Sun microsystems

JavaSoft

JNDI SPI: Java™ Naming and Directory Service Provider Interface

The Java Naming and Directory service provider interface (JNDI SPI).

Please send comments to jndi@java.sun.com

Contents

1 JNDI Service Provider Interface (SPI)

The JNDI SPI provides the means by which different naming and directory service providers can develop and hook up their implementations so that the corresponding services are accessible from applications that use JNDI. In addition, because JNDI allows the use of names that span multiple namespaces, one service provider implementation may need to interact with another in order to complete an operation. The SPI provides methods that allow different provider implementations to cooperate to complete client JNDI operations.

This document describes the components of the SPI and explains how developers can build service providers for JNDI. It is assumed that the reader is familiar with the contents of the **JNDI API** document.

All service provider developers should read the "Security Considerations" section of the **JNDI API** document. It contains important issues that all developers using JNDI, especially those writing service providers, should consider.

2 Implementing the Context Interface

One of the basic tasks in building a context implementation is to define a class that implements the `Context` (or `DirContext`) interface. The following guidelines should be used for developing this class.

2.1 Basic Support

The provider defines implementations for each of the methods in the `Context` interface.

If a method is not supported, it should throw `OperationNotSupportedException`.

For methods in the `Context` or `DirContext` interfaces that accept a name argument (either as a `String` or a `Name`), an empty name denotes the current context. For example, if an empty name is supplied to `lookup()`, that means to return the current context. If an empty name is supplied to `list()`, that means to enumerate the names in the current context. If an empty name is supplied to `getAttributes()`, that means to retrieve the attributes associated with this context.

Appendix A contains an example service provider that implements a flat, in-memory namespace.

2.2 Federation Support

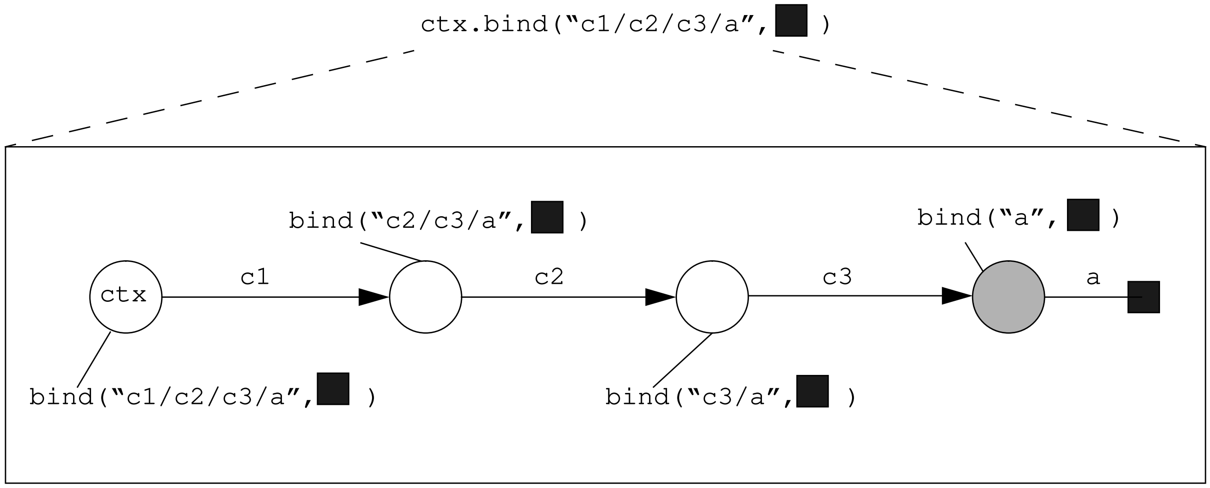
2.2.1 Names

A context in a federation will be given a composite name with each context operation. This composite name may span multiple namespaces, or it may have only a single compound name component (which in turn may be made up of one or several atomic names) that belongs to a single namespace. The context implementation must determine which part of the name is to be resolved/processed in its context and pass the rest onto the next context. This may be done by syntactically examining the name, or dynamically by resolving the name.

2.2.2 Resolving Through a Context

A context participates in a federation by performing the resolution phase of all of the context operations. The `lookup()` method must always be supported. Support for other methods is optional, but if the context is to participate in a federation, then the resolution implicit in all operations must be supported.

Figure 1: Example of Resolving through Intermediate Contexts to Perform a bind().



For example, suppose a context does not support the `bind()` operation. When that context is being used as an intermediate context for `bind()`, it must perform the resolution part of that operation to enable the operation to continue to the next context. It should only throw `OperationNotSupportedException` if it is being asked to create a binding in its own context. Figure 1 shows an example of how the `bind()` operation is passed through intermediate contexts to be performed in the target context.

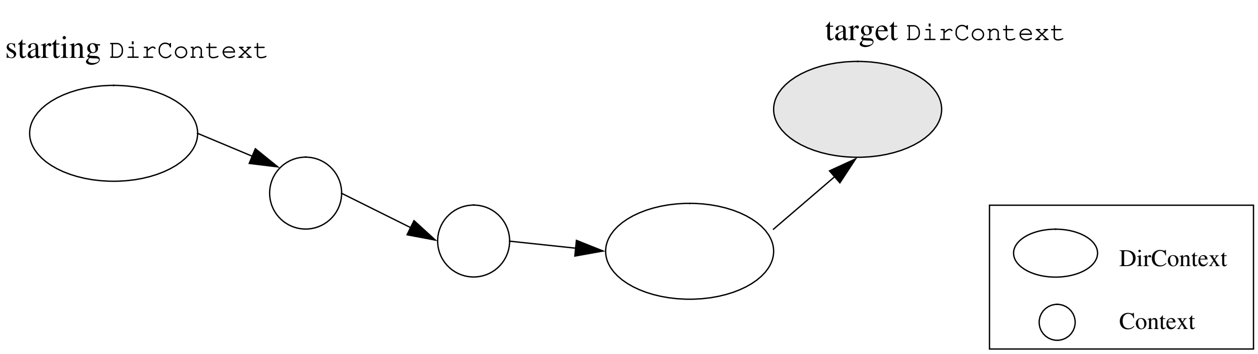
2.2.3 Resolving Through to Subinterfaces of Context

To invoke a `DirContext` method (such as `getAttributes()`), the application first obtains an initial `DirContext`, and then perform the operation on the `DirContext`.

```
DirContext ctx = new InitialDirContext();
Attributes attrs = ctx.getAttributes(someName);
```

From the provider's perspective, in order to retrieve the attributes, `getAttributes()` might need to traverse multiple naming systems. Some of these naming systems only support the `Context` interface, not the `DirContext` interface. These naming systems are being used as intermediaries for resolving towards the target context. The target context must support the `DirContext` interface. Figure 2 shows an example of this.

Figure 2: Example of Resolving Through Intermediate non-DirContexts



In order for intermediate naming systems to participate in the federation for extensions of `Context`, they must implement the `Resolver` interface. The `Resolver` interface is used by the JNDI framework to resolve through intermediate contexts that do not support a particular subinterface of `Context`. It consists of two overloaded forms of the method `resolveToClass()`. This method is used to partially resolve a name, stopping at the first context that is an instance of the required subinterface. By providing support for this method and the resolution phase of all methods in the `Context` interface, a provider can act as an intermediate context for extensions (subinterfaces) of `Context`.

```
public interface Resolver {
    public ResolveResult resolveToClass(Name name, Class contextType)
        throws NamingException;
    public ResolveResult resolveToClass(String name,
                                        Class contextType)
        throws NamingException;
}
```

2.2.4 Continuing an Operation in a Federation

In performing an operation on a name that spans multiple namespaces, a context that is acting as an intermediate context in an intermediate naming system needs to pass the operation onto the next naming system. The context does this by first constructing a `CannotProceedException` containing information pinpointing how far it has proceeded. In so doing it sets the resolved object, resolved name, remaining name, and environment parts of the exception.[1] (In the case of the `Context.rename()` method, it also sets the "resolved newname" part.)

It then obtains a *continuation context* from JNDI by passing the `CannotProceedException` to static method `NamingManager.getContinuationContext()`

```
public class NamingManager {
    public static Context getContinuationContext(
        CannotProceedException e) throws NamingException;
    ...
}
```

The information in the exception is used by `getContinuationContext()` to create the continuation context instance in which to continue the operation.

To obtain a continuation context for the `DirContext` operations, use `DirectoryManager.getContinuationDirContext()`.

```
public class DirectoryManager {
    public static getContinuationDirContext(
        CannotProceedException e) throws NamingException;
    ...
}
```

Upon receiving the continuation context, the operation should be continued using the remainder of the name that has not been resolved.

1. The `CannotProceedException` may well have been thrown by one of the context's internal methods when it discovered that the name being processed is beyond the scope of the context. The process by which the exception is produced is dependent on the implementation of the context.

For example, when attempting to continue a `bind()` operation, the code in the provider might look as follows:

```
public void bind(Name name, Object obj) throws NamingException {
    ...
    try {
        internal_bind(name, obj);
        ...
    } catch (CannotProceedException e) {
        Context cctx = NamingManager.getContinuationContext(e);
        cctx.bind(e.getRemainingName(), obj);
    }
}
```

In this example, `bind()` depends on an internal method, `internal_bind()`, to carry out the actual work of the bind and to throw a `CannotProceedException` when it discovers that it is going beyond this naming system. The exception is then passed to `getContinuationContext()` in order to continue the operation. If the operation cannot be continued, the continuation context will throw the `CannotProceedException` to the caller of the original `bind()` operation.

