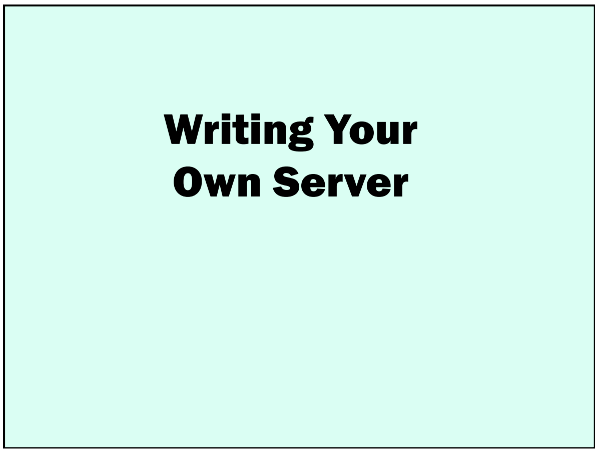
CS209

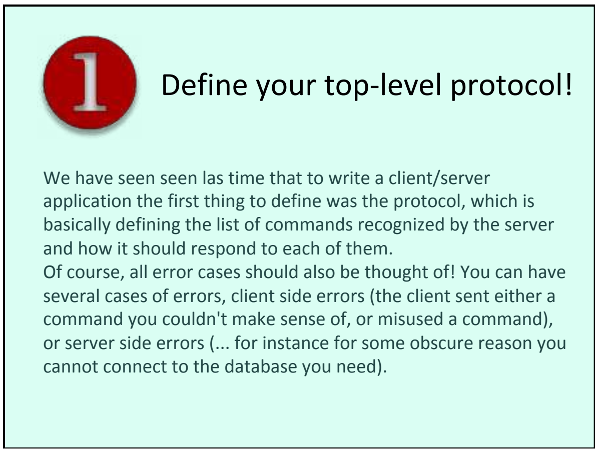
Computer system design and application

Stéphane Faroult
faroult@sustc.edu.cn

Zhao Yao zhaoy6@sustc.edu.cn

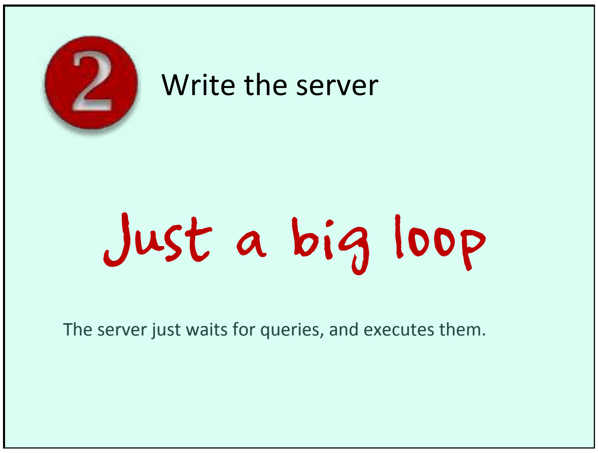


Writing Your Own Server



1 Define your top-level protocol!

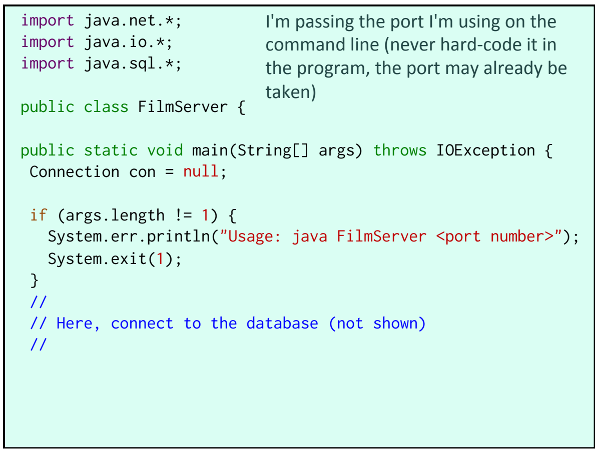
We have seen seen las time that to write a client/server application the first thing to define was the protocol, which is basically defining the list of commands recognized by the server and how it should respond to each of them.
Of course, all error cases should also be thought of! You can have several cases of errors, client side errors (the client sent either a command you couldn't make sense of, or misused a command), or server side errors (... for instance for some obscure reason you cannot connect to the database you need).



2 Write the server

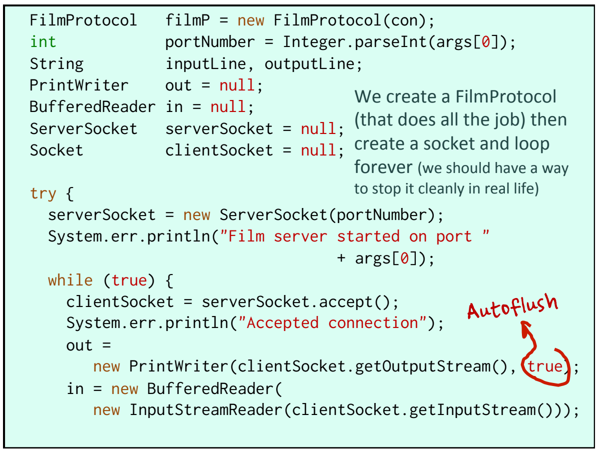
Just a big loop

The server just waits for queries, and executes them.



```
import java.net.*;
import java.io.*;
import java.sql.*;

public class FilmServer {

public static void main(String[] args) throws IOException {
 Connection con = null;

 if (args.length != 1) {
   System.err.println("Usage: java FilmServer <port number>");
   System.exit(1);
 }
 //
 // Here, connect to the database (not shown)
 //
```

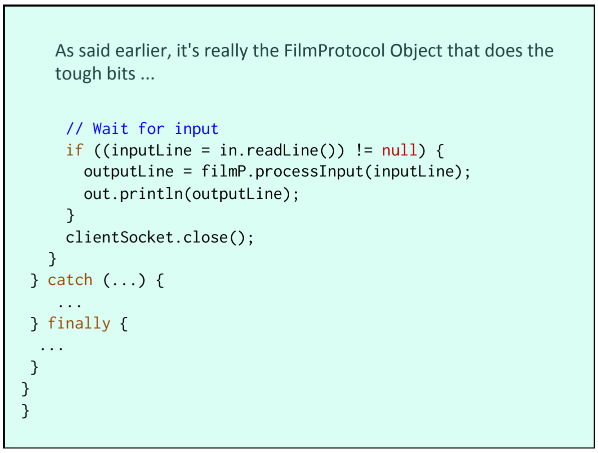
I'm passing the port I'm using on the command line (never hard-code it in the program, the port may already be taken)



```
FilmProtocol   filmP = new FilmProtocol(con);
int            portNumber = Integer.parseInt(args[0]);
String         inputLine, outputLine;
PrintWriter    out = null;
BufferedReader in = null;
ServerSocket   serverSocket = null;
Socket         clientSocket = null;

try {
  serverSocket = new ServerSocket(portNumber);
  System.err.println("Film server started on port "
                               + args[0]);
  while (true) {
    clientSocket = serverSocket.accept();
    System.err.println("Accepted connection");
    out =
       new PrintWriter(clientSocket.getOutputStream(), true);
    in = new BufferedReader(
       new InputStreamReader(clientSocket.getInputStream()));
```

