Novell Developer Kit

www.novell.com

March 1, 2006

CLASSES FOR JAVA* AND JNDI PROVIDERS (NJCL)

Novell®

Legal Notices

Novell, Inc.
404 Wyman Street, Suite 500
Waltham, MA 02451
U.S.A.
www.novell.com

Online Documentation: To access the online documentation for this and other Novell developer products, and to get updates, see developer.novell.com/ndk. To access online documentation for Novell products, see www.novell.com/documentation.

Novell Trademarks

Third-Party Materials

Contents

About This Guide

Welcome to Classes for Java and JNDI Providers (NJCL), a component of the Novell® Developer Kit (NDK) software development kit. NJCL is a powerful and robust framework that serves as the foundation for several of Novell's Java applications, utilities, distributed services, and network management tools, and it provides access to a broad range of NetWare services.

Two versions of the NJCL are provided, NJCL, which requires Novell client software, and NJCL—Clientless. For more information about the clientless version, see Section 2.11, "Remote Session Manager," on page 96. This documentation covers both versions of the NJCL (with the exception of Javadoc reference pages, which are separately generated). Both versions require you to download NLM and NetWare Libraries for C (../../../../clib.htm).

This guide contains the following sections:

- "Fundamental NJCL and JNDI Concepts" on page 13
- "Novell Services (including JNDI Providers)" on page 39
- "NJCL and JNDI Solutions" on page 101
- "NJCL and JNDI Tasks" on page 127
- "API Reference" on page 133
- "Sample Source Code" on page 135
- "Revision History" on page 139

Audience

This guide is intended for Java developers interested in accessing NetWare® services by using Java.

Feedback

We want to hear your comments and suggestions about this manual and the other documentation included with this product. Please use the User Comments feature at the bottom of each page of the online documentation.

Documentation Updates

For the related developer support postings for Classes for Java and JNDI Providers, see the Developer Support Forums (http://developer.novell.com/ndk/devforums.htm).

Additional Information

For the most recent version of this guide, see the Classes for Java and JNDI Providers NDK page (http://developer.novell.com/ndk/njcl.htm).

Documentation Conventions

In Novell documentation, a greater-than symbol (>) is used to separate actions within a step and items in a cross-reference path.

Fundamental NJCL and JNDI Concepts

1

This chapter presents the fundamental concepts that should be understood before developing Java applications using NJCL and JNDI. The concepts topics are presented in an order that is logical rather than alphabetical. In some cases there are links within a concept topic to other related topics. The concepts contained within this chapter, together with the Novell® Services chapter, should provide you with a good understanding of the NJCL (including Novell Providers for JNDI) software development kit.

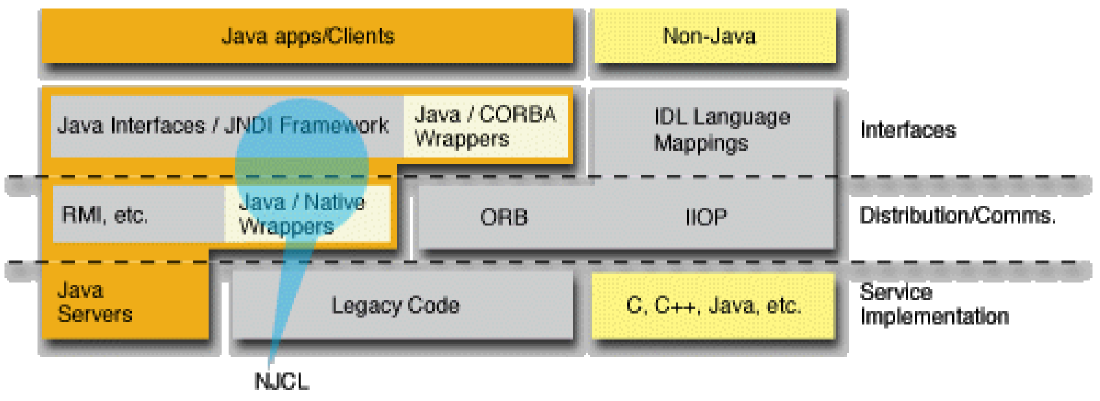
This section covers the following topics:

1.1 NJCL Basics and Architecture

NJCL plays a pivotal role in Novell's Java architecture by providing access to NetWare® services. These services are used by applications, components, and so forth, to build solutions. NJCL might be considered library-oriented middle-ware.

The following graphic represents the overall Novell architecture for Java and shows where NJCL fits into this architecture. As illustrated, NJCL implements the Java and/or native wrappers for legacy NetWare through the JNDI framework.

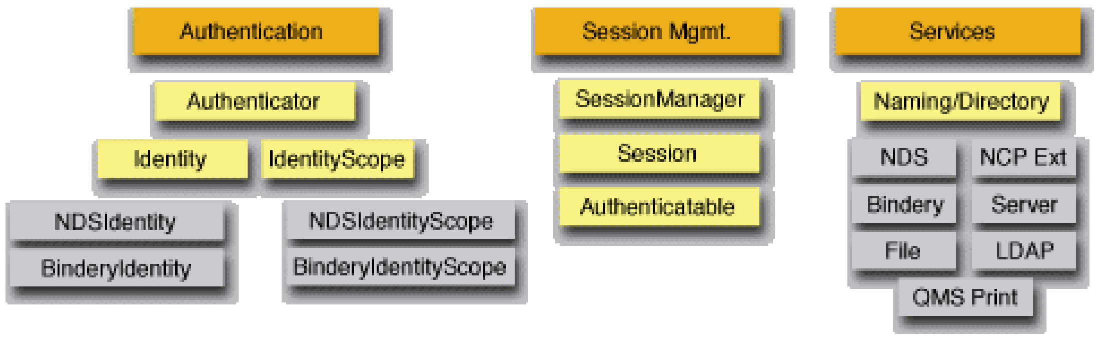
Figure 1-1 *Overall Novell architecture for Java*



In the past, NetWare has had several interfaces to access services, including the file system, bindery, and NDS. One of the first things to understand about NJCL is the JNDI interface, which provides a single interface that helps bring these diverse services together into a coordinated whole. The naming interface presented in JNDI has been used as a framework to coordinate the services provided by NJCL.

The architecture of NJCL consists basically of three areas of functionality, as shown in the following graphic and discussed below.

Figure 1-2 *NJCL architecture*



- Authentication

  Describes the security classes that provide login, logout, and token management functionality. The key authentication interfaces include Authenticator, Identity, and IdentityScope.

- Session Management

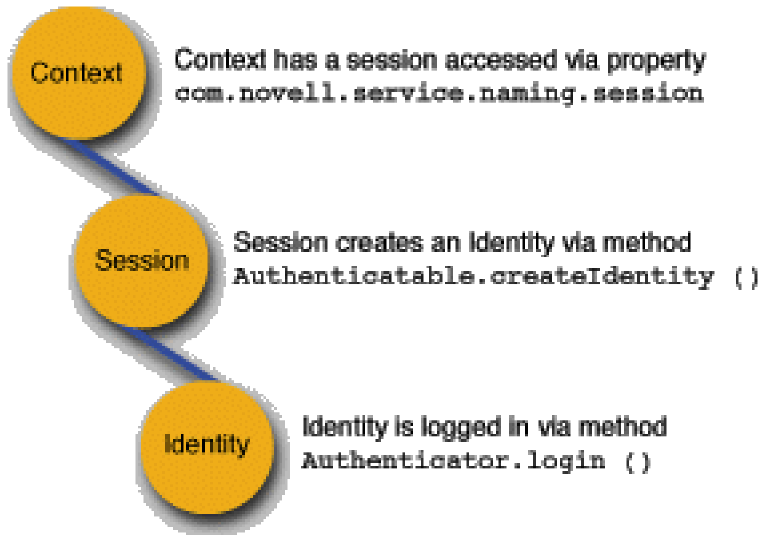
  Describes the client connection model and how connections are managed. The services provided by the Session Manager ties the naming and directory services to the authentication services. The key session management interfaces include SessionManager, Session, and Authenticable.

- Naming/Directory Services

  Includes descriptions of the majority of the functionality represented by NJCL. These services are included in the JNDI providers (NDS, Bindery, File System, LDAP, NCPExtensions, Queue Management System, and Server). Novell's NetWare services actually tie together several of the naming/directory services (NDS, bindery, and file).

The following graphic illustrates how the three functional components of NJCL are tied together. This example shows authentication using contexts.

Figure 1-3 *NJCL Authentication Using Contexts*


Context
Context has a session accessed via property
com.novell.service.naming.session
Session
Session creates an Identity via method
Authenticatable.createIdentity ()
Identity
Identity is logged in via method
Authenticator.login ()

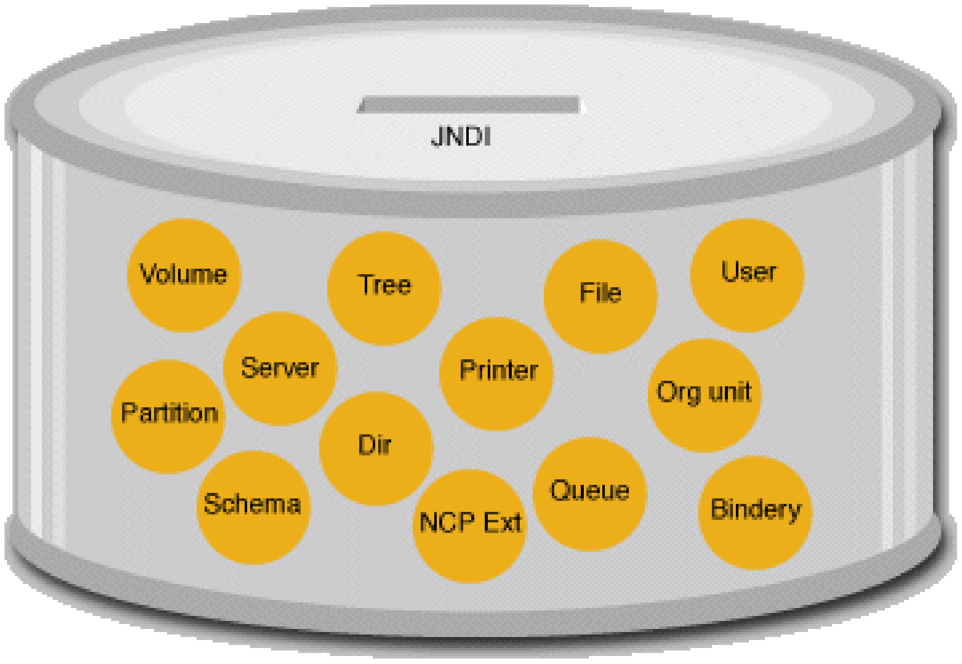
The NJCL implementation is built upon the existing Novell C libraries, NWCalls and NWNet for both Win32 and NLM platforms. The NJCL infrastructure deals with both fat clients (those clients that are installed) and thin clients (those clients that download any code needed at runtime). This strategy creates a back end implementation that can be accessed remotely for the thin client while also handling installed clients. Therefore, at this time all NJCL programs require a NetWare client.

1.2 How NJCL and JNDI Work Together

In any distributed computing environment, naming and directory services largely determine how other sevices and applications coordinate. This is true of NJCL and JNDI. JNDI is the API and framework through which an application can access one or more naming systems through a single interface.

One of the more difficult concepts to understand in NJCL is how NJCL and JNDI are related. The following graphic may help to describe how NJCL and JNDI are integrated.

Figure 1-4 *NJCL and JNDI Integration*

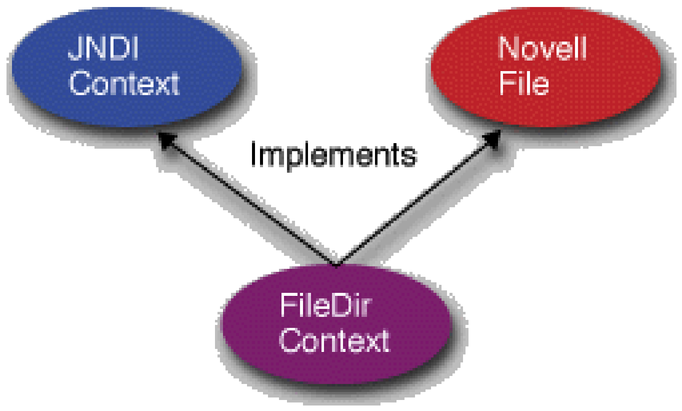


When you want your application to access the objects in the can, such objects Server, Printer, or File JNDI provides a single point of access. JNDI acts as the lid or opening to the can containing the objects. Thus, to find an NCP Extension, you access it through the same opening as if you were trying to find a User.

Some of the important services of JNDI are naming, attributes, search, schema capabilities, and so forth. In addition to the JNDI methods for naming and accessing objects, the objects themselves may include additional, non-naming methods. If you want to access methods for file read and write, or print a file to a printer, you use the interface for the real object, not the JNDI interface. Thus, JNDI provides the consistent access methods to get to objects, but does not define the actual objects.

The key to getting this all to work is the concept of duality in the objects. The ability of Java to mix two interfaces into a single object is used heavily in NJCL to provide objects that have a consistent naming interface but unique functionality. This makes it possible for an object to implement multiple interfaces. The example shown in the following graphic refers to a FileDirContext, which is a directory entry in the file system.

Figure 1-5 *JNDI Context Duality*



The FileDirContext entry may have additional subordinates (either files or directories) that will be accessible through the JNDI naming methods. At the same time, it provides access to file streams for its actual data. If an application is interested in one or the other interface for FileDirContext, it need only narrow the object to the desired type and use the appropriate methods. For further information on narrowing, see the "Narrowing" on page 25.

1.3 JNDI Basics and Architecture

The Java Naming and Directory Interface (JNDI) is an industry-wide, open, consistent interface that gives developers a common interface for navigating the many naming systems that exist in the computing world today. JNDI is a pure Java API, specified in the Java programming language, that provides access to any naming or directory service implementation, but yet it is defined to be independent of any specific directory service implementation. As such it greatly simplifies the code needed to browse and utilize the functionality of directory services such as NDS, X.500, and LDAP, as well as file systems, DNS, and so forth. Thus, a variety of directories, new and existing, can be accessed in a common way using JNDI.

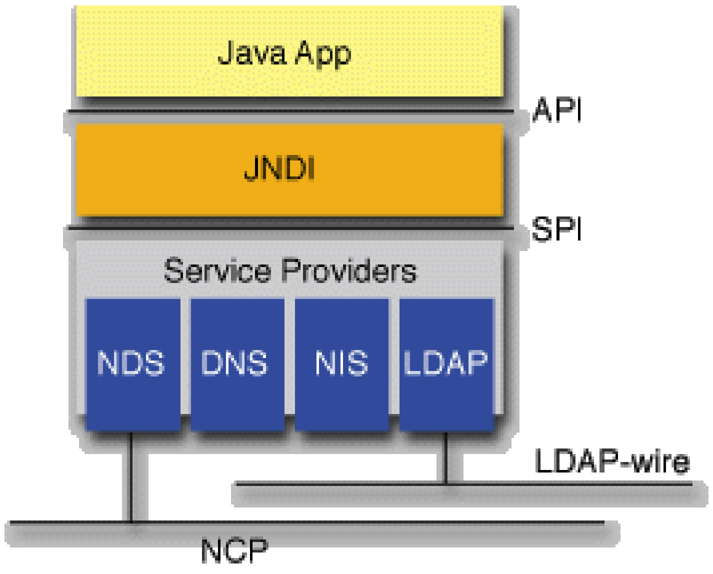
JavaSoft owns, maintains, and documents the core specification and functionality defined in JNDI, which is one of JavaSoft's extended API offerings and is part of the JavaSoft Enterprise API. You can access the official JNDI API specification at http://java.sun.com/products/jndi/1.2/javadoc/overview-summary.html (http://java.sun.com/products/jndi/1.2/javadoc/overview-summary.html)

In the design of JNDI, JavaSoft worked closely with Novell drawing upon Novell's expertise in directory services and networked file systems to help with the effort. JavaSoft allows JNDI to be distributed by Java licensees, thus allowing it to be distributed with this NDK.

As a Java interface, JNDI provides a common layer of abstraction that insulates the application programmer from multiple interfaces for dealing with the naming aspects of an object system. The naming implementation is totally invisible. Directory service developers can benefit from a service-provider capability that enables them to incorporate their respective implementations without requiring changes to the client.

JNDI is the API and framework through which an application can access one or more naming systems through a single interface, the Service Provider Interface (SPI). This concept is illustrated in the following graphic.

Figure 1-6 *JNDI Architecture*



JNDI has the following key benefits:

- It integrates multiple naming and directory interfaces into a single interface.
- It takes advantage of the Java environment by using native Java objects for all processing. As a result, messy object type conversions are eliminated.

If you've written a program that works with NetWare and NDS, chances are you had to hard-code operations on names, such as creation, deletion, and renaming, among others, for each naming system with which you had to work. JNDI levels the playing field, making all naming systems look alike.

In JNDI, you access files, directories, NCP extensions, servers, and NDS objects through the Context interfaces. You can also access the attributes and schema of directory contexts through the DirContext interfaces.

Although JNDI provides an interface to access named objects in naming systems, providers are required to implement certain JNDI methods for accessing a particular naming system. JNDI acts as a mediator between the application and one or more naming system providers.

Because JNDI supports more than one naming system, the functionality for a given context is dependent on the naming system from which it comes. For example, contexts in the server naming system have different functionality from contexts in the file naming system. So while you would use the exact same code to access the attributes of the context from the server naming system as the context from the file naming system, the attributes returned by the two contexts may be completely different.

Because JNDI abstracts the differences behind contexts, you can access and work with all contexts and their functionality using only the core interfaces and their methods.

1.4 The Components of JNDI

JNDI provides both an API for almost all software developers and a Service Provider Interface (SPI) for the underlying service providers. Java applications sit on top of the JNDI API framework, providing access to the Service Provider Interface (SPI) and JNDI classes. The Service Providers then provide access to the naming systems in a 1-to-1 relationship.

The JNDI framework API is contained in three interface packages:

- "Naming Interface" on page 19 in the javax.naming package, which provides the naming operations.
- "Directory Interface" on page 20 in the javax.naming.directory package, which provides the directory operations.
- "Service Provider Interface" on page 21 in the javax.naming.spi package, which contains the SPI.

1.4.1 Naming Interface

The naming interface organizes information hierarchically and maps human-friendly names to addresses or objects that are machine-friendly. It allows access to named objects through multiple namespaces. There are many interfaces and classes defined in JNDI; however, the core interface in the javax.naming package is Context, a context without attributes. It contains the heart of the JNDI features, and it is extended or implemented by several other classes and interfaces in the system.

The defined methods in the Context interface allow you to perform basic operations on named objects, such as adding a name-to-object binding, looking up the object bound to a specified name, listing the bindings, removing a name-to-object binding, creating and destroying subcontexts, and so forth. Each method is a one-to-one mapping to behaviors of the context.

Because all contexts are abstracted behind the Context interface, you can access all contexts through the Context interface methods. This makes all contexts have a uniform interface.

Methods in the Context interface that will be commonly used include the following:

- lookup( ) returns the name of an object of whatever class is required by the Java application.
- list( ) returns an enumeration of the names and the class names of their bound objects in the named context.
- listBindings( ) returns an enumeration of the names and their bound objects in the named context. Calling listBindings( ) on a directory returns the name of objects bound to that specified directory.

Other key classes and interfaces in the javax.naming package include:

- Binding, which represents a name-to-object binding found in a context.
- CompositeName, which represents a composite name that is a sequence of component names spanning multiple namespaces.
- CompoundName, which represents a name from a hierarchical namespace.
- InitialContext, which is the starting context for performing naming operations. A client obtains an initial context to provide a starting point for name resolution.
- Name, which represents a generic human-friendly name. Every name is relative to a context and every naming operation is performed on a context.

- NameParser, which is used to manipulate a name in a namespace.
- Reference, which represents a reference to an object found outside the naming/directory system.

1.4.2 Directory Interface

JNDI includes a directory service interface that provides access to directory objects, which can contain attributes, thereby providing attribute-based searching and schema support. The major interfaces in this package are DirContext, Attribute, and Attributes.

The DirContext interface defines methods for examining and updating attributes associated with directory objects. The DirContext interface extends the javax.naming.Context interface, and is a context containing zero or more attributes. Usually a DirContext represents a named object such as a file, a directory, or an NDS object, each of which has attributes that you can access and modify. The main difference between a DirContext and a Context is that a DirContext allows you to retrieve and modify attributes and schema data, whereas a Context does not.

DirContext methods allow you to perform operations on named objects with attributes. Each directory object contains a set of zero or more objects of the class Attribute. An Attribute represents an attribute associated with a named object and is denoted by a string identifier; and it can have zero or more values of any type.

Further, since a DirContext is also an instance of Context, it inherits all the methods in the Context interface, and thus you can perform all Context methods on a DirContext. Additional DirContext methods allow you to get, set, and modify attributes, as well as access the schema and the schema class definition of an object. Each method is a one-to-one mapping to operations you can perform on attributes of the context.

Most of the contexts in the Novell services implement the DirContext interface. However, some contexts, which represent logical objects that have no attributes, implement the Context interface (such as Trees, Servers, File System, NCPExtensions, and Bindery). All NDS objects and the files and directories of the NetWare file system are implemented as DirContexts.

Methods in the DirContext interface that will be commonly used include the following:

- search( ) supports context-based searching, and returns the matching directory objects along with the requested attributes.
- createSubcontext( ) creates a new subcontext with a name that you pass into the method as a parameter, binds the subcontext in the target context you specify, and associates attributes you pass in as a parameter.
- getAttribute( ) returns a NamingEnumeration of the Attributes associated with the named object that you pass into the method as a parameter.
- getSchema( ) retrieves the schema root of the DirContext you specify.

Other key classes and interfaces in the javax.naming.directory package include the following:

- Attribute, which is an actual attribute associated with a named object of a context. An attribute consists of an identifier (attrID) and a set of zero or more values. Each value in the set is typed as a java.lang.Object that you narrow to the appropriate class or interface.The definition of the attribute and the syntax of its value(s) are contained in the schema if the underlying directory service supports schemas.

- Attributes, which represent a collection or set of individual instances of attributes that are associated with a DirContext object, and attributes that use java.util.Enumeration methods to access attribute objects in the attributes set. The Attributes of a DirContext represent attributes of a context in the service provider.
- InitialDirContext, which is the starting context for performing directory operations. It represents the programmatic entry point to a service provider.
- SearchControls, which encapsulates factors that determine the scope of search and what gets returned as a result of the search.
- SearchResult, which represents an item returned as a result of the DirContext.search( ) method.

1.4.3 Service Provider Interface

JNDI does not include service providers; however, JavaSoft does provide a Service Provider Interface (SPI), which third parties such as Novell use to implement service providers. The SPI provides a way for service providers to develop and hook up their naming and directory implementations to the JNDI. The JNDI allows specification of names that span multiple namespaces. Thus, the SPI provider methods allow different provider implementations to cooperate so as to complete client JNDI operations.

The JNDI SPI is not for application developers, and you do not need to use the SPI unless you are writing a provider for JNDI. The SPI is for developers who write a service provider implementation for a naming system, such as Novell Directory Services (NDS). Novell ships service providers for JNDI that enable you to access NetWare and a host of NetWare services, including NDS. This means that programmers can have access to NetWare services without worrying about the SPI.

A JNDI service provider is actually a context implementation, which provides a class that implements the Context or DirContext interface in the JNDI API. By implementing the interface and providing the actual methods of the interface, the vendor of different naming and directory service providers can expose their services to JNDI-enabled applications.

JNDI has a concept of federation, which means a context in one naming system can be bound to a context in another naming system. It is the decision of the service provider vendor to determine if federation is supported. If it is supported, name resolution across multiple namespaces is handled in the service provider and is completely transparent to the application programmer. The implementation of federation enables a single application to access data across multiple namespaces seamlessly because different service providers cooperate with each other to complete JNDI operations. The NJCL fully supports context federation and provides multiple service providers for different NetWare namespaces. This enables NetWare application programmers to navigate through the global Novell namespace to access different NetWare services. For a more detailed explanation of federation, see “Context Federation” on page 24.

Context interfaces from different naming systems implement the JNDI Service Provider Interfaces differently depending on the semantics of the naming system. “Novell Services (including JNDI Providers)” on page 39 describes how each context implements these interfaces.

1.5 Clarifying Contexts

The following distinguish among the several different uses of the term "context":

- “JNDI Context” on page 22
- “Initial Context” on page 22

1.5.1 JNDI Context

As a central concept in JNDI, context is used in a variety of ways to refer to different aspects of JNDI programming. First, it is an object whose state is a set of bindings with distinct atomic names. A context has an associated naming convention and it provides naming operations. Examples of contexts are files, directories, servers, and anything else that can have bindings.

An object is a context if something can be bound to the object, or if the object has subordinates that could be described as bindings. A context that is bound to another as a subordinate is known as a subcontext. A directory is a context, since files and other directories can be bound to it. There is no hard and fast rule that excludes an object from being a context; however, items that have no container abilities cannot be contexts. The naming system decides which nodes will be terminal, or leaf nodes.

Sometimes we refer to context as a named object; however, this is not exactly accurate, since a printer can have a name and not be a context. Since nearly all of the named objects you will work with are contexts, we refer to named objects and contexts interchangeably.

1.5.2 Initial Context

An initial context is a programmatic entry or starting point to a service provider for resolution of names within a namespace. The only time you must work directly with an initial context is when first accessing a service provider. You must set the system properties to reflect the service provider with which you want to work. Then once the properties are set, you create an initial context for that service provider.

Creating an initial context requires that you specify the appropriate initial context factory. You can learn which are the appropriate initial context factories for the naming systems by viewing the documentation for an individual service provider in "Novell Services (including JNDI Providers)" on page 39. The following initial contexts are defined:

Initial context	Initial context factory
File system	FS_INITIAL_CONTEXT_FACTORY
NDS	NDS_INITIAL_CONTEXT_FACTORY
NetWare	NW_INITIAL_CONTEXT_FACTORY
Queue management	QMS_INITIAL_CONTEXT_FACTORY
Server	SERVER_INITIAL_CONTEXT_FACTORY

Usually the appropriate provider to work with is the Section 2.7, “NetWare Provider,” on page 77, which was designed to federate to all service providers, and acts as an effective root to all other providers.

1.5.3 Context Binding

The association between an object and its atomic name or some other property is known as context binding. A name-to-object binding contains the name of the bound object, the name of the object's class, and the object itself. A binding can be associated with a NULL object, and a context can contain zero bindings.

When you create a subordinate context, or bind in an object reference to an existing context, you are creating an association between two objects-the existing context and the new subordinate. This association is a binding, which must be given a name. When you list the subordinates of a context, and then look up one of them, you are performing the opposite of binding. You are resolving a context by name, thus you are looking at the object that was bound using the given name.

1.5.4 Context Interface

The most concrete use of the term is in the javax.naming.Context interface. This is the primary JNDI interface, which defines the naming methods: list, lookup, search, bind, unbind, and so forth.

Subclasses of the javax.naming.Context interfaces include the javax.naming.InitialContext class, the javax.naming.directory.DirContext interface, and the javax.naming.directory.InitialDirContext class. These subclasses are also often referred to generically as contexts to emphasize their commonality.

Classes that implement the javax.naming.Context interface (directly or using a derivative subclass) are also referred to as contexts. Thus, a file context actually refers to the file class or object that represents a file while also implementing the Context interface as its naming method. Again, it may also be referred to generically as a context to emphasize its commonality with the rest of the naming systems.

1.5.5 Implementor of Context

A component of a service provider that implements part of the behavior of a naming system is also referred to as a context. In this sense, a context is the implementation of a volume, a directory, a file, and so forth, and is closely associated with the service provider interface.

A context’s implementation resides in the bytecode for the service provider to which it belongs. This implementation is often private so that you are insulated from the details of how a particular context works. Because JNDI is the public interface for the underlying NetWare naming systems, you need learn only how to use JNDI, not the many different interfaces.

Sometimes you need to know the class of a particular context, usually when making a decision about which non-JNDI interfaces or attribute IDs you need to work with. You can find out what kind of class the context is through the instanceof operator in Java. Even though the implementation details may be private, you can find out what implementation you are working with, then look up the attribute IDs in the naming system.

1.5.6 DirContext Interface

An extension of the JNDI Context interface that includes attributes is the DirContext interface. Any DirContext can also act as a naming context, allowing you to perform context operations on a DirContext.

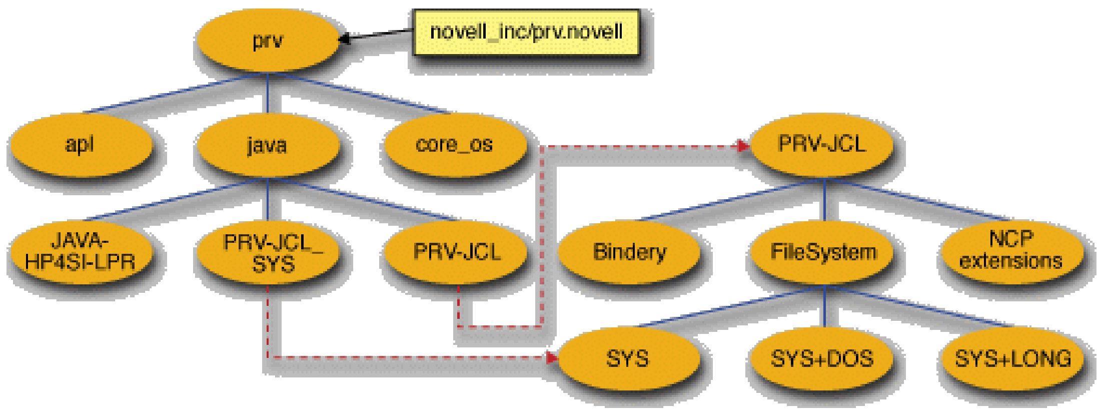
A DirContext has overloaded methods that allow you to perform certain naming and directory operations automatically. Usually, these methods have the same name as the equivalent Context methods, only with an overloaded signature that requires you to pass in an AttributeSet.

A DirContext can also return schema information. For example, the method getSchema( ) returns the root of the schema of the DirContext, whereas the method getSchemaClassDefinition( ) returns the definition of the DirContext. These schema objects are each in turn a DirContext, so you can call standard DirContext methods to work with a DirContext schema in addition to its own attributes. The schema objects provide descriptive information about the DirContext that returned them.

1.5.7 Context Federation

Federation is a context in one naming system referring to a context, or object, in another naming system. It is actually a binding between a context in one name system and an object in another name system. The following graphic shows some examples of federation in the NDS naming system.

Figure 1-7 *NJCL Federation in NDS Naming System*



Another example of federation occurs when you are in the NDS naming system, and you encounter an NDS object that is a container for a NetWare volume, and you want to list the subordinates of the volume container. When you perform the list operation, you have federated into the file system naming system and are no longer in the NDS naming system.

Novell’s JNDI service providers use two types of federation: explicit and implicit.

Explicit federation occurs when you explicitly bind a referencable object from one naming system into another. The application will see explicit bindings/federations when list( ) is called. Novell’s service providers use the following names for explicit federation:

- Trees
- Servers
- FileSystem

- NCPExtensions
- Bindery

As an example, the following name uses explicit federation:

```
Servers/MY_SERVER/FileSystem/SYS/MyFile
```

If the current service provider for the application was the NetWare service provider, federation would occur at the following points in the name:

- servers/
- FileSystem/

Implicit federation is implemented by the service provider, which decides how to federate. This happens when you have a context at the terminal of a naming system, and you call the list( ) method with the context's name followed by a trailing slash (/). The trailing slash (/) is the key to the implicit federation.

For example:

```
Trees/MY_TREE/MY_SERVER.MY_OU.MY_ORG
```

would take you to the MY_SERVER object in the NDS directory.

```
Trees/MY_TREE/MY_SERVER.MY_OU.MY_ORG/
```

would take you to the actual server.

In this case, the NDS service provider federates into the Server service provider because MY_SERVER is a server object. Notice that the Server key word was not used.

1.5.8 NDS Context

This term denotes a logical position in a hierarchical NDS tree or an identifier for the current position in a tree. This concept should not be confused with a JNDI context.

The Full Distinguished Name (FDN) of an NDS object shows the context of the object. For example, the name joe.engineering.novell refers to a user object in the context of engineering.novell.

1.6 Narrowing

Narrowing is the act of casting an object to a particular class. In JNDI you will typically narrow objects from a JNDI interface, such as Context or DirContext, to its concrete class, typically part of a Novell provider package.

Narrowing involves using the dynamic features of Java, specifically Java's syntactical ability to return a string value for the name of a class. This is useful because you can learn the object's class name, then perform methods supported by the class.

For example, if you were working with an NCPExtensionDirContext, you would narrow the java.lang.Object returned from getValues( ) to the appropriate NJCL class for an NCP extension.

1.7 Object Names

In JNDI, there are several types of object names. All object names are interpreted relative to the context that uses them, hence they are often referred to as relative object names.

- An Atomic name is the indivisible component of an object name.
- A Compound name is a sequence of one or more atomic names that identify an object.
- A Composite name is an object name that spans multiple naming systems.

For example, if the current context was at the root of the NetWare namespace, you would specify the name of an object in the following manner:

```
Servers/My_Server/SYS/My_Dir/myfile.txt
```

The relative object name in the context Servers/My_Server/SYS would be

```
My_Dir/myfile.txt
```

1.8 JNDI Object Naming

This section covers the following topics:

- Section 1.8.1, "NetWare Namespace," on page 27
- Section 1.8.2, "NDS Namespace," on page 28
- Section 1.8.3, "Server Namespace," on page 29
- Section 1.8.4, "File System Namespace," on page 29

The naming convention for object names followed by Novell's JNDI service providers is not the same as that of traditional NetWare names. For example, to specify the complete name and path of a file on a NetWare server you would use the following form:

```
MY_SERVER/SYS:PUBLIC\MYFILE
```

With Novell's JNDI service providers, assuming the initial context is the NetWareInitialContext, the same file's path would be

```
Servers/MY_SERVER/FileSystem/SYS/PUBLIC/MYFILE
```

Names used in JNDI naming conventions include Trees, Servers, Bindery, FileSystem, and NCPExtensions. These names are keywords that must be used in name strings at their appropriate location. Novell's JNDI service providers use these keywords to resolve names to the proper objects.

The naming convention that you use depends on your initial context factory (see "Initial Context" on page 22), which determines what namespace you are using. See the following table to find a list of object names for a given namespace.

Namespace	Initial context factory	Supported naming conventions	See section
File system	FS_INITIAL_CONTEXT_FACTORY	My_Server/...	"File System Namespace" on page 29

Namespace	Initial context factory	Supported naming conventions	See section
NDS	NDS_INITIAL_CONTEXT_FACTORY	My_Tree/...	"NDS Namespace" on page 28
NetWare	NW_INITIAL_CONTEXT_FACTORY	Servers/..., Trees/...	"NetWare Namespace" on page 27
Server	SERVER_INITIAL_CONTEXT_FACTORY	My_Server/...	"Server Namespace" on page 29

1.8.1 NetWare Namespace

Use NetWare object names relative to the NetWare initial context (NW_INITIAL_CONTEXT_FACTORY), which could be thought of as the root of Novell's NetWare naming systems.

- Server Objects

  Servers/My_Server - A server
- Bindery Objects

  Servers/My_Server/Bindery - The bindery root

  Servers/My_Server/Bindery/A-SERVER+4 - A bindery object (server)
- File System Objects

  Servers/My_Server/FileSystem - The file system root

  Servers/My_Server/FileSystem/My_Volume - A file system volume

  Servers/My_Server/FileSystem/My_Volume/My_Dir/my_file.txt - A file
- NCPExtensions Objects

  Servers/My_Server/NCPExtensions - The NCPExtensions root

  Servers/My_Server/NCPExtensions/SNMP - An NCPExtensions object
- NDS Objects

  Trees/My_Tree - The root of the tree

  Trees/My_Tree/My_Org - An organization in the tree

  Trees/My_Tree/O=My_Org - An organization in the tree

  Trees/My_Tree/My_Org_Unit.My_Org - An organizational unit within the organization

  Trees/My_Tree/OU=My_Org_Unit.O=My_Org - An organizational unit within the organization

  Trees/My_Tree/My_User.My_Org_Unit.My_Org - A user in the organizational unit

  Trees/My_Tree/CN=My_User.OU=My_Org_Unit.O=My_Org - A user in the organizational unit

  NDS names have the following properties:

  - The names are in partial dot form (separated by dots instead of slashes).
  - The names can be typed or untyped.

