Threads. So far, we have been running a single thread of control, but it’s often convenient to be able to run either several threads independently and concurrently, or have threads branch off and yet still communicate with one another. We’ll look at a few elementary approaches to multi-threading (for a more in-depth discussion, follow the link to the Java Specification on the course homepage and read ch 17) ...

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

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

to create a new thread sausage which can be configured and run. However sausage.run( ); won’t do anything ... the computer doesn’t know anything special about running sausages!! Better would be to extend the Thread class and then redefine run( ) in the derived class ...

```java
public class PingPong extends Thread { // from the java.sun.com thread tutorial
    private String word ; // the word to print
    private int delay ; // the delay in millisecs to pause

    public PingPong ( String parole , int pendant ) {
        this.word = parole ;    this.delay = pendant ;
    } // end constructor

    public void run ( ) { // overriding Thread's run ( ) method
        try {
            for (   ;   ;   ) { // never stop (unless interrupted) !!!!
                System.out.print ( word + " " ) ;    sleep ( delay ) ;
            }
        }  catch ( InterruptedException ie )  {  return ;  }
    } // end overriding of the run ( ) method

} // end class PingPong
```

Then if we have ...

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

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

ping PONG ping ping ping PONG ping ping ping PONG ping

Hence two separate and independently running threads.
• There’s another more or less equivalent way of doing this, especially useful if you want to inherit from some class and don’t want to use up your one inheritance opportunity by having to extend the `Thread` class ...

```java
public class RunPingPong implements Runnable { // from the “java programming language” book
    private String word;
    private int delay;

    public RunPingPong(String parole, int pendant) {
        this.word = parole;
        this.delay = pendant;
    } // end constructor

    public void run() {
        try {
            for ( ; ; ) {
                System.out.print(word + " ");
                Thread.sleep(delay);
            } // only exit from for loop is via an interrupt
        } catch (InterruptedException ie) { return; }
    } // end implementing the run() method
} // end class RunPingPong
```

Then if we have ...

```java
public static void main(String[] args) {
    Runnable little, bigger;
    little = new RunPingPong("ping", 333);
    bigger = new RunPingPong("PONG", 1000);
    new Thread(little).start();
    new Thread(bigger).start();
} // end main method
```

we’ll get the same behaviour as before.

• We’ll say more about `interfaces` like `Runnable` in the next section. For now you can think of them as mold-like superclasses which only have methods - declared, but never defined. They come with an implied contract to `define` every method they have if you want to `implement` (néé extend) them. For the case of `Runnable`, it only declares one method, namely `run()`.
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 what if the same account data (e.g. `balance`) could be accessed simultaneously by independent threads?

It could be argued that each thread ran ‘correctly’, but because thread B read the value of `balance` after thread A’s read but before thread A had completed its calculation, thread B incremented ‘the wrong’ value, hence leaving the balance as if that $2000 have never been deposited. Such a situation is called a *race condition*.

The same situation could occur even with a simple expression like `oops++`; which technically comprises three operations: read `oops`, add one to `oops`, write `oops` back into memory. Concurrent writes or read/writes on a value are dangerous!!! (Concurrent reads are safe.) To safeguard this situation ...
Synchronizing per se makes no guarantee of the order of access, but does ensure that only one synchronized method at a time can have access to any data fields addressed within that method.

(Synched methods of any given instantiated object block each other, and synched static methods block each other at the class level, but there is no mutual blocking of static vs non-static methods. Note that because a child class could potentially override a parent’s synched method, it’s the case that in the child that method is actually synched only if it’s explicitly declared as synched in the child class.)

**DANGER!** Synchronizing is not a universal panacea, indeed it can be quite devastating if used without care .......... Imagine a bunch of quick processes waiting while a synched laborious process rambles on. Worse still, suppose you have two instances \( x \) and \( y \) of a class \( G \) having synched methods \( \text{hug}(\ ) \) and \( \text{hugback}(\ ) \) which act on \( G \)’s, and suppose \( \text{hug} \) invokes \( \text{hugback}(\ ) \). Then

\[
\text{thread A} \quad \text{\hspace{0.5cm} x.\text{hug}( y )} \quad \text{\hspace{1cm} y.\text{hugback}( x )} \quad \text{waiting} \\
\quad \text{so thread A has a lock on } y \quad \text{so now A wants a lock on } x \text{ but is blocked by B} \\
\text{thread B} \quad \text{\hspace{0.5cm} y.\text{hug}( x )} \quad \text{\hspace{1cm} x.\text{hugback}( y )} \quad \text{waiting} \\
\quad \text{so thread B has a lock on } x \quad \text{so now B wants a lock on } y \text{ but is blocked by A} \\
\]

this’ll be ok as soon as B is done and releases the lock

OOPS!!!

“deadlock”

public class BankAcc {
    private float balance;

    public synchronized float spend(float amt) {
        balance -= amt;
        return balance;
    }

    public synchronized float deposit(float amt) {
        balance += amt;
        return balance;
    }
}

// end class BankAcc
• We can also synch whole chunks of code as a ‘local’ statement. Consider the following method to convert an int array to absolute values ...

```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 ] ;
    } // end synched block on the values array
} // end abs method
```

(Although safe in a single-threaded environment without the ‘synchronized’ epithet, it must be synched if multi-threaded, otherwise some other thread might access value[i] after the abs method reads the boolean test, and then overwrite value[i] with 23, so that when abs multiplies and writes, it will leave the array with value[i] = -23. Being able to synch smaller chunks of code is a valuable option, since in general we don’t want to force other processes to have to wait longer than necessary, plus it can help evade some potential deadlocks. In particular this allows us to have a more fine-grained control on acquiring and releasing locks, for instance, we could use this to allow parallel calls to multiple methods within a class yet still protecting access to disjoint data fields.)

• If you already have code written without any thought of multi-threading, rather than rework the whole code with intricate synchs, 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 synchronised access is needed, then it’s usually easier just to use a synched statement as above. (More on threads later in the course.)