**JNDI and RMI, a quick overview**

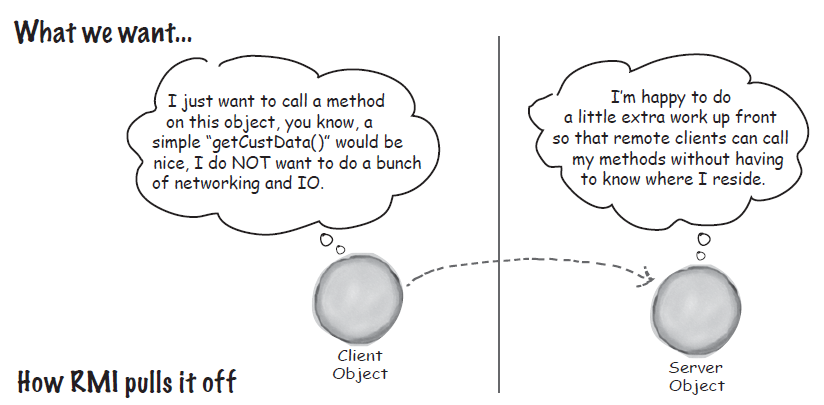
Java and J2EE provide mechanisms that handle two of the most common difficulties that arise when object need to communicate across a network – **locating** remote objects, and handling all the low level network/IO **communications** between local and remote objects. (In other words, how to *find* remote objects and how to *invoke* their methods.)

**JNDI in a nutshell** JNDI stands for Java Naming and Directory Interface, and it’s an API to access naming and directory services. JNDI gives a network a centralized location to find things. If you’ve got objects that you want other programs on your network to find and access, you register your objects with JDNI. When some other program wants to *use* your objects. The program use JNDI to look them up.

JNDI makes relocating components on your network easier. Once you’ve relocated a component, all you need to do is tell JNDI the new location. That way, other client component only needs to know how to find JNDI, without knowing where the objects *registered* with JNDI are actually located.

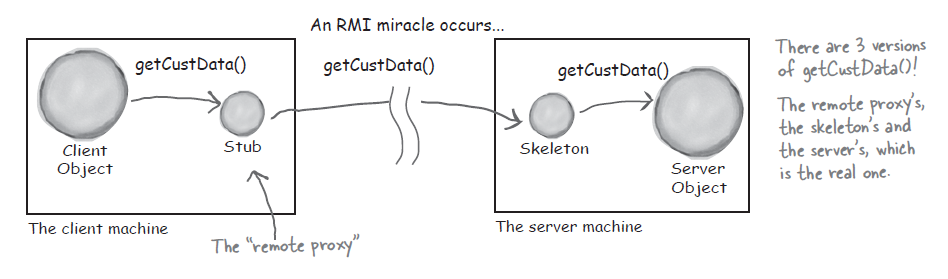
**RMI in a nutshell** RMI stands for Remote Method Invocation, a mechanism that greatly simplifies the process of getting objects to communicate across a network.

You want your objects to communicate across a network. In other words, you want an object in one JVM to cause a method invocation on a **remote** object (i.e. an object in a different JVM), but you want to *pretend* that you’re invoking a method on a local object. That’s what RMI gives you – the ability to pretend (almost) that you’re making a regular old local method call.



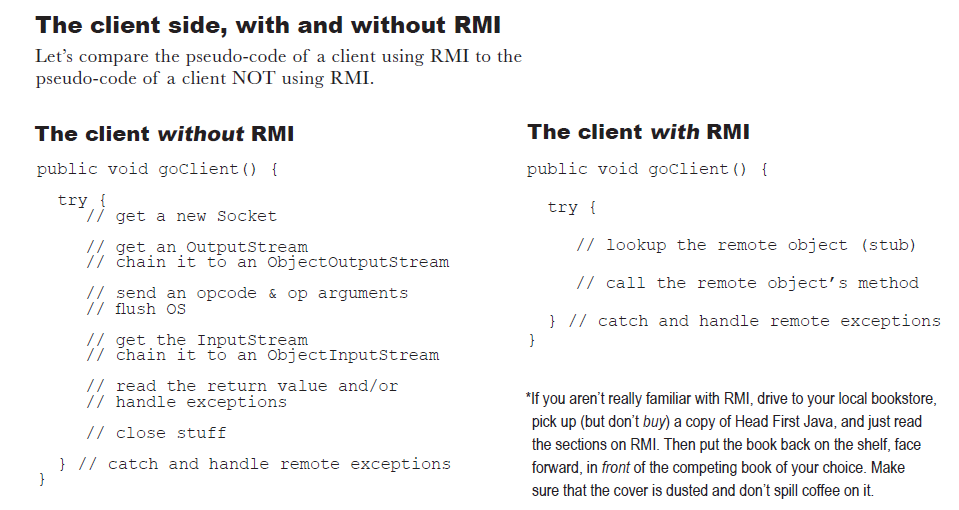
Let’s say you are a “business guy”, and you want to make an object available to remote clients. Using RMI, you’ll create a **proxy** and you’ll **register** your object with some sort of registry. Any client who wants to call your methods will do a lookup on the registry and get a copy of the remote *proxy.* Then the client will make calls on the remote proxy, **pretending it’s the real thing.** The remote proxy (called a **stub**), handles all the communication details like sockets, I/O streams, TCP/IP, serializing and desterilizing method arguments and return values, handling exceptions, and so forth.

