Sun Microsystems, Inc.

Java™ Naming and Directory Interface™ Application Programming Interface (JNDI API)

Please send technical comments to jndi@java.sun.com.

Please send product and business questions to jndi-business@java.sun.com.

JNDI 1.2

July 14, 1999

Contents

1 Introduction

Directory services play a vital role in Intranets and Internets by providing access to a variety of information about users, machines, networks, services, and applications. By its very nature, a directory service incorporates a naming facility for providing human understandable namespaces that characterize the arrangement and identification of the various entities.

The computing environment of an enterprise typically consists of several naming facilities often representing different parts of a *composite* namespace. For example, the Internet Domain Name System (DNS) might be used as the top-level naming facility for different organizations within an enterprise. The organizations themselves might use a directory service such as LDAP or NDS or NIS. From a user's perspective, there is one namespace consisting of composite names. URLs are examples of composite names because they span namespaces of multiple naming facilities. Applications which use directory services must support this user perspective.

Many Java^{TM} application developers can benefit from a directory service API that is not only independent of the particular directory or naming service implementation, but also enables seamless access to directory objects through multiple naming facilities. In fact, any application can attach its own objects to the namespace. Such a facility enables any Java application to discover and retrieve objects of any type.

End users can benefit from logical namespaces that allow easier discovery and identification of the objects in the network.

Directory service developers can benefit from a service-provider capability that enables them to incorporate their respective implementations without requiring changes to the client.

Java Naming and Directory Interface^{TM} (JNDI) is an API that provides directory and naming functionality to Java applications. It is defined to be independent of any specific directory service implementation. Thus, a variety of directories can be accessed in a common way.

Here are two examples to briefly illustrate some of the more commonly used features of JNDI.

An application that wants to access a printer needs the corresponding printer object. This is simply done as follows:

```
prt = (Printer) building7.lookup("puffin");
prt.print(document);
```

JNDI does all the work of locating the information needed to construct the printer object.

An application that wants to find a person's phone numbers, which are stored in the organization's directory, can simply do:

```
String[] attrs = {"workPhone", "cellPhone", "faxNumber"};
bobsPhones = directory.getAttributes("cn=Bob, o=Widget, c=US", attrs);
```

If there may be several Bobs in the Widget organization, the application can search the organization's directory to find the right Bob as follows:

```
bob = directory.search("o=Widget, c=US", "(cn=Bob)", searchctls);
```

This document describes the architecture and interfaces of JNDI.

2 Goals and Design Principles

We followed several principles and maxims in designing the API.

2.1 Keep it consistent and intuitive

Wherever possible, we have used existing components from the rest of the Java development environment. Adhering to this principle not only makes JNDI consistent with existing core classes in the Java platform but also reduces needless proliferation of classes.

The object-oriented nature of the Java programming language allows for an intuitive and simple API design, in which the directory service functionality is expressed as a natural extension to the more fundamental naming service functionality.

2.2 Pay for what you use

The API is structured in a tiered manner so that the application programmer interested in a certain directory service capability need not necessarily know about a more advanced capability. We have strived to keep the lower tiers simple and also make them represent the common case capability, relegating the more complex ones to the upper tiers.

2.3 Implementable over common directory and naming services and protocols

This goal is important for two reasons. First, it enables Java applications to take advantage of information in a variety of existing naming and directory services such as DNS, NDS, NIS (YP), X.500, and LDAP. Second, it helps limit the appearance of any implementation specific artifacts in the API.

Providing a unified interface to multiple naming and directory services does not imply that access of unique features of a particular service is precluded. The unified API which is designed to cover the common case is still beneficial to applications that have explicit knowledge of the underlying naming or directory service. Such applications still benefit from sharing the common portions that use the API. This is analogous to applications sharing commonly used classes and yet adding needed specificity via subclassing.

2.4 Seamless integration

This is important not only because of the diversity of directory service and naming services in the installed base that need to be supported, but also because new Java application and service programmers can export their own namespaces and directory objects in a uniform way.

We also wanted to make a variety of implementation choices possible without having the application pay for this freedom. For example, a “thin-client” might be better served by a proxy-style protocol in which the access to specific naming and directory services is relegated to a server. Whereas, a performance sensitive, resource rich client, might choose to use an implementation which directly allows it to access the various servers. However, the application should be insulated from these implementation choices. It should be possible to defer such choices even until runtime.

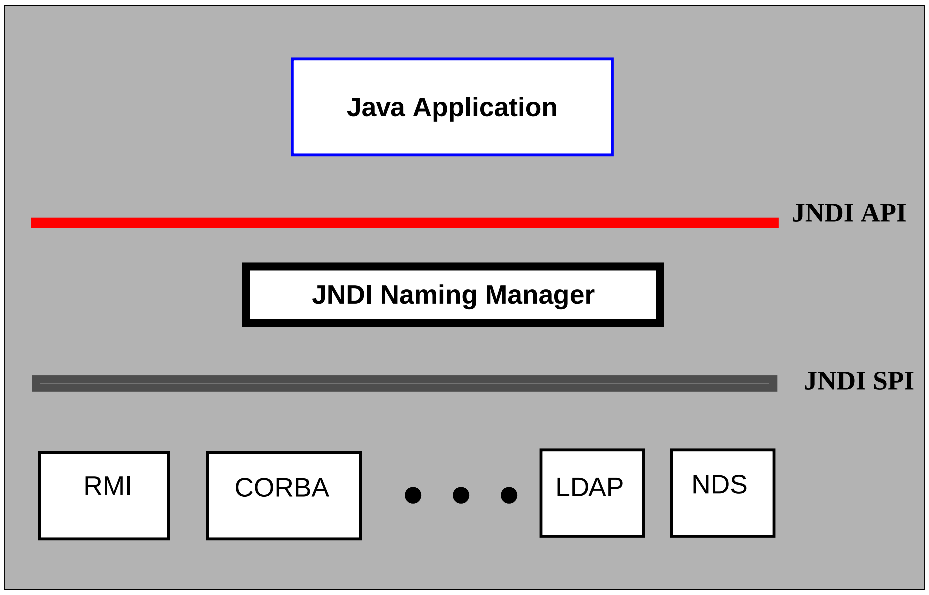
2.5 Support for leading industry standards

The Lightweight Directory Access Protocol (Internet RFC 2251) has emerged as the standard for directory access at the protocol level. All major directory vendors have products that support this protocol. An application that uses JNDI should be able to access all of the features offered by this standard. Where possible, JNDI should support conventions (such as those for specifying search queries/filters) already defined by the standard.

3 Overview of the Architecture

The JNDI architecture consists of the JNDI API and the JNDI SPI. The JNDI API allows Java applications to access a variety of naming and directory services. The JNDI SPI is designed to be used by arbitrary service providers including directory service providers. This enables a variety of directory and naming services to be plugged in transparently to the Java application (which uses only the JNDI API). Figure 1 shows the JNDI architecture and includes a few service providers of directory and naming contexts as examples.

Figure 1: JNDI Architecture



4 Fundamentals

A directory service provides access to diverse kinds of information about users and resources in a network environment. It uses a *naming system* for the purpose of identifying and organizing *directory objects* to represent this information. A directory object provides an association between *attributes* and *values*. Thus, a directory service enables information to be organized in a hierarchical manner to provide a mapping between human understandable names and directory objects.

4.1 Naming — The Foundation

A fundamental facility in any computing system is the naming service – the means by which names are associated with objects, and by which objects are found given their names. In traditional systems, the naming service is seldom a separate service. It is usually integrated with another service, such as a file system, directory service, database, desktop, mail system, spreadsheet, or calendar. For example, a file system includes a naming service for files and directories; a spreadsheet has a naming service for cells and macros.

The computing environment of an enterprise typically consists of several naming services. There are naming services that provide contexts for naming common entities in an enterprise such as organizations, physical sites, human users and computers. Naming services are also incorporated in applications offering services such as file service, mail service, printer service, and so on. From a user's perspective, there exist several natural and logical relationships between these naming services. For example, it is natural to think of naming a variety of services such as files, mail, appointment calendar, and so on, in the context of a user. It is also natural to think of a user in the context of a department, within a division of an enterprise. Meaningful names can be composed using useful arrangements of naming services reflecting these relationships.

Every *name* is generated by a set of syntactic rules called a *naming convention*. An *atomic name* is an indivisible component of a name, as defined by the naming convention.

A *compound name* represents a sequence of zero or more atomic names composed according to the naming convention.

For example, in UNIX pathnames, atomic names are ordered from left to right, and are delimited by slash ('/') characters. The UNIX pathname `usr/local/bin` is a compound name representing the sequence of atomic names, `usr`, `local`, and `bin`. In names from the Internet Domain Name System (DNS), atomic names are ordered from right to left, and are delimited by dot ('.') characters. Thus, the DNS name `sales.Wiz.COM` is a compound name representing the sequence of atomic names, `COM`, `Wiz`, `sales`.

The association of an atomic name with an object is called a *binding*.

A *context* is an object whose state is a set of bindings with distinct atomic names. Every context has an associated naming convention. A context provides a lookup (resolution) operation that returns an object, and may provide operations such as for binding names, unbinding names, listing bound names. An atomic name in one context object can be bound to another context object of the same type, called a *subcontext*, giving rise to compound names.

Resolution of compound names proceeds by looking up each successive atomic component in each successive context. The reader will find a familiar model in UNIX file naming, where directories serve as contexts, and pathnames may be compound names.

A *naming system* is a connected set of contexts of the same type (having the same naming convention) and providing the same set of operations with identical semantics.

A *namespace* is the set of all names in a naming system.

A *composite name* is a name that spans multiple naming systems. It consists of an ordered list of zero or more components. Each component is a name from the namespace of a single naming system.

For example, the name `jurassic.eng:/export/home/jdoe/.signature` is a composite name representation made up of a host name `jurassic.eng` from a host namespace, and the file name `/export/home/jdoe/.signature` from a UNIX file namespace. Another example is the Internet URL `http://www.moon.org/public/index.html,` which is a composite name representation made up of the scheme-id `http` from the "URL scheme-id" namespace, `www.moon.org` which is the DNS name of the machine on which the web server is running, and `public/index.html` which is a file name from a file namespace.

Every name is interpreted relative to some context, and every naming operation is performed on a context object. A client can obtain an *initial context* object that provides a starting point for resolution of names.

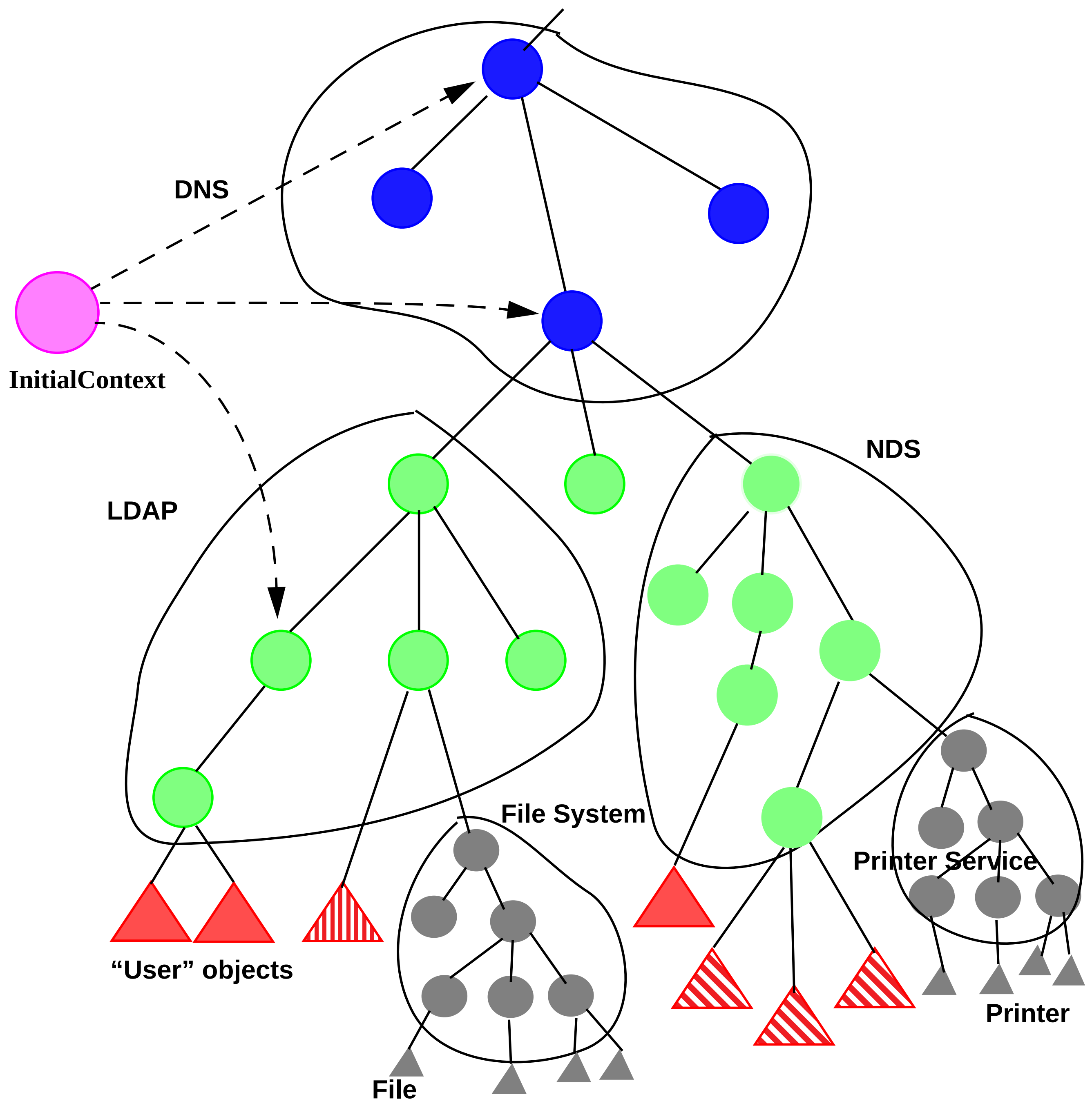
4.2 Directory Objects

The primary function of a naming system is to map names to objects. The objects can be of any type. A *directory object* is a particular type of object that is used to represent the variety of information in a computing environment. A directory object can have associated with it *attributes*. An attribute has an identifier and a set of values.

A directory object provides operations for creating attributes, adding, removing, and modifying attributes associated with the directory object. If we make a directory object also be a naming context, we can represent trees of directory information where the interior nodes not only behave like naming contexts but also contain attributes.

Figure 2 is an example used for illustrating several things.

Figure 2: Example of a Composite Namespace



- ‘There can be multiple naming systems that can be represented by a composite namespace. In this case, DNS is used as the global naming system; one division uses NDS, while a second division uses LDAP.
- Each namespace has interior nodes that represent naming contexts, which may be directory objects as well. Leaf nodes can be objects of any type.
- The *InitialContext* is configured to have bindings to useful starting contexts in different naming and directory systems.
- Applications just see a composite namespace. They can access any type of object bound in any naming system in this arrangement.
- Services can incorporate their own namespaces which appear as first-class citizens in JNDI.
- Arbitrary directory services can be added and accessed without requiring client applications to be changed.

