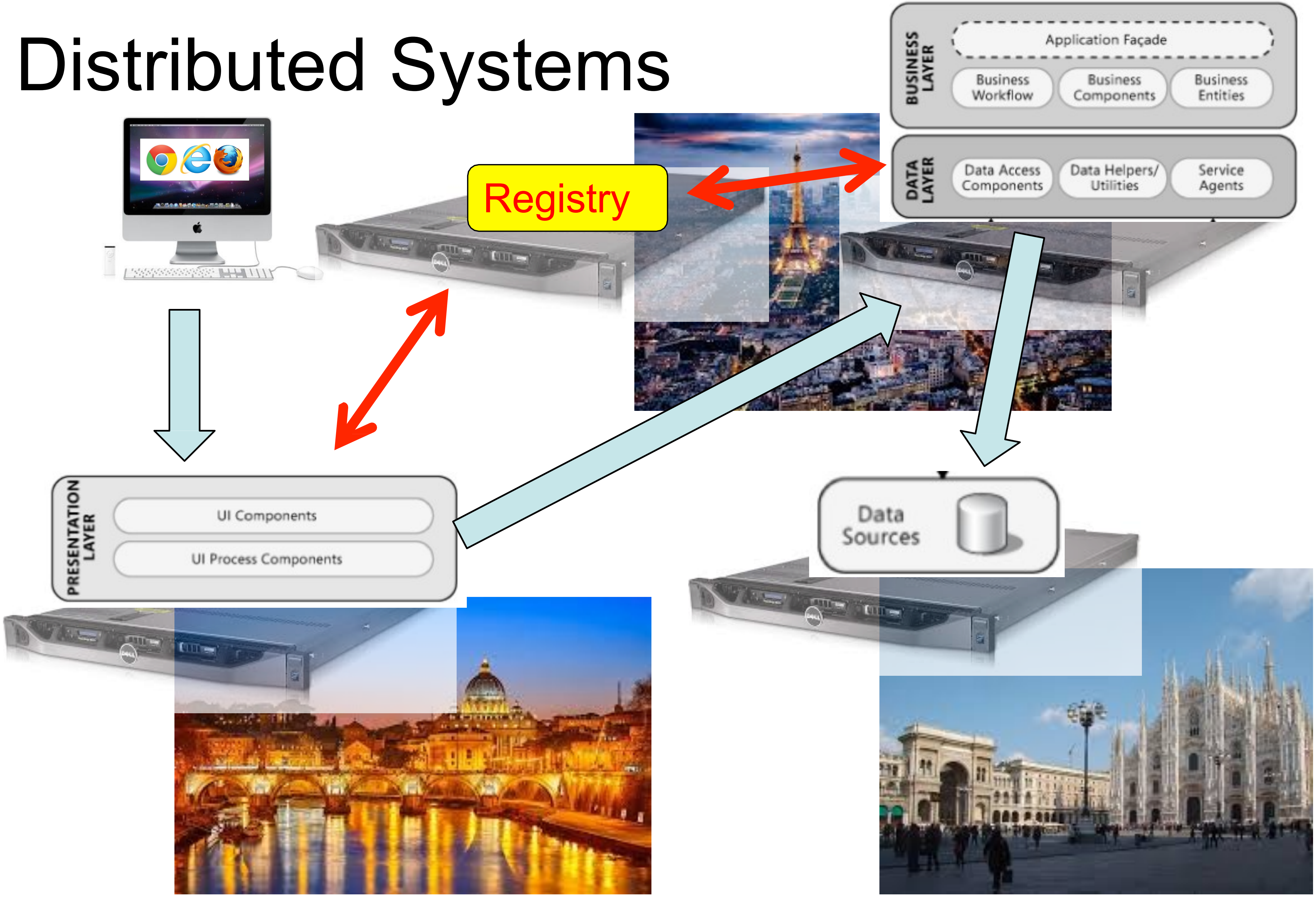
JNDI

Java Naming and Directory Interface

See also:
http://docs.oracle.com/javase/jndi/tutorial/

Distributed Systems



Naming service

A naming service is an entity that

- **associates names with objects.We call this *binding* names to objects.** *This is similar to a telephone company 's associating a person 's name with a specific residence 's telephone number*

- **provides a facility to find an object based on a name.We call this *looking up* or *searching* for an object.***This is similar to a telephone operator finding a person 's telephone number based on that person 's name and connecting the two people.*

In general,a naming service can be used to find any kind of generic object, like a file handle on your hard drive or a printer located across the network.