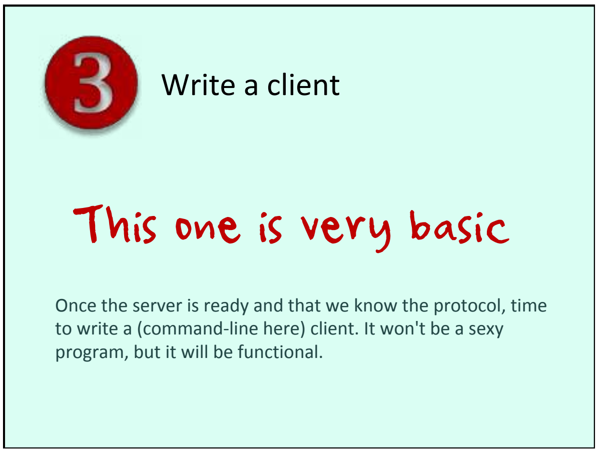
We create a FilmProtocol (that does all the job) then create a socket and loop forever (we should have a way to stop it cleanly in real life)

Autoflush



As said earlier, it's really the FilmProtocol Object that does the tough bits ...

```
    // Wait for input
    if ((inputLine = in.readLine()) != null) {
      outputLine = filmP.processInput(inputLine);
      out.println(outputLine);
    }
    clientSocket.close();
  }
 } catch (...) {
   ...
 } finally {
  ...
 }
}
}
```



3 Write a client

This one is very basic

Once the server is ready and that we know the protocol, time to write a (command-line here) client. It won't be a sexy program, but it will be functional.

```
import java.io.*;
import java.net.*;

public class FilmClient {

  public static void main(String[] args) throws IOException {

   if (args.length != 2) {
     System.err.println(
         "Usage: java FilmClient <host name> <port number>");
     System.exit(1);
   }

   String  hostName = args[0];
   int     portNumber = Integer.parseInt(args[1]);
   boolean loop = true;
```

The client must be told where to connect.

```
   while (loop) {
     try (
       Socket         sock = new Socket(hostName, portNumber);
       PrintWriter    out =
          new PrintWriter(sock.getOutputStream(), true);
       BufferedReader in = new BufferedReader(
          new InputStreamReader(sock.getInputStream()));
     ) {
       BufferedReader stdIn =
         new BufferedReader(new InputStreamReader(System.in));
       String fromServer;
       String fromUser;
```

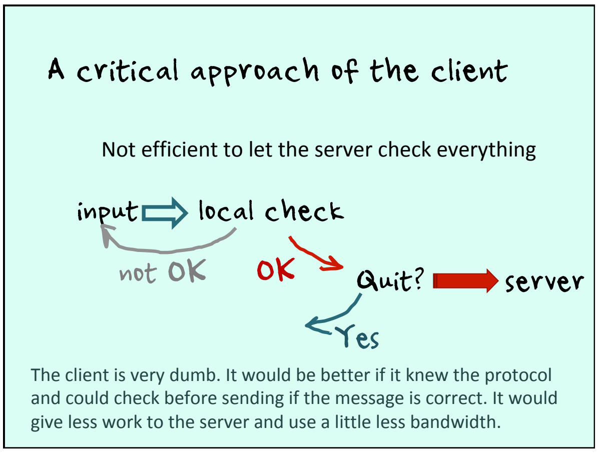
As any query is independent (there is no "session") I'm creating (and closing) one connection for each query. Better suited to my server in that case.

```
       System.out.print("Query> ");
       // read from input
       fromUser = stdIn.readLine();
       // send to server
       out.println(fromUser);
       // read from server
       while ((fromServer = in.readLine()) != null) {
         if (fromServer.length() > 0) {
           System.out.println(fromServer);
         }
         if (fromServer.equals("Goodbye")) {
           loop = false;
           break;
         }
       }
```

When I exit the server sends an acknowledgement. Note that I just expect raw data from the server (an error message would be raw data)

```
     } catch (UnknownHostException e) {
       System.err.println("Don't know about host " + hostName);
       System.exit(1);
     } catch (IOException e) {
       System.err.println("Couldn't get I/O to " + hostName);
       System.exit(1);
     }
   }
 }
}
```

Handling here connection errors.



A critical approach of the client

Not efficient to let the server check everything

input → local check

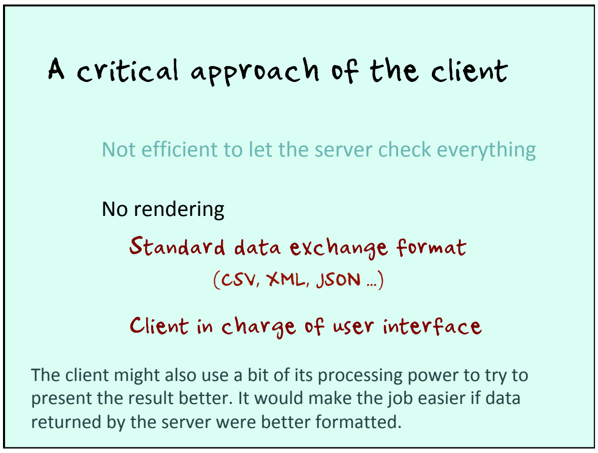
not OK

OK

Quit? → server

Yes

The client is very dumb. It would be better if it knew the protocol and could check before sending if the message is correct. It would give less work to the server and use a little less bandwidth.



A critical approach of the client

Not efficient to let the server check everything

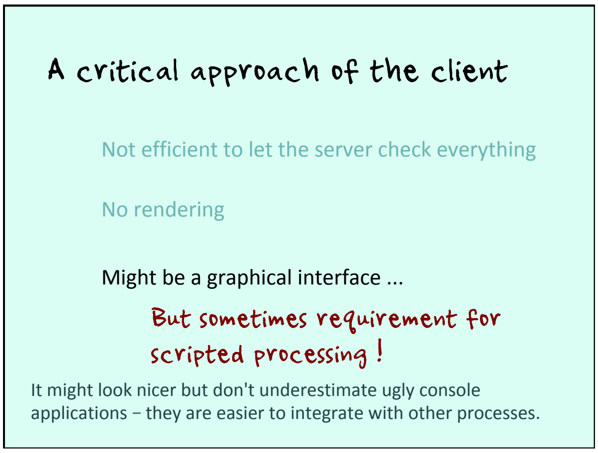
No rendering

Standard data exchange format

(CSV, XML, JSON ...)