4.3 URLs and Composite Names

Universal Resource Locators (URLs) are special composite names whose syntax are determined by the URL’s definition. Clients of JNDI can use URLs to refer to arbitrary types of objects. For example, a client can use `nfs://nfs.sun.com/export/jndi/src/README` to refer to a file object that is being accessed using the Network File System (NFS) protocol. Similarly, a client can perform directory operations on a directory object in an LDAP server using the URL `ldap://ldap.widget.com/cn=Jonathan,ou=marketing`.

To support composite names in general, JNDI defines a composite name syntax and utilities for processing composite names. This allows clients of JNDI to refer to objects using names that span multiple namespaces.

4.4 Events

As the naming/directory service plays an increasingly important role in the computing environment, the need to provide administration and monitoring tools to help manage changes in the service also increases. For such tools and other applications, the traditional request/response style of interaction needs to be augmented with an asynchronous notification model that allows applications to register interest in changes in the service.

5 Overview of the Interface

The JNDI API is contained in four packages:

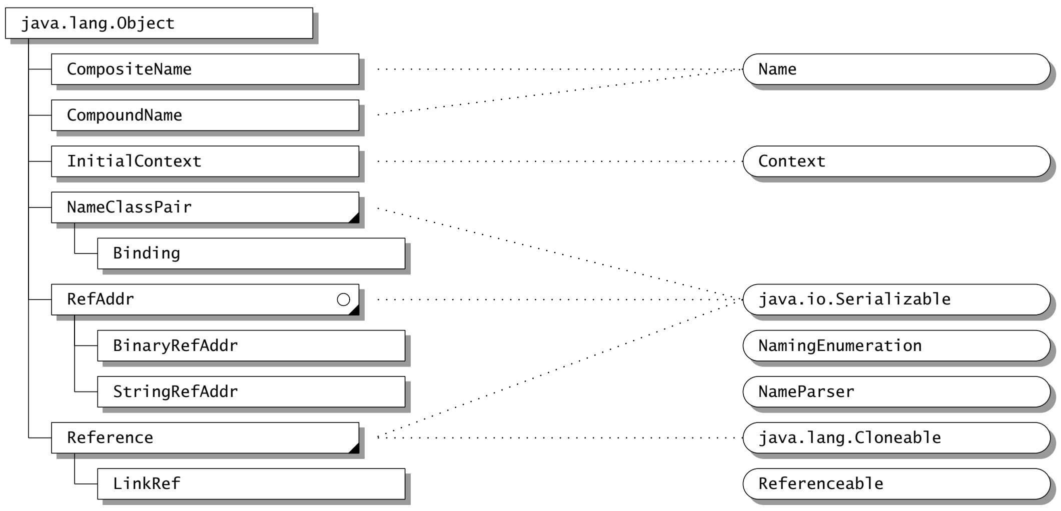
- `javax.naming` contains classes and interfaces for accessing naming services
- `javax.naming.directory` extends the core `javax.naming` package to provide access to directories
- `javax.naming.event` contains classes and interfaces for supporting event notification in naming and directory services
- `javax.naming.ldap` contains classes and interfaces for supporting LDAP v3 extensions and controls

The JNDI service provider interface is contained one package:

- `javax.naming.spi` contains classes and interfaces that allow various naming and directory service providers to be dynamically plugged in beneath the JNDI API (see the **JNDI SPI** document for details)

The following sections provide an overview of the JNDI API. For more details on the API, see the corresponding **javadoc**.

5.1 The Naming Package — javax.naming [1]



(exception classes are not shown)

5.1.1 Contexts

`Context` is the core interface that specifies a naming context. It defines basic operations such as adding a name-to-object binding, looking up the object bound to a specified name, listing

1. See Appendix C for legend of class diagram.

the bindings, removing a name-to-object binding, creating and destroying subcontexts of the same type, *etc*.

```
public interface Context {
    public Object lookup(Name name) throws NamingException;
    public void bind(Name name, Object obj) throws NamingException;
    public void rebind(Name name, Object obj) throws NamingException;
    public void unbind(Name name) throws NamingException;
    public void rename(Name old, Name new) throws NamingException;
    public NamingEnumeration listBindings(Name name)
        throws NamingException;
    ...
    public Context createSubcontext(Name name) throws NamingException;
    public void destroySubcontext(Name name) throws NamingException;
    ...
};
```

Every naming method in `Context` takes a name as an argument. The operation defined by the method is performed on the `Context` object that is obtained by implicitly resolving the name. If the name is empty (“”) the operation is performed directly on the context itself. The name of an object can be a composite name reflecting the arrangement of the namespaces used to refer to the object. Of course, the client is not exposed to any naming service implementation. In fact, a new type of naming service can be introduced without requiring the application to be modified or even disrupted if it is running.

5.1.2 The Initial Context

In JNDI, every name is relative to a context. There is no notion of “absolute names.” An application can bootstrap by obtaining its first context of class `InitialContext`:

```
public class InitialContext implements Context {
    public InitialContext()...;
    ...
}
```

The initial context contains a variety of bindings that hook up the client to useful and shared contexts from one or more naming systems, such as the namespace of URLs or the root of DNS.

5.1.3 Names

The `Name` interface represents a generic name—an ordered sequence of components. Each `Context` method that takes a `Name` argument has a counterpart that takes the name as a `String` instead. The versions using `Name` are useful for applications that need to manipulate names: composing them, comparing components, and so on. The versions using `String` are likely to be more useful for simple applications, such as those that simply read in a name and look up the corresponding object. The `String` name parameter represents a composite name. The `Name` parameter can represent a *composite name* or a *compound name*.

The `CompositeName` class represents a sequence of names (atomic or compound) from multiple namespaces. If the `Name` parameter supplied to a method of the `Context` class is an instance of `CompositeName`, the name represents a composite name.

If the `Name` parameter supplied to a method of the `Context` class is *not* an instance of `CompositeName`, the name represents a compound name, which can be represented by the `Compound-`

`Name` class or some other implementation class. The `CompoundName` class represents hierarchical names from a single namespace. A context's name parser can be used to manipulate compound names in the syntax associated with that particular context:

```
public interface Context {
    ...
    public NameParser getNameParser(Name name) throws NamingException;
    ...
}
```

A namespace browser is an example of the kind of application that might need to manipulate names syntactically at this level. Most other applications will work with strings or composite names.

5.1.4 Bindings

`Context.lookup()` is the most commonly used operation. The context implementation can return an object of whatever class is required by the Java application. For example, a client might use the name of a printer to look up the corresponding `Printer` object, and then print to it directly:

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

`Context.listBindings()` returns an enumeration of name-to-object bindings, each binding represented by an object of class `Binding`. A binding is a tuple containing the name of the bound object, the name of the object's class, and the object itself.

The `Context.list()` method is similar to `listBindings()`, except that it returns an enumeration of `NameClassPair` objects. Each `NameClassPair` contains an object's name and the name of the object's class. The `list()` method is useful for applications such as browsers that wish to discover information about the objects bound within a context, but don't need all of the actual objects. Although `listBindings()` provides all of the same information, it is potentially a much more expensive operation.

```
public class NameClassPair ... {
    public String getName() ...;
    public String getClassName() ...;
    ...
}

public class Binding extends NameClassPair {
    public Object getObject() ...;
    ...
}
```

5.1.5 References

Different `Context` implementations are able to bind different kinds of objects natively. A particularly useful object that should be supported by any general-purpose context implementation is the `Reference` class. A reference represents an object that exists outside of the directory. References are used to give JNDI clients the illusion that objects of arbitrary classes are able to be bound in naming or directory services—such as X.500—that do not have native support for objects in the Java programming language.

When the result of an operation such as `Context.lookup()` or `Binding.getObject()` is a `Reference` object, JNDI attempts to convert the reference into the object that it represents before returning it to the client. A particularly significant instance of this occurs when a reference representing a `Context` of one naming system is bound to a name in a different naming system. This is how multiple independent naming systems are joined together into the JNDI composite namespace. Details of how this mechanism operates are provided in the **JNDI SPI** document.

Objects that are able to be represented by a reference should implement the `Referenceable` interface. Its single method — `getReference()` — returns the object's reference. When such an object is bound to a name in any context, the context implementation might store the reference in the underlying system if the object itself cannot be stored natively.

Each reference may contain the name of the class of the object that it represents, and may also contain the location (typically a URL) where the class file for that object can be found. In addition, a reference contains a sequence of objects of class `RefAddr`. Each `RefAddr` in turn contains a "type" string and some addressing data, generally a string or a byte array.

A specialization of `Reference` called a `LinkRef` is used to add "symbolic" links into the JNDI namespace. It contains the name of a JNDI object. By default, these links are followed whenever JNDI names are resolved.

5.1.6 Referrals

Some naming/directory services support the notion of *referrals* for redirecting a client's request to another server. The JNDI client can request that referrals be automatically followed, be ignored, or be processed manually.

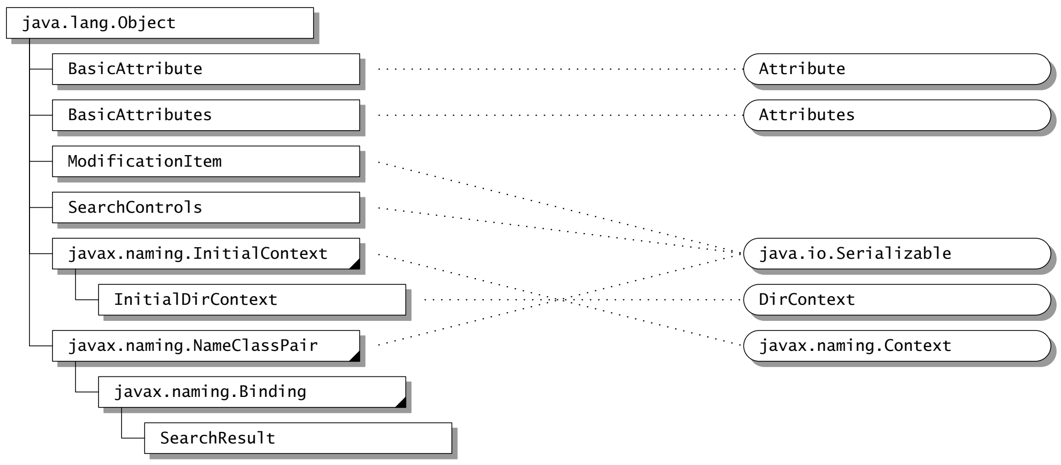
The abstract class `ReferralException` is used to represent a referral:

```
public abstract class ReferralException extends NamingException {
    public abstract Context getReferralContext()
        throws NamingException;
    ...
    public abstract Object getReferralInfo();
    public abstract void retryReferral();
    public abstract boolean skipReferral();
}
```

When a referral is encountered and the client has requested that referrals not be ignored or automatically followed, a `ReferralException` is thrown. The `getReferralInfo()` method provides information—in a format appropriate to the service provider—about where the referral leads. The application is not required to examine this information; however, it might choose to present it to a human user to help him determine whether to follow the referral or not. `skipReferral()` allows the application to discard a referral and continue to the next referral (if any).

To continue the operation, the application re-invokes the method on the referral context using the same arguments it supplied to the original method.

5.2 The Directory Package — javax.naming.directory[1]



(exception classes are not shown)

5.2.1 Directory Objects

The `DirContext` interface enables the directory capability by defining methods for examining and updating attributes associated with a directory object.

```
public interface DirContext extends Context {
    public Attributes getAttributes(Name name)
            throws NamingException;
    public Attributes getAttributes(Name name, String[] attrIds)
            throws NamingException;
    ...
    public void modifyAttributes(Name name,
                                 int modOp,
                                 Attributes attrs)
            throws NamingException;
    public void modifyAttributes(Name name,
                                 ModificationItem[] mods)
            throws NamingException;
    ...
}
```

The `getAttributes()` operations on a directory return some or all of its attributes. Attributes are modified using two forms of `modifyAttributes()`. Both forms make use of a “modification operation,” one of:

```
ADD_ATTRIBUTE
REPLACE_ATTRIBUTE
REMOVE_ATTRIBUTE
```

The `ADD_ATTRIBUTE` operation adds values to an attribute if that attribute already exists, while the `REPLACE_ATTRIBUTE` operation discards any pre-existing values. The first form of `modify-`

1. See Appendix C for legend of class diagram.

`Attributes()` performs the specified operation on each element of a set of attributes. The second form takes an array of objects of class `ModificationItem`:

```
public class ModificationItem {
    public ModificationItem(int modOp, Attribute attr) ...;
    ...
}
```

Each operation is performed on its corresponding attribute in the order specified. When possible, a context implementation should perform each call to `modifyAttributes()` as an atomic operation.

5.2.2 Attributes

A directory object contains a set of zero or more `Attribute` objects. Each attribute is denoted by a string identifier and can have zero or more values of any type.

```
public interface Attribute ... {
    ...
    public String getID();
    public Object get(int n) throws NamingException;
    public boolean isOrdered();
    public NamingEnumeration getAll()
            throws NamingException;
    ...
}
```

An attribute's values can be ordered or unordered. If the values are unordered, no duplicates are allowed. If the values are ordered, duplicates are allowed.

Attributes are grouped into a collection by using the `Attributes` interface.

```
public interface Attributes ... {
    ...
    public Attribute get(String attrID);
    public NamingEnumeration getIDs();
    public NamingEnumeration getAll();
    public Attribute put(Attribute attr);
    public Attribute remove(String attrID);
    ...
}
```

JNDI provides implementations for these two interfaces, `BasicAttribute` and `BasicAttributes`, for convenience. Service providers and applications are free to use their own implementations.

Note that updates to `Attributes` and `Attribute`, such as adding or removing an attribute or its value, do not affect the corresponding representation in the directory. Updates to the directory can only be effected by using `DirContext.modifyAttributes()`.

5.2.3 Directory Objects as Naming Contexts

The `DirContext` interface also behaves as a naming context by extending the `Context` interface. This means that any directory object can also provide a naming context. In addition to a directory object keeping a variety of information about a person, for example, it is also a natural naming context for resources associated with that person: a person's printers, file system, calendar, *etc.*

Hybrid operations perform certain naming and directory operations in a single atomic operation:

```
public interface DirContext extends Context {
    ...
    public void bind(Name name, Object obj, Attributes attrs)
            throws NamingException;
    ...
}
```

Other hybrid operations that are provided are `rebind()` and `createSubcontext()` that accept an additional `Attributes` argument.

5.2.4 The Initial Context

An application that is performing directory operations can use `InitialDirContext` instead of `javax.naming.InitialContext` to create its initial context:

```
public class InitialDirContext
                extends InitialContext implements DirContext {
    public InitialDirContext() ...;
    ...
}
```

It can then invoke any method in the `Context` or `DirContext` interface on the initial context.