Directory service

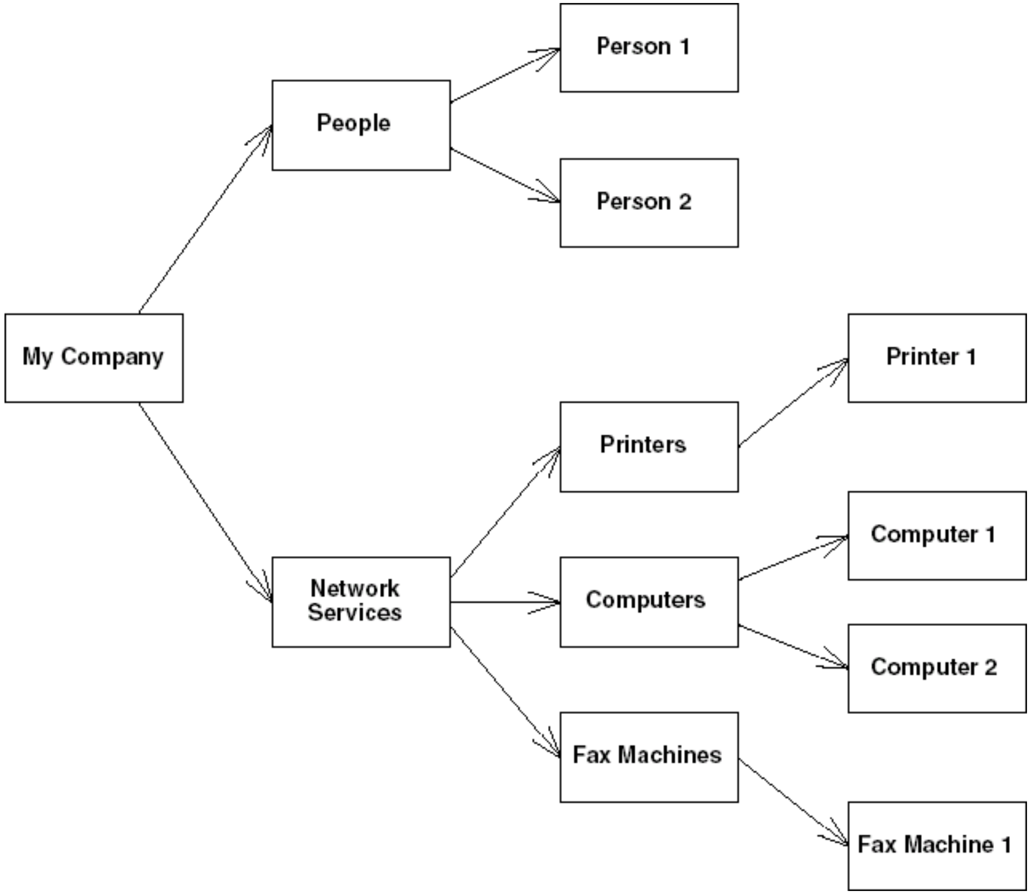
A *directory object* differs from a generic object because you can store *attributes* with directory objects. *For example,you can use a directory object to represent a user in your company.You can store information about that user,like the user 's password,as attributes in the directory object.*

A *directory service* is a naming service that has been extended and enhanced to provide directory object operations for manipulating attributes.

A *directory* is a system of directory objects that are all connected. *Some examples of directory products are Netscape Directory Server and Microsoft 's Active Directory.*

Directory service

Directories are similar to DataBases, except that they typically are organized in a hierarchical tree-like structure. Typically they are optimized for reading.



Examples of Directory services

Azure Active Directory (Microsoft)

Lotus Notes (IBM)

LDAP (Lightweight Directory Access Protocol)

See also https://en.wikipedia.org/wiki/Directory_service

JNDI concepts

JNDI is a system for Java-based clients to interact with naming and directory systems. JNDI is a bridge over naming and directory services, that provides one common interface to disparate directories.

Users who need to access an LDAP directory use the same API as users who want to access an NIS directory or Novell's directory. All directory operations are done through the JNDI interface, providing a common framework.