Client in charge of user interface

The client might also use a bit of its processing power to try to present the result better. It would make the job easier if data returned by the server were better formatted.



A critical approach of the client

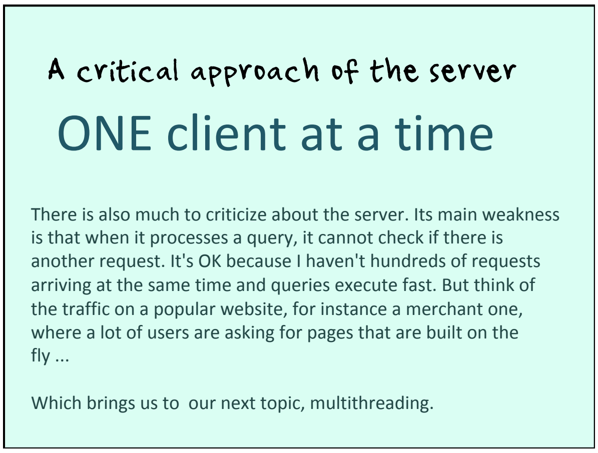
Not efficient to let the server check everything

No rendering

Might be a graphical interface ...

But sometimes requirement for scripted processing !

It might look nicer but don't underestimate ugly console applications – they are easier to integrate with other processes.

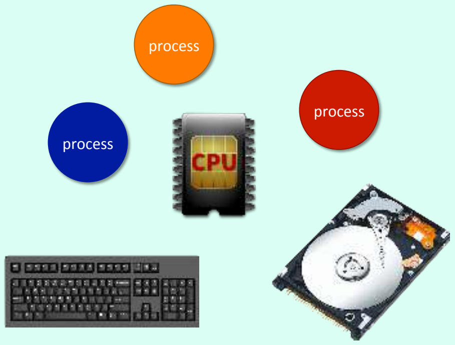


A critical approach of the server

ONE client at a time

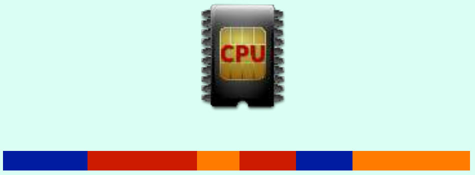
There is also much to criticize about the server. Its main weakness is that when it processes a query, it cannot check if there is another request. It's OK because I haven't hundreds of requests arriving at the same time and queries execute fast. But think of the traffic on a popular website, for instance a merchant one, where a lot of users are asking for pages that are built on the fly ...

Which brings us to our next topic, multithreading.



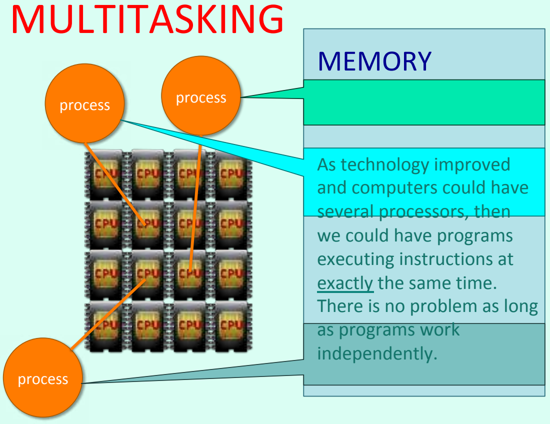
On a computer, several programs run at the same time. They may be using the same resources, but each one in turn.

Very early (in the 1960s) people had the idea of simulating the simultaneous execution of programs on one processor by letting one program execute its instructions for a very short period of time or until it was needing to input/output data, whichever came first. As input/output has always been very slow compared to processor speed, then the context of the program was saved, the context of another program was restored, and this second program was running for another short period of time.



TIME SLICING

MULTITASKING

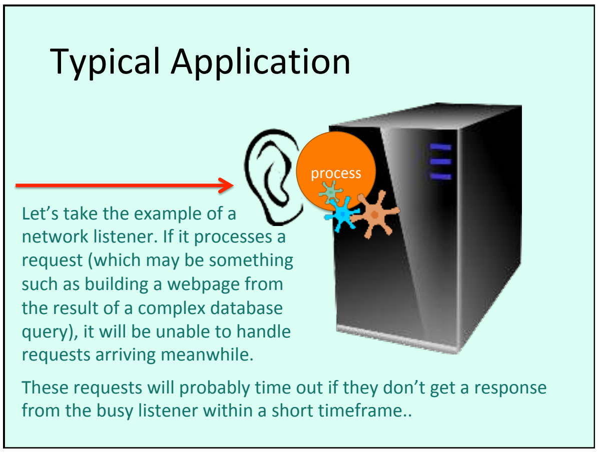


As technology improved and computers could have several processors, then we could have programs executing instructions at exactly the same time. There is no problem as long as programs work independently.

One process = One task

Sometimes you need to do several things at once

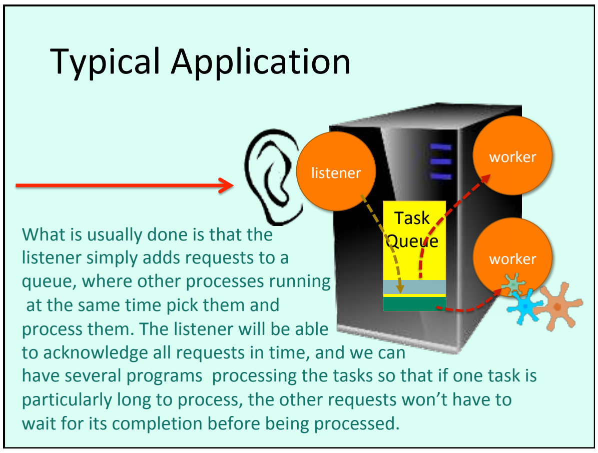
There are many cases where you want independent tasks to be performed within the same program.



