(Lack of) Parallelism

- Standard programming in imperative programming languages is intrinsically sequential: first do this then that then that.
- OO does not really change that, except perhaps if objects run their own code, e.g. Java initialisers.
- In Java, this is not parallel either.

(non-parallel) examples

static variables at class level:

```
static final int longcomp=abc.call();
```

static initialisers, at class level (just as methods):

```
static{ System.out.println("I'm running"); }
```

instance initialisers, at class level; as above, without the keyword **`static`**.

All these have well-defined execution scenarios.

Why parallelism?

- **interact** with real-life parallelism, e.g. mail server, multi-user OS, etc.
- **minimalism**: do not enforce sequential order unless the problem requires it
- increase component-**independence** - one window crashes, the program carries on
- exploit **hardware**

Parallelism in Java

- a process in Java is called a thread
- threads are objects of class **`Thread`**
- threads can be started and then run (fairly) independently {often rather unfairly}
- we can wait for threads to finish
- we also talk about the *main thread of control*, although it isn't a **`Thread`** object

How?

- we typically write a subclass of the class `Thread` (`java.lang.Thread`)
- this subclass needs to implement the method `public void run()` - the code of the thread
- we ***create*** a thread object
- we fork off the thread by calling its `start()` method - this then executes the `run()` method

not in the Thread constructor, please!

Why?

...don't we just run the `run()` method?

We can, it just is an ordinary method of an ordinary object.

But: it would not be executed in a separate thread, but in the thread in which we call.

In particular, `start()` terminates quickly as it only forks off a new thread; `run()` runs to completion.

`java.lang.Thread` (selection)

```
void start();
final void join();
static void sleep(long millisecs);
static void yield();
final boolean isAlive();
final void setPriority(int prio);
Thread (Runnable r);
```

Example

```
class Devil extends Thread {
  Socket soul;
  Devil (Socket s) { soul=s; }
  public static void main (...) {
    try { ServerSocket losers;
      losers=new ServerSocket(666);
      for(;;){
        Devil me=new Devil(losers.accept());
        me.start();
    } } catch (IOException e) {...}
  }
```

...continued

```
public void run() {
  try { PrintWriter out;
    out=new PrintWriter(soul.getOutputStream());
    out.print("You must come and watch me");
    out.println(" in our Morris dancing group");
    ...
    out.close();
    soul.close();
  } catch (IOException e) {...}
}
```

Genuine Parallelism

So when we start a thread, it runs physically parallel to the other threads?

It could be. If we have sufficient processors and our virtual machine makes use of them...

More likely:

Time sharing

Hugging a resource

Example

```
class MMMMMMMMMM extends Thread {
  private String message;
  public void run() {
    for (int i=0; i<10000; i++)
      System.out.println(message);
  }
  MMMMMMMMMM (String m) {
    message = m; }
}
```

...continued

```
public static void main(...) {
  MMMMMMMMM a,b,c;
  a=new MMMMMMMMM("hello");
  b=new MMMMMMMMM("world!");
  c=new MMMMMMMMM("42");
  a.start(); b.start(); c.start();
  a.join(); b.join(); c.join();
}
```

Side Remark

Why are there the three **`join`** calls at the end?

The main thread of control should be the *last* to terminate. If it runs out of things to do, it should wait for the other threads to finish.

Otherwise, the command shell in which you run the Java program may *fail to recognise* that the program has finished - when it has.

How does this behave?

on my PC
- first 262 lines `hello`
- then 18 times `world!` followed by `42`
- then 743 times `world!`
- etc. (no pattern)

on myrtle
- 997x `hello`
- 10000 lines `42`
- 2998 lines `world!`
- 9003 lines `hello`
- 7002 lines `world!`

that was it!

Question

What would the output have been, had we called **`a.run()`** etc. instead of **`a.start()`** etc.?

Unfair threads

if threads can be scheduled unfairly, how do we ever get behaviour that resembles parallelism?

none if we do not need it, i.e. if we can run the threads one after the other

otherwise threads interact with other threads

they may have to wait for them and block

Example: Thread with result

```
class Computation extends Thread {
  private Object result;
  public Object getResult()
    { return result; }
  public void run() {
    // something complicated
    // that stores something in result
  }
}
```

User

```
class User extends Thread {
  public void run() {
    Computation c=new Computation();
    c.start();
    // do our own stuff
    c.join(); // wait for c to finish
    use(c.getResult());
} }
```

Problems

- what if c computes a result on the hoof, i.e. without terminating?
  - how would we know the result is ready to be collected?
- what happens if the user thread calls `c.getResult()` without a preceding `c.join()`?
- what happens if the field `result` is not defined as **`private`** and we access it directly?

Answers and Half-solutions

- we could implement a "ready to be collected" method; but how do we make this safe?
- the user has to exercise restraint - just put recommendations of good usage into the doc
- different threads accessing the same field directly is problematic; to avoid being outwitted by compiler optimisations use **`volatile`** [better: do not! keep the fields **`private`**!]

SUN's advice on threads

Don't use them!

...unless you really have to!

For threads with result they also provide the class **`SwingWorker`** (not in the jdk). This uses some synchronisation features.