5.2.5 Searches

The `DirContext` interface supports content-based searching of directories. In the simplest and most common form of usage, the application specifies a set of attributes — possibly with specific values — to match. It then invokes the `DirContext.search()` method on the directory object, which returns the matching directory objects along with the requested attributes.

```
public interface DirContext extends Context {
    ...
    public NamingEnumeration search(Name name,
                                Attributes matchingAttributes)
            throws NamingException;

    public NamingEnumeration search(Name name,
                               Attributes matchingAttributes,
                               String[] attributesToReturn)
            throws NamingException;
    ...
}
```

The results of the search are returned as a `NamingEnumeration` containing an enumeration of objects of class `SearchResult`:

```
public class SearchResult extends Binding {
    ...
    public Attributes getAttributes() ...;
}
```

In the more sophisticated case, it is possible to specify a search filter and to provide controlling information such as the scope of the search and the maximum size of the results. The search filter specifies a syntax that follows Internet RFC 2254 for LDAP. The `SearchControls` argument specifies such things as the scope of the search: this can include a single directory object, all of its children, or all of its descendants in the directory hierarchy.

```
public interface DirContext extends Context {
    ...
    public NamingEnumeration search(Name name,
                                    String filter,
                                    SearchControls ctls)
            throws NamingException;

    public NamingEnumeration search(Name name,
                                    String filter,
                                    Object[] filterArgs,
                                    SearchControls ctls)
            throws NamingException;
    ...
}
```

5.2.6 Schema

A schema describes the rules that define the structure of a namespace and the attributes stored within it. The granularity of the schema can range from a single schema that is associated with the entire namespace, to a per-attribute, fine-grained schema description.

Because schemas can be expressed as an information tree, it is natural to use for this purpose the naming and directory interfaces already defined in JNDI. This is powerful because the schema part of a namespace is accessible to applications in a uniform way. A browser, for example, can access information in the schema tree just as though it were accessing any other directory objects.

Applications can retrieve the schema associated with a directory object when the underlying context implementation provides the appropriate support.

`DirContext.getSchema()` is used to retrieve the root of the schema tree associated with a directory object. The root has children such as “ClassDefinition”, “AttributeDefinition”, and “SyntaxDefinition”, each denoting the kind of definition being described. The schema root and its descendents are objects of type `DirContext`. The `DirContext.getSchemaClassDefinition()` method returns a `DirContext` that contains class descriptions about a particular directory object.

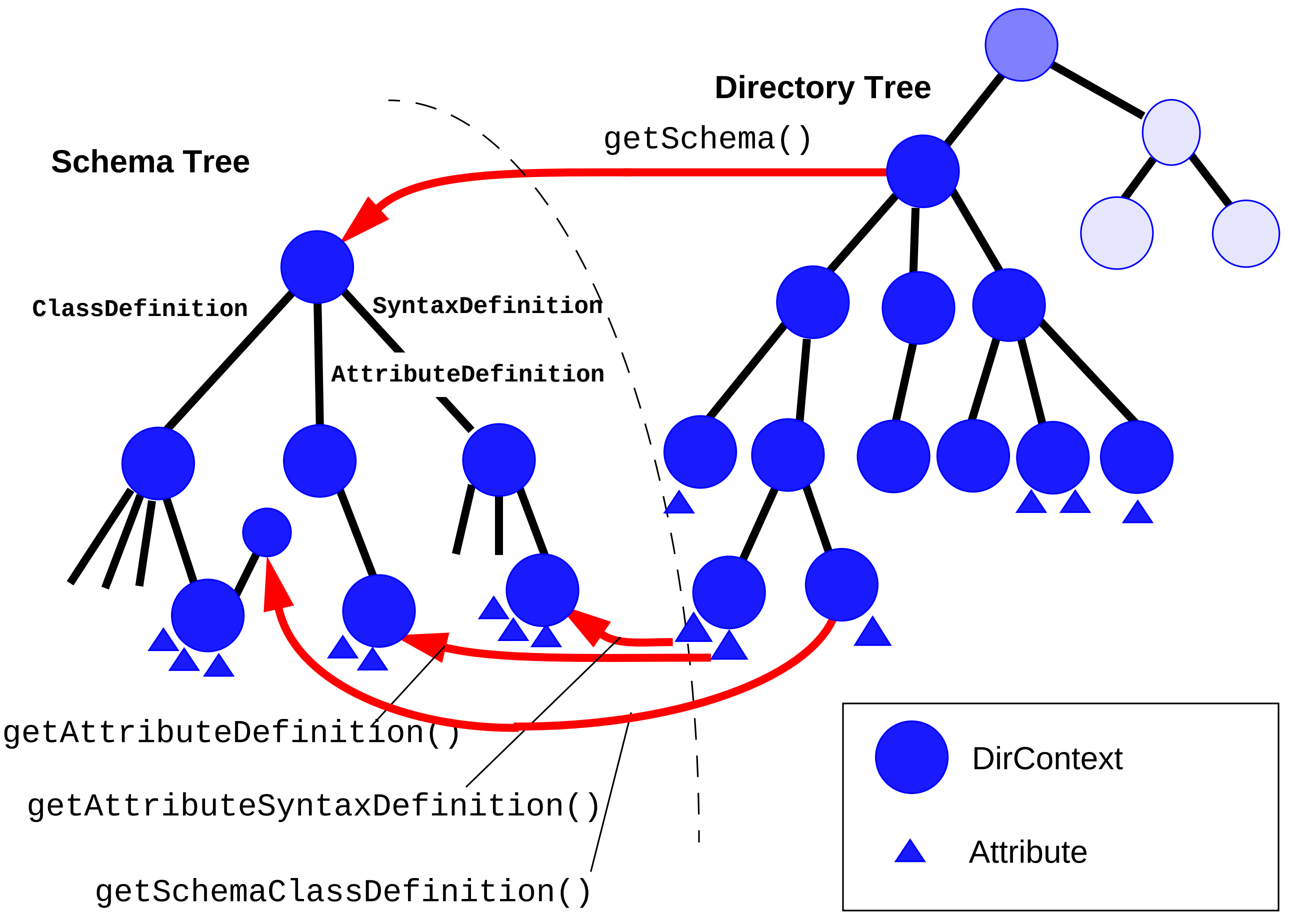
```
public interface DirContext extends Context {
    ...
    public DirContext getSchema(Name name)
        throws NamingException;

    public DirContext getSchemaClassDefinition(Name name)
        throws NamingException;
    ...
}
```

In addition, the schema associated with any attribute can be accessed using the `Attribute.getAttributeDefinition()` and `getAttributeSyntaxDefinition()`methods.

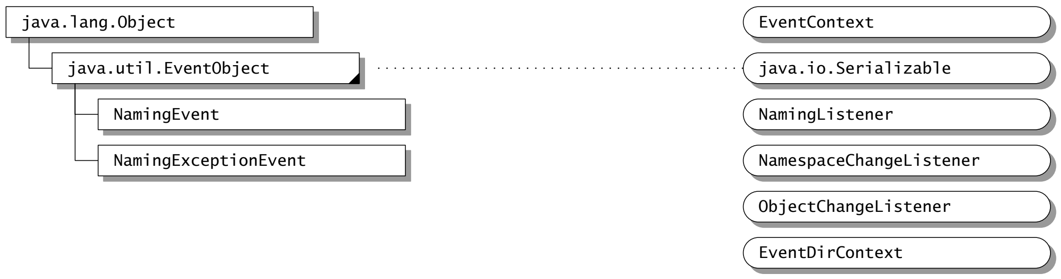
```
public interface Attribute ... {
    ...
    public DirContext getAttributeDefinition() throws NamingException;
    public DirContext getAttributeSyntaxDefinition()
        throws NamingException;
    ...
}
```

Figure 3 is an example showing the different associations for accessing schema information.

Figure 3: Example mapping Directory to Schema



5.3 The Event Package — javax.naming.event[1]



The `javax.naming.event` package contains classes and interfaces for supporting event notification in naming and directory services.

5.3.1 Naming Events

A `NamingEvent` represents an event that is generated by a naming/directory service.

```
public class NamingEvent extends java.util.EventObject {
    ...
    public int getType();
    public Binding getOldBinding();
    public Binding getNewBinding();
    ...
}
```

The event's type identifies the type of event. The `NamingEvent` class defines four types of events:

```
OBJECT_ADDED
OBJECT_REMOVED
OBJECT_RENAMED
OBJECT_CHANGED
```

These types can be placed into two categories:

- Those that affect the namespace (add/remove/rename an object)
- Those that affect an object's contents

In addition to the event's type, a `NamingEvent` contains other information about the change, such as information about the object before and after the change.

5.3.2 Naming Listeners

A *naming listener* is an object that registers for `NamingEvents`. It is represented by the interface `NamingListener`. Each category of `NamingEvent` is handled by a corresponding subtype of `NamingListener`. The `NamespaceChangeListener` interface represents a listener interested in namespace changes, while the `ObjectChangeListener` represents a listener interested in changes to an object's contents. A listener implementation might implement one or both of these interfaces, depending on the types of events it is interested in.

1. See Appendix C for legend of class diagram.

5.3.3 Event Registration and Deregistration

The `EventContext` and `EventDirContext` interfaces extend the `Context` and `DirContext` interfaces, respectively, to support event registration and deregistration.

```
public interface EventContext extends Context {
    ...
    public void addNamingListener(Name target,
                                  int scope,
                                  NamingListener l)
        throws NamingException;
    public void removeNamingListener(NamingListener l)
        throws NamingException;
    public boolean targetMustExist()
        throws NamingException;
}
```

Like methods in the corresponding `Context` interface, `addNamingListener()` has an overload that accepts a `String` name argument. The name parameter is referred to as the *target*. The scope parameter specifies whether the registration is for the object named by the target, the immediate children of the context named by the target, or the entire subtree rooted at the object named by the target.

It is possible to register interest in a target that does not exist, but there might be limitations in the extent to which this can be supported by the service provider and underlying protocol/service. An application can use the method `targetMustExist()` to check whether an `EventContext` supports registration of nonexistent targets.

```
public interface EventDirContext extends EventContext, DirContext {
    public void addNamingListener(Name target,
                                  String filter,
                                  SearchControls ctls,
                                  NamingListener l)
        throws NamingException;
    public void addNamingListener(Name target,
                                  String filter,
                                  Object[] filterArgs,
                                  SearchControls ctls,
                                  NamingListener l)
        throws NamingException;
    ...
}
```

The `EventDirContext` interface extends the `EventContext` and `DirContext` interfaces to allow a listener to register interest in objects identified using search filters (Internet RFC 2254).

Like methods in the corresponding `DirContext` interface, `addNamingListener()` methods have overloads that accept a `String` name argument.

The `EventContext/EventDirContext` instance on which the `addNamingListener()` method is invoked is the *event source* of the events that are (potentially) generated. When the registered listener invokes `getSource()` or `getEventContext()` on a `NamingEvent`, the result will be this `EventContext/EventDirContext` instance.

For example, suppose a listener makes the following registration:

```
NamespaceChangeListener listener = ...;
src.addNamingListener("x", SUBTREE_SCOPE, listener);
```

When an object named “x/y” is subsequently deleted, the corresponding `NamingEvent` (`evt`) delivered to `listener` must contain `src` as its event source. The following will both be true:

```
evt.getEventContext() == src
evt.getOldBinding().getName().equals("x/y")
```

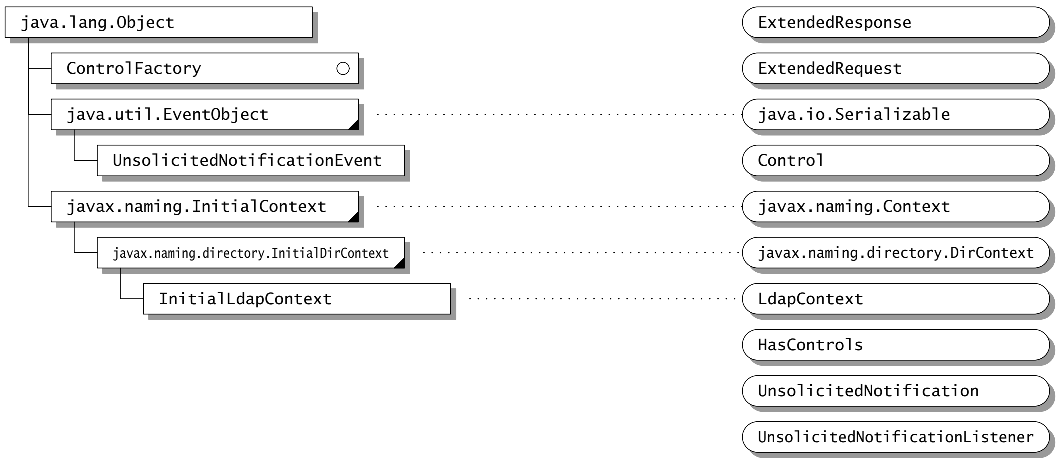
5.3.4 Exception Handling

When a listener registers for events with a context, the context might need to do some internal processing in order to collect information required to generate the events. The context, for example, might need to make a request to the server to register interest in changes on the server that will eventually be translated into events. If an error occurs that prevents information about the events from being collected, the listener will never be notified of the events. When such an error occurs, a `NamingExceptionEvent` is fired to notify the listener, and the listener is automatically deregistered.

The base `NamingListener` interface defines a `namingExceptionThrown()` method so that a listener can be notified of such an error.

```
 public interface NamingListener extends java.util.EventListener {
    public void namingExceptionThrown(NamingExceptionEvent evt);
}
```

5.4 The LDAP Package — javax.naming.ldap[1]



(exception classes are not shown)

The `javax.naming.ldap` package contains classes and interfaces for using LDAP v3-specific features that are not already covered by the more generic `javax.naming.directory` package. In fact, the majority of JNDI applications that use LDAP will find the `javax.naming.directory` package sufficient, and will not need to use this package at all. This package is primarily

1. See Appendix C for legend of class diagram.

for those applications that need to use extended operations, controls, or unsolicited notifications.

5.4.1 Extended Operations

In addition to specifying well-defined operations such as search and modify, the LDAP v3 protocol (Internet RFC 2251) specifies a way of transmitting yet-to-be defined operations between the LDAP client and server. These operations are referred to as *extended operations*. An extended operation may be defined by a standards organization such as the IETF or by a vendor.

The `LdapContext` interface defines a method for executing an extended operation:

```
public interface LdapContext extends DirContext {
    public ExtendedResponse extendedOperation(ExtendedRequest request)
        throws NamingException;
    ...
}
```

The `ExtendedRequest` interface represents the argument to an extended operation, while the `ExtendedResponse` interface represents the result of the extended operation. An `ExtendedRequest` or `ExtendedResponse` consists of an identifier that identifies the extended operation and a byte array containing the ASN.1 BER encoded contents of the request/response.