2.3 Referrals

LDAP-style directory services support the notion of *referrals* for redirecting a client's request to another server. A referral differs from the federation continuation mechanism described above in that a referral may be presented to the JNDI client, who then decides whether to follow it, whereas a `CannotProceedException` should be returned to the client only when no further progress is possible. Another difference is that an individual service provider offers the capability of continuing the operation (and itself determines the mechanism for doing so). In a federation, the mechanism of continuation is beyond the scope of individual service providers: individual providers benefit from the common federation mechanism provided by the JNDI SPI.

A service provider that supports referrals defines a subclass of `ReferralException` and provides implementations for its two abstract methods. `getReferralContext()` returns a context at which to carry on the operation, and `getReferralInfo()` returns information on where the referral leads to, in a format appropriate to the service provider.

The environment property `java.naming.referral` specifies how the service provider should treat referrals. If the service provider is asked to throw an exception when a referral is encountered, or if the provider encounters problems following a referral, it throws a `ReferralException` to the application. To continue the operation, the application re-invokes the method on the referral context using the same arguments it supplied to the original method. The following code sample shows how `ReferralException` may be used by an application:[1]

1. Note that this is code in the *application*. In "Continuing an Operation in a Federation", the code sample presented is code in the *service provider*.

```
while (true) {
    try {
        bindings = ctx.listBindings(name);
        while (bindings.hasMore()) {
            b = (Binding) bindings.next();
            ...
        }
        break;
    } catch (ReferralException e) {
        ctx = e.getReferralContext();
    }
}
```

This convention of re-invoking the method using the original arguments is a simple one for applications to follow. This places the burden on the implementation of the `ReferralException` to supply enough information to the implementation of the referral context for the operation to be continued. Note that this will likely render some of the arguments passed to the re-invoked operation superfluous. The referral context implementation is free to ignore any redundant or unneeded information.

It is possible for an operation to return results in addition to a referral. For example, when searching a context, the server might return several results in addition to a few referrals as to where to obtain further results. These results and referrals might be interleaved at the protocol level. If referrals require user interaction (i.e. not followed automatically), the service provider should return the results through the search enumeration first. When the results have been returned, the referral exception can then be thrown. This allows a simple programming model to be used when presenting the user with a clear relationship between a referral and its set of results.

2.4 Schema Support

JNDI defines the `Attribute` class for representing an attribute in a directory. An attribute consists of an attribute identifier (a string) and a set of attribute values, which can be any object in the Java programming language. There are also methods defined in the `Attribute` class for obtaining the attribute's definition and syntax definition from the directory's schema.

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

The default implementation of `Attribute` does not provide real implementations for these methods. A directory provider that has support for such schema information should provide subclasses of `Attribute` that implement these two methods based on its schema mechanisms. The provider should then return instances of these subclasses when asked to return instances of `Attribute`. The provider, when it receives an unextended `Attribute` instance, should use reasonable defaults to determine the attribute's definition and syntax, using information such as the attribute values' class names or conventions used for the attribute identifier.

The `DirContext` interface contains schema-related methods:

```
public class DirContext {
    ...
    public DirContext getSchema(Name name) throws NamingException;
    public DirContext getSchema(String name) throws NamingException;

    public DirContext getSchemaClassDefinition(Name name)
        throws NamingException;
    public DirContext getSchemaClassDefinition(String name)
        throws NamingException;

}
```

`getSchema()` returns the schema tree for the named object, while `getSchemaClassDefinition()` returns the schema class definition for the named object. Some systems have just one global schema and, regardless of the value of the `name` argument, will return the same schema tree. Others support finer grained schema definitions, and may return different schema trees depending on which context is being examined.

2.5 Support for Objects in the Java Programming Language

JNDI encourages providers to supply implementations of the `Context` and `DirContext` interfaces that are natural and intuitive for the Java application programmer. For example, when looking up a printer name in the namespace, it is natural for the programmer to expect to get back a printer object on which to operate.

```
Context ctx = new InitialContext();
Printer prt = (Printer)ctx.lookup(somePrinterName);
prt.print(someStreamOfData);
```

However, what is bound in the underlying directory or naming services typically are not objects in the Java programming language but merely reference information which can be used to locate or access the actual object. This case is quite common, especially for Java applications accessing and sharing services in an existing installed base. The reference in effect acts as a "pointer" to the real object. In the printer example, what is actually bound might be information on how to access the printer (e.g. its protocol type, its server address). To enable this easy-to-use model for the application developer, the provider must do the transformation of the data stored in the underlying service into the appropriate objects in the Java programming language.

There are different ways to achieve this goal. One provider might have access to all the implementation classes of objects that a directory can return; another provider might have a special class loader for locating implementation classes for its objects. JNDI supports automatic generation of objects using information bound in the namespace via the use of the `Reference` class (see "References and Referenceable" on page 17) and URLs (see "URL Context Factory" on page 16). By providing 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 invent separate mechanisms on their own. However, this does not preclude providers from using their own mechanisms for achieving the same goal.

To enable this feature in their contexts, the service provider can use the `getObjectInstance()` method from `NamingManager` to convert information bound in the namespace into objects.

```
Object NamingManager.getObjectInstance(Object refInfo,
                                        Name name,
                                        Context nameCtx,
                                        Hashtable env);
```

For example, suppose printers are represented in the namespace using `References`. To turn a printer `Reference` into a live `Printer` object, the service provider would use the `getObjectInstance()` method. In this way, the underlying service need not know anything specific about printers.

```
Object lookup(Name name) {
    ...
    Reference ref = <some printer reference looked up from directory>;
    return (NamingManager.getObjectInstance(ref, name, this, env));
}
```

When constructing objects to be returned for the following JNDI methods, the service provider should call `getObjectInstance()`, or its own mechanism for generating objects from the bound information, if it wants this feature to be enabled in their contexts.

```
javax.naming.Context.lookup()
javax.naming.Context.lookupLink()
javax.naming.Binding.getObject()
javax.naming.directory.SearchResult.getObject()
```

For `Binding` and `SearchResult`, the provider should either pass an object that is the result of calling `getObjectInstance()` or its equivalent to the constructor, or override the default implementation of `Binding` and `SearchResult` so that their `getObject()` implementations call `getObjectInstance()` or its equivalent before returning.

2.6 Context Environment Support

Each instance of `Context` (or `DirContext`) can have associated with it an *environment* which contains preferences expressed by the application of how it would like to access the services offered by the context. Examples of information found in an environment are security-related information that specify the user's credentials and desired level of security (none, simple, strong), and configuration information, such as the server to use. Appendix A of the **JNDI API** document specifies a preliminary list of environment properties.

Environment properties are defined generically in order to ensure maximum portability. Individual service providers should map these generic properties to characteristics appropriate for their service. Properties that are not relevant to a provider are silently ignored. The environment may also be used for storing service-specific properties or preferences, in which case their applicability across different providers is limited.

2.6.1 Initializing a Context's Environment

When creating an initial context (either `InitialContext` or `InitialDirContext`), the application can supply an environment as a parameter. The parameter is represented as a `Hashtable` or any of its subclasses (e.g. `Properties`). The service provider should make a copy of the contents of the environment so that changes by the caller to the argument would not affect what the provider sees and vice versa. Note also that if the environment argument is a `Properties` instance, enumeration and `Hashtable.get()` on the argument only examine the top-level properties (not any nested defaults). This is the expected behavior. The provider is not expected to retrieve or enumerate values in the `Properties` instance's nested defaults.

2.6.2 Inheritance

The environment is inherited from parent to child as the context methods proceed from one context to the next. The entire environment of a context instance is inherited by the child context instances, regardless of whether certain properties within the environment are ignored by a particular context.

A service provider must pass on the environment from one context instance to the next in order to implement this "inheritance" trait of environments. Within one provider it can do so by passing the environment as an argument to the `Context` constructor, or to the `NamingManager.getObjectInstance()` method for creating `Context` instances.

Across providers in a federation, this is supported by passing the environment as part of the `CannotProceedException` parameter of the `NamingManager.getContinuationContext()` method, which in turn will use this environment when creating an instance of the context in which to continue the operation.

Inheritance can be implemented in any way as long as it preserves the semantics that each context has its own view of its environment. For example, a copy-on-write implementation could be used to defer copying of the environment until it is absolutely necessary.

2.6.3 Updates to the Environment

The environment of a context can be updated via the use of the `addToEnvironment() and removeFromEnvironment()` methods in the `Context` interface.

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

    public Object removeFromEnvironment(String propName)
        throws NamingException;
}
```

These methods update the environment of this instance of `Context`. An environment property that is not relevant to the provider is silently ignored but maintained as part of the environment. The updated environment affects this instance of `Context`, and will be inherited by any new child `Context` instances, but does not affect any `Context` instances already in existence. A lookup of the empty name on a `Context` will return a new `Context` instance with an environment inherited as with any other child.

3 The Initial Context

Since all JNDI methods are performed relative to a context, an application needs a starting context in order to invoke JNDI methods. This starting context is referred to as the *initial context*. The bindings in the initial context are determined by policies set forth by the initial context service provider, perhaps using standard policies for naming global and enterprise-wide namespaces. For example, the initial context might contain a binding to the Internet DNS namespace, a binding to the enterprise-wide namespace, and a binding to a personal directory belonging to the user who is running the application.

An application obtains an initial context by making the following call:

```
Context ctx = new InitialContext();
```

An alternate constructor allows an environment to be passed as an argument. This allows the application to pass in preferences or security information to be used in the construction of the initial context.

