JAVA Naming and Directory Interface (JNDI)

The Java Naming and Directory Interface (JNDI) is an API that supports accessing naming and directory services in Java programs. The purpose of a naming service is to associate names with objects and provide a way to access objects based on their names. You should be familiar with naming systems; you use them every day when you browse the filesystem on your computer or surf the Web by typing in a URL. Objects in a naming system can range from files in a filesystem and names located in Domain Name System (DNS) records, to Enterprise JavaBeans (EJB) components in an application server and user profiles in an LDAP (Lightweight Directory Access Protocol) directory. If you want to use Java to write an application such as a search utility, a network-enabled desktop, an application launcher, an address book, a network management utility, or a class browser--in short, anything that accesses objects in a naming system--JNDI is a good candidate for writing that application. As its name implies, JNDI doesn't just deal with naming services. JNDI also encompasses directory services, which are a natural extension of naming services. The primary difference between the two is that a directory service allows the association of attributes with objects, such as an email address attribute for a user object, while a naming service does not. Thus, with a directory service, you can access the attributes of objects and search for objects based on their attributes. You can use JNDI to access directory services like LDAP and Novell Directory Services (NDS) directories. As an enterprise programmer, you will most likely use JNDI to access Enterprise JavaBeans; the EJB specification requires that you use JNDI to locate EJB components on the network. But you can also use JNDI to find remote objects in an RMI registry on a remote server. And most enterprise Java suppliers, such as BEA WebXPress, IBM, Novell, Sun, and SCO, support JNDI access to their naming systems.

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.
Other URL schemes are allowed as well, including RMI (rmi://localhost:1099) and LDAP. Many applications, though will stick to the java:comp/env tree.
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

There are three commonly used levels of naming scope in JBoss:
names under java:comp,
names under java:,
any other name.
java:comp context and its subcontexts are only available to the application component associated with that particular context.
Subcontexts and object bindings directly under java: are only visible within the JBoss server virtual machine and not to remote clients.
Any other context or object binding is available to remote clients, provided the context or object supports serialization.
An example of where the restricting a binding to the java: context is useful would be a javax.sql.DataSource connection factory that can only be used inside of the JBoss server where the associated database pool
resides. On the other hand, an EJB home interface would be boung to a globally visible name that should accessible by remote client.

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.

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.

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.

JNDI Architecture

The architecture of JNDI is somewhat like the JDBC architecture, in that both provide a standard protocol-independent API built on top of protocol-specific driver or provider implementations. This layer insulates an application from the actual data source it is using, so, for example, it doesn't matter whether the application is accessing an NDS or LDAP directory service. The JNDI architecture includes both an application programming interface (API) and a service provider interface (SPI), as shown in Figure 6-1. A Java application uses the JNDI API to access naming and directory services, primarily through the Context and DirContext interfaces. The JNDI API is defined in the javax.naming and javax.naming.directory packages.