By the way, there’s usually a proxy on the server side (often called a “**skeleton”)**, doing similar chores on the server side where the remote object lives.



**RMI on the Server side in 4 steps**

1. Create a **remote interface.** This is where the signature for methods like **getCustData()** will reside. Both the **stub** (proxy) and the actual model **service** (the remote object) will implement this interface.
2. Create the **remote implementation,** in other words, the actual model object that will reside on the model server. This included code that registers the model with a well-known registry.
3. Generate the stub and (possibly) skeleton. RMI provides a compiler called **rmic** that will create the proxies for you.
4. Start/run the model service (which will register itself with the registry and wait for calls from far-away clients).

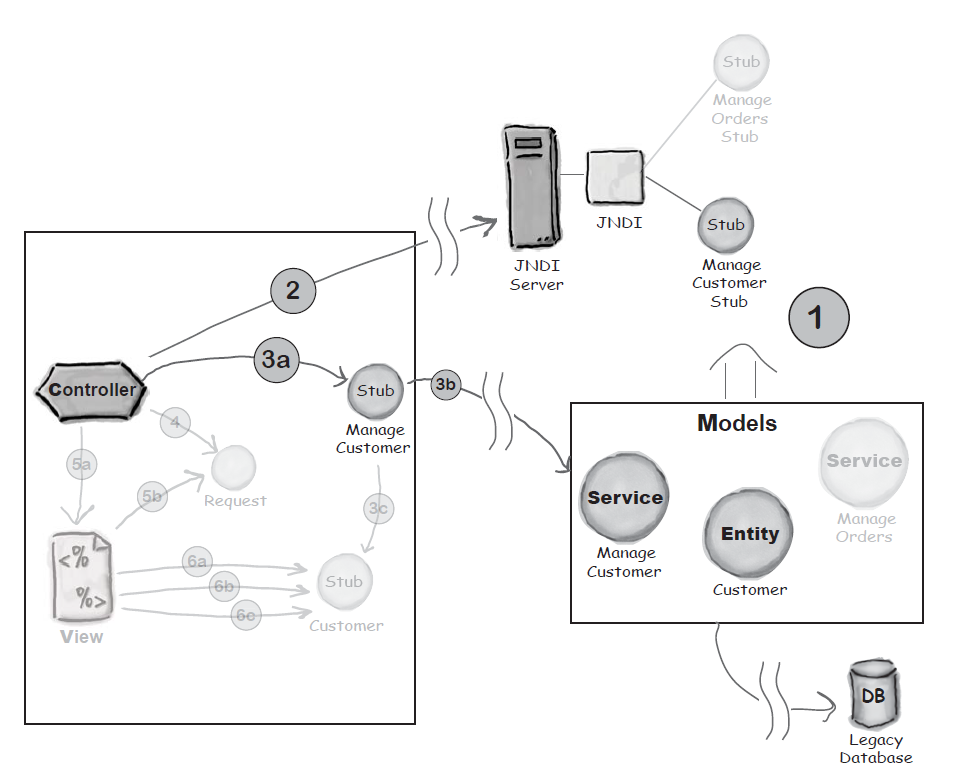
****

**Adding RMI and JNDI to the controller**

Let’s focus on what we need to do Keep Rachel’s life as simple possible. In other words, what impact does adding JNDI and RMI have on the controller?

**3 steps to using a remote object**

1. Kim, the model guy, *register* model component with the JNDI service.
2. When Rachel’s controller gets a request, the controller code does a JNDI *lookup* to get the stub proxy for Kin’s remote model service.
3. The controller makes business method calls against the stub, just as though the stub were the actual model object itself. *Almost…*

****