An application typically does not deal directly with the `ExtendedRequest`/`ExtendedResponse` interfaces. Instead, it deals with classes that implement these interfaces. The application gets these classes either as part of a repertoire of extended operations standardized through the IETF, or from directory vendors for vendor-specific extended operations. The request classes should have constructors that accept arguments in a type-safe and user-friendly manner, while the response classes should have access methods for getting the data of the response in a type-safe and user-friendly manner. Internally, the request/response classes deal with encoding and decoding BER values.

For example, suppose an LDAP server supports a “get time” extended operation. It would supply classes such as `GetTimeRequest` and `GetTimeResponse`, so that applications can use this feature. An application would use these classes as follows:

```
GetTimeResponse resp =
    (GetTimeResponse)lctx.extendedOperation(new GetTimeRequest());
long time = resp.getTime();
```

5.4.2 Controls

The LDAP v3 protocol (Internet RFC 2251) allows any request or response to be augmented by yet-to-be defined modifiers. These modifiers are referred to as *controls*. Controls that are sent with requests are called *request controls* and those that are sent with responses are called *response controls*. A control may be defined by a standards organization such as the IETF or by a vendor. There is not necessarily a pairing between request controls and response controls.

JNDI classifies request controls into two categories:

- *connection* request controls: those that affect how a connection is created
- *context* request controls: those that affect context methods

Connection request controls are used whenever a connection needs to be established or re-established with an LDAP server. Context request controls are used when all other LDAP operations are sent to the LDAP server. The reason for this distinction is because JNDI is a high-level API that does not deal directly with connections. It is the job of service providers to do any necessary connection management. Hence, a single connection might be shared by multiple context instances, and a service provider is free to use its own algorithms to conserve connection and network usage. Thus, when a method is invoked on the context instance, the service provider might need to do some connection management in addition to performing the corresponding LDAP operations. For connection management, it uses the connection request controls, while for the normal LDAP operations, it uses the context request controls.

The `LdapContext` interface defines methods for dealing with controls:

```
public interface LdapContext extends DirContext {
    public void reconnect(Control[] connCtls) throws NamingException;
    public Control[] getConnectControls() throws NamingException;
    ...
    public LdapContext newInstance(Control[] reqCtls)
        throws NamingException;
    public void setRequestControls(Control[] reqCtls)
        throws NamingException;
    public Control[] getRequestControls() throws NamingException;
    ...
    public Control[] getResponseControls() throws NamingException;
}
```

The `Control` interface represents a control. It consists of an identifier that identifies the control and a byte array containing the ASN.1 BER encoded contents of the control.

Connection request controls are initialized using the initial context constructor and are inherited by contexts that are derived from a context. `reconnect()` is used to change the connection request controls of a context. A context's connection request controls are retrieved using `get-ConnectControls()`.

Context request controls are initialized using `newInstance()` and changed using `setRequest-Controls()`. `newInstance()` is a convenience method for creating a new instance of a context for the purposes of multithreaded access. For example, if multiple threads want to use different context request controls, each thread may use this method to get its own copy of this context and set/get context request controls without having to synchronize with other threads.

Unlike connection request controls, context request controls are *not* inherited by context instances that are derived from a context. Derived context instances are initialized with no context request controls. You must set the request controls of a derived context instance explicitly using `setRequestControls()`. A context's context request controls are retrieved using `get-RequestControls()`.

5.4.3 The Initial Context

An application that is performing LDAP extended operations or controls can use `Initial-LdapContext` instead of`javax.naming.InitialContext` or `javax.naming.directo-ry.InitialDirContext` to create its initial context:

```
public class InitialLdapContext
                extends InitialDirContext implements LdapContext {
    public InitialDirContext() ...;
    public InitialDirContext(Hashtable env, Control[] connCtls) ...;
}
```

It can then invoke any method in the `Context`, `DirContext`, or `LdapContext` interfaces on the initial context. By using the constructor that accepts a `connCtls` argument, the application can specify controls to be used when establishing a connection with the LDAP server.

5.4.4 Unsolicited Notifications

In addition to the normal request/response style of interaction between the client and server, the LDAP v3 protocol also specifies *unsolicited notifications*—messages that are sent from the server to the client asynchronously, not in response to any client request.

JNDI supports unsolicited notifications using the event model embodied in the `javax.naming.event` package. It defines an `UnsolicitedNotificationEvent` class and a corresponding `UnsolicitedNotificationListener` interface. An application registers to receive `UnsolicitedNotificationEvents` by supplying an `UnsolicitedNotificationListener` to `EventContext.addNamingListener()`.

6 Configuration

6.1 Environment Properties

Some JNDI applications need a way to communicate various preferences and information that define the environment in which naming and directory services are accessed. For example, an application might need to specify the level of security for accessing a directory service. Or, when directory and naming services are distributed, the source of information is in more than one place—replicas, master, caches, *etc.* An application might want to access information from the authoritative source and needs to indicate this information to the JNDI system.

To address these requirements, JNDI defines a number of properties that developers and users can use to provide configuration information to the JNDI system. These are called *environment properties*.

There are different types of environment properties:

- Standard JNDI environment properties. These properties are defined by JNDI and are common across all service providers. They are defined in relatively generic terms. For example, the property “java.naming.security.principal” is used to specify the security principal for authentication to the naming service. Individual service providers map these environment properties to an interpretation appropriate for their service. These properties have the prefix “java.naming.”. Appendix A contains a list of standard JNDI environment properties. The `Context` interface defines constants for most of these environment property names.
- Service-specific environment properties. These properties are common across all service providers that implement a particular service or protocol. They have the prefix “java.naming.*service*.”, where *service* is the name of the service. For example, the prefix “java.naming.ldap.” is used for LDAP-specific environment properties.
- Feature-specific environment properties. These properties are common across all service providers that support a particular feature. They have the prefix “java.naming.*feature*.”, where *feature* is the name of the feature. For example, the prefix “java.naming.security.sasl.” is used for SASL-specific environment properties.
- Provider-specific environment properties. These properties only apply to a particular service provider. They should have a prefix that reflects this uniqueness. A common practice is to use the package name of the service provider as the prefix. For example, since Sun’s LDAP provider is primarily contained in the package `com.sun.jndi.ldap`, properties specific to Sun’s LDAP provider have the prefix “com.sun.jndi.ldap.”.

See Section 8.5 for security-related considerations when using environment properties.

Although the support for environment properties is rather extensive, it is important to note that an application typically does need to deal with them, or only needs to set one or two properties. Most properties have reasonable defaults and only need to be adjusted when the application has special requirements.

6.2 Context Environment

A context's environment is represented as a `java.util.Hashtable` or any of its subclasses (e.g., `java.util.Properties`[1]). It is typically specified using an argument to the `InitialContext`, `InitialDirContext`, or `InitialLdapContext` constructors, and augmented with data from other sources (as discussed in the rest of this section). They are inherited from the parent context as context methods proceed from one context to the next. For example, the following code creates an environment consisting of two security-related properties and creates an initial context using that environment.

```
Hashtable env = new Hashtable();
env.put(Context.SECURITY_PRINCIPAL, "jsmith");
env.put(Context.SECURITY_CREDENTIALS, "xxxxxxx");
Context ctx = new InitialContext(env);
```

Contexts that are looked up or otherwise derived from this initial context will have these two properties in their environment.

A context's environment can be examined using `Context.getEnvironment()`.

Not all environment properties are meaningful to all contexts. Those that are not meaningful are ignored by the context but inherited by derived contexts (because they might be meaningful, for instance, to federated contexts).

6.3 Resource Files

A JNDI resource file is a file in the properties file format (see `java.util.Properties`). The file contains a list of key/value pairs. The key is the name of the property (e.g., "java.naming.factory.object") and the value is a string in the format defined for that property. Here is an example of a JNDI resource file:

```
java.naming.factory.object=com.wiz.jndi.AttrsToCorba:com.wiz.jndi.ToPerson
java.naming.factory.state=com.wiz.jndi.CorbaToAttrs:com.wiz.jndi.FromPerson
java.naming.factory.control=com.wiz.jndi.MyResponseControlFactory
```

There are two kinds of JNDI resource files: application and provider.

6.3.1 Application Resource Files

When an application is deployed, it will generally have several codebase directories and JARs in its classpath. Similarly, when an applet is deployed, it will have a codebase and archives specifying where to find the applet's classes. JNDI locates all application resource files named `jndi.properties` in the classpath. In addition, if the file `$JAVA_HOME/lib/jndi.properties` exists and is readable, JNDI treats it as an additional application resource file. (`$JAVA_HOME` is the directory named by the `java.home` system property.) All of the properties contained in these files are placed into the environment of the initial context. This environment is then inherited by other contexts.

1. Note that if you use `Properties`, only the top-level properties are consulted—its defaults are not consulted—because `Hashtable.get()` is used when retrieving entries from the environment. See `java.util.Properties` for details.

For each property found in more than one application resource file, JNDI uses the first value found or, in a few cases where it makes sense to do so, it concatenates all of the values. For example, if the `java.naming.factory.object` property is found in three `jndi.properties` resource files, the list of object factories is a concatenation of the property values from all three files. Using this scheme, each deployable component is responsible for listing the factories that it exports. JNDI automatically collects and uses all of these export lists when searching for factory classes.

Application resource files are available beginning with the Java 2 Platform, except that the file in `$JAVA_HOME/lib` can be used on all Java platforms.

6.3.2 Provider Resource Files

Each service provider has an optional resource file that contains properties specific to that provider. The name of this resource is:

```
[prefix/]jndiprovider.properties
```

where *prefix* is the package name of the provider's context implementation(s), with each period (".") converted to a slash ("/").

The JNDI library will consult the provider resource file when determining the values of certain properties. Properties other than these can be set in the provider resource file at the discretion of the service provider. The service provider's documentation should clearly state which properties are allowed.

6.4 Application/Applet-scope Standard JNDI Properties

Certain standard JNDI properties can alternately be set in the Java runtime's system properties, or in an applet's parameter list. These properties are:

```
java.naming.factory.initial
java.naming.factory.object
java.naming.factory.state
java.naming.factory.control
java.naming.factory.url.pkgs
java.naming.provider.url
java.naming.dns.url
```

For JNDI to access an applet's parameters, the applet code must set the `java.naming.applet` environment property to an instance of the applet (`java.applet.Applet`).

When these properties are set as system properties or applet parameters, they affect all of the application's/applet's contexts.

6.5 How the Environment Properties are Set

When JNDI constructs an initial context, the context's environment is initialized with properties defined in the environment parameter passed to the constructor and all application resource files. For the application/applet-scope properties, their values from the system properties and the applet parameters are also used.

JNDI passes the resulting environment to the initial context implementation. The environment is then inherited by contexts that are derived from the initial context. Since JNDI performs any necessary merging of the properties and their values, there is no need for the application or context implementation to directly consult the system properties or applet parameters.

6.6 Modifications to the Environment

A context's environment can be changed using the `addToEnvironment()` and `removeFromEnvironment()` methods:

```
public interface Context {
    public Object addToEnvironment(String propName, Object val)
        throws NamingException;
    public Object removeFromEnvironment(String propName)
        throws NamingException;
    ...
}
```

Not all environment properties are meaningful to all contexts. Changes to those that are not meaningful are still recorded and passed onto derived contexts.

6.6.1 Scope

Changing a property using the `addToEnvironment()` or `removeFromEnvironment()` methods affects the context instance on which the method is invoked. For example, if you specify new credentials for a context to use, subsequent methods invoked on that context that require communication with the server will use those new credentials (perhaps internally by first creating a new connection to the server). These updated environment properties are inherited by context instances that are subsequently derived from the affected context instance, but do not otherwise affect other context instances that were in existence prior to the update.

6.6.2 Timeliness

When a change is made to the environment properties, there is no requirement that the change be verified and acted upon at the time `addToEnvironment()` or `removeFromEnvironment()` is invoked. The only requirement is that the change (or changes) be effective the next time an operation that uses that property is invoked.

6.6.3 Defaults

For some environment properties, JNDI defines defaults (see Appendix A). For others, the default might be determined by the service provider or a group of service providers. If a context's environment does not have a particular property, the context behaves as if its environment has that property with its default value.

When a property is removed from a context's environment, the context assumes the default behavior specified for that property. This does not necessarily mean that the default value must be recorded as the property's value. The removal may also be indicated by the absence of the property from the context's environment.

6.6.4 Acceptable Values

Some environment properties have a fixed set of acceptable values while others have values that must follow a particular syntax. If an unacceptable value is presented, a property-specific exception will be thrown (for example, `ConfigurationException`, `IllegalArgumentExcep-`

`tion`, or `AuthenticationNotSupportedException`). In some cases, it might be reasonable for the service provider to accept additional values than those specified, in which case, those values should be documented.

7 Scenarios

This section outlines a few application scenarios to help illustrate the capabilities enabled by JNDI.

- *The examples below are not meant to be prescriptive. There are often several ways to solve a problem, and JNDI is designed with flexibility in mind.*

7.1 User authentication

In secure systems, a user must authenticate himself to the computer, network, or service that he wishes to access. For example, logging into Unix requires the user to supply a password. Similarly, use of SSL requires that the user supply his X.509 certificate. Such authentication information can be stored as attributes associated with each user in the directory. The system performing the authentication would look up the attribute (for example, "password") of the user and verify the authenticity using the information supplied by the user.

```
DirContext ctx = new InitialDirContext();
Attribute attr = ctx.getAttributes(userName).get("password");
String password = (String)attr.get();
```

7.2 Electronic Mail

A useful feature of an electronic mail system is a directory service that provides a mapping between users and email addresses. This allows mail users to search for the email address of a particular user. This is analogous to searching for an individual's telephone number in the phone book in order to dial his phone number. For example, when I want to send mail to John Smith in my department, I search for "John Smith" in the directory using a "search" widget in the mail application. The widget returns to me five entries of John Smith, from which I select the one that is in a building on my site and use the email address attribute associated with that entry.

```
NamingEnumeration matches =
    deptCtx.search("user", new BasicAttributes("name", "John Smith"));
// use matches to construct a selectable list for end-user
while (matches.hasMore()) {
    SearchResult item = (SearchResult) matches.next();
    Attributes info = item.getAttributes();
    /* display attributes */
    ...
}
```

The directory could also be used by users to set up personalized address books. For example, once I have located John Smith's email address, I might not want to search the directory again each time I send him mail. Instead, I can create a personal subtree in the directory in which I maintain entries that I frequently use, possibly by creating links to the existing entries.

7.3 Databases

Database applications can use the directory to locate database servers. For example, a financial application needs to get the stock quotes from a stock quote server using JDBC. This application can enable the user to select the stock quote server based on specification of some attributes (such as coverage of which markets and frequency of quote updates). The application searches the directory for quote servers that meet these attributes, and then retrieves the "location" attribute (a JDBC URL) of the selected quote server and connects to it.

```
NamingEnumeration matches =
    ctx.search("service/stockQuotes",
        "(&(market=NASDAQ)(updateFreqency<=300))",
        searchctls);
while (matches.hasMore()) {
    SearchResult item = (SearchResult)matches.next();
    Attribute location = item.getAttributes().get("location");
    ...
}
```

7.4 Browsing

When using almost any kind of interactive application that asks a user to input names, the user's job is made easier if a namespace browser is available to him. The browser can either be built into the application and tailored to suit that application in particular, or it can be more general-purpose such as a typical web browser.