- The specification of the object name after the tree name is with the leaf object on the left.

  For example:

  `My_User.My_Org_Unit.My_Org`

  not

  `My_Org.My_Org_Unit.My_User`

1.8.2 NDS Namespace

Use NDS object names relative to the NDS initial context (NDS_INITIAL_CONTEXT_FACTORY), which could be thought of as the root of Novell's NDS naming systems.

- Server Objects

  My_Tree/CN=My_Server.O=My_Org - A server
- Bindery Objects

  My_Tree/CN=My_Server.O=My_Org/Bindery - The bindery root

  My_Tree/CN=My_Server.O=My_Org/Bindery/A-SERVER+4 - A bindery object
- File System Objects

  My_Tree/CN=My_Server.O=My_Org/FileSystem - A file sytem root

  My_Tree/CN=My_Server.O=My_Org/FileSystem/My_Volume - A file system volume

  My_Tree/CN=My_Server.O=My_Org/FileSystem/My_Volume/My_Dir/my_file.txt - A file system file
- NCPExtensions Objects

  My_Tree/CN=My_Server.O=My_Org/NCPExtensions - The NCPExtensions root

  My_Tree/CN=My_Server.O=My_Org/NCPExtensions/SNMP - An NCPExtensions object
- NDS Objects

  My_Tree - The root of the tree

  My_Tree/My_Org - An organization in the tree

  My_Tree/O=My_Org - An organization in the tree

  My_Tree/My_Org_Unit.My_Org - An organizational unit within the organization

  My_Tree/OU=My_Org_Unit.O=My_Org - An organizational unit within the organization

  My_Tree/My_User.My_Org_Unit.My_Org - A user in the organizational unit

  My_Tree/CN=My_User.OU=My_Org_Unit.O=My_Org- A user in the organizational unit

  NDS object names have the following properties:

  - The names are in partial dot form (separated by dots instead of slashes).
  - The names can be typed or untyped.
  - The specification of the object name after the tree name is from left to right with the leaf object on the left.

    For example:

    `My_User.My_Org_Unit.My_Org`

    not

```
My_Org.My_Org_Unit.My_User
```

1.8.3 Server Namespace

Use Server object names relative to the ServerDirContext (SERVER_INITIAL_CONTEXT_FACTORY), which could be thought of as the root of Novell's Server naming system.

- Server Objects

  My_Server - A server object.
- Bindery Objects

  My_Server/Bindery - The bindery root

  My_Server/Bindery/A-SERVER+4 - A bindery object (server)
- File System Objects

  My_Server/FileSystem - The file system root

  My_Server/FileSystem/My_Volume - A file system volume

  My_Server/FileSystem/My_Volume/My_Dir/my_file.txt - A file
- NCPExtensions Objects

  My_Server/NCPExtensions - The NCPExtensions root

  My_Server/NCPExtensions/SNMP - An NCPExtensions object
- NDS Objects

  You can get NDS objects directly from NDSInitialContext (See "NDS Namespace" on page 28 for how this is done).

  To get NDS objects in indirectly you must first federate into the bindery, then use the extended tree name obtained by calling list() to federate into the NDS namespace. An example of this is as follows:

  ```
  My_Server/Bindery/<tree name>+<extension>+278>/
  ```

1.8.4 File System Namespace

Use File System object names relative to the FileSystem initial context (FS_INITIAL_CONTEXT_FACTORY), which could be thought of as the root of Novell's File System naming system.

- File System Objects

  My_Server - The file system root

  My_Server/My_Volume - A file system volume

  My_Server/My_Volume/My_Dir/my_file.txt - A file

1.9 Naming System Concepts

The native naming system is the mechanism that enables JNDI to perform complex naming operations. A naming system is a connected set of contexts of the same type that have the same naming convention, the same set of operations, and identical semantics. A namespace is an instance of a naming system, and as such contains the set of all names in the naming system. A file system, an NDS directory, and even the components of a URL are all naming systems, because they have named objects that can be organized in particular naming conventions.

The only way to determine if two contexts are in the same naming system is to use the equals( ) method on the NameParser instance returned by a given context.

A native naming system can be written in Java for full functionality, or it can be written in C, which requires the use of JNDI. An example of a native naming system is the NetWare file system. It has no Java representation but is implemented in the C language. Without JNDI, you would have to access it through java.io.File, which would prevent you from using advanced file system functionality not present in File. Or you would have to access it through the Java Native Interface (JNI) or from outside of Java.

A naming system can have a Schema (see Section 1.12, “Schema Concepts,” on page 32), which is a certain structure within which contexts in a naming system can interact. For example, in the file system, a user can put a file in a directory, a directory in a directory, or a directory or a file on a volume. But the user can’t put a directory into a file or a volume into a directory.

The file system is a well-known naming system that is comprised of particular contexts, so it is a good example of how a naming system works. The file system naming system contains the following contexts:

- Volume

  A volume mounted on a NetWare server containing directories and files directly subordinate to the volume in the file system.
- Directory

  A directory is subordinate to a volume on a file system and contains directories and files.
- File

  A file is subordinate to a volume or directory on a file system.

1.10 Service Provider Concepts

A service provider implements contexts in a naming system. It also implements an interface, such as JNDI, which is the programmatic abstraction of contexts and their methods, whereas service providers contain the implementation of JNDI contexts.

This section covers the following topics:

- Section 1.10.1, “Service Provider Initial Context,” on page 30
- Section 1.10.2, “Novell Service Providers,” on page 31

1.10.1 Service Provider Initial Context

Just as a naming system is comprised of contexts organized by function, a service provider is comprised of the implementations of those contexts. A service provider also has an initial context factory, which can produce entry points into the provider. Contexts in naming systems map JNDI operations to the semantics of the underlying naming system. This means that calling a Context or DirContext method on a DirContext from a different service provider can produce different results.

The file system is a well-known naming system, so it is a good example of the composition of a service provider. The File System service provider contains the following contexts:

- FileSystemInitialDirContext

  The programmatic entry point to the File System service provider. When setting the system properties, use the FileSystemInitialDirContextFactory.
- VolDirContext

  The implementation of a volume mounted on a NetWare server.
- DirectoryContext

  The implementation of a directory.
- FileContext

  The implementation of a file.

1.10.2 Novell Service Providers

Novell has implemented JNDI methods for named objects in Novell naming systems. Although these providers are public implementations of Context and DirContext, you cannot access provider classes except through JNDI. Following is a list of the Novell services (including JNDI Providers) that Novell has currently implemented.

- "Bindery Provider" on page 47

  Describes the Bindery provider for JNDI, which manages objects stored in a NetWare server's bindery.
- "NetWare File System Provider" on page 68

  Describes the NetWare file system provider for JNDI, how to access file streams, create and delete files, and access the attributes of the NetWare file system.
- "NetWare Provider" on page 77

  Describes the NetWare provider for JNDI, which acts as a logical root namespace for the Novell Providers for JNDI.
- "NCP Extensions Provider" on page 52

  Describes the NetWare Core Protocol (NCP) Extensions provider for JNDI, and how to access these NetWare operating system extensions.
- "NDAP Provider for Novell Directory Services (NDS)" on page 55

  Describes the NDS provider for JNDI and the NDS service, including NDS objects, schema, and partitions.
- "Queue Management System (QMS) Provider" on page 81

  Describes the Queue Management System provider for JNDI, including the creation, submission, and servicing of queue jobs.
- "Server Provider" on page 85

  Describes the Server provider for JNDI, and the server management and control functionality.

1.11 Attribute Concepts

An attribute is comprised of an attribute ID (the name of the attribute) and one or more values. Each value is an object of class java.lang.Object, meaning that a value can be a String, Integer, and so forth. In JNDI, an attribute is an atomic property of a DirContext represented in the interface javax.naming.directory.Attribute and the class javax.naming.directory.BasicAttribute.

An attribute set is a collection of attributes of a DirContext, and in JNDI is represented in the javax.naming.directory.Attributes interface. Individual attributes are accessed by using the javax.naming.NamingEnumeration interface, which allows you to traverse the attribute set using the next( ) method that returns individual Attribute objects.

Dynamic attributes are attributes that can optionally be associated with an object. For example, if you call getAttributes( ) at a context for an NDS object, you will get an AttributeSet that contains only the attributes that have been defined (assigned a value) for the object. If a new attribute is added to the object and you again retrieve the object's attributes, you will get a set of attributes containing the newly added attribute.

The NDS schema specifies attributes as mandatory or optional. Mandatory attributes must be defined and given a value before the object can be created. For example, the NDS user class has an attribute called Surname. Whenever you create an NDS user object, the Surname attribute must be given a value.

Optional attributes may be added to the object, and may or may not be assigned a value, after its creation. For example, the NDS user class has an attribute called FAX. It is not a requirement that the FAX attribute be defined when the object is created.

1.12 Schema Concepts

Each Novell naming system has a schema, which contains the syntactic information about the naming system. It provides information about the relationships between contexts, information about the naming system's attributes, the syntaxes of attributes, the composition of attribute values, and the relationships between classes in the naming system.

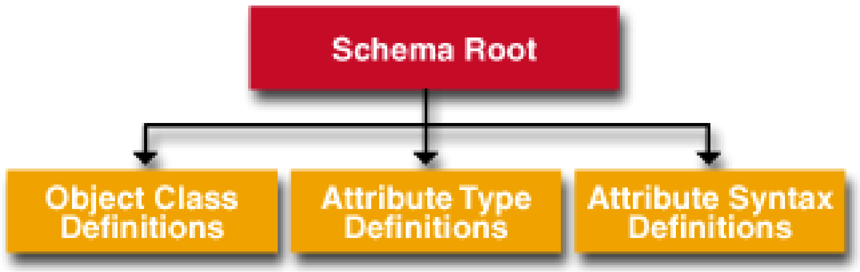
The detail specifications for the NDS schema are in the *NDK: Novell eDirectory Schema Reference*, which is part of the NDS Libraries for C Documentation. The following schema topics are briefly presented here:

- "Schema Structure" on page 33
- "Schema Components" on page 33

1.12.1 Schema Structure

The schema is structured as shown in the following graphic:

Figure 1-8 *Schema Architecture*



Although the schema is not a required part of a naming system, it makes the programmer's job of figuring out how to use a naming system easier. Novell has implemented a schema for each of its naming system providers because it enhances the integration between naming systems. Each schema is implemented by the respective service provider to simulate a schema.

The schema objects are important to NetWare programmers because NDS has a rigorously defined schema, and in order to accomplish certain tasks in NDS, you must manipulate the schema. For example, the containment rules embedded within the schema root of the NDS provider tell you that it is meaningful to put an organizational unit in an organization

```
(OU=DOCUMENTATION.O=NOVELL)
```

and that you can put a user object in an organizational unit.

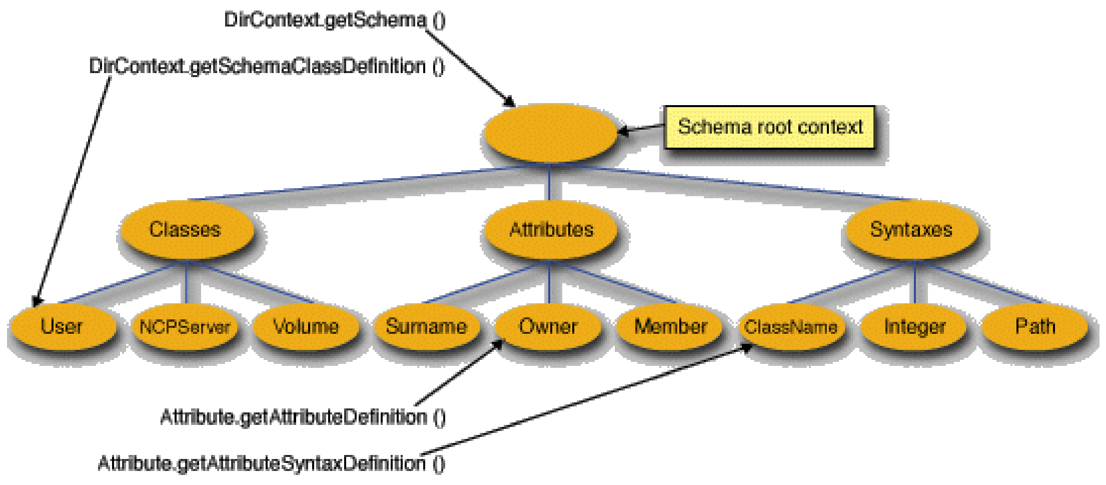
```
(CN=BILLY.OU=DOCUMENTATION)
```

But the rules also tell you that it is not meaningful to put an organization in an organizational unit, nor an organizational unit in a user.

1.12.2 Schema Components

The components stored in the schema consist of the schema root, which is the DirContext that contains all schema components for the entire naming system. Of all the Novell service providers, the schema of the NDS provider is the only one with a dynamic schema managed by the back-end data store. The following graphic shows JNDI attributes with the NDS schema.

Figure 1-9 *JNDI Attributes with NDS Schema*



Each context in a naming system has its own schema definition. You can find the definition using a DirContext's getSchemaClassDefinition( ) method. The schema has a root, which contains attribute type definitions, object class definitions, and attribute syntax definitions. You can access the schema root using a DirContext's getSchema( ) method.

The result of DirContext.getSchema( ) will be a DirContext with three subordinates, ObjectClassDefs, AttributeTypeDefs, and AttributeSyntaxDefs. Getting the schema class definition for the DirContext merely returns the schema for that object. Getting the schema returns the schema for the entire naming system.

An object class is a component of the schema that defines

- What attribute IDs an object can have.
- What objects a particular object can contain.
- What objects can contain a particular object.

An attribute type is a component of the schema that defines

- The constraints of an attribute.
- The syntaxID of an attribute.

Performing the getAttributes( ) method on a syntaxID returns a set of attributes that contains the syntax definitions of the attributes.

An attribute syntax is a component of the schema that defines the composition of an attribute value. In NDS, the syntax definition defines the class composition of an attribute value.

1.13 Glossary

The terminology in NJCL and JNDI may be familiar to you, but some terms are used in different ways. To ensure that you have a good understanding of NJCL and JNDI, a list of key terms with their defined uses is presented here. In some cases, links are included to more detailed discussions in other Concepts topics.

1.13.1 Atomic Name

The indivisible component of an object name. For further information, see Section 1.7, "Object Names," on page 26.

1.13.2 Attribute

Information associated with a directory object. An attribute consists of a type identifier and a set of attribute values. For a more complete description, see Section 1.11, “Attribute Concepts,” on page 32.

1.13.3 Binding

The association or mapping of an atomic name with an object. For a more complete discussion of binding, see “Context Binding” on page 23.

1.13.4 Composite Name

A name that spans multiple naming systems.

1.13.5 Composite Name Resolution

The process of resolving a name that spans multiple naming systems.

1.13.6 Composite Namespace

The arrangement of namespaces from autonomous naming systems to form one logical namespace. Sometimes referred to as a federated namespace.

1.13.7 Compound Name

A name in the namespace of a single naming system. It is a sequence of zero or more atomic names composed according to the naming convention of that naming system. For a detailed discussion of the different types of names, see “Object Names” on page 26.

1.13.8 Context

An object whose state is a set of bindings with distinct atomic names. See “Clarifying Contexts” on page 21 for more detailed information on the uses of context.

1.13.9 Context Factory

A specialization of an object factory that accepts information about how to create a context, such as a reference, and returns an instance of the context.

1.13.10 Directory

A network database of hierarchical sets of objects.

1.13.11 Directory Object

An object that is stored in the directory, which contains name and attribute information. Sometimes referred to as a directory entry.

1.13.12 Directory Service

Provides operations for creating, adding, removing, and modifying the attributes associated with objects in a directory.

1.13.13 Environment Properties

Configuration parameters used to customize the operation of an object or service provider.

1.13.14 Federated Naming System

An aggregate of autonomous naming systems that cooperate to support name resolution of composite names through a standard interface. Each member of the federation has autonomy in its choice of operations and naming conventions. A federated naming service provides operations on a federated naming system.

1.13.15 Federation

A binding between a context in one naming system and an object in another naming system. For a more detailed discussion, see "Context Federation" on page 24 in the Clarifying Contexts section.

1.13.16 Initial Context

The starting point for resolution of names in naming and directory operations. For a more detailed description, see "Initial Context" on page 22 in the Clarifying Contexts section.

1.13.17 Name

A human-friendly identifier for identifying an object or a reference to an object. For a discussion of the different types of names, see "Object Names" on page 26.

1.13.18 Name Resolution

The process of resolving a name to the object to which it is bound. See "Context Binding" on page 23 for more information on name resolution.

1.13.19 Name space

A set of all the names in a naming system or instance of a name system.

1.13.20 Naming convention

The set of syntactic rules that govern how a name is generated. These rules determine whether a name is valid or invalid in the context in which the name is used.

1.13.21 Naming Service

Provides the operations on a naming system. For more detailed information, see "Naming System Concepts" on page 29.

1.13.22 Naming System

A connected set of contexts of the same type (having the same naming convention). For more detailed information, see Section 1.9, “Naming System Concepts,” on page 29.

1.13.23 Object Factory

A producer of objects that accepts some information about how to create an object, such as a reference, and then returns an instance of that object.

1.13.24 Reference

Represents an object that exists outside the naming system. JNDI attempts to convert the reference into the object that it represents before returning it to the client.

1.13.25 Schema

Specifies the types of objects a directory may contain, and the mandatory and optional attributes that directory objects of different types are to have. It may also specify the structure of the namespace and the relationship between different types of objects. For a more detailed discussion, see “Schema Concepts” on page 32.

1.13.26 Search Filter

A logical expression specifying the attributes that the directory objects being requested should have and be used by the directory to locate those objects.

1.13.27 Service Provider

An implementation of a context or initial context that can be plugged in dynamically to the JNDI architecture to be used by an application. For a more detailed discussion, see “Service Provider Concepts” on page 30.

1.13.28 Subcontext

A context that is bound in another context of the same type (having the same naming convention).

Novell Services (including JNDI Providers)

2

The following services are currently implemented in Novell® naming systems:

The Novell services are provided in alphabetical order for reference purposes. The Authentication Services and Session Manager Services are not JNDI providers as are all the others.

2.1 Introduction

Before discussing each of the services included here, there are some concepts applicable to all of them that you need to understand.

- What are the important components (interfaces and classes)?
- What do you need to know to access these interfaces and classes?
- What features (methods) in these interfaces and classes are important?
- What information do you need in order to use these features (methods)?

The following topics contain generic concepts that apply to all the services:

NOTE: This new format has been implemented for nine of the Novell Services. The other service, NetWare® File System Provider, will be revised and updated to this new format in a future release of the NDK.

2.1.1 Important Components

Each of the important components (interfaces and classes) necessary for implementing the service are listed and briefly described. Hypertext links are provided to the Reference Guide documentation for each of these components.

2.1.2 Initial Context Implementations

This topic discusses how to construct an initial context implementation for the JNDI Providers. In order to access the service components you must construct an initial context implementation in one of three ways:

- An initial context environment parameter implementation

  The property java.naming.factory.initial contains the fully-qualified class name of the initial context factor. The class must implement the initialContextFactory interface and have a public constructor that does not take any arguments. JNDI will load the initial context factory class and then invoke getInitialContext() on it to obtain a DirContext instance to be used as the initial context.

  An application that wants to use this initial context must supply the java.naming.factory.initial property either in the environment passed to the InitialContext or InitialDirContext constructors, or as one of the program's system properties. If the property is supplied as part of the environment, the system property is not consulted.

  Following is an example of how an initial context factory implementation might be constructed:

  ```
  Hashtable properties = new Hashtable();

  properties.put("jndi.initial.context.factory", "<class name>";

  DirContext initCtx = new InitialDirContext(properties);
  ```

  After the initial context has been implemented, if you need to get the session environment object use similar code to the following:

  ```
  InitialContext initCtx = new InitialContext (hash);

  hash = initCtx.getEnvironment ();

  Session sess = (Session) hash.get (Environment.SESSION.OBJECT);
  ```

- A URL initial context implementation

  If a URL string is passed to the initial context, it will be resolved using the corresponding URL context implementation. This is independent of any initial context implementations obtained using java.naming.factory.initial environment or system property.

  The URL context implementation is obtained using an object factory for the URL scheme identified in the URL string. The factory's class name is of the form urlSchemeURLContextFactory in the package specified using the java.naming.factory.url.pkgs environment or system property, which contains a colon-separated list of package prefixes. Each package prefix in this property is tried in the order specified to load the factory class.

  The object factory class implements the objectFactory interface and has a public constructor that takes no arguments. It provides a getObjectInstance() method, which will create instances of DirContext for the URL scheme. These instances will then be used to carry out the originally intended DirContext operation on the URL supplied to the initial context.

A two-step example using QMS is as follows:

```
contextFactory = new
com.novell.services.qms.naming.QMSInitialContextFactory(env);
```

```
<instance> = factory.GetInitialContext(env);
```

- An initial context factory builder implementation

  If an initial context factory builder has been installed, the application is effectively defining its own policy on how to locate and construct initial context implementations. When a factory has been installed, it is solely responsible for creating the initial context implementation. None of the default policies (java.naming.factory.initial property or URLcontext implementations) normally used by JNDI are employed.

  A service provider for an initial context factory builder must define a class that implements InitialContextFactoryBuilder. This class's createInitialContextFactory() method generates instances of InitialContextFactory. An application that wants to use this factory must first install it as follows:

  ```
  NamingManager.setInitialContextFactoryBuilder (factory);
  ```

2.1.3 Important Methods

The important methods provided by the service components are listed and briefly described. Hypertext links to the Reference Guide documentation are provided for each method listed. For a complete list of available methods you must go to the Reference Guide documentation.

2.1.4 Relationship of Classes and Interfaces

The relationship among the service components (interfaces and classes) is illustrated in diagrams. This is followed by a listing of the components and a discussion of the relationships among them. Hypertext links to the Reference Guide are provided for the most important service components.

2.2 Authentication Services

2.2.1 Introduction

The Authenticator enables application level authentication by prompting for user credentials when an authentication request is issued. Authentication is achieved via the Java class com.novell.java.security.Authenticator (../api/com/novell/java/security/Authenticator.html). The Authenticator uses the underlying requester to perform the actual authentication. The SessionManager and therefore the JNDI providers use the Authenticator for all authentication processes including login and logout. The Authentication service also provides the ability to create and change user passwords.

To take advantage of Network Attached Storage (NAS) login capability, call the setUseNAS (../api/com/novell/service/security/NdsIdentity.html#setUseNAS(boolean)) method.

Seven samples that demonstrate how to use the Authenticator can be found in "Authenticator Samples" on page 135 of Sample Code and Demos. The samples demonstrate how to perform a login, logout, token creation, token modification, token verification, and getting all authenticated identities.

The Authenticator is mechanism and storage independent. It is responsible for managing the authentication process for identities within a JVM. The Authenticator is designed to avoid two common types of application dependencies: authentication mechanism dependence and authentication secrets storage dependency. These design choices were made for two reasons:

- To allow Administrators to make policy decisions regarding authentication mechanisms rather than programmers.
- To insulate application developers from worrying about long-term secret storage.

By encapsulating the authentication protocols and long-term secret storage within the Authenticator, applications are not built with such dependencies. This provides administrators with the flexibility to decide the type of authentication mechanism employed in the organization, and it enables an organization to design and deploy schemes that can be verified at a particular security level.

For example, suppose a corporation decides on a security policy that uses smart-cards for all authentication within the corporation. After deployment of the smart-card systems an enterprise e-mail system is purchased that employs a password authentication mechanism. To integrate and deploy the e-mail application, the corporation has four choices:

- Issue common credentials for every employee for the e-mail application, thereby circumventing the non-repudiation achieved with the smart-cards.
- Issue unique credentials for every employee for the e-mail application, which increases the work for both the administrators and the users.
- Convince the vendor to support the corporation's smart-cards and wait for the upgrade.
- Choose another e-mail system that supports the corporate security policy.

Essentially, the customer either suffers with a less than satisfactory application, or the application distributor loses a customer to one of its competitors.

Since the Authenticator is mechanism and storage independent, applications that use this API do not suffer from these problems.

NOTE: The APIs required for the construction of authentication protocols and long-term authentication secrets are not part of the public release at this time; they will be made public in a future version.

2.2.2 Authentication Components

The important components (interfaces and classes) of Authentication are:

- Authenticator (../api/com/novell/java/security/Authenticator.html) class manages the authentication process for identities within a JVM. The Authenticator achieves application authentication by prompting for user credentials when an authentication request is issued.

- Identity (../api/com/novell/java/security/Identity.html) class represents identities, which are real-world objects, such as people, companies, and organizations whose identities can be authenticated using their public keys, or more abstract (or concrete) constructs such as software services.
- Principal (../api/com/novell/java/security/Principal.html) interface represents a principal, which can be an individual, a company, a program thread, or anything that can have an identity.
- PublicKey (../api/com/novell/java/security/PublicKey.html) interface serves to group and provide type safety for all public key interfaces. All specialized public key interfaces extend this PublicKey interface.
- Key (../api/com/novell/java/security/Key.html) interface is the top level interface for all keys. It defines the functionality shared by all key objects.
- IdentityScope (../api/com/novell/java/security/IdentityScope.html) class represents a scope for identities. It is an identity itself, and therefore has a name, can have a scope, and can optionally have a public key and associated certificates.
- Password (../api/com/novell/service/security/Password.html) class creates and holds passwords.
- PasswordIdentity (../api/com/novell/service/security/PasswordIdentity.html) interface identifies an object that can have a password set without using the authentication GUI. It is intended that an object implementing this interface also extends the Identity class.
- PasswordIdentityFactory (../api/com/novell/service/security/PasswordIdentityFactory.html) interface identifies an object that can create PasswordIdentity objects.
- BinderyIdentity (../api/com/novell/service/security/BinderyIdentity.html) class represents an authenticatable entity in a bindery system (a user, an administrator, and so forth). Every BinderyIdentity is associated with a server identity scope, which is then associated with the bindery administrative domain scope.
- BinderyIdentityScope (../api/com/novell/service/security/BinderyIdentityScope.html) class represents an authenticatable container in a bindery system, such as a server.
- BinderyPasswordIdentity (../api/com/novell/service/security/BinderyPasswordIdentity.html) class represents a bindery identity with a password.
- NdsIdentity (../api/com/novell/service/security/NdsIdentity.html) class represents an authenticatable entity in NDS, such as a user, an administrator, and so forth. Every NdsIdentity is associated with an identity scope, which is typically an NDS context.
- NdsIdentityScope (../api/com/novell/service/security/NdsIdentityScope.html) class represents an authenticatable container in NDS such as a context or a tree.
- NdsPasswordIdentity (../api/com/novell/service/security/NdsPasswordIdentity.html) class represents an NDS identity with a password.
- XplatIdentity (../api/com/novell/service/security/XplatIdentity.html) class represents an authenticatable entity in Xplat such as a user, an administrator, and so forth.
- XplatIdentityScope (../api/com/novell/service/security/XplatIdentityScope.html) class represents an authenticatable container in a Xplat system such as a server.

2.2.3 Important Authentication Methods

All methods of the Authenticator use an Identity class, which can represent any authenticatable object including user objects. A concrete subclass of Identity, which represents the entity being authenticated, should be constructed and passed to the Authenticator. For example, NdsIdentity may be used to specify an authenticatable NDS user. The Login sample shows NdsIdentity being instantiated and later passed to the Authenticator.login() method.

The IdentityScope object is an Identity that can contain many Identities. IdentityScopes are used to hierarchically associate related Identities. For example, NdsIdentityScope is used to specify each NDS context for a user object.

Listed below are some important Authentication methods of which to be aware. For a complete list of available methods, go to the two security packages in the API Reference documentation - com.novell.java.security package (../api/com/novell/java/security/package-summary.html) and com.novell.service.security package (../api/com/novell/service/security/package-summary.html).

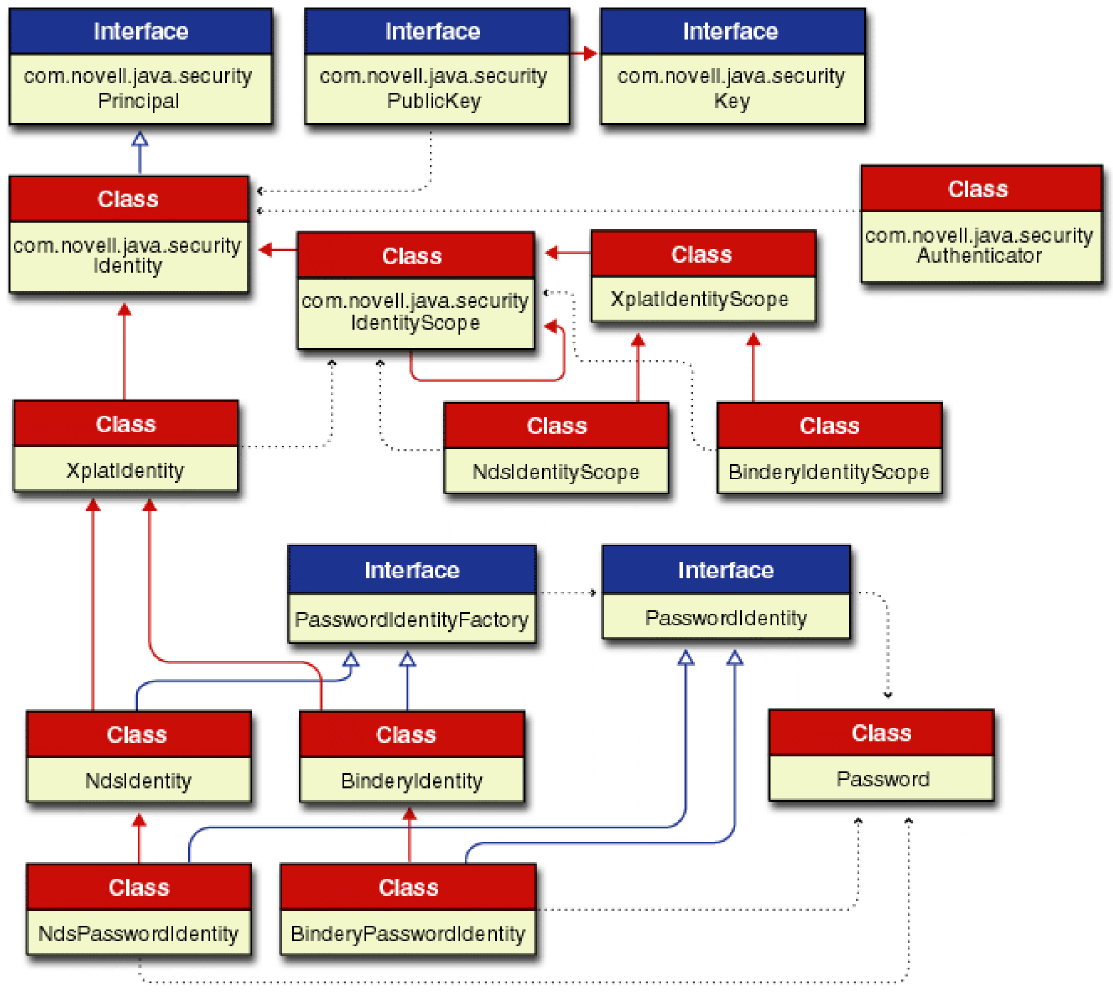
- Authenticator.login (../api/com/novell/java/security/Authenticator.html#login(com.novell.java.security.Identity)) - Performs a login of the specified Identity to its scope.
- Authenticator.logout (../api/com/novell/java/security/Authenticator.html#logout(com.novell.java.security.Identity)) - Logs out the specified Identity object from its identity scope and destroys the associated short-term credentials.
- Authenticator.createTokens (../api/com/novell/java/security/Authenticator.html#createTokens(com.novell.java.security.Identity, com.novell.java.security.Identity)) - Creates new authentication login secrets or tokens for an Identity object. Typically, administrators use this method to initialize a user object with some secret. For password systems, this means creating a new user password.
- Authenticator.modifyTokens (../api/com/novell/java/security/Authenticator.html#modifyTokens(com.novell.java.security.Identity)) - Modifies existing authentication secrets or tokens for an Identity object. Typically, end-users use this method to modify their authentication tokens. For password systems this means changing a user password.
- Authenticator.verifyTokens (../api/com/novell/java/security/Authenticator.html#verifyTokens(com.novell.java.security.Identity)) - Verifies existing authentication login secrets or tokens for an Identity object. For password systems this means verifying a user password.
- Authenticator.getIdentities (../api/com/novell/java/security/Authenticator.html#getIdentities()) - Returns a list of currently authenticated Identity objects.
- Authenticator.getIdentities (../api/com/novell/java/security/Authenticator.html#getIdentities(com.novell.java.security.IdentityScope)) - Returns a list of currently authenticated Identity objects within the specified scope.
- Password.setPassword (../api/com/novell/service/security/Password.html#setPassword(com.novell.service.security.Password))- Set the password to equal the specified Password.
- PasswordIdentity.setPassword (../api/com/novell/service/security/PasswordIdentity.html#setPassword(com.novell.service.security.Password)) - Set the password for this identity to the specified Password.
- PasswordIdentity.setNewPassword (../api/com/novell/service/security/PasswordIdentity.html#setNewPassword(com.novell.service.security.Password)) - Set the new password for this identity to the specified Password.
- PasswordIdentity.getPassword (../api/com/novell/service/security/PasswordIdentity.html#getPassword()) - Return the password for this identity.

- PasswordIdentity.getNewPassword (../api/com/novell/service/security/PasswordIdentity.html#getNewPassword()) - Return the new password for this identity.
- PasswordIdentityFactory.getPasswordIdentityInstance (../api/com/novell/service/security/PasswordIdentityFactory.html#getPasswordIdentityInstance()) - Instantiate a password identity class.

2.2.4 Relationship of Authentication Classes and Interfaces

The relationship between the Authenticator interfaces and classes is illustrated in the diagram below, followed by a brief description of the relationship each interface and class has with other interfaces and classes.

Figure 2-1 *Authentication Interfaces and Classes*



In this diagram of interfaces and classes a solid blue line ending with a hollow arrow represents an implementation of an interface. A solid red line ending with a solid arrow represent an extension of a class. The dotted black lines ending with a small solid arrow imply some relationships between the different interfaces and classes. Links are provided to the Reference Guide documentation for each of these important authentication interfaces and classes.

- Authenticator (../api/com/novell/java/security/Authenticator.html) class manages the authentication process for identities, providing methods for logging in and out; creating, verifying and modifying tokens; and getting currently authenticated identities within a given scope. It obtains the scope from the Identity class.
- Identity (../api/com/novell/java/security/Identity.html) class implements the Principal interface, and is extended by the IdentityScope, LdapIdentity, and XplatIdentity classes. It also represents the entity that is authenticated by the Authenticator class.
- Principal (../api/com/novell/java/security/Principal.html) interface is extended by the Identity class.
- PublicKey (../api/com/novell/java/security/PublicKey.html) interface extends the Key interface, and is used by several methods in the Identity class.
- Key (../api/com/novell/java/security/Key.html) interface is extended by the PublicKey interface.
- IdentityScope (../api/com/novell/java/security/IdentityScope.html) is an implementation class that extends the Identity class and provides the scope for the identity, and it is extended by the XplatIdentityScope class.
- XplatIdentityScope (../api/com/novell/service/security/XplatIdentityScope.html) is an implementation class that extends the IdentityScope class, and is extended by the BinderyIdentityScope and NdsIdentityScope classes.
- BinderyIdentityScope (../api/com/novell/service/security/BinderyIdentityScope.html) is an implementation class that extends the XplatIdentityScope class.
- NdsIdentityScope (../api/com/novell/service/security/NdsIdentityScope.html) is an implementation class that extends the XplatIdentityScope class.
- Password (../api/com/novell/service/security/Password.html) class constructs and holds passwords, and provides these passwords to the PasswordIdentity interface.
- PasswordIdentity (../api/com/novell/service/security/PasswordIdentity.html) interface identifies an object that can have a password set without using the authentication GUI. An object implementing this interface should also extend the Identity class.
- PasswordIdentityFactory (../api/com/novell/service/security/PasswordIdentityFactory.html) interface is implemented by the NdsIdentity, BinderyIdentity and LdapIdentity classes. It identifies an object that can create PasswordIdentity objects, and it provides a method for instantiating a PasswordIdentity class.
- BinderyIdentity (../api/com/novell/service/security/BinderyIdentity.html) class extends the XplatIdentity class and implements the PasswordIdentityFactory interface. It is also extended by the BinderyPasswordIdentity class.
- BinderyPasswordIdentity (../api/com/novell/service/security/BinderyPasswordIdentity.html) class extends the BinderyIdentity class and implements the PasswordIdentity interface.
- NdsIdentity (../api/com/novell/service/security/NdsIdentity.html) class extends the XplatIdentity class and implements the PasswordIdentityFactory interface. It is also extended by the NdsPasswordIdentity class.
- NdsPasswordIdentity (../api/com/novell/service/security/NdsPasswordIdentity.html) class extends the NdsIdentity class and implements the PasswordIdentity interface.