```
Hashtable env = new Hashtable(5, 0.75);[1]
env.put(Context.SECURITY_PRINCIPAL, "jsmith");
env.put(Context.SECURITY_CREDENTIALS, "xxxxxxx");
Context ctx = new InitialContext(env);
```

Subsequent to getting an initial context, the application can invoke `Context` methods.

```
Object obj = ctx.lookup("this/is/a/test");
```

The `InitialContext` class selects an actual initial context implementation using a default algorithm that can be overridden by installing an *initial context factory builder* (described below).

The `InitialDirContext` is an extension of `InitialContext`. It is used for performing directory operations using the initial context. The algorithms and policies described in this section also apply to `InitialDirContext`. Places where `DirContext` is required instead of `Context` have been noted.

3.1 Implementing An Initial Context

An initial context implements the `Context` or `DirContext` interface. Its implementation should follow the same guidelines outlined in “Implementing the Context Interface” on page 2.

In addition to the implementation classes for `Context` and/or `DirContext`, the provider must also supply an implementation for `InitialContextFactory`, which is responsible for generating instances of the initial context. `InitialContextFactory` contains a single method, `getInitialContext()`.

1. You can also use a subclass of `Hashtable` (e.g. `Properties`) for this.

```
public interface InitialContextFactory {
    public Context getInitialContext(Hashtable env)
        throws NamingException;
}
```

This method generates instances of `Context` or `DirContext` that serve as initial contexts. The implementation class for `InitialContextFactory` must be public and contain a public constructor that does not take any arguments. Appendix A contains an example of an `InitialContextFactory`.

3.2 Making An Initial Context Implementation Available to JNDI

There are three ways in which an initial context implementation is made available to JNDI:

- The `java.naming.factory.initial` environment or system property.
- URL Context Implementations.
- An initial context implementation factory.

3.2.1 The java.naming.factory.initial Property

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

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

3.2.2 URL Context Implementations

If a URL string[1] is passed to the initial context, it will be resolved using the corresponding URL context implementation. This is independent of any initial context implementations obtained using the `java.naming.factory.initial` environment or system property.

The URL context implementation is obtained using an object factory for the URL scheme identified in the URL string. The factory's class name is of the form *urlScheme*`URLContextFactory` in the package specified using the `java.naming.factory.url.pkgs` environment or system property. `java.naming.factory.url.pkgs` contains a colon-separated list of package prefixes. Each package prefix in this property is tried in the order specified to load the factory class. If none of the prefixes work, the default package prefix `com.sun.jndi.url` is tried. The factory's fully qualified class name is constructed using the following rule:

1. The mention of "URL" in this document refers to a URL string as defined by RFC 1738 and its related RFCs. It is any string that conforms to the syntax described therein, and may not always have corresponding support in the `java.net.URL` class or Web browsers. The URL string is either passed as the `String` name parameter, or as the first component of the `Name` parameter.

package prefix + "." + URL scheme + "." + class name

For example, if the *urlScheme* is "`ldap`" and `java.naming.factory.url.pkgs` contains "`com.widget:com.wiz.jndi`", JNDI will attempt to locate the corresponding object factory class by loading the following classes until one is successfully instantiated:

```
com.widget.ldap.ldapURLContextFactory
com.wiz.jndi.ldap.ldapURLContextFactory
com.sun.jndi.url.ldap.ldapURLContextFactory
```

The object factory class implements the `ObjectFactory` interface (see "URL Context Factory" on page 16) and has a public constructor that takes no arguments. It provides a `getObjectInstance()` method, which will create instances of `Context` or `DirContext` for the URL scheme. These instances will then be used to carry out the originally intended `Context` or `DirContext` operation on the URL supplied to the initial context.

3.2.3 Initial Context Factory Builder

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

A service provider for an initial context factory builder must define a class that implements `InitialContextFactoryBuilder`. This class's `createInitialContextFactory()` method generates instances of `InitialContextFactory`.

An application that wants to use this factory must first install it.

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

3.3 Throwing Exceptions

When the environment or system property `java.naming.factory.initial` is set to a non-`null` value, the `InitialContext` and `InitialDirContext` constructors will invoke `NamingManager.getInitialContext()` using the environment properties supplied. If the initial context cannot be created, for example as a result of an authentication problem, the initial context factory can throw an exception to indicate this problem. Note however that it is up to the provider implementation *when* it verifies and indicates to users of the initial context any environment property- or connection- related problems. It can do so lazily—delaying until an operation is performed on the context, or eagerly, at the time the context is created.

3.4 Implementing a Subclass of InitialContext

When there is a need to provide an initial context that supports an interface that extends from `Context` or `DirContext`, the service provider should supply a subclass of `InitialContext` (or `InitialDirContext`). To add support for URLs in the same way `InitialContext` and `InitialDirContext` do, the subclass would use the protected methods available in `InitialContext` as follows.

For example, suppose `XXXContext` is a subinterface of `DirContext`. Its initial context implementation would define `getURLOrDefaultInitXXXCtx()` methods (for both `Name` and `String` parameters) that retrieve the real initial context to use.

```
public class InitialXXXContext extends InitialDirContext {
    ...

    protected XXXContext getURLOrDefaultInitXXXCtx(Name name)
        throws NamingException {
        Context answer = getURLOrDefaultInitCtx(name);
        if (!(answer instanceof XXXContext)) {
            throw new NoInitialContextException("Not an XXXContext");
        }
        return (XXXContext)answer;
    }
    // similar code for getURLOrDefaultInitXXXCtx(String name)
}
```

When providing implementations for the new methods in the `XXXContext` interface that accept a name argument, `getURLOrDefaultInitXXXCtx()` is used in the following way.

```
public Object XXXMethod1(Name name, ...) throws NamingException {
    return getURLOrDefaultInitXXXCtx(name).XXXMethod1(name, ...);
}
```

When providing implementations for the new methods in the `XXXContext` interface that do not have a name argument, use `InitialContext.getDefaultInitCtx()`.

```
protected XXXContext getDefaultInitXXXCtx() throws NamingException {
    Context answer = getDefaultInitCtx();
    if (!(answer instanceof XXXContext)) {
        throw new NoInitialContextException("Not an XXXContext");
    }
    return (XXXContext)answer;
}

public Object XXXMethod2(Args args) throws NamingException {
    return getDefaultInitXXXCtx().XXXMethod2(args);
}
```

The implementation would also provide appropriate constructors for the class.

Client programs that use this new initial context would look as follows.

```
import com.widget.jndi.InitialXXXContext;
...
XXXContext ctx = new InitialXXXContext(env);
Object obj = ctx.lookup(name);
ctx.XXXMethod1(name, ...);
```

3.4.1 Using the SPI to get the Initial Context

The client application can bypass the use of `InitialContext` and `InitialDirContext` by calling `javax.naming.spi.getInitialContext()` directly to return an arbitrary subclass of `Context`. This has the disadvantage of losing the URL support provided by `InitialContext`.

(The service provider can, of course, provide the URL support on its own.) This style of usage may be suitable for a client application that sets its own initial context factory builder.

```
import javax.naming.spi.*;
NamingManager.setInitialContextFactoryBuilder(myBuilder);
Context ctx = NamingManager.getInitialContext(env);
...
Object obj = ctx.lookup(name);
(XXXContext)ctx.XXXMethod1(name,...);
```

4 Objects Bound in the Namespace

A natural way for a printer client to use the JNDI namespace is to look up a printer name in the namespace and get back a printer object on which to perform printing methods.

```
Context ctx = new InitialContext();
Printer prt = (Printer)ctx.lookup(somePrinterName);
prt.print(someStreamOfData);
```

This is possible if the printer object is directly bound in the namespace. However, as mentioned earlier, there are many directories and naming services in which names are not bound directly to objects, but rather to information used to locate or communicate with the actual object. In the printer example, perhaps what is bound in the namespace is the address of the printer server. At the same time, we do not want the a directory or naming service implementation to know explicitly about printer addresses and printer objects and how to transform one into the other.

JNDI addresses the different ways in which information about objects can be stored and the desire to turn such information into objects that Java applications can use via the use of *object factories*.

4.1 Object Factories

JNDI provides a generic way for creating objects (including instances of `Context`) using information stored in the namespace. That information may be of arbitrary type (`Object`). For example it may be a `Reference`, or a URL, or any other data required to create the object. Turning such information stored in the namespace into an object is supported through the use of *object factories*. An object factory is a class that implements the `ObjectFactory` interface, which contains a single method:

```
public interface ObjectFactory {
    public Object getObjectInstance(Object refObj,
                                    Name name,
                                    Context nameCtx,
                                    Hashtable env)
        throws Exception;
}
```

Given some reference information (`refObj`), optional information about the name of the object and where it is bound, and optionally some additional environment information (for example, some identity or authentication information about the user creating the object), `getObjectInstance()` will create an instance of the object for which this factory is responsible. For example, for a printer object factory, `getObjectInstance()` would return instances of printers. If an object cannot be created using the arguments supplied, `getObjectInstance()` should return `null`. The `getObjectInstance()` method should only thrown an exception if no other object factories should subsequently be tried. Consequently, `getObjectInstance()` should be careful about runtime exceptions that might be thrown from its implementation.

4.1.1 Context Factory

A context factory is an object factory that creates instances of `Context`. The implementation of these contexts for a particular naming or directory service is referred to as a *service provider* or *context implementation*.

4.1.2 URL Context Factory

A URL context factory is a special kind of context factory. It follows these rules when implementing `ObjectFactory.getObjectInstance()`.

- If `refObj` is `null`, create a context for resolving URLs of the scheme associated with this factory. The resulting context is not tied to a specific URL. For example, invoking

  ```
  getObjectInstance(null, null, null, env)
  ```

  on an "ldap" URL context factory returns a context that can resolve LDAP URLs (e.g. "`ldap://ldap.wiz.com/o=wiz,c=us`" or "`ldap://ldap.umich.edu/o=umich,c=us`", ...).

- If `refObj` is a URL string, create the object identified by the URL. For example, invoking

  ```
  getObjectInstance("ldap://ldap.wiz.com/o=wiz,c=us", null, null, env);
  ```

  on an "ldap" URL context factory returns a context for resolving LDAP names (e.g. "`cn=Jane Smith`") relative to the context "`o=wiz,c=us`" on the LDAP server `ldap.wiz.com`.

- If `refObj` is an array of URL strings, the assumption is that the URLs are equivalent in terms of the context to which they refer. Verification of whether the URLs are, or need to be, equivalent is up to the context factory. The order of the URLs in the array is not significant. The object returned by `getObjectInstance()` is the same as that for the single URL case—it is an object (perhaps a context) named by the URLs.

URL context factories are used by the initial context when it is passed a URL to resolve. URL context factories are also used for creating objects in the Java programming language from URLs stored in the namespace (see "URLs as Reference Information" on page 19).

4.1.3 Making Object Factories Available to JNDI

The method `NamingManager.getObjectInstance()` is used to turn reference information into objects in the Java programming language. `NamingManager.getObjectInstance()` locates and instantiates an instance of `ObjectFactory` and invokes the `getObjectInstance()` method on the factory.

In addition to being a public method to be used by service providers to turn reference information into objects "Support for Objects in the Java Programming Language" on page 7), `NamingManager.getObjectInstance()` is also used internally (for example, in the implementation of `getURLContext()` and `getContinuationContext()`).

There are four ways in which object factories are made available to JNDI:

- Information in `Reference`,
- URLs as reference,

- Use of the `java.naming.factory.object` system property,
- Installation of an *object factory builder*.

4.2 References and Referenceable

JNDI defines a `Reference` class to provide a uniform way of representing reference information stored in the namespace. A `Reference` contains a list of addresses and class information about the object to which this reference refers. An object that has a `Reference` implements the `Referenceable` interface. The `Referenceable` interface contains a single method for retrieving the reference of the object.

```
public interface Referenceable {
    public Reference getReference() throws NamingException;
}
```

4.2.1 Storing References in the Namespace

When binding a `Referenceable` object in the namespace, the information bound is the `Reference` of the object. When the object is looked up, the `Reference` is used to create an instance of the corresponding object. The `bind()` and `lookup()` operations are inverses of each other with regard to how they treat references.

In the printer example, a particular implementation of `Printer`, say `BSDPrinter`, might have the following class declaration:

```
public class BSDPrinter implements Printer, Referenceable {
    String serverName;