JNDI advantages

-You only need to learn a single API to access all sorts of directory service information, such as security credentials, phone numbers, electronic and postal mail addresses, application preferences, network addresses, machine configurations, and more.

-JNDI insulates the application from protocol and implementation details.

-You can use JNDI to read and write whole Java objects from directories.

- You can link different types of directories, such as an LDAP directory with an NDS directory, and have the combination appear to be one large, federated directory.

JNDI advantages

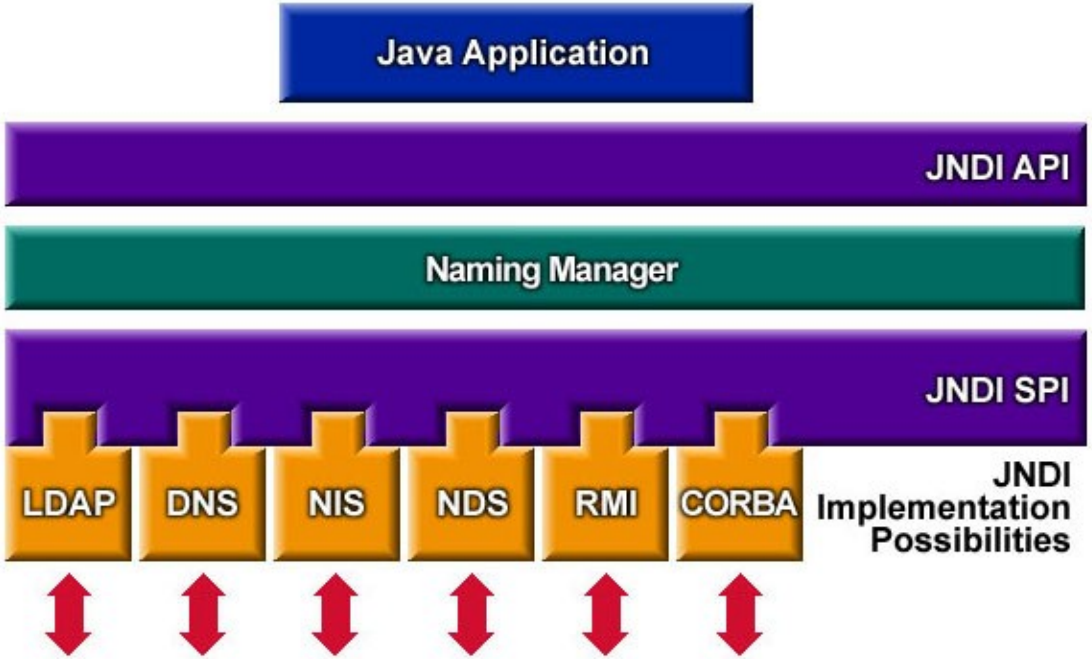
Applications can store factory objects and configuration variables in a global naming tree using the JNDI API.

JNDI, the Java Naming and Directory Interface, provides a global memory tree to store and lookup configuration objects. JNDI will typically contain configured Factory objects.

JNDI lets applications cleanly separate configuration from the implementation. The application will grab the configured factory object using JNDI and use the factory to find and create the resource objects.

In a typical example, the application will grab a database DataSource to create JDBC Connections. Because the configuration is left to the configuration files, it's easy for the application to change databases for different customers.

JNDI Architecture



JNDI concepts

An ***atomic name*** *is a simple,basic,indivisible component of a name.For example,in the string /etc/fstab ,etc and fstab are atomic names.*

A ***binding*** *is an association of a name with an object.*

A ***context*** *is an object that contains zero or more bindings. Each binding has a distinct atomic name. Each of the mtab and exports atomic names is bound to a file on the hard disk.*

A ***compound name*** *is zero or more atomic names put together. e.g. the entire string /etc/fstab is a compound name. Note that a compound name consists of multiple bindings.*

JNDI names

JNDI names look like URLs.
A typical name for a database pool is java:comp/env/jdbc/test.
The java: scheme is a memory-based tree. comp/env is the standard location for Java configuration objects and jdbc is the standard location for database pools.

