In diesem Kapitel:

- *Einführung in JNDI*
- *Die JNDI Spezifikation*
  - *Einführung*
  - *Architektur*
  - *Packages und Klassen*
- *Übersicht - warum JNDI*
  - *Naming,*
  - *Directory,*
  - *Event,*
  - *LDAP*

Java Naming and Directory Interfaces

1.1. Einleitung – Grundsätzliches

Directory Services spielen eine vitale Rolle in Intranets und Internets, weil sie den Zugriff auf unterschiedlichste Informationen und 'Hosts' (in Internet Sinne : irgend etwas, was eine IP Adresse besitzt), Netzwerke, Services und Applikationen erlauben oder ermöglichen. Ein Verzeichnisdienst umfasst auf Grund seiner eigentlichsten Funktion eine Einrichtung zum Nachschlagen von Namen in Namensräumen ('naming facility'), welche auch für Menschen lesbar sind und welche unterschiedliche Entitäten zusammen fassen eventuell unter unterschiedlichen Gesichtspunkten.

Typischerweise umfasst eine Rechnerumgebung unterschiedliche Naming Facilities (Einrichtungen zum Nachschlagen vonNamen in Namensräumen), mit deren Hilfe komplexe Verzeichnisdienste aufgebaut werden können.

Das Internet Domain Name System (DNS) kann auf einer relativ hohen Ebene als übergeordneter Verzeichnisdienst eingesetzt werden, organisationsübergreifend. Daneben wird man aber vermutlich auch Verzeichnisdienste wie LDAP, NDS oder NIS verwenden (siehe unten für eine Erläuterung dieser Begriffe).

Aus Bentzersicht hat man vielleicht ein Adressierungsschema, welches aus zusammengesetzten Namen besteht. URLs sind Beispiele für zusammengesetzte Namen.

Viele Java Applikationensentwickler nutzen Verzechnisdienste und setzen diese in ihren Applikationen ein. Ein Verzeichnisdienst API wäre also durchaus etwas sinnvolles. Dieses API sollte so definiert sein, dass es unterschiedliche Verzeichnisschemata unterstützt.

Es wäre auch denkbar, dass eine Applikation ihre Objekte in einem Verzeichnisdienst einträgt und eine beliebige Java Applikation Objekte belieben Typus mit Hilfe des Verzeichnisdienstes herunterladen könnte.

Der Benutzer kann auch von Verzeichnisdiensten profitieren, da er Objekte (im weitesten Sinne) besser lokalisieren kann und sie dadurch eher nutzt.

Entwickler von Verzeichnisdiensten könnten von Service- Providern profitieren, welche diese Dienste direktanbieten: das spart Entwicklungszeit und erlaubt eine bessere Portabilität, Wartung und eine höhere Rekonfigurationsfähigkeit. Die Software wird damit leichter an neue Situation anpassbar.

JAVA NAMING AND DIRECTORY SERVICES

Java Naming and Directory Interface (JNDI) ist ein API, welches Java Applikationen Verzeichnis- und Namensgebungs- Funktionalität anbietet. Das API ist Verzeichnisdienst unabhängig, sollte also in Zusammenarbeit mit andern Diensten funktionieren.

Beispiele:

1. eine Applikation möchte ein Dokument drucken und benötigt ein entsprechendes Printer Objekt.

   Das lässt sich schematisch folgendermassen bewerkstelligen:

   ```
   prt = (Printer) D200.lookup("LaserWriter");
   prt.print(document);
   ```

   Mit Hilfe von JNDI kann das Printerobjekt gefunden und der Anwendung zur Verfügung gestellt werden.

2. Eine Applikation, welche eine Person in einem Telefonverzeichnis sucht, welches als Verzeichnisdienst implementiert ist, könnte etwa folgendermassen aussehen:

   ```
   String[] attrs = {"bueroNummer", "natelNummer",
                     "faxNummer", "privateNummer"};
   nicolesNummer = directory.getAttributes("cn=Nicole, o=ETH,
                       c=CH", attrs);
   ```

   Falls mehrere Nicole's im ETH Verzeichnis eingetragen sind (vermutlich trifft dies zu), kann man weiter suchen:

   ```
   nicole = directory.search("o=ETH, c=CH", "(cn=Nicole)",
                             searchctls);
   ```

In den folgenden Abschnitten wollen wir die Architektur und die Interfaces von JNDI kennen lernen, sowie andere Verzeichnisdienste mindestens grundsätzlich besprechen.

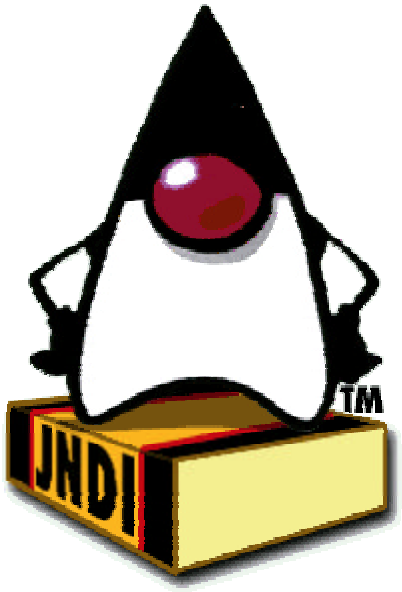
1.2. Theorie

1.2.1. Ziele und Entwurfsprinzipien von JNDI

Welche Ziele wurden bei der Definition von JNDI verfolgt und welche Design Prinzipien wurden festgelegt und befolgt?

Was und wie sollte JNDI sein?

1.2.1.1. Konsistent und intuitiv



Wann immer möglich sollten bestehende Teile und Ideen aus anderen Java Bereichen und APIs übernommen werden. Damit einher geht das Ziel, dieses neue Interface konsistent mit bestehenden APIs zu gestalten, also keine unnötige Profilierung anzustreben.

Da Java von Grund auf Objekt- orientiert entwickelt wurde, sollte dieses Ziel erreichbar sein, im Sinne einer natürlichen Erweiterung der Servicefunktionalitäten von Java.

1.2.1.2. Bezahle nur was Du brauchst

Wenn ein Applikationsprogrammierer nur zehn Prozent des Systems benötigt, sollte er auch nur zehn Prozent des Systems einbauen müssen, also keine Überladung mit unwichtigen Klassen, die nie benötigt werden.

1.2.1.3. Implementierbar neben und über andere gängige Verzeichnis und Namensdienste und Protokolle

Dieses Ziel ist aus folgendem Grund wichtig:

1. es erlaubt den Java Applikationen die üblichen Dienste wie DNS, NDS, NIS, X.500 und LDAP zu nutzen
2. damit lassen sich API spezifische Details reduzieren (die andern Dienste werden eingebunden)

Dabei muss aber darauf geachtet werden, keine spezifischen Dienste vorauszusetzen, da diese eventuell in einer bestimmten Umgebung nicht zur Verfügung stehen. Es soll nur eine einheitliche Schnittstelle auf unterschiedliche Namens- und Verzeichnisdienste geschaffen werden.

1.2.1.4. Problemlose Integration

Dieser Punkt ist aus mehreren Gründen wichtig:

1. es existieren verschiedene weit verbreitete und bewährte Verzeichnisdienste, die auch weiterhin bestehen bleiben
2. Java Applikationen sollten die Möglichkeit erhalten, problemlos die Verzeichnis- und Namensdienste extern nutzen, also auslagern zu können.

Der Entwickler sollte die Freiheit haben, die optimal passenden Dienste einzusetzen, ohne wesentlich eingeschränkt zu werden.

Ein sehr 'schwacher' Client, irgend ein Java unterstütztes Gerät, benötigt eher eine Proxy Lösung, um die Dienste an einen Server delegieren zu können. Ein mächtiger Client auf der andern Seite, wird aus vielen Gründen eher direkten Zugriff auf eine Verzeichnis wünschen.

Diese Designentscheide sollten aber so gekapselt werden können, dass sie erst möglichst spät gefällt werden müssen.

1.2.1.5. Unterstützung der führenden Industriestandards

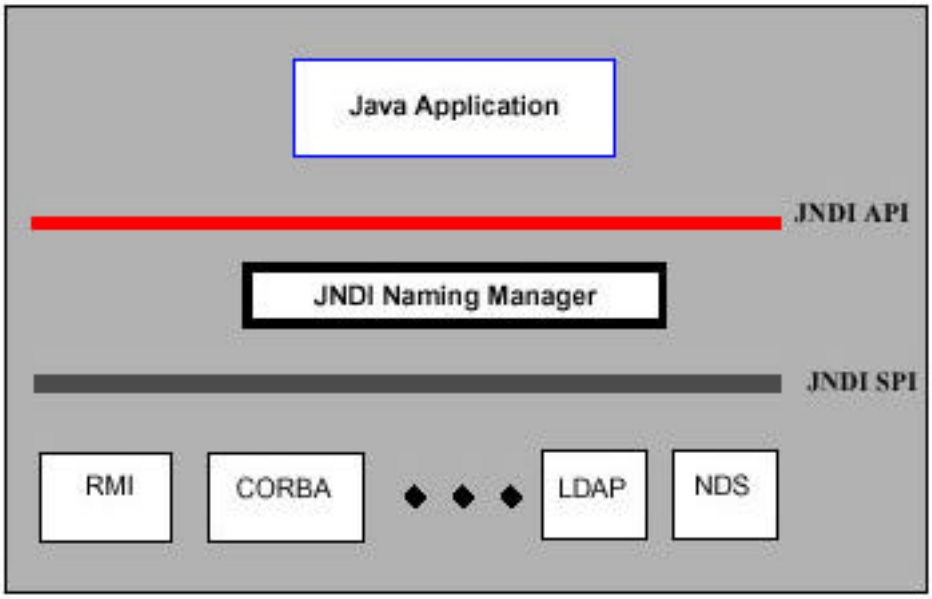
Das Lightweight Directory Access Protocol (LDAP Internet RFC 2251) hat sich als Standard für Verzeichnisdienste auf der Protokollebene durchgesetzt. Alle wichtigen Hersteller unterstützen dieses Protokoll oder es ist zumindest möglich Lösungen für (fast) jede Platform zu finden.

Eine Applikation, welche JNDI einsetzt, sollte daher in der Lage sein, alle Funktionen, die LDAP anbietet zu nutzen und zu unterstützen (beispielsweise Search Queries/Filter).

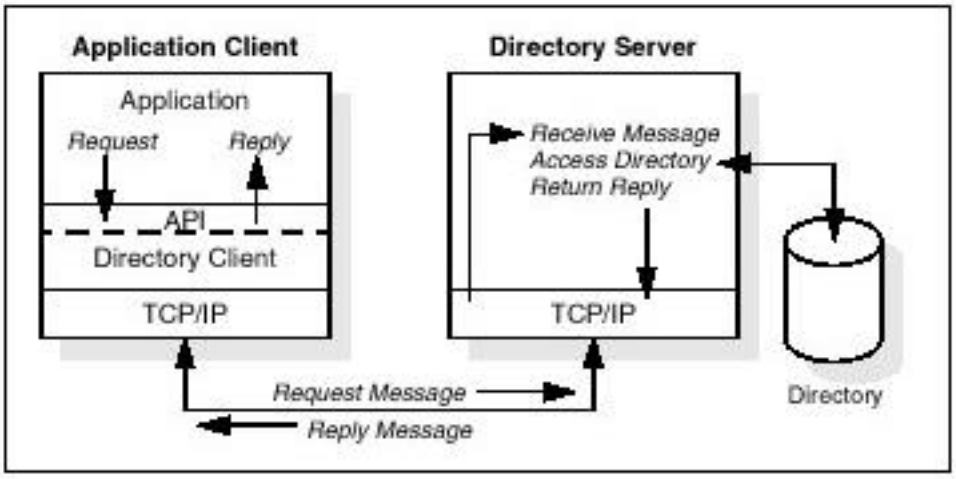
1.2.2. Achitektur von JNDI

Die JNDI Architektur besteht aus dem JNDI API und dem JNDI SPI. Das JNDI API gestattet es Java Applikationen unterschiedliche Namens- und Verzeichnisdienste zu nutzen.
Das JNDI SPI sollte eine einfache Integration der JNDI Applikationen in die Server Umgebung bestehender Dienste erlauben

Das folgende Bild soll dies veranschaulichen: auf der Serverseite sind unterschiedliche Dienste vorhanden oder können genutzt werden. Die Java Applikation kann sich mit dem API Zugriff darauf verschaffen.