    BSDPrinter(String srv) {
        ...
    }
    public void print(InputStream data) throws PrinterException {
    }
    public Reference getReference() throws NamingException {
        return new Reference("Printer",
                             new StringRefAddr("bsd", serverName));
    }
}
```

When this object is bound in the namespace, the service provider uses `getReference()` to retrieve the object's reference, in this case its protocol type ("`bsd`") and server name (the instance variable `serverName`), and stores this information in the namespace. When the reference is retrieved from the namespace, the object factory mechanism described in "Class information in Reference" is used to turn the reference into an instance of `BSDPrinter`.

It is not a requirement that all service providers use `Reference`. A service provider may bind other reference-like information in the namespace (such as a URL, or the serialized form of a serializable object), and use that information to create corresponding objects to be returned to applications. `Reference` was introduced so that different providers need not invent different ways of achieving the same result.

4.2.2 Class information in Reference

A `Reference` contains methods for returning the class name and location of the object factory. The following methods are found in `Reference`.

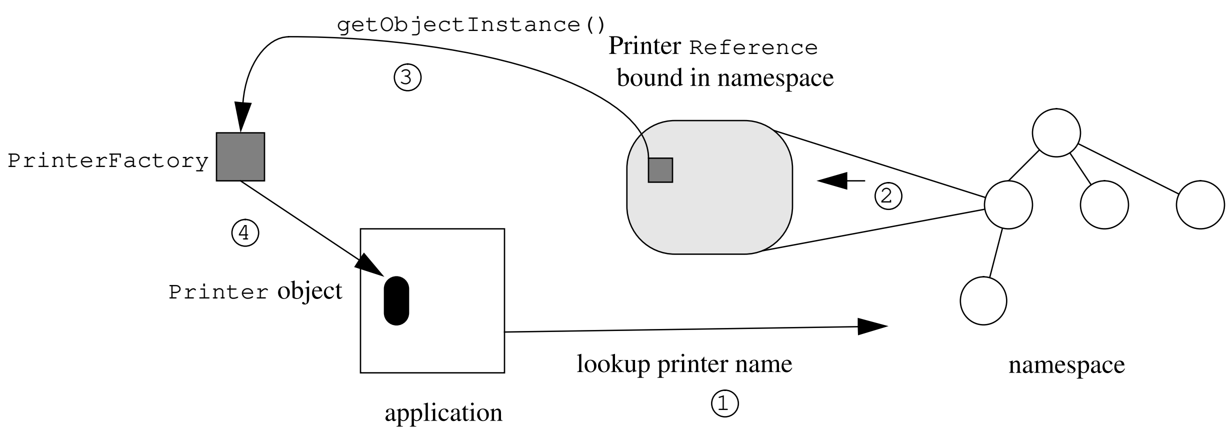
```
public class Reference {
    ...
    public String getClassName();
    public String getFactoryClassName();
    public String getFactoryClassLocation();
}
```

If the object is an instance of `Reference` or `Referenceable`, its corresponding object factory can be located using information in `Reference`. The `getFactoryClassName()` method retrieves the name of the factory class that implements the `ObjectFactory` interface. This factory must implement the `ObjectFactory` interface and have a public constructor that takes no arguments. `getFactoryClassLocation()` retrieves the location of the class implementation for the factory. This will typically be a URL of the factory's class file.

The object is created by invoking the `getObjectInstance()` method on the object factory instance with the `Reference` and environment as arguments. This creates an instance of a class identified by `getClassName()`.

Note that all the classes necessary to instantiate the object returned to the application are made available using mechanisms provided by JNDI. The application doesn't have to install the classes locally.

Figure 3: Example Using Reference to Get Back An Object From the Namespace



Returning to the printer example, `BSDPrinter` uses the `Reference` class to store information regarding how to construct instances of `BSDPrinter` and address information for communicating with the print server. The `Reference` contains the class name of the object ("`Printer`"), the class name of the printer object factory ("`PrinterFactory`") and a URL for loading the factory's class implementation. Using the factory class name and implementation location, the provider first loads the implementation of `PrinterFactory` and creates an instance of a

PrinterFactory. The provider then invokes getObjectInstance() on the factory to create an instance of Printer using the address information in the reference. For example, one address in the reference may have an address of type “bsd” containing the print server’s host name (“lobby-printserver”). The PrinterFactory uses the address type (“bsd”) to decide to create a BSDPrinter instance and passes the address contents (“lobby-printserver”) to its constructor. The resulting BSDPrinter object is returned as the result of lookup().

When the application invokes print() on the BSDPrinter instance returned by lookup(), the data is sent to the print server on the machine “lobby-printserver” for printing. The application need not know the details of the Reference stored in the namespace, the protocol used to perform the job, or whether the BSDPrinter class was defined locally or loaded over the network. The transformation of the information stored in the underlying service into an object that implements the Printer interface is done transparently through the cooperation of the service provider (which stores bindings of printer names to printer address information), the printer service provider (which provides the PrinterFactory and BSDPrinter classes), and the JNDI SPI framework (which ties the two together to return an object that the application can directly use).

4.3 URLs as Reference Information

When an object in the namespace is bound to a URL string, or an array of URL strings. the object factory is identified using the same mechanism used to identify the factory when a URL is passed to the initial context (see “URL Context Implementations” on page 11). If this mechanism does not successfully locate an object factory, the java.naming.factory.object property (described in the next section) is used.

For the printer example, instead of using a Reference to represent a printer in the namespace, a URL may be stored (perhaps something like “printer:bsd://lobby-printserver”). The NamingManager.getObjectInstance() method will look for and create the URL context factory class printerURLContextFactory. If successful, naming manager passes the URL to the factory to create a Printer instance.

Note that this approach differs from the Reference approach in that the classes for the URL context factory (printerURLContextFactory) must be available to the application (perhaps by appropriate setting of class paths). In the Reference approach, the classes for the factory are located dynamically.

4.4 Storing Serializable Objects

When an object being bound in the namespace is Serializable but not Referenceable, the service provider should if possible store the serialized form of the object. When the object is later looked up, the object should be deserialized and returned.

Note that for a service provider to store serialized objects it must be able to store binary data, and it must not have a data size limit too small for the serialized objects in question. Not all service providers meet these requirements. Note also that, as with the use of URLs for storing reference information, the dynamic class-loading facility of the Reference mechanism cannot be used. The required classes must be made available to the application by some other means (such as the appropriate setting of class paths).

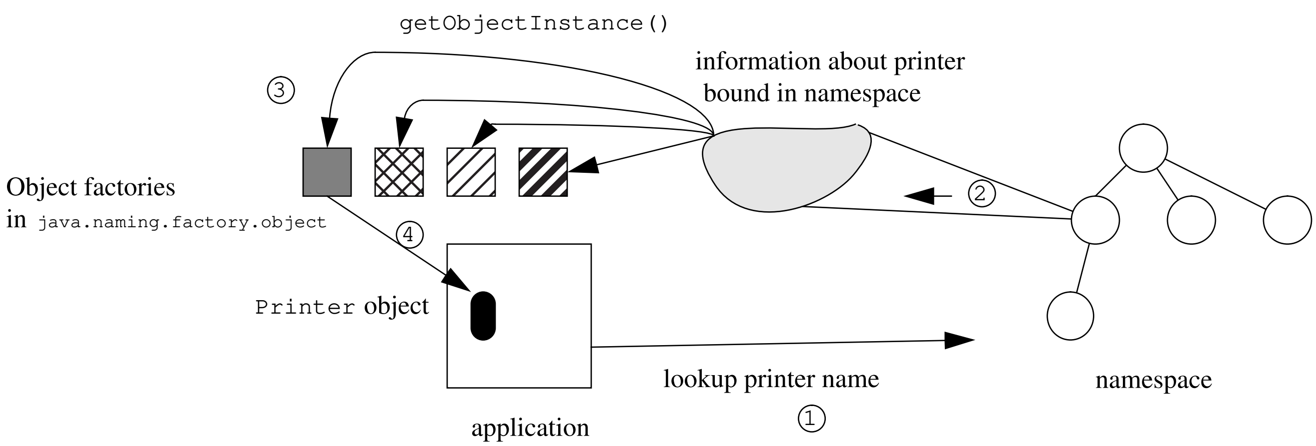
4.5 The java.naming.factory.object Property

In addition to extracting factory information from `References`, or using URLs, factories may be made available to JNDI with the `java.naming.factory.object` property.

The property `java.naming.factory.object` contains a colon-separated list of fully-qualified class names of object factories. Each class must implement the `ObjectFactory` interface and have a public constructor that takes no arguments. For each class in the list, JNDI attempts to load and instantiate the factory class, and to invoke the `ObjectFactory.getObjectInstance()` method on it using the object and environment arguments supplied. If the creation is successful, the resulting object is returned, otherwise, JNDI goes on to attempt the same procedure on the next class in the list.

The `java.naming.factory.object` property is made available to an application either in the environment property set passed to the `InitialContext` or `InitialDirContext` constructors, or as one of the program's system properties. If the property is supplied as part of the environment, the system property is not consulted.

Figure 4: Example using java.naming.factory.object to Get Back an Object from the Namespace



For the printer example, instead of using a `Reference` to represent a printer in the namespace, some other information is stored. When that information is later retrieved, the object factories specified `java.naming.factory.object` are tried in turn to attempt to turn that information into a `Printer` instance.

A service provider for such an object must do the following:

1. Define the class for the object (e.g `BSDPrinter`).
2. Define the class for reference information for the object. This is the object that will be bound in the namespace. This need not be `Reference`. It can be anything that will be understood by its corresponding object factory (e.g. some string containing the server name "`printer type=bsd; host=lobby-printserver`").

3. Define a factory class that implements `ObjectFactory` (e.g. `PrinterFactory`). This class's `getObjectInstance()` method will create an instance of the class from step 1 (e.g. `BSDPrinter`) when given an instance of class from step 2 (e.g. "`printer type=bsd; host=lobby-printserver`").

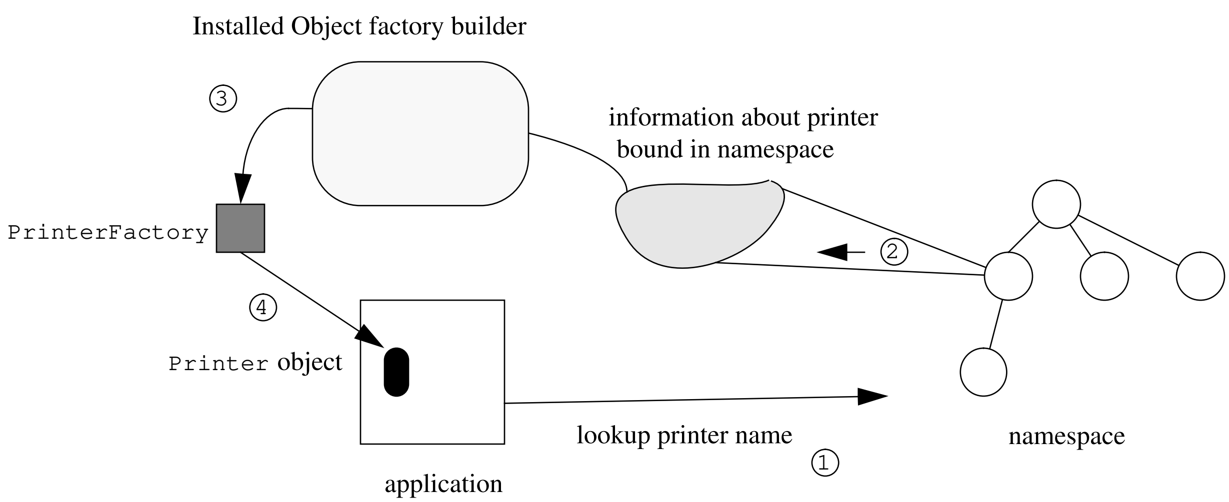
The service provider should automatically convert between the actual object (e.g. `BSDPrinter`) and the reference information (step 2, e.g. "`printer type=bsd; host=lobby-printserver`") when binding or looking up the object.

An application that wants to use a particular factory for generating objects must include the factory's class name in its `java.naming.factory.object` environment or system property and make the factory's classes and object classes available.

4.6 Object Factory Builder

If an *object factory builder* has been installed, the application is effectively defining its own policy of how to locate and construct object factories. When a builder has been installed, it is solely responsible for creating object factories. None of the default policies (`Reference`, URL string, or `java.naming.factory.object` property) normally used by JNDI are employed.

Figure 5: Example using an Object Factory Builder to Get Back an Object from the Namespace



A service provider for an object factory builder must do the following:

1. Define object factories that implement `ObjectFactory`.
2. Define a class that implements `ObjectFactoryBuilder`. This class's `createObjectFactory()` method will use the constructors for the `ObjectFactory` classes in step 1.

An application that wants to use this factory builder must first install it.

```
NamingManager.setObjectFactoryBuilder(builder);
```

5 Making Context Implementations Available to JNDI

In general, the JNDI mechanisms for creating generic objects using object factories described in "Object Factories" on page 15 also apply to how JNDI creates `Context` instances using context factories. However, if there is just one provider (that obtained through the initial context), and there is no need for that provider to use other providers, then the initial context is the sole controller of how context implementations are located. (See service provider example in Appendix A). This section is only relevant for serving composite namespaces, in which multiple providers are involved.

A service provider must define a class that implements the `Context` interface, and a class that implements the `ObjectFactory` interface for creating instances of this `Context` class. A provider can use either `Reference`s, URLs, or the other alternatives described in "Objects Bound in the Namespace" on page 15 to create instances of `Context` using object factories for their `Context` classes. Usually, context implementations act as object factories for `Context` classes (i.e. they implement the `ObjectFactory` interface). In these factories, `ObjectFactory.getObjectInstance()` returns instances of `Context` (or `DirContext`). The provider must ensure that the object factories are made known to JNDI either via the use of `Reference`, URLs, the `java.naming.factory.object` property, or by installing its own object factory builder.

6 Overview of the Interface[1]

The JNDI SPI is contained in the package `javax.naming.spi`. The following sections provide an overview of the SPI. For more details on the SPI, see the corresponding **javadoc**.

6.1 NamingManager and DirectoryManager

The `NamingManager` class contains static methods that perform provider-related operations. For example, it contains methods to create instances of objects using `Reference`, to obtain an instance of the initial context using the `java.naming.factory.initial` property, and to install `ObjectFactoryBuilder` and `InitialContextFactoryBuilder`. The `DirectoryManager` class provides similar static methods for `DirContext` related operations.

6.2 Federation Support

The `Resolver` interface defines a method for providers to implement that allows them to participate in a federation for supporting extended interfaces to `Context`. See “Resolving Through to Subinterfaces of Context” on page 3 for more details.

`ResolveResult` is the return value of calling `Resolver.resolveToClass()`. It contains the object to which resolution succeeded, and the remaining name yet to be resolved.

6.3 Object Factories

`ObjectFactory` is the interface for supporting creation of objects using information stored in the namespace. See “Object Factories” on page 15 for more details.

`ObjectFactoryBuilder` is the interface for creating object factories. See “Object Factory Builder” on page 21 for more details.

6.4 Initial Contexts

`InitialContextFactory` is the interface for creating an initial context instance. See “Implementing An Initial Context” on page 10 for more details.

`InitialContextFactoryBuilder` is the interface for creating `InitialContextFactory` instances. See “Initial Context Factory Builder” on page 12 for more details.

1. See Appendix B for legend of class diagram.

Appendix A: Service Provider Example

This appendix contains a simple service provider. It implements a flat namespace (with no federation support). It shows how to produce a context implementation by providing all the methods in the `Context` interface.

An instance of this context is bound directly as the initial context. This example provides the corresponding `InitialContextFactory` definition.

A.1 Simple Flat Context

A.1.1 Context Implementation

