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 multithreading...

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

```
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...

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

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

    public void run ()
      { try { for (;;)
                { System.out.print (word + " ");
                  sleep (delay); }
            }
        catch (InterruptedException e) {return;}
      }
  }
```

Then if we have ...

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

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

ping PONG ping ping ping PONG ping ping PONG ping
ping PONG ping ping PONG ping ping ping PONG ping

Here we have two separate & independently running threads.

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

```
class BankAcc
  { private float balance
    public BankAcc (float amt) { balance = amt; }
    public synchronized float spend (float amt)
       { balance -= amt; return balance; }
    public synchronized float deposit (float amt)
       { balance += amt; return balance; }
  }
```

Whichever of these is the 1st thread that finds the object free, locks it until done.

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 'local' statement ...

```
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 that 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 rewrite 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...

```
synchronized void hangOn()
{
    while (! condition)
      wait();
    now do whatever you wanted when condition is 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 — an if would be disastrous! Teamed with this code should be something like...

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

alternatively, use notify() to jog 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 elusive.

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 puzzle example...

```
class Cue
  {
    Qnode front, back;
    boolean empty;

    public Cue ()
      { front = new Qnode (); back = front; empty = true; }

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

    public synchronized Object get ()
      {
        try { while (empty)
                wait ();
          }
        catch (InterruptedException e)
          { return null; }
        Object temp = front.data;
        if (front == back)
          { front.data = null; empty = true; }
        else { front = front.next; }
        return temp;
      }
  }
```

```
class Qnode
  {
    Object data;
    Qnode next;
    public Qnode (Object x)
      { data = x; next = null; }
    public Qnode () { this (null); }
  }
```

There are actually ways of setting various levels of priority on threads. These priorities are integers ranging from 1 = MIN_PRIORITY to 10 = MAX_PRIORITY. There is also a 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, Java 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 ≥ timeout, not exactly timeout; also it remains in active control of any locks, unlike wait(). The yield() method allows another thread to run — this means ANY other runnable thread, including the one which just yielded! As an example ...

```
class Bubble extends Thread
  {
    static boolean aPrestori;
    static int freq;
    private String word;
    Bubble (String dabar) { word = dabar; }

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

    public static void main (String[] args) throws Exception
      { freq = Integer.parseInt (args[1]);
        aPrestori = new Boolean(args[0]).booleanValue();

        Thread cur = currentThread();
        cur.setPriority(Thread.MAX_PRIORITY);
        for (int i=2; i<args.length; i++)
          new Bubble (args[i]).start();
      }
  }
```

because of parseInt !!

Then...

Bubble false 2 Yes No ⟶ Yes Yes No No

Bubble true 2 Yes No ⟶ Yes No Yes No

(at least on some runs)

Multithreading is often thought of as a tricky exercise, the ideas are straightforward, but logical conundra 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 (essentially) 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 schematic...

thread A	thread B
threadB.interrupt();	while (!isInterrupted()) { do fun stuff }

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 catching 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...

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

prints periodically

interrupt() causes this exception & clears interrupt flag

forces interruption!

These 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 JoinUp
  { 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!"); }
      }
  }
```

main thread starts

default from Thread class

continue in main thread keeping busy!

spawns + starts a new thread

now the main thread sits, waiting to join the CalcThread root, but has to wait until calc finishes

The formal declarations are...

```
public final void join() throws InterruptedException

public final synchronized void join(long timeout)
        throws InterruptedException
```

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

When a thread dies, its object remains, so the 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 while the 'heavyweight' processes of the underlying AWT, there are times when we have to execute care in the way Swing handles threads.

Swing with Threads

The primary thread for drawing & event handling is the event-dispatching thread. This is automatically used by the paint() and actionPerformed methods. If a component's paint() method has been called, or if a top-level Swing component that had show(), pack(), or setVisible(true) invoked, or if a component is added to such a container, then it is said to have been realized.

Essentially, the 'normal' rule is that "once a Swing component has been realised, all code affecting or depending on the state of that component should be executed in the event-dispatching thread. Exceptions to this do exist; although most Swing components are not thread-safe, a few have been explicitly declared safe in the API, so these can be used with less fear.

Other exceptions ...

- applets - can construct/show in init()
- repaint(), revalidate(), invalidate() are safe, so can be called from any thread (since they either queue requests in the event-dispatching thread, or effectively set flags).
- add-xxxListener(), remove-xxxListener() can also be called from any thread since they have no effect on event dispatching in progress.

So why might we want to manipulate threads when doing GUIs?

- busy initialisation or calculation

  don't make your GUI wait for network connections, extensive loading of files/classes, big calculations. It's better to start the GUI, spawn a thread to do the slow stuff, & then continue with the GUI, being prepared to update it when the slow stuff is done.

- repetitive processes

  spawn a thread to handle time interval delayed & repeated stuff.

- updating depends on external events

  again, a separate thread is best here.

There are several useful utilities available to save us from having to get involved in detailed thread coding. Two from the SwingUtilities class are ...

invokeLater() — requests some code to be executed in the event-dispatching thread. This method would be called by another thread & returns immediately without waiting for the code to execute. The code has to be put in the run() method of a Runnable object and specify the Runnable object as the argument of invokeLater(). For example ...

```
Runnable goof = new Runnable(){
                    public void run(){ woof(); }};

SwingUtilities.invokeLater( goof );
```

invokeAndWait() — similar to invokeLater() save that it waits for the code to finish before returning. Not used as much for the obvious potential lock problems. An example where it might be used is the following, where the goal is to print the combined contents of 2 textfields — so it's worth waiting for them to be given content ...

```
void printStuff() throws Exception
{
    final String[] a = new String[2];
    Runnable r = new Runnable() {
                    public void run() {
                        a[0] = textField0.getText();
                        a[1] = textField1.getText();
                    } };
    SwingUtilities.invokeAndWait(r);
    System.out.println(a[0] + " " + a[1]);
}
```

There are 2 classes which can also be used to ease the process of writing multithreaded GUI code ...

SwingWorker.java ← for background threads

which should be downloaded directly from ...

java.sun.com/products/jfc/tsc/articles/threads/src/SwingWorker.java

and the Timer class, which lives in Swing. ← for repeated acting