제 27 장 네이밍과 디렉토리 인터페이스

네이밍은 어떤 객체에 고유한 이름을 부여하여 찾기 쉽도록 하는 것을 말하며, 디렉토리는 네이밍을 확장하여 각 객체가 여러 가지 속성을 가질 수 있는 경우를 나타낸다.
주변에서 사용하는 간단하는 네이밍의 예로는 DNS 와 같이 사람이 알기 힘든 IP 주소에 도메인과 호스트 이름을 부여하거나, 파일 시스템과 같이 시스템의 파일 핸들에 파일 경로를 부여하는 경우를 들 수 있다.
디렉토리의 경우, 일반적인 디렉토리 기능에 사용하는 인터넷 표준인 LDAP 을 비롯하여 네트웍 서비스를 관리하는 노벨 디렉토리 서비스나 솔라리스의 NIS 등의 경우를 볼 수 있다.
자바에서는 네이밍과 디렉토리를 사용하는 표준화된 인터페이스를 자바 네이밍과 디렉토리 인터페이스(Java Naming and Directory Service, JNDI)라는 이름으로 제공한다.

이 장에서는 다음을 다룬다.

- 네이밍
- 디렉토리
- 이벤트 처리
- 서비스 제공자

27.1. JNDI API

JNDI 는 네이밍과 디렉토리에 대한 통일된 자바 인터페이스를 제공하기 위해 만들어졌으며, 크게 세 가지 부분으로 나뉜다.

(1) 네이밍 : 네이밍은 다양한 객체에 사람이 이해하기 쉬운 이름을 붙이는 기능이다. 파일 시스템, DNS, LDAP 등에서는 각 이름이 계층 구조를 가지며, JNDI 에서는 각 이름의 계층 구조를 지원한다.
(2) 디렉토리 : 디렉토리는 네이밍을 확장하여 객체에 속성을 부여할 수 있다. 따라서, 객체의 속성을 본다거나, 속성의 조건에 맞는 객체를 검색하는 등의 기능을 지원한다.
(3) 서비스 제공자 : JNDI 서비스를 구현한 라이브러리들이며, JDK 에는 기본으로 LDAP, DNS, RMI 레지스트리, COS 네이밍을 지원하는 서비스 제공자들이 각각 포함되어 있다.

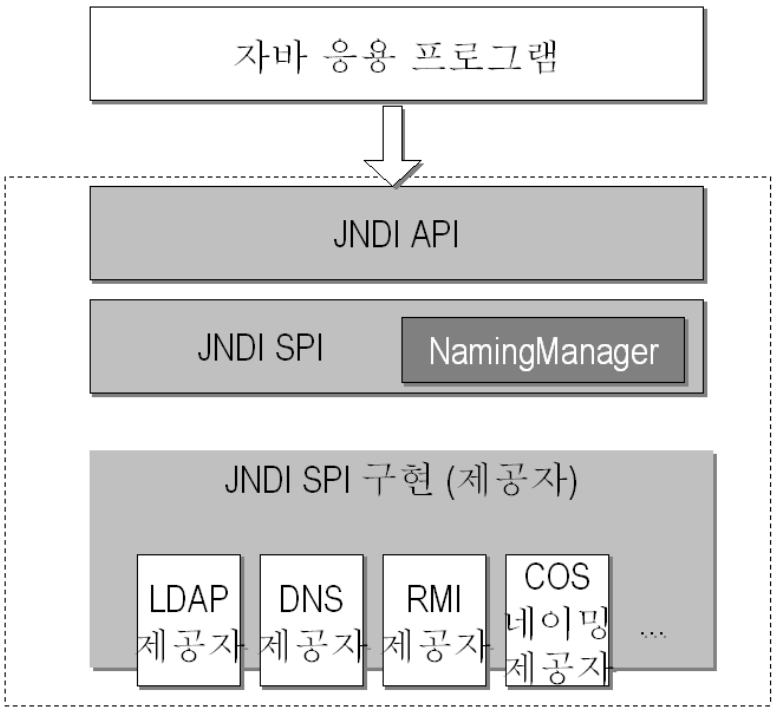
JNDI 에 관련된 정보를 찾아보려면 다음 JNDI 홈페이지를 참고할 수 있다.

```
http://java.sun.com/products/jndi
```

JNDI 홈페이지에서는 JDK 에 포함되지 않은 몇 가지 서비스 제공자 라이브러리들을 다운로드할 수 있는데 이 장에서는 이 중 파일 시스템 서비스 제공자를 예제로 사용할 것이므로 다운로드하기 바란다. CD-ROM 부록에도 포함되어 있다.
그리고, 다음 JNDI 튜토리얼이 많은 도움이 될 것이다.

http://java.sun.com/products/jndi/tutorial/index.html

다음 그림은 JNDI 의 기본 구성을 보여준다.