```
/*
 * Copyright (c) 1998.  Sun Microsystems. All rights reserved.
 */
package examples.spi.flat;

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

/**
 * A sample service provider that implements a flat namespace in memory.
 */

class FlatCtx implements Context {
    Hashtable myEnv;
    private Hashtable bindings = new Hashtable(11);
    static NameParser myParser = new FlatNameParser();

    FlatCtx(Hashtable environment) {
        myEnv = (environment != null)
        ? (Hashtable)(environment.clone())
        : null;
    }

    public Object lookup(String name) throws NamingException {
        if (name.equals("")) {
           // Asking to look up this context itself.  Create and return
            // a new instance with its own independent environment.
            return (new FlatCtx(myEnv));
        }
        Object answer = bindings.get(name);
        if (answer == null) {
            throw new NameNotFoundException(name + " not found");
        }
        return answer;
    }

    public Object lookup(Name name) throws NamingException {
        // Flat namespace; no federation; just call string version
        return lookup(name.toString());
    }

    public void bind(String name, Object obj) throws NamingException {
        if (name.equals("")) {
            throw new InvalidNameException("Cannot bind empty name");
        }
        if (bindings.get(name) != null) {
            throw new NameAlreadyBoundException(
                    "Use rebind to override");
        }
```

```
        bindings.put(name, obj);
    }

    public void bind(Name name, Object obj) throws NamingException {
        // Flat namespace; no federation; just call string version
        bind(name.toString(), obj);
    }

    public void rebind(String name, Object obj) throws NamingException {
        if (name.equals("")) {
            throw new InvalidNameException("Cannot bind empty name");
        }
        bindings.put(name, obj);
    }

    public void rebind(Name name, Object obj) throws NamingException {
        // Flat namespace; no federation; just call string version
        rebind(name.toString(), obj);
    }

    public void unbind(String name) throws NamingException {
        if (name.equals("")) {
           throw new InvalidNameException("Cannot unbind empty name");
        }
        bindings.remove(name);
    }

    public void unbind(Name name) throws NamingException {
        // Flat namespace; no federation; just call string version
        unbind(name.toString());
    }

    public void rename(String oldname, String newname)
            throws NamingException {
        if (oldname.equals("") || newname.equals("")) {
           throw new InvalidNameException("Cannot rename empty name");
        }

        // Check if new name exists
        if (bindings.get(newname) != null) {
            throw new NameAlreadyBoundException(newname +
                                                " is already bound");
        }

        // Check if old name is bound
        Object oldBinding = bindings.remove(oldname);
        if (oldBinding == null) {
            throw new NameNotFoundException(oldname + " not bound");
        }

        bindings.put(newname, oldBinding);
    }

    public void rename(Name oldname, Name newname)
```

```
            throws NamingException {
        // Flat namespace; no federation; just call string version
        rename(oldname.toString(), newname.toString());
    }

    public NamingEnumeration list(String name)
            throws NamingException {
        if (name.equals("")) {
            // listing this context
            return new FlatNames(bindings.keys());
        }

        // Perhaps 'name' names a context
        Object target = lookup(name);
        if (target instanceof Context) {
            return ((Context)target).list("");
        }
        throw new NotContextException(name + " cannot be listed");
    }

    public NamingEnumeration list(Name name)
            throws NamingException {
        // Flat namespace; no federation; just call string version
        return list(name.toString());
    }

    public NamingEnumeration listBindings(String name)
            throws NamingException {
        if (name.equals("")) {
            // listing this context
            return new FlatBindings(bindings.keys());
        }

        // Perhaps 'name' names a context
        Object target = lookup(name);
        if (target instanceof Context) {
            return ((Context)target).listBindings("");
        }
        throw new NotContextException(name + " cannot be listed");
    }

    public NamingEnumeration listBindings(Name name)
            throws NamingException {
        // Flat namespace; no federation; just call string version
        return listBindings(name.toString());
    }

   public void destroySubcontext(String name) throws NamingException {
        throw new OperationNotSupportedException(
                "FlatCtx does not support subcontexts");
    }

    public void destroySubcontext(Name name) throws NamingException {
        // Flat namespace; no federation; just call string version
```

```
        destroySubcontext(name.toString());
    }

    public Context createSubcontext(String name)
            throws NamingException {
        throw new OperationNotSupportedException(
                "FlatCtx does not support subcontexts");
    }

    public Context createSubcontext(Name name) throws NamingException {
        // Flat namespace; no federation; just call string version
        return createSubcontext(name.toString());
    }

    public Object lookupLink(String name) throws NamingException {
        // This flat context does not treat links specially
        return lookup(name);
    }

    public Object lookupLink(Name name) throws NamingException {
        // Flat namespace; no federation; just call string version
        return lookupLink(name.toString());
    }

    public NameParser getNameParser(String name)
            throws NamingException {
        return myParser;
    }

    public NameParser getNameParser(Name name) throws NamingException {
        // Flat namespace; no federation; just call string version
        return getNameParser(name.toString());
    }

    public String composeName(String name, String prefix)
            throws NamingException {
        Name result = composeName(new CompositeName(name),
                                  new CompositeName(prefix));
        return result.toString();
    }

    public Name composeName(Name name, Name prefix)
            throws NamingException {
        Name result = (Name)(prefix.clone());
        result.addAll(name);
        return result;
    }

    public Object addToEnvironment(String propName, Object propVal)
            throws NamingException {
        if (myEnv == null) {
            myEnv = new Hashtable(5, 0.75f);
    }
    return myEnv.put(propName, propVal);
```

```
}

public Object removeFromEnvironment(String propName)
        throws NamingException {
    if (myEnv == null)
        return null;

return myEnv.remove(propName);
}

public Hashtable getEnvironment() throws NamingException {
if (myEnv == null) {
    // Must return non-null
    myEnv = new Hashtable(3, 0.75f);
}
    return myEnv;
}

public void close() throws NamingException {
myEnv = null;
bindings = null;
}

// Class for enumerating name/class pairs
class FlatNames implements NamingEnumeration {
    Enumeration names;

    FlatNames (Enumeration names) {
        this.names = names;
    }

    public boolean hasMoreElements() {
        return names.hasMoreElements();
    }

    public boolean hasMore() throws NamingException {
        return hasMoreElements();
    }

    public Object nextElement() {
        String name = (String)names.nextElement();
      String className = bindings.get(name).getClass().getName();
        return new NameClassPair(name, className);
    }

    public Object next() throws NamingException {
        return nextElement();
    }
}

// Class for enumerating bindings
class FlatBindings implements NamingEnumeration {
    Enumeration names;
```

```
            FlatBindings (Enumeration names) {
                this.names = names;
            }

            public boolean hasMoreElements() {
                return names.hasMoreElements();
            }

            public boolean hasMore() throws NamingException {
                return hasMoreElements();
            }

            public Object nextElement() {
                String name = (String)names.nextElement();
                return new Binding(name, bindings.get(name));
            }

            public Object next() throws NamingException {
                return nextElement();
            }
        }
    };
```

A.1.2 Name Parser

```
/*
 * Copyright (c) 1998.  Sun Microsystems. All rights reserved.
 */
package examples.spi.flat;

import javax.naming.NameParser;
import javax.naming.Name;
import javax.naming.CompoundName;
import javax.naming.NamingException;
import java.util.Properties;

class FlatNameParser implements NameParser {

    static Properties syntax = new Properties();
    static {
        syntax.put("jndi.syntax.direction", "flat");
        syntax.put("jndi.syntax.ignorecase", "false");
    }
    public Name parse(String name) throws NamingException {
        return new CompoundName(name, syntax);
    }
}
```

A.1.3 Initial Context Factory

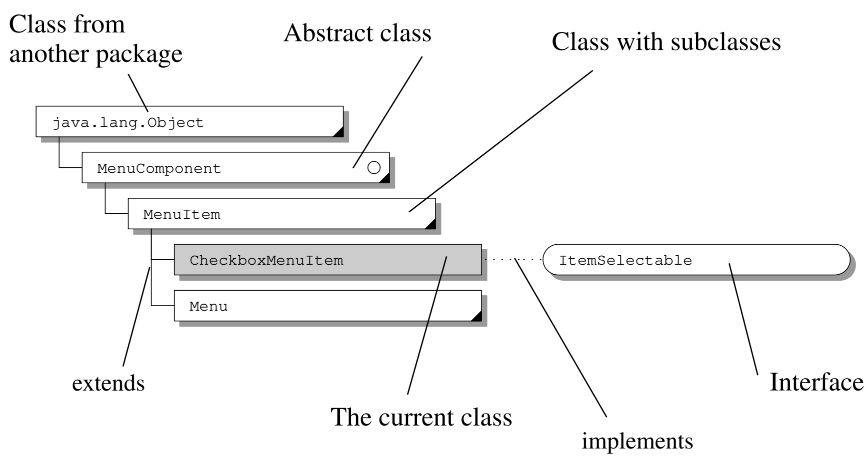
```
/*
 * Copyright (c) 1998.  Sun Microsystems. All rights reserved.
 */
package examples.spi.flat;