- XplatIdentity (../api/com/novell/service/security/XplatIdentity.html) class extends the Identity class, and is extended by the BinderyIdentity and NdsIdentity classes.

2.3 Bindery Provider

2.3.1 Introduction

The Bindery provider for JNDI provides contexts that implement the behavior of the NetWare 3.x Bindery, such as creating, deleting and renaming Bindery objects and references to servers and trees, and discovering and accessing the objects present in the Bindery.

NetWare 3.x servers use the bindery database to identify and store information about network objects. The Bindery name provider allows applications to read and modify standard information stored in the bindery and to create and manage their own object data.

The NetWare 4.0 OS replaced the bindery with NDS (an object database), which offers many advantages over the bindery, including a hierarchical structure and global naming.

However, to maintain compatibility with bindery-based servers and to work effectively with such NetWare features as file trustee rights, NDS provides built-in bindery context. This is provided by a bindery-like database maintained by NDS for objects contained in the local directory partitions of a server.

NDS generates object IDs and makes them available to bindery clients using the local file system, queue management system, and other bindery-oriented services. These values are dynamic, not remaining consistent over time; however, NDS object IDs and the object IDs returned by Bindery are the same.

Some of the objects within a server's bindery represent links to objects that are contained in other name spaces (such as servers, trees, and queues). Instances of these objects will support federation to their appropriate name spaces.

The Bindery serves many purposes, some of which are the following:

- Identifying users of the file system

  This is done through login control and file trustee rights. Users, user groups, print servers, and other objects that require access to the NetWare file system must be represented in the bindery.
- Network advertising

  The bindery advertises services and circulates their network addresses to NetWare servers and other network services.
- Storing application-specific data

  For example, applications often use the bindery to maintain lists of users that can access the application services.

A bindery object is created by calling the createSubcontext() method on the BinderyDirContext object, passing it a set of attributes defining the new bindery object. The caller is required to define values for the Object Type, Object Flags and Object Security attributes. The server will assign values for the Object ID and Has-Properties Flag attributes. Additional properties can also be added as attributes either through the createSubcontext() method or through a modifyAttributes() request.

Each bindery object can be assigned one or more properties. These can be considered optional attributes as opposed to the mandatory attributes described above. Properties identify categories of information associated with an object. Each property provides storage space appropriate to the associated values. The Bindery provider represents these properties as dynamic attributes, using the property name as the attribute ID.

2.3.2 Bindery Components

The important components (interfaces and classes) of Bindery are the following:

- BinderyDirContext (../api/com/novell/service/bindery/BinderyDirContext.html) class implements the JNDI DirContext interface (via AtomicDirContext), and represents the NetWare Bindery for a given server. The bindery objects are returned as instances or subclasses of the BinderyObjectDirContext class.
- BinderyObjectDirContext (../api/com/novell/service/bindery/BinderyObjectDirContext.html) class represents an individual object found in the NetWare bindery, extensions of which include QueueBinderyObjectDirContext, ServerBinderyObjectDirContext and TreeBinderyObjectDirContext, which represent dynamic queue, server and tree entries, respectively, in the bindery.
- QueueBinderyObjectDirContext (../api/com/novell/service/bindery/QueueBinderyObjectDirContext.html) class represents a bindery object that is a Queue in the NetWare 3.x Bindery, and is responsible for federating from the Bindery name system to the QMS name system. The lookup() method is what causes the implicit federation to the Queue context. You must use a trailing slash "/" to perform the federation.
- ServerBinderyObjectDirContext (../api/com/novell/service/bindery/ServerBinderyObjectDirContext.html) class represents a bindery object that is a Server in the NetWare 3.x Bindery, and is responsible for federating from the Bindery name system to the Server name system. The lookup() method is what causes the implicit federation to the Server context. You must use a trailing slash "/" to perform the federation.
- TreeBinderyObjectDirContext (../api/com/novell/service/bindery/TreeBinderyObjectDirContext.html) class represents a bindery object that is an NDS Tree in the NetWare 3.x Bindery, and is responsible for federating from the Bindery name system into the NDS name system. With the exception of lookup(), methods you perform on a TreeBinderyObjectDirContext are performed by its parent class, BinderyObjectDirContext. The lookup() method is the only one that TreeBinderyObjectDirContext overloads, and is what causes the implicit federation to the NDS context. You must use a trailing slash "/" to perform the federation.
- BinderyPropertyAttrVal (../api/com/novell/service/bindery/BinderyPropertyAttrVal.html) class holds all information that is found in a Bindery Property, and is used as an Attribute Value attached to the Properties Attribute of a BinderyObject.
- BinderyPropertyDataSegment (../api/com/novell/service/bindery/BinderyPropertyDataSegment.html) class holds a 128-byte data segment containing the data of a Bindery Property, and is used as an Attribute Value attached to the Bindery Properties attribute.

- BinderyEnvironment (../api/com/novell/service/bindery/BinderyEnvironment.html) class defines the set of constant strings that uniquely controls the behavior of the bindery name provider. It also contains convenience functions for managing the bindery objects.

2.3.3 Bindery Initial Context Implementation

In order to access the Bindery interfaces and classes you must specify an initial context factory to use, or you can access the Bindery through federation from another name space. Following is an example of how the Bindery initial context factory implementation might be done.

```
Hashtable properties = new Hashtable();

property.put(Context.INITIAL_CONTEXT_FACTORY,

   "com.novell.service.bindery.BinderyInitialContextFactory");

property.put(Context.PROVIDER_URL, <server name>);

DirContext initCtx = new InitialDirContext(properties);
```

To access the Bindery through federation you might do the following:

```
serverDirContext = lookup(<server name>+"/");
```

For a detailed discussion of initial context implementation see "Initial Context Implementations" on page 40 in the Novell Services Introduction.

2.3.4 Important Bindery Methods

Listed below are some important Bindery methods, all of which are inherited from Javasoft's Context and/or DirContext interfaces. All of the following methods can be accessed by going to the Javasoft JNDI 1.2 Specification (http://java.sun.com/products/jndi/1.2/javadoc/) Web Page, and from there you access the class and then the method index. For a list of additional methods, go to the Bindery package (../api/com/novell/service/bindery/package-summary.html) of the Reference Guide documentation.

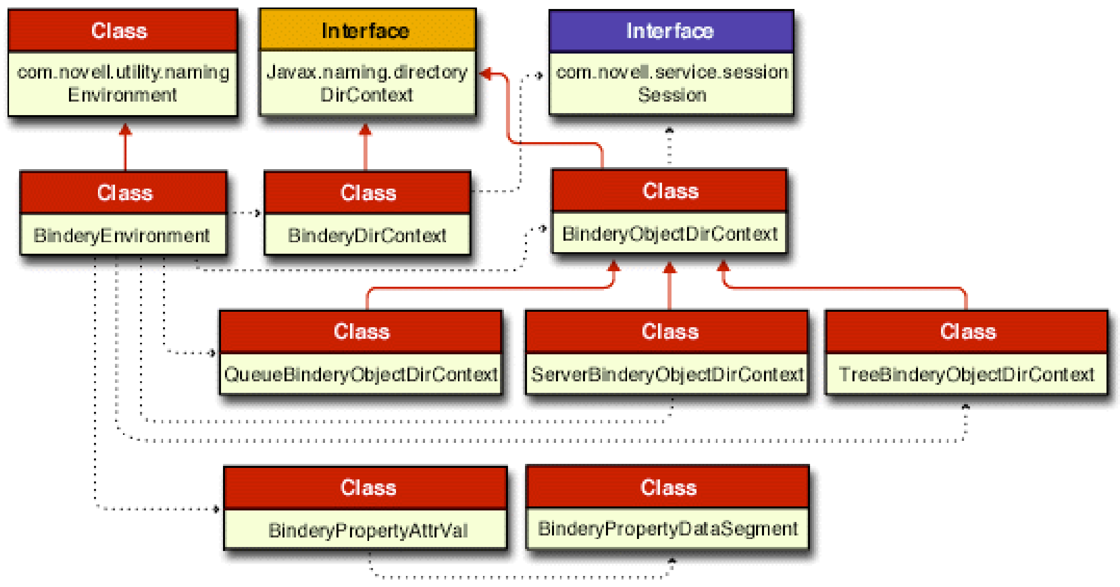
- DirContext.createSubcontext(Name, Attributes) (http://java.sun.com/products/jndi/1.2/javadoc/javax/naming/directory/DirContext.html#createSubcontext(javax.naming.Name, javax.naming.directory.Attributes)) - Creates a new bindery object with the passed in set of attributes, and binds it in the target context. You must define values for the object type, object flags and object security. The server will assign values for the object ID and the Has-Properties flag.
- DirContext.getAttributes(String, String[]) (http://java.sun.com/products/jndi/1.2/javadoc/javax/naming/directory/DirContext.html#getAttributes(javax.lang.String, javax.lang.String)) - Retrieves selected attributes associated with a string-named object.
- DirContext.getSchema(String) (http://java.sun.com/products/jndi/1.2/javadoc/javax/naming/directory/DirContext.html#getSchema(javax.lang.String)) - Retrieves the schema associated with the string-named object.
- Context.list(String) (http://java.sun.com/products/jndi/1.2/javadoc/javax/naming/Context.html#list(javax.lang.String)) - Returns an enumeration of the names and the class names of their bound objects in the string-named context.

- Context.lookup(String) (http://java.sun.com/products/jndi/1.2/javadoc/javax/naming/Context.html#lookup(javax.lang.String)) - Retrieves the named object in the server naming system using its string name. Use a trailing slash "/" to indicate implicit federation to the next naming system (Queue, Server, or Tree).
- DirContext.modifyAttributes(String, int, Attributes) (http://java.sun.com/products/jndi/1.2/javadoc/javax/naming/directory/DirContext.html#modifyAttributes(javax.lang.String, int, javax.naming.directory.Attributes)) - Modifies the attributes associated with a string-named object.
- DirContext.modifyAttributes(String, ModificationItem[]) (http://java.sun.com/products/jndi/1.2/javadoc/javax/naming/directory/DirContext.html#modifyAttributes(javax.lang.String,%20javax.naming.directory.ModificationItem) - Modifies the attributes associated with a string-named object using an ordered list of modifications.
- DirContext.search(String, Attributes) (http://java.sun.com/products/jndi/1.2/javadoc/javax/naming/directory/DirContext.html#search(javax.lang.String, javax.naming.directory.Attributes)) - Searches in a single context for objects that contain a specified set of attributes and retrieve their attributes.
- DirContext.search(String, String, Object, SearchControls) (http://java.sun.com/products/jndi/1.2/javadoc/javax/naming/directory/DirContext.html#search(javax.lang.String, javax.lang.String, javax.langObject, javax.naming.directory.SearchControls)) - Searches in the named context or object for entries that satisfy the given search filter.
- DirContext.search(String, String, SearchControls) (http://java.sun.com/products/jndi/1.2/javadoc/javax/naming/directory/DirContext.html#search(javax.lang.String, javax.lang.String, javax.naming.directory.SearchControls)) - Searches in the named context or named object for entries that satisfy the given search filter.

2.3.5 Relationship of Bindery Classes and Interfaces

The relationship between the Bindery interfaces and classes is illustrated in the diagram below, followed by a brief description of the relationship each interface and class has with other interfaces and classes.

Figure 2-2 *Bindery Interfaces and Classes*



Class
com.novell.utility.naming Environment
Interface
Javax.naming.directory DirContext
Interface
com.novell.service.session Session
Class
BinderyEnvironment
Class
BinderyDirContext
Class
BinderyObjectDirContext
Class
QueueBinderyObjectDirContext
Class
ServerBinderyObjectDirContext
Class
TreeBinderyObjectDirContext
Class
BinderyPropertyAttrVal
Class
BinderyPropertyDataSegment

In this diagram of interfaces and classes a solid red line ending with a solid arrow represents an extension of a class. The dotted black lines ending with a small solid arrow imply some relationships between the different interfaces and classes. Links are provided to the Reference Guide documentation for each of these important bindery interfaces and classes.

- BinderyDirContext (../api/com/novell/service/bindery/BinderyDirContext.html) class extends the JNDI DirContext interface (via AtomicDirContext), and gets a session environment object from the Session interface.
- BinderyObjectDirContext (../api/com/novell/service/bindery/BinderyObjectDirContext.html) class extends the JNDI DirContext interface (via AtomicDirContext), and is extended by the QueueBinderyObjectDirContext, ServerBinderyObjectDirContext, and TreeBinderyObjectDirContext classes. It also gets a session environment object from the Session interface.
- QueueBinderyObjectDirContext (../api/com/novell/service/bindery/QueueBinderyObjectDirContext.html) class extends the BinderyObjectDirContext class, and is used to federate into the Queue name space.
- ServerBinderyObjectDirContext (../api/com/novell/service/bindery/ServerBinderyObjectDirContext.html) class extends the BinderyObjectDirContext class, and is used to federate into the Server name space.
- TreeBinderyObjectDirContext (../api/com/novell/service/bindery/TreeBinderyObjectDirContext.html) class extends the BinderyObjectDirContext class, and is used to federate into the NDS name space.

- BinderyPropertyAttrVal (../api/com/novell/service/bindery/BinderyPropertyAttrVal.html) class holds all information found in a Bindery Property, and is used as an attribute value attached to the properties attribute of a bindery object.
- BinderyPropertyDataSegment (../api/com/novell/service/bindery/BinderyPropertyDataSegment.html) class holds a 128-byte data segment containing the data of a bindery property, and is used as an attribute value attached to the bindery properties attribute.
- BinderyEnvironment (../api/com/novell/service/bindery/BinderyEnvironment.html) class extends the Environment class, and provides constants used by all other bindery classes.

2.4 NCP Extensions Provider

2.4.1 Introduction

The NCPExtensions provider for JNDI allows access to the NetWare operating system extensions by providing contexts that implement the behavior of NetWare NCP extensions, such as discovering and accessing NCP extensions.

The NCPExtension interface allows you to send requests to and receive replies from NCP extensions registered on a NetWare server. You cannot create new NCP extensions using JNDI; you can only access existing NCP extensions. You will usually use an NCP extension if you have written it yourself in C or if a third party has provided it.

You cannot enter the NCP Extensions naming system directly. Since an NCP extension is bound to a server, you can access it only from the Server naming system.

2.4.2 NCP Extensions Components

The important components (interfaces and classes) of NCP extensions are the following:

- NCPExtension (../api/com/novell/service/ncpext/NCPExtension.html) interface represents an NCP extension registered on the NetWare server that allows you to send requests to and receive replies from the NCP extension.
- NCPExtensionDirContext (../api/com/novell/service/ncpext/NCPExtensionDirContext.html) class represents an NCPExtension loaded on a NetWare server. An NCPExtensionDirContext has one attribute, NCPExtensionInfo, with a single value that is an instance of NCPExtensionInfoImpl.
- NCPExtensionInfo (../api/com/novell/service/ncpext/NCPExtensionInfo.html) interface provides an attribute of an NCP extension with a single value that is an instance of NCPExtensionInfoImpl.
- NCPExtensionInfoImpl (../api/com/novell/service/ncpext/NCPExtensionInfoImpl.html) class provides the value for attribute NCPExtensionInfo, and methods for getting information about the NCP extension loaded on the server.

2.4.3 NCP Extensions Initial Context Implementation

In order to access the NCP extensions interfaces and classes you must federate using the Server. Following is an example of how the NCPExtension federation from the Server might be done.

```
Hashtable properties = new Hashtable();

property.put(Context.INITIAL_CONTEXT_FACTORY,

   "com.novell.service.bindery.ServerInitialContextFactory");

property.put(Context.PROVIDER_URL, <server name>);

DirContext initCtx = new InitialDirContext(properties);
```

To access the NCPExtension through federation you might do the following:

```
serverDirContext = ctx;

ctx.lookup("ncpextensions"+"/");
```

For a detailed discussion of initial context implementation see "Initial Context Implementations" on page 40 in the Novell Services Introduction.

2.4.4 Important NCP Extensions Methods

Listed below are some important NCP extensions methods of which to be aware. For a complete list of available methods, go to the Reference Guide (../api/overview-summary.html) documentation.

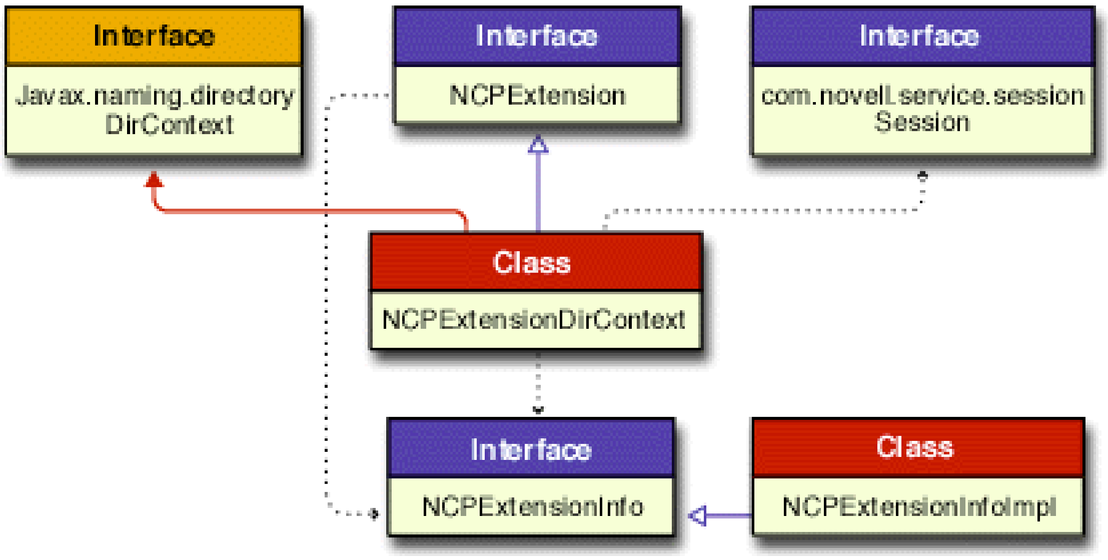
- NCPExtension.send (/api/com/novell/service/ncpext/NCPExtension.html) - Sends a request to an NCPExtension and receives a byte array containing the requested data.
- NCPExtensionInfo.getExtensionID (../api/com/novell/service/ncpext/NCPExtensionInfo.html#getExtensionID()) - Returns the NCP Extension ID.
- NCPExtensionInfo.setExtensionID (../api/com/novell/service/ncpext/NCPExtensionInfo.html#setExtensionID(int)) - Sets the NCP Extension ID.
- NCPExtensionInfo.getExtensionName (../api/com/novell/service/ncpext/NCPExtensionInfo.html#getExtensionName()) - Returns the NCP Extension name.
- NCPExtensionInfo.setExtensionName (../api/com/novell/service/ncpext/NCPExtensionInfo.html#setExtensionName(java.lang.String)) - Sets the NCP Extension name.
- NCPExtensionInfo.getMajorVersion (../api/com/novell/service/ncpext/NCPExtensionInfo.html#getMajorVersion()) - Returns the NCP Extension major version.
- NCPExtensionInfo.setMajorVersion (../api/com/novell/service/ncpext/NCPExtensionInfo.html#setMajorVersion(int)) - Sets the NCP Extension major version.
- NCPExtensionInfo.getMinorVersion (../api/com/novell/service/ncpext/NCPExtensionInfo.html#getMinorVersion()) - Returns the NCP Extension minor version.
- NCPExtensionInfo.setMinorVersion (../api/com/novell/service/ncpext/NCPExtensionInfo.html#setMinorVersion(int)) - Sets the NCP Extension minor version.
- NCPExtensionInfo.getQueryData (../api/com/novell/service/ncpext/NCPExtensionInfo.html#getQueryData()) - Returns the NCP Extension query data.

- NCPExtensionInfo.setQueryData (/api/com/novell/service/ncpext/NCPExtensionInfo.html) - Sets the NCP Extension query data.
- NCPExtensionInfo.getRevision (../api/com/novell/service/ncpext/NCPExtensionInfo.html#getRevision()) - Returns the NCP Extension revision.
- NCPExtensionInfo.setRevision (../api/com/novell/service/ncpext/NCPExtensionInfo.html#setRevision(int)) - Sets the NCP Extension revision.

2.4.5 Relationship of NCP Extensions Classes and Interfaces

The relationship between the NCP extensions interfaces and classes is illustrated in the diagram below, followed by a brief description of the relationship each interface and class has with other interfaces and classes.

Figure 2-3 *NCP Extensions Interfaces and Classes*



In this diagram of interfaces and classes a solid blue line ending with a hollow arrow represents an implementation of an interface. A solid red line ending with a solid arrow represent an extension of a class. The dotted black lines ending with a small solid arrow imply some relationships between the different interfaces and classes. Links are provided to the Reference Guide documentation for each of these important NCPExtension interfaces and classes.

- NCPExtension (../api/com/novell/service/ncpext/NCPExtension.html) interface is implemented by the NCPExtensionDirContext class.
- NCPExtensionDirContext (../api/com/novell/service/ncpext/NCPExtensionDirContext.html) class implements the NCPExtension interface and extends the Javasoft DirContext interface indirectly through the ComponenetDirContext class.
- NCPExtensionInfo (../api/com/novell/service/ncpext/NCPExtensionInfo.html) interface is implemented by the NCPExtensionInfoImpl class.
- NCPExtensionInfoImpl (../api/com/novell/service/ncpext/NCPExtensionInfoImpl.html) class implements the NCPExtensionInfo interface.

2.5 NDAP Provider for Novell Directory Services (NDS)

2.5.1 Introduction

The NDAP provider for NDS provides access to NDS services through JNDI. It also provides access to contexts that implement the behavior and attributes of NDS objects, NDS partitions, and the NDS schema. The detail specifications for the schema of the NDS naming system are provided in the *NDK: Novell eDirectory Schema Reference*, which is part of the NDS Libraries for C documentation. The NDAP provider for NDS requires the Novell client.

All NDS directory contexts in the NDS provider extend the NdsObject class, which provides methods that allow you to perform non-naming operations on JNDI contexts. Single attribute values in the NDS provider extend the NdsAttributeValue class, of which there are 28 separate subclasses representing NDS attribute syntaxes.

2.5.2 NDS Provider Components

The important components (interfaces and classes) of the NDS provider are the following:

- NdsObject (../api/com/novell/service/nds/NdsObject.html) interface is the root class for all contexts that represent NDS objects. It provides a window into functionality not provided by JNDI. NdsObject extends the DirContext, NdsIteratorFactory, Referenceable, and Partitionable interfaces. It defines methods dealing with rights as well as methods related to operational attributes.
- NdsAttributeValue (../api/com/novell/service/nds/NdsAttributeValue.html) interface defines a common way of manipulating and passing the value of any syntax object in the NDS schema to an interface that accepts NDS attribute values. The 28 subclasses representing single attribute syntax values extend the NdsAttributeValue interface. See the NDS Provider Package (../api/com/novell/service/nds/package-summary.html) for a listing and details of these subclasses.
- NdsIteratorFactory (../api/com/novell/service/nds/NdsIteratorFactory.html) interface generates NDS iterators used to create a large virtual list. It consists of four methods of the same name (createIterator) for creating NDS iterators. These methods differ only in the parameters passed in. See "NDS Iterator Implementation" on page 63 for further details relating to NDS iterator functionality.
- NdsNCPServer (../api/com/novell/service/nds/NdsNCPServer.html) interface defines NDS servers that provide NCP transport and session services. It is intended to represent both bindery-based and NDS-based NCP servers.
- NdsPartition (../api/com/novell/service/nds/NdsPartition.html) interface defines methods for aborting partition operations.

- NdsReplica (../api/com/novell/service/nds/NdsReplica.html) interface defines variables and methods for replica operations.
- SchemaAttribute (../api/com/novell/service/nds/SchemaAttribute.html) interface defines the attribute types and constraints (flags) that restrict the attribute values.
- SchemaClass (../api/com/novell/service/nds/SchemaClass.html) interface defines for each class the object components, which define the types of objects that can exist in the NDS tree.
- SchemaSyntax (../api/com/novell/service/nds/SchemaSyntax.html) interface defines the standard data types for the values within specified attribute types stored in NDS.

2.5.3 NDS Initial Context Implementation

In order to access the NDS interfaces and classes you must construct an initial context factory implementation. Following is an example of how the NDS initial context factory implementation might be constructed.

```
Hashtable properties = new Hashtable();

property.put(Context.INITIAL_CONTEXT_FACTORY,
   "com.novell.service.nds.naming.
       NdsInitialContextFactory");

property.put(Context.PROVIDER_URL, <server name>);

DirContext initCtx = new InitialDirContext(properties);
```

The following contexts federate to other naming systems:

- DirectoryMapDirContext implicitly federates to the "NetWare File System Provider" on page 68.
- VolumeDirContext implicitly federates to the "NetWare File System Provider" on page 68.
- NCPServerDirContext implicitly federates to the "NCP Extensions Provider" on page 52.
- QueueDirContext implicitly federates to the "Queue Management System (QMS) Provider" on page 81.

The following contexts extend NdsDirContext to allow access to partition and replica functionality, but they do not implicitly federate to other naming system.

- NdsPartitionDirContext
- NdsPartitionTreeRootContext
- NdsReplicaContext

After the initial context has been implemented, you can then get the session environment object using code similar to the following:

```
InitialContext initCtx = new InitialContext (hash);

hash = initCtx.getEnvironment ();

Session sess = (Session) hash.get (Environment.SESSION.OBJECT);
```

For a detailed discussion of initial context implementation, see "Initial Context Implementations" on page 40 in the Novell Services Introduction.

2.5.4 Important NDSProvider Methods

Listed below are some important NDS provider methods of which to be aware. For a complete list of available methods, go to the Reference Guide (../api/overview-summary.html) documentation for NJCL and JNDI.

- NdsObject.getAttributeEffectiveRights (../api/com/novell/service/nds/NdsObject.html#getAttributeEffectiveRights(com.novell.service.nds.NdsObject)) - Returns a NamingEnumerator of the effective rights for the specified NDS trustee object. There are 5 additonal versions of this method with different parameters.
- NdsObject.getBaseClass (../api/com/novell/service/nds/NdsObject.html#getBaseClass()) - Returns the name of the object class (immediate parent class) that was used to create the object.
- NdsObject.getDistinguishedName (../api/com/novell/service/nds/NdsObject.html#getDistinguishedName()) - Returns the distinguished name of the NDS object.
- NdsObject.getModificationTime (../api/com/novell/service/nds/NdsObject.html#getModificationTime()) - Returns the object modification time as a Date type.
- NdsObject.getObjectEffectiveRights (../api/com/novell/service/nds/NdsObject.html#getObjectEffectiveRights(com.novell.service.nds.NdsObject)) - Returns the effective rights of the specified NDS trustee object. Another version of this method uses a String object parameter.
- NdsObject.getObjectFlags (../api/com/novell/service/nds/NdsObject.html#getObjectFlags()) - Returns the DSI Entry Flag ID for the object. These 12 flags are defined in the NDS Libraries for C documentation (NSA > NDS: Values > DSI Entry Flags Values).
- NdsObject.getSubordinateCount (../api/com/novell/service/nds/NdsObject.html#getSubordinateCount()) - Returns the number of objects immediately subordinate to the DirContext.
- NdsObject.getTreeName (../api/com/novell/service/nds/NdsObject.html#getTreeName()) - Returns the name of the NDS tree where the object resides.
- NdsAttributeValue.approximate (../api/com/novell/service/nds/NdsAttributeValue.html#approximate(java.lang.Object)) - Compares two objects for approximate equality.
- NdsAttributeValue.getNdsSyntaxId (../api/com/novell/service/nds/NdsAttributeValue.html#getNdsSyntaxId()) - Returns the integer that represents the syntax ID.
- NdsAttributeValue.supportsMatchingRules (../api/com/novell/service/nds/NdsAttributeValue.html#supportsMatchingRules(int)) - Returns a Boolean indicating whether the matching rules passed in are supported.
- NdsIteratorFactory.createIterator (../api/com/novell/service/nds/NdsIteratorFactory.html#createIterator(java.lang.String, javax.naming.directory.Attributes)) - Creates an NdsIterator Object based on the parameters passed in. There are 5 versions of this method, each with a different parameter set.
- NdsNCPServer.getUTCTime (../api/com/novell/service/nds/NdsNCPServer.html#getUTCTime()) - Returns the Universal Coordinated Time (UTC) setting of the server.

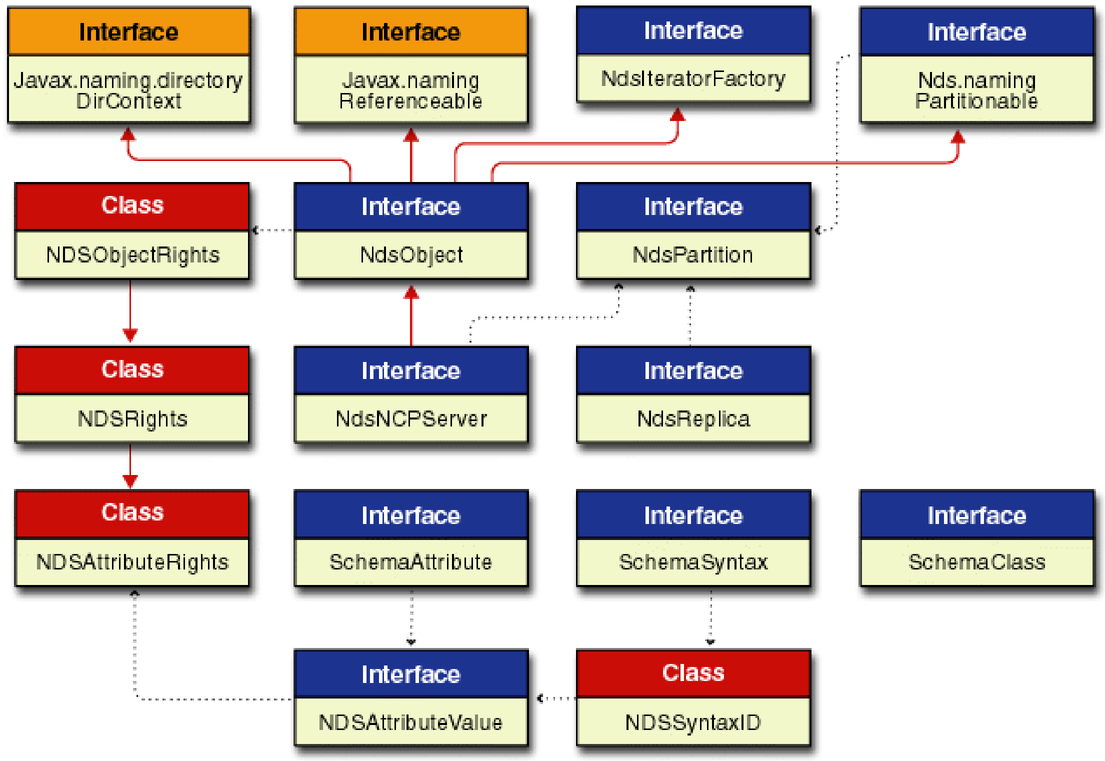
- NdsNCPServer.listPartitions (../api/com/novell/service/nds/NdsNCPServer.html#listPartitions()) - Provides a list of replica partition names stored on this NCP server.
- NdsPartition.abortOperations (../api/com/novell/service/nds/NdsPartition.html#abortOperations(javax.naming.Name)) - Aborts all pending partition operations. Another version of this method passes in a String parameter rather than Name.
- NdsReplica.receiveUpdates (../api/com/novell/service/nds/NdsReplica.html#receiveUpdates()) - Receives updates from the master replica.
- NdsReplica.sendUpdates (../api/com/novell/service/nds/NdsReplica.html#sendUpdates()) - Sends updates to all other replicas.
- NdsReplica.setReplicaType (../api/com/novell/service/nds/NdsReplica.html#setReplicaType(int)) - Sets the replica type for the partition.
- NdsReplica.syncReplica (../api/com/novell/service/nds/NdsReplica.html#syncReplica(int)) - Synchronizes replicas restricted by the specified time delay. Another version of this method specifies the server and flags.

Additional methods are available from the javax.naming.Context and javax.naming.directory.DirContext classes (http://java.sun.com/products/jndi/1.2/javadoc/overview-summary.html).

2.5.5 Relationship of NDS Provider Interfaces and Classes

The relationship between the NDS Provider interfaces and classes is illustrated in the following four diagrams.

Figure 2-4 *NDS Object Interfaces and Classes*



The NDS Object diagram shows the relationship of the different interfaces and classes to the NdsObject interface. In this diagram of interfaces and classes a solid red line ending with a solid arrow represent an extension of a class. The dotted black lines ending with a small solid arrow imply some relationships between the different interfaces and classes. Links are provided to the Reference Guide documentation for each of these important NDS object interfaces and classes.

The following links are provided for you to access the Reference Guide documentation for the most important interfaces and classes:

- NdsObject (../api/com/novell/service/nds/NdsObject.html) interface extends DirContext, Referencable, Partitionable, and NdsIteratorFactory interfaces, and is extended by the NdsNCPServer interface.
- NdsIteratorFactory (../api/com/novell/service/nds/NdsIteratorFactory.html) interface is extended by NdsObject. See Figure 2.9 for a graphical representation of the relationship between NDS iterator interfaces and classes.
- NdsNCPServer (../api/com/novell/service/nds/NdsNCPServer.html) interface extends NdsObject and has a relationship with NdsPartition.

The NdsAttributeValue interface relationships are shown in Figures 2-6, 2-7, 2-8, and 2-9. The other interfaces shown in the above diagram (NdsReplica, SchemaClass, SchemaAttribute and

SchemaSyntax) do not have direct relationships (implementation or extension) to other NDS interfaces and classes.