Typical Application

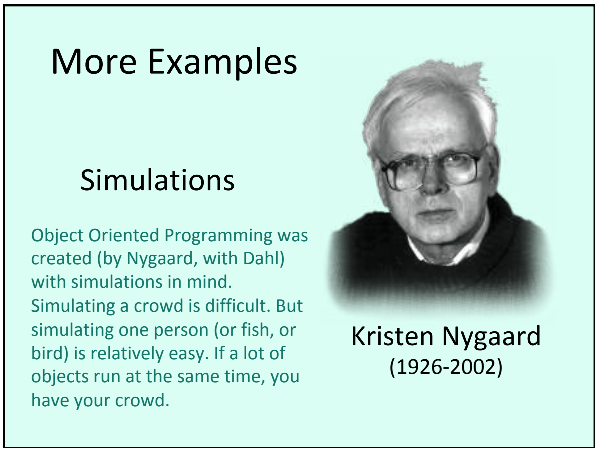
Let's take the example of a network listener. If it processes a request (which may be something such as building a webpage from the result of a complex database query), it will be unable to handle requests arriving meanwhile.

These requests will probably time out if they don't get a response from the busy listener within a short timeframe..



Typical Application

What is usually done is that the listener simply adds requests to a queue, where other processes running at the same time pick them and process them. The listener will be able to acknowledge all requests in time, and we can have several programs processing the tasks so that if one task is particularly long to process, the other requests won't have to wait for its completion before being processed.

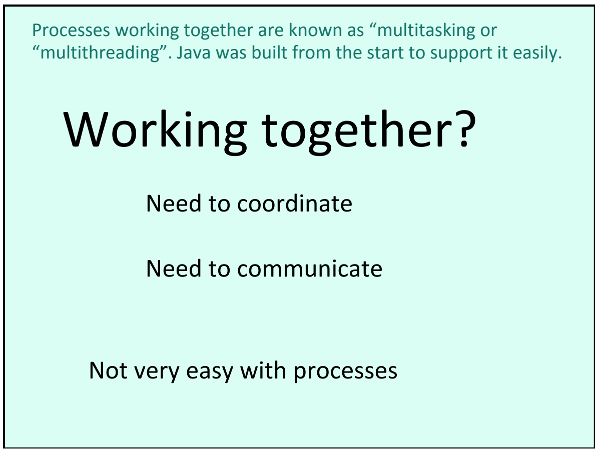


More Examples

Simulations

Object Oriented Programming was created (by Nygaard, with Dahl) with simulations in mind. Simulating a crowd is difficult. But simulating one person (or fish, or bird) is relatively easy. If a lot of objects run at the same time, you have your crowd.

Kristen Nygaard
(1926-2002)



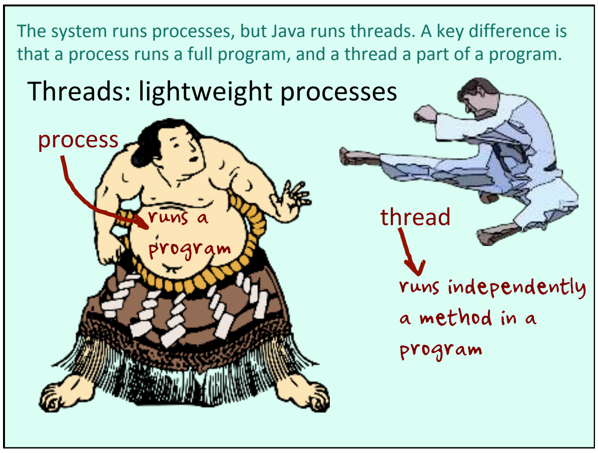
Processes working together are known as "multitasking or "multithreading". Java was built from the start to support it easily.

Working together?

Need to coordinate

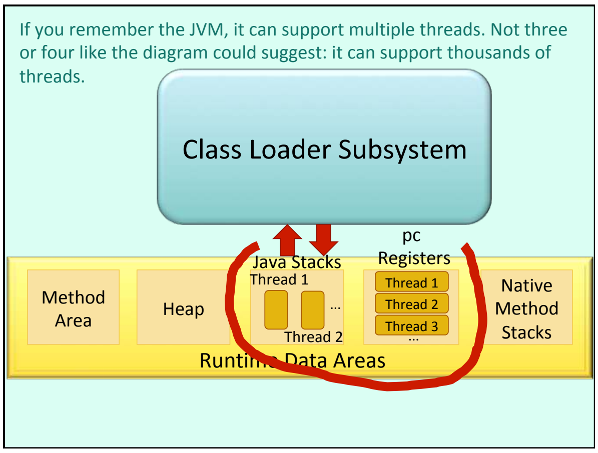
Need to communicate

Not very easy with processes

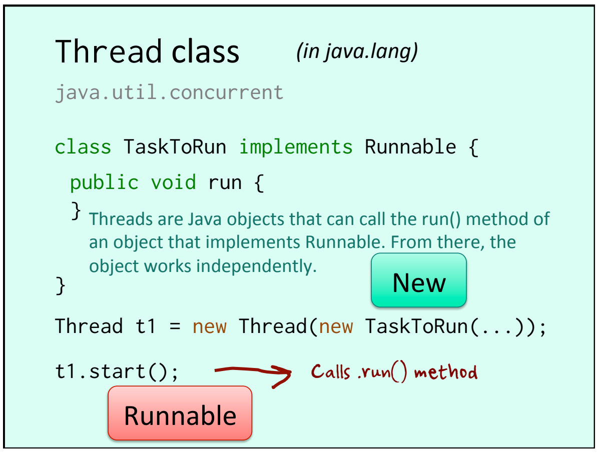


The system runs processes, but Java runs threads. A key difference is that a process runs a full program, and a thread a part of a program.

Threads: lightweight processes



If you remember the JVM, it can support multiple threads. Not three or four like the diagram could suggest: it can support thousands of threads.



Thread class *(in java.lang)*

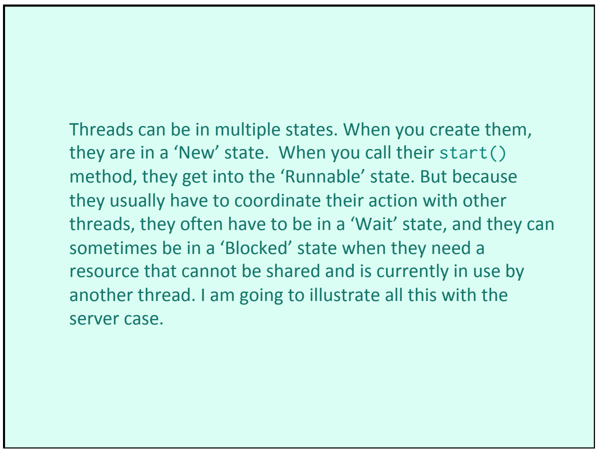
```
java.util.concurrent

class TaskToRun implements Runnable {
  public void run {
  }
}
Thread t1 = new Thread(new TaskToRun(...));
t1.start();
```

Threads are Java objects that can call the run() method of an object that implements Runnable. From there, the object works independently.