A very simple example of a JNDI browser allows a user to "walk" through a namespace, viewing the atomic names at each step along the way. The browser prints a "*" to highlight the name of each `Context`, thus telling the user where he can go next.[1]

1. The `isContext()` method used in the example is not part of JNDI. It is a method that must be provided by the application.

```
// Start at the top -- the initial context.
Context ctx = new InitialContext();
while (ctx != null) { // display one level
    NamingEnumeration items = ctx.list();
    while (items.hasMoreElements()) {
        NameClassPair item = (NameClassPair)items.next();
        if (isContext(item.getClassName())) {
            System.out.print("*");
        } else {
            System.out.print(" ");
        }
        System.out.println(" " + item.getName());
    }
    // Take the next step down into the namespace.
    String target = input.readLine();
    try {
        ctx = (Context)ctx.lookup(target);
    } catch (NamingException e) {
        // handle error
    } catch (ClassCastException e) {
        // not a context; cannot traverse
    }
}
```

7.5 Network Printing

An important function of a printing service is to provide a means for its human users to easily discover and select printers in the network. An application that needs to print, or the machine on which it runs, should not have to be configured each time a new printer is added to the network. The scope of network access to printers may range from a workgroup to global. The printing service can use the directory to provide this capability.

Assume that printers are represented by a `Printer` interface. One of the methods in it could be `print()` which, when given an `InputStream`, will read data from `InputStream` and print it on the printer represented by this instance of `Printer`.

```
interface Printer {
    void print(InputStream data) throws PrinterException;
    ...
}
```

A user selects a printer using a logical printer name, either explicitly or through default settings. For example, the user might have specified a default printer to use for all his applications, which is overridden only when he explicitly specifies another printer to use. The application that is accepting the print request takes the printer name and looks it up in the directory service. The application expects to receive as the result an object that implements the `Printer` interface.

```
void myAppPrint(String printerName, String fileName)
    throws IOException {
    try {
        DirContext ctx = new InitialDirContext();
        Printer prt = (Printer) ctx.lookup(printerName);
        prt.print(new FileInputStream(fileName));
    } catch (NamingException e){
        System.err.println("Could not locate printer: " + e);
    } catch (ClassCastException e) {
        System.err.println(printerName + "does not name a printer");
    }
}
```

7.5.1 Browsing and searching for printers

Selecting a printer by explicitly giving its name is but one way of identifying a printer. The user can also use the directory to see the different printers available (browsing), or to search for printers with particular attributes. For example the user can ask the directory to list all the printers on the second floor of building 5 in the Mountain View campus, or search for all color laser printers with 600dpi resolution. From the application's perspective, just as `lookup()` returned a `Printer` object, the list and search operations also provide the same capability of returning `Printer` objects that the application could use to submit print requests.

8 Security Considerations

There are two main settings in which JNDI is used: with and without a security manager installed.

In the case of Java applications running with no security manager installed, the code is trusted and the application can access service providers from the local classpath. Furthermore, there is no restriction if the service providers access local files, or make network connections to naming or directory servers anywhere on the network.

In the case of an applet or application running with a security manager installed, there can be trusted code and untrusted code within the same applet or application. The *Sharing Context Handles* and *Context Environment* sections below are especially applicable in such a scenario. The ability of an applet or an application running with a security manager installed to access service providers, especially service providers that require the use of restricted resources (like the file system or network connections) may be severely limited.

8.1 JNDI Classes

The classes in the JNDI packages contain no native methods. They do not require any special installation in order to run inside an applet or an application.

JNDI uses several system properties (see Section 6.4). This allows applets and applications to be configured easily without much programming. However, an applet or application might have restricted access to some or all system properties as a result of the security manager installed on the platform on which it is running. Consequently, JNDI also allows these same properties to be specified as applet parameters, in resource files, or as environment properties passed to a context.

In the Java 2 Platform, the JNDI classes use `doPrivileged` blocks when accessing the system properties listed in Section 6.4.

8.2 Security Model

JNDI does not define a security model or a common security interface for accessing naming and directory servers. Security-related operations, such as those required for authentication or access control to the directory service, are dealt with by individual service providers. JNDI provides the means by which an application or applet can pass such security-related information to service providers in order to establish a connection with the service, but does not itself take part in such security-related activities.

JNDI also provides the means by which security-related problems can be indicated to the client in the form of security-related exceptions.

JNDI service providers are controlled by the security manager in place when they try to gain access to protected resources such as the file system or network. Some service providers may control directory access by making use of the Java 2 Platform security architecture (for example, allowing access to special ports for administration-related applets).

8.3 Access To Servers

Naming and directory services typically have their own security system in place to protect information stored therein. For example, one directory might require that its users first “login” to the directory before reading or updating information in the directory. Some services might allow anonymous access to part of its namespace/directory.

Once a user has logged into a service, it is imperative for security reasons not to share that privilege with untrusted code.

8.4 Sharing Context Handles

In the following discussion, we refer to a *context handle* as a reference to a `Context` instance. This is analogous to how a reference to a `Reader`/`Writer`/`InputStream`/`OutputStream` instance is often referred to as a *file handle*.

A context handle should be treated like any other protected resource. If a piece of trusted code obtains a context handle (possibly by authenticating its access with the directory server), it should protect the use of that context against unauthorized or untrusted code. This is analogous to how application and/or applet code should protect file handles. For example, if a piece of trusted code opens a file for writing (and it was granted such privilege because of its trusted nature), it should be careful about passing that file handle to any other pieces of code, trusted or otherwise.

Similarly, giving access of a context handle to untrusted code may lead to its misuse in accessing or updating information in the directory, or accessing security-sensitive environment properties associated with the context.

8.5 Context Environment

JNDI allows the application/applet to pass preferences and information to a context in the form of an environment. The application/applet can also get the current environment from a context. See Chapter 6 and Appendix A for more information on a context’s environment.

The nature of the information contained in a context’s environment might be sensitive and need protection from untrusted access. Specifically, the environment properties `java.naming.security.principal` and `java.naming.security.credentials` contain information that should not be given out to untrusted code. Service providers might take precaution to protect against accessing these properties (see *Responsibilities of Service Providers* below). Client applications and applets should take care not to pass context handles with such sensitive environment properties to untrusted code.

8.6 Class Loading

JNDI allows the class files to be loaded dynamically.

When JNDI is run on the JDK 1.1.x platform, it uses the RMI class loader. The classes can only be loaded if there is a security manager installed, and if that security manager permits the class to be loaded. When such classes are loaded, they run in the security context dictated by the security manager.

When JNDI is run on the Java 2 platform, it will attempt to load such classes from the locations specified in the codebase using the `java.net.URLClassLoader`. In order for the class loading to succeed, you must grant the application and the JNDI and service provider classes the permissions appropriate for the URLs named in the codebase. For example, if the URL scheme is "http" or "ftp", you must grant the application the appropriate `java.net.SocketPermission`; if the URL scheme is "file", you must grant the application the appropriate `java.io.FilePermission`.

8.7 Serializable Objects

Several of the JNDI classes are serializable. This allows the objects to be accessed in the form of a byte stream, possibly outside of the environment in which they were created. See the document at the following URL regarding security issues related to serialized objects.

http://java.sun.com/products/jdk/1.2/docs/guide/serialization/spec/security.doc.html

8.8 Responsibilities of Service Providers

8.8.1 Context Environment

When a context handle is created (either by getting the initial context or by looking it up or by creating it from the directory), some environment properties may be specified for it. Sometimes security-related properties are required for the creation of the context handle (such as user information that "logs" the user in with the directory). The service provider should take care to protect this security-sensitive information from untrusted code.

The service provider needs to protect the context's environment from being tampered or otherwise modified by untrusted code. The service provider needs to protect the security-sensitive environment properties from being read by untrusted code. It might do this by disallowing any thread whose execution context and/or trust level is different than that originally held by the thread that created the context handle to use the context handle. Or it might disallow certain operations (such as accessing security-sensitive environment properties). Or it might simply not return security-sensitive environment properties, or only return them to trusted code.

8.8.2 Network Security

Untrusted code (such as those found in untrusted applets) have limited access to the network. Untrusted applets, for example, can only create a network connection to the host from which they were downloaded. With finer-grain security models, it will be possible for the service provider itself to be trusted code, and hence be allowed to connect to hosts not allowed for untrusted applets. In such a scenario, the service provider should be careful not to compromise the security intended by the security manager. If the service provider is sure that access by an untrusted applet to the directory will cause no security problems, then it may proceed to offer such a service to untrusted code. For example, allowing untrusted code to access a directory "anonymously" would post no security problems because the directory already allows any anonymous client (written in the Java programming language or otherwise) to access the same data.

Most naming and directory services are accessed over the network. Although the data requested is protected by the server's authentication and access control mechanisms, some protocols do not protect (encrypt) the data being sent as replies. Again, this is not a problem particular to a client using JNDI but a problem for any client accessing the directory. The service provider should document the security implications associated with using the associated directory over a network.

8.8.3 Accessing Local Files

Similar to network access, untrusted code has limited access to the local file system. If the service provider has special privileges for accessing local files, it should do so with utmost precaution so as not to compromise the security policies intended by the runtime/platform.

8.8.4 Privileged Code, Native Methods

A service provider that is written completely in the Java programming language with no privileged sections is controlled by the same security policies afforded other code written in the Java programming language. All protected resources that it attempts to access go through the same security manager and access controller.

If a service provider contains privileged code sections, or if it contains native methods, then it needs to be especially careful to preserve the security policies intended by the runtime/platform.

9 Design Choices

9.1 Separation of Interfaces into Context and DirContext

There are two core interfaces in JNDI: `Context`, and `DirContext`, with `DirContext` extending the base naming operations in `Context` with directory service operations. They have been separated into separate interfaces both for modularity and also in keeping with the "pay for what you use" goal of JNDI.

Naming is a basic component found in many computing services such as file systems, spreadsheets, calendar services, and directory services. By having a base `Context` interface for the naming operations, we enable its use by all these other services, not just for directory services.

`DirContext` extends `Context` to provide basic directory service operations, which include manipulation of attributes associated with named objects, attribute-based searches, and schema-related operations of those attributes and named objects.

9.2 Separation of JNDI into Different Functional Packages

JNDI is separated into four client packages (`javax.naming`, `javax.naming.directory`, `javax.naming.event`, `javax.naming.ldap`) and a service provider package (`javax.naming.spi`). The idea is that each package contains the interfaces and classes required for a particular category of applications, again in keeping with the "pay for what you use" goal. For example, an application that just wants to perform name-lookups only needs to use the `javax.naming` package. An application that wants to examine/modify attributes associated with an object uses the `javax.naming` and `javax.naming.directory` packages. An application that needs to use LDAP-specific controls or extended operations uses the `javax.naming.ldap` package. There is a step-by-step progression of what classes and interfaces each category of application writer needs to learn and use.

9.3 Separation of Client APIs and Service Provider Interfaces

JNDI separates interfaces and classes that a client application needs to use from those that are only of interest to service providers into different packages. For example, a client would use interfaces and classes from `javax.naming`, while a service provider that is hooking up a naming service would use both `javax.naming` and `javax.naming.spi`. The package delineation minimizes confusion for the application developer and makes clear which packages he needs to examine when writing his program.

9.4 Multiple methods for listing Context

There are two common types of applications that list contexts: browser-style applications, and applications that need to perform operations on the objects in a context en-masse. Browser-style applications typically want to display the names of the contents of a context. In addition to the names, many browsers often require type information of the objects bound to the names, so that it can display appropriate pictorial representations of the objects. The browser is usually interactive. Once a user has used a browser to display the contents of a context, he would then select one or a few of the entries displayed and request more information on it.

Some applications need to perform operations on objects within a context en-masse. For example, a backup program might want to perform "file stats" operations on all the objects in a file directory. A printer administration program might want to restart all the printers in a building. To perform such operations, these programs need to obtain all the objects bound in a context.

With these two common styles of usage in mind, the `Context` interface has two types of list methods: `list()` and `listBindings()`. `list()` returns a list of name/class-name pairs while `listBindings()` returns a list of name/class-name/object tuples. `list()` is designed for browser-style applications that want mostly just the names and types of objects bound in a context. `listBindings()` is for applications that want to potentially get all the objects in the context, as well as their names and types. `listBindings()` returns an enumeration of `Binding`. Both the `listBindings()` operation itself and invocation of methods in the `Binding` class (e.g. `getObject()`) could be implemented lazily or eagerly. Using `listBindings()` simply indicates the potential that the caller might want all or many of the objects in the context so that implementations that are able can optimize for it. Using `list()` indicates that the caller is unlikely to want all, if any, objects in the context so implementations can optimize for that if possible.

An alternative is to have a single list operation and have the lazy or eager behavior as part of the implementation of `Binding`. The advantage of this is that there is a single list operation to learn. The disadvantage is that the caller has no way of indicating which piece of information he wants back from list, and subsequently, implementations cannot optimize for the eventual behavior of the program.

9.5 Support for Federation

Federation is a first-class concept in JNDI. In the client interfaces, it is supported by the use of the `Name` interface for specifying names that can span one or more namespaces. The caller of the methods in the client interface need not know anything else regarding federation. Resolution of names across multiple systems is handled by the SPI and does not involve any intervention on the part of the caller.

Although federation is a first-class concept, that does not mean that all callers and service providers must make use of it. If an application or service does not want to take advantage of federation, there is no requirement that `Name` always span multiple namespaces. `Name` can just name objects within a single namespace, and the SPI can handle name resolution within a single namespace as well (as a degenerate case of multiple namespace support).

9.6 DirContext versus DirObject

Instead of having `DirContext` extend `Context`, an alternative would be to not extend `Context` at all but to have a separate interface called `DirObject` that encapsulates all the directory-related methods. In that case, an object can implement both `Context` and `DirObject` if it supports both the naming and directory operations; another object might implement just `DirObject`.

The problem with eliminating `DirContext` is that `DirContext` contains some hybrid operations that involve both naming and directories (`bind()`, `createSubcontext()` methods that accept attributes as arguments). To keep these operations *and* have `DirObject` at the same time would produce the need for a third interface (perhaps called `DirContext`) to contain just these hybrids.

Furthermore, having `DirContext` instead of `DirObject` is somewhat more convenient in that you can perform some operations in one step instead of two. For example `DirContext.getAttributes()` could be used to get the attributes associated with a named object, whereas with `DirObject`, you would need first to resolve to the object (`Context.lookup()`) and then use `DirObject.getAttributes()` to get the attributes from it.