[그림 1 – JNDI 구성]

27.1.1. 네이밍

27.1.1.1. 네이밍 연산

네이밍은 사람이 이해하기 쉬운 이름을 각 객체에 붙여주는 기능을 나타낸다.
네이밍 서비스에서는 이름을 사용하여 실제 객체를 찾아볼 수 있으며, 계층 구조에 따라 이름을 관리할 수 있다.
다음은 네이밍에서 사용되는 몇 가지 용어이다.

(1) 바인딩(binding) : 객체를 어떤 이름과 연결시키는 것을 바인딩이라고 부른다. DNS 의 경우에는 IP 주소를 호스트 이름으로 바인딩한다.
(2) 문맥(context) : 바인딩된 이름과 객체 쌍을 가지는 존재를 문맥이라고 부른다. 예를 들어 파일 시스템의 경우 디렉토리가 문맥에 해당하여 하위 문맥과 파일 엔트리들을 가진다. 서로 다른 문맥에서는 같은 이름을 허용한다.
(3) 엔트리 : 네이밍 혹은 디렉토리에서 문맥에 바인딩된 객체를 엔트리라고 부른다.
(4) 네이밍 시스템 : 동일한 종류의 문맥들의 연결된 집합을 네이밍 시스템이라고 부른다. 또, 네이밍 시스템에서 사용되는 이름들을 이름 공간(namespace)이라고 한다.
(5) 초기 문맥(initial context) : JNDI 에서는 모든 네이밍과 디렉토리 연산인 어떤 문맥에 대해 상대적으로 정의된다. 초기 문맥은 네이밍과 디렉토리 연산을 수행하기 위한 시작점이 되는 문맥이다.
(6) 참조(reference) : 지정한 객체를 지정할 때 직접 객체를 직렬화하여 저장하지 않고 객체를 만들 수 있는 정보만을 제공하는 경우에 사용하는 것으로 객체의 생성 정보를 담고 있다.

네이밍에 관련된 클래스와 인터페이스들은 javax.naming 패키지에 정의되어 있다.
JNDI 연산은 문맥에 대한 연산이므로, 가장 먼저 해야할 일은 먼저 초기 문맥을 만드는 것이다. InitialContext 클래스가 초기 문맥을 나타낸다.
초기 문맥을 가져오려면 초기 문맥을 만드는 팩토리 클래스를 포함한 필요한 정보를 지정해야 한다. 별도의 파일로 이 정보들을 지정하거나 InitialContext 클래스의 생성자에 Hashtable 객체로 지정할 수 있다. JNDI 서비스 제공자에 따라 초기 문맥 팩토리 클래스 외에 여러 가지 정보들을 추가로 필요로 할 수 있다. 표준적인 정보 키들은 Context 클래스에 상수로 정의되어 있다.
다음은 Hashtable 을 사용하여 초기 문맥을 생성하는 예이다.

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
    "com.sun.jndi.fscontext.FSContextFactory");

try {
    Context ctx = new InitialContext(env);
  } catch (NamingException e) {
    e.printStackTrace();
  }
```

문맥을 나타내는 Context 인터페이스에는 문맥을 설정하는 데 필요한 부가 정보의 키 값들

이 상수로 선언되어 있다. INITIAL_CONTEXT_FACTORY 상수는 초기 문맥 팩토리 클래스를 나타내는 문자열이며 실제로는 java.naming.factory.initial 이라는 값을 가진다.
이 값을 jndi.properties 라는 별도의 파일에 지정할 수도 있다.
즉, 다음 내용을 가진 jndi.properties 파일을 만든 다음, InitialContext 클래스의 기본 생성자를 호출할 수 있다.

```
java.naming.factory.initial=com.sun.jndi.fscontext.FSContextFactory
```

소스 코드는 다음과 같이 변경된다.