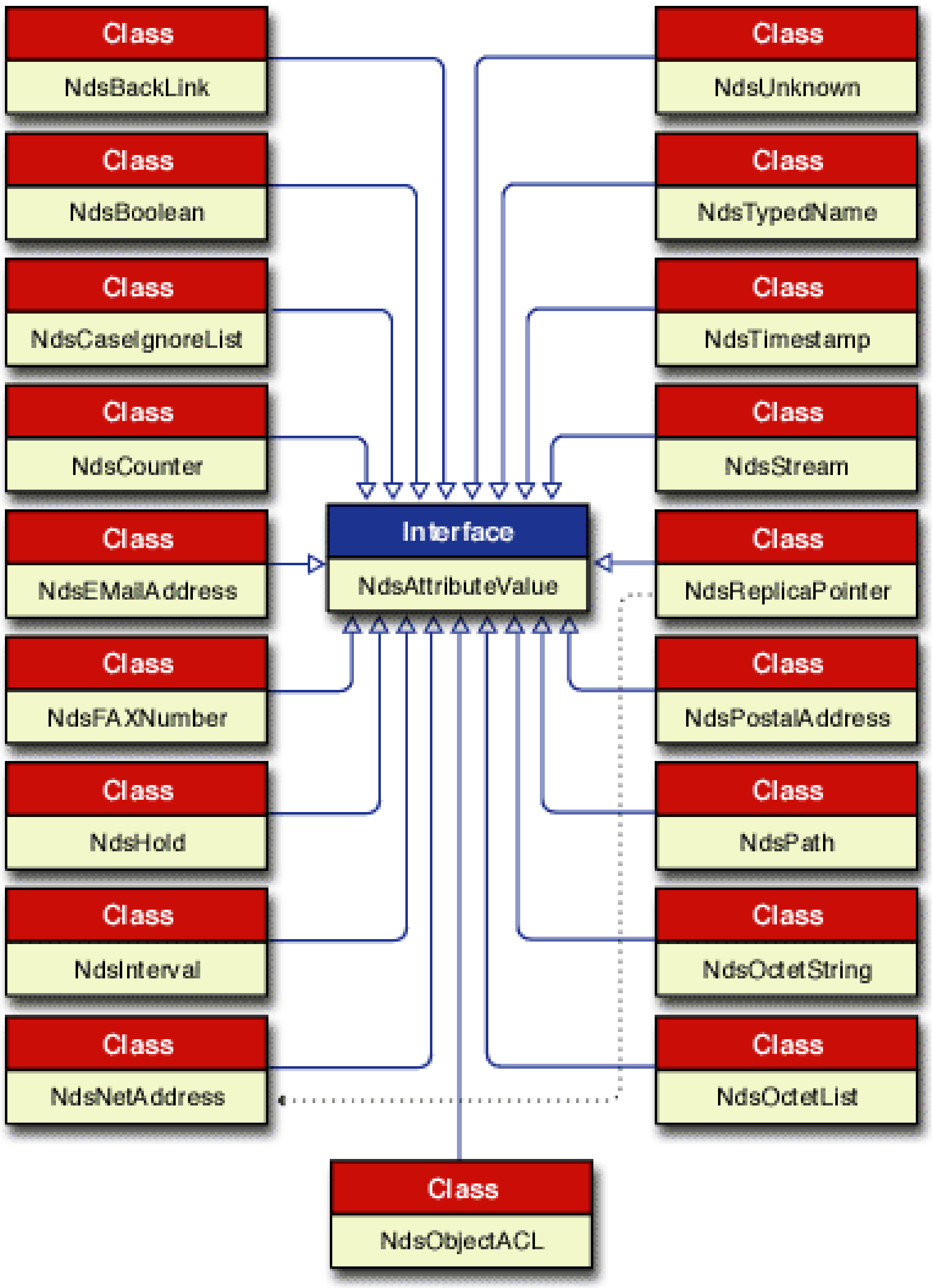


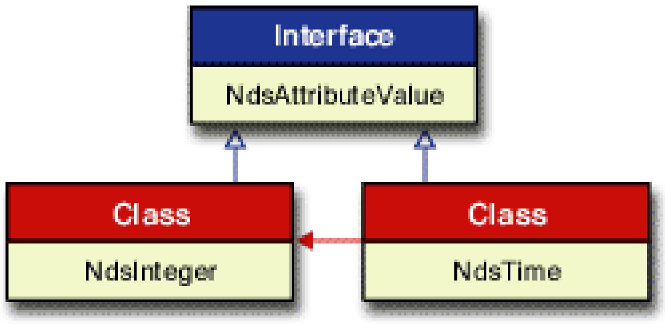
Figure 2-5 *NDS Attribute Value Interfaces and Classes*

The NDS Attribute Value diagram above shows the relationship of the NdsAttributeValue interface to 19 of the 28 subclasses (single attribute values). Each of these subclasses implements the NdsAttributeValue interface. In this diagram of interfaces and classes a solid blue line ending with a hollow arrow represents an implementation of an interface. The dotted black lines ending with a small solid arrow imply some relationships between the different interfaces and classes. Links are provided to the Reference Guide documentation for each of these important NDS attribute value interfaces and classes.

The following links are provided for you to access the Reference Guide documentation for the most important interfaces and classes:

- NdsAttributeValue (../api/com/novell/service/nds/NdsAttributeValue.html) interface is implemented by all subclasses (single attribute values).
- See the NDS Provider Package (../api/com/novell/service/nds/package-summary.html) for information on the subclasses that implement the NdsAttributeValue interface.

Figure 2-6 *NDS Integer Class Diagram*

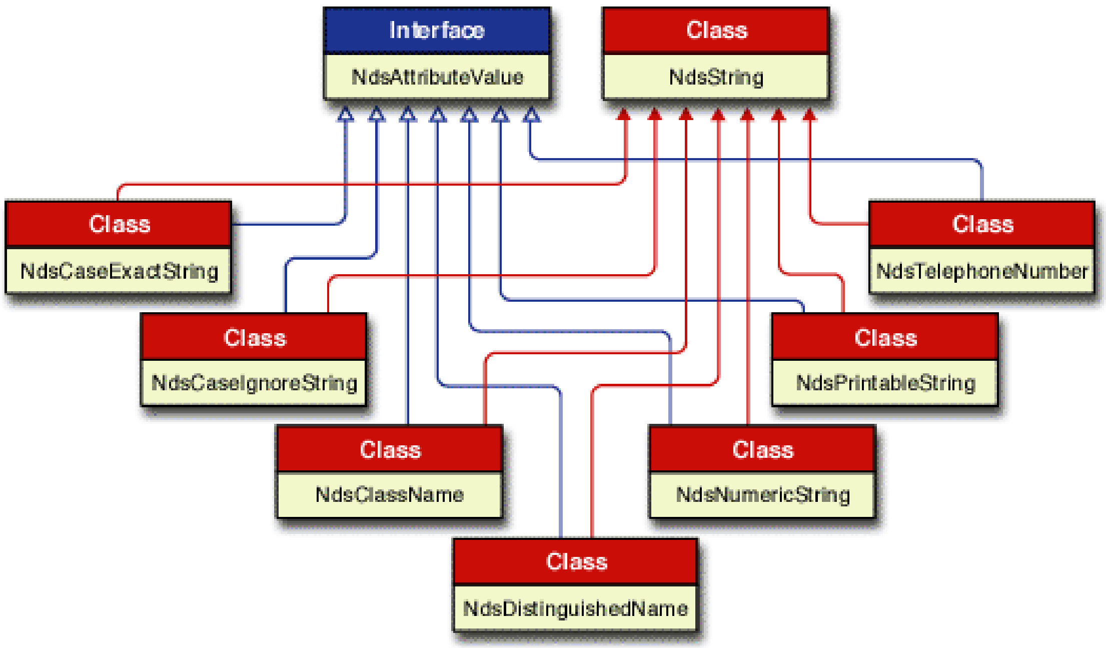


The NDS Integer diagram shows the relationship of the NdsInteger and NdsTime classes to the NdsAttributeValue interface. NdsTime extends NdsInteger and implements NdsAttributeValue. In this diagram of interfaces and classes a solid blue line ending with a hollow arrow represents an implementation of an interface. A solid red line ending with a solid arrow represent an extension of a class. Links are provided to the Reference Guide documentation for each of these important integer interfaces and classes.

The following links are provided for you to access the Reference Guide documentation for the most important interfaces and classes:

- NdsAttributeValue (../api/com/novell/service/nds/NdsAttributeValue.html) interface is implemented by NdsInteger and NdsTime.
- NdsInteger (../api/com/novell/service/nds/NdsInteger.html) class implements the NdsAttributeValue interface and is extended by NdsTime.
- NdsTime (../api/com/novell/service/nds/NdsTime.html) class implements the NdsAttributeValue interface and extends the NdsInteger class.

***Figure 2-7** NDS String Class Diagram*



The NDS String diagram shows the relationship of those subclasses that implement the NdsAttributeValue and extend the NdsString class. In this diagram of interfaces and classes a solid blue line ending with a hollow arrow represents an implementation of an interface. A solid red line ending with a solid arrow represent an extension of a class. Links are provided to the Reference Guide documentation for each of these important NDS string interfaces and classes.

The following links are provided for you to access the Reference Guide documentation for the most important interfaces and classes:

- NdsAttributeValue (../api/com/novell/service/nds/NdsAttributeValue.html) interface is implemented by NdsCaseExactString, NdsCaseIgnoreString, NdsClassName, NdsDistinguishedName, NdsNumericString, NdsPrintableString, and NdsTelephoneNumber.
- NdsString (../api/com/novell/service/nds/NdsString.html) class is extended by NdsCaseExactString, NdsCaseIgnoreString, NdsClassName, NdsDistinguishedName, NdsNumericString, NdsPrintableString, and NdsTelephoneNumber.
- NdsCaseExactString (../api/com/novell/service/nds/NdsCaseExactString.html) class provides access to attribute values of the CaseExactString syntax. It implements the NdsAttributeValue interface and extends the NdsString class.
- NdsCaseIgnoreString (../api/com/novell/service/nds/NdsCaseIgnoreString.html) class provides access to attribute values of the CaseIgnoreString syntax. It implements the NdsAttributeValue interface and extends the NdsString class.
- NdsClassName (../api/com/novell/service/nds/NdsClassName.html) class provides access to attribute values of the ClassName syntax. It implements the NdsAttributeValue interface and extends the NdsString class.
- NdsDistinguishedName (../api/com/novell/service/nds/NdsDistinguishedName.html) class provides access to attribute values of the DistinguishedName syntax. It implements the NdsAttributeValue interface and extends the NdsString class.

- NdsNumericString (../api/com/novell/service/nds/NdsNumericString.html) class provides access to attribute values of the NumericString syntax. It implements the NdsAttributeValue interface and extends the NdsString class.
- NdsPrintableString (../api/com/novell/service/nds/NdsPrintableString.html) class provides access to attribute values of the PrintableString syntax. It implements the NdsAttributeValue interface and extends the NdsString class.
- NdsTelephoneNumber (../api/com/novell/service/nds/NdsTelephoneNumber.html) class provides access to attribute values of the TelephoneNumber syntax. It implements the NdsAttributeValue interface and extends the NdsString class.

2.5.6 NDS Iterator Implementation

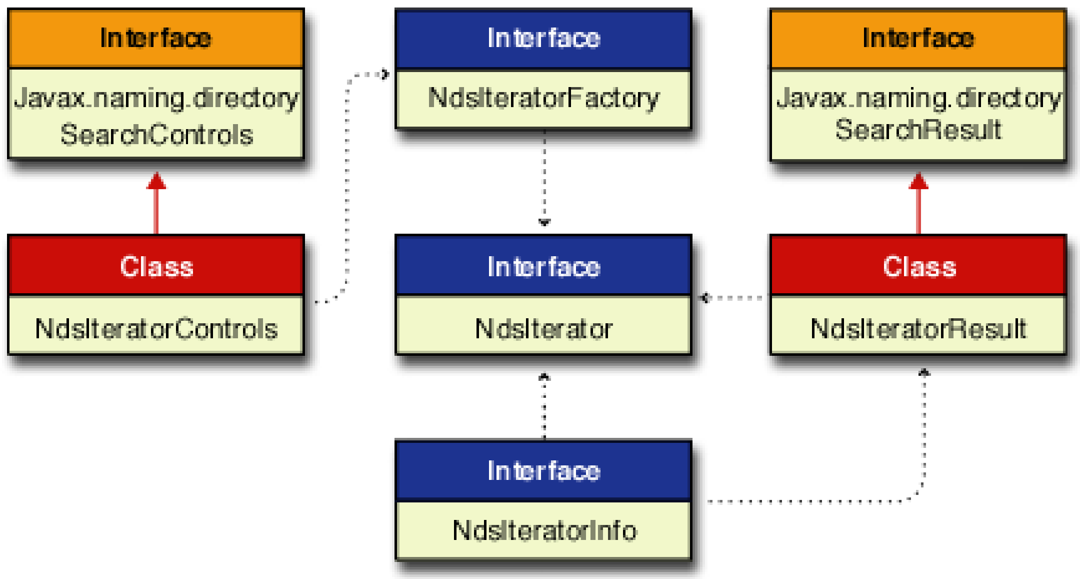
An Iterator in NDS is a concept for handling large virtual lists. It provides list and search functionality but can deal with very large result sets, on the order of hundreds of thousands. An iterator for contexts allows the programmer to traverse the search result in either direction, creating a large virtual list of objects on the server containing all the entries that pass the search filter. The search does not span servers, but is restricted to the particular server in question.

The iterator implementation for NDS requires the use of three interfaces and two classes:

- "NdsIteratorFactory Interface" on page 64
- "NdsIterator Interface" on page 66
- "NdsIteratorInfo Interface" on page 66
- "NdsIteratorControls Class" on page 67
- "NdsIteratorResult Class" on page 67

The following diagram (Figure 2-9) shows the relationship between these interfaces and classes. The subsections that follow present more detailed information on the NDS Iterator implementation.

Figure 2-8 *NDS Iterator Interfaces and Classes*



In this diagram of interfaces and classes a solid red line ending with a solid arrow represent an extension of a class. The dotted black lines ending with a small solid arrow imply some relationships between the different interfaces and classes. Links are provided in the following subsections to the Reference Guide documentation for each of these important NDS Iterator interfaces and classes.

2.5.7 NdsIteratorFactory Interface

The NdsIteratorFactory interface generates NDS iterators used to create a large virtual list. It consists of four methods of the same name (createIterator) for creating and defining NDS iterator searches. These methods differ in the parameters passed in. See the NdsIteratorFactory interface (../api/com/novell/service/nds/NdsIteratorFactory.html) in the Reference Guide for more information on the format for calling these methods.

The createIterator(String, Attributes, String[]) method searches the children of a single context for objects that contain a specified set of attributes, and then it returns the attributes specified in the attributesToReturn parameter in an NdsIterator object. The search is performed using the default javax.naming.directory.SearchControls settings.

For an object to be selected, each attribute in the matchingAttributes parameter must match some attribute of the object. If matchingAttributes is NULL all objects in the target context are returned. If matchingAttributes is empty no attributes will be returned. The precise definition of 'equality' used in comparing attribute values is defined by the underlying directory service, which might use a schema to specify a different equality operation. See the search(Name, Attributes, String[]) method in Sun’s JNDI 1.1.1 Javax.naming.directory.DirContext interface for a full description of this functionality.

The createIterator(String, Attributes) method searches for objects that contain a specified set of attributes, and then returns all the attributes in an NdsIterator object. It is equivalent to supplying NULL for the atributesToReturn parameter in the createIterator(String, Attributes, String[]) method described above.

The createIterator (String, String, NdsIteratorControls) method searches in the named context or named object for entries that satisfy the given search 'filter', and then returns the results as an NdsIterator object. It performs the search as specified by the controls cons parameter, which is defined in the NdsIteratorControls constructor.

The format for and the interpretation of ’filter’ follows RFC 2254 with the following interpretations for the attr and value variables mentioned in the RFC:

- The attr variable is a string representing the attribute’s identifier. This is equivalent to the ID parameter passed to the constructor of Attribute.
- The value variable is the string representation of the attribute’s value. The translation of this string representation into the attribute’s value is directory-specific.

For the assertion '(someCount=127)', for example, attr is 'someCount' and value is '127'. The provider determines that the attribute’s value is an integer, based on the attribute ID ('someCount') and possibly its schema. It then parses the string '127' appropriately. If the directory does not support a string representation of some or all of its attributes, the form of the search method that accepts filter arguments in the form of Objects can be used instead. The service provider for such a directory would then translate the filter arguments to its service-specific representation for filter evaluation.

RFC 2254 defines certain operators for the filter, including substring matches, equality, approximate match, greater than, less than, and so forth. These operators are mapped to operators with corresponding semantics in the underlying directory. For example, for the equals operator, suppose the directory has a matching rule defining 'equality' of the attributes in the filter. This rule would be used for checking equality of the attributes specified in the filter with the attributes of objects in the directory. Similarly, if the directory has a matching rule for ordering, this rule would be used for making 'greater than' and 'less than' comparisons. Not all of the operators defined in RFC 2254 are applicable to all attributes. When an operator is not applicable, the exception InvalidSearchFilterException is thrown.

The createIterator (String, String, Object[], NdsIteratorControls) method searches in the named context or named object for entries that satisfy the given search filter. It performs the search as specified by the search controls cons parameter, which is defined in the NdsIteratorControls constructor.

The filterExpr parameter is interpreted according to RFC 2254, with the exception that it may contain variables of the form {i} where i is an integer. The variable {i} specifies the i’th element from the array filterArgs, which is to be substituted for the string '{i}'. It is provider-specific whether or not the substitution requires that the element’s string representation be used. Interpretation of filterExpr is otherwise identical to how filter is interpreted in the search(Name, String, SearchControls) form of search.

The createIterator(String, NdsIteratorControls, String, String) method lists subordinates in the named context or named object for entries that satisfy the given filter of class and subordinate names, then returns the objects without attribute names or values in an NdsIterator object.

The items stored in NdsIteratorControls that apply to the list iterator with their default values are:

- scalability = NdsIteratorControls.DS_ITR_PREFER_SCALABLE
- timeout = 0
- retObj = false

The name for the classNames filter is the name of an object class such as User, Computer, or Server. The value given for the subordinateName filter can be one of the following:

- The left-most name of an object, such as Adam or Graphics Printer
- A string with asterisks (*), such as A* or Gr*
- NULL, which means any name is valid.

The location of the subordinate object(s) in the NDS tree is immediately subordinate to the object specified by objectName. The relationship between className and subordinateName is an ’AND’ relationship. When className and subordinateName are provided, a list of immediate subordinate objects restricted by both filters is returned. When className is NULL and subordinateName is NULL, a list of all immediate subordinates is returned. When className is provided and subordinateName is NULL, a list of immediate subordinates restricted only by the className filter is returned. When className is NULL and subordinateName is provided, a list of immediate subordinates restricted only by the subordinateName filter is returned.

The following examples show how to use wildcards for untyped names:

- c* - Any object whose left-most name begins with a 'c' character.
- M*y Any object beginning with 'M' and ending with'y' such as Mary.

If the wildcard name specified for subordinateName includes a type, such as 'CN'; the name must include the equals (=) sign. The following examples show how to use wildcards for typed names:

- cn=* - Any object whose left-most name is a common name.
- cn=c* - Any object whose left-most name is a common name and begins with 'c'.
- o*=* - Any object whose left-most name has a naming attribute beginning with an 'o', such as O or OU.
- o*=c* Any object whose left-most name has a naming attribute beginning with an 'o', and whose name begins with 'c'.

2.5.8 NdsIterator Interface

The NdsIterator interface creates a large virtual list of objects on the particular server in question containing all the entries that pass the search filter. An NdsIterator has a current position identified by a number between 0 and 1001 in the large virtual list. This number represents a relative position. Zero always indicates that the current position is at the top of the list, and 1000 indicates the bottom. A number of 500 means the position is near the middle of the list, and the number 1001 (symbol DS_ITR_EOF) indicates the position is at the end of the list just after the last entry in the list or if no entries are present.

The position numbers are used in the setPosition(int position, int timeout) method to set the current iterator position in the list of search results objects. The timeout parameter sets the time in seconds allowed before returning. For no time limit set timeout to zero.

Another version of the setPosition() method passes in another NdsIterator that is used to set the current iterator position. It is not necessary that the two iterators be identical. The system tries to find the closest match in the source iterator and positions the destination iterator accordingly.

A third setPosition() method implements the 'type down' functionality by passing in an attribute and a string value to use for type down. For example, if you have a list sorted by surname you can specify 'D' and it will position to the first name starting with D. If you specify 'DA' it positions to the first name starting with DA, and so forth. If there are no entries matching the value string, it positions to the first one that is greater than the specified value or EOF.

Other methods to assist in navigating up and down the list of search result entries include skip(), hasNext(), next(), hasPrevious(), previous(), current(), and count(). See the NdsIterator interface (../api/com/novell/service/nds/NdsIterator.html) in the Reference Guide for more detailed information on each of these methods.

2.5.9 NdsIteratorInfo Interface

The NdsIteratorInfo interface encapsulates information about an NdsIterator. An NdsIteratorInfo object is returned when the NdsIterator.getInfo() method is called. It is also used as a parameter in NdsIteratorResult constructors. NdsIteratorInfo defines two methods that are used to provide positionable and scalable information about an Iterator. See the NdsIteratorInfo interface (../api/com/novell/service/nds/NdsIteratorInfo.html) in the Reference Guide for detailed information on these methods.

2.5.10 NdsIteratorControls Class

The NdsIteratorControls class encapsulates factors that determine the scope of iterators and what gets returned as a result of the iterator. The sort index is an array of strings with the first string the primary index, the second string the secondary index, and so forth. The default sort indexes are Baseclass and Name. An NdsIteratorControls instance is not synchronized against concurrent multithreaded access. Multiple threads trying to access and modify a single NdsIteratorControls instance should synchronize access to the object. The serialized form of an NdsIteratorControls object consists of the sort indexes (an array of strings) and the serialized form of SearchControls.

One NdsIteratorControls constructor uses the default sort index 'CN' while the other constructor creates an NdsIteratorControls object using the following arguments:

- scope - The search scope (DirContext.OBJECT_SCOPE, DirContext.ONELEVEL_SCOPE, or DirContext.SUBTREE_SCOPE).
- maxCount - The maximum number of entries to return. If 0, return all entries that satisfy the filter.
- timeout - The number of seconds to wait before returning. If 0, wait indefinitely.
- attrs - The identifiers of the attributes to return along with the entry. If NULL, return all attributes; and if empty, return no attributes.
- retobj - If TRUE, return the object bound to the name of the entry; if FALSE, do not return the object.
- deref - If TRUE, dereference links during search.
- indexSelect - An optional string selecting the index to be used in creating the iterator. If NULL or empty, the index is selected by the server based on the search filter. This argument applies only to SKADS iterators.
- sortKey - List of attributes specifying the sort keys to use when sorting the objects. Up to three attributes may be specified. This argument applies only to non-SKADS iterators.
- scalability - Determines whether the NdsIterator must connect to a SKADS server, tries to connect to a SKADS server first, or must connect to a non-SKADS server. The scalability argument must be one of the following values:

  DS_ITR_REQUIRE_SCALABLE (return an error if unable to connect to a SKADS server)

  DS_ITR_PREFER_SCALABLE (first tries to connect to a SKADS server but if unable to do so, the iterator functionality will be emulated by the client libraries)

  DS_ITR_FORCE_EMULATION (forces emulation mode even if it can connect to a SKADS server)

The six NDSIteratorControls methods allow manipulation of the sort indexes. See the NdsIteratorControls class (../api/com/novell/service/nds/NdsIteratorControls.html) in the Reference Guide for more detailed information on these methods.

2.5.11 NdsIteratorResult Class

An enumeration of NdsIteratorResult objects is returned from several of the NdsIterator methods. Each NdsIteratorResult object contains the name of the object and other information about the object. The name is either relative to the target context of the search (which is named by the name parameter), or it is a URL string. If the target context is included in the enumeration (as is possible when the cons parameter specifies a search scope of SearchControls.OBJECT_SCOPE or

SearchControls.SUBSTREE_SCOPE), its name is the empty string. See the NdsIteratorResult class (../api/com/novell/service/nds/NdsIteratorResult.html) in the Reference Guide for more detailed information on the constructors and methods.

2.6 NetWare File System Provider

2.6.1 Introduction

The NetWare File System Provider for JNDI provides contexts that represent objects in the file system, such as NetWare volumes, directories and files. It also describes how to access file streams, create and delete files, and access attributes of the file system.

2.6.2 File System Provider Components

The components (interfaces and classes) of the NetWare File System are contained in two packages:

- Package com.novell.java.io (../api/com/novell/java/io/package-summary.html) provides functionality for file I/O, including file streams access.
- Package com.novell.java.file.nw (../api/com/novell/service/file/nw/package-summary.html) provides access to all other file system functionality.

The com.novell.java.io streams components are extended in the com.novell.service.file.nw package to provide NetWare specific extensions of the various streams interfaces.

The most important interfaces and classes in these packages are described below. All components of each package may be accessed by going to the links provided above.

- com.novell.service.file.nw.NameSpace (../api/com/novell/service/file/nw/NameSpace.html) class provides an interface for getting the default name space for the platform on which the JVM is executing. It also provides a central interface for changing NetWare file system name space integer values into strings and vice versa.
- com.novell.java.io.NFile (../api/com/novell/java/io/NFile.html) interface has been defined in terms of the java.io.File interface. Instances of the NFile class represent the name of a file or directory on the host file system. This class is intended to provide an abstraction that deals with most of the machine dependent complexities of files and path names in a machine-independent fashion.
- com.novell.service.file.nw.NetwareFile (../api/com/novell/service/file/nw/NetwareFile.html) interface is used for getting the NetWare file system file object attribute values, which include Directory Entry Information, Extended Attributes, Effective Rights, and Trustees.
- com.novell.service.file.nw.DirectoryEntryInformation (../api/com/novell/service/file/nw/DirectoryEntryInformation.html) class results in a mutable object, and provides the NetWareFile interface with 26 attribute values of Directory Entry Information. A directory entry can be a file or a directory.

- com.novell.service.file.nw.EffectiveRights (../api/com/novell/service/file/nw/EffectiveRights.html) class provides an attribute interface for effective rights of a file or directory. It contains information about the effective rights for a users name for which to return effective rights and the rights for the currently logged or requested user.
- com.novell.service.file.nw.ExtendedAttribute (../api/com/novell/service/file/nw/ExtendedAttribute.html) class provides constructors and methods for the support of an extended attribute, which is a name and a corresponding byte array of data associated with a given file. The given file can have multiple extended attributes, which can be enumerated using the EAEnumerator interface.
- com.novell.service.file.nw.EAEnumerator (../api/com/novell/service/file/nw/EAEnumerator.html) interface provides for enumeration of extended attributes of a given file, which can have multiple extended attributes.
- com.novell.service.file.nw.Trustee (../api/com/novell/service/file/nw/Trustee.html) class provides variables, constructors and methods for the support of a trustee, which is composed of a name identifying the trustee, the rights associated with the named trustee and a compare string. Trustees can be associated with a given directory entry (file or directory), which can have multiple trustees that may be enumerated using the TrusteeEnumerator interface.
- com.novell.service.file.nw.TrusteeEnumerator (../api/com/novell/service/file/nw/TrusteeEnumerator.html) interface provides for enumeration of trustees of a given directory entry (file or directory), which can have multiple trustees.
- com.novell.service.file.nw.TrusteePathEnumerator (../api/com/novell/service/file/nw/TrusteePathEnumerator.html) interface provides an attribute for enumeration of all paths on a volume to which a user is assigned as a trustee. This interface contains the user name for which to return trustee paths, and the paths for the currently logged or requested user.
- com.novell.service.file.nw.NetwareDirectory (../api/com/novell/service/file/nw/NetwareDirectory.html) interface is used for getting the NetWare file system directory object attribute values, which include Directory Space Information.
- com.novell.service.file.nw.DirectorySpaceInformation (../api/com/novell/service/file/nw/DirectorySpaceInformation.html) class provides support for NetWare file system directory attribute values and operations.
- com.novell.service.file.nw.NetwareVolume (../api/com/novell/service/file/nw/NetwareVolume.html) interface is used for getting the NetWare file system volume object attribute values, which include Volume Information, Volume Restriction, Volume Utilization, and Trustee Path Enumeration.
- com.novell.service.file.nw.VolumeInformation (../api/com/novell/service/file/nw/VolumeInformation.html) class provides variables, constructors and methods for accessing volume information attribute values.
- com.novell.service.file.nw.VolumeRestriction (../api/com/novell/service/file/nw/VolumeRestriction.html) class provides variables, constructors and methods for accessing volume restriction attribute values associated with a given volume, which can have multiple restrictions.
- com.novell.service.file.nw.VolumeRestrictionEnumerator (../api/com/novell/service/file/nw/VolumeRestrictionEnumerator.html) interface provides for obtaining an enumeration of volume restriction attributes of a volume, which can have multiple restrictions.
- com.novell.service.file.nw.VolumeUtilization (../api/com/novell/service/file/nw/VolumeUtilization.html) class provides variables, constructors and methods for accessing the volume utilization attribute value. This attribute value is unique in that it requires the

objectName to be set by the user before it can obtain the values. Because of this the attribute value cannot be setup for searching; only the default constructor is available.

- com.novell.java.io.DataAccessable (../api/com/novell/java/io/DataAccessable.html) interface enables you to read and write streams with JNDI provider contexts by identifying an object that may support input/output streams and random access to data. A typical usage of an object of this type is to pass it to the various constructors of the NInputStream, NOutputStream and RandomAccess classes (or provider specific extensions of these classes). These constructors all take a DataAccessable object as a parameter and then implement themselves in terms of this input parameter.
- com.novell.service.file.nw.DataAccessableParameters (../api/com/novell/service/file/nw/DataAccessableParameters.html) interface provides the file systems implementation of the custom data parameter passed as the custom parameter to the various NInputStream, NOutputStream and RandomAccess constructors. This class defines constants that can be used for the various constructors in these classes, or this object can be instantiated and passed into the custom parameter constructors. The custom parameters associated with file streams are: openFlags and dataSelector.
- com.novell.java.io.NInputStream (../api/com/novell/java/io/NInputStream.html) class provides input stream functionality to some data device object, which implements the DataAccessable interface. The various constructors of this class open an input stream on (or with) the DataAccessable object passed into the constructors.
- com.novell.service.file.nw.NFileInputStream (../api/com/novell/service/file/nw/NFileInputStream.html) class opens an input stream on a NetWare file or Extended Attribute. It extends the NInputStream class by adding convenience constructors for the various custom parameters. The DataAccessableParameters class can also be used by the user application layer to obtain constant flags for the various constructors or it can instantiate a DataAccessableParameters object, which it passes into the NInputStream constructor.
- com.novell.java.io.NOutputStream (../api/com/novell/java/io/NOutputStream.html) class provides output stream functionality to some data device object, which implements the DataAccessable interface. The various constructors of this class open an output stream on (or with) the DataAccessable object passed into the constructors.
- com.novell.service.file.nw.NFileOutputStream (../api/com/novell/service/file/nw/NFileOutputStream.html) class opens an output stream on a NetWare file or Extended Attribute. It extends the NOutputStream class by adding convenience constructors for the various custom parameters. The DataAccessableParameters class can also be used by the user application layer to obtain constant flags for the various constructors or it can instantiate a DataAccessableParameters object, which it passes into the NOutputStream constructor.
- com.novell.java.io.RandomAccess (../api/com/novell/java/io/RandomAccess.html) class provides random access functionality to some data devices, which implement the DataAccessable interface. The various constructors of this class open access on (or with) the DataAccessable object passed into the constructors.
- com.novell.service.file.nw.NRandomAccessFile (../api/com/novell/service/file/nw/NRandomAccessFile.html) class opens a random access on a NetWare file. It extends the RandomAccess class by adding convenience constructors for the various custom parameters. The DataAccessableParameters class can also be used by the user application layer to obtain constant flags for the various constructors or it can instantiate a DataAccessableParameters object, which it passes into the RandomAccess constructor.

2.6.3 File System Initial Context Implementations

In order to access the File System interfaces and classes, you must specify an initial context factory to use. Following is an example of how the File System initial context factory implementation might be done.

```
Hashtable properties = new Hashtable();

property.put(Context.INITIAL_CONTEXT_FACTORY,

   "com.novell.service.file.nw.FileSystemInitialContextFactory");

property.put(Context.PROVIDER_URL, <server name>);

DirContext initCtx = new InitialDirContext(properties);
```

To access the Server through federation you might do the following:

```
serverDirContext = lookup(<server name>+"/");
```

After the initial context has been implemented, you can then get the session environment object using code similar to the following:

```
InitialContext initCtx = new InitialContext (hash);

hash = initCtx.getEnvironment ();

Session sess = (Session) hash.get (Environment.SESSION.OBJECT);
```

For a detailed discussion of initial context implementation see "Initial Context Implementations" on page 40 in the Novell Services Introduction.

2.6.4 Important File System Provider Methods

Listed below are some of the most important File System Provider methods available. All methods are accessible by going to each of the file system components.

File System Methods (com.novell.service.file.nw Package)

- NameSpace.getPlatformDefaultNameSpace (../api/com/novell/service/file/nw/NameSpace.html#getPlatformDefaultNameSpace()) - Returns the default NetWare name space depending on the OS under which the current JVM is running.
- NetwareFile.getDirectoryEntryInformation (../api/com/novell/service/file/nw/NetwareFile.html#getDirectoryEntryInformation()) - Returns a DirectoryEntryInformation object associated with this directory entry (file or directory).
- NetwareFile.setDirectoryEntryInformation (../api/com/novell/service/file/nw/NetwareFile.html#setDirectoryEntryInformation(com.novell.service.file.nw.DirectoryEntryInformation, int)) - Modifies the back end DirectoryEntryInformation object associated with this directory entry (file or directory).
- NetwareFile.getEAEnumerator (../api/com/novell/service/file/nw/NetwareFile.html#getEAEnumerator()) - Returns the EAEnumerator object associated with this directory entry (file or directory).

- NetwareFile.getEffectiveRights (../api/com/novell/service/file/nw/NetwareFile.html#getEffectiveRights()) - Returns the EffectiveRights object associated with this directory entry (file or directory).
- NetwareFile.setExtendedAttribute (../api/com/novell/service/file/nw/NetwareFile.html#setExtendedAttribute(com.novell.service.file.nw.ExtendedAttribute, int)) - Modifies the back end ExtendedAttribute object associated with this directory entry (file or directory).
- NetwareFile.getTrusteeEnumerator (../api/com/novell/service/file/nw/NetwareFile.html#getTrusteeEnumerator()) - Returns the TrusteeEnumerator object associated with this directory entry (file or directory).
- NetwareFile.setTrustee (../api/com/novell/service/file/nw/NetwareFile.html#setTrustee(com.novell.service.file.nw.Trustee, int)) - Modifies the back end Trustee attribute object associated with this directory entry (file or directory).
- NetwareDirectory.getDirectorySpaceInformation (../api/com/novell/service/file/nw/NetwareDirectory.html#getDirectorySpaceInformation()) - Returns the DirectorySpaceInformation object associated with this directory entry (file or directory).
- NetwareDirectory.setDirectorySpaceInformation (../api/com/novell/service/file/nw/NetwareDirectory.html#setDirectorySpaceInformation(com.novell.service.file.nw.DirectorySpaceInformation, int)) - Modifies the back end DirectorySpaceInformation object associated with this directory entry (file or directory).
- NetwareVolume.getVolumeInformation (../api/com/novell/service/file/nw/NetwareVolume.html#getVolumeInformation()) - Returns the VolumeInformation object associated with this volume.
- NetwareVolume.getVolumeUtilization (../api/com/novell/service/file/nw/NetwareVolume.html#getVolumeUtilization()) - Returns the VolumeUtilization object associated with this volume.
- NetwareVolume.getVolumeRestrictionEnumerator (../api/com/novell/service/file/nw/NetwareVolume.html#getVolumeRestrictionEnumerator()) - Returns the VolumeRestrictionEnumerator object associated with this volume.
- NetwareVolume.getTrusteePathEnumerator (../api/com/novell/service/file/nw/NetwareVolume.html#getTrusteePathEnumerator()) - Returns the TrusteePathEnumerator object associated with this volume.
- NetwareVolume.setVolumeRestriction (../api/com/novell/service/file/nw/NetwareVolume.html#setVolumeRestriction(com.novell.service.file.nw.VolumeRestriction, int)) - Modifies the back end VolumeRestriction object associated with this volume.

File Stream Accessor Methods (com.novell.java.io Package)

- DataAccessable.supportsInputStream (../api/com/novell/java/io/DataAccessable.html#supportsInputStream()) - Reports if this accessor supports input stream.
- DataAccessable.supportsOutputStream (../api/com/novell/java/io/DataAccessable.html#supportsOutputStream()) - Reports if this accessor supports output stream.
- DataAccessable.supportsRandomAccess (../api/com/novell/java/io/DataAccessable.html#supportsRandomAccess()) - Reports if this accessor supports random access.

- DataAccessable.supportsSubordinateInputStream (../api/com/novell/java/io/DataAccessable.html#supportsSubordinateInputStream()) - Reports if this accessor supports subordinate input streams.
- DataAccessable.supportsSubordinateOutputStream (../api/com/novell/java/io/DataAccessable.html#supportsSubordinateOutputStream()) - Reports if this accessor supports subordinate input streams.
- DataAccessable.supportsSubordinateRandomAccess (../api/com/novell/java/io/DataAccessable.html#supportsSubordinateRandomAccess()) - Reports if this accessor supports subordinate random access.

2.6.5 Relationship of File System Interfaces and Classes

Because the com.novell.service.file.nw package components are extended from the com.novell.java.io streams components to provide NetWare specific extensions of the various streams interfaces, the relationships between the NetWare File System components is illustrated in two diagrams:

- "NetWare File System Components" on page 74
- "NetWare Stream Accessor Components" on page 76

NetWare File System Components

The relationship between the NetWare File System interfaces and classes is illustrated in the diagram below, followed by a brief description of the relationship each interface and class has with other interfaces and classes.