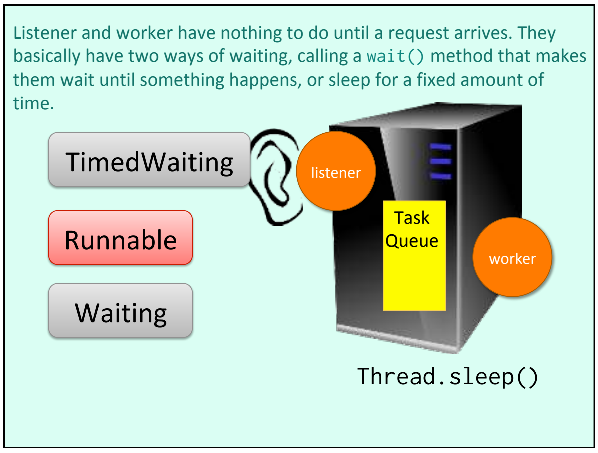
New

Calls .run() method

Runnable



Threads can be in multiple states. When you create them, they are in a 'New' state. When you call their `start()` method, they get into the 'Runnable' state. But because they usually have to coordinate their action with other threads, they often have to be in a 'Wait' state, and they can sometimes be in a 'Blocked' state when they need a resource that cannot be shared and is currently in use by another thread. I am going to illustrate all this with the server case.



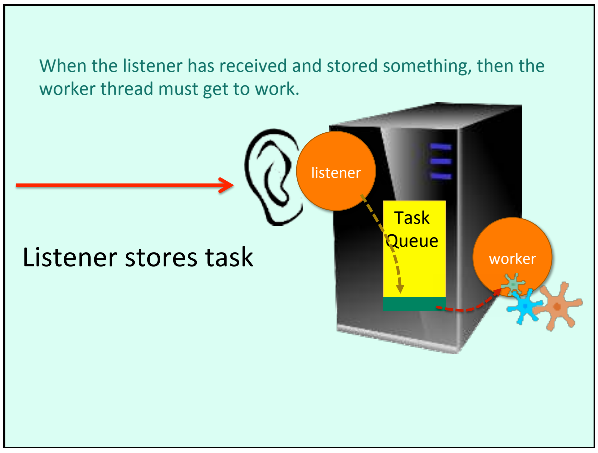
Listener and worker have nothing to do until a request arrives. They basically have two ways of waiting, calling a `wait()` method that makes them wait until something happens, or sleep for a fixed amount of time.

TimedWaiting

Runnable

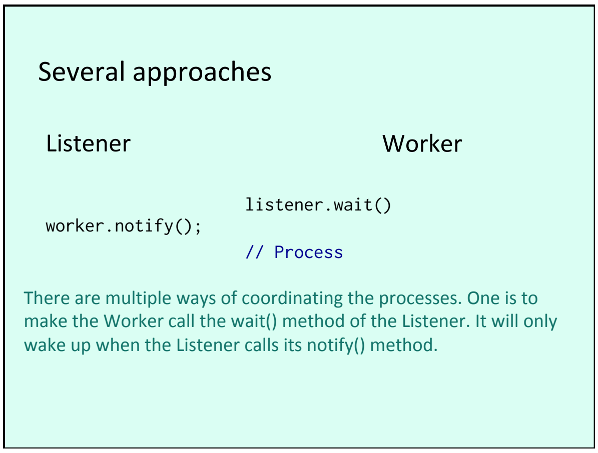
Waiting

`Thread.sleep()`



When the listener has received and stored something, then the worker thread must get to work.

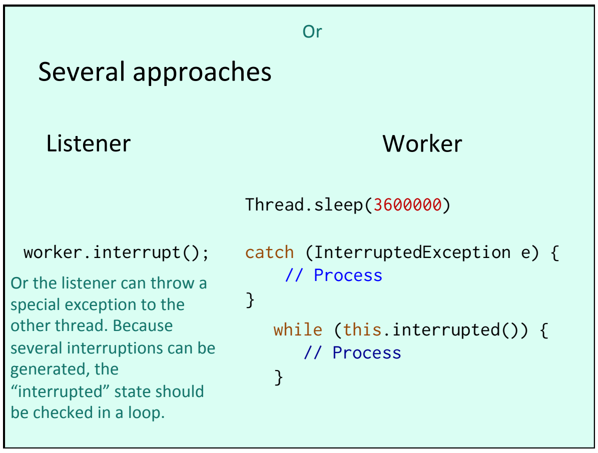
Listener stores task



Several approaches

```
Listener                     Worker

                             listener.wait()
worker.notify();
                             // Process
```

There are multiple ways of coordinating the processes. One is to make the Worker call the wait() method of the Listener. It will only wake up when the Listener calls its notify() method.



Or

Several approaches

```
Listener                     Worker

                             Thread.sleep(3600000)

worker.interrupt();          catch (InterruptedException e) {
                                // Process
                             }
                                while (this.interrupted()) {
                                   // Process
                                }
```

Or the listener can throw a special exception to the other thread. Because several interruptions can be generated, the "interrupted" state should be checked in a loop.

Several approaches Polling

Listener Worker

```
while (true) {
  try {
    Thread.sleep(1000);
    if (something to do) {
      // just do it
    }
  } catch (InterruptedException e) {
    break;
  }
}
```

Or the Worker may ignore the Listener, check by itself, and keep on checking until it's interrupted – which means no more work to expect.

`.interrupt()`

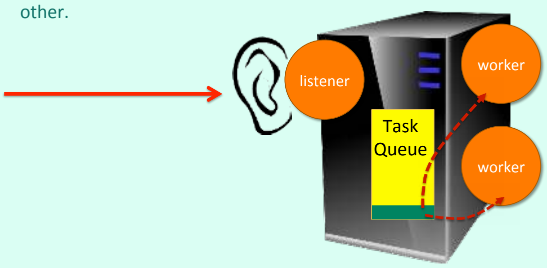
This method is how you can « get the attention » of another thread.

Very useful!

interrupt a thread that works far too long

You usually get exceptions from what you call, and « throw » sends an exception to the caller. Here it' an exception that is thrown to a different thread and comes « from the outside ».

However, coordination is a bit more complicated: we have a shared resource, which is the queue. You want every task to be processed once, not several times by Workers ignorant of each other.