```
try {
    Context ctx = new InitialContext();
  } catch (NamingException e) {
    e.printStackTrace();
  }
```

실행할 때 jndi.properties 파일은 클래스 경로에 위치해야 한다.

다음은 Context 클래스의 기본 기능들이다.

(1) public Object lookup(Name name) throws NamingException;
주어진 이름에 바인딩된 객체를 찾아준다.
(2) public void bind(Name name, Object obj) throws NamingException;
주어진 이름으로 객체를 바인딩한다.
(3) public void rebind(Name name, Object obj) throws NamingException;
bind() 메소드와 같은 일을 하지만, 이미 주어진 이름이 바인딩된 경우에도 에러를 내지 않고 다시 바인딩한다
(4) public void unbind(Name name) throws NamingException;
주어진 이름을 바인딩 해제한다.
(5) public void rename(Name oldName, Name newName) throws NamingException;
바인딩 이름을 변경한다.
(6) public NamingEnumeration list(Name name) throws NamingException;
문맥의 엔트리들을 나열한다. 반환되는 NamingEnumeration 객체에는 NameClassPair 객체들이 들어 있다.
(7) public NamingEnumeration listBindings(Name name) throws NamingException;
list() 메소드와 유사하지만, 문맥의 엔트리에 해당하는 모든 정보를 가져온다. 백업과 같

은 용도로 사용하기 위해 정의된 메소드이므로 list() 메소드보다 좀더 비싼 연산일 가능성이 높다. 반환되는 NamingEnumeration 객체에는 Binding 객체들이 들어 있다.

(8) public Context createSubcontext(Name name) throws NamingException;
하위 문맥을 생성하여 주어진 이름으로 바인딩한다.

(9) public void destroySubcontext(Name name) throws NamingException;
주어진 이름의 하위 문맥을 삭제한다.

(10) public void close() throws NamingException;
현재 문맥에 할당된 리소스들을 바로 해지한다.

각 메소드들 중 Name 인자를 가지는 메소드들은 Name 자료형 대신에 String 자료형으로 이름을 지정하도록 오버로드된 메소드들을 함께 선언하고 있다.
다음 예제는 파일 시스템 JNDI 서비스 제공자를 사용하여 문맥 연산을 사용하는 방법을 예시하고 있다. 프로그램을 실행하려면 파일 시스템 JNDI 서비스 제공자 라이브러리를 클래스 경로에 포함시켜야 한다.

<예제 1 시작 – NamingEx.java>

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

/**
 * 네이밍 예제
 */

public class NamingEx {
  public static void main(String[] args) {
    Hashtable env = new Hashtable();
     env.put(Context.INITIAL_CONTEXT_FACTORY,
        "com.sun.jndi.fscontext.RefFSContextFactory");

    try {
      Context ctx = new InitialContext(env);

      ctx = (Context) ctx.lookup("c:\\j2sdk1.4.1");
      // 문맥의 엔트리를 나열한다.
      NamingEnumeration enum = ctx.list("");
      while (enum.hasMore()) {
```

```
        NameClassPair entry = (NameClassPair) enum.next();
        System.out.println("entry is " + entry);
      }
      enum.close();

      ctx = (Context) ctx.lookup("c:\\temp");

      // 상대 경로로 하위 문맥 (여기에서는 디렉토리) 생성
      Context subCtx = ctx.createSubcontext("test");
      // 하위 문맥을 testDir라는 이름으로 바인딩
      // 현재 test라는 이름으로도 바인딩되어 있으므로
      // 같은 문맥 엔트리가 두 개의 이름으로 바인딩된다.
      ctx.bind("testDir", subCtx);

      // test 엔트리를 test2로 이름 변경
      ctx.rename("testDir", "test2");

      // test2 엔트리를 삭제
      ctx.unbind("test2");

      // test 하위 문맥 제거
      ctx.destroySubcontext("test");
    } catch (NamingException e) {
      e.printStackTrace();
    }
  }

}
```

<예제 1 끝 – NamingEx.java>

위 프로그램을 실행하는 예이다. JNDI 서비스 제공자 파일들이 클래스 경로에 포함되어야 한다.

```
java -cp .;lib\fscontext.jar;lib\providerutil.jar NamingEx <엔터>
```

27.1.1.2. JNDI 에서의 이름

이름을 나타내는 Name 인터페이스는 크게 두 가지로 나뉜다. 하나는 단일한 이름 공간 안에서 계층 구조를 나타내는 이름들을 합한 이름인 CompoundName 클래스이며, 다른 하나는 서로 다른 이름 공간의 이름들이 복합된 CompositeName 클래스이다.
각 문맥은 문자열 형태로 넘겨진 이름을 파싱하여 Name 객체로 만드는 일을 하는 NameParser 객체를 가지고 있다.
NameParser 는 JNDI 서비스 제공자에 따라 또 이름에 다라, 달라질 수 있다.
일반적으로 CompoundName 의 경우 같은 NameParser 가 사용되며, CompositeName 의 경우에는 서로 다른 NameParser 가 사용될 것이다.
예를 들어, 다음은 LDAP 서비스 제공자에서 사용되는 CompoundName 이다.

```
cn=Michael Jackson,ou=People,o=MyOrg
```