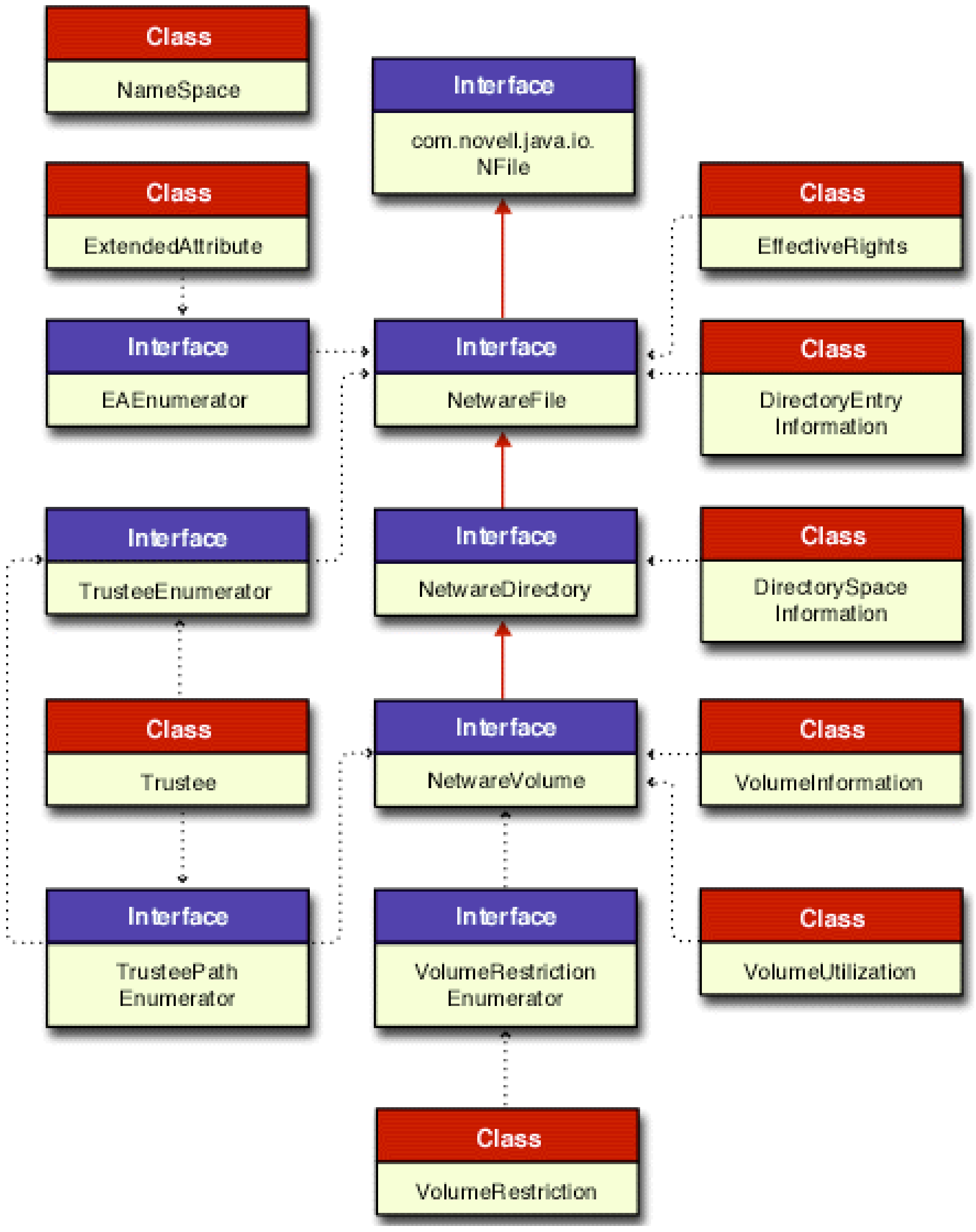


Figure 2-9 *NetWare File System Components*

In this diagram of interfaces and classes a solid red line ending with a solid arrow represents an extension of a class. The dotted black lines ending with a small solid arrow imply some relationships between the different interfaces and classes. Links are provided to the Reference Guide documentation for each of these important file system interfaces and classes.

- com.novell.service.file.nw.NameSpace (../api/com/novell/service/file/nw/NameSpace.html) class has no direct relationship with other interfaces and classes of the NetWare file system. Its main purpose is to get the default name space for the platform on which the JVM is executing.
- com.novell.java.io.NFile (../api/com/novell/java/io/NFile.html) interface is extended by the NetwareFile interface in the com.novell.service.file.nw package.
- com.novell.service.file.nw.NetwareFile (../api/com/novell/service/file/nw/NetwareFile.html) interface extends the com.novell.java.io.NFile interface. NetwareFile interface methods are available for obtaining and setting Directory Entry Information, Effective Rights, Extended Attributes, and Trustee attribute values associated with a file object.
- com.novell.service.file.nw.DirectoryEntryInformation (../api/com/novell/service/file/nw/DirectoryEntryInformation.html) class provides the NetWareFile interface with 26 attribute values of Directory Entry Information associated with a directory entry, which can be a file or a directory.
- com.novell.service.file.nw.EffectiveRights (../api/com/novell/service/file/nw/EffectiveRights.html) class provides the NetWareFile interface with the effective rights for a file or a directory.
- com.novell.service.file.nw.ExtendedAttribute (../api/com/novell/service/file/nw/ExtendedAttribute.html) class provides the NetWareFile interface with extended attribute information associated with a given file, which can have multiple extended attributes that are enumerated using the EAEnumerator interface.
- com.novell.service.file.nw.EAEnumerator (../api/com/novell/service/file/nw/EAEnumerator.html) interface enumerates the extended attributes of a given file, which can have multiple extended attributes.
- com.novell.service.file.nw.Trustee (../api/com/novell/service/file/nw/Trustee.html) class provides the NetWareFile interface with information identifying a trustee, the rights associated with the named trustee, and a compare string for a given directory entry (file or directory), which can have multiple trustees that are enumerated using the TrusteeEnumerator interface.
- com.novell.service.file.nw.TrusteeEnumerator (../api/com/novell/service/file/nw/TrusteeEnumerator.html) interface enumerates the trustees of a given directory entry (file or directory), which can have multiple trustees.
- com.novell.service.file.nw.TrusteePathEnumerator (../api/com/novell/service/file/nw/TrusteePathEnumerator.html) interface provides information on all paths of a volume to which a user is assigned as a trustee.
- com.novell.service.file.nw.NetwareDirectory (../api/com/novell/service/file/nw/NetwareDirectory.html) interface extends the NetwareFile interface and provides methods for obtaining and setting directory object attribute values, which include directory space information.
- com.novell.service.file.nw.DirectorySpaceInformation (../api/com/novell/service/file/nw/DirectorySpaceInformation.html) class provides the NetwareDirectory interface with directory object attribute values.
- com.novell.service.file.nw.NetwareVolume (../api/com/novell/service/file/nw/NetwareVolume.html) interface extends the NetwareDirectory interface. Methods are provided for getting the NetWare file system volume object attribute values, which include volume

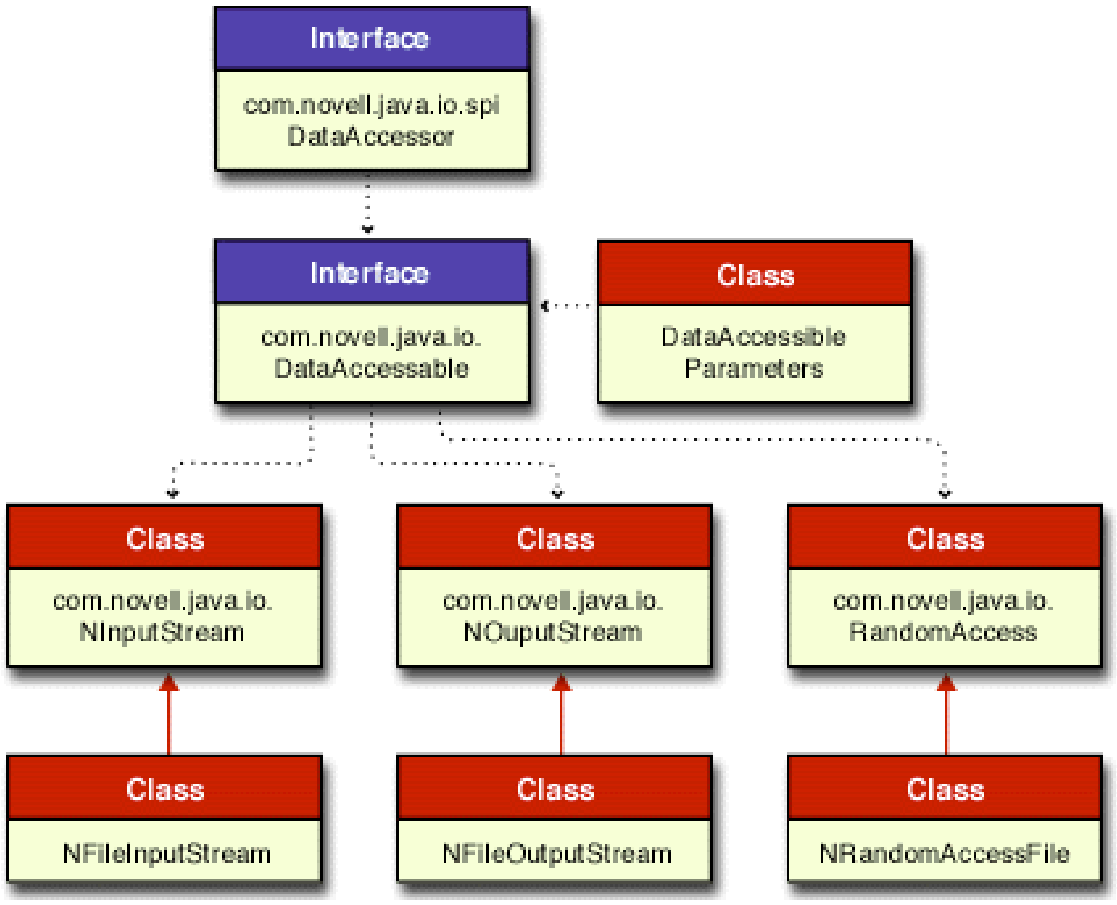
information, volume restriction information, volume utilization information, and trustee path enumeration information.

- com.novell.service.file.nw.VolumeInformation (../api/com/novell/service/file/nw/VolumeInformation.html) class provides the NetwareVolume interface with volume information attribute values associated with a given volume.
- com.novell.service.file.nw.VolumeRestriction (../api/com/novell/service/file/nw/VolumeRestriction.html) class provides the NetwareVolume interface with volume restriction attribute values associated with a given volume, which can have multiple restrictions.
- com.novell.service.file.nw.VolumeRestrictionEnumerator (../api/com/novell/service/file/nw/VolumeRestrictionEnumerator.html) interface enumerates volume restriction attributes of a volume, which can have multiple restrictions.
- com.novell.service.file.nw.VolumeUtilization (../api/com/novell/service/file/nw/VolumeUtilization.html) class provides the NetwareVolume interface with volume utilization attribute values.

NetWare Stream Accessor Components

The relationship between the NetWare stream accessor interfaces and classes is illustrated in the diagram below, followed by a brief description of the relationship each interface and class has with other interfaces and classes.

Figure 2-10 *NetWare Stream Accessor Components*


Interface
com.novell.java.io.spi DataAccessor
Interface
com.novell.java.io. DataAccessable
Class
DataAccessible Parameters
Class
com.novell.java.io. NInputStream
Class
com.novell.java.io. NOuputStream
Class
com.novell.java.io. RandomAccess
Class
NFileInputStream
Class
NFileOutputStream
Class
NRandomAccessFile

In this diagram of interfaces and classes a solid red line ending with a solid arrow represents an extension of a class. The dotted black lines ending with a small solid arrow imply some relationships between the different interfaces and classes. Links are provided to the Reference Guide documentation for each of these important file stream interfaces and classes.

- com.novell.java.io.spi.DataAccessor (../api/com/novell/java/io/spi/DataAccessor.html) interface provides an interface that allows read and/or write access to a given resource. DataAccessor is not used by the user application layer, but is used by Stream service provider implementors.
- com.novell.java.io.DataAccessable (../api/com/novell/java/io/DataAccessable.html) interface provides a DataAccessable object to the various constructors of the NInputStream, NOutputStream and RandomAccess classes (or provider specific extensions of these classes). These constructors all take a DataAccessable object as a parameter and then implement themselves in terms of this input parameter.
- com.novell.service.file.nw.DataAccessableParameters (../api/com/novell/service/file/nw/DataAccessableParameters.html) interface passes the custom data parameter through DataAccessable objects to the various NInputStream, NOutputStream and RandomAccess constructors. The custom parameters associated with file streams are: openFlags and dataSelector.
- com.novell.java.io.NInputStream (../api/com/novell/java/io/NInputStream.html) class is extended by the com.novell.service.file.nw.NFileInputStream class. The various constructors of the NInputStream class open an input stream on (or with) the DataAccessable object passed into the constructors.
- com.novell.service.file.nw.NFileInputStream (../api/com/novell/service/file/nw/NFileInputStream.html) class extends the com.novell.java.io.NInputStream class by adding convenience constructors for the various custom parameters.
- com.novell.java.io.NOutputStream (../api/com/novell/java/io/NOutputStream.html) class is extended by the com.novell.service.file.nw.NFileOutputStream class. The various constructors of the NOutputStream class open an output stream on (or with) the DataAccessable object passed into the constructors.
- com.novell.service.file.nw.NFileOutputStream (../api/com/novell/service/file/nw/NFileOutputStream.html) class extends the com.novell.java.io.NOutputStream class by adding convenience constructors for the various custom parameters.
- com.novell.java.io.RandomAccess (../api/com/novell/java/io/RandomAccess.html) class is extended by the com.novell.service.file.nw.NRandomAccessFile class. The various constructors of the RandomAccess class open a random access file on (or with) the DataAccessable object passed into the constructors.
- com.novell.service.file.nw.NRandomAccessFile (../api/com/novell/service/file/nw/NRandomAccessFile.html) class extends the com.novell.java.io.RandomAccess class by adding convenience constructors for the various custom parameters.

2.7 NetWare Provider

- "Introduction" on page 78
- "NetWare Provider Components" on page 78
- "NetWare Initial Context Implementations" on page 78
- "Important NetWare Provider Methods" on page 79

* “Relationship of NetWare Classes and Interfaces” on page 80

2.7.1 Introduction

The NetWare provider for JNDI acts as a logical root name space for the Novell providers for JNDI. This Novell service provides contexts that list all the known NetWare servers and trees. The individual servers and trees available to the user are returned by performing lookup operations on specific server and tree names.

The NetWare name space uses SLP to list servers and trees. If the SLP configuration is incorrect, the returned list will be incomplete. If an incomplete list is suspected, check the SLP configuration.

This NetWare provider is the most general among Novell’s JNDI providers. Specifying com.novell.service.nw.NetWareInitialContextFactory (../api/com/novell/service/nw/NetWareInitialContextFactory.html) as the initial context factory is the only required setting to start browsing the NetWare name space. From the NetWare name space, a JNDI application can navigate to all other Novell naming systems. Optionally, a user can specify a tree or server name when creating an initial context, and the provider will return a corresponding NDS or Server context.

The NetWare naming system is used solely as a top level context to list the known NetWare trees and servers. It has no attributes or binding capabilities. Therefore its contexts are not subclasses of javax.naming.DirContext, and it contains no schema information.

2.7.2 NetWare Provider Components

The important components (interfaces and classes) of the NetWare provider are the following:

* NetWareInitialContextFactory (../api/com/novell/service/nw/NetWareInitialContextFactory.html) class creates different initial contexts, depending on the PROVIDER_URL entry.
* NetWareInitialContext (../api/com/novell/service/nw/NetWareInitialContext.html) class provides the root of the name space. It has two bindings to subcontexts with names "Trees" and "Servers".
* NetWareServersContext (../api/com/novell/service/nw/NetWareServersContext.html) class provides bindings to all known NetWare servers.
* NetWareTreesContext (../api/com/novell/service/nw/NetWareTreesContext.html) class provides bindings to all known NetWare trees.

2.7.3 NetWare Initial Context Implementations

In order to access the NetWare interfaces and classes, you must specify an initial context factory to use. Following is an example of how the NetWare initial context factory implementation might be done.

```
Hashtable properties = new Hashtable();

property.put(Context.INITIAL_CONTEXT_FACTORY,

   "com.novell.service.bindery.NetWareInitialContextFactory");

property.put(Context.PROVIDER_URL, <server name>);
```

```
DirContext initCtx = new InitialDirContext(properties);
```

To access the Server through federation you might do the following:

```
serverDirContext = lookup(<server name>+"/");
```

After the initial context has been implemented, you can then get the session environment object using code similar to the following:

```
InitialContext initCtx = new InitialContext (hash);

hash = initCtx.getEnvironment ();

Session sess = (Session) hash.get (Environment.SESSION.OBJECT);
```

For a detailed discussion of initial context implementation see "Initial Context Implementations" on page 40 in the Novell Services Introduction.

2.7.4 Important NetWare Provider Methods

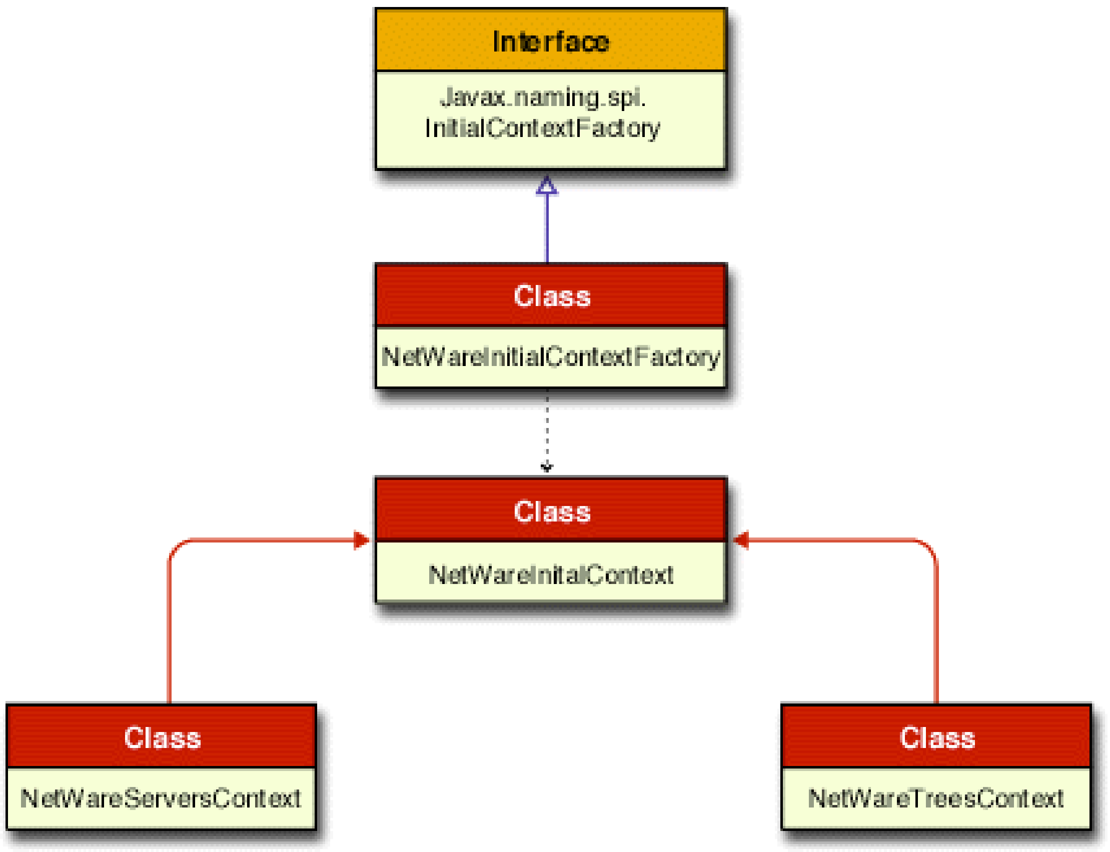
Listed below is the only NetWare Provider method available:

- NetWareInitialContextFactory.getInitialContext (../api/com/novell/service/nw/NetWareInitialContextFactory.html#getInitialContext(java.util.Hashtable)) - Creates an initial context based on the PROVIDER_URL entry in the info parameter. The syntax for the URL is [[NetWare://][<SapHost>]/"Trees | Servers"/<DomainName>]. If the URL is not set or is empty, an NetWareInitialContext will be returned.

2.7.5 Relationship of NetWare Classes and Interfaces

The relationship between the NetWare interfaces and classes is illustrated in the diagram below, followed by a brief description of the relationship each interface and class has with other interfaces and classes.

Figure 2-11 *NetWare Interfaces and Classes*



In this diagram of interfaces and classes a solid blue line ending with a hollow arrow represents an implementation of an interface. A solid red line ending with a solid arrow represent an extension of a class. The dotted black lines ending with a small solid arrow imply some relationships between the different interfaces and classes. Links are provided to the Reference Guide documentation for each of these important NetWare interfaces and classes.

- NetWareInitialContextFactory (../api/com/novell/service/nw/NetWareInitialContextFactory.html) class implements the Javasoft InitialContextFactory interface, and creates different NetWareInitialContext objects depending on the Context.PROVIDER_URL entry (Servers or Trees).
- NetWareInitialContext (../api/com/novell/service/nw/NetWareInitialContext.html) class is created by the NetWareInitialContextFactory class, and is extended by the NetWareServersContext and NetWareTreesContext classes.
- NetWareServersContext (../api/com/novell/service/nw/NetWareServersContext.html) class extends the NetWareInitialContext class.

- NetWareTreesContext (../api/com/novell/service/nw/NetWareTreesContext.html) class extends the NetWareInitialContext class.

2.8 Queue Management System (QMS) Provider

2.8.1 Introduction

The Queue Management System provider for JNDI allows the creation, submission, and servicing of queue jobs. The QMS service provides contexts that implement the behavior of NetWare queues and queue jobs through a central storage and queueing mechanism that enables users to create jobs that are added to the queue. The QMSQueue interface represents the central storage and queuing mechanism that is responsible for managing jobs placed in it. The jobs can be serviced remotely by an application at another node on the network. A queue job is an individual entry in the QMS queue. You can list, create, delete, and modify jobs in NetWare queues using JNDI methods.

Queues are the simplest and most direct of all the methods that developers can use to distribute processes on the network. Although not a suitable solution for all situations, QMS offers several advantages over other methods.

- Queues are managed through existing network tools and are suited to applications that deal with large workloads and that need the flexibility and control inherit in the queuing process.
- Security and access for queues are controlled by the network operating system. An application can ensure that only qualified users work with a job's related data.
- QMS provides great flexibility. Although every queue must adhere to QMS's standard structure, QMS allows applications to define their own specifications within the QMS format. Thus, queues can transmit information that is specific to a specialized service.
- The queue storage is centralized and accessible to a variety of clients.

The QMSQueue interface contains QMSJobs, which are created through the QMSQueue.createJob() method. Once the job is created, there are several job controls that can be set. When these controls have been set, a job is submitted to the queue through the QMSJob.submit() method, which returns an output stream used for storing job data. When the output stream is closed using the close() method, the job is ready for servicing.

A QMSJob can represent almost any application task, depending on how the developer has defined the job service protocol. One defined job service protocol is the Printing protocol, in which print jobs are submitted to a queue that is then serviced by a print server. The data to be printed is placed in the output stream provided when the print job is submitted to the queue.

Other job service protocols can be defined for handling any number of job types, including compile servers, archive servers, and so forth. The queue type and job type are useful in making sense of all the possible protocols. When a job is submitted to a queue, it can be targeted for a specific server, targeted to run after a specific time, given a useful description, and so forth.

Applications that service queue jobs are called job servers, which are required to register with (attach to) the queue before they can begin servicing jobs from the queue. Depending on the security requirements established for the queue, the job server will also be required to authenticate to the network before it can actually begin servicing jobs. This registry process helps guarantee that only authorized job servers are allowed to handle queue jobs.

The client that submits a job to a queue grants certain rights to the job server. For this reason, a queue maintains a list of authorized queue servers, users, and operators. In order for a job server to service jobs from a given queue, it must be in the queue server list, or have security equivalence to an object in the queue server list. The same applies for queue users and operators (managers).

2.8.2 QMS Components

The important components (interfaces and classes) of QMS are the following:

- QMSQueue (../api/com/novell/service/qms/QMSQueue.html) interface provides the factory for and the central container (queue) responsible for managing QMSJobs placed in it.
- QMSJob (../api/com/novell/service/qms/QMSJob.html) interface represents a job (almost any application task) in the queue. The job service protocol must be defined by the developer.
- QueueDirContext (../api/com/novell/service/qms/naming/QueueDirContext.html) class is an implementation class that represents a queue. It implements both the JNDI DirContext interface (via the AtomicDirContext) as well as the QMSQueue interface.
- QueueJobDirContext (../api/com/novell/service/qms/naming/QueueJobDirContext.html) class is an implementation class that represents queue jobs. It implements both the JNDI DirContext interface (via the AtomicDirContext) as well as the QMSJob interface.
- QMSOutputStream (../api/com/novell/service/qms/QMSOutputStream.html) class provides the standard output stream functionality to NetWare queue jobs.
- QMSEnvironment (../api/com/novell/service/qms/naming/QMSEnvironment.html) class defines the set of constant Strings that uniquely controls the behavior of queues and queue jobs. It also contains functions for managing the queues and queue jobs.

2.8.3 QMS Initial Context Implementation

In order to access the QMS interfaces and classes you must construct an initial context factory implementation for starting up the queue. Following is an example of how the QMS initial context factory implementation might be constructed.

```
Hashtable properties = new Hashtable();

property.put(Context.INITIAL_CONTEXT_FACTORY,

   "com.novell.service.qms.naming.qmsInitialContextFactory");

property.put(Context.PROVIDER_URL, <server name>);

DirContext initCtx = new InitialDirContext(properties);
```

After the initial context has been implemented, you can then get the session environment object using similar code to the following:

```
InitialContext initCtx = new InitialContext (hash);
```

```
hash = initCtx.getEnvironment ();

Session sess = (Session) hash.get (Environment.SESSION.OBJECT);
```

For a detailed discussion of initial context implementation, see "Initial Context Implementations" on page 40 in the Novell Services Introduction.

2.8.4 Important QMS Methods

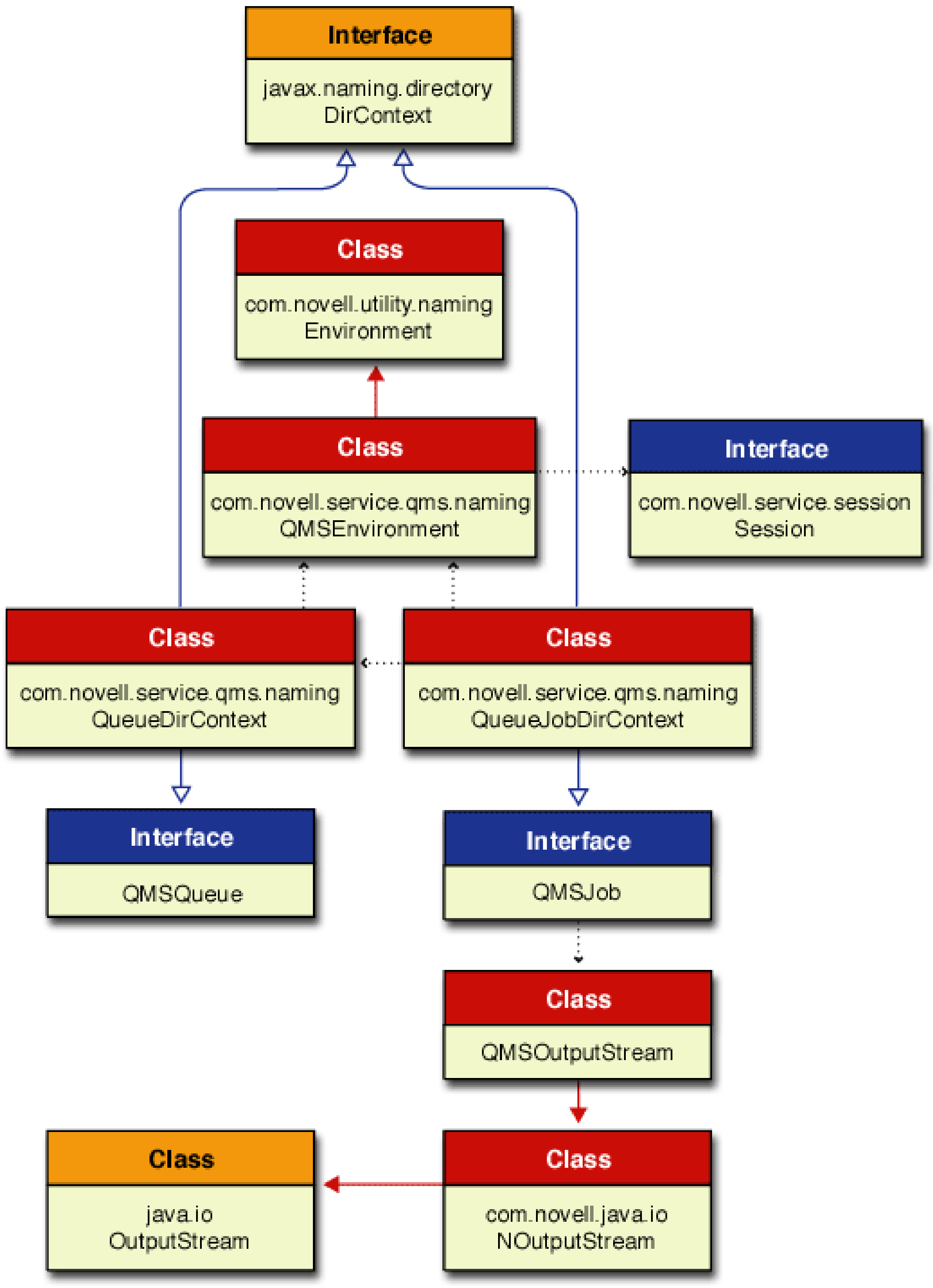
Listed below are some important QMS methods of which to be aware. For a complete list of available methods, go to the Reference Guide (../api/packages.html) documentation.

- QMSQueue.createJob (../api/com/novell/service/qms/QMSQueue.html#createJob()) - Creates a new job (QMSJob) for this queue.
- QMSQueue.createJob (../api/com/novell/service/qms/QMSQueue.html#createJob(int)) - Creates and returns a QMSJob object for an existing job specified by its job ID.
- QMSQueue.listJobs (../api/com/novell/service/qms/QMSQueue.html#listJobs()) - Returns an enumeration of jobs in this queue.
- QMSQueue.setState (../api/com/novell/service/qms/QMSQueue.html#setState(int)) - Sets the state flag of the queue given the new settings (QS_CANT_ADD_JOBS, QS_SERVERS_CANT_ATTACH, or QS_CANT_SERVICE_JOBS).
- QMSQueue.getState (../api/com/novell/service/qms/QMSQueue.html#getState()) - Returns the state of the queue as a bit mask. (See state flag settings above for QMSQueue.setState).
- QMSJob.submit (../api/com/novell/service/qms/QMSJob.html#submit()) - Submits a job into the queue and gets a QMSOutputStream back on which to write data.
- QMSJob.cancel (../api/com/novell/service/qms/QMSJob.html#cancel()) - Cancels the job, removes it from the queue, and aborts the output stream if it is open.
- QMSJob.setPosition (../api/com/novell/service/qms/QMSJob.html#setPosition(int))- Sets/adjusts the position of the job within the queue as specified by the newPosition parameter.
- QMSJob.getPosition (../api/com/novell/service/qms/QMSJob.html#getPosition()) - Returns the position of the job in the queue.
- QMSJob.setControlFlags (../api/com/novell/service/qms/QMSJob.html#setControlFlags(int)) - Sets the job control flags, which indicate the current status of the job. Available job control flags are: QF_AUTO_START, QF_ENTRY_OPEN, QF_ENTRY_RESTART, QF_USER_HOLD, QF_OPERATOR_HOLD.
- QMSJob.getControlFlags (../api/com/novell/service/qms/QMSJob.html#getControlFlags()) - Returns the job control flags for the job. See the job control flags above for QMSJob.setControlFlags.
- QMSJob.getFileName (../api/com/novell/service/qms/QMSJob.html#getFileName()) - Returns the queue job file name. This file is associated with the queue job when it is submitted.
- QMSOutputStream.abort (../api/com/novell/service/qms/QMSOutputStream.html#abort()) - Aborts the currently open output stream. This will cancel the job that has been created on the stream.

2.8.5 Relationship of QMS Classes and Interfaces

The relationship between the QMS interfaces and classes is illustrated in the following diagram.

Figure 2-12 *Queue Management Service Class Diagram*



In this diagram of interfaces and classes a solid blue line ending with a hollow arrow represents an implementation of an interface. A solid red line ending with a solid arrow represent an extension of a class. The dotted black lines ending with a small solid arrow imply some relationships between the different interfaces and classes. Links are provided to the Reference Guide documentation for each of these important Server interfaces and classes.

- QMSQueue (../api/com/novell/service/qms/QMSQueue.html) interface is implemented by the QueueDirContext class.
- QMSJob (../api/com/novell/service/qms/QMSJob.html) interface is implemented by the QueueJobDirContext class, and it writes data to the QMSOutputStream.
- DirContext (http://java.sun.com/products/jndi/1.2/javadoc/javax/naming/directory/DirContext.html) interface is implemented by QueueDirContext and QueueJobDirContext via the AtomicDirContext. It is the directory service interface containing methods for examining and updating attributes associated with objects, and methods for searching the directory.
- QueueDirContext (../api/com/novell/service/qms/naming/QueueDirContext.html) class implements both the JNDI DirContext interface (via the AtomicDirContext) and the QMSQueue interface.
- QueueJobDirContext (../api/com/novell/service/qms/naming/QueueJobDirContext.html) class implements both the JNDI DirContext interface (via the AtomicDirContext) and the QMSJob interface.
- QMSOutputStream (../api/com/novell/service/qms/QMSOutputStream.html) class extends the NOutputStream class and received data written to it by the QMSJob interface.
- QMSEnvironment (../api/com/novell/service/qms/naming/QMSEnvironment.html) class extends the Environment class and is associated with both the QueueDirContext and QueueJobDirContext classes.
- Environment (../api/com/novell/utility/naming/Environment.html) class provides constants for environment variables associated with Novell JNDI providers.
- Session (../api/com/novell/service/session/Session.html) class provides a communication channel between the object that creates a session and the target object and provides methods that are common to all sessions.

2.9 Server Provider

2.9.1 Introduction

The Server provider for JNDI allows for server management and control functionality. The Server Provider has a context that represents a NetWare server. This context allows access to attribute information and provides federation to three name spaces associated with the server: File System, NCPExtensions, and Bindery.

2.9.2 Server Provider Components

The important components (interfaces and classes) of the server provider are the following:

- NWServer (../api/com/novell/service/server/NWServer.html) interface represents a NetWare server, allowing you to perform functions on the NetWare server.
- ServerDirContext (../api/com/novell/service/server/ServerDirContext.html) Represents a NetWare server with the attriburtes enumerated below(ServerCounts, ServerDescription, ServerDiagnostics, ServerLoginStatus, and ServerVersion). The server has three bindings: File System, NCPExtensions, and Bindery.
- ServerCounts (../api/com/novell/service/server/ServerCounts.html) interface corresponds to a specific attribute of a ServerDirContext. The ID of the attribute is equal to ATTR_ID, and its value is an object that implements this interface.
- ServerCountsImpl (../api/com/novell/service/server/ServerCountsImpl.html) class provides the value of ServerCounts, a ServerDirContext attribute, and is an implementation class for the ServerCounts interface.
- ServerDescription (../api/com/novell/service/server/ServerDescription.html) interface corresponds to a specific attribute of a ServerDirContext. The ID of the attribute is equal to ATTR_ID, and its value is an object that implements this interface.
- ServerDescriptionImpl (../api/com/novell/service/server/ServerDescriptionImpl.html) class provides the value of ServerDescription, a ServerDirContext attribute, and is an implementation class for the ServerDescription interface.
- ServerDiagnostics (../api/com/novell/service/server/ServerDiagnostics.html) interface corresponds to a specific attribute of a ServerDirContext. The ID of the attribute is equal to ATTR_ID, and its value is an object that implements this interface.
- ServerDiagnosticsImpl (../api/com/novell/service/server/ServerDiagnosticsImpl.html) class provides the value of ServerDiagnostic, a ServerDirContext attribute, and is an implementation class for the ServerDiagnostic interface.
- ServerLoginStatus (../api/com/novell/service/server/ServerLoginStatus.html) interface corresponds to a specific attribute of a ServerDirContext. The ID of the attribute is equal to ATTR_ID, and its value is an object that implements this interface.
- ServerLoginStatusImpl (../api/com/novell/service/server/ServerLoginStatusImpl.html) class provides the value of ServerLoginStatus, a ServerDirContext attribute, and is an implementation class for the ServerLoginStatus interface.
- ServerVersions (../api/com/novell/service/server/ServerVersions.html) interface corresponds to a specific attribute of a ServerDirContext. The ID of the attribute is equal to ATTR_ID, and its value is an object that implements this interface.
- ServerVersionsImpl (../api/com/novell/service/server/ServerVersionsImpl.html) class provides the value of ServerVersions, a ServerDirContext attribute, and is an implementation class for the ServerVersions interface.