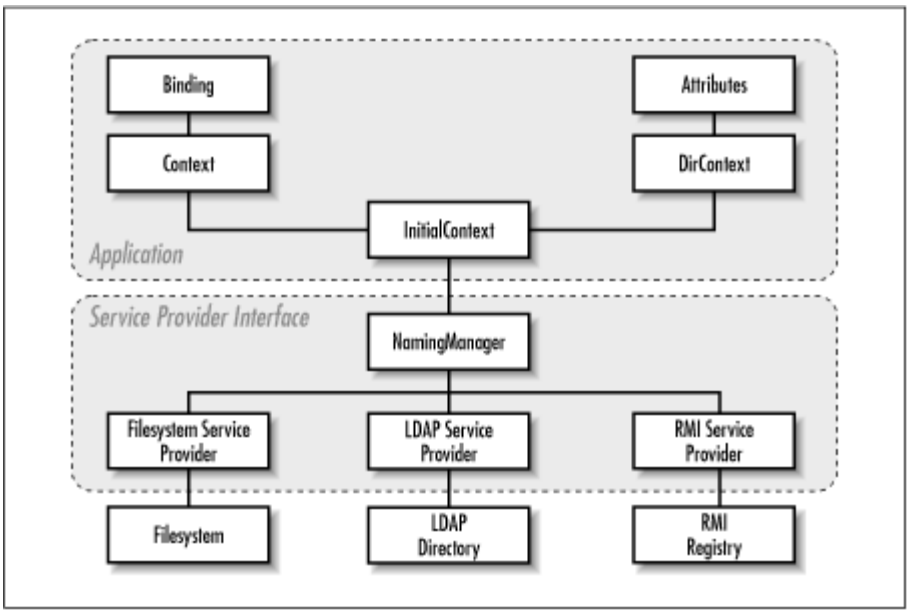
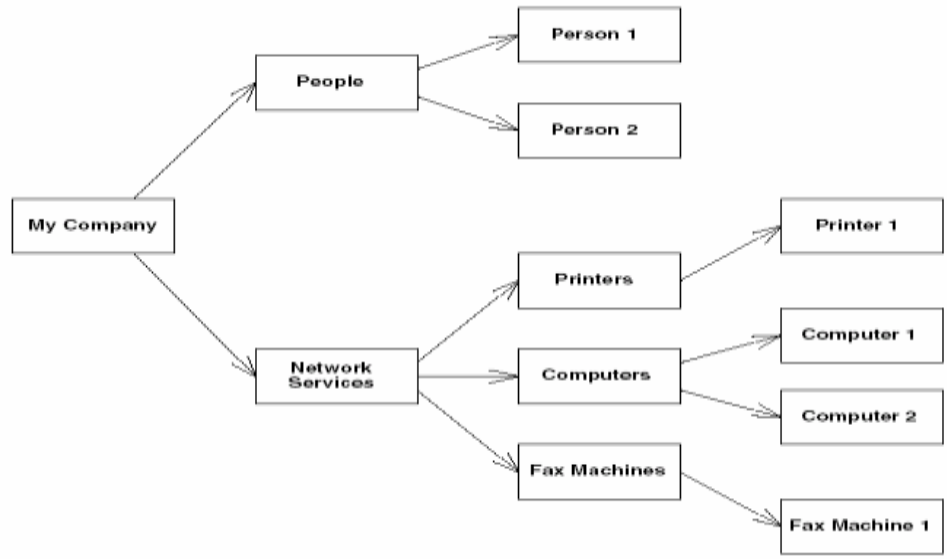


Figure 6-1. The JNDI architecture In order for an application to actually interact with a particular naming or directory service, there must be a JNDI service provider for that service. This is where the JNDI SPI comes in. A service provider is a set of classes that implements various JNDI interfaces for a specific naming or directory service, much like a JDBC driver implements various JDBC interfaces for a particular database system. The provider can also implement other interfaces that are not part of JNDI, such as Novell's NdsObject interface. The classes and interfaces in the javax.naming.spi package are only of interest to developers who are creating service providers. For instance, the NamingManager class defines methods for creating Context objects and otherwise controlling the operation of the underlying service provider. As an application programmer, you don't have to worry about the JNDI SPI. All you have to do is make sure that you have a service provider for each naming or directory service you want to use. Sun maintains a list of available service providers on the JNDI web page listed earlier.

Naming and Directory Services

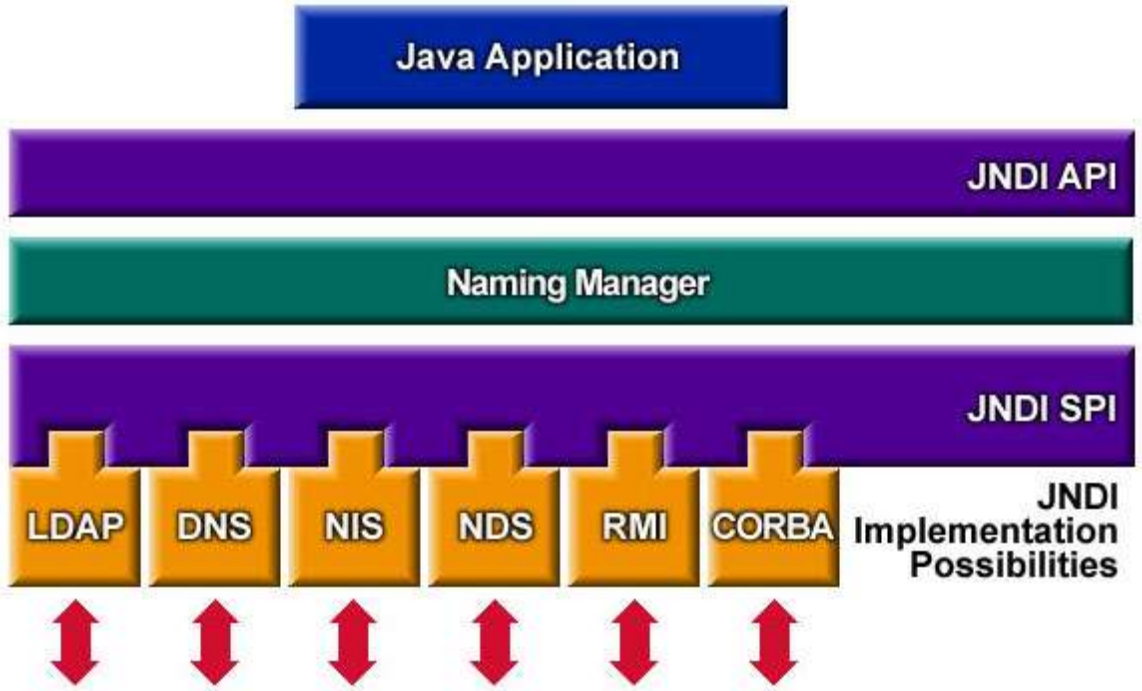
- A ***naming service*** associates names with objects. Thus, you can look up an object by its name.
- A ***directory service*** associates names and attributes with objects. Thus, you not only can look up an object by its name but also get the object's attributes or search for the object based on its attributes.