Schematisch sieht eine Applikation, welche Namens- und Verzeichnis- Dienste einsetzt, etwa folgendermassen aus:



1.2.3. Grundlegende Konstrukte von JNDI

Ein Verzeichnisdienst gestattet den Zugriff auf unterschiedliche Informationen über Benutzer und Resourcen in einer Netzwerkumgebung. Ein Verzeichnisdienst verwendet ein Namenssystem (*naming system*), um Verzeichnisobjekte (*directory objects*) , die die Informationen darstellen, zu identifizieren und zu organisieren.

Ein Verzeichnisobjekt stellt eine Verbindung her zwischen *Attributen* und *Werten*.

Damit lässt sich Information auf hierarchische Art und Weise gruppieren und zusammen fassen und eine Verbindung zwischen den Daten und einer merkbaren und lesbaren darstellung, einem Namen schaffen.

1.2.4. Naming - die Grundlagen

Eine grundlegende Einrichtung in allen Rechnersystemen ist ein Namensservice - ein Werkzeug, mit dessen Hilfe Namen mit Objekten in Verbindung gebracht und Objekte bei gegebenem Namen gefunden werden können.

In traditionellen Systemen ist der Namensdienst selten ein unabhängiger Dienst. Normalerweise sind diese Dienste integriert in andere Dienste, wie etwa ein Dateisystem. ein Verzeichnissystem, Datenbanken, Desktop, Mailsystem, Spreadsheet oder Kalender.

Beispiele:
eine Dateisystem umfasst einen Namensdienst für die Dateien und Verzeichnisse;
ein Spreadsheet besitzt einen Namensdienst für Zellen und Makros.

Eine typische Arbeitsumgebung ist normalerweise in mehrere Namensdienste eingebettet. Diese Dienste umfassen zum Beispiel die Organisation, die physikalischen Gegebenheiten (Gebäude, Büro), Rechnerumgebungen und ähnliches. Namensdienste werden auch in anderen Anwendungen wie zum Beispiel Dateiverzeichnisse, Mailverzeichnissen, Druckerverzeichnisse, ...

Aus Sicht des Benutzers bestehen verschiedene Beziehungen zwischen einzelnen Namensdiensten, logische und natürliche.

Mit Dateiverzeichnis, Mailverzeichnis, Kalender ... ist der Benutzer direkt konfrontiert. Auch die Eingliedrung der Person in eine Organisation (Team, Abteilung, Unternehmen, Sparte, Konzern) ist eine natürliche.

Solche Strukturen legen eine Namensgebung auf natürliche Art und Weise nahe:

- ein *Name* wird gemäss einigen syntaktischen Regeln, der *Namenskonvention*, gebildet.
- ein *atomarer* Name ist eine unteilbare Komponente eines Namens, gemäss der Namenskonvention
- ein *zusammengesetzter* Name (*compound name*) besteht aus einer Sequenz von null oder mehreren atomaren Namen gemäss der Namenskonvention

Betrachten wir einige Beispiele:

1. ein UNIX Pfadnamen:
   atomare Namen sind von links nach rechts geordnet und jeweils durch einen Slash ("/") getrennt;

   Der UNIX Pfadname `usr/local/bin` besteht aus einem zusammengesetzten Namen bestehend aus der Sequenz der atomaren Namen `usr, local,` und `bin`.

2. im Namenssystem aus dem Internet Domain System (DNS) sind die atomaren Namen von rechts nach links angeordnet und jeweils durch einen Punkt (".") getrennt.

   Der DNS Name `HTA.FHZ.CH` ist ein zusammengesetzter Namen bestehend aus der Sequenz der atomaren Namen `CH, FHZ, HTA`.

Die Assoziation eines atomaren Namens mit einem Objekt bezeichnet man als *Bindung* (*bindin*g).

Ein *Context* ist ein Objekt, dessen Zustand aus mehreren Bindungen besteht (mit atomaren Namen).

- Jeder Context besitzt eine spezifische Namenskonvention.
- Ein Context stellt einen Nachschlagmechanismus (lookup ,resolution), der Objekte nachschlagen kann und Objekte zurück liefert und Methoden anbietet wie etwa das Binden der Namen, das Auflösen von Namen (unbinding) und das Auflisten von Namensbindungen.
- ein atomarer Namen in einem Context kann dieses mit einem Contextobjekt des selben Typus, einem Subcontext verbunden werden, wodurch ein zusammengesetzter Namen entsteht (der DNS Namen im obigen Beispiel ist ein Beispiel dafür)

Die Auflösung eines zusammengesetzten Namens geschieht, indem man sukzessive atomare Komponenten im jeweiligen Kontext auflöst.

Beispiel:
In einem Verzeichnis sind mehrere Dateien eingetragen, darunter auch Unterverzeichnisse; in jedem Unterverzeichnis befinden sich wieder mehrere Dateien und eventuell weitere Verzeichnisse.

Unter Namensystem, *naming system*, versteht man zusammengesetzte Contexts des selben Typs (also einheitlicher Namenskonvention) mit bestimmten Methoden / Operationen und einheitlicher Semantik.

Ein Namensraum, *namespace* ist die Menge aller Namen in einem Namensystem.

Ein zusammengesetzter Namen, *composite name*, ist ein Namen, mehrere Namensysteme umfasst. Er besteht aus keiner oder mehreren Komponenten. Jede Komponente ist ein Namen aus dem Namensraum eines Namensystems.

Beispiel

1. `jurassic.eng:/export/home/jdoe/.signature` ist ein zusammengesetzter Namen bestehend aus dem Rechnernamen `jurassic.eng` aus einem Rechner Namensraum und dem Dateinamen `/export/home/jdoe/.signature` aus einem UNIX Datei- Namesraum.

2. ein anderes Beispiel ist die Internet URL
   `http://www.moon.org/public/index.html,`

   sie ist ein zusammengsetzter Namen bestehend aus Schema-ID (Protokoll) `http` aus dem“URL scheme-id” Namesraum,
   `www.moon.org,` dem DNS Namen der Maschine, auf der der Webserver läuft
   und `public/index.html,` dem Dateinamen aus dem Datei- Namespace.

Jeder Namen muss relativ zu einem Context analysiert werden, und jede Namensoperation betrifft ein Contextobjekt.

Ein Client kann einen initialen Context, *initial context* Objekt definieren, welches als Startpunkt für die Auflösung der Namen benutzt werden kann.

1.2.5. Verzeichnisobjekte / Directory Objekte

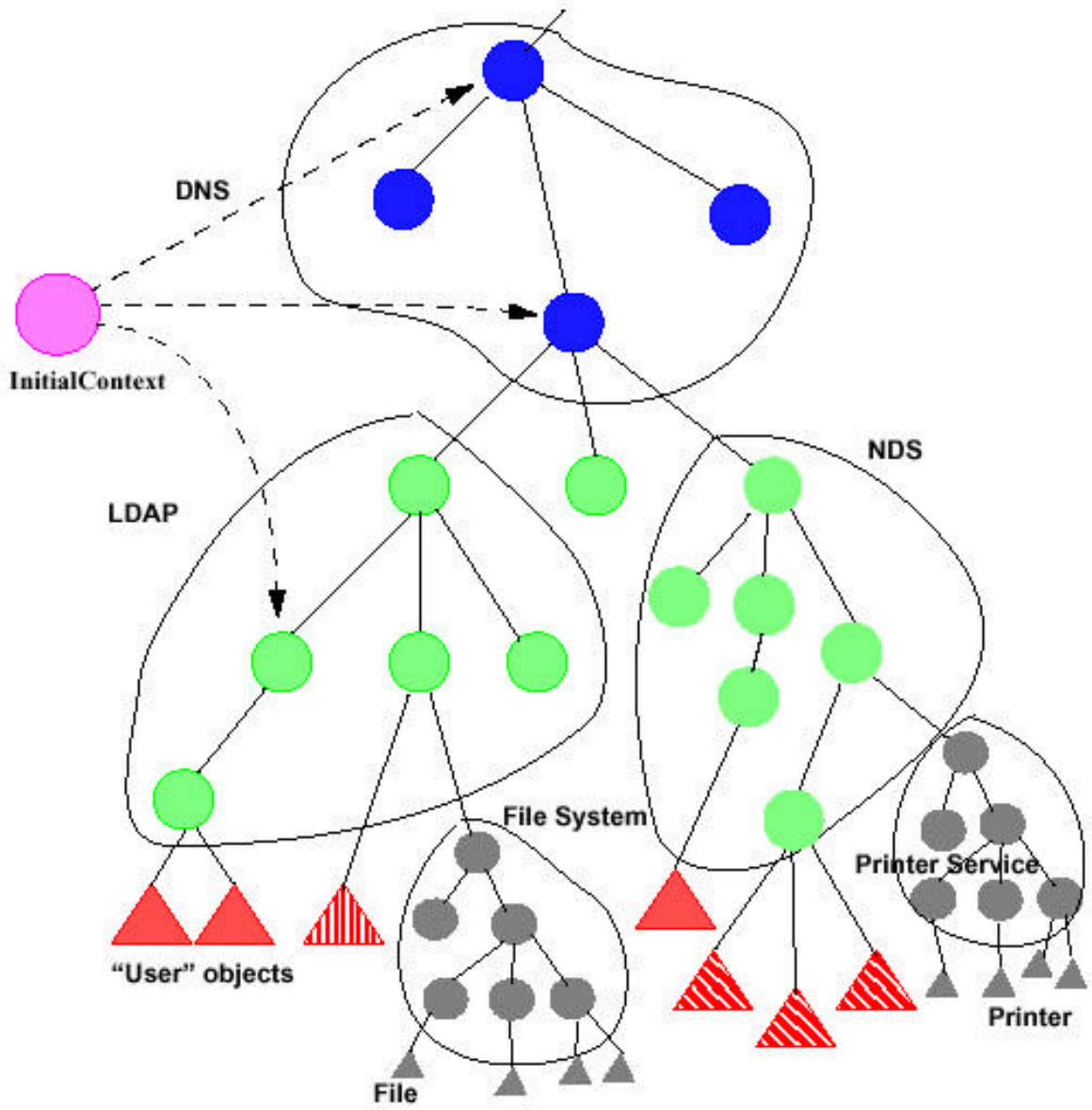
Die Hauptfunktion eines Namensystems besteht in der Abbildung von Namen auf Objekte. Die Objekte können dabei von irgend einem Typus sein. Ein Verzeichnisobjekt, *directory object*, ist ein spezielles Objekt, welches verschiedene Informationen in einem Rechnerumfeld repräsentiert.

- einem Verzeichnisobjekt können *Attribute* zugeordnet sein. Ein Attribut besitzt einen Identifier und einen oder mehrere Werte.
- ein Verzeichnisobjekt stellt Methoden , Operationen zur Verfügung, mit denen Attribute des Objektes kreiert, hinzugefügt, entfernt und modifiziert werden können.

Falls ein Verzeichnisobjekt auch zu einem Namenskontext gemacht wird, können wir Baumstrukturen definieren, bei denen interne Knoten sich nicht nur wie Namenskontexte benehmen, sondern auch Attribute enthalten.

Im folgende Bild zeigen wir verschiedene Sachverhalte:

- mehrere Namensysteme können zusammen einem zusammengsetzten Namensraum bilden.
  In diesem Fall wird DNS als globales Namensystem eingesetzt;
  eine Division verwendet NDS, eine weitere LDAP.
- jeder Namensraum besitzt interne Knoten; diese stellen Namenskontexte dar, die selber auch Verzeichnisobjekte sein können. Ein Blatt kann ein beliebiges Objekt sein.
- der *InitialContext* ist so konfiguriert, dass er brauchbare Bindungen zu hilfreichen Startkontexten in unterschiedlichen Namens- und Verzeichnissystemen besitzt.
- Applikationen sehen nur einen zusammengesetzten Namensraum.
- Services können ihren eigenen Namensraum einbringen.
- beliebige Verzeichnisdienste können hinzugefügt und benutzt werden, ohne dass die Client Applikationen geändert werden müssen.