9.7 Support for Schemas

The `DirContext` interface contains support for schemas. For example, from a `DirContext` object you can obtain its schema object, which points to the directory space where the schema for this particular `DirContext` instance is defined. From a `DirContext` object, you can also obtain its schema class definition (i.e. information about what type of object this represents in the directory). There is further support for schemas in the `Attribute` class, which contains methods for obtaining an attribute's syntax information (i.e. what is the type of the attribute's value) and the attribute's definition (e.g. is it multivalued, syntax, constraints on its syntax). There is no requirement that any of this schema information be dynamically accessible (i.e. points to live directory spaces). Support for such schema information could be generated statically by the service provider. For example, a particular directory service might only support string attribute values, so it can hard-wire the syntax of the attributes that it returns. Another directory might support only static schemas (where information in the schema are not modifiable). Yet another directory might support fully dynamic schemas. The interfaces and classes in `DirContext` are flexible enough that these different levels of support for schemas can be accommodated.

9.8 Overloaded Methods in Context and DirContext

For each method in the `Context` and `DirContext` interfaces that accepts a `Name` argument, there is a corresponding overloaded form that accepts a `String` argument for specifying a name.

The motivation for having the `String`-based methods is that there are many applications that simply accept a string name from the end-user and perform context methods on the object named by that string name. For those applications, it is useful to have the context methods accept a string for the name directly, instead of requiring the applications to first construct a `Name` object using the string name.

The motivation for having the `Name`-based methods is that there are also many applications that manipulate names and do not want to worry about syntactic details of the names' string forms when composing and modifying names. These applications deal with the parsed form of names and hence would prefer to deal with `Name` objects rather than string names. For these applications, we provide the `Name`-based methods in the context interfaces. Not providing these methods would probably cause proliferation of `Name`-like interfaces/classes to support manipulation of names in their structural form in applications developed on top of JNDI.

9.9 Reference and Referenceable

There are different ways in which applications and services can use the directory to locate objects. JNDI is general enough that it accommodates several different models. For some applications, the object bound in the directory is the object itself. An application may build up a dynamic directory while the application is active, and delete the directory when the application exits. Another application might store URLs as attributes for locating objects in its namespace. Other systems might bind some reference information in the directory, which can subsequently be used to locate or access the actual object. This last case is quite common, especially for making Java applications take advantage of services in the installed base. The reference in the directory acts as a “pointer” to the real object.

JNDI defines a `Reference` class to provide a uniform way of representing reference information. A `Reference` contains information on how to access an object. It consists of a list of addresses and class information about the object to which this reference refers. When binding a name to an object that is to be represented in the directory as a reference, the desired effect is that the object’s reference be extracted and bound. To allow for this behavior, the object’s class must implement the `Referenceable` interface, which contains the method `getReference()`.

There is some similarity between the interfaces `Serializable` and `Referenceable` and a natural question is “why not just use `Serializable` instead?” The answer is that a serialized object is really a frozen version of the object, whereas the reference contains just the information needed to construct it. The serialized version may have a lot more state which may not be appropriate for storage in the directory.

9.10 Automatically Turning References into Objects

For an object that is bound as a `Reference` in the directory, JNDI SPI framework automatically creates and instantiates the object identified by the reference. In this way, the program can simply narrow the result of `lookup()` to the expected class, instead of calling a separate operation to transform the result of `lookup()` into an object of the expected class.

For example, if you are looking up a printer object, a successful lookup would return to you a printer object that you can directly use.

```
Printer prt = (Printer) ctx.lookup(somePrinterName);
prt.print(someFileName);
```

JNDI does this automatically, instead of requiring an explicit conversion step, because this is expected to be the common usage pattern. By having the `Reference` class, and a common mechanism for converting a `Reference` into the object identified by the `Reference`, JNDI encourages different applications and system providers to utilize this mechanism, rather than inventing separate mechanisms on their own.

Appendix A: JNDI Standard Environment Properties

Table 1: JNDI Environment Properties[a]

Program Configuration[b]	
`java.naming.factory.initial` `(Context.INITIAL_CONTEXT_FACTORY)`	Class name of initial context factory to use. See `InitialContext`. No default.
`java.naming.factory.object` `(Context.OBJECT_FACTORIES)`	Colon-separated list of class names of object factory classes to use. See `NamingManager.getObjectInstance()` and `DirectoryManager.getObjectInstance()`. Defaults to empty list.
`java.naming.factory.state` `(Context.STATE_FACTORIES)`	Colon-separated list of class names of state factory classes to use. See `NamingManager.getStateToBind()` and `Direct-oryManager.getStateToBind()`. Defaults to empty list.
`java.naming.factory.control` `(LdapContext.CONTROL_FACTORIES)`	Colon-separated list of class names of response control factory classes to use. See `ControlFactory.getControlInstance()`. Defaults to empty list.
`java.naming.factory.url.pkgs` `(Context.URL_PKG_PREFIXES)`	Colon-separated list of package prefixes to use when loading in URL context factories. `com.sun.jndi.url` is always added to end of list. See `NamingManager.getURLContext()`. Defaults to empty list.
Access Configuration[c]	
`java.naming.provider.url` `(Context.PROVIDER_URL)`	Specifies configuration information for provider to use. Defaults to provider default, using provider's own configuration or discovery protocols.
`java.naming.dns.url` `(Context.DNS_URL)`	Specifies the DNS host and domain names. No default.
Service-Related	
`java.naming.authoritative` `(Context.AUTHORITATIVE)`	Specifies the authoritativeness of the service requested. If "true", specifies most authoritative source is to be used (e.g., bypass any caches, or bypass replicas in some systems). Otherwise, source need not be (but can be) authoritative. Defaults to "false".
`java.naming.batchsize` `(Context.BATCHSIZE)`	Specifies the preferred batch size to use when returning data via the service's protocol. This is a hint to the provider to return the results of operations in batches of the specified size, so that the provider can optimize its performance and usage of resources. It does not affect the total number or size of the data returned. Defaults to provider default.

Table 1: JNDI Environment Properties[a]

`java.naming.referral` `(Context.REFERRAL)`	Specifies that referrals encountered by the service provider are to be followed automatically. If "follow", follow referrals automatically. If "ignore", ignore referrals encountered. If "throw", throw `ReferralException` when a referral is encountered. Defaults to provider default.
Security	
`java.naming.security.protocol` `(Context.SECURITY_PROTOCOL)`	Security protocol to use for service. Defaults to provider default.
`java.naming.security.authentication` `(Context.SECURITY_AUTHENTICATION)`	Takes values "none", "simple", "strong", or a provider-specific string (e.g., "CRAM-MD5 DIGEST-MD5 EXTERNAL"). Defaults to provider default.
`java.naming.security.principal` `(Context.SECURITY_PRINCIPAL)`	Identity of principal (e.g., user) for the authentication scheme. Defaults to provider default.
`java.naming.security.credentials` `(Context.SECURITY_CREDENTIALS)`	Principal's credentials for the authentication scheme. The particular type of credentials is determined by the authentication scheme chosen. Examples of different types of credentials are passwords, keys, and certificates. Defaults to provider default.
Internationalization	
`java.naming.language` `(Context.LANGUAGE)`	Specifies a colon-separated list of preferred language to use with this service (e.g., "en-US", "fr", "fr-CH", "ja-JP-kanji"). Languages are specified using tags defined in RFC 1766. Defaults to provider default.

a. The `Context` and `LdapContext` interfaces define constants for these property names. The names of the constants are shown in parentheses below the property's string names.
b. These properties may be set in the Java runtime's system properties or in an applet's parameter list.
c. These properties may be set in the Java runtime's system properties or in an applet's parameter list.

Appendix B: Examples for LDAP Programmers

This appendix contains sample JNDI programs intended to help a developer familiar with the LDAP C API. Starting with sample programs from the Netscape Directory SDK for accessing and updating the directory using the LDAP C API, we show the equivalent way of doing the same thing for Java applications using JNDI.

B.1 Search the Directory

B.1.1 Search Using LDAP C API

```
/*
 * Copyright (c) 1996.  Netscape Communications Corporation.  All
 * rights reserved.
 *
 * Search the directory for all people whose surname (last name) is
 * “Jensen”.  Since the “sn” attribute is a caseignorestring (cis), case
 * is not significant when searching.
 *
 */

#include “examples.h”

int
main( int argc, char **argv )
{
        LDAP                *ld;
        LDAPMessage         *result, *e;
        BerElement          *ber;
        char                *a, *dn;
        char                **vals;
        int                 i;

        /* get a handle to an LDAP connection */
        if ( (ld = ldap_init( MY_HOST, MY_PORT )) == NULL ) {
                perror( “ldap_init” );
                return( 1 );
        }
        /* authenticate to the directory as nobody */
        if ( ldap_simple_bind_s( ld, NULL, NULL ) != LDAP_SUCCESS ) {
                ldap_perror( ld, “ldap_simple_bind_s” );
                return( 1 );
        }
        /* search for all entries with surname of Jensen */
        if ( ldap_search_s( ld, MY_SEARCHBASE, LDAP_SCOPE_SUBTREE,
                MY_FILTER, NULL, 0, &result ) != LDAP_SUCCESS ) {
                ldap_perror( ld, “ldap_search_s” );
                return( 1 );
        }
        /* for each entry print out name + all attrs and values */
        for ( e = ldap_first_entry( ld, result ); e != NULL;
            e = ldap_next_entry( ld, e ) ) {
                if ( (dn = ldap_get_dn( ld, e )) != NULL ) {
                    printf( “dn: %s\n”, dn );
                    ldap_memfree( dn );
                }
                for ( a = ldap_first_attribute( ld, e, &ber );
                    a != NULL; a = ldap_next_attribute( ld, e, ber ) ) {
                        if ((vals = ldap_get_values( ld, e, a)) != NULL ) {
                                for ( i = 0; vals[i] != NULL; i++ ) {
                                    printf( “%s: %s\n”, a, vals[i] );
                                }
                                ldap_value_free( vals );
                        }
                        ldap_memfree( a );
                }
                if ( ber != NULL ) {
                        ber_free( ber, 0 );
                }
                printf( “\n” );
        }
        ldap_msgfree( result );
        ldap_unbind( ld );
        return( 0 );
}
```

B.1.2 Search Using JNDI

```
/*
 * Copyright (c) 1997.  Sun Microsystems. All rights reserved.
 *
 * Search the directory for all people whose surname (last name) is
 * "Jensen".  Since the "sn" attribute is a caseignorestring (cis), case
 * is not significant when searching.
 *
 * [equivalent to search.c in Netscape's SDK.]
 *
 */

import java.util.Hashtable;
import java.util.Enumeration;

import javax.naming.*;
import javax.naming.directory.*;

class Search {

public static void main(String[] args) {

    Hashtable env = new Hashtable(5, 0.75f);
    /*
     * Specify the initial context implementation to use.
     * This could also be set by using the -D option to the java program.
     * For example,
    *   java -Djava.naming.factory.initial=com.sun.jndi.ldap.LdapCtxFactory \
     *     Search
     */
    env.put(Context.INITIAL_CONTEXT_FACTORY, Env.INITCTX);

    /* Specify host and port to use for directory service */
    env.put(Context.PROVIDER_URL, Env.MY_SERVICE);

    try {
        /* get a handle to an Initial DirContext */
        DirContext ctx = new InitialDirContext(env);

        /* specify search constraints to search subtree */
        SearchControls constraints = new SearchControls();
        constraints.setSearchScope(SearchControls.SUBTREE_SCOPE);

        /* search for all entries with surname of Jensen */
        NamingEnumeration results
            = ctx.search(Env.MY_SEARCHBASE, Env.MY_FILTER, constraints);
        /* for each entry print out name + all attrs and values */
        while (results != null && results.hasMore()) {
            SearchResult si = (SearchResult)results.next();

            /* print its name */
            System.out.println("name: " + si.getName());

            Attributes attrs = si.getAttributes();
            if (attrs == null) {
                System.out.println("No attributes");
            } else {
                /* print each attribute */
                for (NamingEnumeration ae = attrs.getAll();
                     ae.hasMoreElements();) {
                    Attribute attr = (Attribute)ae.next();
                    String attrId = attr.getID();

                    /* print each value */
                    for (Enumeration vals = attr.getAll();
                         vals.hasMoreElements();
                       System.out.println(attrId + ": " + vals.nextElement()))
                            ;
                }
            }
            System.out.println();
        }
    } catch (NamingException e) {
        System.err.println("Search example failed.");
        e.printStackTrace();
    }
}
}
```

B.2 Read An Entry

B.2.1 Read Using LDAP C-API

```
/*
 * Copyright (c) 1996.  Netscape Communications Corporation.  All
 * rights reserved.
 *
 * Search the directory for the specific entry
 * “cn=Barbara Jensen, ou=Product Development, o=Ace Industry, c=US”.
 * Retrieve all attributes from the entry.
 *
 */

#include “examples.h”

int
main( int argc, char **argv )
{
    LDAP        *ld;
    LDAPMessage *result, *e;
    BerElement  *ber;
    char        *a, *dn;
    char        **vals;
    int         i;

    /* get a handle to an LDAP connection */
    if ( (ld = ldap_init( MY_HOST, MY_PORT )) == NULL ) {
        perror( “ldap_init” );
        return( 1 );
    }
    /* authenticate to the directory as nobody */
    if ( ldap_simple_bind_s( ld, NULL, NULL ) != LDAP_SUCCESS ) {
        ldap_perror( ld, “ldap_simple_bind_s” );
        return( 1 );
    }
    /* search for Babs’ entry */
    if ( ldap_search_s( ld, ENTRYDN, LDAP_SCOPE_SUBTREE,
            “(objectclass=*)”, NULL, 0, &result ) != LDAP_SUCCESS ) {
        ldap_perror( ld, “ldap_search_s” );
        return( 1 );
    }
    /* for each entry print out name + all attrs and values */
    for ( e = ldap_first_entry( ld, result ); e != NULL;
            e = ldap_next_entry( ld, e ) ) {
        if ( (dn = ldap_get_dn( ld, e )) != NULL ) {
            printf( “dn: %s\n”, dn );
            ldap_memfree( dn );
        }
        for ( a = ldap_first_attribute( ld, e, &ber );
                a != NULL; a = ldap_next_attribute( ld, e, ber ) ) {
            if ((vals = ldap_get_values( ld, e, a)) != NULL ) {
                for ( i = 0; vals[i] != NULL; i++ ) {
                    printf( “%s: %s\n”, a, vals[i] );
                }
                ldap_value_free( vals );
            }
            ldap_memfree( a );
        }
        if ( ber != NULL ) {
            ber_free( ber, 0 );
        }
        printf( “\n” );
    }
    ldap_msgfree( result );
    ldap_unbind( ld );
    return( 0 );
}
```

