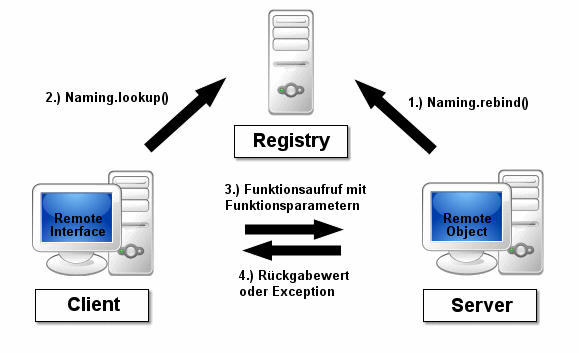
Der Stub kümmert sich Clientseitig um den Transport. Der Stub ersetzt ein Code teil das entweder noch nicht entwickelt ist oder sich auf einem anderen Rechner oder in einem anderen Speicherbereich befindet. Diese Stub-Komponente kann vom Client so angesprochen werden, als wäre die Funktionalität des entfernten Systems lokal vorhanden, statt jedoch diese Funktion selbst auszuführen oder implementiert zu haben, ruft der Stub diese Funktion anhand von netzwerkaufrufen im „entfernen“- System auf.

**Remote Interface**: beschreibt die Funktionen, die auf dem Server zur Verfügung stehen.  
**Remote Object:** stellt das entfernte Objekt dar und liegt auf dem Server. Es implementiert das Remote Interface und das Verhalten der für die Clients zur Verfügung stehenden entfernten Methoden.  
**Remote Reference:** ist eine Referenz auf Remote Objects. Die Clients bekommen die Remote Reference von der RMI Registry.

Vier wichtigste Schritte:

1. Der Server registriert ein Remote Object bei der RMI-Registry unter einem eindeutigen Namen.
2. Der Client sieht bei der RMI-Registry unter diesem Namen nach und bekommt eine Objektreferenz, die seinem Remote Interface entsprechen muss.
3. Der Client ruft eine Methode aus der Objektreferenz auf (dass diese Methode existiert, wird durch das Remote Interface garantiert). Dabei kann ein Objekt einer Klasse X übergeben werden, die der JVM des Servers bisher nicht bekannt ist (das ist möglich, wenn X ein dem Server bekanntes Interface implementiert). In diesem Fall lädt die Server-JVM die Klasse X dynamisch nach, beispielsweise vom Webserver des Client.
4. Die Server-JVM führt die Methode auf dem Remote Object aus, wobei evtl. dynamisch geladener Fremdcode benutzt wird (z. B. Methoden von X), der im Allgemeinen Sicherheitsrestriktionen unterliegt. Dem Client werden die Rückgabewerte dieses Aufrufes gesendet, oder der Client bekommt eine Fehlermeldung (z. B. bei einem Verbindungsabbruch).



 Begriffe:

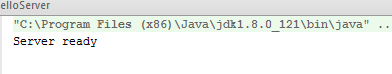
* **Codebase:** In [software development](https://en.wikipedia.org/wiki/Software_development), a codebase (or code base) refers to a whole collection of [source code](https://en.wikipedia.org/wiki/Source_code) that is used to [build](https://en.wikipedia.org/wiki/Software_build) a particular [software system](https://en.wikipedia.org/wiki/Software_system), [application](https://en.wikipedia.org/wiki/Application_software), or [software component](https://en.wikipedia.org/wiki/Software_componentry). Typically, a codebase includes only human-written [source code](https://en.wikipedia.org/wiki/Source_code) files; thus, a codebase usually does not include source code files generated by tools (generated files) or binary library files (object files), as they can be built from the human-written source code. However, it generally does include configuration and property files, as they are the data necessary for the build. This property specifies the locations from which classes that are published by this VM (for example: stub classes, custom classes that implement the declared return type of a remote method call, or interfaces used by a proxy or stub class) may be downloaded. The value of this property is a string in URL format or a space-separated list of URLs that will be the codebase annotation for all classes loaded from the CLASSPATH of (and subsequently marshalled by) this VM. Note: This property must be set correctly in order to dynamically download classes and interfaces using JavaTM Remote Method Invocation (Java RMI). If this property is not set correctly, you will likely encounter exceptions when

attempting to run your server or client.

* **Hostname**: The value of this property represents the host name string that should be associated with remote stubs for locally created remote objects, in order to allow clients to invoke methods on the remote object. The default value of this property is the IP address of the local host, in "dotted-quad" format.
* **useCodebaseOnly**: If this value is true, automatic loading of classes is prohibited except from the local CLASSPATH and from the java.rmi.server.codebase property set on this VM. Use of this property prevents client VMs from dynamically downloading bytecodes from other codebases. This property is ignored in the implementations of 1.2 and 1.2.1 because of a bug.
* **Security.policy**: The java.security.policy property is used to specify the policy file that contains the permissions you intend to grant.
* **SecurityManager**: Um Mißbrauch einzudämmen gibt es SecurityManager. Wird ein SecurityManager gesetzt, so überprüft er, ob für einen bestimmten Vorgang die Genehmigung erteilt worden ist. Ohne Genehmigung endet der Versuch den Vorgang auszuführen mit einer java.security.AccessControlException. Genehmigungen werden erteilt durch einen entsprechenden Eintrag in die Datei java.policy, die sich Im Verzeichnis $JAVA\_HOME/jre/lib/security befindet. Per Default ist in einer JavaApplikation kein SecurityManager gesetzt.   
  Nachteil: Man muss dem Server erlauben eine Verbindung aufzubauen, Anfragen entgegenzunehmen und die Clients auflisten die berechtigt sind einen Verbindung aufzubauen. Kein Zugriff für dynamische IP-Adressen.

RMI Anwendungen sollen in der richtigen Entwicklerumgebung ausgeführt werden damit sie funktionieren. Der Server soll als erstes gestartet werden, dann der Client.

Dokumentation:

1) Server wurde gestartet

2) Client wird ausgeführt und gibt „Hello World!“ zurück