DNS
InitialContext
LDAP
NDS
File System
Printer Service
"User" objects
File
Printer

1.2.6. URLs und zusammengesetzte Namen

Uniform Resource Locators (URLs) sind spezielle zusammengesetzte Namen, deren Syntax durch die URL Definition geregelt ist.

JNDI Clients können URLs benutzen, um beliebige Objekttypen zu referenzieren.

Beispiel
`nfs://nfs.sun.com/export/jndi/src/README`

beschreibt ein Dateiobjekt, auf welches mit Hilfe des Network File System (NFS) Protokolls zugegriffen werden kann.

Analog kann man ein LDAP Objekt als URL darstellen :
`ldap://ldap.pizza-onnection.it/cn=Mario,ou=liquidation`.

Zusammengesetzte Namen werden in JNDI unterstützt. JNDI definiert eine Namenssyntax und stellt Hilfeprogramme bzw. Methoden zur Verfügung, mit deren Hilfe zusammengesetze Namen bearbeitet werden können.

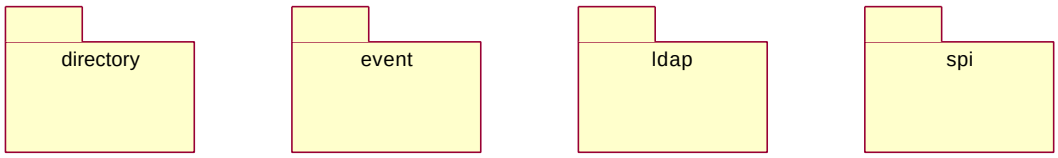
Dadurch kann man in JNDI Objekte beschreiben, referenzieren, welche zu mehreren Namensräumen gehören (wie eine ULR: DNS und Dateisystem).

1.2.7. Ereignisse

Da Namens-/ Verzeichnisservices eine immer wichtigere Rolle im Informatikumfeld spielen, benötigt man auch entsprechende Administrationswerkzeuge. Diese benötigen ein effizientes Programmparadigma - das ereignisgesteuerte : asynchrone Benachrichtigungen, welche einer Applikation gestatten, sich für die Benachrichtigung beim Eintreffen bestimmter Nachrichten (beim Eintreten bestimmter Ereignisse) anzumelden (Publish / Subscribe Design Pattern).

1.2.8. Übersicht über das Interface

Das JNDI API besteht aus vier Paketen:



1.2.9. Package Hierarchie: alle Packages von JNDI

javax.naming, javax.naming.directory, javax.naming.event, javax.naming.ldap, javax.naming.spi

1.2.9.1. Klassen Hierarchie

- class java.lang.**Object**
  - class javax.naming.directory.**BasicAttribute** (implements javax.naming.directory.Attribute)
  - class javax.naming.directory.**BasicAttributes** (implements javax.naming.directory.Attributes)
  - class javax.naming.**CompositeName** (implements javax.naming.Name)
  - class javax.naming.**CompoundName** (implements javax.naming.Name)
  - class javax.naming.ldap.**ControlFactory**
  - class javax.naming.spi.**DirStateFactory.Result**
- class java.util.**EventObject** (implements java.io.Serializable)
  - class javax.naming.event.**NamingEvent**
  - class javax.naming.event.**NamingExceptionEvent**
  - class javax.naming.ldap.**UnsolicitedNotificationEvent**
  - class javax.naming.**InitialContext** (implements javax.naming.Context)
    - class javax.naming.directory.**InitialDirContext** (implements javax.naming.directory.DirContext)
    - class javax.naming.ldap.**InitialLdapContext** (implements javax.naming.ldap.LdapContext)
    - class javax.naming.directory.**ModificationItem** (implements java.io.Serializable)
  - class javax.naming.**NameClassPair** (implements java.io.Serializable)
  - class javax.naming.**Binding**
    - class javax.naming.directory.**SearchResult**
    - class javax.naming.spi.**NamingManager**
    - class javax.naming.spi.**DirectoryManager**
  - class javax.naming.**RefAddr** (implements java.io.Serializable)
  - class javax.naming.**BinaryRefAddr**
  - class javax.naming.**StringRefAddr**
  - class javax.naming.**Reference** (implements java.lang.Cloneable, java.io.Serializable)
  - class javax.naming.**LinkRef**
    - class javax.naming.spi.**ResolveResult** (implements java.io.Serializable)

        - class javax.naming.directory.**SearchControls** (implements java.io.Serializable)
- class java.lang.**Throwable** (implements java.io.Serializable)
- class java.lang.**Exception**
    - class javax.naming.**NamingException**
        - class javax.naming.directory.**AttributeInUseException**
        - class javax.naming.directory.**AttributeModificationException**
    - class javax.naming.**CannotProceedException**
    - class javax.naming.**CommunicationException**
    - class javax.naming.**ConfigurationException**
    - class javax.naming.**ContextNotEmptyException**
    - class javax.naming.**InsufficientResourcesException**
    - class javax.naming.**InterruptedNamingException**
        - class javax.naming.directory.**InvalidAttributeIdentifierException**
        - class javax.naming.directory.**InvalidAttributesException**
        - class javax.naming.directory.**InvalidAttributeValueException**
    - class javax.naming.**InvalidNameException**
        - class javax.naming.directory.**InvalidSearchControlsException**
        - class javax.naming.directory.**InvalidSearchFilterException**
    - class javax.naming.**LimitExceededException**
    - class javax.naming.**SizeLimitExceededException**
    - class javax.naming.**TimeLimitExceededException**
    - class javax.naming.**LinkException**
    - class javax.naming.**LinkLoopException**
    - class javax.naming.**MalformedLinkException**
    - class javax.naming.**NameAlreadyBoundException**
    - class javax.naming.**NameNotFoundException**
    - class javax.naming.**NamingSecurityException**
    - class javax.naming.**AuthenticationException**
    - class javax.naming.**AuthenticationNotSupportedException**
    - class javax.naming.**NoPermissionException**
    - class javax.naming.**NoInitialContextException**
        - class javax.naming.directory.**NoSuchAttributeException**
    - class javax.naming.**NotContextException**
    - class javax.naming.**OperationNotSupportedException**
    - class javax.naming.**PartialResultException**
    - class javax.naming.**ReferralException**
        - class javax.naming.ldap.**LdapReferralException**
        - class javax.naming.directory.**SchemaViolationException**
    - class javax.naming.**ServiceUnavailableException**

1.2.9.2. Interface Hierarchie

- interface java.lang.**Cloneable**
  - interface javax.naming.directory.**Attribute** (also extends java.io.Serializable)
  - interface javax.naming.directory.**Attributes** (also extends java.io.Serializable)
  - interface javax.naming.**Name** (also extends java.io.Serializable)
  - interface javax.naming.**Context**
    - interface javax.naming.directory.**DirContext**
    - interface javax.naming.event.**EventDirContext** (also extends javax.naming.event.EventContext)
    - interface javax.naming.ldap.**LdapContext**
    - interface javax.naming.event.**EventContext**
    - interface javax.naming.event.**EventDirContext** (also extends javax.naming.directory.DirContext)
- interface java.util.**Enumeration**
  - interface javax.naming.**NamingEnumeration**
- interface java.util.**EventListener**
  - interface javax.naming.event.**NamingListener**
  - interface javax.naming.event.**NamespaceChangeListener**
  - interface javax.naming.event.**ObjectChangeListener**
  - interface javax.naming.ldap.**UnsolicitedNotificationListener**
  - interface javax.naming.ldap.**HasControls**
  - interface javax.naming.ldap.**UnsolicitedNotification** (also extends javax.naming.ldap.ExtendedResponse)
  - interface javax.naming.spi.**InitialContextFactory**
  - interface javax.naming.spi.**InitialContextFactoryBuilder**
  - interface javax.naming.**NameParser**
    - interface javax.naming.spi.**ObjectFactory**
    - interface javax.naming.spi.**DirObjectFactory**
    - interface javax.naming.spi.**ObjectFactoryBuilder**
  - interface javax.naming.**Referenceable**
    - interface javax.naming.spi.**Resolver**
- interface java.io.**Serializable**
  - interface javax.naming.directory.**Attribute** (also extends java.lang.Cloneable)
  - interface javax.naming.directory.**Attributes** (also extends java.lang.Cloneable)
  - interface javax.naming.ldap.**Control**
  - interface javax.naming.ldap.**ExtendedRequest**
  - interface javax.naming.ldap.**ExtendedResponse**
  - interface javax.naming.ldap.**UnsolicitedNotification** (also extends javax.naming.ldap.HasControls)
  - interface javax.naming.**Name** (also extends java.lang.Cloneable)
    - interface javax.naming.spi.**StateFactory**
    - interface javax.naming.spi.**DirStateFactory**

1.2.9.3. Grobübersicht

- `javax.naming.directory`
  erweitert das Paket `javax.naming,` um auf Verzeichnisse zugreifen zu können
- `javax.naming.event`
  enthält Klassen und Interfaces, um Ereignisdienste unterstützen zu können
- `javax.naming.ldap`
  enthält Klassen und Interfaces für die Unterstützung von LDAP v3 Erweiterungen

Das JNDI Service Provider Interface (SPI) ist in einem separaten Paket enthalten:

- `javax.naming.spi`
  enthält Klassen und Interfaces, mit deren Hilfe verschiedene Namens- und Verzeichnisdienste Provider dynamisch unter das JNDI API geschoben werden können (Details sind in der **JNDI SPI** Dokumentation enthalten)

In den folgenden Abschnitten geben wir eine Übersicht über die verschiedenen JNDI APIs. Details zu den einzelnen APIs müssten Sie der **javado**c Dokumentation entnehmen.

