LAB 1 - Socket & RMI based apps

We will develop middleware level code.
The typical example is to use an rpc call in which you make the call to a long running service in a system, which usually repsonds immediately if it has started working. The client can monitor the execution of the service with blocking sycnhrtonous calls (getStatus). When the answer is "finished", the client can have the results. In this cases you have to develop multithreading. In middleware infrastructure, some aspects of multithreading are hidden, so the dev doesn't have to worry about it.

Socket Server

typically in a socket level programming the server will have a min part thta tyupically performs the role of a dispatcher, aka it is in charge of creating a thread for each client that connects. In RPC there is the concept of context, aka the status of the dialogue between a client and the server
It's organized in 3 classes:

- ServerMain,
  - import java.io e java.net beacuse it interacts with the network and java.util for some calsses of the colection framework
  - only one method, main. It should maintain a reference to the threads it is creating, and it does so by using a list of clienthreads. Then it tries to creates a new server socket that is listening on the port specified as argument, and catches any exception. Then initialize a null socket variable. Then an endless loop that tries to lis.accept on the socket variable and breaks if it catches an exception. If ti works it creates a new clientthread on the socket, creates a new thread, it starts the thread and finally adds the clientthread to the list. None of this code has anything to do with the application, it's only network stuff.
- ServerThread, contains most of the application code
  - its a thread so implements runnable
  - needs a boolean running variable and a variable to store the result (array in this case)
  - the method atomicAction contains the critical part of our application. In our case it generates a random integer and adds it to the array. the method is synchronized, which means the operation is safe among the threads, aka prevents race conditions etc.

    - the method isRunning return the status of the server thread (the boolean variable)
    - the method getResult returns the result if running is false, else it returns null
    - the method run sets the boolean running to true and startas a loop to calculate our result (in our case ten random numbers aka ten calls to atomic action). at the end it sets running to false.
- ClientThread (in charge of managing and maintaining the interaction with each client connected)
    - is a thread so the calss implemetns runnable
    - it needs a variable socket to manage the socket that it will receive by the constructor, an object server thread initialized to null and a boolean running variable
    - The constructor simply assingns the socket received to the variable edelcared
    - the method run sets the boolean ruinnign to true, and instantiates a scanner (to read the messages form the socket) and a printwriter to write on the socket(first declared to null and the in a try catch they are created).
    - At this point we have all we need. infinite loop while running and interpret the protocol.
    - in our case we have 3 possible commands
        - start: a new serverthread is created on the empty variable off before, a new thread on the serverthread is created and is started
        - getStatus: answers on the prwintwriter with the status of the serverthread (st.isRuninng= running or finished)
        - getresult: asnwers with the result (which is a series of numbers so they are sent one by one) and then runing is set to false.
    - socket, printwriter and scanner are closed
    - everything is in an try-catch
    - most of the code has nothing to do with the application (hust the runniong loop)

Socket Client

All the code is network code as it doesn't do any computation but only queries the server and UI.

just one method:

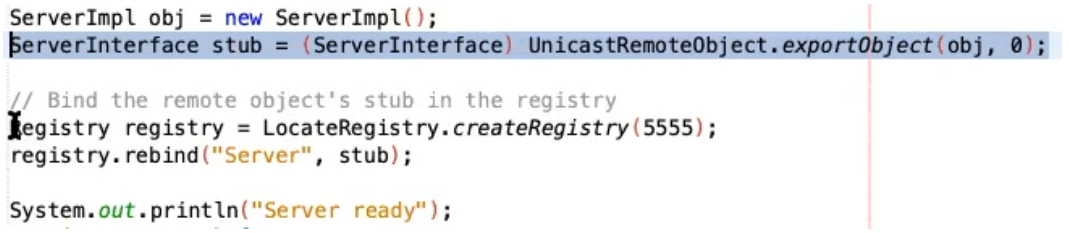
- main starts using the ip adn port of the server, used to crreate a new socket

- need an output stream and a printwriter to write on the socket and a scanner to read the socket.
- sends the start command to the server with thte prwitnwriter and sleep for a bit
- it loops asking for the status until it receives the finished information, at which point it asks for the result with an exit condition (determined by the protocol)-

---

JavaRMI Server

- Interface
  - Middleware focuses on application level code so i deploy the interface without worrying about the protocol but focusing more on a conceptual level, so we basically just define the methods. To make sure the context is kept, when creating a task, it is assigned an id and it used to identify it (the thread), like a label. Note that ServerInterface is declared as public interface ServerInterface extends Remote
- ServerImpl
  - Class that implements ServerInterface, so that the skeleton is generated automatically, and then the application logic is implemented
  - It uses an hashtable to map integers to the threads (serverThread), because we are not using sockets anymore, and keeps track of the total allocated threads
  - The builder allocates the hashtable and sets the number of allocated theads to 0.
  - The startTask method creates a new ServerThread and a new Thread using the serverThread. the serverthread is added to the hashtable with an id, and the thread is started. Total number of threads is incremented and the id is returned. An easy id is the number of alllocated threads.
  - The method isReady returns the status (isRunnig) of the thread that is requested throug its id
  - The method getResults returns the result by using a simple method, so all the marshaling that need the loop and a way to define the end of the result is not needed (the method already returns an array)
  - there is no network level code
- ServerThread
  - Is exactly the same as in socket example
- ServerMain

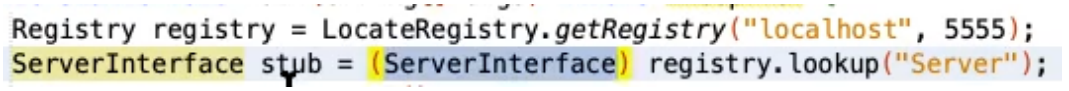
- Creates a new ServerImpl object, then beacuse rmi specific, we specify that a ServerInterface object(stub) is created from the object obj through unicastremoteobject.exportObject, so basically dynamically create the stub.
- After we bind the remote object's stub in the registry, so the server can tell the client that it is available
- 
```
ServerImpl obj = new ServerImpl();
ServerInterface stub = (ServerInterface) UnicastRemoteObject.exportObject(obj, 0);

// Bind the remote object's stub in the registry
Registry registry = LocateRegistry.createRegistry(5555);
registry.rebind("Server", stub);

System.out.println("Server ready");
```
- catch any exception

JavaRMI Client

When working with middleware the interface should be shared between the server and the client, because the client can generate its own version of the stub starting from the interface. So the interface file and the package should be exactly the same.

- ExRMCI (Client)
  - The client is very simple, since it only needs to connect to the server.
  - First of all it connects to the registry, then it looks up the symbolic name of the server and obtains the stub.
  - 
```
Registry registry = LocateRegistry.getRegistry("localhost", 5555);
ServerInterface stub = (ServerInterface) registry.lookup("Server");
```
  - At this point the client can invoke the service. it starts teh procedure by calling stub.startTask() and saing the return value (id).
  - after sleeping a bit, while the computation is not yet ready, then it keeps waiting (while(stub.isReady(id)), otherwise it prints the result that it gets by calling stub.getresults(id)