Examples

java:comp/env	*Configuration environment*
java:comp/env/jdbc	*JDBC DataSource pools*
java:comp/env/ejb	*EJB remote home interfaces*
java:comp/env/cmp	*EJB local home interfaces (non-standard)*
java:comp/env/jms	*JMS connection factories*
java:comp/env/mail	*JavaMail connection factories*
java:comp/env/url	*URL connection factories java:comp/*
UserTransaction	*UserTransaction interface*

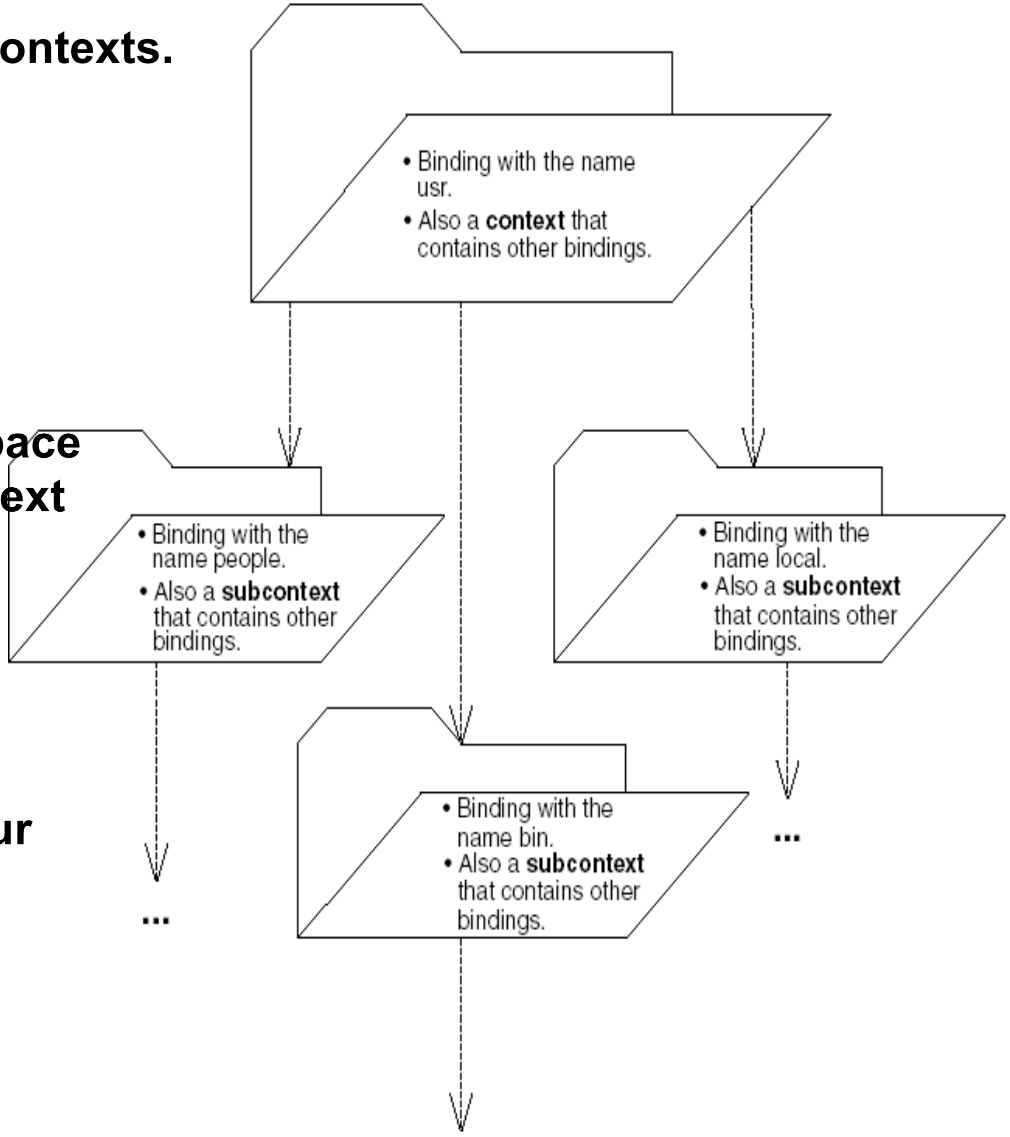
Contexts and Subcontexts

A *naming system* is a connected set of contexts.

A *namespace* is all the names contained within naming system.

The starting point of exploring a namespace is called an *initial context.* An initial context is the first context you happen to use.