import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.spi.InitialContextFactory;

public class FlatInitCtxFactory implements InitialContextFactory {

    public Context getInitialContext(Hashtable env) {
        return new FlatCtx(env);
    }
}
```

Appendix B: 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 C: JNDI Change History

1.1: JNDI Changes Since 1.1Beta1

API-related Changes

- `DirContext.search()` accepts RFC 2254 filters instead of RFC 1960 filters.
- Added constants for environment properties to `Context` interface.
- Changed `Attributes.remove()` and `Attributes.put()` to return `Attribute` instead of `Object`.
- `Name.add()` and `Name.addAll()` (and related methods in `CompositeName` and `CompoundName`) are changed so that they return the updated instance of `Name`. This allows constructs such as `new CompositeName("a").add("b").add("c")` to be supported.
- Moved `implements Cloneable` from `CompositeName` and `CompoundName` to `Name`, where the `clone()` method is declared.
- Turned `Attribute` and `Attributes` into interfaces and renamed the original classes `BasicAttribute` and `BasicAttributes`, respectively.
- Allow class name to be specified in `Binding` and `SearchResult` constructors, and added `NameClassPair.setClassName()`.
- `DirContext.REMOVE_ATTRIBUTE` does not require nonexistent attributes or their values to be ignored.
- Added `throws NamingException` clause to `InitialContext` and `InitialDirContext` constructors so that connection problems can be reported and thrown at construction time.

SPI-related Changes

- `InitialContextFactory.getInitialContext()` must return a non-null initial context (instead of possibly null).
- `InitialContextFactoryBuilder.createInitialConetxtFactory()` and `ObjectFactoryBuilder.createObjectFactory()` now have a `throws NamingException` clause and must return non-null.
- `NamingManager.getObjectInstance()` and `getURLContext()` pass up exceptions thrown for operational errors (e.g. problem instantiating a factory class).

1.1Beta1: JNDI Changes Since 1.0Licensee Release

Package Name Change

JNDI is being packaged as a Java 1.1-compatible Standard Extension. The JNDI packages have been renamed to use the "javax" prefix, following the convention for Java Standard Extensions. The new package names are: `javax.naming`, `javax.naming.directory`, and `javax.naming.spi`.

General Changes

- Property names have been renamed following the convention used by the JDK. They have a "java.naming" prefix. See Appendix A of **JNDI API** document for details on the new names.
- Make `java.naming.provider.url` a system property in addition to being available as an environment property.
- Replaced use of `Properties` with `Hashtable` (`Properties`' superclass) for the environment properties/settings so that service providers and applications can completely enumerate its contents. `Properties` can still be passed as arguments and returned as values when `Hashtable` is called for. But declaring the methods to use `Hashtable` makes clear the fact that nested `Properties` are not examined for the operation at hand.

API-related Changes

As most of these changes are renames, the 1.1Beta1 release of the code includes a Java ClassRenamer[1] program that assists you with the renames. See the instructions for the release for details.

1. Thanks to the Swing team for use and distribution of this program.

- Added `Context.close()` to allow applications to release resources immediately.
- Added `InterruptedNamingException` to indicate a naming operation has been interrupted.
- Class renames: `DSContext->DirContext`, `InitialDSContext->InitialDirContext`, `AttributeSet->Attributes`, `InvalidAttributeSetException->InvalidAttributesException`, `SearchConstraints->SearchControls, InvalidSearchConstraintsException->InvalidSearchControlsException.`
- Make `Attributes`' methods look like `Map`'s[1], `Attribute`'s methods look like `Set`'s, and `Name`, `CompoundName`, `CompositeName`, and `Reference`'s methods look like `List`'s.
- Added protected `Attribute.Attribute()` constructor so that subclasses can avoid allocating `Vector`.
- Added constructors to `Attributes` that accept an attribute.
- Added `throws NamingException` clause to `Attribute`'s schema methods.
- Renamed `DirContext.DELETE_ATTRIBUTE->DirContext.REMOVE_ATTRIBUTE`.
- Replaced `ModificationEnumeration` with `ModificationItem[]`.
- Replaced `RefAddrEnumeration` and `StringEnumeration` with `Enumeration`.
- Replaced `AttributeEnumeration`, `NameClassEnumeration`, `BindingEnumeration`, aand `SearchEnumeration` with `NamingEnumeration` to allow generic means of doing JNDI enumerations.
- `Attribute.getAll()` returns `NamingEnumeration` instead of `Enumeration`.
- `Link.getLinkName()` returns `String` instead of `Name`.
- `BinaryRefAddr.buf` and `StringRefAddr.contents` made private. Deleted `Binary.getAddressBytes()`, `StringRefAddr.getAddressString()`, `BinaryRefAddr.size()`.
- Renamed `RefAddr.getAddressContents()->getContent()`.
- Removed `DSException`, re-parented exceptions to be subclass of `NamingException`
- Removed most constructors from `NamingException` and its subclasses. Each has two constructors: one that accepts an explanation and a public constructor that takes no parameters.
- Removed `Name.toString()`, `equals()`, `hashCode()` as these are already defined by `Object`.
- Constructors for abstract classes `RefAddr` and `ReferralException` are now protected.

SPI-related Changes

- `NamingManager.getObjectInstance()` and `ObjectFactory.getObjectInstance()` allow the caller to supply two optional parameters: a name and a context. The name is the name of the object resolved relative to the context supplied. An object factory can make use of this information to gather further information about the object to create. See the corresponding javadoc for these methods for details. Corresponding fields and accessor methods were added to `CannotProceedException` so that this information, if supplied, can be propagated.
- Constants used in `NamingManager` for property names removed: `ObjectFactoryProperty, InitialContextFactoryProperty`, `PkgPathProperty`. These were used for internal development. Programs should use the appropriate strings instead.
- `NamingManager.getObjectInstance()` returns original input if it cannot create a factory using the reference of the object (it used to return `null`).
- `InitialContext` constructor that takes no parameters calls `NamingManager.getInitialContext()` with a `null` environment instead of empty environment.

1.0Licensee Release: JNDI Changes Since 1.0Beta1

Package Name Change

To allow this release to work in all Java 1.1 systems, the JNDI classes have been temporarily renamed from the `java.naming` hierarchy to `com.sun.java.naming`.

API-related Changes

- `SearchConstraints` now implements `java.io.Serializable`.
- Added `ReferralException.skipReferrals()` to allow application to skip individual referrals.

1. See http://java.sun.com/products/jdk/preview/docs/guide/collections/ for information on `Map`, `Set` and `List`.

- Added constructor to `NoInitialContextException` that accepts an explanation string.
- Added `SchemaViolationException` for reporting schema-related problems.
- Renamed `java.naming.directory.SearchTimeLimitExceededException` to `java.naming.TimeLimitExceededException` so that it can be used by the `java.naming` package. Added `java.naming.LimitExceededException`, which is the super class of `TimeLimitExceededException` and `SizeLimitExceededException` (new as well).
- To assist in debugging and displaying classes, added `AttributeSet.toString()`, `Binding.toString()`, `SearchResult.toString()`.
- Clarified semantics of the overloaded form of `search()` that accepts a matching attribute set (`AttributeSet`). If the matching attribute set is `null` or empty, return all the objects in the target context.
- `AttributeSet` now implements `Cloneable`, and has a `clone()` method.

SPI-related Changes

- Added "set" methods to `NameClassPair`, `Binding`, and `SearchResult` classes and made the protected fields private. This enables service providers to update the fields in these classes without subclassing.
- Added a constructor to `NameClassPair`, `Binding`, and `SearchResult` that accepts a "relative" parameter, and `isRelative()` and `setRelative()` methods. This allows service providers to return names that are not relative to the target context of the search. Non-relative names are named using URL strings.
- Contract between `NamingManager.getObjectInstance()` and `ObjectFactory` is clarified. An object factory returns `null` if it cannot create the object; it only throws an exception (which is passed up to the caller of `NamingManager.getObjectInstance()`) if no other object factories should be tried.
- Replaced `Resolver.resolvePenultimate()` with `Resolver.resolveToClass()`. This allows more efficient implementation of service providers by allowing the resolution to stop at the first context that exports a target class, rather than requiring resolution to proceed to the penultimate context. The final service provider in a chain of federated naming systems no longer needs to implement `Resolver`; only the intermediate providers.must do so.
- Removed `NotDSContextException`. Service providers should use `NotContextException` with the target class name in the explanation to indicate that a particular subclass of `Context` is required but not found.
- The default package prefix for loading URL context factories has changed from "sun.jndi.url" to "com.sun.jndi.url" because of package renaming.

Document Version Numbers Reset

The earlier versions of the JNDI documents were labeled as versions 1.0, 1.1. and 1.2. They should have been "Early Access", "Beta1" and so on, to match the code releases.

1.0Beta1: JNDI Changes Since 1.0Early Access

API-related Changes

- Added `java.naming.ReferralException` to support client-side referrals. This abstract class is used to represent a referral exception, such as that available in LDAP v3. A service provider defines a subclass of `ReferralException` to handle its own style of referrals.

- Added `compareTo()` to `Name` (and related classes `CompositeName`, `CompoundName`).

  public int compareTo(Object obj);

  This method compares this `Name` with the specified `Object` for order. It returns a negative integer, zero, or a positive integer as this `Name` is less than, equal to, or greater than the given `Object`. This method is useful for sorting a list of names.

- Added '`throws NamingException`' to `Referenceable.getReference()` so that the implementor of `getReference()` can throw an exception if it encounters one.

  public Reference getReference() throws NamingException;

- `AttributeSet` was originally case-sensitive. That is, the case of an attribute identifier was considered when retrieving or adding an attribute to the set. To better support service providers that support case-insensitive attribute identifiers, an `AttributeSet` may now be made case-insensitive. This change involved adding a new constructor to `AttributeSet` and a new method for interrogating an attribute set about its handling of case.

  public AttributeSet(boolean caseIgnore);

