Threads

So far, all our programs have been running a single thread of control, but it's often convenient to be able to run either several threads independently & concurrently, or have threads branch off & yet still communicate with one another. We'll look at a few elementary approaches to multi-threading...

There is a Thread class, so we can do...

```java
Thread sausage = new Thread();
```

which creates a new thread sausage which can be configured & run. However, `sausage.run();` won't do anything — the compiler doesn't know anything special about running sausages!! Better would be to extend the thread class and then define `run()` in this derived class...

```java
public class PingPong extends Thread
{
    private String word;
    private int delay;

    public PingPong (String password, int pendant)
    {
        word = password;
        delay = pendant;
    }

    public void run ()
    {
        try
        {
            System.out.println (word + " ");
            sleep (delay);
        }
        catch (InterruptedException e)
        {
            return;
        }
    }
}
```
Then if we have...

```java
    public static void main (String [] args) {
        new PingPong("ping", 333), startC();
        new PingPong("PONG", 1000), startC();
    }
```

we'll get ping appearing on the screen every 1/3 second and PONG every second (they will have functionally different start times)... ping PONG ping ping PONG ping ping ping PONG ping ping ping PONG ping ping ping PONG ping ping ping PONG ping ping ping PONG ping ping ping PONG ping ping ping PONG ping ping ping PONG ping ping ping PONG ping ping ping PONG

Here we have two separate & independently running threads.

Suppose we want some rudimentary control on when particular data fields can be accessed. Consider for example a bank account in a multi-threaded environment, so the same account could potentially be accessed simultaneously for deposit & withdrawal by independent threads — clearly a dangerous thing! To safeguard this situation...

```java
    class BankAcoo {
        private float balance;
        public BankAcoo (float amt) { balance = amt; }
        public synchronized float spend (float amt) {
            balance -= amt; return balance;
        }
        public synchronized float deposit (float amt) {
            balance += amt; return balance;
        }
    }
```
Synchronizing per se makes no guarantee of order of access, but does ensure that only one synchronized method at a time can have access. Synchronized methods of any given instantiated object block each other, and synchronized static methods block each other at the class level, but there is no mutual blocking of static vs. non-static methods. A child class can override a synchronized method so that in the child that method is synchronized only if it's explicitly declared as synchronized.

We can also synchronize whole chunks of code as a 'block' statement...

```java
public static void abs (int [] values)
{
    synchronized (values)
    {
        for (int i = 0; i < values.length; i++)
            if (values[i] < 0) values[i] = -values[i];
    }
}
```

This reference object is now locked so that the `{...}` behaves like a synchronized method. So this method can proceed without any interference on the values array by any other code also synchronized on values.

If you already have code written without any thought of multithreading, rather than rework the whole code with intricate synchronizations, you can "create an extended class to override the appropriate methods, declare them synchronized, and then forward method calls through the super reference. If only occasional synchronized access is needed, then it's easier just to use a synchronized statement as above."
Using synchronized prevents interference between various threads, but to force actual interaction we use wait() and notify() inside a synchronized block or method. Generically...

```java
synchronized void hangOn()
{
    while (!condition)
        wait();
    // now do whatever you wanted when condition true
}
```

Here, `wait()` not only pauses the thread, it also temporarily releases the lock for the duration of this pause. This is said to happen atomically, meaning that the pause and lock release occur simultaneously and indivisibly—otherwise strange things could happen! It's crucial that the condition test be in some loop—or if would be disastrous! Teamed with this code should be something like...

```java
synchronized void readyNow()
{
    here's where to change some value used
    in a 'condition' test
    notifyAll();
}
```

Alternatively, use `notify()` to
just one thread—this has to
be used with great care, the
`notifyAll()` is much safer.

As always, it's helpful to see the methods' declarations, so...
public final void wait (long timeout)  throws InterruptedException

This makes the current thread wait until it is notified or until the timeout (in milliseconds) has expired. To ignore the timeout, set its value to 0. If you want to be really fussy, there's another flavour...

public final void wait (long timeout, int nanos)  throws InterruptedException

which allows you to specify 0-999,999 nanoseconds added to your timeout! The default wait () is equivalent to wait (0).

For notifications...

public final void notifyAll ()

notifies all the threads currently waiting for a condition to change, and is generally the safer and preferred version.

public final void notify ()

will only notify ≤1 thread waiting for a condition to change. **BUT** you can't choose which thread this will be!!!

It's safe, for example, if you are sure that at this time there can only be one thread waiting, and such certainty is evasive.

If you attempt to use these methods on objects from outside the synchronized code which acquired the lock, you will get an **IllegalMonitorStateException**.