2.9.3 Server Initial Context Implementation

In order to access the Server interfaces and classes you must specify an initial context factory to use or you can access the Server through federation from another name space. Following is an example of how the Server initial context factory implementation might be done:

```
Hashtable properties = new Hashtable();

property.put(Context.INITIAL_CONTEXT_FACTORY,

   "com.novell.service.server.ServerInitialContextFactory");

property.put(Context.PROVIDER_URL, <server name>);

DirContext initCtx = new InitialDirContext(properties);
```

To access a server through federation you might do the following:

```
serverDirContext = lookup(<server name>+"/");
```

For a detailed discussion of initial context implementation see "Initial Context Implementations" on page 40 in the Novell Services Introduction.

2.9.4 Important Server Provider Methods

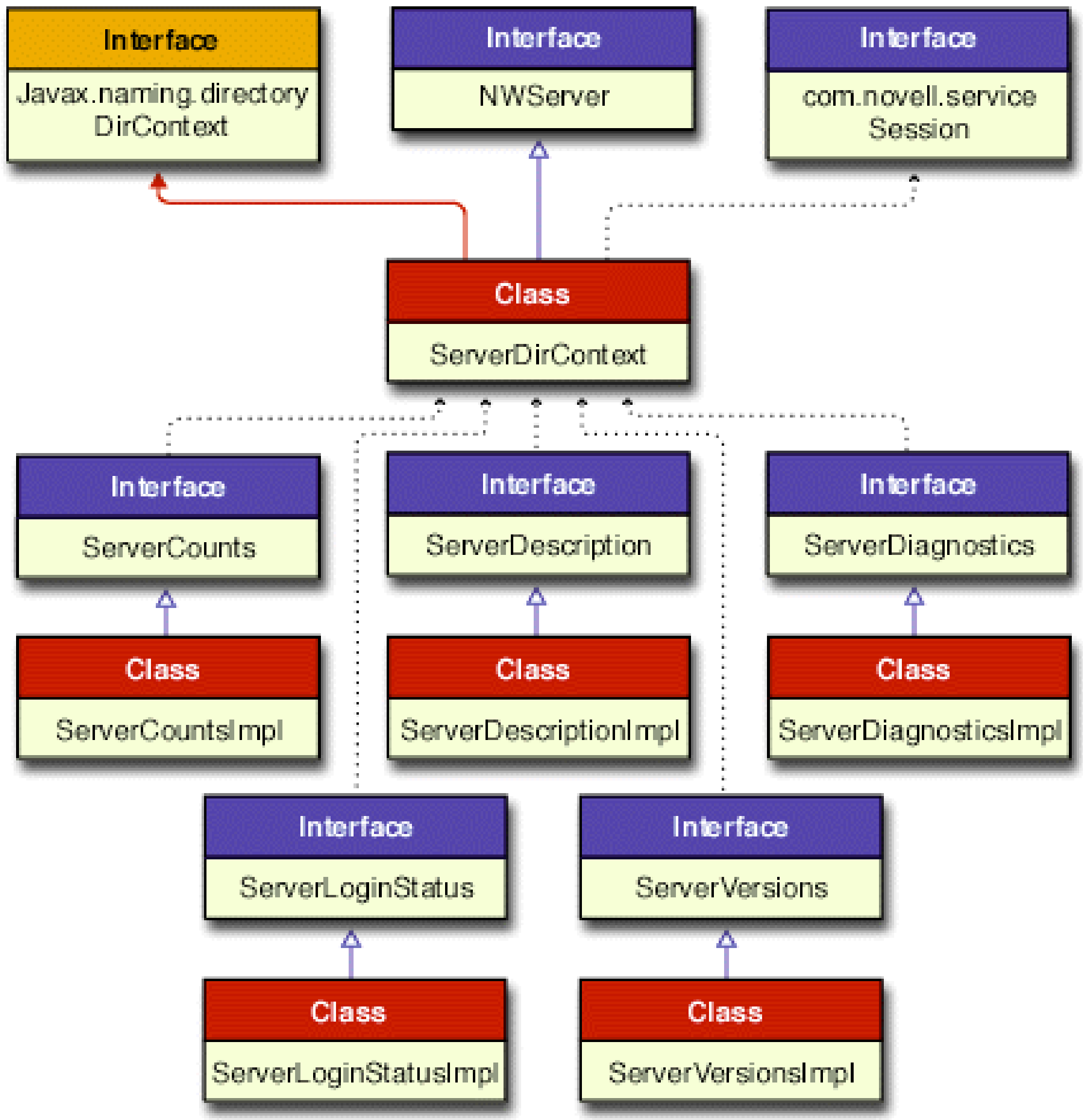
Listed below are some important Server provider methods of which to be aware. For a complete list of available methods, go to the Reference Guide (../api/overview-summary.html) documentation.

- NWServer.getTime (../api/com/novell/service/server/NWServer.html#getTime()) - Returns the time of the NetWare Server.
- NWServer.setTime (../api/com/novell/service/server/NWServer.html#setTime(java.util.Calendar)) - Sets the time of the NetWare Server.
- NWServer.loadNLM (../api/com/novell/service/server/NWServer.html#loadNLM(java.lang.String)) - Loads an NLM on the NetWare Server using the load command parameter for the NLM: {VOLUME NAME:}{PATH}NLMName{.ext}{parameters}.
- NWServer.unloadNLM (../api/com/novell/service/server/NWServer.html#unloadNLM(java.lang.String)) - Unloads the specified NLM from the NetWare Server.

2.9.5 Relationship of Server Classes and Interfaces

The relationship between the Server interfaces and classes is illustrated in the diagram below, followed by a brief description of the relationship each interface and class has with other interfaces and classes.

Figure 2-13 *Server Interfaces and Classes*



In this diagram of interfaces and classes a solid blue line ending with a hollow arrow represents an implementation of an interface. A solid red line ending with a solid arrow represent an extension of a class. The dotted black lines ending with a small solid arrow imply some relationships between the different interfaces and classes. Links are provided to the Reference Guide documentation for each of these important Server interfaces and classes.

- NWServer (../api/com/novell/service/server/NWServer.html) is implemented by the ServerDirContext class. NWServer represents a NetWare server on which various functions can be performed.
- ServerDirContext (../api/com/novell/service/server/ServerDirContext.html) class implements the NWServer interface and extends the Javasoft DirContext interface indirectly through the ComponenetDirContext class. ServerDirContext has bindings to three name spaces with binding names of File System, Bindery, and NCP Extensions.
- Session (../api/com/novell/service/session/Session.html) interface provides the current session for the ServerDirContext.
- ServerCounts (../api/com/novell/service/server/ServerCounts.html) interface is implemented by the ServerCountsImpl class, which provides the ServerCounts attribute value.
- ServerCountsImpl (../api/com/novell/service/server/ServerCountsImpl.html) class implements the ServerCounts interface providing it with the attribute value.
- ServerDescription (../api/com/novell/service/server/ServerDescription.html) interface is implemented by the ServerDescriptionImpl class, which provides the ServerDescription attribute value.
- ServerDescriptionImpl (../api/com/novell/service/server/ServerDescriptionImpl.html) class implements the ServerDescription interface providing it with the attribute value.
- ServerDiagnostics (../api/com/novell/service/server/ServerDiagnostics.html) interface is implemented by the ServerDiagnosticsImpl class, which provides the ServerDiagnostic attribute value.
- ServerDiagnosticsImpl (../api/com/novell/service/server/ServerDiagnosticsImpl.html) class implements the ServerDiagnostic interface providing it with the attribute value.
- ServerLoginStatus (../api/com/novell/service/server/ServerLoginStatus.html) interface is implemented by the ServerLoginStatusImpl class, which provides the ServerLoginStatus attribute value.
- ServerLoginStatusImpl (../api/com/novell/service/server/ServerLoginStatusImpl.html) class implements the ServerLoginStatus interface providing it with the attribute value.
- ServerVersions (../api/com/novell/service/server/ServerVersions.html) interface is implemented by the ServerVersionsImpl class, which provides the ServerVersionsI attribute value.
- ServerVersionsImpl (../api/com/novell/service/server/ServerVersionsImpl.html) class implements the ServerVersions interface providing it with the attribute value.

Documentation for the methods provided by each of the server attribute classes (ServerCounts, ServerDescription, ServerDiagnostics, ServerLoginStatus and ServerVersions) can be viewed by going to the Server Package (../api/com/novell/service/server/package-summary.html) of the Reference Guide and opening each of these attribute classes.

2.10 Session Manager Services

- "Introduction" on page 90
- "Multi User Session Manager" on page 90
- "Session Manager Components" on page 90

2.10.1 Introduction

The client connection model provides connection management. The Session Manager is a collection of packages that allows various session providers to be abstracted under one interface, using one manager to tie them together. The Session Manager uses a Session interface as a base class for most of its concepts. This helps keep the interfaces for working with sessions easy to learn and re-use. See also Section 2.11, "Remote Session Manager," on page 96 for information about the clientless version (../../../../njclc.htm) of NJCL.

Sessions are the building blocks of the API layer and are used to set up a communication channel between two entities. The first entity, the initiating entity, creates a session, and the second entity, the domain, can be anything from a tree to a server to any other object. Sessions, therefore, form the established communication channel between two objects in a domain. Once the object has established a singular session, it can communicate with another object in the same domain. Domains can be hierarchical, with child domains narrowing the scope of their parents.

Sessions are the client/server connections in NetWare and are part of a tree hierarchy where the Session Manager is the top-level session of the tree that ties all session providers together and is a session itself. Sessions are unique in relation to their parent and their domain name, which means two different sessions can have the same domain name, as long as their parents are different.

2.10.2 Multi User Session Manager

The Multi User Session Manager feature allows multiple separate sessions to the same domain. It also provides the capability for multiple distinct authentications sumultaneously into the same domain. For example, User1 and User2 can both be simultaneously authenticated to Tree1. The two sessions, Session1 and Session2, will then have User1 and User2 rights, respectively, when accessing services in Tree1. This feature only works when running on a NetWare server unless you are using the clientless version (../../../../njclc.htm) of NJCL. A source code sample, MultiUserSessionTest.java (../../../samplecode/njcl_sample/Session/MultiUserSessionTest.java.html), is provided as an example of how to use this functionality.

2.10.3 Session Manager Components

The Session Manager is broken down into four packages. The important components (interfaces and classes) of the Session Manager are contained in these packages.

- "Session Package" on page 91 contains components exposed to users of the Session Manager.
- "Session.Bindery Package" on page 92 contains the provider for bindery type sessions.
- "Session.NDS Package" on page 93 contains the provider for NDS type sessions
- "Session.Xplat Package" on page 93 contains the components used to synchronize both bindery and NDS type sessions that have common roots in NetWare.

Session Package

The com.novell.service.session (../api/com/novell/service/session/package-summary.html) package provides the interfaces, classes, and operations that allow users to manage sessions across multiple session providers. Four interfaces and seven classes in the package are provided for public access.

- Authenticatable (../api/com/novell/service/session/Authenticatable.html) interface provides an interface for all objects returned by the Session Manager that are authenticatable, meaning the identity of either or both of the communicating objects can be authenticated. Sessions, as authenticatable objects, can return object identities for use in the com.novell.java.security.Authenticator (../api/com/novell/java/security/Authenticator.html) class and its related packages as well as return the object's current authenticated state. For further information on session authentication, see Section 2.2, "Authentication Services," on page 41.
- Session (../api/com/novell/service/session/Session.html) interface provides a communication channel between the owning/initiating object that creates a session and the target domain (tree, server, or any other object). The Session interface describes only those methods that are common to all sessions. Functionality available only to a specific implementation of a session falls into two separate categories: read-only and read-write.

  Read-only functionality is available through session attributes, which can be retrieved via the getAttributes (../api/com/novell/service/session/Session.html#getAttributes()) methods. While attributes are designed to allow sessions to expose provider-specifics, some session attributes are common to all sessions (like those described in this class) for the sake of convenience. Session attributes are also used by the search (../api/com/novell/service/session/Session.html#search(com.novell.service.session.SessionAttrs)) method to facilitate finding sessions of a certain category.

  Read-write functionality is available through session services and the session environment. Session services are retrieved via the getService (../api/com/novell/service/session/Session.html#getService(java.lang.String)) method, while the session environment can be manipulated via the following methods: getEnvironment (../api/com/novell/service/session/Session.html#getEnvironment()), getEffectiveEnvironment (../api/com/novell/service/session/Session.html#getEffectiveEnvironment()), and setEnvironment (../api/com/novell/service/session/Session.html#setEnvironment(com.novell.service.session.SessionEnv)).
- SessionEnumerator (../api/com/novell/service/session/SessionEnumerator.html) class provides for the enumeration of sessions.
- SessionSearchEnumerator (../api/com/novell/service/session/SessionSearchEnumerator.html) class provides an implementation of SessionEnumerator containing only sessions that match a given search criteria.
- SessionManager (../api/com/novell/service/session/SessionManager.html) interface provides a single interface under which session providers can be abstracted. It provides for a collection of packages using one manager to tie the various abstracted session providers together. Because SessionManager is also a session, its session providers are accessible via its child sessions.

  The SessionManager interface provides no methods or constructors and only two variables (SCOPE_ATTR_ID and SESSION_MANAGER) for defining the scope and domain of the SessionManager. See the SessionManager (../api/com/novell/service/session/SessionManager.html) interface in the Reference Guide for more specific information on these variables.

  As the top level session, SessionManager is responsible for loading and managing initial providers. The SessionManager interface manages initial sessions by loading them via the

InitialSessionFactory property and maintaining the load order as the preferred service-request order.

Getting a SessionManager requires, at a minimum, the following code:

```
// Initialize Factories
 SessionEnv environment = new SessionEnv();

 // Get the session manager
 sm = SessionManagerFactory.getSessionManager(environment);
```

Each initial Session should be able to validate its sessions whenever requested to do so by calling the Session.validateLinks (../api/com/novell/service/session/Session.html#validateLinks()) method.

- SessionService (../api/com/novell/service/session/SessionService.html) interface describes a SessionService object containing service extensions for this session. The SessionService object is returned by the Session.getService (../api/com/novell/service/session/Session.html#getService(java.lang.String)) method. Typically, sessions are used only for gaining access to a set of services. Session services must be exposed and described by the provider offering those services via an object that implements SessionService (../api/com/novell/service/session/SessionService.html). Session providers each have an initial session and zero or more child sessions specific to that provider.
- SessionAttr (../api/com/novell/service/session/SessionAttr.html) class provides for a session attribute, which is an identifying characteristic of the session from which it was retrieved. It constructs a new instance of a session attribute based on two parameters: attrId (the ID for the session attribute), and value (the value of the session attribute). Methods are provided for getting the session attribute ID and the value of a particular session attribute.
- SessionAttrs (../api/com/novell/service/session/SessionAttrs.html) class provides for the creation of a collection of session attribute objects, and methods for adding, modifying, removing, and getting session attribute objects.
- SessionAttrEnumerator (../api/com/novell/service/session/SessionAttrEnumerator.html) class provides for the enumeration of session attributes.
- SessionEnv (../api/com/novell/service/session/SessionEnv.html) class provides a collection of key/value pairs describing a session environment, which is used to modify the default behavior of a session. It constructs a new, empty session environment that is shared, and it provides methods for adding, modifying, removing, getting and otherwise manipulating key/value sets.

  By default, setting the environment on a parent session affects all of its children the same way. This is because the actual environment checked for modified behavior is the inherited, or effective, environment. To override this inherited behavior, the environment key/value pair should be set per child session, or removed from the parent.
- SessionManagerFactory (../api/com/novell/service/session/SessionManagerFactory.html) class provides a class used to create a SessionManager, which is the top level object in a Session hierarchy. The SessionManagerFactory class is responsible for obtaining a SessionManager under a given scope.

Session.Bindery Package

The com.novell.service.session.bindery (../api/com/novell/service/session/bindery/package-summary.html) package provides a default implementation of sessions using a NetWare 3.x or 4.x server (if intended to be authenticated bindery) connections and security models. The bindery is contained in a flat hierarchy with no parent bindery sessions. Only one class, Bindery, of the bindery package is made public.

- Bindery (../api/com/novell/service/session/bindery/Bindery.html) class provides two static session attribute values for Bindery: The provider name for the bindery provider, and the domain name for the bindery initial session.

Session.NDS Package

The com.novell.service.session.nds (../api/com/novell/service/session/nds/package-summary.html) package provides a default implementation of sessions using NetWare 4.x and newer server connection and security models. NDSSessions are contained in a two-level hierarchy with NDSTreeSessions as parents of NDSServerSessions. Only one class, NDS, of the Session.NDS package is made public.

- NDS (../api/com/novell/service/session/nds/NDS.html) class provides three static session attribute values for NDS: The provider name for the NDS provider, the domain name for the NDS initial session, and the attribute ID describing the raw authentication state of the NDS session.

Session.Xplat Package

The com.novell.service.session.xplat (../api/com/novell/service/session/xplat/package-summary.html) package provides static attribute values and environment keys for Xplat-based providers. It provides for both bindery and NDS type sessions that have common roots in NetWare; thus both type sessions share some implementation in the session.xplat package. Only one class, Xplat, of the Session.Xplat package is made public.

- Xplat (../api/com/novell/service/session/xplat/Xplat.html) class provides static attribute values and environment keys for Xplat-based providers.

2.10.4 Important Session Manager Methods

Listed below are some important Session Manager methods of which to be aware. For a complete list of available methods, go to the Reference Guide (../api/overview-summary.html) documentation. Note that all other methods in the Authenticatable interface have been depreciated.

- Authenticatable.createIdentity (../api/com/novell/service/session/Authenticatable.html#createIdentity(java.lang.String)) - Creates and returns an Identity based on the current session.
- Authenticatable.isAuthenticated (../api/com/novell/service/session/Authenticatable.html#isAuthenticated()) - Determines if the object is authenticated.
- Session.close (../api/com/novell/service/session/Session.html#close()) - Closes and invalidates the current session and its children, removing all stored credentials.
- Session.FindSession (../api/com/novell/service/session/Session.html#findSession(java.lang.String)) - Returns the first Session object found with a domain name matching the passed in domainName parameter value.
- Session.FindSessionTop (../api/com/novell/service/session/Session.html#findSessionTop(java.lang.String)) - Returns the first Session object found with a matching domain name, starting at the Session Manager.

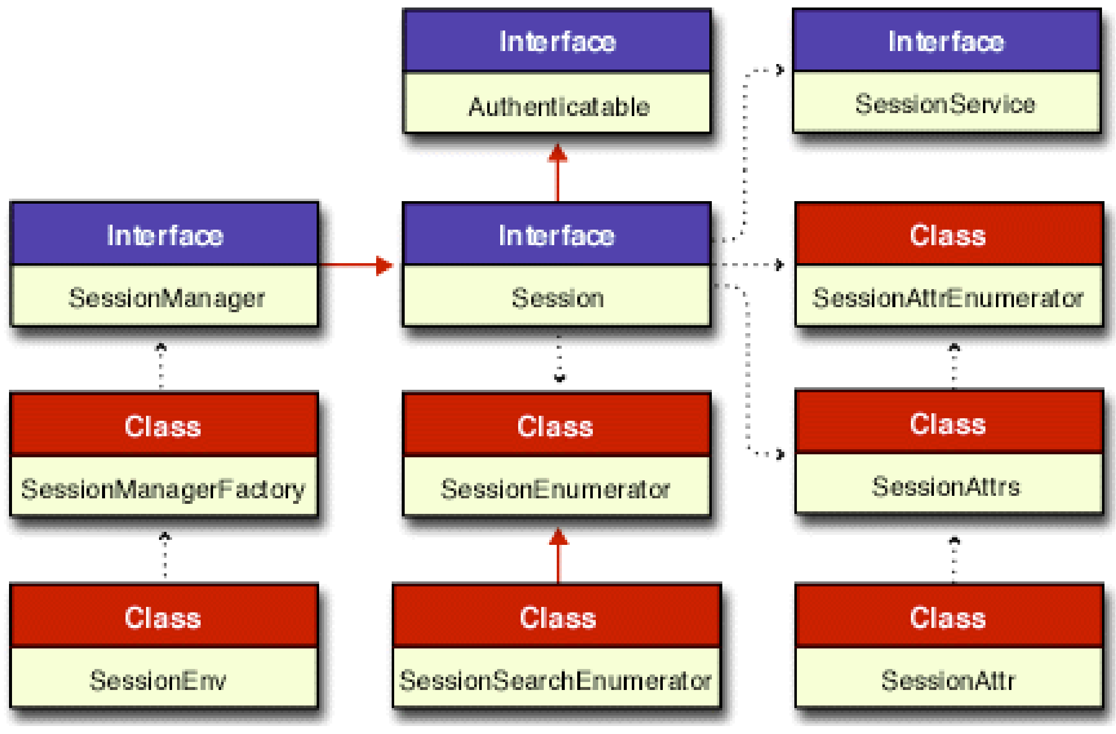
- Session.getAttributes (../api/com/novell/service/session/Session.html#getAttributes()) - Returns a selected default subset of all attributes as a SessionAttrs object for the current session.
- Session.getAttributes (../api/com/novell/service/session/Session.html#getAttributes(java.lang.String[])) - Returns the attributes for the current session listed in the String[] parameter as SessionAttrs objects.
- Session.getChildren (../api/com/novell/service/session/Session.html#getChildren()) - Returns an enumeration of child sessions for the current session as a SessionAttrs object.
- Session.hasChildren (../api/com/novell/service/session/Session.html#hasChildren()) - Determines if the current session has children.
- Session.getDomainName (../api/com/novell/service/session/Session.html#getDomainName()) - Returns the domain name of the current session.
- Session.getEffectiveEnvironment (../api/com/novell/service/session/Session.html#getEffectiveEnvironment()) - Returns the environment for the current session.
- Session.setEnvironment (../api/com/novell/service/session/Session.html#setEnvironment(com.novell.service.session.SessionEnv)) - Sets/replaces the environment for the current session and returns the old environment.
- Session.getEnvironment (../api/com/novell/service/session/Session.html#getEnvironment()) - Returns the environment for the current session.
- Session.getParent (../api/com/novell/service/session/Session.html#getParent()) - Returns the current session's parent session, or NULL if its parent Session doesn't exist.
- Session.hasParent (../api/com/novell/service/session/Session.html#hasParent()) - Determines if the current session has a parent.
- Session.getService (../api/com/novell/service/session/Session.html#getService(java.lang.String)) - Returns a session service extension for the current session.
- Session.getSession (../api/com/novell/service/session/Session.html#getSession(java.lang.String)) - Retrieves sessions relative to the target session and based on the the domain name that is passed in. It returns a child session of the parent based on the session (domainName) from which this operation is called.
- Session.getSession (../api/com/novell/service/session/Session.html#getSession(java.lang.String, com.novell.service.session.SessionEnv)) - Creates and/or retrieves sessions relative to the target session and based on the the domain name that is passed in.
- Session.getSessionTop (../api/com/novell/service/session/Session.html#getSessionTop(java.lang.String)) - Retrieves the first session matching the domain name that is passed in starting at the Session Manager.
- Session.getSessionTop (../api/com/novell/service/session/Session.html#getSessionTop(java.lang.String, com.novell.service.session.SessionEnv)) - Retrieves the first session matching the domain name that is passed in starting at the Session Manager and using environment to control behavior when applicable.
- Session.getUID (../api/com/novell/service/session/Session.html#getUID()) - Return the unique ID for the current session.
- Session.invalidate (../api/com/novell/service/session/Session.html#invalidate()) - Remove the passed in Session from its children, if possible without actually closing the Session.

- Session.isValid (../api/com/novell/service/session/Session.html#isValid()) - Determines if the current session is in a valid state.
- Session.validateLinks (../api/com/novell/service/session/Session.html#validateLinks()) - Validates the children of the current session.
- Session.search (../api/com/novell/service/session/Session.html#search(com.novell.service.session.SessionAttrs)) - Searches for sessions with matching attributes.
- SessionManagerFactory.getPrivate (../api/com/novell/service/session/SessionManagerFactory.html#getPrivate(com.novell.service.session.SessionEnv)) - Returns a unique, private instance of SessionManager.
- SessionManagerFactory.getSessionManager (../api/com/novell/service/session/SessionManagerFactory.html#getSessionManager(com.novell.service.session.SessionEnv)) - Returns a shared instance of SessionManager based on the information in the environment parameter.

2.10.5 Relationship of Session Manager Classes and Interfaces

The relationship between the Session Manager interfaces and classes is illustrated in the diagram below, followed by a brief description of the relationship each interface and class has with other interfaces and classes.

***Figure 2-14** Session Manager Interfaces and Classes*



In this diagram of interfaces and classes a solid red line ending with a solid arrow represent an extension of a class or interface. The dotted black lines ending with a small solid arrow imply some relationships between the different interfaces and classes. Links are provided to the Reference Guide documentation for each of these important Session interfaces and classes.

- Authenticatable (../api/com/novell/service/session/Authenticatable.html) interface is extended by the Session interface.
- Session (../api/com/novell/service/session/Session.html) interface extends the Authenticable interface and is extended by the SessionManager interface.
- SessionEnumerator (../api/com/novell/service/session/SessionEnumerator.html) class provides for an enumeration of sessions.
- SessionSearchEnumerator (../api/com/novell/service/session/SessionSearchEnumerator.html) class provides for an enumeration of sessions that match a given search criteria.
- SessionManager (../api/com/novell/service/session/SessionManager.html) interface extends the Session interface and is created by the SessionManagerFactory class.
- SessionManagerFactory (../api/com/novell/service/session/SessionManagerFactory.html) class provides a static class used to create a SessionManager, and is responsible for obtaining a SessionManager under a given scope.
- SessionEnv (../api/com/novell/service/session/SessionEnv.html) class provides the SessionManagerFactory with a collection of key/value pairs describing a session environment.
- SessionService (../api/com/novell/service/session/SessionService.html) interface describes an object type returned by the Session.getService() (../api/com/novell/service/session/Session.html#getService(java.lang.String)) method, which returns service extensions for the current session.
- SessionAttr (../api/com/novell/service/session/SessionAttr.html) class provides for a session attribute, which is an identifying characteristic of the session from which it was retrieved.
- SessionAttrs (../api/com/novell/service/session/SessionAttrs.html) class provides for the creation of a collection of session attribute objects.
- SessionAttrEnumerator (../api/com/novell/service/session/SessionAttrEnumerator.html) class provides for an enumeration of session attribute objects.

2.11 Remote Session Manager

The Remote Session Manager is available in the clientless version of NJCL (../../../../njclc.htm). Clients are not required to run Novell Client software, but the client machines must be connected to a network that includes a NetWare 5.x server (if NetWare 5.0, the server must be upgraded to run JVM v1.2) which listens for remote calls.

The Remote Session Manager supplies a three-tier solution as follows:

- Tier 1 is the local box, which uses Java's Remote Method Invocation (RMI) protocol to access Novell's JNDI service providers. A Java application requires a JVM and TCP/IP (any JVM-TCP/IP enabled machine). A Java enabled browser is an additional requirement for an applet.
- Tier 2 requires any NetWare 5.x server running the RMI version of NJCL. The NetWare server acts as a bridge translating RMI requests into NCP requests.
- Tier 3 is Novell's back-end services, which makes NCP requests and returns NCP replies.

The Remote Session Manager feature allows the user to specify the remote location where the SessionManager object is to reside. The local box will then access this remote SessionManager via the RMI protocol for any Session. This allows any JVM full access to all the features and functionality of the Novell JNDI providers and the session services. The local session layer will send all requests to the remote SessionManager, which will then return all results back to the local session layer. This process is transparent to the user. There is no need for the jncpv2r.dll or jncpv2r.nlm on the local box when using the remote SessionManager.

This allows an application to take full advantage of the multi-user capability of the session manager when running from any JVM. The multi-user capability is only available when running on a server when using the regular (non-clientless) version of the NJCL. See MultiUserSessionTest.java (../../../samplecode/njcl_sample/Session/MultiUserSessionTest.java.html) for sample code and "Multi User Session Manager" on page 90 for more information about this functionality.

Java Cryptography Extension (JCE) 1.2.1 is needed to encrypt passwords sent from the client to the RMI server. The JCE 1.2.1 zip file is included with the NJCL — Clientless but must be unzipped and made available on both the client ant the server. See the installation instructions in the JCE zip file for details.

Users of NJCL are not required to do anything with JCE except make the jar files available to applications and grant permissions in a policy file. An example of a simple policy file that would not be used for production applications is:

```
grant { permission java.security.AllPermission; };
```

Instructions and examples for more restrictive policy files are included in the JCE zip file.

The remote SessionManager must be running and listening for incoming requests before a local box tries to establish communication. This is done by running

```
java -Xbootclasspath:sys:\java\lib\rt.jar;sys:\java\lib\i18n.jar; -
Djava.security.policy=sys:\<policylocation>\<policy>
com.novell.service.session.spi.SessionManagerServiceImpl
```

where *policylocation* is the path of the security policy and *policy* is the policy filename. This creates a server instance and starts it listening for a connection request.

Only private SessionManagers can be requested from the remote side. This is done by calling the SessionManagerFactory.getPrivate(env) method. See the Session Package (../../../njcl/njcl_enu/apic/com/novell/service/session/package-summary.html) for details.

The SessionEnv.SESSION_MANAGER_URL (../../../njcl/njcl_enu/apic/com/novell/service/session/SessionEnv.html#SESSION_MANAGER_URL) key controls running in RMI or non-RMI mode. The session mananger URL is the URL of the remote server location.

You can specify your own custom sockets for RMI to use with the SessionEnv.RMI_SOCKET_FACTORIES_OBJECT (../../../njcl/njcl_enu/apic/com/novell/service/session/SessionEnv.html#RMI_SOCKET_FACTORIES_OBJECT) key. For more information, see the RMI documentation about custom sockets and the RMISocketFactories (../../../njcl/njcl_enu/apic/com/novell/service/session/spi/RMISocketFactories.html) class.

The following sample code illustrates how to use the remote SessionManager.

```
/
**********************************************************************
*
 Copyright (c) 1999 Novell, Inc. All Rights Reserved.
 THIS WORK IS SUBJECT TO U.S. AND INTERNATIONAL COPYRIGHT LAWS AND
TREATIES.
```

```
 USE AND REDISTRIBUTION OF THIS WORK IS SUBJECT TO THE LICENSE
AGREEMENT
 ACCOMPANYING THE SOFTWARE DEVELOPMENT KIT (SDK) THAT CONTAINS THIS
WORK.
 PURSUANT TO THE SDK LICENSE AGREEMENT, NOVELL HEREBY GRANTS TO
DEVELOPER A
 ROYALTY-FREE, NON-EXCLUSIVE LICENSE TO INCLUDE NOVELL'S SAMPLE CODE
IN ITS
 PRODUCT. NOVELL GRANTS DEVELOPER WORLDWIDE DISTRIBUTION RIGHTS TO
MARKET,
 DISTRIBUTE, OR SELL NOVELL'S SAMPLE CODE AS A COMPONENT OF DEVELOPER'S
 PRODUCTS. NOVELL SHALL HAVE NO OBLIGATIONS TO DEVELOPER OR DEVELOPER'S
 CUSTOMERS WITH RESPECT TO THIS CODE.
***********************************************************************
**/
import java.util.*;
import javax.naming.*;
import javax.naming.directory.*;
import com.novell.service.session.*;

public class RemoteSessionTest
{
   public static void main(String args[])
   {
      if(args.length != 1)
      {
         System.out.println("Usage: Java RemoteSessionTest
rmiServerName");
        System.out.println("Example: java RemoteSessionTest
                        137.65.255.255");
         System.exit(0);
      }

      try
      {
         String rmiServerName = args[0];

         SessionEnv env = new SessionEnv();
         //Comment out the next line to run in non-RMI mode.
         env.add(SessionEnv.SESSION_MANAGER_URL, rmiServerName);
         SessionManager sm = SessionManagerFactory.getPrivate(env);

         Context ctx1 = getNetWareDirContext(sm);
         printEnum(ctx1.toString(), ctx1.list(""));
         Context ctx2 = (Context)ctx1.lookup("Trees");
         printEnum(ctx2.toString(), ctx2.list(""));

         System.out.println("Successful");
      }
      catch(Exception e)
      {
         dumpException(e);
      }
   }
   public static Context getNetWareDirContext(SessionManager sm)
      throws Exception
   {
      Hashtable hash = new Hashtable();
      hash.put(Context.INITIAL_CONTEXT_FACTORY,
               "com.novell.service.nw.NetWareInitialContextFactory");

      //This is the key piece - provide the SessionManager you want
this
        provider to use.

hash.put(com.novell.utility.naming.Environment.SESSION_MANAGER_OBJECT,
sm);

      Context ctx1 = new InitialDirContext(hash);
      Context ctx2 = (Context)ctx1.lookup("");
         //eliminate middle man - (InitialContext)
      return ctx2;
   }

   public static void printEnum(String msg, NamingEnumeration ne)
   {
      int count=0;
      System.out.println(msg);
      while(ne.hasMoreElements())
      {
         count++;
         System.out.println(ne.nextElement());
      }
      System.out.println("Total Elements:" + count);
   }

   synchronized static public void dumpException(Throwable e)
   {
      System.out.flush();
      System.err.flush();
      System.out.println("Dumping Exception...");
      while(e != null)
      {
         e.printStackTrace();
         if (e instanceof HasRootCauses)
         {
            System.out.println("\nRoot causes...");
            Enumeration enum = ((HasRootCauses)e).getRootCauses();
            while (enum.hasMoreElements())
            {
               dumpException((Throwable)enum.nextElement());
            }
         }
         if (e instanceof HasRootCause)
            e = ((HasRootCause)e).getRootCause();
         else if (e instanceof NamingException)
            e = ((NamingException)e).getRootCause();
         else if(e instanceof java.rmi.RemoteException)
```

```
            e = ((java.rmi.RemoteException)e).detail;
         else
            e = null;
         if(e != null)
            System.out.println("Root cause...");
      }
      System.err.flush();
      System.out.flush();
   }
}
```

NJCL and JNDI Solutions

3

The NJCL and JNDI solutions provided here describe how to use some of the classes and interfaces contained in the NJCL and JNDI API libraries. A demo utility is made available for each solution to assist in your development work. Each demo is associated with sample source code to which links are provided within each section.

- Section 3.1, "How to Create a JNDI Shell," on page 101
- Section 3.2, "How to Create a Session Shell," on page 106
- Section 3.3, "How to Create a GUI-based Browser," on page 111
- Section 3.4, "How to Create an NJCL Applet," on page 116
- Section 3.5, "How to Program eDirectory with JNDI," on page 124

3.1 How to Create a JNDI Shell

This solution describes how to create a JNDI shell that accepts commands from an input stream and executes these commands. A demo JNDIShell browser, with accompanying source code, is provided to show the functionality of this JNDI shell solution.

- Section 3.1.1, "Creating a JNDI Shell," on page 101
- Section 3.1.2, "Executing the JNDI Shell Demo," on page 105
- Section 3.1.3, "JNDIShell Source Code," on page 106

3.1.1 Creating a JNDI Shell

The JNDI shell implementation consists of two major components (JNDIShell class and shell class) with other supporting interfaces and classes.

- "JNDIShell Class" on page 101
- "shell Class" on page 102
- "Other Supporting Interfaces and Classes" on page 104

JNDIShell Class

The JNDIShell class contains the JNDI specific shell implementations. Sample source code is provided by the JNDIShell.java (../../../samplecode/njcl_sample/JNDIShell/JNDIShell.java.html) file. The primary purpose of the JNDIShell class is to control the shell state. The specific tasks included in this class are as follows:

1. Defines the input stream from which commands are read and the output stream to which results are written.
2. Defines the continueWStdIn flag and the verbose flag as described in the sample source code.
3. Sets up a constructor for a child shell with the parent shell, initial context, input stream, and output stream as parameters.
4. Sets up constructors for several JNDIShell objects with different parameters, such as initial context, input stream, output stream, defined flags, and so forth.