  ```
  public boolean isCaseIgnored();
  ```

- `Context.setEnvironment()` was insufficient to allow both addition and removal of environment properties. The change is to replace `setEnvironment()` with `addToEnvironment()` and `removeFromEnvironment()`.

  public Properties addToEnvironment(Properties additions) throws NamingException;

  ```
  public Properties removeFromEnvironment(Properties deletions) throws
  NamingException;
  ```

- Added `hasMore()` to `BindingEnumeration`, `NameClassEnumeration` and `SearchEnumeration` so that a service provider can throw an exception when this query fails for some unexpected reason. `Enumeration.hasMoreElements()` cannot throw exceptions. The workaround is for `hasMoreElements()` to return `true` and save the exception until the program calls `next()`. `hasMore()` allows a provider to indicate to the caller that it has encountered an exception while determining whether there are more elements. The caller that wants to be notified of exceptions can use `hasMore()` instead of `hasMoreElements()`.

  public boolean hasMore() throws NamingException;

- Added a new constructor to `OperationNotSupportedException` that accepts an explanation message as argument. This avoids the provider having to use the two steps of creating an empty `OperationNotSupportedException` and then setting the explanation.

- Added `composeName()` methods to `Context` class. These may be used to keep track of the full name of an object as name resolution proceeds from context to context.

- Removed extraneous parameter in `NamingException.getRootCause()`.

SPI-related Changes

- Clarified how URL context factories and contexts are located and created. Eliminated the '`String url`' argument from `NamingManager.getURLContext()` and clarified its semantics.
  `getURLContext(String scheme, Properties env)` now returns a context for resolving URLs with scheme id `scheme`. It is not tied to any specific URLs, only the scheme id. See **JNDI SPI** document and `NamingManager.getURLContext()` for details.

- Clarified how `NamingManager.getObjectInstance()` treats URLs. Formerly, it only treated `References` and `Referenceables` specially. It now treats URLs specially as well. You can now call `getObjectInstance()` with a URL string or an array of URL strings and get back an object identified by the URL. See **JNDI SPI** document and `NamingManager.getObjectInstance()` for details.

- Placed additional requirements on URL context factories on how to treat its arguments so that all URL context factories behave consistently. See **JNDI SPI** document and `ObjectFactory.getObjectInstance()` for details.

- `NamingManager.getContinuationContext()` and `DirectoryManager.getContinuationDSContext()` accept as an argument `CannotProceedException` instead of a resolved object. This allows information required to create a continuation context to be passed using one argument and accommodates a common programming scenario of service providers using `CannotProceedException` to indicate the state of the operation.

- Added a 'remaining newname' part to `CannotProceedException` so that information required to continue a `rename()` can be represented, and an environment part for storing and retrieving the environment to use when resolution continues..

System Properties

- Two new system properties are introduced.
  - `jndi.urlfactory.pkgs`: Specifies package prefixes to use when loading URL context factories. See `NamingManager.getURLContext()`.
  - `jndi.dns.url`: Specifies DNS service location when using DNS names in "jndi" URLs (e.g "`jndi://dnsname/...`").

These can also be passed as environment properties to the `InitialContext` constructor.

Environment Properties

- `jndi.service.host` and `jndi.service.port` have been replaced by the more general `jndi.service.url`. `jndi.service.url` specifies the location information for configuring a context.
  Context service provider are encouraged to use this new environment property. They are still free to use additional environment properties as needed for their provider.
- Added `jndi.service.followReferrals`: Specifies that referrals encountered by the service provider are to be followed automatically.

1.0Early Access: JNDI Changes Since Initial Documentation Release

General Changes

- Renamed packages

```
jndi.ns -> java.naming

jndi.ds -> java.naming.directory

jndi.spi -> java.naming.spi
```

- Added implements `java.io.Serializable` to the following classes and interfaces:

```
Name
NameClassPair
RefAddr
Reference
Attribute
AttributeSet
ModificationItem
ModificationEnumeration
SearchConstraints
```

- Renamed the "count" methods to be more descriptive.

```
Reference.count() -> Reference.getAddressCount()
Name.count() -> Name.getComponentCount()
     [same for CompoundName and CompositeName]
Attribute.count() -> Attribute.getValueCount()
AttributeSet.count() -> AttributeSet.getAttributeCount()
ModificationEnumeration.count() ->
     ModificationEnumeration.getModificationItemCount()
```

- Renamed methods with '`SubContext`' to '`Subcontext`'. The new method names are now `Context.createSubcontext()`, `Context.destroySubcontext()`, and `DSContext.createSubcontext()`.

Name-related Changes

- `NameParser` is now an interface instead of abstract class. None of its methods contain any implementation so it is more flexible for it to be an interface. Removed the `getNamingConvention()` method from `NameParser`.

- Added class hierarchy to `NamingException` for security-related exceptions.

```
NamingException
    NamingSecurityException
        NoPermissionException
        AuthenticationException
        AuthenticationNotSupportedException
```

- Added `throws IllegalNameException` to name-manipulation methods so that they have a way of indicating error. This applies to the `Name` interface, the `CompositeName` and `CompoundName` classes.

```
prependName()
appendName()
insertName()
prependComponent()
appendComponent()
insertComponent()
deleteComponent()
```

- The following constructors throw `IllegalNameException` instead of `NamingException`

```
CompositeName()
CompoundName()
```

DSContext-related Changes:

- Dropped '`WithAttributes`' suffix from `bindWithAttributes()`, `rebindWithAttributes()`, and `createSubContextWithAttributes()`. They are now simply `DSContext.bind()`, `DSContext.rebind()`, and `DSContext.createSubcontext()`, respectively.
- Removed `DSContext.SearchFilter` class and replaced two existing `DSContext.Search()` methods:

```
public SearchEnumeration search(String name, String filterExpr,
    Object[] filterArgs, SearchConstraints constraints);
public SearchEnumeration search(Name name, String filterExpr,
    Object[] filterArgs, SearchConstraints constraints);
```

  where `filterExpr` contains '`{n}`', `n` is an integer and denotes the n'th element in `filterArgs` to substitute in the expression. The reason for this change is that `SearchFilter` had limited capabilities and a full class for it was not justified. These changes make the syntax for substitution of variables within an expression consistent with the formatting methods in `java.text`.

- Renamed `AttributeSet.modify()` to `AttributeSet.replace()` for consistent usage of 'replace' with `Attribute.replaceValue()` and `DSContext.REPLACE_ATTRIBUTE`.
- Changes to `Attribute` class:
  - Added `Attribute.contains()` for testing whether an attribute contains a specified value.
  - `Attribute.add()` throws `AttributeInUseException` instead of the more general `NamingException`.
  - Schema methods return `null` by default. Removed protected variables `syntax` and `attr_defn`.
- Added `InvalidAttributeSetException` to deal with the case of incorrectly or insufficiently specified attribute

sets.

SPI-related Changes

- Renamed some class and interface names in `java.naming.spi` for consistency

  `InitialContextImpl -> InitialContextFactory`

  `InitialContextImplFactory -> InitialContextFactoryBuilder`

  `setInitialContextImplFactory() -> setInitialContextFactoryBuilder()`

  `hasInitialContextImplFactory() -> hasInitialContextFactoryBuilder()`

  `InitialContextImplFactory.createInitialContextImpl() ->`

  `InitialContextFactoryBuilder.createInitialContextFactory()`

  `JNDIManager -> NamingManager`

  `JNDIDSManager -> DirectoryManager`

- Renamed `createObject()` to `getObjectInstance()` so that it is consistent with similar usage in other Java packages.

  `JNDIManager.createObject() -> NamingManager.getObjectInstance()`

  `ObjectFactory.createObject() -> ObjectFactory.getObjectInstance().`

- Renamed property `jndi.initialContext` to `jndi.initialContextFactory` for consistency with method names.
- The `jndi.initialContextFactory` property now contains a single class name instead of a colon-separated list because it does not make sense to have more than one class.
- To provide more flexibility and to avoid `SecurityManager`-related problems in some configurations, the system properties `jndi.initialContextFactory` and `jndi.objectFactories` can be passed as part of the environment properties passed to the constructors for `InitialContext` and `InitialDSContext`, and `ObjectFactory.getObjectInstance()`.
- Some protected methods in `NamingManager` and `DirectoryManager` are now private. This provides more flexibility in subsequent changes to these classes without exposing details of the implementation