lock = new Lock ();
   // Other fields, etc.
}
```

The idea is to define the `Lock` class to give us the desired effects. We can't use synchronized methods for `readMe`, since that would allow only one reader at a time. Therefore, we must use `try` with a `finally` clause to make sure that the read lock gets released. I have done the same for writers to keep things symmetrical.

The problem of implementing the `Lock` class – known, appropriately enough, as the *readers/writers problem* – has an extensive history. Numerous solutions are possible, depending on how you choose to resolve various choices about preferences between readers and writers. Here, for example, is an implementation in which writers get preference: as long as there is a writer waiting to write (by calling `getWriteLock`), no further readers are allowed in.

```
class Lock {
   /** The current number of threads reading. */
   private int readers;
   /** The current number of threads writing or waiting to write. */
   private int writers;

   /** A new, unlocked lock. */
   public Lock () { readers = writers = 0; }

   public synchronized void getReadLock ()
        throws InterruptedException {
      while (writers > 0)
         wait ();
      readers += 1;
   }

   public synchronized void releaseReadLock ()  {
      readers -= 1;
      if (readers == 0 && writers > 0)
         notifyAll ();
   }

   public synchronized void getWriteLock ()
        throws InterruptedException {
      writers += 1;
      try {
        while (readers > 0 || writers > 1)
           wait ();
      } catch (InterruptedException e) {
        /* Undo the effect of the call. */
        writers -= 1;
        throw e;
      }
   }
```

```
    public synchronized void releaseWriteLock () {
        writers -= 1;
        notifyAll ();
    }
}
```

3.4 Volatile storage (a short side trip)

You probably noticed the mysterious qualifier **volatile** in the busy-waiting example. Without this qualifier on a field, a Java implementation is entitled to pretend that no thread will change the contents of a field after another thread has read those contents until that second thread has acquired or released a lock, or executed a `wait`, `notify`, or `notifyAll`. or example, a Java implementation can treat the following two pieces of code as identical, and translate the one on the left into the one on the right:

```
                               tmp = X.z;
while (X.z < 100) {            while (tmp < 100) {
    X.z += 1;                     tmp += 1;
}                              }
                               X.z = tmp;
```

But of course, the effects of these two pieces of code are *not* identical if another thread is also modifying `X.z` at the same time. If we declare `z` to be volatile, on the other hand, then Java must use the code on the left.

By default, fields are not considered volatile because it is potentially expensive (that is, slow) not to be able to keep things in temporary variables (which may often be fast registers). It's not common in most programs, with the exception of what I'll call the *shared, one-way flag idiom.* Suppose that you have a simple variable of some type other than `long` or `double`[1] and that some threads only read from this variable and others only write to it. A typical use: one thread is doing some lengthy computation that might turn out to be superfluous, depending on what other threads do. So we set up a boolean field in some object that is shared by all these threads:

```
volatile boolean abortComputation;
```

[1]The reason for this restriction is technical. The Java specification requires that assigning to or reading from a variable of a type other than `long` and `double`, is *atomic,* which means that any value read from a variable is one of the values that was assigned to the variable. You might think that always has to be so. However, many machines have to treat variables of type `long` and `double` internally as if they were actually two smaller variables, so that assigning to them is actually *two* operations. If a thread reads from such a variable *during* an assignment (by another thread), it can therefore get a weird value that consists partly of what was previously in the variable, and partly what the assignment is storing into the variable. Of course, a Java implementation could "do the right thing" in such cases by quietly performing the equivalent of a **synchronized** statement whenever reading or writing such "big" variables, but that is expensive, and it was deemed better simply to let bad things happen and to warn programmers to be careful.

Every now and then (say at the top of some main loop), the computing thread checks the value of `abortComputation`. The other threads can tell it to stop by setting `abortComputation` to **true**. Making this field volatile in this case has exactly the same effect as putting every attempt to read or write the field in a `synchronized` statement, but is considerably cheaper.

4 Interrupts

Amongst programmers, the traditional definition of an *interrupt* is an "asynchronous transfer of control" to some pre-arranged location in response to some event[2]. That is, one's program is innocently tooling along when, between one instruction and the next, the system inserts what amounts to a kind of procedure call to a subprogram called an *interrupt handler.* In its most primitive form, an interrupt handler is not a separate thread of control; rather it usurps the thread that is running your program. The purpose of interrupts is to allow a program to handle a given situation when it occurs, without having to check constantly (or *poll,* to use the technical term) to see if the situation has occurred while in the midst of other work.

For example, from your program's point of view, an interrupt occurs when you type a `Ctrl-C` on your keyboard. The operating system you are using handles numerous interrupts even as your program runs; one of its jobs is to make these invisible to your program.

Without further restrictions, interrupts are extremely difficult to use safely. The problem is that if one really can't control at all where an interrupt might occur, one's program can all too easily be caught with its proverbial pants down. Consider, for example, what happens with a situation like this:

```
// Regular program                 // Interrupt handler
theMailbox.send (myMessage);       theMailbox.send (specialMessage);
```

where the interrupt handlers gets called in the middle of the regular program's `send` routine. Since the interrupt handler acts like a procedure that is called during the sending of `myMessage`, we now have all the problems with simultaneous sends that we discussed before. If you tried to fix this problem by declaring that the interrupt handler must wait to acquire a lock on `theMailbox` (i.e., changing the normal rule that a thread may hold any number of locks on an object), you'd have an even bigger problem. The handler would then have to wait for the regular program, but the regular program would not be running – the handler would be!

Because of these potential problems, Java's "interrupts" are greatly constrained; in fact, they are really not asynchronous at all. One cannot, in fact, interrupt a thread's execution at an arbitrary point, nor arrange to execute an arbitrary handler. If `T` points to an active `Thread` (one that has been started, and has not yet terminated), then the call `T.interrupt()` puts the thread associated with `T` in *interrupted status*. When that thread subsequently performs a `wait` – or if it is waiting at the time it is interrupted – two things happen in sequence:

- The thread re-acquires all its locks on the object on which it executed the `wait` call in the usual way (that is, it waits for any other thread to release its locks on the object).

[2] Actually, 'interrupt' is also used to refer to the precipitating event itself; the terminology is a bit loose.

- The thread then throws the exception `java.lang.InterruptedException`. This is a checked exception, so your program must have the necessary `catch` clause or `throws` clause to account for this exception. At this point, the thread is no longer in interrupted status (so if it chooses to wait again, it won't be immediately re-awakened).

You've probably noticed that being interrupted requires the coöperation of the interrupted thread. This discipline allows the programmers to insure that interrupts occur only where they are non- disruptive, but it makes it impossible to achieve effects such as stopping a renegade thread "against its will." In the current state of the Java library, it appears that this is, in fact, impossible. That is, there is no general way, short of executing `System.exit` and shutting down the entire program, to reliably stop a runaway thread. While poking around in the Java library, you may run across the methods `stop, suspend, resume,` and `destroy` in class `Thread`. All of these methods are either *deprecated,* meaning that they are present for historical reasons and the language designers think you shouldn't use them, or (in the case of `destroy`) unimplemented. I suggest that you join in the spirit of deprecation and avoid these methods entirely. First, it is very tricky to use them safely. For example, if you manange to `suspend` a thread while it has a lock on some object, that lock is not released. Second, because of the peculiarities of Java's implementation, they won't actually do what you want. Specifically, the `stop` method does not stop a thread immediately, but only at certain (unspecified) points in the program. Threads that are in an infinite loop doing nothing but computation may never reach such a point.

5 Programming Discipline

The preceding sections, I hope, have given you some notion that concurrent programming is tricky. Basically, the problem is *nondeterminism* in how the instructions from multiple threads interleave in time, which causes trouble whenever instructions from several threads attempt to modify the same variables. There was a great deal of research, largely in the late 1960s and 1970s, on how to control the resulting confusion. Various languages from that era (such as Concurrent Pascal) incorporated language features to this end. Java, with its synchronized methods and its `wait` and `notify` methods, adopted the forms of some of these features, but didn't, I fear, get them quite right, allowing us programmers a bit more freedom than is good for us.

I suggest, therefore, that to avoid some of the more egregious bugs, you restrict communication between multiple threads to *monitors.* For our purposes, a monitor is an object whose class is defined according to the following restrictions:

- All non-final instance variables (non-static fields) are private.
- Likewise, any other objects reachable from[3] the object are either monitors themselves or are *immutable* (that is, like Java Strings, they cannot be modified).

[3] An object x is *reachable* from a pointer if the pointer points to x or an object from which x is reachable. Likewise, x is reachable from a variable if the variable contains a pointer from which x is reachable. Finally, x is reachable from another object, y, if y is x itself, or if it contains a non-private instance variable from which x is accessible (recursively), or if one of its non-private methods can return a pointer from which x is accessible.

- All non-private methods that access any instance variables are synchronized. As discussed in §3.4, it is is unnecessary to be synchronized if the sole effect of a method is to write or read a "small," volatile primitive variable; in effect, reading and writing such variables are miniature synchronized procedures in their own right.
- Alternatively, one can use a read-lock scheme (see §3.3) for methods that only read instance variables, and explicit write locks for methods that modify instance variables.