5. Sets the current JNDI context associated with the shell to facilitate relative context operations.
6. Sets up a loop in the main run method for reading in the command line from the input stream, checking and processing the command line, registering and de-registering commands, and writing the results to the output stream.
7. Provides the following methods for streams operations, resolving names, registering and de-registering commands, and performing relative context operations:
    - addCommand(String, Command) - Binds a command to a string command name in the shell.
    - removeCommand(String) - Unbinds a command from a string command name in the shell.
    - getCommand(String) - Returns a command bound to the specific command name.
    - resolve(String) - Resolves a name relative to the initial context specified in the constructor.
    - resolveRelative(NamedContext, String) - Resolves a name relative to a provided context.
    - resolveRelativeAtomic(NamedContext, String) - Resolves an atomic name relative to a provided context.
    - getCurrCtx - Returns the current context as a NamedContext object.
    - setCurrCtx(NamedContext) - Sets the current context to the context specified.
    - setPrompt(String) - Sets the prompt to that specified.
    - getVerbose - Returns the verbose state.
    - setVerbose(boolean) - Sets the verbose state to TRUE or FALSE.
    - getInputStream - Returns the input stream attached to the shell.
    - getOutputStream - Returns the output stream attached to the shell.

shell Class

The shell class is a generic shell that does general initialization, parses the command line, and controls input and output. Its primary purpose is to set up the environment in which the JNDIShell class can operate. This generic shell contains two principal methods (main and runShell) and 23 internal classes, which define the different default commands that are implemented.

Sample source code is provided by the shell.java (../../../samplecode/njcl_sample/JNDIShell/shell.java.html) file.

main Method

The primary purpose of the main method is to interpret any strings passed to it as command line parameters. More specifically it performs the following functions:

- Takes as input a string array of arguments and sets up a command line parser for interpreting the strings passed in.
- Initializes optional variables and flags, as well as input and output streams.
- Looks for the initial context factory property, which is necessary for the class java.naming.InitialContext to function correctly. The InitialContext gets this system property and treats the contents as a fully distinguished class name, which should implement java.naming.spi.InitialContextFactory.
- Loads and runs the shell with the default command bindings by calling the runShell method.

runShell Method

The primary purpose of the runShell method is to run the shell given a set of minimal options and the necessary input and output streams. More specifically it performs the following functions:

- Sets up the environment for and spawns a JNDIShell object according to the options specified.
- Defines the default commands and binds the command-name pairs to the shell. These commands may be loaded and unloaded during the course of the shell execution. You can implement your own commands by creating a class that implements either the Command interface or the CommandWithHelp interface.
- Sets the default initialContext to be

  ```
  com.novell.service.nw.NetWareInitialContetFactory
  ```

  if no other initial context is provided.
- Runs the shell given a set of minimal options and the necessary input and output streams.

runShell Command Classes

A command class is included for each of the 23 default shell commands. Because each command class implements the CommandWithHelp interface, each has a getHelpStrs method for printing the command help information. These commands are bound to the shell, and because of the load and unload commands, all of them can be changed during the course of shell execution.

- Load Command (*load*) adds a command class to the shell under a command name.
- Unload Command (*unload*) removes a command from the shell.
- Dir Command (*dir*) lists bindings of the current context or relative context.
- Cd Command (*cd*) changes the current context to a relative context.
- CdAtomic Command (*cda*) changes the current context to a literal relative context. It tracks the name using the current context's name parser.
- CdSchemaRoot Command (*cds*) starts a new shell with the initial context at the schema root of the specified relative DirContext.
- CdSchemaClassDef Command (*cdcdef*) starts a new shell with the initial context at the schema class definition of the specified relative DirContext.
- CdSchemaAttributeDef Command (*cdadef*) starts a new shell with the initial context at the schema attribute definition of the specified attribute of a relative DirContext.
- CdSchemaSyntaxDef Command (*cdsdef*) starts a new shell with the initial context at the schema syntax definition of the specified attribute of a relative DirContext.
- Attr Command (*attr*) dumps all or specified attributes of a DirContext.
- Show Command (*show*) prints the toString representation of a bound object.
- Rename Command (*ren*) renames a bound object.
- Bind Command (*bind*) binds a relative object to the current context under a specified name.
- Rebind Command (*rebind*) rebinds a relative object to the current context under a specified name.
- Unbind Command (*unbind*) unbinds a relative object to the current context under a specified name.
- Create Command (*create*) creates a subcontext relative to the current context.

- Destroy Command (*del*) destroys a subcontext relative to the current context.
- Env Command (*env*) displays the environment of the current context, allowing modifications.
- Shell Command (*shell*) executes a shell script.
- Exec Command (*exec*) executes an OS-specific command.
- VerboseToggle Command (*v*) toggles the verbose flag in the shell.
- Exit Command (*exit, quit, q*) exits the shell.
- Help Command (*help, h, ?*) prints help for all of the default bound commands.

Other Supporting Interfaces and Classes

The JNDIShell class and the shell class use the following additional interfaces and classes.

- Command interface (../../../samplecode/njcl_sample/JNDIShell/Command.java.html) - Provides a generic interface for a command. A class that implements this interface must be registered with the shell. When the command is entered, all arguments are passed to the command through the execute method. This interface is used by both the JNDIShell and shell classes.
- CommandWithHelp interface (../../../samplecode/njcl_sample/JNDIShell/CommandWithHelp.java.html) - Provides a generic interface for a command that supports help through extending Help by adding the getHelpStrs method. A class that implements this interface must be registered with the shell. When the command is entered, all arguments are passed to the command through the execute method. This interface is used by only the shell class.
- LineReader class (../../../samplecode/njcl_sample/JNDIShell/LineReader.java.html) - Provides support for reading text input lines from an arbitrary input stream or reader. This class is used only in the JNDIShell class.
- NamedContext class (../../../samplecode/njcl_sample/JNDIShell/NamedContext.java.html) - Contains a Context and the name that can be used to look up that context from a specified initial context. This class is used by both the JNDIShell and shell classes.
- NamedObject class (../../../samplecode/njcl_sample/JNDIShell/NamedObject.java.html) - Contains an object and the name required to look it up. This demonstrates functionality that can be used to look up an object from the specified name. This class is used by both the JNDIShell and shell classes.
- NullOutputStream class (../../../samplecode/njcl_sample/JNDIShell/NullOutputStream.java.html) - Extends OutputStream and acts as a sink for bytes that should disappear. This class is used only by the JNDIShell class.
- ShellException class (../../../samplecode/njcl_sample/JNDIShell/ShellException.java.html) - Used to pass error messages from a command to the shell. If an error occurs in a command, a ShellException can be thrown with a message. The name of the command, along with the message, is then printed, and execution continues. If the exitShell flag is set, then the shell will exit. This class is used by both the JNDIShell and shell classes.
- shell.ini file (../../../samplecode/njcl_sample/JNDIShell/shell.ini.html) - Contains the initial environment settings. This configuration file can be edited to use JNDI providers other than those supplied by Novell® by setting the environment key 'java.naming.factory.initial' to the class name of the desired initial context factory.

3.1.2 Executing the JNDI Shell Demo

The JNDI Shell demo is a text-based browser that allows you to explore JNDI providers. It is designed for users who have a basic knowledge of the JNDI Context and DirContext interfaces. JNDIShell can call basic JNDI Context and DirContext methods.

Some possible uses for JNDIShell are:

- As an exploratory tool to discover how different JNDI providers work.
- As a debugging tool, allowing you to save debugging time by providing calls to basic NJCL Context and DirContext methods. To test specific situations, run JNDIShell and execute the command you desire.

To run the JNDIShell demo utility, follow the procedures below.

Setting up JNDIShell locally

1. Go to the c:\novell\Ndk\samples\njcl_sample\jndishell directory (default download).
2. Compile JNDIShell by entering the following on the command line:
   ```
   javac shell.java
   ```
3. Run JNDIShell by entering the following on the command line:
   ```
   java shell
   ```
4. Get command line help by adding the '-?' option. For example, enter
   ```
   java shell -?
   ```

Implementing Your Own Commands

1. Create a class that implements either the Command or the CommandWithHelp interfaces.
2. Load the command into the shell by typing
   ```
   load <cmdclass> <cmdname>
   ```
   where <cmdclass> is the name of the class you created and <cmdname> is the command name to which you want to bind the class.

Changing the Initial Environment

Edit the shell.ini file to change the initial environment. To use the JNDI providers other than those supplied by Novell, set the environment key 'java.naming.factory.initial' to the class name of the desired initial context factory.

Getting Help on Commands

For all commands, at the command line, enter

```
?
```

For one particular command, at the command line, enter

```
? <cmd>
```

3.1.3 JNDIShell Source Code

This source code demonstrates how to create the functionality of a generic shell that accepts commands from an input stream and executes these commands.

shell.java (../../../samplecode/njcl_sample/JNDIShell/shell.java.html) sets up a useful environment in which the JNDIShell class can operate. This is the file that is compiled and executed.

JNDIShell.java (../../../samplecode/njcl_sample/JNDIShell/JNDIShell.java.html) is the JNDI shell class that accepts commands from an input stream using LineReader and then executes these commands.

Command.java (../../../samplecode/njcl_sample/JNDIShell/Command.java.html) provides a generic interface for commands.

CommandWithHelp.java (../../../samplecode/njcl_sample/JNDIShell/CommandWithHelp.java.html) provides a generic interface for commands that support help.

LineReader.java (../../../samplecode/njcl_sample/JNDIShell/LineReader.java.html) provides support for reading text input lines from an arbitrary input stream or reader.

NamedContext.java (../../../samplecode/njcl_sample/JNDIShell/NamedContext.java.html) provides a context and the name that can be used to look up that context from an initial context.

NamedObject.java (../../../samplecode/njcl_sample/JNDIShell/NamedObject.java.html) provides an object and the name required to look up the object.

NullOutputStream.java (../../../samplecode/njcl_sample/JNDIShell/NullOutputStream.java.html) acts as a sink for bytes that should disappear.

ShellException.java (../../../samplecode/njcl_sample/JNDIShell/ShellException.java.html) is used to pass error messages from a command to the shell.

shell.ini (../../../samplecode/njcl_sample/JNDIShell/shell.ini.html) sets up the initial environment configuration.

3.2 How to Create a Session Shell

This JNDI solution describes how to create a text-based browser that allows you to explore the Session Manager hierarchy. The SessionShell code was taken from the JNDIShell code and altered to work with the Session Manager package. A SessionShell demo utility, with accompanying source code, is provided to show the functionality of this browser solution.

- Section 3.2.1, “Creating a Session Shell,” on page 106
- Section 3.2.2, “Executing the SessionShell Demo,” on page 110
- Section 3.2.3, “SessionShell Source Code,” on page 111

3.2.1 Creating a Session Shell

The Session shell implementation consists of two major components (SessionShell class and shell class) with other supporting interfaces and classes.

- “SessionShell Class” on page 107
- “shell Class” on page 107
- “Other Supporting Interfaces and Classes” on page 109

SessionShell Class

The SessionShell class contains the Session Manager specific shell implementations. Sample source code is provided in the SessionShell.java (../../../samplecode/njcl_sample/SessionShell/SessionShell.java.html) file. The primary purpose of the SessionShell class is to control the shell state. The specific tasks included in this class are as follows:

1. Defines the input stream from which commands are read and the output stream to which results are written.
2. Defines the continueWStdIn flag and the verbose flag as described in the sample source code, and initializes the current session.
3. Sets up a constructor for a child shell with the parent shell, initial session, input stream, and output stream as parameters.
4. Sets up constructors for several SessionShell objects with different parameters, such as initial session, input stream, output stream, defined flags, and so forth.
5. Sets up a loop in the main run method for reading in the command line from the input stream, checking and processing the command line, registering and de-registering commands, and writing the results to the output stream.
6. Provides the following methods for streams operations, resolving names, registering and de-registering commands, and performing relative context operations.
    - addCommand(String, Command) - Binds a command to a string command name in the shell.
    - removeCommand(String) - Unbinds a command from a string command name in the shell.
    - getCommand(String) - Returns the command bound to the specified command name.
    - getCommandNames - Returns all command names bound to the shell.
    - clearCommands - Clears all bound commands in the shell.
    - getCurrSession - Returns the current session.
    - setCurrSession(Session) - Sets the current session to the specified Session.
    - setPrompt(String) - Sets the prompt to that specified.
    - getVerbose - Returns the verbose state.
    - setVerbose(boolean) - Sets the verbose state to TRUE or FALSE.
    - getInputStream - Returns the input stream attached to the shell.
    - getOutputStream - Returns the output stream attached to the shell.

shell Class

The shell class is a generic shell that does general initialization, parses the command line, and controls input and output. Its primary purpose is to set up the environment in which the SessionShell class can operate. This generic shell contains two principal methods (main and runShell) and 22 internal classes, which define the different default commands that are implemented.

Sample source code is provided in the shell.java (../../../samplecode/njcl_sample/SessionShell/shell.java.html) file.

main Method

The primary purpose of the main method is to interpret any strings passed to it as command line parameters. It performs the following specific functions:

- Takes as input a string array of arguments and sets up a command line parser for interpreting the strings passed in.
- Initializes optional variables and flags (continueWStdIn and verbose), the shell.ini environment configuration file, and the input and output streams.
- Looks for the initial session factory property, and sets the default to be "com.novell.service.session.nds.NDSInitialSessionFactory:" + "com.novell.service.session.bindery.BinderyInitialSessionFactory".
- Adds several other properties as defaults, including any environment properties set in the shell.ini configuration startup file.
- Loads and runs the shell with the default command bindings by calling the runShell method.

runShell Method

The primary purpose of the runShell method is to run the shell given a set of minimal options and the necessary input and output streams. It performs the following specific functions:

- Sets up the environment for and spawns a SessionShell object according to the options specified.
- Defines the default commands and binds the command-name pairs to the shell. These commands may be loaded and unloaded during the course of the shell execution. You can implement your own commands by creating a class that implements either the Command interface or the CommandWithHelp interface.
- Runs the shell given a set of minimal options and the necessary input and output streams.

runShell Command Classes

A command class is included for each of the 22 default shell commands. Because each command class implements the CommandWithHelp interface, each has a getHelpStrs method for printing the command help information. These commands are bound to the shell, and because of the load and unload commands, all of them can be changed during the course of shell execution.

- Load Command (*load*) adds a command class to the shell under a command name.
- Unload Command (*unload*) removes a command from the shell.
- Dir Command (*dir*) lists bindings of the current context or relative context.
- Cd Command (*cd*) changes the current context to a relative context.
- Find Command (*find*) changes the current session to a relative session via the findSession method.
- Attr Command (*attr*) dumps all or specified attributes of a DirContext.
- Debug Command (*debug*) turns debugging on and off.
- Finalizer Command (*final*) runs the finalizer.
- Trace Command (*trace*) turns instruction and/or method tracing on and off.
- Close Command (*close*) closes the current session.

- Keep Command (*keep*) keeps the current session.
- Update Command (*update*) updates session links.
- Loging Command (*loging*) attempts to log in to the domain using the GUI authenticator based only on user name.
- Login Command (*login*) attempts to log in to the domain based on the username and password identity.
- Logoutg) Command (*logoutg*) attempts to log out of the domain using the GUI authenticator based only on the user name.
- Logout Command (*logout*) attempts to log out of the domain based on the user name and password identity.
- Pause Command (*pause*) pauses the shell.
- Shell Command (*shell*) executes a shell script.
- Exec Command (*exec*) executes an OS-specific command.
- VerboseToggle Command (*v*) toggles the verbose flag in the shell.
- Exit Command (*exit, quit, q*) exits the shell.
- Help Command (*help, h, ?*) prints help for all of the default bound commands.

Other Supporting Interfaces and Classes

The SessionShell class and the shell class uses the following additional interfaces and classes.

- Command interface (../../../samplecode/njcl_sample/SessionShell/Command.java.html) - Provides a generic interface for a command. A class that implements this interface must be registered with the shell. When the command is entered, all arguments are passed to the command through the execute method. This interface is used by both the SessionShell and shell classes.
- CommandWithHelp interface (../../../samplecode/njcl_sample/SessionShell/CommandWithHelp.java.html) - Provides a generic interface for a command that supports help through extending Help by adding the getHelpStrs method. A class that implements this interface must be registered with the shell. When the command is entered, all arguments are passed to the command through the execute method. This interface is used only by the shell class.
- LineReader class (../../../samplecode/njcl_sample/SessionShell/LineReader.java.html) - Provides support for reading text input lines from an arbitrary input stream or reader. This class is used only in the SessionShell class.
- NullOutputStream class (../../../samplecode/njcl_sample/SessionShell/NullOutputStream.java.html) - Extends OutputStream and acts as a sink for bytes that should disappear. This class is used only by the SessionShell class.
- ShellException class (../../../samplecode/njcl_sample/SessionShell/ShellException.java.html) - Used to pass error messages from a command to the shell. If an error occurs in a command, a ShellException can be thrown with a message. The name of the command along with the message are then printed, and execution continues. If the exitShell flag is set, then the shell will exit. This class is used by both the SessionShell and shell classes.
- shell.ini file (../../../samplecode/njcl_sample/SessionShell/shell.ini.html) - Contains the initial environment settings. This configuration file can be edited to use session providers other than

those supplied by Novell by changing the com.novell.service.session.spi.InitialSessionFactory environment key to be different from the default.

3.2.2 Executing the SessionShell Demo

The SessionShell demo is designed for users who have a basic knowledge of the Session Manager. Basic session manager methods can be called. Some possible uses for SessionShell are:

- As a tool to explore and discover how Session Manager works.
- As a debugging tool, SessionShell allows you to save debugging time by providing calls to Session Manager methods. To test specific situations, run SessionShell and execute the desired command.

To run the SessionShell demo utility, follow the procedures given below.

Setting up SessionShell Locally

1. Go to the c:\novell\Ndk\samples\njcl_sample\SessionShell directory.
2. Compile SessionShell by entering the following at the command line:
   ```
   javac shell.java
   ```
3. Run SessionShell by entering the following at the command line:
   ```
   java shell
   ```
4. Get command line help by adding the '-?' line option. For example, at the command line, enter
   ```
   java shell -?
   ```

Implementing Your Own Commands

To implement your own commands, create a class that implements either the Command or the CommandWithHelp interfaces. Then load the command into the shell by typing

```
load <cmdclass> <cmdname>
```

where <cmdclass> is the name of the class you created and <cmdname> is the command name to which you want to bind the class.

Changing the initial environment

To change the initial environment, edit the shell.ini file. To use different providers, change com.novell.service.session.spi.InitialSessionFactory to be different from the default.

Getting Help on Commands

For all commands in the shell, at the command line, enter

```
?
```

For one particular command, at the command line, enter

```
? <cmd>
```

3.2.3 SessionShell Source Code

This source code demonstrates how to create the functionality of a session shell that accepts commands from an input stream and executes these commands.

shell.java (../../../samplecode/njcl_sample/SessionShell/shell.java.html)sets up a useful environment in which the SessionShell class can operate. This is the file that is compiled and executed.

SessionShell.java (../../../samplecode/njcl_sample/SessionShell/SessionShell.java.html) Provides a generic session shell that accepts commands from an input stream and executes these commands.

Command.java (../../../samplecode/njcl_sample/SessionShell/Command.java.html) provides a generic interface for commands.

CommandWithHelp.java (../../../samplecode/njcl_sample/SessionShell/CommandWithHelp.java.html) provides a generic interface for commands that support help.

LineReader.java (../../../samplecode/njcl_sample/SessionShell/LineReader.java.html) provides support for reading text input lines from an arbitrary input stream or reader.

NullOutputStream.java (../../../samplecode/njcl_sample/SessionShell/NullOutputStream.java.html) acts as a sink for bytes that should disappear.

ShellException.java (../../../samplecode/njcl_sample/SessionShell/ShellException.java.html) is used to pass error messages from a command to the shell.

shell.ini file (../../../samplecode/njcl_sample/SessionShell/shell.ini.html) sets up the initial environment configuration.

3.3 How to Create a GUI-based Browser

This JNDI solution describes how to create the functionality of a GUI-based browser that accepts commands from an input stream and executes these commands. A demo NSIBrowser utility, with accompanying source code, is provided to show the functionality of this browser solution.

- Section 3.3.1, “Creating a GUI Browser,” on page 111
- Section 3.3.2, “Executing the NSIBrowser Utility,” on page 115
- Section 3.3.3, “NSIBrowser Source Code,” on page 115

3.3.1 Creating a GUI Browser

The GUI Browser implementation consists of two major components (NSIBrowser class and NSIBrowserFrame class) with other supporting interfaces and classes.

- “NSIBrowser Class” on page 111
- “NSIBrowserFrame Class” on page 112
- “Other Supporting Interfaces and Classes” on page 114

NSIBrowser Class

The NSIBrowser class contains the main method for the NSI Browser (GUI) application. Sample source code is provided by the NSIBrowser.java (../../../samplecode/njcl_sample/NSIBrowser/NSIBrowser.java.html) file. The primary purpose of the NSIBrowser class is to construct and start up the browser frame. The specific tasks included in this class are as follows:

1. Sets up the configuration file (NSIBrowser.ini) information and property names (initial context, host, list classes and list bindings).

2. Sets up the starting main method used for the standalone NSIBrowser application. The main method takes as input the command line arguments and initializes an NSIBrowser object (browser). It then calls the browser.init method.
3. The browser.init method is a required part of an applet and is responsible for constructing and displaying the program's window (frame). It first initializes a mainFrame object from the NSIBrowserFrame object, and then it sets the size of the browser mainFrame. It then checks for a NSIbrowser configuration file by calling the readConfigFile method, following which the NSIBrowser properties and initial context are set in the mainFrame.
4. The readConfigFile method checks for an NSI browser configuration file and uses it, if it exists, by reading in the configuration file information and updating the system properties with any changes needed (only previously undefined properties). It merges the configuration file with the system properties and checks to make sure the initial context property is set.
5. A saveConfigFile method is provided for saving the updated configuration information (system properties) to the configuration file.

NSIBrowserFrame Class

The NSIBrowserFrame class provides for displaying, processing, and updating the program's one or more browser window(s) and its components. Sample source code is provided by the NSIBrowserFrame.java (../../../samplecode/njcl_sample/NSIBrowser/NSIBrowserFrame.java.html) file. The components of the main window include the following:

- Title Bar - Displays the application's name and version.
- Menu Bar - Provides structure of main- and sub-menus.
- Current Path Area - Shows the current path.
- Name List Area - Shows the object available at the current path.
- Initial Context Area - Displays the initial context setting.

The NSIBrowser constructor creates the browser's main window (frame) and all of its components. In addition to the main window components listed above, the constructor does the following:

- Constructs and adds the file menu.
- Constructs and adds the search menu.
- Constructs and adds the options menu.
- Constructs and adds the help menu.
- Sets the window's menu bar to the one just constructed.
- Constructs and adds the name list display area.
- Adds the current path line.
- Adds the status line.
- Makes sure notification is given for window events.
- Eliminates wasted space before the frame gets displayed by calling a pack method.

The following methods are provided for processing and updating the program's browser windows.

actionPerformed Method

The actionPerformed(ActionEvent) method is the action handler for the main window (frame). It handles any menu item selections, including closing the window and terminating the application, and it handles event buttons and events in the listbox.

WindowListener functions

The following WindowListener methods are implemented:

- The windowClosed(WindowEvent) method closes the current window.
- The windowDeiconified (WindowEvent) method deiconifies the current window.
- The windowIconified (WindowEvent) method iconifies the current window.
- The windowActivated (WindowEvent) method activates the current window.
- The windowDeactivated (WindowEvent) method deactivates the current window.
- The windowOpened (WindowEvent) method opens the current window.
- The windowClosing (WindowEvent) method closes the current window.

setProperties Method

The setProperties(Properties) method set the properties for the browser frame based on the Properties parameter input.

updateList Method

The updateList method updates the name list component using the initial context and the current name using the initCtx and currName variables.

resetInitCtx Method

The resetInitCtx method updates the initial context variable (initCtx) using the new initial context as specified in the system properties.

showAttributes Method

The showAttributes method displays the attributes (if any) for the currently selected item.

lookupContext Method

The lookupContext method looks up a context and begins a new browser frame for that context.

startNewContextFrame Method

The startNewContextFrame(String, Context) method starts a new browser frame for a given context based on the input parameters (title name and starting context).

callSpecialHandler Method

The callSpecialHandler method handles ’special’ button requests. A ’special’ request indicates that the user wants to interact with the ’real’ object. Since this can happen in so many different ways, a list of property entries is used to find a special handler for the object selected, based on its class name. This allows anyone interested to extend this browser to do just about anything.

Other Supporting Interfaces and Classes

The NSIBrowser class and the NSIBrowserFrame class use the following additional interfaces and classes:

- FileSpecialHandler class (../../../samplecode/njcl_sample/NSIBrowser/FileSpecialHandler.java.html) - implements the NSISpecialHandler class to create a special handler for a FileDirContext object. This interface is used only by the NSIBrowserFrame class.
- NSIAttributeFrame class (../../../samplecode/njcl_sample/NSIBrowser/NSIAttributeFrame.java.html) - is responsible for displaying a list of attributes for a given DirContext. This interface is used only by the NSIBrowserFrame class.
- NSIForm class (../../../samplecode/njcl_sample/NSIBrowser/NSIForm.java.html) - provides a common way for entry forms to be created. Neither the NSIBrowser class nor the NSIBrowserFrame class uses this class.
- NSIInitialContextFrame class (../../../samplecode/njcl_sample/NSIBrowser/NSIInitialContextFrame.java.html) - handles setting the initial context parameter for JNDI. This interface is used by both the NSIBrowser class and the NSIBrowserFrame class.
- NSILookupFrame class (../../../samplecode/njcl_sample/NSIBrowser/NSILookupFrame.java.html) - handles looking up an object for JNDI. This interface is used only by the NSIBrowserFrame class.
- NSIMessageBox class (../../../samplecode/njcl_sample/NSIBrowser/NSIMessageBox.java.html) - displays a dialog box with a title, optional message and OK button. This interface is used only by the NSIBrowserFrame class.
- NSISpecialHandler class (../../../samplecode/njcl_sample/NSIBrowser/NSISpecialHandler.java.html) - defines an interface for special handlers. A special handler is designed to handle the specifics of a particular object type. This object type is defined by object class and the binding is stored as a property in the browser’s configuration file or system properties. This interface is used only by the NSIBrowserFrame class.
- QuickSort class (../../../samplecode/njcl_sample/NSIBrowser/QuickSort.java.html) - sorts a partial or an entire array. This interface is used only by the NSIBrowserFrame class.
- Sortable class (../../../samplecode/njcl_sample/NSIBrowser/Sortable.java.html) - sorts two objects input as parameters. This interface is used only by the NSIBrowserFrame class.
- SortableString class (../../../samplecode/njcl_sample/NSIBrowser/SortableString.java.html) - sorts two strings input as parameters. This interface is used only by the NSIBrowserFrame class.
- NSIBrowser.ini file (../../../samplecode/njcl_sample/NSIBrowser/NSIBrowser.ini.html) - Contains the initial environment settings. This configuration file can be edited to use JNDI providers other than those supplied by Novell by setting the environment key 'java.naming.factory.initial' to the class name of the desired initial context factory.

3.3.2 Executing the NSIBrowser Utility

NSIBrowser is a GUI-based browser that allows you to explore JNDI providers. It is designed for users who have a basic knowledge of the JNDI Context and DirContext interfaces. Basic JNDI Context and DirContext methods can be called. Some possible uses for NSIBrowser are

- As an exploratory tool to discover how different JNDI providers work.
- As a debugging tool, allowing you to save debugging time by providing calls to basic NJCL Context and DirContext methods. To test your code, run NSIBrowser and execute the command you desire.

To run the NSIBrowser demo utility, follow the procedures below.

Setting up NSIBrowser Locally

1. Go to the c:\novell\Ndk\samples\njcl_sample\nsibrowser directory (default download).
2. Compile NSIBrowser by entering the following command on the command line:

   ```
   javac *.java
   ```
3. Run NSIBrowser by entering the following command on the command line:

   ```
   java NSIBrowser
   ```

Implementing Your Own Special Handlers

1. Create a class that implements the NSISpecialHandler interface.
2. Add the handler to the NSIBrowser.ini file using the following syntax:

   ```
   jnos.browser.handler.<classname>=<handler>
   ```

   where <classname> is the full class name of the class you want to handle and <handler> is the full class name of your custom handler to which you want to bind the class. For example, the FileSpecialHandler looks like this (although it is hard coded into the sample):

   ```
   jnos.browser.handler.com.novell.service.file.nw.naming.fileDirCont
   ext=FileSpecial Handler
   ```

Changing the Initial Environment

Edit the shell.ini file to change the initial environment. To use the JNDI providers other than those supplied by Novell, set the environment key 'java.naming.factory.initial' to the class name of the desired initial context factory.

3.3.3 NSIBrowser Source Code

These samples demonstrate how to create the functionality of a GUI-based browser that accepts commands from an input stream and executes these commands.

NSIBrowser.java (../../../samplecode/njcl_sample/NSIBrowser/NSIBrowser.java.html) contains the main method for the NSI Browser (GUI) application, and is responsible for constructing and starting up the browser frame.

NSIBrowserFrame.java (../../../samplecode/njcl_sample/NSIBrowser/NSIBrowserFrame.java.html) provides for displaying, processing, and updating the program's one or more browser window(s) and its components.

FileSpecialHandler.java (../../../samplecode/njcl_sample/NSIBrowser/FileSpecialHandler.java.html) provides an implementation of a special handler for a FileDirContext object.

NSIAttributeFrame.java (../../../samplecode/njcl_sample/NSIBrowser/NSIAttributeFrame.java.html) displays a list of attributes for a given DirContext.

NSIForm.java (../../../samplecode/njcl_sample/NSIBrowser/NSIForm.java.html) provides a common way for entry forms to be created.

NSIInitialContextFrame.java (../../../samplecode/njcl_sample/NSIBrowser/NSIInitialContextFrame.java.html) handles setting the initial context parameter for JNDI.

NSILookupFrame.java (../../../samplecode/njcl_sample/NSIBrowser/NSILookupFrame.java.html) handles looking up an object for JNDI.

NSIMessageBox.java (../../../samplecode/njcl_sample/NSIBrowser/NSIMessageBox.java.html) constructs and shows a dialog box with a title and an OK button.

NSISpecialHandler.java (../../../samplecode/njcl_sample/NSIBrowser/NSISpecialHandler.java.html) Defines an interface for special handlers designed to handle the specifics of a particular object type defined by object class.

QuickSort.java (../../../samplecode/njcl_sample/NSIBrowser/QuickSort.java.html) sorts a partial or an entire array.

Sortable.java (../../../samplecode/njcl_sample/NSIBrowser/Sortable.java.html) sorts two objects input as parameters.

SortableString.java (../../../samplecode/njcl_sample/NSIBrowser/SortableString.java.html) sorts two strings input as parameters.

NSIBrowser.ini file (../../../samplecode/njcl_sample/NSIBrowser/NSIBrowser.ini.html) sets up the initial environment configuration.

3.4 How to Create an NJCL Applet

This JNDI solution describes how to create the functionality of an NJCL applet, which shows some of the capability of JNDI and NJCL. A demo NJCLApplet utility, with accompanying source code, is provided to show the functionality of this browser solution.

- Section 3.4.1, “Creating an NJCL Applet,” on page 116
- Section 3.4.2, “Executing the NJCL Applet Utility,” on page 121
- Section 3.4.3, “NJCLApplet Source Code,” on page 124

3.4.1 Creating an NJCL Applet

The NJCL applet consists of one major components (NJCLApplet class) with other supporting interfaces and classes.

- “NJCLApplet Class” on page 117
- “Other Supporting Interfaces and Classes” on page 118

The NJCL applet is intended to demonstrate the following popular features of NJCL and the Novell providers for JNDI:

JNDI Features

- list
- lookup

- attributes
- search
- schema

Session Management

- public session manager

Authentication

- login
- logout

Services

- NDS replicas and partitions
- File streams
- Print queue file submission
- Server statistics

NJCLApplet Class

The NJCLApplet class is the applet root for the NJCL applet demo. It implements the InterfaceCompare interface, and it contains the init method, which is called by the applet manager, for the NJCL applet. Source code for this demo applet is provided in the NJCLApplet.java (../../../samplecode/njcl_sample/NJCLApplet/src/NJCLApplet.java.html) file. The NJCLApplet class initializes the rootFrame, the PopupHandler, the pluginParamStr, and the paramInfo (which consists of an array of strings describing each parameter). The following methods are provided for setting up and launching the NJCL applet:

init

The init method first initializes the properties, which are based on the system properties, the default properties, and properties passed in as parameters. It then combines the default properties with the applet parameters passed in, keeping the applet parameters if specified. Any plug-ins specified in the HTML are used first. Finally init sets up the browser implementation as follows:

1. Gets an AboutFrame object, which provides an About dialog (frame) listing the applet's features. Because this About frame provides a lot of useful information that may be needed as a continual guide, it is created with all controls, and is configured so that it is hidden and then shown when needed.
2. Constructs a ContextTreeNode object, which represents a JNDI Context as a tree node. The node is constructed based on a frame to which this node belongs and the user object for the node. A number of methods are provided for configuring and manipulating the parameters and variables.
3. Gets the initial context and changes it to a real context based on the Context property passed in.
4. Sets up the Message Box for reporting errors and exceptions.

getParameterInfo

The getParameterInfo method returns a string array of information about the supported applet parameters.

getAppletInfo

The getAppletInfo method returns a String of information about the application suitable for an About dialog. The information string includes the applet name, version, author, and copyright information.

implementsInterface

The implementsInterface(Object) method determines if a class implements the interface passed in as the Object parameter and returns a boolean set to TRUE if the passed-in object is the correct interface.

getDefaultProperties

The getDefaultProperties method returns the default properties of the applet. The properties returned are the default or hard coded set.

main

The main(String) method allows the applet to be launched as an application. The input parameter String contains the command line arguments.

Other Supporting Interfaces and Classes

The NJCLApplet class utilizes the following additional interfaces and classes.

- AboutFrame class (../../../samplecode/njcl_sample/NJCLApplet/src/AboutFrame.java.html) - provides an About dialog (frame) that lists the applet's features. The AboutFrame constructor creates the AboutFrame object with all controls. The getAboutFrame(URL) method returns an AboutFrame based on the URL parameter, which specifies where the about.html file is located. An ActionListener event handler method, actionPerformed(ActionEvent), is provided for describing what action occurred.
- ContextFrame class (../../../samplecode/njcl_sample/NJCLApplet/src/ContextFrame.java.html) - provides a graphical view control for a JNDI context. The constructor creates the GUI portion of the ContextFrame object, given a root node as a ContextTreeNode object. A setStatusLabel(String) method sets the status label to the given message and returns the previous contents, and a setRootFlag(boolean) method sets the root flag for the frame. An ActionListener event handler method, actionPerformed(ActionEvent), is provided for describing what action occurred. A group of MouseListener interface methods are provided for various mouse events. Finally, the following handler methods are provided for action events:

  attributeButton_Action(ActionEvent) for handling the Attribute button and pop-up menu item.
  classDefMenuItem_Action(ActionEvent) for handling the ClassDef pop-up menu item.
  lookupButton_Action(ActionEvent) for handling the Lookup button and pop-up menu item.
  rebindMenuItem_Action(ActionEvent) for handling the Rebind pop-up menu item.
  renameMenuItem_Action(ActionEvent) for handling the Rename pop-up menu item.
  schemaMenuItem_Action(ActionEvent) for handling the Schema button.

lookupMenuItem_Action(ActionEvent) for handling the lookup menu item.
searchMenuItem_Action(ActionEvent) for handling the search menu item.
showClassesMenuItem_Action(ActionEvent) for handling the show classes menu item.
refreshMenuItem_Action(ActionEvent) for handling the refresh menu item.
aboutMenuItem_Action(ActionEvent) for handling the about menu item.