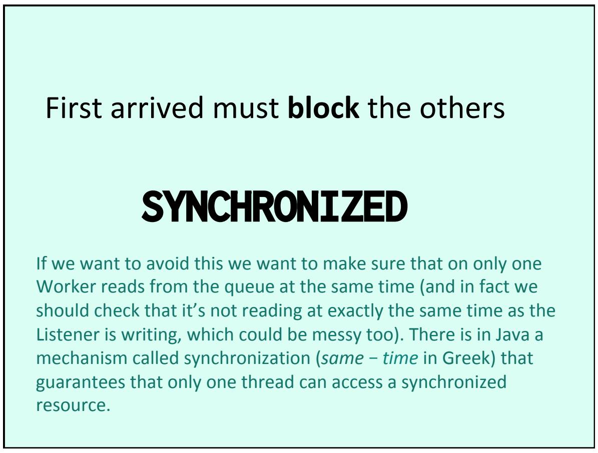


Task = Buy 3,000 XYZ shares for customer ABC

```
$ javac NaiveQueue.java
$ java NaiveQueue 10
-- Main thread starting 10 workers
-- Threads started
W3 processing Task1
W2 processing Task1
W7 processing Task2
W4 processing Task1
W1 processing Task1
W9 processing Task3
W7 processing Task4
W1 processing Task5
W6 processing Task6
W5 processing Task6
...
```

Race condition

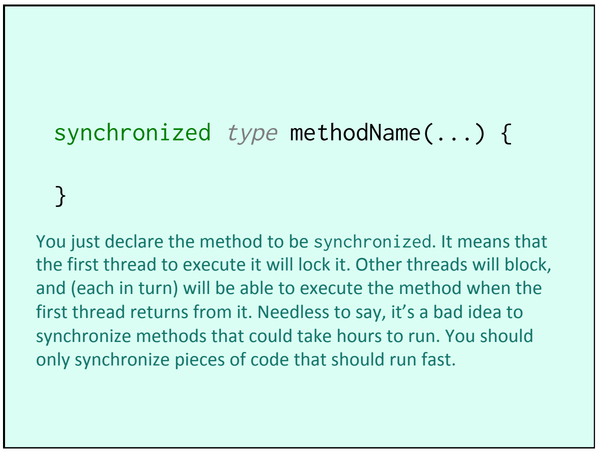
If we aren't careful enough, two (or more) Workers may check the queue at the same time, see the same task, and all process it. It's called a *race condition*.



First arrived must **block** the others

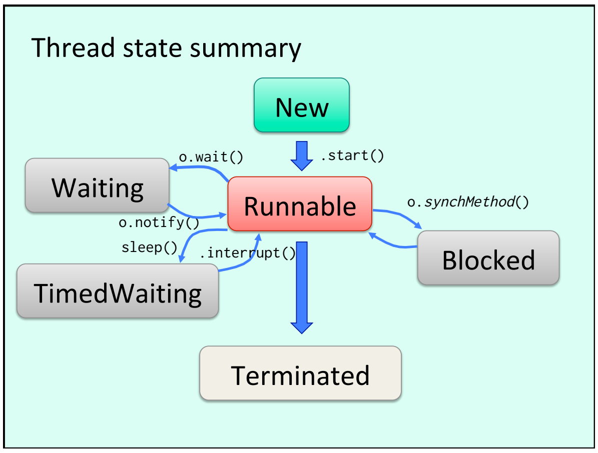
SYNCHRONIZED

If we want to avoid this we want to make sure that on only one Worker reads from the queue at the same time (and in fact we should check that it's not reading at exactly the same time as the Listener is writing, which could be messy too). There is in Java a mechanism called synchronization (*same – time* in Greek) that guarantees that only one thread can access a synchronized resource.



```
synchronized type methodName(...) {

}
```

You just declare the method to be synchronized. It means that the first thread to execute it will lock it. Other threads will block, and (each in turn) will be able to execute the method when the first thread returns from it. Needless to say, it's a bad idea to synchronize methods that could take hours to run. You should only synchronize pieces of code that should run fast.



Thread state summary

New

.start()

Runnable

o.wait()

Waiting

o.notify()

sleep()

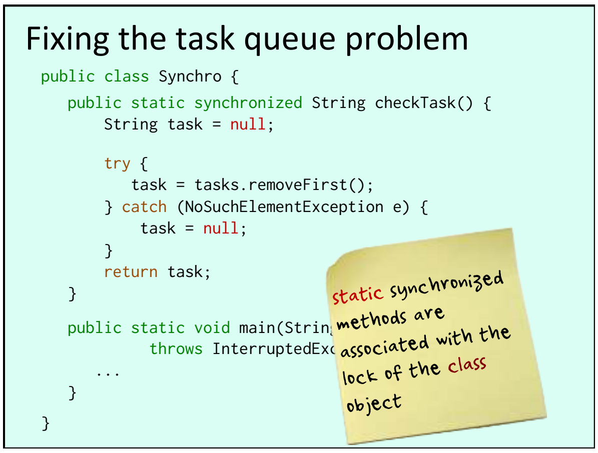
.interrupt()

TimedWaiting

o.*synchMethod*()

Blocked

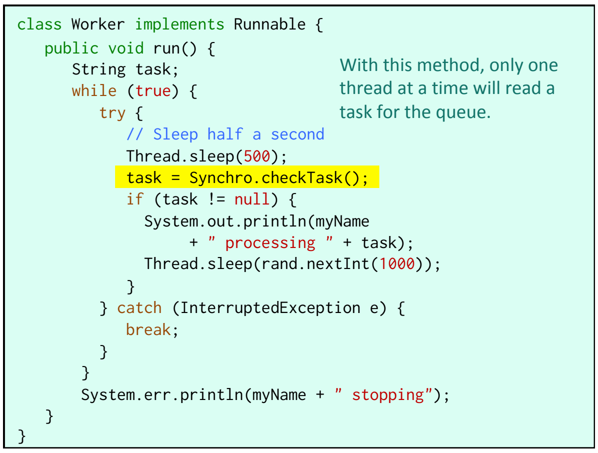
Terminated



Fixing the task queue problem

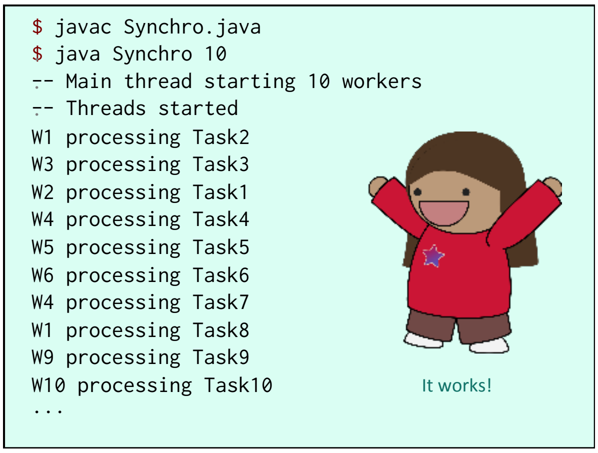
```
public class Synchro {
   public static synchronized String checkTask() {
       String task = null;

       try {
          task = tasks.removeFirst();
       } catch (NoSuchElementException e) {
          task = null;
       }
       return task;
   }

   public static void main(Strin
            throws InterruptedExc
       ...
   }
}
```