1.2.10. Das Naming Package — javax.naming

1.2.10.1. Klassen Hierarchie

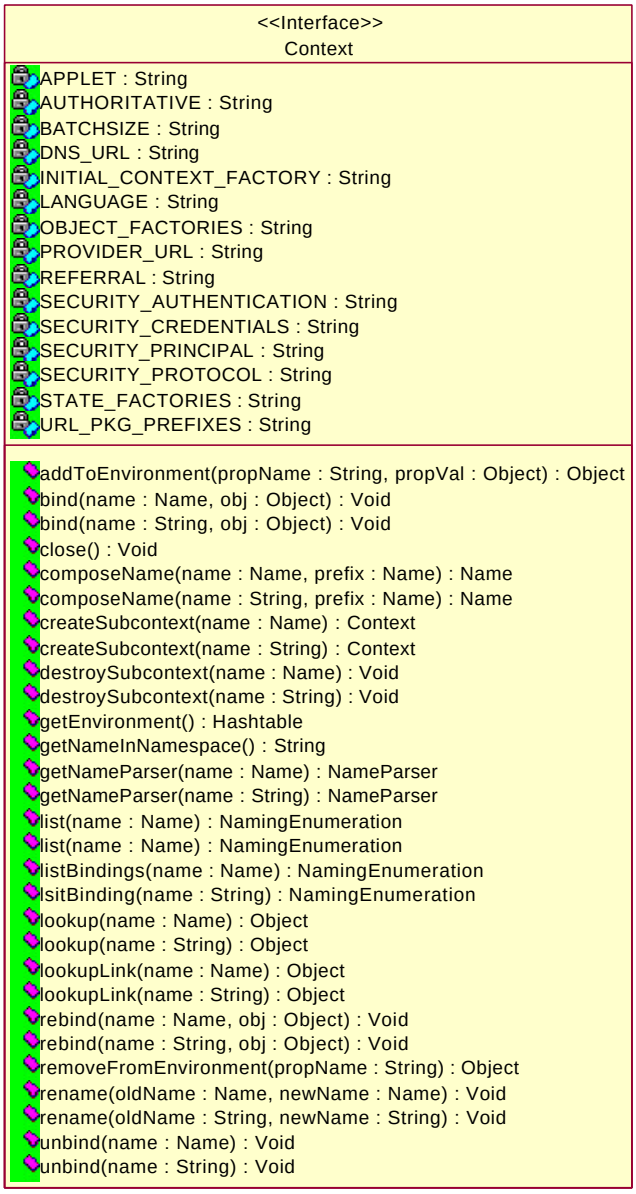
- class java.lang.**Object**
  - class javax.naming.**CompositeName** (implements javax.naming.Name)
  - class javax.naming.**CompoundName** (implements javax.naming.Name)
  - class javax.naming.**InitialContext** (implements javax.naming.Context)
  - class javax.naming.**NameClassPair** (implements java.io.Serializable)
  - class javax.naming.**Binding**
  - class javax.naming.**RefAddr** (implements java.io.Serializable)
  - class javax.naming.**BinaryRefAddr**
  - class javax.naming.**StringRefAddr**
  - class javax.naming.**Reference** (implements java.lang.Cloneable, java.io.Serializable)
  - class javax.naming.**LinkRef**
- class java.lang.**Throwable** (implements java.io.Serializable)
- class java.lang.**Exception**
  - class javax.naming.**NamingException**
  - class javax.naming.**CannotProceedException**
  - class javax.naming.**CommunicationException**
  - class javax.naming.**ConfigurationException**
  - class javax.naming.**ContextNotEmptyException**
  - class javax.naming.**InsufficientResourcesException**
  - class javax.naming.**InterruptedNamingException**
  - class javax.naming.**InvalidNameException**
  - class javax.naming.**LimitExceededException**
  - class javax.naming.**SizeLimitExceededException**
  - class javax.naming.**TimeLimitExceededException**
  - class javax.naming.**LinkException**
  - class javax.naming.**LinkLoopException**
  - class javax.naming.**MalformedLinkException**
  - class javax.naming.**NameAlreadyBoundException**
  - class javax.naming.**NameNotFoundException**
  - class javax.naming.**NamingSecurityException**
  - class javax.naming.**AuthenticationException**
  - class javax.naming.**AuthenticationNotSupportedException**
  - class javax.naming.**NoPermissionException**
  - class javax.naming.**NoInitialContextException**
  - class javax.naming.**NotContextException**
  - class javax.naming.**OperationNotSupportedException**
  - class javax.naming.**PartialResultException**
  - class javax.naming.**ReferralException**
  - class javax.naming.**ServiceUnavailableException**

1.2.10.2. Interface Hierarchy

- interface java.lang.**Cloneable**

- interface javax.naming.**Name** (also extends java.io.Serializable)
- interface javax.naming.**Context**

- interface java.util.**Enumeration**
  - interface javax.naming.**NamingEnumeration**
  - interface javax.naming.**NameParser**
  - interface javax.naming.**Referenceable**
- interface java.io.**Serializable**
  - interface javax.naming.**Name** (also extends java.lang.Cloneable)



1.2.10.3. Context

`Context` ist das wichtigste Interface, mit dem der Naming Context beschrieben wird. Diese Schnittstelle definiert die grundlegenden Methoden, wie Hinzufügen eines Namen-Objekt Paares, nachschlagen eines Objekts zu einem Namen, Entfernen einer Namen- Objekt Bindung, kreieren und zerstören eines Subkontextes und ähnliches:

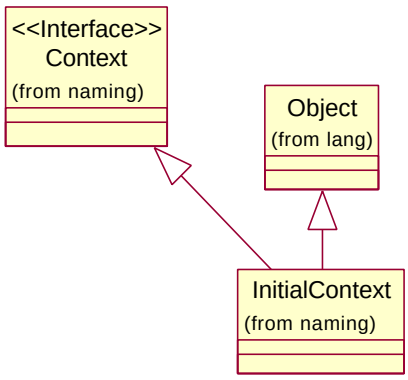
```
public Context
createSubcontext(Name name)
throws NamingException;

public void
destroySubcontext(Name name)
throws NamingException;
...
};
```

1.2.10.4. Der Initial Context

In JNDI wird jeder Namen relativ zu einem Context definiert. Es gibt keine "absolute Namen". Eine Applikation kann bootstrappen, indem sie einen ersten Context, den initialen Context, von der Klasse `InitialContext` erhält:

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



Attribute und Methoden werden gemäss dem Interface Context implementiert.

Alle Namensoperationen werden relativ zu diesem initialen Context durchgeführt, insbesondere die Namensauflösung. Verschiedene Parameter werden als Umgebungsparameter an den Konstruktor des initialen Kontextes übergeben, zum Beispiel als Applet Parameter (Context.APPLET). Diese speziellen Parameter sind im Interface Context definiert.

JNDI bestimmt die Werte der einzelnen Parameter, indem die Werte aus den zwei folgenden Quellen zusammen genommen werden:

1. zuerst werden die Werte des Konstruktors und die Applet- und System- Parameter verwendet
2. zusätzlich werden die Werte in der Ressourcedatei berücksichtigt (jndi.properties).

Für jede Eigenschaft, welche in beiden Datenquellen gefunden werden, wird der Wert der Eigenschaft wiefolgt bestimmt:

- falls es sich um eine JNDI Eigenschaft handelt, welche eine Liste definiert (siehe Enumeration in der Definition der Schnittstelle Context), dann wird der Wert einfach indiese Liste eingefügt.

- in den andern Fällen wird der erste gefundene Wert verwendet.

In "java.naming.factory.initial" steht der Namen der Contextfactory für den initialen Context. Falls als Namen eine URL verwendet wird, wird eine URL Contextfactory verwendet.

Diese Standardregel kann mit Hilfe eines Aufrufes von NamingManager.setInitialContextFactoryBuilder() modifiziert werden.

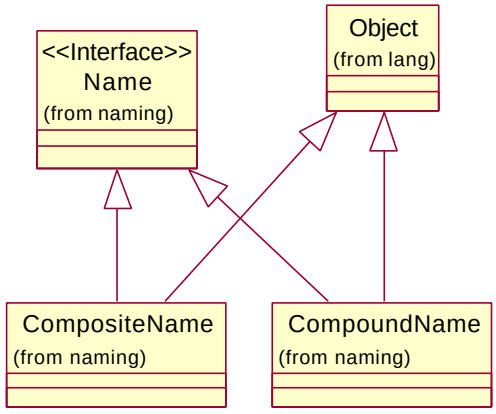
Falls der initiale Context nicht instanziert werden kann, wird eine NoInitialContextException geworfen.

Falls die Umgebungsvariable "java.naming.factory.initial" nicht null ist, wird der InitialContext Konstruktor versuchen, den initialen Kontext gemäss der dort vorhandenen Beschreibung zu konstruieren.

Eine InitialContext Instanz braucht nicht zum vornherein synchronisiert zu werden. Der initiale Context enthält mehrere Verbindungen, mit denen ein Client an ein oder mehrere Namensysteme gebunden werden, beispielsweise URLs oder DNS.

1.2.10.5. Namen

Das `Name` Interface repräsentiert einen generischen Namen - eine geordnete Sequenz von Komponenten. Wie Sie oben gesehen haben kennt die `Context` Schnittstelle zu jeder Methode, die Argumente vom Typus `Name` besitzt, auch eine Methode, mit gleichem Namen, mit einem `String` Argument.



Die Versionen mit dem `Name` Argument sind immer dann sinnvoll, wenn die Anwendung Namen manipulieren muss:

zusammensetzen von Namen, Vergleich von Namen und so weiter;
die Versionen mit `String` sind immer dann einsetzbar, wenn die Applikation relativ einfach ist und es um das Lesen oder Nachschlagen eines Namens oder des dazugehörigen Objektes geht.

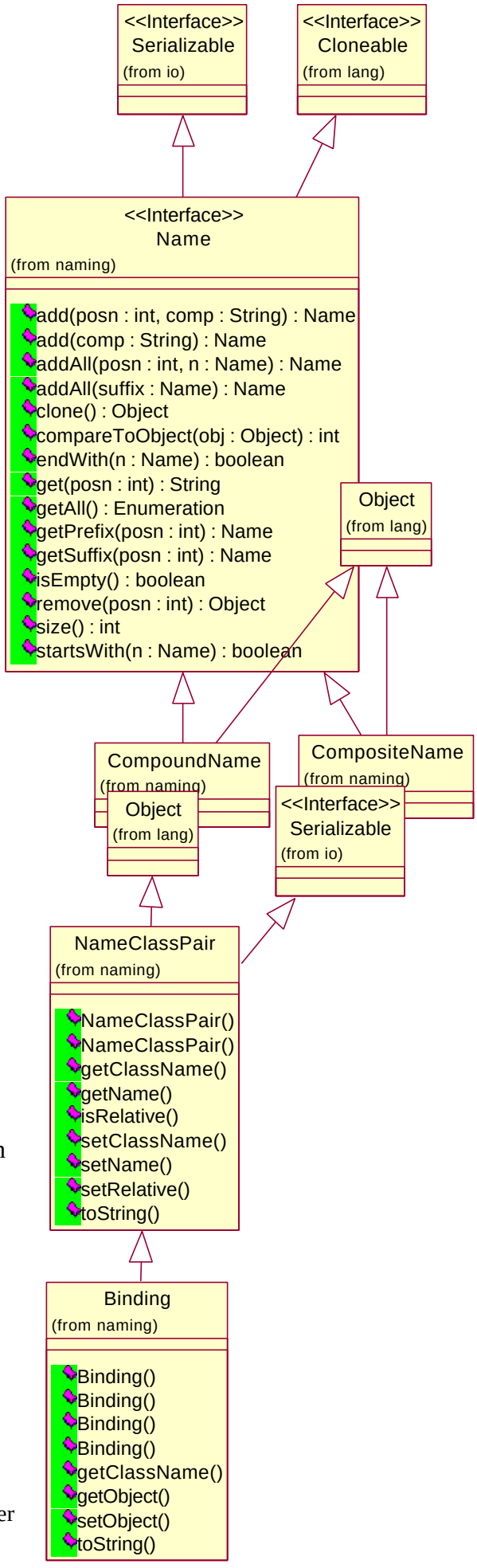
Der `String` Namesparameter stellt einen zusammengesetzten Namen dar; der `Name` Parameter kann einen *composite name* oder einen *compound nam*e darstellen:

Die `CompositeName` Klasse repräsentiert eine Sequenz von Namen (atomaren oder zusammengesetzten) aus mehreren Namensräumen.

Die `CompoundName` Klasse repräsentiert hierarchische Namen aus einem einzigen Namensraum. Man kann auch einen Namensparser einsetzen, um zusammengsetzte Namen zu manipulieren:

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

Ein Namensraum Browser wäre eine Anwendung, welche Namen syntaktische bearbeiten muss.



1.2.10.6. Bindings

`Context.lookup()` ist die wahrscheinlich am meisten verwendete Methode. Die Context Implementation kann ein Objekt eines beliebigen Typs zurück liefern.

Beispiel:
der Client kann ein `Printer` Objekt nachschauen und anschliessend mit dessen Hilfe irgend etwas ausdrucken:

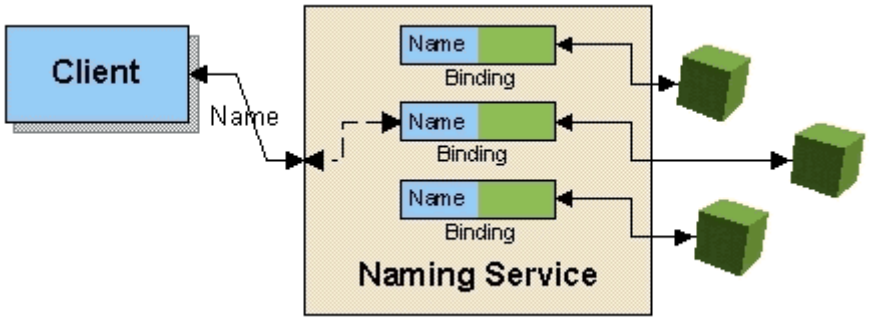
```
Printer printer = (Printer)
ctx.lookup("treekiller");
printer.print(report);
```

`Context.listBindings()` liefert eine Liste der Bindungen (Name-Objekt). Jede Bindung enthält den Namen der gebundenen Objekts, den

Namen der Klasse des Objekts und das Objekt selbst.

