Threads in Java

To put code in a thread, extend the built-in Thread class and override run:

```java
class HelloThread extends Thread {
    public void run() {
        System.out.println("hello");
    }
}
```
Threads in Java

To run a thread, instantiate the class and call `start` (not `run!`):

```java
Thread t1, t2;
t1 = new HelloThread();
t2 = new HelloThread();
t1.start();
t2.start();
try {
    t1.join();
    t2.join();
} catch (InterruptedException i) {
    System.exit(1);
}
```

Copy
Thread-Local Data

Use fields in the `Thread` class for thread-local data:

class HelloThread extends Thread {
    int id;
    HelloThread(int id) { this.id = id; }
    public void run() {
        System.out.println("hello from " + id);
    }
}

... 

    t1 = new HelloThread(0);
    t2 = new HelloThread(1);
    ...

Copy
Concurrent Modification

Modifying a variable from multiple threads is as wrong in Java as in C:

```java
int counter = 0;

class CountThread extends Thread {
    public void run() {
        counter++;  // unpredictable!
    }
}
```
Synchronization

Java’s `synchronized` is similar to Peril-L’s `exclusive`, but mutual exclusion is based on an object instead of a statement:

```java
Integer counter = 0;

class CountThread extends Thread {
    public void run() {
        synchronized (counter) { // ok
            counter++;
        }
    }
}
```
Synchronization

Java’s *synchronized* is similar to Peril-L’s *exclusive*, but mutual exclusion is based on an object instead of a statement:

```java
Object thing = new Object();
int counter = 0;

class CountThread extends Thread {  
  public void run() {    
    synchronized (thing) { // ok
      counter++;
    }
  }
}
```
Java’s **synchronized** is similar to Peril-L’s **exclusive**, but mutual exclusion is based on an object instead of a statement:

```java
int counter = 0;

class CountThread extends Thread {
    public void run() {
        synchronized (this) { // wrong!
            counter++;
        }
    }
}
```
Synchronization

If a method has the `synchronized` attribute, then each call is implicitly wrapped with `synchronized`:

```java
class Thing {
    int counter;
    public synchronized void inc() {
        counter++;
    }
}
...
Thing t = new Thing();
...
synchronized (t) { t.inc() }
t.inc(); // equivalent to previous line
...
```

If a method has arguments, however, the argument expressions are not included in the implicit `synchronized`
Synchronization

Some standard classes, such as Vector, have only synchronized methods.

```java
Vector counter = new Vector();
LinkedList counter2 = new LinkedList();

class CountThread extends Thread {
    public void run() {
        counter.add(this); // ok
        counter2.add(this); // not ok!
    }
}
```
Synchronization

Using only *synchronized* methods does not mean that your code is thread-safe:

```java
Vector v = new Vector();

class CountThread extends Thread {
    public void run() {
        v.set(0, 1 + (Integer)v.get(0)); // wrong
    }
}
```
Synchronization

Using only `synchronized` methods does not mean that your code is thread-safe:

```java
Vector v = new Vector();

class CountThread extends Thread {
    public void run() {
        v.set(0, 1 + (Integer)v.get(0)); // wrong
    }
}
```
Communicating Between Threads

Sometimes you need to get a value from one thread to another:

```java
class PutThread extends Thread {
    public void run() {
        int v = new Random().nextInt();
        \... v ...; // send v
        System.out.println("sent " + v);
    }
}

class GetThread extends Thread {
    public void run() {
        int v = ...; // receive v
        System.out.println("got " + v);
    }
}
```
Communicating Between Threads

Use `synchronized`?

```java
Integer box;

class PutThread extends Thread {
  ...
  synchronized (box) { box = v; }
  ...
}

class GetThread extends Thread {
  ...
  synchronized (box) { v = box; }
  ...
}
```

Doesn’t ensure put before get!
Communicating Between Threads

Typical newbie “solution”:

```java
boolean ready;
int box;

class PutThread extends Thread {
    ...
    box = v;  ready = true;
    ...
}

class GetThread extends Thread {
    ...
    while (!ready) { Thread.sleep(10); }
    v = box;
    ...
}
```
Communicating Between Threads

Typical newbie “solution”:

```java
boolean ready;
int box;

class PutThread extends Thread {
    ...
    box = v; ready = true;
    ...
}

Thread extends Thread {
    ...
    while (!ready) { Thread.sleep(10); }
    v = box;
    ...
}
```

(not sync'ed)
Communicating Between Threads

Typical newbie “solution”:

```java
boolean ready;
int box;

class PutThread extends Thread {
    ...
    box = v;  ready = true;
    ...
}

class GetThread extends Thread {
    ...
    while (!ready) { Thread.sleep(10); }
    v = box;
    ...
}
```

not sync'ed

wasted cycles, increased latency
Communicating Between Threads

Java includes lots of data structures to solve these kinds of problems:

```
SynchronousQueue q;

class PutThread extends Thread {
    ...
    q.add(v);
    ...
}

class GetThread extends Thread {
    ...
    v = (Integer)q.take();
    ...
}
```
Communicating Between Threads

From scratch, analogous to POSIX support:

```java
boolean ready;
int box;
Lock lock = new ReentrantLock();
Condition nowReady = lock.newCondition();

class PutThread extends Thread {
    ...
    lock.lock();
    box = v;
    ready = true;
    nowReady.signal();
    lock.unlock();
    ...
}
```
Communicating Between Threads

From scratch, continued:

class GetThread extends Thread {
  ...
  lock.lock();
  while (!ready) {
    nowReady.await();
  }
  v = box;
  lock.unlock();
  ...
}

Copy