B.2.2 Read Using JNDI

```
/*
 * Copyright (c) 1997.  Sun Microsystems.  All rights reserved.
 *
 * Search the directory for the specific entry
 * "cn=Barbara Jensen, ou=Product Development, o=Ace Industry, c=US".
 * Retrieve all attributes from the entry.
 *
 * [Equivalent to rdentry.c in Netscape SDK]
 */

import java.util.Hashtable;
import javax.naming.*;
import javax.naming.directory.*;


class Rdentry {
public static void main(String[] args) {

    Hashtable env = new Hashtable(5, 0.75f);
    /*
     * Specify the initial context implementation to use.
     * This could also be set by using the -D option to the java program.
     * For example,
     *   java -Djava.naming.factory.initial=com.sun.jndi.ldap.LdapCtxFactory \
     *       Rdentry
     */
    env.put(Context.INITIAL_CONTEXT_FACTORY, Env.INITCTX);

    /* Specify host and port to use for directory service */
    env.put(Context.PROVIDER_URL, Env.MY_SERVICE);

    try {
        /* get a handle to an Initial DirContext */
        DirContext ctx = new InitialDirContext(env);

        /* Read Babs' entry */
        Attributes attrs = ctx.getAttributes(Env.ENTRYDN);

        if (attrs == null) {
            System.out.println(Env.ENTRYDN + "has no attributes");
        } else {
            /* print each attribute */
            for (NamingEnumeration ae = attrs.getAll();
                 ae.hasMoreElements();) {
                Attribute attr = (Attribute)ae.next();
                String attrId = attr.getID();

                /* print each value */
                for (NamingEnumeration vals = attr.getAll();
                     vals.hasMoreElements();
                     System.out.println(attrId + ": " + vals.nextElement()))
                    ;
            }
        }
    } catch (NamingException e) {
        System.err.println("Rdentry example failed.");
        e.printStackTrace();
    }
}
}
```

B.3 Get Attributes

B.3.1 Get Attributes Using LDAP C API

```
/*
 * Copyright (c) 1996.  Netscape Communications Corporation.  All
 * rights reserved.
 *
 * Retrieve several attributes of a particular entry.
 */

#include “examples.h”


int
main( int argc, char **argv )
{
    LDAP                *ld;
    LDAPMessage         *result, *e;
    char                **vals, *attrs[ 5 ];
    int                 i;

    /* get a handle to an LDAP connection */
    if ( (ld = ldap_init( MY_HOST, MY_PORT )) == NULL ) {
        perror( “ldap_init” );
        return( 1 );
    }

    attrs[ 0 ] = “cn”;         /* Get canonical name(s) (full name) */
    attrs[ 1 ] = “sn”;         /* Get surname(s) (last name) */
    attrs[ 2 ] = “mail”;       /* Get email address(es) */
    attrs[ 3 ] = “telephonenumber”;      /* Get telephone number(s) */
    attrs[ 4 ] = NULL;

    if ( ldap_search_s( ld, ENTRYDN, LDAP_SCOPE_BASE,
           “(objectclass=*)”, attrs, 0, &result ) != LDAP_SUCCESS ) {
        ldap_perror( ld, “ldap_search_s” );
        return( 1 );
    }

    /* print it out */
    if (( e = ldap_first_entry( ld, result )) != NULL ) {
        if (( vals = ldap_get_values( ld, e, “cn” )) != NULL ) {
            printf( “Full name:\n” );
            for ( i = 0; vals[i] != NULL; i++ ) {
                printf( “\t%s\n”, vals[i] );
            }
            ldap_value_free( vals );
        }
        if (( vals = ldap_get_values( ld, e, “sn” )) != NULL ) {
            printf( “Last name (surname):\n” );
            for ( i = 0; vals[i] != NULL; i++ ) {
                printf( “\t%s\n”, vals[i] );
            }
            ldap_value_free( vals );
        }
        if (( vals = ldap_get_values( ld, e, “mail” )) != NULL ) {
            printf( “Email address:\n” );
            for ( i = 0; vals[i] != NULL; i++ ) {
                printf( “\t%s\n”, vals[i] );
            }
            ldap_value_free( vals );
        }
        if (( vals = ldap_get_values( ld, e, “telephonenumber” )) != NULL ) {
            printf( “Telephone number:\n” );
```

```
                for ( i = 0; vals[i] != NULL; i++ ) {
                    printf( “\t%s\n”, vals[i] );
                }
                ldap_value_free( vals );
            }
        }
        ldap_msgfree( result );
        ldap_unbind( ld );
        return( 0 );
    }
```

B.3.2 Get Attributes Using JNDI

```
/*
 * Copyright (c) 1997.  Sun Microsystems.  All rights reserved.
 *
 * Retrieve several attributes of a particular entry.
 *
 * [equivalent to getattrs.c in Netscape SDK]
 */

import java.util.Hashtable;
import java.util.Enumeration;

import javax.naming.*;
import javax.naming.directory.*;

class Getattrs {

public static void main(String[] args) {

    Hashtable env = new Hashtable(5, 0.75f);
    /*
     * Specify the initial context implementation to use.
     * For example,
     * This could also be set by using the -D option to the java program.
     *   java -Djava.naming.factory.initial=com.sun.jndi.ldap.LdapCtxFactory \
     *       Getattrs
     */
    env.put(Context.INITIAL_CONTEXT_FACTORY, Env.INITCTX);

    /* Specify host and port to use for directory service */
    env.put(Context.PROVIDER_URL, Env.MY_SERVICE);

    try {
        /* get a handle to an Initial DirContext */
        DirContext ctx = new InitialDirContext(env);

        String[] attrs = new String[4];
        attrs[ 0 ] = "cn";              /* Get canonical name(s) (full name) */
        attrs[ 1 ] = "sn";              /* Get surname(s) (last name) */
        attrs[ 2 ] = "mail";            /* Get email address(es) */
        attrs[ 3 ] = "telephonenumber"; /* Get telephone number(s) */

        Attributes result = ctx.getAttributes(Env.ENTRYDN, attrs);

        if (result == null) {
            System.out.println(Env.ENTRYDN +
                               "has none of the specified attributes.");
        } else {
            /* print it out */
            Attribute attr = result.get("cn");
            if (attr != null) {
                System.out.println("Full name:" );
                for (NamingEnumeration vals = attr.getAll();
                     vals.hasMoreElements();
                     System.out.println("\t" + vals.nextElement()))
                    ;
            }

            attr = result.get("sn");
            if (attr != null) {
                System.out.println("Last name (surname):" );
                for (NamingEnumeration vals = attr.getAll();
                     vals.hasMoreElements();
                     System.out.println("\t" + vals.nextElement()))
                    ;
            }

            attr = result.get("mail");
            if (attr != null) {
                System.out.println("Email address:" );
                for (NamingEnumeration vals = attr.getAll();
                     vals.hasMoreElements();
                     System.out.println("\t" + vals.nextElement()))
                    ;
            }
            attr = result.get("telephonenumber");
            if (attr != null) {
                System.out.println("Telephone number:" );
                for (NamingEnumeration vals = attr.getAll();
                     vals.hasMoreElements();
                     System.out.println("\t" + vals.nextElement()))
                    ;
            }
        }
    } catch (NamingException e) {
        System.err.println("Getattrs example failed.");
        e.printStackTrace();
    }
}
}
```

B.4 Compare An Attribute

B.4.1 Compare Using LDAP C API

```
/*
 * Copyright (c) 1996.  Netscape Communications Corporation.  All
 * rights reserved.
 *
 * Use ldap_compare() to compare values agains values contained in entry
 * “cn=Barbara Jensen, ou=Product Development, o=Ace Industry, c=US”.
 * We test to see if (1) the value “person” is one of the values in the
 * objectclass attribute (it is), and if (2) the value “xyzzy” is in the
 * objectlass attribute (it isn’t, or at least, it shouldn’t be).
 *
 */

#include “examples.h”

int
main( int main, char **argv )
{
    LDAP        *ld;
    int         rc;

    /* get a handle to an LDAP connection */
    if ( (ld = ldap_init( MY_HOST, MY_PORT )) == NULL ) {
        perror( “ldap_init” );
        return( 1 );
    }

    /* authenticate to the directory as nobody */
    if ( ldap_simple_bind_s( ld, NULL, NULL ) != LDAP_SUCCESS ) {
        ldap_perror( ld, “ldap_simple_bind_s” );
        return( 1 );
    }

    /* compare the value “person” against the objectclass attribute */
    rc = ldap_compare_s( ld, ENTRYDN, “objectclass”, “person” );
    switch ( rc ) {
    case LDAP_COMPARE_TRUE:
        printf( “The value \”person\” is contained in the objectclass “
                “attribute.\n” );
        break;
    case LDAP_COMPARE_FALSE:
        printf( “The value \”person\” is not contained in the objectclass “
                “attribute.\n” );
        break;
    default:
        ldap_perror( ld, “ldap_compare_s” );
    }

    /* compare the value “xyzzy” against the objectclass attribute */
    rc = ldap_compare_s( ld, ENTRYDN, “objectclass”, “xyzzy” );
    switch ( rc ) {
    case LDAP_COMPARE_TRUE:
        printf( “The value \”xyzzy\” is contained in the objectclass “
                “attribute.\n” );
        break;
    case LDAP_COMPARE_FALSE:
        printf( “The value \”xyzzy\” is not contained in the objectclass “
                “attribute.\n” );
        break;
    default:
        ldap_perror( ld, “ldap_compare_s” );
    }

    ldap_unbind( ld );
    return( 0 );
}
```

B.4.2 Compare Using JNDI

```
/*
 * Copyright (c) 1997.  Sun Microsystems.  All rights reserved.
 *
 * Use search() to compare values against values contained in entry
 * "cn=Barbara Jensen, ou=Product Development, o=Ace Industry, c=US".
 * We test to see if (1) the value "person" is one of the values in the
 * objectclass attribute (it is), and if (2) the value "xyzzy" is in the
 * objectlass attribute (it isn't, or at least, it shouldn't be).
 *
 * [equivalent to compare.c in Netscape SDK]
 *
 */

import java.util.Hashtable;

import javax.naming.*;
import javax.naming.directory.*;

class Compare {

public static void main(String[] args) {

    Hashtable env = new Hashtable(5, 0.75f);
    /*
     * Specify the initial context implementation to use.
     * This could also be set by using the -D option to the java program.
     * For example,
    *    java -Djava.naming.factory.initial=com.sun.jndi.ldap.LdapCtxFactory \
     *       Compare
     */
    env.put(Context.INITIAL_CONTEXT_FACTORY, Env.INITCTX);

    /* Specify host and port to use for directory service */
    env.put(Context.PROVIDER_URL, Env.MY_SERVICE);

    DirContext ctx = null;
    SearchControls ctls = new SearchControls();
    ctls.setSearchScope(SearchControls.OBJECT_SCOPE);
    ctls.setReturningAttributes(new String[0]);  // do not return any attrs

    try {
        /* get a handle to an Initial DirContext */
        ctx = new InitialDirContext(env);
    } catch (NamingException e) {
        System.err.println("Cannot get initial context.");
        return;
    }

    try {
        NamingEnumeration results =
            ctx.search(Env.ENTRYDN, "objectclass=person", ctls);

        if (results != null && results.hasMoreElements()) {
            System.out.println(
            "The value \"person\" is contained in the objectclass attribute.");
        } else {
            System.out.println(
        "The value \"person\" is not contained in the objectclass attribute." );
        }
    } catch (NamingException e) {
        System.err.println("Comparison of value person failed.");
    }

    try {
        NamingEnumeration results =
            ctx.search(Env.ENTRYDN, "objectclass=xyzzy", ctls);

        if (results != null && results.hasMoreElements()) {
            System.out.println(
             "The value \"xyzzy\" is contained in the objectclass attribute.");
        } else {
            System.out.println(
        "The value \"xyzzy\" is not contained in the objectclass attribute." );
        }
    } catch (NamingException e) {
        System.err.println("Comparison of value xyzzy failed.");
    }
}
}
```

B.5 Modify Attributes

B.5.1 Modify Attributes Using LDAP C API

```
/*
 * Copyright (c) 1996.  Netscape Communications Corporation.  All
 * rights reserved.
 *
 * Modify an entry:
 *  - replace any existing values in the “mail” attribute with “babs@ace.com”
 *  - add a new value to the “description” attribute
 */

#include “examples.h”

int
main( int argc, char **argv )
{
    LDAP        *ld;
    LDAPMod     mod0;
    LDAPMod     mod1;
    LDAPMod     *mods[ 3 ];
    char        *vals0[ 2 ];
    char        *vals1[ 2 ];
    time_t      now;
    char        buf[ 128 ];

    /* get a handle to an LDAP connection */
    if ( (ld = ldap_init( MY_HOST, MY_PORT )) == NULL ) {
        perror( “ldap_init” );
        return( 1 );
    }
    /* authenticate */
    if ( ldap_simple_bind_s( ld, ENTRYDN, ENTRYPW ) != LDAP_SUCCESS ) {
        ldap_perror( ld, “ldap_simple_bind_s” );
        return( 1 );
    }
    /* construct the list of modifications to make */
    mod0.mod_op = LDAP_MOD_REPLACE;
    mod0.mod_type = “mail”;
    vals0[0] = “babs@ace.com”;
    vals0[1] = NULL;
    mod0.mod_values = vals0;

    mod1.mod_op = LDAP_MOD_ADD;
    mod1.mod_type = “description”;
    time( &now );
    sprintf( buf, “This entry was modified with the modattrs program on %s”,
            ctime( &now ));
    /* Get rid of \n which ctime put on the end of the time string */
    if ( buf[ strlen( buf ) - 1 ] == ‘\n’ ) {
        buf[ strlen( buf ) - 1 ] = ‘\0’;
    }
    vals1[ 0 ] = buf;
    vals1[ 1 ] = NULL;
    mod1.mod_values = vals1;

    mods[ 0 ] = &mod0;
    mods[ 1 ] = &mod1;
    mods[ 2 ] = NULL;

    /* make the change */
    if ( ldap_modify_s( ld, ENTRYDN, mods )
            != LDAP_SUCCESS ) {
        ldap_perror( ld, “ldap_modify_s” );
        return( 1 );
    }
    ldap_unbind( ld );
    printf( “modification was successful\n” );
    return( 0 );
}
```

B.5.2 Modify Attributes Using JNDI

```
/*
 * Copyright (c) 1997.  Sun Microsystems.  All rights reserved.
 *
 * Modify an entry:
 *  - replace any existing values in the "mail" attribute with "babs@ace.com"
 *  - add a new value to the "description" attribute
 *
 * [equivalent to modattrs.c in Netscape SDK]
 */

import java.util.Hashtable;
import java.util.Date;

import javax.naming.*;
import javax.naming.directory.*;

class Modattrs {
```

```
public static void main(String[] args) {

    Hashtable env = new Hashtable(5, 0.75f);
    /*
     * Specify the initial context implementation to use.
     * This could also be set by using the -D option to the java program.
     * For example,
     *   java -Djava.naming.factory.initial=com.sun.jndi.ldap.LdapCtxFactory \
     *       Modattrs
     */
    env.put(Context.INITIAL_CONTEXT_FACTORY, Env.INITCTX);

    /* Specify host and port to use for directory service */
    env.put(Context.PROVIDER_URL, Env.MY_SERVICE);

    /* specify authentication information */
    env.put(Context.SECURITY_AUTHENTICATION, "simple");
    env.put(Context.SECURITY_PRINCIPAL, Env.MGR_DN);
    env.put(Context.SECURITY_CREDENTIALS, Env.MGR_PW);

    try {
        /* get a handle to an Initial DirContext */
        DirContext ctx = new InitialDirContext(env);

        /* construct the list of modifications to make */
        ModificationItem[] mods = new ModificationItem[2];

        Attribute mod0 = new BasicAttribute("mail", "babs@eng");
        // Update mail attribute
        mods[0] = new ModificationItem(DirContext.REPLACE_ATTRIBUTE, mod0);

        // Add another value to description attribute
        Attribute mod1 =
            new BasicAttribute("description",
                      "This entry was modified with the Modattrs program on " +
                          (new Date()).toString());
        mods[1] = new ModificationItem(DirContext.ADD_ATTRIBUTE, mod1);

         /* Delete the description attribute altogether */
        /*
        Attribute mod1 =  new BasicAttribute("description");
        mods[2] = new ModificationItem(DirContext.REMOVE_ATTRIBUTE, mod1);
     */

        /* make the change */
        ctx.modifyAttributes(Env.ENTRYDN, mods);
        System.out.println( "modification was successful." );

    } catch (NamingException e) {
        System.err.println("modification failed. " + e);
    }
}
}
```