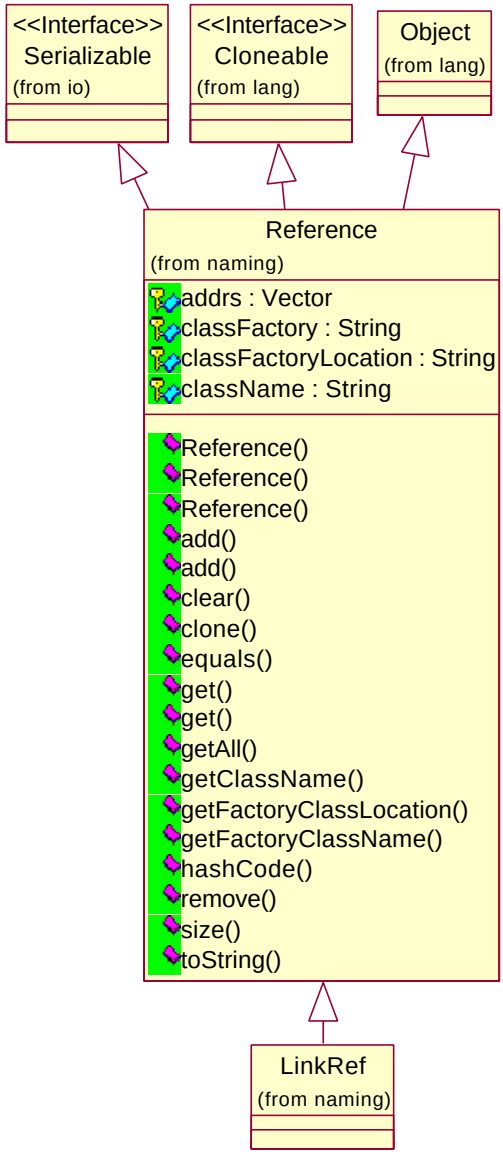
`Context.list()` liefert eine Liste von `NameClassPair` Objekten. Jedes `NameClassPair` enthält den Namen eines Objekts und den Namen der Klasse des Objekts Die `list()` Methode kann beispielsweise eingesetzt werden, um einem Browser Informationen über die Objekte in einem Context zu liefern, die aver das aktuelle Objekt selbst nicht benötigen.

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



1.2.10.7. Reference

Unterschiedliche `Context` Implementationen können unterschiedliche Objekte binden. Die Reference Klasse ist dabei besonders hilfreich, weil eine Referenz ein Objekt darstellt, welches ausserhalb des Verzeichnisses existiert.



`Reference` Objekte geben dem JNDI Client die Illusion, dass ein bestimmter Namens- oder Verzeichnisdienst - zum Beispiel X.500 - ein Objekt eines beliebigen Typs binden kann, zum Beispiel Beispiel Java Objekte.

Wann immer eine Methode, wie zum Beispiel `Context.lookup()` oder `Binding.getObject()` ein `Reference` Objekt zurück liefert, versucht JNDI die Referenz in das Objekt umzuwandeln, welches durch die Referenz repräsentiert wird.

Damit lassen sich auch unterschiedliche Namensysteme zusammen binden: eine Referenz kann auf einen `Context` verweisen.

Wie dies im Detail geschieht, ist in der **JNDI SPI** Dokumentation beschrieben. Objekte, die in der Lage sind, als Referenz dargestellt zu werden, sollten die Schnittstelle `Referenceable` implementieren. Deren einzige Methode — `getReference()` — liefert die Objektreferenz.

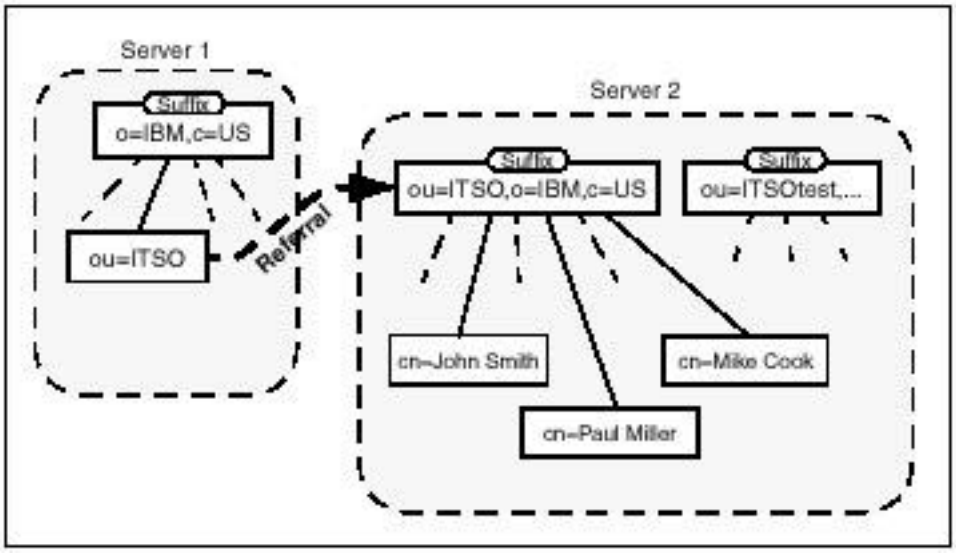
Wenn ein solches Objekt an einen Namen in einem Context gebunden ist, kann dieses Objekt auch ausserhalb des Contextes, im darunter liegenden System, gespeichert werden.

Jede Referenz kann den Namen der Klasse des Objekts sowie dessen Lokation enthalten. (typischerweise ein URL). Zudem enthält eine Referenz eine Sequent von Objekten der Klasse `RefAddr`. Jede `RefAddr` ihrerseits enthält eine “type” Zeichenkette und Daten zur Adressierung, im allgemeinen eine Zeichenkette und ein Byte Array.

Ein Spezialfall von `Reference, LinkRef` wird eingesetzt, um“symbolische” Links in den JNDI Namespace einzufügen. Es enthält den Namen eines JNDI Objekts.

1.2.10.8. Referrals

Einige Namens- / Vezeichnis- Dienste unterstützen *referrals*, mit deren Hilfe eine Client Anfrage an einen andern Server umgeleitet werden kann. Der JNDI Client kann verlangen, dass einer solchen Umleitung / Weiterleitung gefolgt wird, dass sie ignoriert wird oder dass manuell entschieden werden muss.



Referrals werden durch die abstrakte Klasse `abstract class ReferralException` repräsentiert:

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

Wann immer ein Referral angetroffen wird, wird eine `ReferralException` geworfen. Die `getReferralInfo()` Methode liefert Informationen—in einem passenden Format—darüber, wohin diese Umleitung führt. Die Applikation kann diese Information ignorieren oder an ein GUI weiterleiten, um eventuell zusätzlich benötigte Umleitungsinformationen zu erhalten. `skipReferral()` gestattet der Applikation eine Umleitung zu ignorieren.

1.2.11. Das Directory Package — javax.naming.directory

1.2.11.1. Klassen Hierarchie

- class java.lang.**Object**
  - class javax.naming.directory.**BasicAttribute** (implements javax.naming.directory.Attribute)
  - class javax.naming.directory.**BasicAttributes** (implements javax.naming.directory.Attributes)
  - class javax.naming.**InitialContext** (implements javax.naming.Context)
    - class javax.naming.directory.**InitialDirContext** (implements javax.naming.directory.DirContext)
    - class javax.naming.directory.**ModificationItem** (implements java.io.Serializable)
  - class javax.naming.**NameClassPair** (implements java.io.Serializable)
  - class javax.naming.**Binding**
    - class javax.naming.directory.**SearchResult**
    - class javax.naming.directory.**SearchControls** (implements java.io.Serializable)
- class java.lang.**Throwable** (implements java.io.Serializable)
- class java.lang.**Exception**
  - class javax.naming.**NamingException**
    - class javax.naming.directory.**AttributeInUseException**
    - class javax.naming.directory.**AttributeModificationException**
    - class javax.naming.directory.**InvalidAttributeIdentifierException**
    - class javax.naming.directory.**InvalidAttributesException**
    - class javax.naming.directory.**InvalidAttributeValueException**
    - class javax.naming.directory.**InvalidSearchControlsException**
    - class javax.naming.directory.**InvalidSearchFilterException**
    - class javax.naming.directory.**NoSuchAttributeException**
    - class javax.naming.directory.**SchemaViolationException**

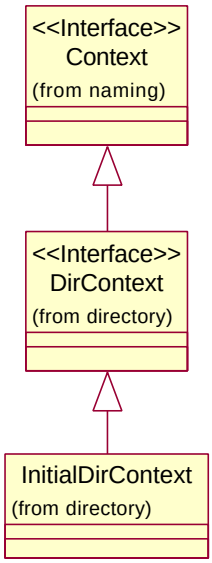
1.2.11.2. Interface Hierarchie

- interface java.lang.**Cloneable**
  - interface javax.naming.directory.**Attribute** (also extends java.io.Serializable)
  - interface javax.naming.directory.**Attributes** (also extends java.io.Serializable)
  - interface javax.naming.**Context**
    - interface javax.naming.directory.**DirContext**
- interface java.io.**Serializable**
  - interface javax.naming.directory.**Attribute** (also extends java.lang.Cloneable)
  - interface javax.naming.directory.**Attributes** (also extends java.lang.Cloneable)

1.2.11.3. Directory Objekte

Das `DirContext` Interface stellt Methoden zur Verfügung, mit deren Hilfe Attribute, welche zu einem Verzeichnisobjekt gehören, abgefragt und modifiziert werden können.

```
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;
...
}
```

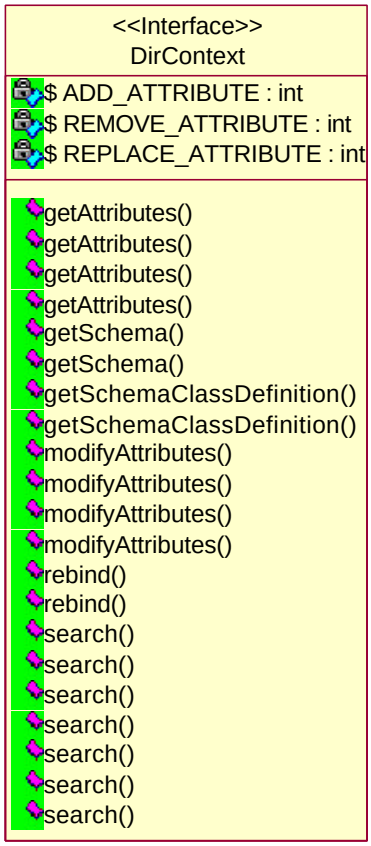


Die `getAttributes()` Methoden liefern alle Attribute eines Directory Objektes.

Attribute werden mit Hilfe der `modifyAttributes()` Methoden modifiziert. Dabei sind drei Operationen definiert:

- `ADD_ATTRIBUTE`
- `REPLACE_ATTRIBUTE`
- `REMOVE_ATTRIBUTE`

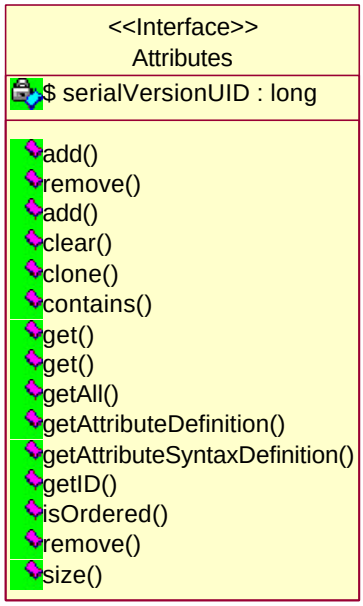
Die `ADD_ATTRIBUTE` Methode fügt Werte einem Attribut hinzu; `REPLACE_ATTRIBUTE` ersetzt ohne Rücksicht auf vorherige Attribute.