static synchronized methods are associated with the lock of the class object



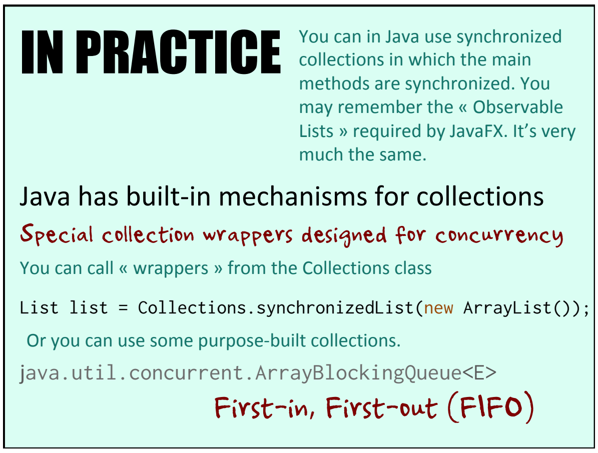
```
class Worker implements Runnable {
   public void run() {
      String task;
      while (true) {
         try {
            // Sleep half a second
            Thread.sleep(500);
            task = Synchro.checkTask();
            if (task != null) {
              System.out.println(myName
                   + " processing " + task);
              Thread.sleep(rand.nextInt(1000));
            }
         } catch (InterruptedException e) {
            break;
         }
       }
       System.err.println(myName + " stopping");
   }
}
```

With this method, only one thread at a time will read a task for the queue.



```
$ javac Synchro.java
$ java Synchro 10
-- Main thread starting 10 workers
-- Threads started
W1 processing Task2
W3 processing Task3
W2 processing Task1
W4 processing Task4
W5 processing Task5
W6 processing Task6
W4 processing Task7
W1 processing Task8
W9 processing Task9
W10 processing Task10
...
```

It works!



IN PRACTICE

You can in Java use synchronized collections in which the main methods are synchronized. You may remember the « Observable Lists » required by JavaFX. It's very much the same.

Java has built-in mechanisms for collections

Special collection wrappers designed for concurrency

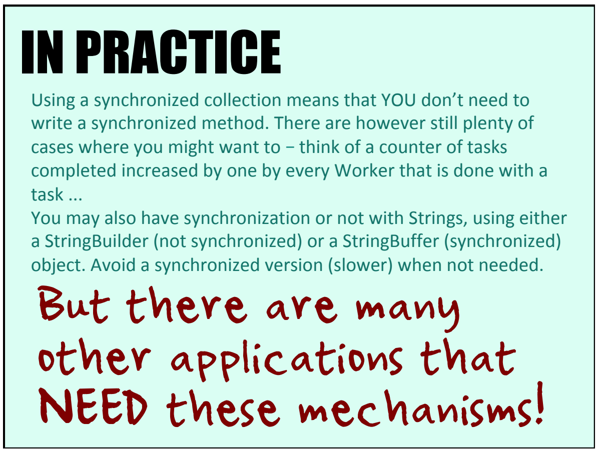
You can call « wrappers » from the Collections class

```
List list = Collections.synchronizedList(new ArrayList());
```

Or you can use some purpose-built collections.

`java.util.concurrent.ArrayBlockingQueue<E>`

First-in, First-out (FIFO)



IN PRACTICE

Using a synchronized collection means that YOU don't need to write a synchronized method. There are however still plenty of cases where you might want to – think of a counter of tasks completed increased by one by every Worker that is done with a task ...

You may also have synchronization or not with Strings, using either a StringBuilder (not synchronized) or a StringBuffer (synchronized) object. Avoid a synchronized version (slower) when not needed.

But there are many other applications that NEED these mechanisms!

WARNING

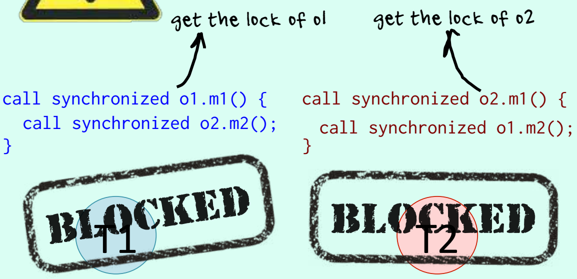
The fact that a thread can be blocked by another one is good, as long as one of the threads can, after a more or less long time, complete what it has to do, then release the locks and unblock any other thread waiting. But you can have problems, because of multiple locks that can be held at the same moment by different thread.

Because locks are associated with objects, **different objects** can call the same synchronized method at the same time!

There would be no problem with a synchronized static method, because then the (single) lock class would be held by only one thread. But when the methods are attached to different objects, the code can be executed at the same time and the locks held by two distinct threads.

This can lead to interesting situations.

Deadlocks

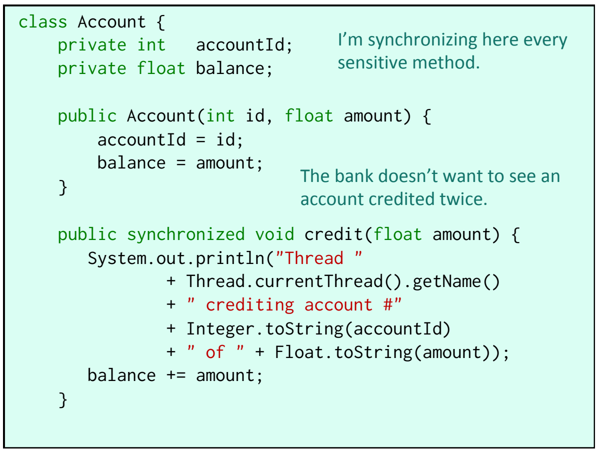


```
call synchronized o1.m1() {
  call synchronized o2.m2();
}
```

```
call synchronized o2.m1() {
  call synchronized o1.m2();
}
```