- InterfaceCompare interface (../../../samplecode/njcl_sample/NJCLApplet/src/InterfaceCompare.java.html) - defines the implementsInterface(Object) method, which checks to see if a class implements a given interface. The input Object parameter is the object to be checked. TRUE is returned if the object implements the right interface.
- LookupFrame class (../../../samplecode/njcl_sample/NJCLApplet/src/LookupFrame.java.html) - provides a graphical input dialog for retrieving the lookup string. The constructor creates the GUI portion of the lookup frame. An inner class, LookupWindowAdapter, is provided for handling focus change and window close events. An ActionListener event handler method, actionPerformed(ActionEvent), is provided for handling the action events. Finally, a getText method returns the text entered, or NULL if none is entered.
- MessageBox class (../../../samplecode/njcl_sample/NJCLApplet/src/MessageBox.java.html) - provides a message dialog box for the applet based on the parent Frame object.
- PopupHandler interface (../../../samplecode/njcl_sample/NJCLApplet/src/PopupHandler.java.html) - defines a snap-in that keys off a pop-up menu. Custom object handlers can be defined to provide additional functionality for an object based on its class or any of the interfaces it implements. This allows for endless extensibility to the common browser interface provided in this applet. A single method, popupRequested(JFrame, JPopupMenu, Object), is provided for the registered handlers when the pop-up event occurs. The JFrame parameter is the frame in which the pop-up is being requested; the JPopupMenu is the pop-up menu being created; and the Object parameter is the object being customized, which is likely to be a ContextTreeNode object.
- RenameFrame class (../../../samplecode/njcl_sample/NJCLApplet/src/RenameFrame.java.html) - provides a graphical input dialog for renaming a context. The constructor creates the GUI portion of the rename frame. An inner class, RenameWindowAdapter, handles focus change and window close events. An ActionListener event handler method, actionPerformed(ActionEvent), is provided for handling action events. A getText method returns the text entered, or NULL if no text is entered.
- SearchFrame class (../../../samplecode/njcl_sample/NJCLApplet/src/SearchFrame.java.html) - provides a graphical input dialog for retrieving the search string. The constructor creates the GUI portion of the search frame (SearchFrame object) based on the JFrame parameter. An inner class, SearchWindowAdapter, handles focus change and window close events. An ActionListener event handler method, actionPerformed(ActionEvent), is provided for handling action events. A getText method returns the text entered, or NULL if no text is entered.
- Util class (../../../samplecode/njcl_sample/NJCLApplet/src/Util.java.html) - provides the following general purpose utility functions, which are not graphical in nature and have no JNDI dependencies:

getPropertiesFromParams(Applet, String) retrieves the properties from the Applet parameters.

mergeProperties(Properties, Properties, boolean) constructs a property list by merging the new property list with the original property list. If the boolean parameter, preserve, is set to TRUE the new properties will be added to the original property list, not overwriting them.

getHashTableFromProperties(Properties) converts the recursive Properties object into a flag Hashtable.

loadExtensions(InterfaceCompare, String) returns an array of objects that implements a given interface. The input parameter InterfaceCompare contains the compare object to use against the objects found in the directory specified by the String parameter.

loadExtensions(InterfaceCompare, Properties, String) returns an array of objects that implements a given interface. The input parameter InterfaceCompare contains the compare object to use against the objects found in the directory; the Properties parameter contains the properties to search for the plug-ins; and the String parameter, paramName, contains the parameter name of the plug-ins to load.

getExceptionTrace(Throwable) returns a String that includes an exception's stack trace. The Throwable object input parameter contains the exception for which to get the trace.

addGridBagComponent(Container, Component, int, int, int, int, double, double) sets up the GridBagConstraints for an object and adds it to the Container using a GridBagLayout.

An internal class, ClassFilenameFilter, is provided to accept only ".class" files. Its only method, accept(File, String) determines if the input directory path parameter (File) and file name (String) are acceptable for the filter.

- AuthPopupHandler.java (../../../samplecode/njcl_sample/NJCLApplet/src/plugins/AuthPopupHandler.java.html) - provides a snap-in that keys off a pop-up menu for authentication. It requires a public NULL constructor. The popupRequested(JFrame, JPopupMenu, Object) method calls for registered handlers when the pop-up event occurs. The JFrame parameter is the frame in which the pop-up is being requested; the JPopupMenu parameter is the pop-up menu being created; and the Object parameter is the object being customized, which is likely to be a ContextTreeNode.

  An internal class, NWAuthActionListener, handles NetWare® session style authentication. This internal class has two constructors: a default NULL constructor required for plug-ins, and a constructor with four parameters that hangs onto a session.

  The following ActionListener methods are provided for handling action events:

  actionPerformed(ActionEvent) describes what action occurred.

  login_Action(ActionEvent) handles login menu item actions.

  logout_Action(ActionEvent) handles logout menu item actions.

- DataPopupHandler.java (../../../samplecode/njcl_sample/NJCLApplet/src/plugins/DataPopupHandler.java.html) - provides a snap-in that keys off of a pop-up menu for file display. The handler basically provides a text/hex file viewer for anything that can have a stream (via DataAccessable). It requires a public NULL constructor. The popupRequested(JFrame, JPopupMenu, Object) method calls for registered handlers when the pop-up event occurs. The JFrame parameter is the frame in which the pop-up is being requested; the JPopupMenu parameter is the pop-up menu being created; and the Object parameter is the object being customized, which is likely to be a ContextTreeNode.

  An internal class, DataActionListener, handles NetWare data objects. This internal class has two constructors: A default NULL constructor required for plug-ins, and a constructor with four parameters that hangs onto the context.

  The following ActionListener methods are provided for handling action events:

  actionPerformed(ActionEvent) describes what action occurred.

  data_Action(ActionEvent) handles data menu item actions.

  Another internal class, DataViewFrame, presents a text or hex file viewer. The constructor creates a file viewer in text mode based on two parameters: the name of the file to be viewed and the context to be used to access the file. An ActionListener method, actionPerformed(ActionEvent) handles four different button action events: ASCII button, hex button, save button and close button. Two additional methods are provided for setting the main panel view: showText sets the main panel view to text mode view, and showHex sets the main panel view to hex mode view.

- ServerPopupHandler.java (../../../samplecode/njcl_sample/NJCLApplet/src/plugins/ServerPopupHandler.java.html) - provides a snap-in that keys off a pop-up menu for simple server management. It requires a public NULL constructor. The popupRequested(JFrame, JPopupMenu, Object) method calls for registered handlers when the pop-up event occurs. The JFrame parameter is the frame in which the pop-up is being requested; the JPopupMenu parameter is the pop-up menu being created; and the Object parameter is the object being customized, which is likely to be a ContextTreeNode.

  An internal class, SWServerActionListener, handles NetWare server objects. This internal class has two constructors: A default NULL constructor required for plug-ins, and a constructor with four parameters that hangs onto the server.

  The following ActionListener methods are provided for handling action events:

  actionPerformed(ActionEvent) describes what action occurred.

  server_Action(ActionEvent) handles server menu item actions.

  Another internal class, ServerMgmtFrame, presents and handles server management features. This internal class has two constructors: a default NULL constructor required for plug-ins, and a constructor with two parameters: the name of the server and the server context to be used. An ActionListener method, actionPerformed(ActionEvent) handles three different button action events: load NLM button, unload NLM button, and close button.

3.4.2 Executing the NJCL Applet Utility

The purpose of this utility is to demonstrate a few of the capabilities of JNDI and NJCL. NJCL includes several providers for JNDI. These providers for JNDI encompass much of NetWare’s functionality. These providers are defined under the Java package com.novell.service and are included in the following sub-packages:

- bindery - NetWare 3.x legacy resource database
- file - NetWare file system
- ncpext - NetWare Core Protocol Extensions
- nds - Novell Directory Services (NDS)
- nw - Provides a desktop metaphor to NDS and legacy servers
- qms - NetWare Queue Management System (legacy print)
- server - NetWare server

The browser metaphor allows you to explore the features available. A pop-up menu is available by right-clicking an object in the browser. This menu contains both generic JNDI features (above the line) and custom NetWare features (below the line) to demonstrate the mixture of JNDI and NetWare calls.

The following JNDI features are demonstrated:

- Context.list feature shows the browser tree.
- Context.lookup feature is used when a tree is expanded and also in the View|Lookup menu item and pop-up menu on any Context.

- Context.rename feature is used to view the pop-up menu on any object.
- DirContext.getAttributes feature is used to view the Attribute button and the pop-up menu on any DirContext.
- DirContext.getSchema feature is used to view the pop-up menu on any DirContext.
- DirContext.search feature is used to view the View|Search menu item.

The following Novell Provider features are demonstrated:

- Authentication feature shows Login and Logout support through a pop-up menu on any context with a Session.
- File feature shows file view/edit support through a pop-up menu on any object supporting DataAccessable (files).
- Server feature shows server information and NLM load and unload through a pop-up menu on a server object.

Setting up the Server for the Web Administrator

1. Install NetWare 5.
2. Install the latest NetWare 5 Support Pack.
3. Install the Novonyx Fasttrack Server for NetWare.
4. Create a directory under SYS:Novonyx\suitespot\docs for your applet.

   Map a network drive to this directory as follows:

   ```
   map i:=\\myserver\sys\novonyx\sujitespot\docs
   ```

   Following this drive mapping create an applet directory as follows:

   ```
   i:
   md njcl
   ```
5. Copy the jar files, class files, and HTML files to the applet directory just created.

   ```
   map i:=\\myserver\sys\novonyx\sujitespot\docs\njcl
   copy c:\novell\java\lib i:
   copy demo.jar i:
   copy plugins.jar i:
   ```

Setting up the Applet's HTML File for the Web Administrator

1. Create the applet's HTML file (possibly with a text editor), specifying the applet and the archives. Then specify the following parameters (no file I/O and no System property access is allowed):
   - Initial context factory
   - Provider URL
   - Security information
   - Other parameters as needed

   Following is an example applet HTML file.

   ```
   <HTML>
   <HEAD<
   ```

```
<TITLE>NJCL Applet - (C) Copyright Novell, Inc.</TITLE>
</HEAD>
<BODY>
<APPLET CODE="NJCLApplet.class"
ARCHIVE="demo.jar,plugins.jar,njcl.jar,jndi.jar,swing.jar,jgl3.1.0
.jar" WIDTH=600 HEIGHT=500>
PARAM name="java.naming.factory.initial"
value="com.novell.service.ldap.LdapCtxFactory">
<PARAM name="java.naming.provider.url" value="ldap://
123.123.123.256/">
>PARAM name="java.naming.security.authentication" value="simple">
<PARAM name="plugins" value="AuthPopupHandler;DataPopupHandler;
ServerPopupHandler;">
</APPLET>
</BODY>
</HTML>
```

2. Install and run the Java Plug-in HTML converter.

   This converter is installed somewhere in your Start menu. Do not change the generated HTML because there are too many oddities in it. You should change the source HTML and run the converter again.

3. Copy the converted HTML file to the Web server as follows:

   ```
   map i:=\\myserver\sys\novonyx\suitespot\docs\njcl
   ```

   ```
   copy ldap.html i:
   ```

Setting Up the Client for the Application User

1. Install and load the Netscape Communicator (tested with version 4.5)
2. Load the HTML file you created for the applet

   ```
   http://123.123.123.256/njcl/ldap.html
   ```

   If the Java plug-in is not available, the browser will take you to the proper page where the plug-in can be downloaded. After installing the plug-in, reload the Web page. Note: the NJCLApplet has been tested with JRE 1.1.7b and JRE 1.2 plug-ins.

Troubleshooting

If you do not see the applet and you get the message "Applet not Initialized", use the Java plug-in console window to view why the applet may be failing to load. Load the Java plug-in control panel from the start menu and set the Show Java Console option to ON (in the Basic panel).

If you need to debug or report an error, turn off the JIT (in the Advanced panel) so you can see/report the code line numbers. The JRE 1.2 does a better job of reporting errors; however, you can force the applet to load with the JRE 1.2 by selecting it in the Advanced panel. When you run an applet, the Java console shows which JRE version is being used.

If you do not see new applet features, or a bug that has been fixed still causes a failure on your machine, the local JAR files may need to be updated or removed. The browser and Java plug-ins use the classpath set in the client. If NJCL has been installed, the classpath usually points to some local copy of the JAR files. These will be used before the network is used, thus changes made to the Web site will be unavailable until the local JAR files are updated or removed. Also, keep in mind that the local JAR files cannot be updated while they are open (applet is running).

If you've updated the HTML file but the changes don't seem to take effect, make the changes in the original file and run the HTML again. We recommend that you not make changes to the converted applet HTML file because it is difficult to make the changes correctly. Instead, change the original file and rerun the HTML. Also, if the backup file exists, the HTML converter will not convert the file and does not report an error. Be sure to delete the backup before running the HTML converter, or specify another backup path.

3.4.3 NJCLApplet Source Code

These samples demonstrate how to create the functionality of an NJCL applet, which shows some of the capability of JNDI and NJCL.

NJCLApplet.java (../../../samplecode/njcl_sample/NJCLApplet/src/NJCLApplet.java.html) demonstrates some of the more popular features of NJCL and the Novell providers for JNDI.

AboutFrame.java (../../../samplecode/njcl_sample/NJCLApplet/src/AboutFrame.java.html) provides an About dialog (frame) listing the application's features.

ContextFrame.java (../../../samplecode/njcl_sample/NJCLApplet/src/ContextFrame.java.html) provides a graphical view control for a JNDI context.

InterfaceCompare.java (../../../samplecode/njcl_sample/NJCLApplet/src/InterfaceCompare.java.html) defines a method for checking to see if a class implements a given interface.

LookupFrame.java (../../../samplecode/njcl_sample/NJCLApplet/src/LookupFrame.java.html) provides a graphical input dialog for retrieving the lookup string.

MessageBox.java (../../../samplecode/njcl_sample/NJCLApplet/src/MessageBox.java.html) provides a message box for the application.

PopupHandler.java (../../../samplecode/njcl_sample/NJCLApplet/src/PopupHandler.java.html) defines a snap-in that keys off a pop-up menu.

RenameFrame.java (../../../samplecode/njcl_sample/NJCLApplet/src/RenameFrame.java.html) provides a graphical input dialog for renaming a context.

SearchFrame.java (../../../samplecode/njcl_sample/NJCLApplet/src/SearchFrame.java.html) provides a graphical input dialog for retrieving the search string.

Util.java (../../../samplecode/njcl_sample/NJCLApplet/src/Util.java.html) provides some general purpose utility functions that are not graphical and have no JNDI dependencies.

AuthPopupHandler.java (../../../samplecode/njcl_sample/NJCLApplet/src/plugins/AuthPopupHandler.java.html) Plug-in provides a snap-in that keys off a pop-up menu for authentication.

DataPopupHandler.java (../../../samplecode/njcl_sample/NJCLApplet/src/plugins/DataPopupHandler.java.html) Plug-in provides a snap-in that keys off a pop-up menu for file display.

ServerPopupHandler.java (../../../samplecode/njcl_sample/NJCLApplet/src/plugins/ServerPopupHandler.java.html) Plug-in provides a snap-in that keys off a pop-up menu for simple server management.

3.5 How to Program eDirectory with JNDI

The Novell DeveloperNet University has developed a White Pages Application course, which can help you learn eDirectory Services programming. This tutorial provides instructions in writing the OrgBuilder white pages application.

To access this tutorial, go to the following URL: http://developer.novell.com/education/tutorials/whitepages_jndi/index.htm (http://developer.novell.com/education/tutorials/whitepages_jndi/index.htm)

NJCL and JNDI Tasks

4

This section contains tasks that explain how to perform various NJCL and JNDI functions. The first section list those operations most usually performed by basic and advanced JNDI applications. The other Tasks topics contain information on programming various functions in the different NJCL and JNDI areas. For information on tasks associated with complete solutions see "NJCL and JNDI Solutions" on page 101, which has a demo utility and sample source code for each solution.

- Section 4.1, "Tasks Performed by JNDI Applications," on page 127
- Section 4.2, "JNDI Tasks," on page 127
- Section 4.3, "Authenticator Tasks," on page 130
- Section 4.4, "Session Tasks," on page 130

4.1 Tasks Performed by JNDI Applications

All JNDI applications generally perform the same basic operations, which are discussed below.

- Section 4.1.1, "Operations Performed by Basic JNDI Applications," on page 127
- Section 4.1.2, "Operations Performed by Advanced JNDI Applications," on page 127

4.1.1 Operations Performed by Basic JNDI Applications

1. Get an initial context for a naming system.
2. List subordinates of an initial context.
3. Look up a subordinate context by its name.
4. Search for a context subordinate to another context.

4.1.2 Operations Performed by Advanced JNDI Applications

1. Access the attributes of a directory context.
2. Add or delete attributes of a directory context.
3. Access the schema class definition of a directory context.
4. Examine a context's schema to determine whether it is a particular NDS class.
5. Extend a provider's schema to enhance the naming system's functionality.
6. Create new providers for access to naming systems not covered in the core Novell® providers.
7. Establish explicit bindings to a given context.

4.2 JNDI Tasks

This section covers the following tasks:

- Section 4.2.1, "Creating an Initial Context," on page 128
- Section 4.2.2, "Looking up a Context," on page 129

* Section 4.2.3, “Accessing Attributes,” on page 129

4.2.1 Creating an Initial Context

To create an initial context, you must do the following:

1. Identify all the values needed for the naming system initial context.
   a. Initial Context Factory

      The first value is the initial context factory, which is listed in the documentation for the naming system's service provider. The initial context factories are also stored as static String variables in the NJCL class com.novell.utility.naming.Environment (../api/com/novell/utility/naming/Environment.html).

      For example, the NDS initial context factory has the String value

      ```
      Environment.NDS_INITIAL_CONTEXT_FACTORY
      ```
   b. Provider URL

      The second value is the provider URL. The NDS provider URL takes the form of

      ```
      nds://[tree name]/[fully distinguished name]
      ```

      as in

      ```
      nds://planetary/.admin.administration
      ```
   c. Other Information On a Per-provider Basis

      You may have to specify other values in order to create the initial context. Refer to the documentation for a naming system's service provider to learn about other values it may require.
2. Set the JNDI properties for the initial context by creating a Hashtable object with these values.

   Creating an initial context requires that you set certain JNDI properties for the creation of that context. JNDI uses key/value pairs associated with java.util.Hashtable to represent a JNDI property. The JNDI interface javax.naming.Context stores the hashtable keys as static String variables. You determined the value for this key in the previous step.

   For example, the key for the initial context factory pair has the String value

   ```
   Context.INITIAL_CONTEXT_FACTORY
   ```

   and the key for the provider URL pair has the String value

   ```
   Context.PROVIDER_URL
   ```

   Use the Hashtable.put() method to insert key/value pairs into the hashtable as in the following:

   ```
   Hashtable hash = new Hashtable();
   hash.put("key", "value");
   ```
3. Create a new initial context using this Hashtable object.

   In order to construct a new instance of an initial context, you must tell the JNDI naming manager about the appropriate JNDI properties it should use to create that context. You set those properties in the previous step when creating the Hashtable object. Once you have created the initial context, you do not need to do anything else with it in order to start accessing the naming system with JNDI.

Code Sample:

The Java class Acl.java (../../../samplecode/njcl_sample/NDS/Acl.java.html) has a static method called createInitialContext(), which demonstrates how to create an initial context. See the section of code in Acl.java beginning with the following line of code:

```
private static DirContext createInitialContext
   (String _providerURL)
```

Related Topics:

- "Novell Services (including JNDI Providers)" on page 39

4.2.2 Looking up a Context

To look up a context, you must do the following:

1. Get a context.
2. Call its lookup method.
3. Cast it to the object you want.

Code Sample:

The Java class NetWareObjectLister.java (../../../samplecode/njcl_sample/General/NetWareObjectLister.java.html) has the following line of code that demonstrates how to look up a context:

```
Object obj =
   initialCtx.lookup(MyEnvironment.DISTINGUISHED_NAME);
```

4.2.3 Accessing Attributes

To access attributes, you must do the following:

1. Get a DirContext.
2. Call its getAttributes method.

   The getAttributes method takes the name of the context whose attributes you want to get as a parameter.

   To get your own context's attributes, pass in an empty string.

   When the getAttributes() method is called, an Attributes interface is returned.
3. With the Attributes interface, do one of the following:
   - Call the getAll() method, which returns a Naming Enumeration containing all of the attributes as a group of Attributes.
   - Call the getIds() method, which returns a Naming Enumeration containing all of the Ids found in the Attributes interface as a group of Strings.

   Both of these methods return a Naming Enumeration, which can contain objects of different classes.
4. Traverse through the group of Attributes or Strings until you find the one you want.
5. If you have an individual Attribute, you need to get its values in order to work with it.

Code Sample:

The Java class Acl.java (../../../samplecode/njcl_sample/NDS/Acl.java.html) has a static method called printACL(), which demonstrates how to access attributes. See the section of code in Acl.java beginning with the following line or code:

```
private static boolean printACL(DirContext target)
```

Related Reference Guide Topics:

- com.novell.service.nds.NdsObjectACL (../api/com/novell/service/nds/NdsObjectACL.html)

4.3 Authenticator Tasks

- Section 4.3.1, "Creating an Identity Object (Login)," on page 130

Other topics are currently being developed.

4.3.1 Creating an Identity Object (Login)

To create an Identity object (login), you must do the following:

1. Create an Identity object.
2. Construct the Identity with its appropriate identity scopes.

Code Sample:

The Java class Login.java (../../../samplecode/njcl_sample/Authenticator/Login.java.html) demonstrates how to create an identity object.

Related Topics:

- "Authentication Services" on page 41

Related Reference Guide Topics:

- com.novell.java.security.Authenticator (../api/com/novell/java/security/Authenticator.html)
- com.novell.java.security.Identity (../api/com/novell/java/security/Identity.html)
- com.novell.java.security.IdentityScope (../api/com/novell/java/security/IdentityScope.html)

4.4 Session Tasks

- Section 4.4.1, "Authenticating a Session," on page 130

Other topics are currently being developed.

4.4.1 Authenticating a Session

To authenticate a session, you must do the following:

1. Get a context.

2. Get the context's session.
3. Create an Identity from the context's session.
4. Use the Authenticator to log it in.

Code Sample:

The Java class LoginSession.java (../../../samplecode/njcl_sample/Session/LoginSession.java.html) demonstrates how to authenticate a session.

Related Topics:

- "Authentication Services" on page 41
- "Session Manager Services" on page 89

Related Reference Guide Topics:

- com.novell.service.session.Session (../api/com/novell/service/session/Session.html)
- com.novell.service.session.Authenticatable (../api/com/novell/service/session/Authenticatable.html)
- com.novell.java.security.Authenticator (../api/com/novell/java/security/Authenticator.html)
- com.novell.java.security.Identity (../api/com/novell/java/security/Identity.html)

API Reference 5

NJCL (../api/overview-summary.html)

The published, supported classes and interfaces of Class Libraries for Java, including the NJDI service providers.

NOTE: The NJCL class libraries documented in this reference are recommended for programmer use. Other classes and interfaces not documented here are implementation specific, and are not necessary. Some of the source code for undocumented NJCL class libraries may be provided for debugging purposes; however, only the documented class libraries are supported by Novell®.

NJCL—Clientless (../apic/overview-summary.html)

The Clientless version of the published, supported classes and interfaces of Novell's Class Libraries for Java, including the NJDI service providers.

JDK 1.2.2 (http://java.sun.com/products/archive/j2se/1.2.2_017)

The complete reference to the Java core interfaces and classes. This documentation resides on the Internet.

Sample Source Code 6

The sample source code listings provided with this software development kit show how to use many of the classes and interfaces contained in the JNDI and NJCL API libraries. Sample source code is provided for most of the Novell® services. See "NJCL and JNDI Solutions" on page 101 for additional sample source code associated with the solutions and accompanying demo utilities.

This section covers the following samples:

- Section 6.1, "Authenticator Samples," on page 135
- Section 6.2, "Bindery Samples," on page 136
- Section 6.3, "NCP Extensions Samples," on page 136
- Section 6.4, "NDS Provider Samples," on page 136
- Section 6.5, "NetWare File System Samples," on page 136
- Section 6.6, "NetWare (General) Samples," on page 137
- Section 6.7, "Queue Samples," on page 138
- Section 6.8, "Server Samples," on page 138
- Section 6.9, "Session Samples," on page 138

6.1 Authenticator Samples

These samples show how to do authentications, including working with tokens, identities, and login and logout.

CreateTokens.java (../../../samplecode/njcl_sample/Authenticator/CreateTokens.java.html) creates a token or authentication login secret using an NDS or bindery identity.

CreateTokensPI.java (../../../samplecode/njcl_sample/Authenticator/CreateTokensPI.java.html) creates a token or authentication login secret using a password identity.

ModifyTokens.java (../../../samplecode/njcl_sample/Authenticator/ModifyTokens.java.html) modifies a token or authentication login secret using a passed in NDS or bindery identity.

ModifyTokensPI.java (../../../samplecode/njcl_sample/Authenticator/ModifyTokensPI.java.html) modifies a token or authentication login secret using the passed in password identity.

VerifyTokens.java (../../../samplecode/njcl_sample/Authenticator/VerifyTokens.java.html) verifies a token or authentication login secret using a passed in NDS or bindery identity.

VerifyTokensPI.java (../../../samplecode/njcl_sample/Authenticator/VerifyTokensPI.java.html) verifies a token or authentication login secret using the passed in password identity.

GetIdentities.java (../../../samplecode/njcl_sample/Authenticator/GetIdentities.java.html) generates a list of currently authenticated Identity objects either with or without an IdentityScope.

Login.java (../../../samplecode/njcl_sample/Authenticator/Login.java.html) performs a login of the passed in Identity object to its scope.

LoginPI.java (../../../samplecode/njcl_sample/Authenticator/LoginPI.java.html) performs a login of the passed in password Identity object to its scope.

Logout.java (../../../samplecode/njcl_sample/Authenticator/Logout.java.html) performs a logout of the passed in Identity object to its scope.

LogoutPI.java (../../../samplecode/njcl_sample/Authenticator/LogoutPI.java.html) performs a logout of the passed in password Identity object to its scope.

6.2 Bindery Samples

This sample demonstrate how to perform manipulations of bindery objects.

NetWareObjectLister.java (../../../samplecode/njcl_sample/General/NetWareObjectLister.java.html) provides a section of code describing how to list all the bindery objects associated with the specified server.

6.3 NCP Extensions Samples

This sample demonstrates how to perform various NCPExtension handler tasks.

SendNCPExtReq.java (../../../samplecode/njcl_sample/NCP/SendNCPExtReq.java.html) demonstrates how to find a server and an NCPExtension handler with which to exchange requests.

6.4 NDS Provider Samples

These samples demonstrate how to perform various directory service tasks, such as adding and deleting trustees from an object's ACL, creating a NetWare® user, searching an NDS tree using the search filter syntax, and so forth.

Acl.java (../../../samplecode/njcl_sample/NDS/Acl.java.html) determines what trustees are already in an object’s ACL, and allows users to add and delete trustees from the object’s ACL.

Createuser.java (../../../samplecode/njcl_sample/NDS/Createuser.java.html) creates a user object in an NDS tree with nameand surname subordinate to the URL (provider_url].

NDSSearch.java (../../../samplecode/njcl_sample/NDS/NdsSearch.java.html) allows users to search an NDS tree using the RFC 1960 search filter syntax.

NetWareObjectLister.java (../../../samplecode/njcl_sample/General/NetWareObjectLister.java.html) provides a section of code describing how to get and output various lists of NDS objects, attributes and trees through JNDI.

6.5 NetWare File System Samples

These samples demonstrate how to perform various manipulations of files in the NetWare File System, such as creating and deleting files, accessing file streams, accessing file attributes, and so forth.

Bind.java (../../../samplecode/njcl_sample/File/Bind.java.html) demonstrates how to explicitly bind foreign (non-file system) objects to the file system provider’s contexts.

DirectoryAttrList.java (../../../samplecode/njcl_sample/File/DirectoryAttrList.java.html) demonstrates using the dynamic and static attribute interfaces of a directory DirContext.

DirectoryCreator.java (../../../samplecode/njcl_sample/File/DirectoryCreator.java.html) demonstrates how to create a new directory in an existing directory DirContext.

DirectoryDeleter.java (../../../samplecode/njcl_sample/File/DirectoryDeleter.java.html) demonstrates how to delete a directory that is subordinate to an existing directory DirContext.

DynamicAttributeValueInterface.java (../../../samplecode/njcl_sample/File/DynamicAttributeValueInterface.java.html) demonstrates how to declare methods for handling the various attribute values in the dynamic attribute value interfaces.

ExtendedAttribute.java (../../../samplecode/njcl_sample/File/ExtendedAttribute.java.html) demonstrates how to create and read an extended attribute on a file or directory DirContext.

FileAttrList.java (../../../samplecode/njcl_sample/File/FileAttrList.java.html)demonstrates how to use the dynamic and static attribute interfaces of a file DirContext.

FileCreator.java (../../../samplecode/njcl_sample/File/FileCreator.java.html) demonstrates how to create a new file subordinate to an existing directory DirContext.

FileDeleter.java (../../../samplecode/njcl_sample/File/FileDeleter.java.html) demonstrates how to delete a file that is subordinate to an existing directory DirContext.

FileList.java (../../../samplecode/njcl_sample/File/FileList.java.html) demonstrates how to displays a lists of the subordinate object names bound to a given directory or volume Context.

GenericAttrList.java (../../../samplecode/njcl_sample/File/GenericAttrList.java.html) demonstrates how to displays a lists of the Attribute IDs and their associated value types for a generic JNDI DirContext.

GenericObjectList.java (../../../samplecode/njcl_sample/File/GenericObjectList.java.html) demonstrates how to displays a lists of the object names bound to a specified generic JNDI DirContext.

Schema.java (../../../samplecode/njcl_sample/File/Schema.java.html) demonstrates how to discover schema information about a specified file system name space's DirContext.

Search.java (../../../samplecode/njcl_sample/File/Search.java.html) demonstrates how to search on attribute values.

StaticAttributeValueInterface.java (../../../samplecode/njcl_sample/File/StaticAttributeValueInterface.java.html) demonstrates how to use a handler for NFile types of objects.

Streams.java (../../../samplecode/njcl_sample/File/Streams.java.html) demonstrates how to do input stream, output stream and random access to an existing file.

Textual.java (../../../samplecode/njcl_sample/File/Textual.java.html) demonstrates how to do a textual display of the various attribute values.

TextualDynamic.java (../../../samplecode/njcl_sample/File/TextualDynamic.java.html) demonstrates how to utilize the dynamic attribute value interfaces by declaring methods for handling the various attribute values.

TextualStatic.java (../../../samplecode/njcl_sample/File/TextualStatic.java.html) demonstrates how to utilize the static attribute value interfaces by declaring methods for handling the various attribute values.

UnBind.java (../../../samplecode/njcl_sample/File/UnBind.java.html) demonstrates how to explicitly unbind a foreign object from the file system provider's Contexts.

VolumeAttrList.java (../../../samplecode/njcl_sample/File/VolumeAttrList.java.html) demonstrates using the Dynamic and Static attribute interfaces of a directory DirContext.

6.6 NetWare (General) Samples

These sample some general NetWare functionality.

DumpException.java (../../../samplecode/njcl_sample/General/DumpException.java.html) Takes a Throwable and dumps its exception trace in order to determine the root cause.

MyEnvironment.java (../../../samplecode/njcl_sample/General/MyEnvironment.java.html) demonstrates how to set the variables that reflect your network environment.

NetWareObjectLister.java (../../../samplecode/njcl_sample/General/NetWareObjectLister.java.html) demonstrates programmatic access to NetWare (Bindery and Server) and NDS through JNDI.

6.7 Queue Samples

These samples demonstrates how to perform queue management functions.

QueueCreateDelete.java (../../../samplecode/njcl_sample/Queue/QueueCreateDelete.java.html) Uses JNDI to create and delete a queue.

QueueJob.java (../../../samplecode/njcl_sample/Queue/QueueJob.java.html) reads the specified file, creates a job in the specified queue, and sends it to a print queue on the specified server.

6.8 Server Samples

This sample demonstrate how to perform manipulations of server objects.

NetWareObjectLister.java (../../../samplecode/njcl_sample/General/NetWareObjectLister.java.html) provides a section of code describing how to list all the servers on the network and the attributes of a particular server.

6.9 Session Samples

These samples demonstrate the use of session functionality.

CreateAndListSessions.java (../../../samplecode/njcl_sample/Session/CreateAndListSessions.java.html) creates a session and searches for all current sessions.

LoginSession.java (../../../samplecode/njcl_sample/Session/LoginSession.java.html) finds a context and logs in a session associated with the context.

MultiUserSessionTest.java (../../../samplecode/njcl_sample/Session/MultiUserSessionTest.java.html) demonstrates the capability of multiple separate sessions to the same domain.

NdsServerSessionInfo.java (../../../samplecode/njcl_sample/Session/NdsServerSessionInfo.java.html) demonstrates access to explicitly defined NDS session attributes or all NDS session attributes.

Revision History

A

The following table outlines all the changes that have been made to the Classes for Java and JNDI Providers documentation (in reverse chronological order) since May 1999.

NDK Release	Description of Changes
March 1, 2006	Updated links to sample code and fixed links in Javadoc.
October 5, 2005	Transitioned to revised Novell documentation standards.
March 2, 2005	Added the About This Guide section. Updated the links to the Java site for the Java Technology Products Download Archive (http://java.sun.com/products/archive/index.html).
October 6, 2004	For legal reasons, removed references to the following samples: SearchTreeNode, NameClassCompare, AttributeCompare, AttributeFrame, and ContextTreeNode.
June 9, 2004	Added dependency on NLM and NetWare Libraries for C to the Preface. Fixed link to the SendNCPExtReq.java sample from "NCP Extensions Samples" on page 136.
February 18, 2004	Removed sections that were only place holders for additional information.
October 8, 2003	Updated templates and table sizes. Doesn't need to be rebuilt for this.
March 2003	Combined documentation for both the Clientless and Client version into one document.
September 2002	Changed "NDSObject" references to "NdsObject" in "NDS Provider Components" on page 55.
February 2002	Changed book title from "Novell Class Libraries for Java" to "Classes for Java and JNDI Providers (NJCL)". Updated documentation to reflect this change. Added initial context table to "Initial Context" on page 22. Reorganized "JNDI Object Naming" on page 26 and added information about the relationship between object naming and the initial context used. Added "File System Namespace" on page 29. Added information about using NAS login to "Authentication Services" on page 41.
September 2001	Made changes to improve document accessibility and added text alternatives to figures.
February 2001	Added information about the "Remote Session Manager" on page 96. Added information about the clientless version of NJCL in "About This Guide" on page 11.
July 2000	Fixed several broken link problems in Javadoc API Reference Guide and in Developer's Guide. Added and corrected Javadoc comments in two NetWare File System classes: DirectorySpaceInformation (../api/com/novell/service/file/nw/DirectorySpaceInformation.html) and VolumeRestriction (../api/com/novell/service/file/nw/VolumeRestriction.html).

NDK Release	Description of Changes
May 2000	Fixed several broken links in Javadoc API Reference Guide. Corrected several dead-end links in Tasks chapter of Developer's Guide.
March 2000	Added an "NJCL and JNDI Solutions" on page 101 chapter to the Developer's Guide documentation by taking existing documentation on four demo utilities from the "Sample Source Code" on page 135 chapter, and then added a fifth section consisting of links to a Developer Net University tutorial. Added two new packages to Javadoc API documentation: NDS Naming Package (../api/com/novell/service/nds/naming/package-summary.html) and NetWare File System Naming Package (../api/com/novell/service/file/nw/naming/package-summary.html). Revised the QMS Naming Package (../api/com/novell/service/qms/naming/package-summary.html).
January 2000	Made some revisions to the "NJCL and JNDI Tasks" on page 127 chapter. Made corrections in links to Javadoc API documentation that were broken as a result of converting from Javadoc v1.1 to v1.2. Fixed numerous links in the Javadoc API documentation that were broken as a result of the conversion from Javadoc v1.1 to v1.2.
November 1999	Revised the "NetWare File System Provider" on page 68 section in the Novell Services chapter of the Developer's Guide. Added 11 new graphics to the Novell Services chapter. Edited all Javadoc comments for the NetWare File System package (../api/com/novell/service/file/nw/package-summary.html) and Novell Java IO package (../api/com/novell/java/io/package-summary.html) files. Converted Javadoc from v1.1 to v1.2 resulting in numerous broken links in both Javadoc comments and Developer's Guide documentation. Manually fixed as many broken links as able to find.
September 1999	Revised the "NDAP Provider for Novell Directory Services (NDS)" on page 55 section in the Novell Services chapter of the Developer's Guide. Edited Javadoc comments in all files of the services.nw, services.ncpext, services.server, services.bindery and services.ldap packages.
July 1999	Revised the Authentication, Bindery, LDAP, NetWare, NCPExtensions, QMS, Server, and Session Manager sections in the Novell Services chapter of the Developer's Guide.
June 1999	Reorganized navigation links for the entire Developer's Guide documentation. Rewrote the preface and overview chapters. Reorganized and rewrote much of the Concepts chapter (previously the Architecture and Glossary chapters), and added 8 new graphics. Reorganized and revised the "NJCL and JNDI Tasks" on page 127 chapter. Reorganized the links to the sample source code and the sample demo utilities.