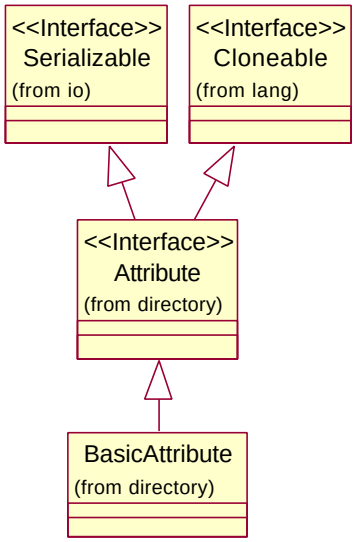

```
public class ModificationItem {
public ModificationItem(int modOp, Attribute
attr) ...;
...
}
```

1.2.11.4. Attribute

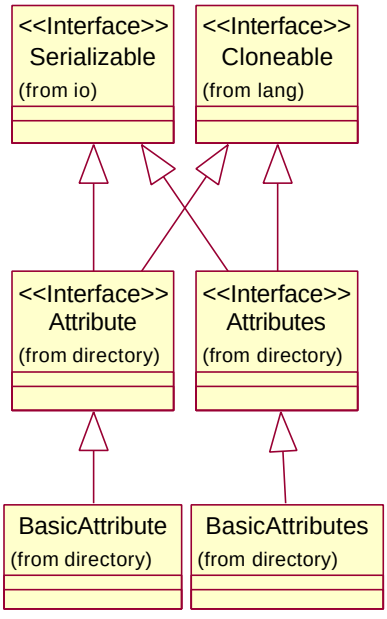
Ein Verzeichnis enthält kein oder mehrere `Attribute` Objekte. Jedes Attribute wird mit Hilfe einer Zeichenkette und keinem oder mehreren Werten beliebigen Typs.




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



Die Attributwerte können geordnet oder ungeordnet sein. Falls die Werte geordnet sind, darf jeder Wert nur einmal auftreten, sonst sind die selben Werte mehrfach erlaubt.



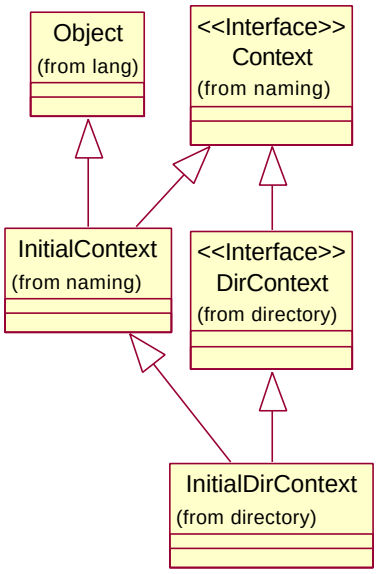
Attribute werden mit Hilfe des `Attributes` **`(Mehrzahl)`** Interfaces gruppiert.

```
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 stellt Implementationen für diese zwei Interfaces zur Verfügung, `BasicAttribute` und `BasicAttributes`. Service Provider und Applikationen können auch ihre eigene Implementation definieren. Mit `DirContext.modifyAttributes()` modifiziert den Verzeichniseintrag.

1.2.11.5. Directory Objekte als Naming Contexts

Das `DirContext` Interface kann auch als Namenskontext eingesetzt werden, da es das Interface `Context` erweitert.

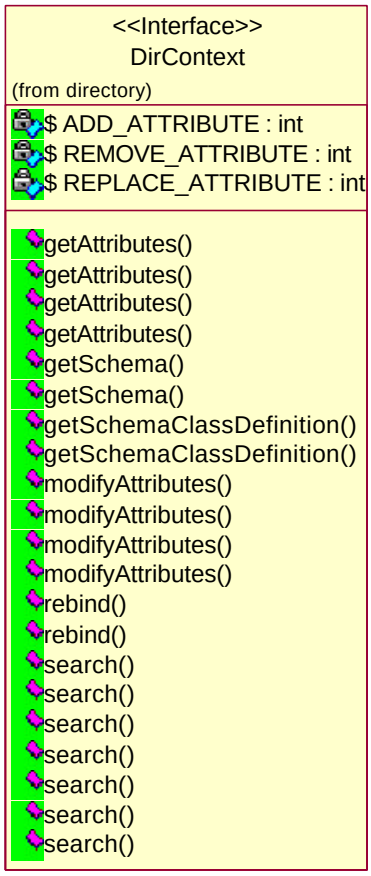


Damit können wir hybride Operationen (Namen und Verzeichnis) in einer einzigen atomaren Operation ausüben:

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

1.2.11.6. Der Initial Context

Eine Applikation, welche Verzeichnisoperationen durchführen muss, kann `InitialDirContext` an Stelle von `javax.naming.InitialContext` einsetzen, um einen initialen Context zu kreieren, wie man aus obigem Diagramm entnehmen kann:

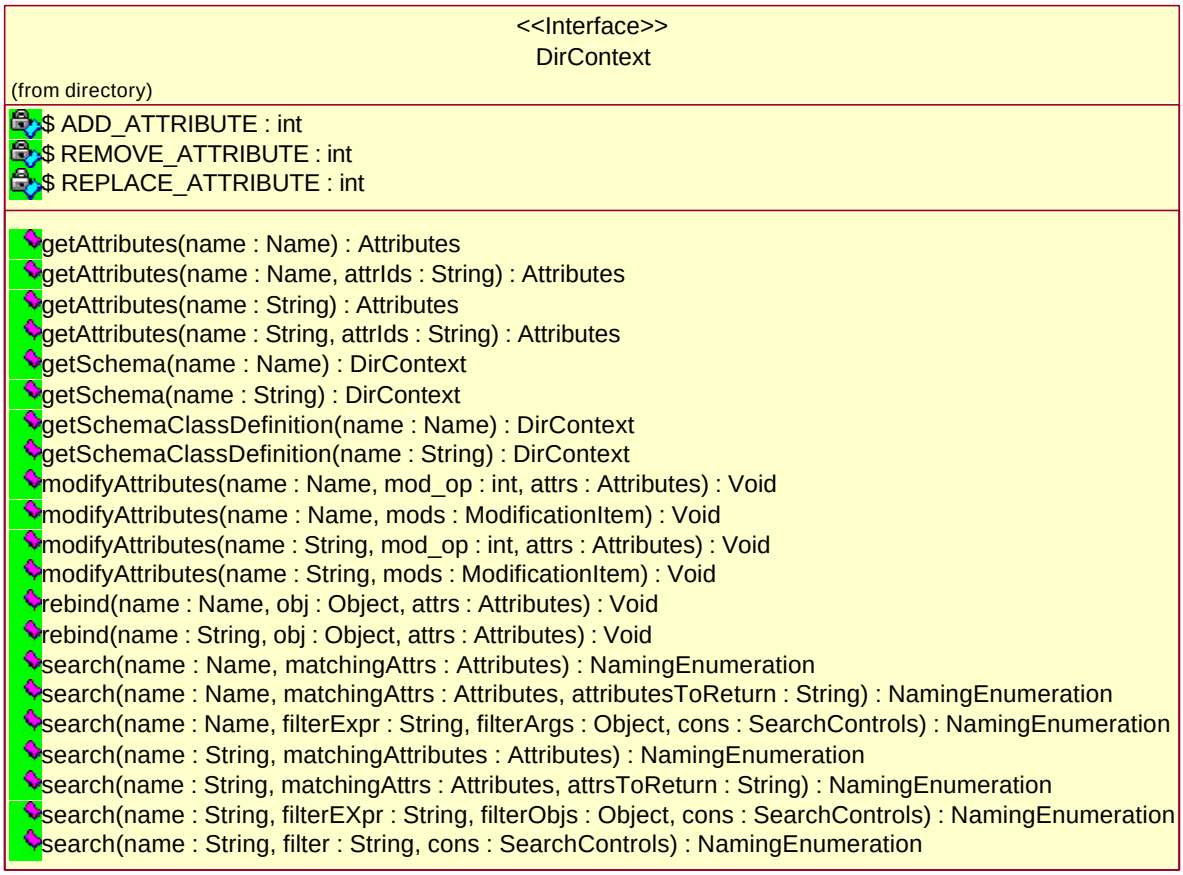



```
public class InitialDirContext
extends InitialContext implements
DirContext {
public InitialDirContext() ...;
...
}.
```

1.2.11.7. Suchen

Das `DirContext` Interface unterstützt inhaltsbasiertes Suchen von Verzeichnissen. Die `search` Methoden liefern zum Beispiel das Verzeichnis und allfällig gefundene Attribute (siehe nächste Seite für die vollständigen Signaturen der Methoden).

```
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;
...
}
```

In komplexeren Fällen kann man auch Filter und zusätzliche Kontrollinformationen spezifizieren, zum Beispiel gemäss Internet RFC 2254 für LDAP.

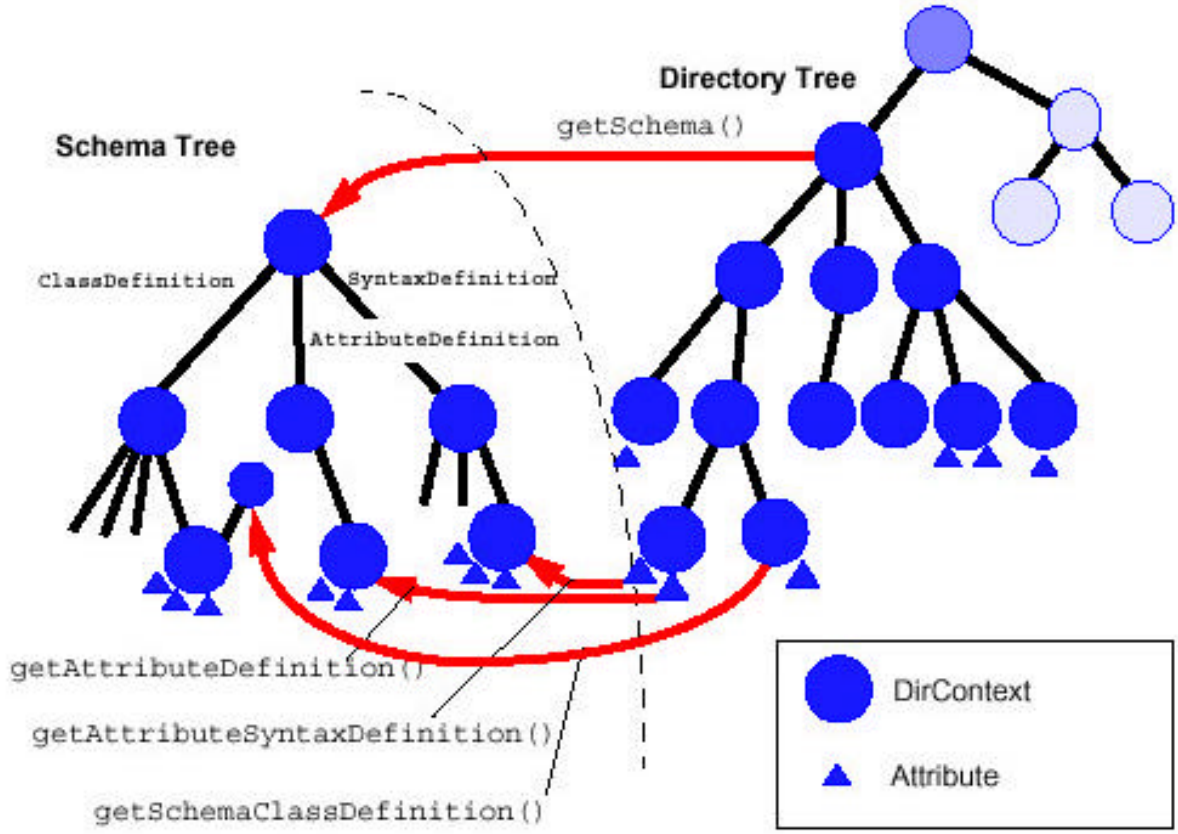
```
.
public interface DirContext extends Context {
...
public NamingEnumeration search(Name name, String filter,
                                                SearchControls ctls)
throws NamingException;
public NamingEnumeration search(Name name, String filter,
                         Object[] filterArgs,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

1.2.11.8. Schema

Ein Schema beschreibt die Regel, welche die Struktur eines Namensraumes beschreibt. Schemas gibt es auf unterschiedlichen Ebenen, sehr fein und sehr grob. Anschaulich stellt man ein Schema als Baum dar. Daher können wir die bereits in JNDI vorhandenen Darstellungen, Objekte, Interfaces und Methoden einsetzen.