마찬가지로 다음은 파일 시스템 서비스 제공자에서 사용되는 CompoundName 이다.

```
c:\j2sdk1.4.1\docs\api\index.html
```

반면 다음은 LDAP 서비스 제공자와 파일 시스템 서비스 제공자에서 각각 처리되는 부분이 복합된 CompositeName 이다.

```
cn=Michael Jackson,ou=People,o=MyOrg/songs/MP3/Heal the world.mp3
```

JNDI 에서는 자신의 네이밍 시스템에서 처리하지 못하는 부분을 다른 네이밍 시스템으로 넘길 수 있도록 하여, 여러 개의 이름 공간에 걸친 CompositeName 을 지원하는 것을 페더레이션(federation)이라고 한다.

27.1.2. 디렉토리와 LDAP

디렉토리는 네이밍에 속성을 추가한 개념이다. 객체에 바인딩된 속성을 본다거나 속성에 기반하여 객체를 찾는 등의 연산이 추가된다.
또, JNDI 는 일반적인 디렉토리 서비스 규약인 LDAP 을 지원하며, LDAP 규약이 정의하고 있는 검색 필터와 검색 컨트롤 등의 개념도 지원한다.
디렉토리와 LDAP 에 관련된 기본 클래스들은 javax.naming.directory 패키지에 정의되어 있으며, 확장된 LDAP 규약을 지원하기 위한 클래스들이 javax.naming.ldap 패키지에 정의되어 있다.
디렉토리 관련 API 들을 테스트하기 위해서는 디렉토리를 지원하는 서비스 제공자가 필요한데 여기에서는 LDAP 서비스 제공자를 사용하기로 한다. LDAP 서비스 제공자는 JDK 에 기

본 포함되어 있으며, 접속할 LDAP 서버가 필요한데 다음 URL 에서 공개 소스 프로젝트로 제공되는 LDAP 서버를 다운로드할 수 있다.

```
http://www.openldap.org
```

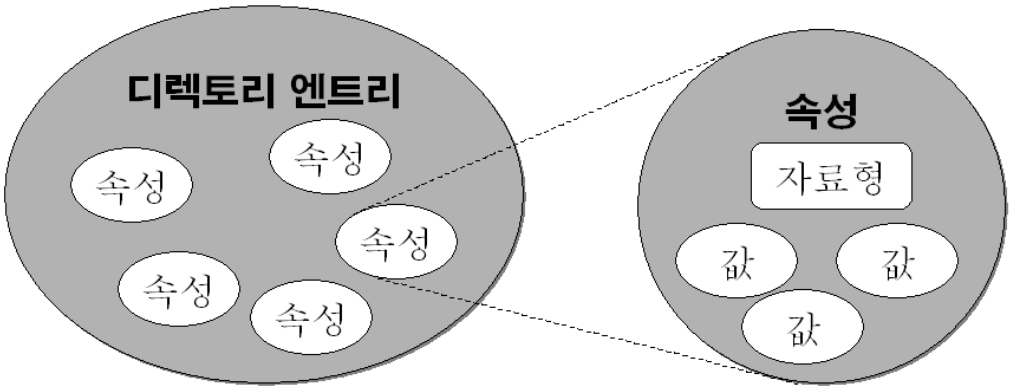
이 책의 예제들은 넷스케이프 디렉토리 서버의 테스트 버전을 다운로드하여 테스트하였다. 다음 URL 에서 테스트 버전을 다운로드할 수 있다. 사용된 버전은 윈도우용 넷스케이프 디렉토리 서버 6.01 이다.

```
http://enterprise.netscape.com/
```

27.1.2.1. 디렉토리 연산

디렉토리는 네이밍을 확장하여 속성 개념을 추가하며, 스키마를 지원한다.

(1) 속성(attribute) : 디렉토리 엔트리들은 속성을 가질 수 있는데 속성은 속성 이름과 속성 값으로 구성된다. 하나의 속성은 속성 이름에 대응하는 값을 여러 개 혹은 가지지 않을 수도 있다. [그림 2]는 디렉토리 엔트리와 속성 그리고 값의 상관 관계를 보여준다.