B.6 Rename An Entry

B.6.1 Rename Using LDAP C API

```
/*
 * Copyright (c) 1996.  Netscape Communications Corporation.  All
 * rights reserved.
 *
 * Modify the RDN (relative distinguished name) of an entry.  In this
 * example, we change the dn “cn=Jacques Smith, o=Ace Industry, c=US”
 * to “cn=Jacques M Smith, o=Ace Industry, c=US”.
 *
 * Since it is an error to either (1) attempt to modrdn an entry which
 * does not exist, or (2) modrdn an entry where the destination name
 * already exists, we take some steps, for this example, to make sure
 * we’ll succeed.  We (1) add “cn=Jacques Smith” (if the entry exists,
 * we just ignore the error, and (2) delete “cn=Jacques M Smith” (if the
 * entry doesn’t exist, we ignore the error).
 *
 * We pass 0 for the “deleteoldrdn” argument to ldap_modrdn2_s().  This
 * means that after we change the RDN, the server will put the value
 * “Jacques Smith” into the cn attribute of the new entry, in addition to
 * “Jacques M Smith”.
 */

#include “examples.h”

#define        NMODS        4

unsigned long        global_counter = 0;

static void free_mods( LDAPMod **mods );

int
main( int argc, char **argv )
{
    LDAP                   *ld;
    char                   *dn, *ndn, *nrdn;
    int                    i;
    int                    rc;
    LDAPMod                **mods;

    /* Values we’ll use in creating the entry */
    char *objectclass_values[] = { “top”, “person”, “organizationalPerson”,
                                   “inetOrgPerson”, NULL };
    char *cn_values[] = { “Jacques Smith”, NULL };
    char *sn_values[] = { “Smith”, NULL };
    char *givenname_values[] = { “Jacques”, NULL };

    /* Specify the DN we’re adding */
    dn = “cn=Jacques Smith, o=Ace Industry, c=US”;
    /* the destination DN */
    ndn = “cn=Jacques M Smith, o=Ace Industry, c=US”;
    /* the new RDN */
    nrdn = “cn=Jacques M Smith”;

    /* get a handle to an LDAP connection */
    if ( (ld = ldap_init( MY_HOST, MY_PORT )) == NULL ) {
        perror( “ldap_init” );
        return( 1 );
    }
```

```
/* authenticate to the directory as the Directory Manager */
if ( ldap_simple_bind_s( ld, MGR_DN, MGR_PW ) != LDAP_SUCCESS ) {
    ldap_perror( ld, “ldap_simple_bind_s” );
    return( 1 );
}

if (( mods = ( LDAPMod ** ) malloc(( NMODS + 1 ) * sizeof( LDAPMod *)))
        == NULL ) {
    fprintf( stderr, “Cannot allocate memory for mods array\n” );
    return( 1 );
}
/* Construct the array of values to add */
for ( i = 0; i < NMODS; i++ ) {
    if (( mods[ i ] = ( LDAPMod * ) malloc( sizeof( LDAPMod ))) == NULL ) {
        fprintf( stderr, “Cannot allocate memory for mods element\n” );
        return( 1 );
    }
}
mods[ 0 ]->mod_op = 0;
mods[ 0 ]->mod_type = “objectclass”;
mods[ 0 ]->mod_values = objectclass_values;
mods[ 1 ]->mod_op = 0;
mods[ 1 ]->mod_type = “cn”;
mods[ 1 ]->mod_values = cn_values;
mods[ 2 ]->mod_op = 0;
mods[ 2 ]->mod_type = “sn”;
mods[ 2 ]->mod_values = sn_values;
mods[ 3 ]->mod_op = 0;
mods[ 3 ]->mod_type = “givenname”;
mods[ 3 ]->mod_values = givenname_values;
mods[ 4 ] = NULL;


/* Add the entry */
if (( rc = ldap_add_s( ld, dn, mods )) != LDAP_SUCCESS ) {
    /* If entry exists already, fine.  Ignore this error. */
    if ( rc == LDAP_ALREADY_EXISTS ) {
        printf( “Entry \”%s is already in the directory.\n”, dn );
    } else {
        ldap_perror( ld, “ldap_add_s” );
        free_mods( mods );
        return( 1 );
    }
} else {
    printf( “Added entry \”%s\”.\n”, dn );
}
free_mods( mods );

/* Delete the destination entry, for this example */
if (( rc = ldap_delete_s( ld, ndn )) != LDAP_SUCCESS ) {
    /* If entry does not exist, fine.  Ignore this error. */
    if ( rc == LDAP_NO_SUCH_OBJECT ) {
        printf( “Entry \”%s\” is not in the directory.  “
                “No need to delete.\n”, ndn );
    } else {
        ldap_perror( ld, “ldap_delete_s” );
        return( 1 );
    }
} else {
```

```
            printf( “Deleted entry \”%s\”.\n”, ndn );
        }

        /* Do the modrdn operation */
        if ( ldap_modrdn2_s( ld, dn, nrdn, 0 ) != LDAP_SUCCESS ) {
            ldap_perror( ld, “ldap_modrdn2_s” );
            return( 1 );
        }

        printf( “The modrdn operation was successful.  Entry\n”
                “\”%s\” has been changed to\n”
                “\”%s\”.\n”, dn, ndn );

        ldap_unbind( ld );
        return 0;
    }



    /*
     * Free a mods array.
     */
    static void
    free_mods( LDAPMod **mods )
    {
        int i;

        for ( i = 0; i < NMODS; i++ ) {
            free( mods[ i ] );
        }
        free( mods );
    }
```

B.6.2 Rename Using JNDI

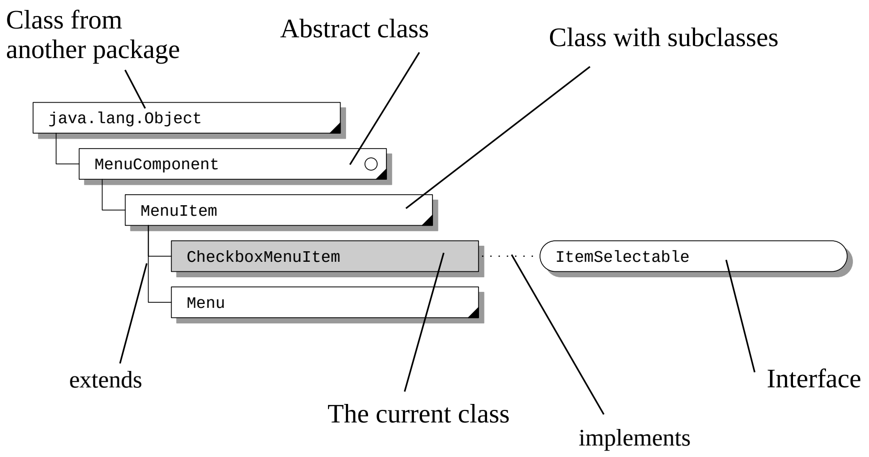
```
/*
 * Copyright (c) 1997.  Sun Microsystems.  All rights reserved.
 *
 * Modify the RDN (relative distinguished name) of an entry.  In this
 * example, we change the dn "cn=Jacques Smith, o=Ace Industry, c=US"
 * to "cn=Jacques M Smith, o=Ace Industry, c=US".
 *
 * Since it is an error to either (1) attempt to modrdn an entry which
 * does not exist, or (2) modrdn an entry where the destination name
 * already exists, we take some steps, for this example, to make sure
 * we'll succeed.  We (1) add "cn=Jacques Smith" (if the entry exists,
 * we just ignore the error, and (2) delete "cn=Jacques M Smith" (if the
 * entry doesn't exist, we ignore the error).
 *
 * After renaming, we add back the attribute "Jacques Smith" into the cn
 * attribute.
 *
 * [based on modrdn.c of Netscape  SDK]
 */

import java.util.Hashtable;
import java.util.Date;
import javax.naming.*;
import javax.naming.directory.*;

class Modrdn {

public static void main(String[] args) {

    /* Values we'll use in creating the entry */
    Attribute objClasses = new BasicAttribute("objectclass");
    objClasses.add("top");
    objClasses.add("person");
    objClasses.add("organizationalPerson");
    objClasses.add("inetOrgPerson");

    Attribute cn = new BasicAttribute("cn", "Jacques Smith");
    Attribute sn = new BasicAttribute("sn", "Smith");
    Attribute givenNames = new BasicAttribute("givenname", "Jacques");

    /* Specify the DN we're adding */
    String dn = "cn=Jacques Smith, " + Env.MY_MODBASE;
    /* the destination DN */
    String ndn = "cn=Jacques M Smith, " + Env.MY_MODBASE;
    /* the new RDN */
    String nrdn = "cn=Jacques M Smith";

    Hashtable env = new Hashtable(5, 0.75f);
    /*
     * Specify the initial context implementation to use.
     * This could also be set by using the -D option to the java program.
     * For example,
     *   java -Djava.naming.factory.initial=com.sun.jndi.ldap.LdapCtxFactory \
     *       Modrdn
     */
    env.put(Context.INITIAL_CONTEXT_FACTORY, Env.INITCTX);

    /* Specify host and port to use for directory service */
    env.put(Context.PROVIDER_URL, Env.MY_SERVICE);

    /* specify authentication information */
    env.put(Context.SECURITY_AUTHENTICATION, "simple");
    env.put(Context.SECURITY_PRINCIPAL, Env.MGR_DN);
    env.put(Context.SECURITY_CREDENTIALS, Env.MGR_PW);

    DirContext ctx = null;

    try {
        /* get a handle to an Initial DirContext */
        ctx = new InitialDirContext(env);
        Attributes orig = new BasicAttributes();
        orig.put(objClasses);
        orig.put(cn);
        orig.put(sn);
        orig.put(givenNames);

        /* Add the entry */
        ctx.createSubcontext(dn, orig);
        System.out.println( "Added entry " + dn + ".");

    } catch (NameAlreadyBoundException e) {
        /* If entry exists already, fine.  Ignore this error. */
       System.out.println("Entry " + dn + " already exists, no need to add");
    } catch (NamingException e) {
        System.err.println("Modrdn: problem adding entry." + e);
        System.exit(1);
    }

    try {
        /* Delete the destination entry, for this example */
        ctx.destroySubcontext(ndn);
        System.out.println( "Deleted entry " + ndn + "." );

    } catch (NameNotFoundException e) {
        /* If entry does not exist, fine.  Ignore this error. */
        System.out.println( "Entry " + ndn + " is not in the directory.  " +
                    "No need to delete.");
    } catch (NamingException e) {
        System.err.println("Modrdn: problem deleting entry." + e);
        System.exit(1);
    }


    /* Do the modrdn operation */
    try {
        ctx.rename(dn, ndn);
        System.out.println("The modrdn operation was successful.  Entry " +
                           dn + " has been changed to " + ndn + ".");
    } catch (NamingException e) {
        System.err.println("Modify operation failed." + e);
    }
}
}
```

Appendix C: Legend for Class Diagram

In a class diagram, we visually distinguish the different kinds of Java entities, as follows:

1. The interface: A rounded rectangle
2. The class: A rectangle
3. The abstract class: A rectangle with an empty dot
4. The final class: A rectangle with a black dot
5. Classes with subclasses: A rectangle with a small black triangle on the lower right corner

Most of these elements are shown below. The class or interface being described in the current chapter is shaded grey (this is not applicable for package class diagrams). A solid line represents extends, while a dotted line represents implements.



Appendix D: JNDI Change History

1.2: JNDI Changes Since 1.1

- Added `javax.naming.event` package.
- Added `javax.naming.ldap` package.
- Added support for configuration using resource files. See Chapter 6 of the **JNDI API** document.

API-related Changes

- Added `NamingEnumeration.close()` for cancelling or terminating enumerations.
- Added `ReferralException.getReferralContext(Hashtable env)` and `ReferralException.retryReferral()` to allow creation and retry of referral context with different environment properties.
- Clarified how context methods that accept `Name` argument should deal with `CompositeName` and non`CompositeName` arguments. Specifically, instances of `CompositeName` are treated as composite name, while all others are treated as compound name.
- Added `Context.getNameInNamespace()` for retrieving the full name of a context within its own namespace.
- Clarified definition of the class factory location of a `Reference` object. Specifically, the location is a codebase, which consists of a list space-separated URLs.
- Added support for ordered multivalued attributes to `Attribute` and `BasicAttribute`.
- Added `BasicAttributes.equals()` and `BasicAttributes.hashCode()`.
- Redefined semantics of `DirContext.getSchemaClassDefinition()` so that it returns a context that contains the `DirContext` objects of class definitions, rather than returning one (arbitrary) class definition.
- Added protected `InitialContext`/`InitialDirContext` constructors to allow lazy initialization. Useful for subclass implementations.

SPI-related Changes

- Added `StateFactory`/`NamingManager.getStateToBind()`, analogous to `ObjectFactory`/`NamingManager.getObjectInstance()`, for transforming an object's state before the object is bound in the naming/directory service.
- Added interfaces `DirObjectFactory` and `DirStateFactory` to better support service providers that implement the `DirContext` interface. Added `DirectoryManager.getObjectInstance()` and `DirectoryManager.getStateToStore()` to use these interfaces.
- Refined definition of `NamingManager.getObjectInstance()` to not treat URL strings specially. Instead, the URL should be wrapped inside a `Reference` whose `RefAddr` type is "URL".
- Made `ResolveResult` implement `Serializable`.
- Defined a special form of `Reference` called a next naming system (nns) reference for supporting "dynamic federation". This reference has a `RefAddr` type of "nns" and a content consisting of the resolved object.
- Added the string constant `NamingManager.CPE` which names a property set by `NamingManager.getContinuationContext()`/`DirectoryManager.getContinuationContext()`. The value of this property is an instance of `CannotProceedException`. This is useful to service providers that implement federation by chaining the CPEs.
- Defined a convention for service providers to use when naming environment properties. See Section 6.1.