To acquire an initial context, you use an *initial context factory*.
An initial context factory basically *is* your JNDI driver.



• Binding with the name usr.
• Also a **context** that contains other bindings.
• Binding with the name people.
• Also a **subcontext** that contains other bindings.
• Binding with the name local.
• Also a **subcontext** that contains other bindings.
• Binding with the name bin.
• Also a **subcontext** that contains other bindings.
...
...

Acquiring an initial context

When you acquire an initial context, you must supply the necessary information for JNDI to acquire that initial context.

For example, if you're trying to access a JNDI implementation that runs within a given server, you might supply:

- *The IP address of the server*
- *The port number that the server accepts*
- *The starting location within the JNDI tree*
- *Any username/password necessary to use the server*

Acquiring an initial context

```
package examples;

public class InitCtx {
   public static void main(String args[]) throws Exception {
      // Form an Initial Context
      javax.naming.Context ctx =
            new javax.naming.InitialContext();
      System.err.println("Success!");
      Object result = ctx.lookup("PermissionManager");
   }
}
```

```
java
-Djava.naming.factory.initial=org.jnp.interfaces.NamingContextFactory
-Djava.naming.provider.url=jnp://193.205.194.162:1099
-Djava.naming.factory.url.pkgs=org.jboss.naming:org.jnp.interfaces
examples.InitCtx
```

Acquiring an initial context

java.naming.factory.initial: The name of the environment property for specifying the initial context factory to use. The value of the property should be the fully qualified class name of the factory class that will create an initial context.

java.naming.provider.url: The name of the environment property for specifying the location of the service provider the client will use. The NamingContextFactory class uses this information to know which server to connect to. The value of the property should be a URL string

Everything but the host component is optional. The following examples are equivalent because the default port value(on JBOSS) is 4447 (used to be 1099).
remote://www.jboss.org:4447/
www.jboss.org:4447

used to be: jnp://www.jboss.org:1099/

Acquiring an initial context

ja.va.naming.factory.url.pkgs:
The name of the environment property for specifying the list of package prefixes to use when loading in URL context factories. The value of the property should be a colon-separated list of package prefixes for the class name of the factory class that will create a URL context factory.
For the JBoss JNDI provider this must be
org.jboss.ejb.client.naming

(used to be: org.jboss.naming:org.jnp.interfaces).
This property is essential for locating the remote: and java: URL context factories of the JBoss JNDI provider.

Operations on a JNDI context

list() *retrieves a list of contents available at the current context.This typically includes names of objects bound to the JNDI tree,as well as subcontexts.*

lookup() *moves from one context to another context,such as going from c:\ to c:\windows. You can also use lookup()to look up objects bound to the JNDI tree.The return type of lookup()is JNDI driver specific.*

rename() *gives a context a new name*

Operations on a JNDI context

createSubcontext()*creates a subcontext from the current context,such as creating c:\foo \bar from the folder c:\foo.*

destroySubcontext()*destroys a subcontext from the current context,such as destroying c:\foo \bar from the folder c:\foo.*

bind()*writes something to the JNDI tree at the current context.As with lookup(),JNDI drivers accept different parameters to bind().*

rebind()*is the same operation as bind,except it forces a bind even if there is already something in the JNDI tree with the same name.*

JNDI Examples

Accessing rmiregistry

Using JNDI to access rmiregisty

see http://docs.oracle.com/javase/8/docs/technotes/guides/jndi/jndi-rmi.html

```
package jndiaccesstormiregistry;

import java.util.Properties;
import javax.naming.CompositeName;
import javax.naming.Context;
import javax.naming.InvalidNameException;
import javax.naming.LinkRef;
import javax.naming.NamingException;
import javax.naming.directory.InitialDirContext;

public class Demo {

    public static void main(String[] args) {
        // Identify service provider to use
        Properties env = new Properties();
        env.put(Context.INITIAL_CONTEXT_FACTORY,
                "com.sun.jndi.rmi.registry.RegistryContextFactory");
        env.put(Context.PROVIDER_URL, "rmi://localhost:1099");
```

```
private static void perr(Exception ex, String message) {
        System.out.println(message);
        ex.printStackTrace();
        System.exit(1);
   }
```