`DirContext.getSchema()` liefert die Wurzel des Schemabaumes

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



siehe
getAttributeDefinition() und getAttributeSyntaxDefinition()Methoden:

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

1.2.12. Das Event Package — javax.naming.event

Das `javax.naming.event` Package enthält Klassen und Schnittstellen, welche Event Notification in Namens- und Verzeichnis- Diensten verwenden.

1.2.12.1. Klassen Hierarchie

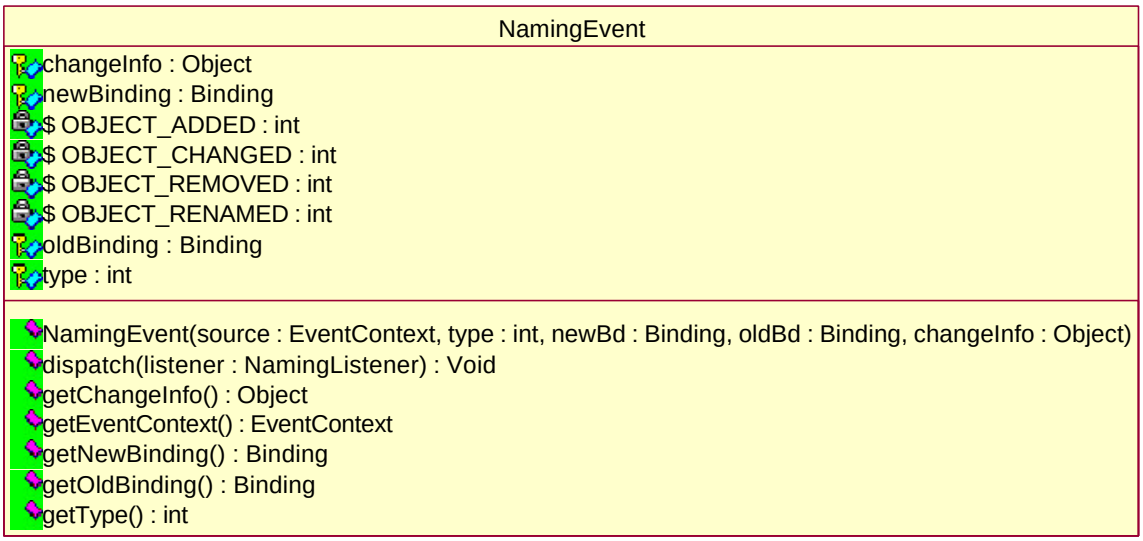
- class java.lang.**Object**
- class java.util.**EventObject** (implements java.io.Serializable)
  - class javax.naming.event.**NamingEvent**
  - class javax.naming.event.**NamingExceptionEvent**

1.2.12.2. Interface Hierarchie

- interface javax.naming.**Context**
  - interface javax.naming.directory.**DirContext**
  - interface javax.naming.event.**EventDirContext** (also extends javax.naming.event.EventContext)
  - interface javax.naming.event.**EventContext**
  - interface javax.naming.event.**EventDirContext** (also extends javax.naming.directory.DirContext)
- interface java.util.**EventListener**
  - interface javax.naming.event.**NamingListener**
  - interface javax.naming.event.**NamespaceChangeListener**
  - interface javax.naming.event.**ObjectChangeListener**

5.3.1 Naming Events

Ein `NamingEvent` Objekt repräsentiert ein Ereignis, welches durch einen Namens / Verzeichnis Dienst generiert wird.




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

Die Attribute definieren den Typus des Ereignisses : `NamingEvent` definiert vier Typen von Events:

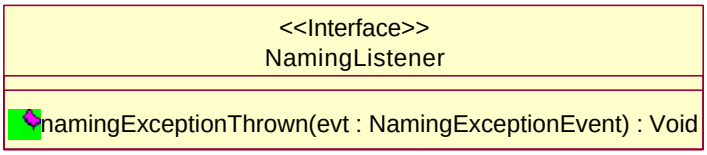
```
OBJECT_ADDED
OBJECT_REMOVED
OBJECT_RENAMED
OBJECT_CHANGED
```

Diese Typen können in zwei Kategorien eingeteilt werden:

- solche, die den Namensraum verändern (hinzufügen / entfernen / umbenennen eines Objekts)
- solche, die den Inhalt eines Objekts beeinflussen

1.2.12.3. Naming Listeners

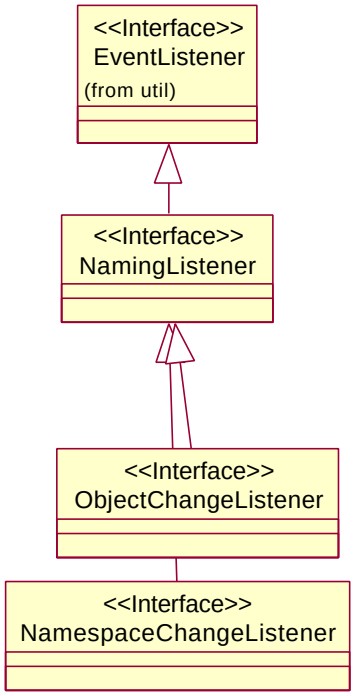
Ein *naming listener* ist ein Objekt, welches auf `NamingEvent`s reagiert. Diese Objekte werden durch das Interface `NamingListener` beschrieben.



Jede Kategorie von `NamingEvent` wird durch einen entsprechenden Subtyp von `NamingListener` beschrieben:

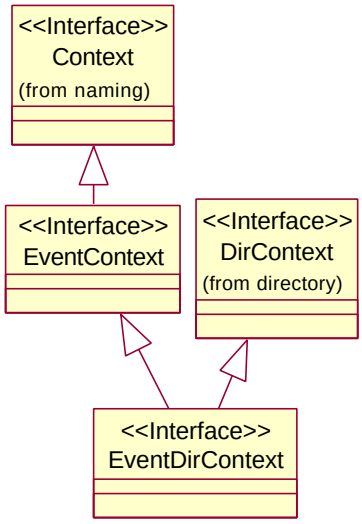
- der `NamespaceChangeListener` stellt einen Listener dar, der sich für Änderungen des Namensraumes interessiert.
- der `ObjectChangeListener` interessiert sich für Änderungen des Inhalrs der Objekte.

Ein Listener kann das eine oder das andere dieser zwei Interfaces implementieren. Je nach Ereignis, für die sich ein Objekt interessiert, wird das eine oder das andere Interface implementiert.



1.2.12.4. Event Registration und Deregistration

Die Interfaces `EventContext` und `EventDirContext` erweitern `Context` und (im Falle von `EventDirContext`) `DirContext` Interfaces, um Ereignisse zu registieren und zu deregistrieren.




```
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;
}
```

Wie die Methoden im Interface `Context` wird die Methode `addNamingListener()` überladen durch eine Methode, welche eine Zeichenkette als Argument akzeptiert.

Der Parameter `name` wird als *target* bezeichnet.
Der Parameter `scope` spezifiziert für welches Objekt die Registration gilt (das von `target` bezeichnete Objekt, das Childobjekt oder der gesamte Subtree ab dem Objekt `target`).

Eine Applikation kann die Methode `targetMustExist()` verwenden, um zu überprüfen, ob ein `EventContext` die Registration eines nichtexistierenden Targets unterstützt.