Test Case

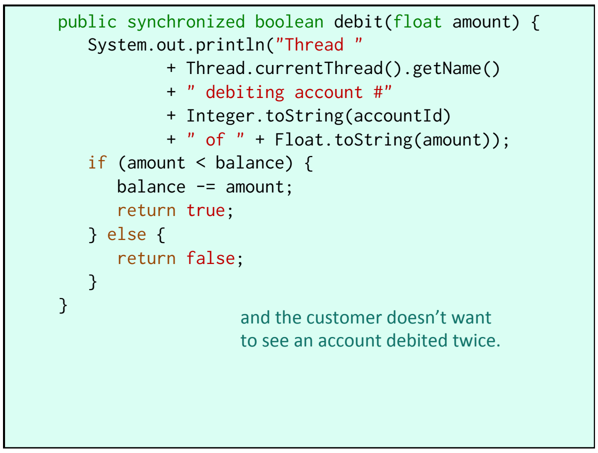
Two people try to transfer money from their account to the other's account at the same time



```
class Account {
    private int   accountId;
    private float balance;

    public Account(int id, float amount) {
        accountId = id;
        balance = amount;
    }

    public synchronized void credit(float amount) {
       System.out.println("Thread "
               + Thread.currentThread().getName()
               + " crediting account #"
               + Integer.toString(accountId)
               + " of " + Float.toString(amount));
       balance += amount;
    }
```

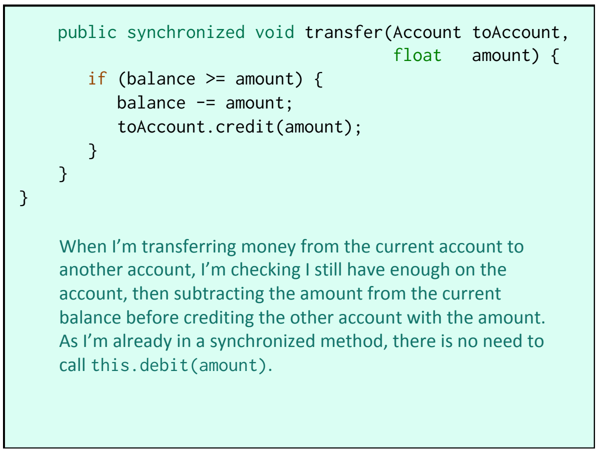
I'm synchronizing here every sensitive method.

The bank doesn't want to see an account credited twice.



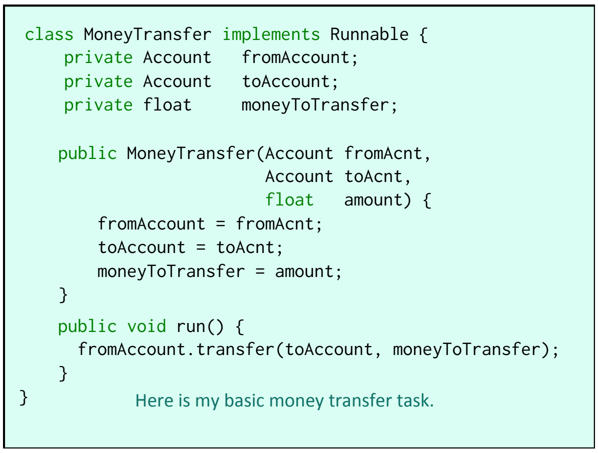
```
    public synchronized boolean debit(float amount) {
       System.out.println("Thread "
               + Thread.currentThread().getName()
               + " debiting account #"
               + Integer.toString(accountId)
               + " of " + Float.toString(amount));
       if (amount < balance) {
          balance -= amount;
          return true;
       } else {
          return false;
       }
    }
```

and the customer doesn't want to see an account debited twice.



```
    public synchronized void transfer(Account toAccount,
                                      float   amount) {
       if (balance >= amount) {
          balance -= amount;
          toAccount.credit(amount);
       }
    }
}
```

When I'm transferring money from the current account to another account, I'm checking I still have enough on the account, then subtracting the amount from the current balance before crediting the other account with the amount. As I'm already in a synchronized method, there is no need to call `this.debit(amount)`.



```
class MoneyTransfer implements Runnable {
    private Account   fromAccount;
    private Account   toAccount;
    private float     moneyToTransfer;

   public MoneyTransfer(Account fromAcnt,
                        Account toAcnt,
                        float   amount) {
       fromAccount = fromAcnt;
       toAccount = toAcnt;
       moneyToTransfer = amount;
   }

   public void run() {
     fromAccount.transfer(toAccount, moneyToTransfer);
   }
}
```

Here is my basic money transfer task.

```
public class BankingTransaction {
    private final static Account account1 =
                             new Account(1, 1000);
    private final static Account account2 =
                             new Account(2, 500);

    public static void main(String[] args) {
        Thread t1 = new Thread(new MoneyTransfer(account1,
                          account2, (float)500));
        Thread t2 = new Thread(new MoneyTransfer(account2,
                          account1, (float)250));

        t1.start();
        t2.start();
    }
}
```

Now if you run this program, sometimes it will work fine, sometimes it will remain stuck. Random problem.

You normally avoid deadlocks by always locking resources in the same order. Here it's a bit different. You might imagine having a third thread watching and interrupting a stuck task ... databases do that.

Check thread termination

```
thr.isAlive()
```

Wait for thread termination

```
thr.join()
```

Sometimes the main thread (that started all the other ones running) must gather work done by the others and kind of finalize operations. It can check if a thread is still running, and if this is the case call its `.join()` method that blocks until the thread has ternminated.

```
for (int i = 0; i < threadCount; i++) {
   thr = new Thread(...):
   threadList.add(thr);
   thr.start();
}
Iterator iter = threadList.iterator();
while (iter.hasNext()) {
   thr = iter.next();
   if (thr.isAlive()) {
     thr.join();
   }
}
// Now use what threads have done
```

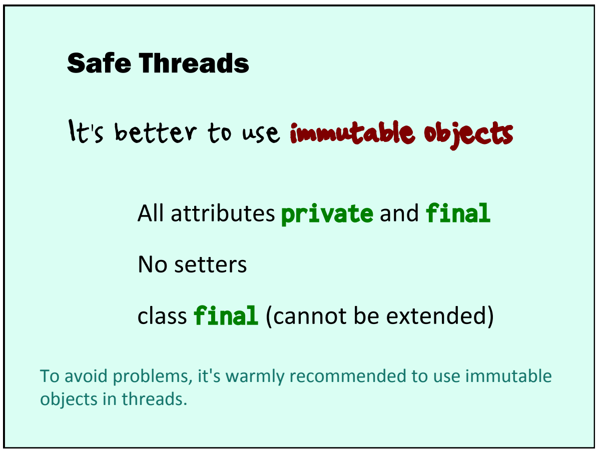
This is a simple example of master thread starting children threads and waiting for the termination of each of them.

It is possible to assign a priority to threads

Starvation

Not locked, but cannot make any progress

For the record, you can increase the priority of some threads but it can be dangerous for the other ones.



Safe Threads

It's better to use **immutable objects**

All attributes **private** and **final**

No setters

class **final** (cannot be extended)

To avoid problems, it's warmly recommended to use immutable objects in threads.