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



Common Naming/directory services:

- DNS
- RMI Registry
- CORBA naming service
- Lightweight Directory Access Protocol (LDAP)
- OS File system.
- Windows registry
- Windows Active Directory
- JNDI provides a unified approach (API) to access any naming/directory service.



Javax.naming package:

Context

 The javax.naming package defines a ***Context*** interface, which is the core interface for looking up, binding/unbinding, renaming objects and creating and destroying subcontexts.

 The most commonly used operation is ***lookup()***. You supply lookup() the name of the object you want to look up, and it returns the object bound to that name.

Printer printer = (Printer)ctx.lookup("treekiller");
printer.print(report);

Initial Context

 In the JNDI, all naming and directory operations are performed relative to a context.There are no absolute roots. Therefore the JNDI defines an ***initial context***, which provides a starting point for naming and directory operations. Once you have an initial context, you can use it to look up other contexts and objects.

Javax.naming.directory package:

Directory Context

 The ***DirContext*** interface represents a ***directory context***. It defines methods for examining and updating attributes associated with a directory object.

 You use ***getAttributes()*** to retrieve the attributes associated with a directory object (for which you supply the name). Attributes are modified using ***modifyAttributes()***.

You can add, replace, or remove attributes and/or attribute values using this operation.

 DirContext contains methods for performing content-based searching of the directory. In the simplest and most common form of usage, the application specifies a set of attributes--possibly with specific values-- to match and submits this attribute set to the ***search()*** method. Other overloaded forms of ***search()*** support more sophisticated functionality.

Example: Accessing DNS using JNDI

Step 1: Import packages

```
//import all of the classes and interfaces from the two packages
javax.naming and javax.naming.directory
import javax.naming.*;
import javax.naming.directory.*;
// import all of the classes and interfaces from java.util
import java.util.*;
```

Step 2: Prepare Properties

```
// Prepare properties to create initial context
Hashtable env = new Hashtable();
// This property is used to select the DNS service provider as the initial context
env.put(Context.INITIAL_CONTEXT_FACTORY,
"com.sun.jndi.dns.DnsContextFactory");
// This property is used to specify whether all responses must be authoritative
env.put(Context.AUTHORITATIVE, "false");
// This property is used to specify that recursion is disallowed on DNS queries
env.put("com.sun.jndi.dns.recursion", "true");
// This property specifies the number of milliseconds to use as the initial timeout
// period (i.e., before any doubling)
env.put("com.sun.jndi.dns.timeout.initial", "2000");
// This property specifies the number of times to retry each server
env.put("com.sun.jndi.dns.timeout.retries", "3");
// This property specifies the host name and port of the DNS server used by the initial
DNS context
env.put(Context.PROVIDER_URL, "dns://hal.cs.wmich.edu/
dns://dns.wmich.edu/");
```

Step 3: Create Initial Context and then Add/remove/modify/search!

```
try
{
// Create the initial directory context
```

```
DirContext ictx = new InitialDirContext(env);
// Ask for the value of a specifiy attribute
Attributes attrs1 = ictx.getAttributes("www.yahoo.com", new String[] {"A"});
// Ask for all attributes of the object
Attributes attrs2 = ictx.getAttributes("www.yahoo.com");
// Iterate through the attributes and print their values.
NamingEnumeration ne = attrs1.getAll();
while( ne.hasMore() )
{
Object attr = ne.next();
System.out.println("Attribute = " + attr);
}
}
catch
(NamingException e)
{
// Catch exception and print a message
System.err.println("Problem getting attribute:" + e);
}
```