Using JNDI to access rmiregisty

```
    CompositeName cn=null;
    try { cn = new CompositeName("foo"); }
    catch (InvalidNameException ex) { perr(ex,"Invalid name!"); }
    LinkRef lr=new LinkRef(cn);
    Context ctx=null;
    try { ctx = new InitialDirContext(env);}
    catch (NamingException ex) { perr(ex,"Invalid InitialDirContext!");}
    String name= "myVar3";
    try { Object o=ctx.lookup(name);}
    catch (NamingException ex) {
        System.out.println(name+" is not registered");
        try { ctx.bind(name,lr); }
        catch (NamingException ex1) { perr(ex,"Unable to bind "+name);}
    }
    LinkRef result=null;
    try { result = (LinkRef)ctx.lookup(name) ;}
    catch (NamingException ex) { perr(ex,"Unable to lookup "+name);}
    try { System.out.println(result.getLinkName()); }
    catch (NamingException ex) {perr(ex,"Unable to get name from LinkRef ");}
    try { ctx.close();}
    catch (NamingException ex) {perr(ex,"Error on close");}
}}
```

create the object to be stored: in this case a (storable) type of String

acquire the context

if the name is not yet registered, register it

look up the name

print its value

close the connection

Using JNDI to access rmiregisty

NOTE: we are forcing rmiregistry to do something it wasn't designed for (storing strings)

rmiregistry is FLAT – no subcontexts!

An interesting additional reading about rmiregistry:

http://www.drdobbs.com/jvm/a-remote-java-rmi-registry/212001090?pgno=1

JNDI Examples

Accessing LDAP

A JNDI-LDAP example

```
package jndiaccesstoldap;
import javax.naming.Context;
import javax.naming.directory.InitialDirContext;
import javax.naming.directory.DirContext;
import javax.naming.directory.Attributes;
import javax.naming.NamingException;
import java.util.Hashtable;
public class Getattr {
   public static void main(String[] args) {
      // Identify service provider to use
      Hashtable env = new Hashtable(11);
      env.put(Context.INITIAL_CONTEXT_FACTORY,
         "com.sun.jndi.ldap.LdapCtxFactory");
   //env.put(Context.PROVIDER_URL, "ldap://ldap.unitn.it:389/o=JNDITutorial");
   env.put(Context.PROVIDER_URL, "ldap://ldap.unitn.it:389/o=personale");
```

```
      try {
         // Create the initial directory context
         DirContext ctx = new InitialDirContext(env);

         // Ask for all attributes of the object
         Attributes attrs = ctx.getAttributes("cn=Ronchetti Marco");

         // Find the surname ("sn") and print it
         System.out.println("sn: " + attrs.get("sn").get());

         // Close the context when we're done
         ctx.close();
      } catch (NamingException e) {
         System.err.println("Problem getting attribute: " + e);
}}}
```

A JNDI example on an open LDAP

```
static void list(DirContext ctx, String listKey) throws Exception {
        NamingEnumeration<NameClassPair> cp = ctx.list(listKey);
        while (cp.hasMore()) {
            System.out.println(cp.next());
        }
        System.out.println("=================" );
    }
```

```
public class Demo {

    public static void main(String[] args) throws Exception {
        // Identify service provider to use
        Properties env = new Properties();
        env.put(Context.INITIAL_CONTEXT_FACTORY,"com.sun.jndi.ldap.LdapCtxFactory");
        env.put(Context.PROVIDER_URL, "ldap://ldap.virginia.edu");
        // Create the initial directory context
        DirContext ctx = new InitialDirContext(env);
        list(ctx,"o=University of Virginia,c=US");
        DirContext ctx1 = (DirContext) ctx.lookup("o=University of Virginia,c=US");
        list(ctx1,"ou=Arts & Sciences Graduate");
        DirContext ctx2 = (DirContext) ctx1.lookup("ou=Arts & Sciences Graduate");
        list(ctx2,"ou=casg");
        DirContext ctx3 = (DirContext) ctx2.lookup("ou=casg");
        Attributes attrs = ctx3.getAttributes("cn=Amy Marion Coddington (amc4gc)");
        NamingEnumeration<? extends Attribute> ne = attrs.getAll();
        while (ne.hasMore()) {
            System.out.println(ne.next());
        }
        ctx.close();
    }
}
```

http://its.virginia.edu/network/publicldap.html