As a quickie example...
class Queue
{
    Node front, back;
    boolean empty;

    public Queue()
    {
        front = new Node();
        back = front;
        empty = true;
    }

    public synchronized void append(Object x)
    {
        if (empty)
        {
            front.data = x;
            empty = false;
        }
        else
        {
            back.next = new Node(x);
            back = back.next;
        }
        notifyAll();
    }

    public synchronized Object get()
    {
        try
        {
            while (empty)
            {
                wait();
            }
            Object temp = front.data;
            if (front == back)
            {
                front.data = null;
                empty = true;
            }
            else
            {
                front = front.next;
            }
            return temp;
        }
        catch (InterruptedException e)
        {
            return null;
        }
    }
}
There are actually ways of setting various levels of priority on threads. These priorities are integers ranging from \( 1 = \text{MIN\_PRIORITY} \) to \( 10 = \text{MAX\_PRIORITY} \). There is also a \( \text{NORM\_PRIORITY} = 5 \), which is the default priority given to the first user thread. Naturally there are two thread methods...

public final void setPriority (int newPriority)
public final int getPriority ()

A thread inherits its priority from the thread which created it, though it can be changed at any time. Effectively, JVM runs the highest priority thread(s) that are not currently blocked. As a general guide, the continually running part of your program should have a relatively low priority, allowing user input to have higher priority! There are two other methods worth noting in this context...

public static void sleep (long timeout)  
throws InterruptedException

public static void yield ()

Since timeout is in milliseconds, there's also a sleep method taking two arguments, with the second being additional nanoseconds. You should note that sleep pauses the thread for \( \geq \) timeout, not \( \text{exactly} \) timeout; also it remains in active control of any locks, unlike \text{wait} (). The \text{yield}() method allows another thread to run -- this means \text{ANY} other runnable thread, including the one which just yielded! As an example...
class Bubble extends Thread {
    static boolean aprestoi;
    static int freq;
    private String word;
    Bubble (String data) { word = data; }

    public void run ()
    
    
    {
        for (int i=0; i<freq; i++)
        {
            System.out.println (word + "");
            if (aprestoi) yield ();
        }
    }

    public static void main (String[] args) throws Exception {
        freq = Integer.parseInt (args[1]);
        aprestoi = new Bashnek (args[0], bashnekValue ());
        Thread cur = currentThread ();
        cur.setPriority (Thread.MAX_PRIORITY);
        for (int i=2; i<args.length; i++)
            new Bubble (args[i]), start ();
    }

    Then...

    Bubble false 2 Yes No → Yes Yes No No
    at least on some runs

    Bubble true 2 Yes No → Yes No Yes No

Multithreading is often thought of as a tricky exercise, the ideas are straightforward, but logical problems can arise...

**Events**

| #1 | thread A invokes a synchronized method \( X.f \) which puts a lock on object \( X \). |
| #2 | thread B invokes a synchronized method \( Y.g \) which puts a lock on object \( Y \). |
| #3 | \( X.f \) now invokes a synchronized method \( Y.h \), but object \( Y \) is currently locked by the thread B, so thread A must wait until B finishes. |
| #4 | \( Y.g \) now invokes a synchronized method \( X.ouch \), but object \( X \) is currently locked by the thread A, so thread B must wait until A finishes, which will never happen !!!! |

Java has no means (currently) of detecting or preventing such deadlocks — so multithread with great care!

We've seen threads run independently, protect objects from multiple simultaneous access, wait & notify, yield, and be assigned priorities. We should look now at ways of terminating threads.

Normally, a thread ends when its \( run() \) returns. However, consider the following code schematics...
<table>
<thead>
<tr>
<th>thread A</th>
<th>thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>thread B. interrupt();</td>
<td>while (!isInterrupted())</td>
</tr>
<tr>
<td></td>
<td>{ do fun stuff }</td>
</tr>
</tbody>
</table>

Thread A's interruption of thread B doesn't actually force thread B to halt immediately (unless B is sleeping or waiting), it's more an attention-getter — it changes the value of the thread's interrupt flag, so that when/if the thread checks that value (via isInterrupted()) it can halt its execution.

If an `InterruptedException` is thrown, for example by `sleep()` or `wait()`, then the catch block of this exception clears the interrupt flag, so a check of its value will indicate no interruption called for. To handle that, we could do something like...

```java
void tick (int ceiling; long weighted) {
  try {
    for (int i = 0; i < count; i++) {
      System.out.print(i);  
      System.out.flush();  
      Thread.sleep (weighted * 1000);  
    }
  }
  catch (InterruptedException e) {
    Thread.currentThread().interrupt();
  }
}
```

prints * periodically
There are methods...

public final void stop()
public final synchronized void stop (Throwable e)
which can be awkward to control, and are 'deprecated',
and also...

public final void suspend()

similarly awkward, and similarly deprecated.

Finally, in this mood, we can have one thread
wait for another to finish by using join()...

class CalcThread extends Thread
private double ans;
public void run () { ans = calculate (); }
public double getAns () { return ans; }
public double calculate ()
    { do amazing stuff to calculate answer; }

class TotalUp
public static void main (String [] args)
    { CalcThread calc = new CalcThread ();
      calc.start ();
      do something fun here;
try { calc.join ();
      System.out.println ("ans is " + calc.getAns ()�
  }
catch (InterruptedException e) {
      System.out.println ("no ans: interrupted!");
  }
The formal declarations are...

```java
public final void join() throws InterruptedException
```

```java
public final synchronized void join(long timeout)
```

```java
throws InterruptedException
```

with (of course) a version with nanoseconds as well.

When a thread dies, its object remains, so its state can still be accessed.

There are many occasions where being restricted to having to extend the class Thread in order to re-define the `run()` method isn’t feasible — for example, when you need to inherit from another class. For this purpose the more common approach is to implement the interface `Runnable`, which also has a `run()` method declared. For details, check the online API.

We move now into a quick look at GUIs, with our main emphasis on Swing. The relevance is that unlike the “heavyweight” processes of the underlying AWT, there are times when we have to exercise care in the way Swing handles threads.