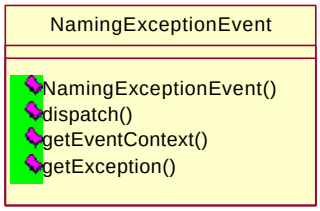
```
public interface EventDirContext extends EventContext, DirContext {
public void addNamingListener(Name target,String filter,
                        SearchControls ctls, NamingListener l)
throws NamingException;
public void addNamingListener(Name target,String filter,
      Object[] filterArgs, SearchControls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

Das `EventDirContext` Interface erweitert das `EventContext` und `DirContext` Interfaces um einem Listener zu erlauben, sich für Objekte zu interessieren / einzutragen, welche mit Hilfe eines Search Filters identifiziert werden (Internet RFC 2254).

Auch hier wird die Methode (wie im `DirContext` Interface) `addNamingListener()` überladen (String Argument als Namen).

1.2.12.5. Exception Handling

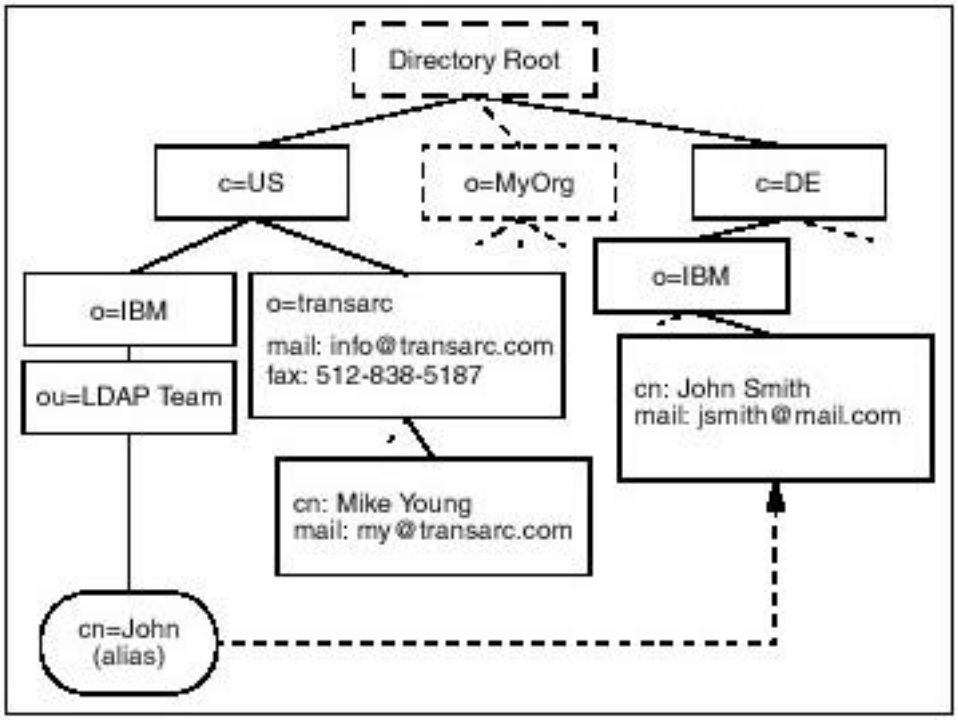
Ein Listener kann sich für bestimmte Ereignisse interessieren und eintragen. Falls nun ein Fehler beim Eintreffen des Ereignisses eintritt, wird der Listener aus der Liste der Interessenten entfernt und eine `NamingExceptionEvent` geworfen.




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

1.2.13. Das LDAP Package — javax.naming.ldap

Das `javax.naming.ldap` Package enthält Klassen und Interfaces für den Einsatz von LDAP v3-specifischen Funktionen, welche im Package `javax.naming.directory` nicht enthalten sind. Wie eine LDAP Adresse aufgebaut ist ersehen Sie aus dem folgenden



Beispiel. Die Bedeutung der Felder ist in der unteren Tabelle wiedergegeben.

Die meisten JNDI Applikationen, welche LDAP benutzen, kommen in der Regel mit der

Attribute Type	String
CommonName	CN
LocalityName	L
StateOrProvinceName	ST
OrganizationName	O
OrganizationalUnitName	OU
CountryName	C
StreetAddress	STREET
domainComponent	DC
userid	UID

Funktionalität vom `javax.naming.directory` aus.

1.2.13.1. Klassen Hierarchie

- class java.lang.**Object**
  - class javax.naming.ldap.**ControlFactory**
- class java.util.**EventObject** (implements java.io.Serializable)
  - class javax.naming.ldap.**UnsolicitedNotificationEvent**
  - class javax.naming.**InitialContext** (implements javax.naming.Context)
    - class javax.naming.directory.**InitialDirContext** (implements javax.naming.directory.DirContext)
    - class javax.naming.ldap.**InitialLdapContext** (implements javax.naming.ldap.LdapContext)
- class java.lang.**Throwable** (implements java.io.Serializable)
- class java.lang.**Exception**
  - class javax.naming.**NamingException**
  - class javax.naming.**ReferralException**
  - class javax.naming.ldap.**LdapReferralException**

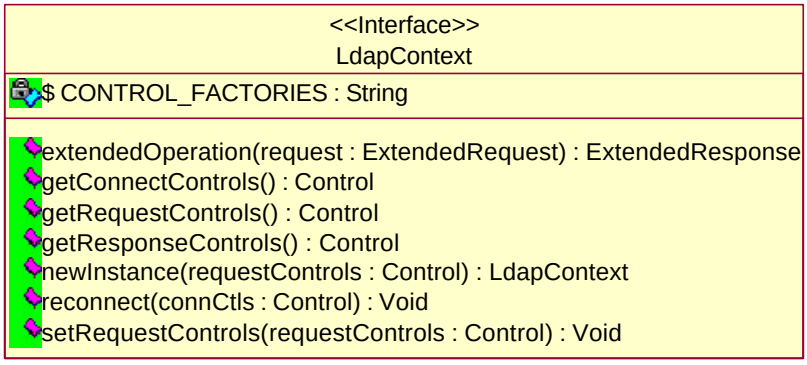
1.2.13.2. Interface Hierarchie

  - interface javax.naming.**Context**
    - interface javax.naming.directory.**DirContext**
    - interface javax.naming.ldap.**LdapContext**
- interface java.util.**EventListener**
  - interface javax.naming.event.**NamingListener**
  - interface javax.naming.ldap.**UnsolicitedNotificationListener**
  - interface javax.naming.ldap.**HasControls**
  - interface javax.naming.ldap.**UnsolicitedNotification** (also extends javax.naming.ldap.ExtendedResponse)
- interface java.io.**Serializable**
  - interface javax.naming.ldap.**Control**
  - interface javax.naming.ldap.**ExtendedRequest**
  - interface javax.naming.ldap.**ExtendedResponse**
  - interface javax.naming.ldap.**UnsolicitedNotification** (also extends javax.naming.ldap.HasControls)

Diese Klassen und deren Methoden werden lediglich für spezielle Anwendungen benötigt, welche vertiefte LDAP Funktionalität benötigen.

1.2.13.3. Erweiterte LDAP Funktionalität

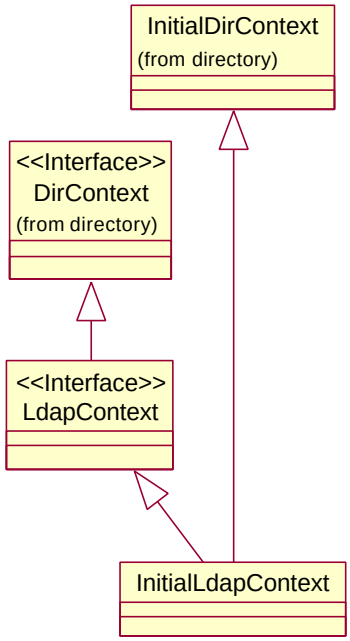
Neben den Standardoperationen wie `search` und `modify` spezifiziert das LDAP v3 Protokoll (Internet RFC 2251) Möglichkeiten, Methoden zwischen dem Client und dem Server zu definieren. Diese werden als *extended operations* bezeichnet. Eine 'extended operation' kann durch ein Standardisierungsgremium wie IETF oder durch einen Hersteller



spezifiziert werden.

Das `LdapContext` Interface definiert eine Methode, um erweiterte Methoden auführen zu können

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



1.2.13.4. Controls

Das LDAP v3 Protokoll (Internet RFC 2251) erlaubt es einen Reques oder eine Response zu erweitern, mit Hilfe von zu definierenden Modifiers, die man als *control*s bezeichnet.

Controls, welche mit Anfragen / request benutzt werden, bezeichnet man als *request controls* und jene welche mit den Responses verwendet werden, als *response control*s. Controls werden entweder durch eine Organisation oder einen Hersteller definiert.

JNDI klassifiziert Request Controls in zwei Kategorien:
• *connection* request controls: dieses beeinflussen die Art und Weise wie eine Connection kreiert wird
• *context* request controls: diese beeinflussen die Context Methoden

Connection Request Controls werden zum Aufbauen oder Wiederaufbauen einer Verbindung mit einem LDAP Server benötigt.

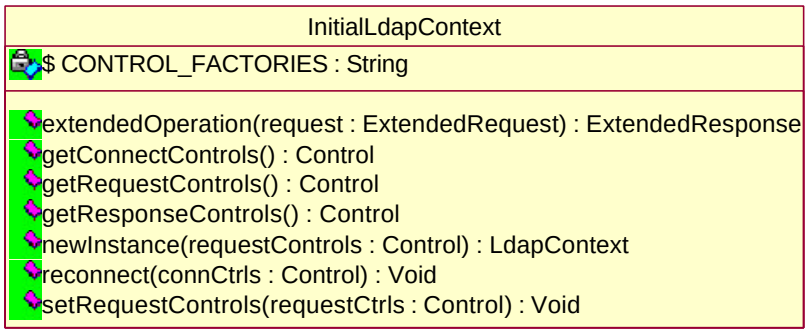
Context Request Controls werden von allen andern LDAP Methoden eingesetzt, welche Verbindungen zu einem LDAP Server benutzen.

Diese Unterscheidung wurde nötig, weil LDAP als Protokoll auf einem sehr hohen Abstraktions- Level sich nicht direkt um die verbindungen kümmert sondern diese Aufgabe an Services / Dienste delegiert.

```
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;
}
```

1.2.13.5. Der Initiale Context

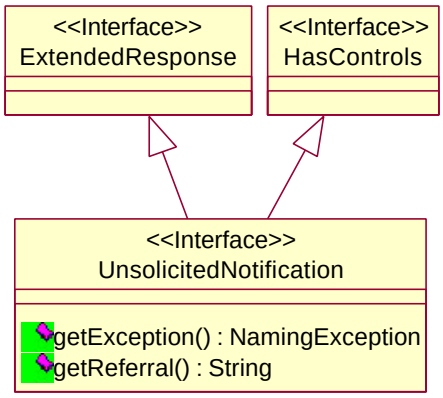
Eine Applikation, welche LDAP 'extended operations' oder 'controls' ausführt, kann den `InitialLdapContext` anstelle von `javax.naming.InitialContext` oder `javax.naming.directory.`




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

1.2.13.6. Unsolicited Notifications

Neben dem normalen Request/Response Interaktionsstil zwischen Clients und Servers kann das LDAP v3 Protokoll sogenannte *unsolicited notification*s—Messages, welche asynchron vom Server zum Client gesandt werden, also nicht als Antwort auf eine Client Anfrage.



JNDI unterstützt 'unsolicited notifications' mit Hilfe des Event Modells aus dem `javax.naming.event` Package.

1.2.14. Konfiguration

1.2.14.1. Environment Properties

Einige JNDI Applikationen benötigen eine Möglichkeit, um mit Namens- und Verzeichnisdiensten in variablen Umgebungen und Präferenzen kommunizieren zu können.
Beispiel:
Sicherheitsanforderungen können am Besten als Properties eventuell in einer Propertiesdatei definiert werden. Die Anwendungsprogramme werden dadurch vor laufenden Änderungen bewahrt.

Es gibt in JNDI verschiedene Arten von *environment properties:*

- Standard JNDI Umgebungsparameter:
  Diese Parameter sind unabhängig vom Provider definiert und stehen immer zur Verfügung.
  Beispiel:
  "java.naming.security.principal" wird benutzt, um zu spezifizieren, womit die Security spezifiziert werden kann.
- Service-spezifische Umgebungsparameter:
  Diese werden zur Implementation spezieller Dienste und Protokolle eingesetzt. Sie haben einen Präfix "java.naming.*service*.", wobei *service* der Name des angebotenen Services ist.
  Beispiel:
  "`java.naming.ldap`." definiert LDAP-spezifische Umgebungseigenschaften.
- Feature-spezifische Umgebungsparameter:
  Diese sind allen Providern gemeinsam, werden also einfach auf Grund der Zusatzdienste bezeichnet. Sie haben den Präfix "java.naming.*featur*e.", wobei *feature* der Name des Zusatzdienstes bezeichnet.
  Beispiel:
  "`java.naming.security.sasl`." wird eingesetzt zum setzen von SASL-spezifischen Umgebungsparametern.
- Provider-spezifische Umgebungsparameter:
  Diese Parameter sind Provider- spezifisch und werden nict von allen Providern angeboten.
  Beispiel:
  Sun's LDAP Provider wird im wesentlichen mit Hilfe des Packages `com.sun.jndi.ldap` beschrieben. Spezifische Parameter für diesen LDAP Provider haben den Präfix "`com.sun.jndi.ldap`.".

Die einzelnen möglichen Parameter sind in der API Spezifikation wieder gegeben (Anhang).

1.2.15. Szenarios

Nach soviel Theorie möchten Sie sicher wissen wozu das Ganze gut ist.
Hier einige mögliche Einsätze für JNDI:

1.2.15.1. User Authentisierung

In gesicherten Systemen muss ein Benutzer sich authentisieren, gegenüber einem Rechner, einem Netzwerk, einem Dienst.
Beispiel:
im Falle eines Unix Netzwerkes muss sich der Benutzer mit dem Benutzernamen und einem Passwort anmelden; falls er SSL verwenden möchte, muss er zudem ein X.509 Zertifikat liefern. Diese Authentisierungsinformationen kann man als Attribute des Benutzers abspeichern. Die Benutzer selbst kann man in eine Benutzerhierarchie einordnen.

Das System schaut dann zum Beispiel das Attribut "password" zum Benutzer nach und verifiziert das eingegebene Passwort mit dem abgespeicherten.

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

1.2.15.2. Electronic Mail

Bei einem e-Mail System ist die Verbindung zwischen dem Benutzer und seiner e-Mail Adresse von speziellem Interesse.

```
NamingEnumeration matches =
deptCtx.search("user", new BasicAttributes("name", "Hans Heinrich"));
while (matches.hasMore()) {
SearchResult item = (SearchResult) matches.next();
Attributes info = item.getAttributes();
/* Anzeige der Attribute */
...
}
```

Damit lässt sich auch ein personalisiertes e-mail Buch aufbauen.

1.2.15.3. Datenbanken

In Datenbankanwendungen, speziell verteilten Anwendungen, muss man oft den datenbankserve suchen. Die DB- Server lassen sich leicht in ein Verzeichnis eintragen, gemäss Funktion oder geografischer Lokation.

```
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");
...
}
```

1.2.15.4. Browsing

Oft benötigt man in Eingabeprogrammen eine Auswahl aus vorgegebenen Eingabewerten. Diese lassen sich hierarchisch anordnen und mit Hilfe eines Contexts verwalten:

```
Context ctx = new InitialContext();
while (ctx != null) { // Anzeige eines levels
      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());
      }
      // nächste Ebene
      String target = input.readLine();
      try {
            ctx = (Context)ctx.lookup(target);
      } catch (NamingException e) {
            // Fehlerbehandlung
      } catch (ClassCastException e) {
            // keine Context
      }
}
```

1.2.15.5. Netzwerk Printing

Wie bereits weiter vorne im Skript erläutert, kann man Drucker in einen Namensraum oder einen Verzeichnisbaum eintragen:

```
interface Printer {
void print(InputStream data) throws PrinterException;
...
}
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("Kann Printer nicht finden:" + e);
      } catch (ClassCastException e) {
            System.err.println(printerName + " ist falsch");
      }
}
```

JAVA NAMING AND DIRECTORY SERVICES