[그림 2 – 디렉토리 엔트리와 속성, 값의 상관 관계]

(2) 스키마(schema) : 스키마는 이름 공간과 그 안에 저장되는 속성들의 구조를 정의하는 규칙이다. 스키마에는 일반적으로 어떤 종류의 객체를 어디에 추가할 수 있는지, 어떤 필수 속성을 가지는지 혹은 어떤 선택적 속성을 가질 수 있는지 등의 정보가 지정된다. 스키마는 계층 구조로 정의되기 때문에 JNDI 에서는 스키마를 일반 디렉토리 문맥처럼 처리한다. LDAP 규약의 경우, 스키마 정보는 클래스 정의, 속성 정의, 구문 정의, 검색 규칙 등을 포함한다.

(3) 검색(search) : 디렉토리는 속성에 기반한 검색을 지원한다. LDAP 규약의 경우, 검색 필터를 사용하여 검색 기준을 지정할 수 있다.

JNDI 에서 디렉토리는 네이밍의 경우와 마찬가지로 문맥에서 출발한다. 디렉토리에 관련된 객체 자료형들은 javax.naming.directory 패키지에 정의되어 있다.

네이밍에 사용되었던 Context 인터페이스를 확장한 DirContext 인터페이스가 디렉토리 문맥을 나타낸다.

또, 디렉토리 초기 문맥을 나타내는 클래스는 IntialContext 를 상속하는 InitialDirContext 클래스이다.

다음은 DirContext 인터페이스는 Context 인터페이스가 선언하는 여러 메소드들을 상속하면

서 속성 인자를 추가로 지정할 수 있는 버전의 메소드들을 오버로드하여 선언한다. 그외에도 속성, 검색, 스키마에 관련된 메소드들을 선언하고 있다. 각 메소드들 중 인자로 Name 자료형을 받는 메소드들은 Context 의 경우와 마찬가지로 String 자료형을 받는 메소드들을 오버로드하고 있다.

(1) public Attributes getAttributes(Name name) throws NamingException;
지정된 디렉토리 엔트리의 모든 속성을 가져온다.

(2) public Attributes getAttributes(Name name, String[] attrIds) throws NamingException;
지정된 디렉토리 엔트리의 속성들 중 지정된 속성들만 가져온다. 없는 속성이 지정된 경우에는 무시한다.

(3) public void modifyAttributes(Name name, int mod_op, Attributes attrs) throws NamingException;
지정된 디렉토리 엔트리의 속성을 수정한다. 속성의 추가, 삭제, 대체 연산을 지원한다.

(4) public void modifyAttributes(Name name, ModificationItem[] mods) throws NamingException;
지정된 디렉토리 엔트리의 속성을 수정한다. ModificationItem 클래스는 속성 수정 연산과 속성을 지정하는 객체이다.

(5) public DirContext getSchema(Name name) throws NamingException;
지정된 엔트리에 연결된 스키마 정보를 가져온다.

(6) public DirContext getSchemaClassDefinition(Name name) throws NamingException;
지정된 엔트리의 클래스 정의를 담은 스키마 객체를 포함하는 문맥을 가져온다.

(7) public NamingEnumeration search(Name name, Attributes matchingAttributes) throws NamingException;
지정된 문맥 엔트리에서 속성을 만족하는 엔트리를 검색하여 SearchResult 객체들을 포함한 NamingEnumeration 객체를 반환한다. 이때, 엔트리의 모든 속성을 반환한다.

(8) public NamingEnumeration search(Name name, Attributes matchingAttributes, String[] attributesToReturn) throws NamingException;
지정된 문맥 엔트리에서 속성을 만족하는 엔트리를 검색하여 SearchResult 객체들을 포함한 NamingEnumeration 객체를 반환한다. 이때, 엔트리의 지정된 속성들만 반환한다.

(9) public NamingEnumeration search(Name name, String filter, SearchControls cons) throws NamingException;
지정된 엔트리에서 검색 필터를 만족하는 엔트리를 검색하여 SearchResult 객체들을 포함한 NamingEnumeration 객체를 반환한다. SearchControls 클래스는 검색 범위와 검색 결과 개수 제한, 검색 시간 제한 등을 지정할 수 있다.

(10) public NamingEnumeration search(Name name, String filterExpr, Object[] filterArgs, SearchControls cons) throws NamingException;
지정된 엔트리에서 검색 필터를 만족하는 엔트리를 검색하여 SearchResult 객체들을 포

함한 NamingEnumeration 객체를 반환한다. 검색 필터 식이 인자를 포함하고 있으며, 검색 필터 식의 인자에 들어갈 변수를 filterArgs 인자에 지정한다.

javax.naming.ldap 패키지에는 LDAP 규약 버전 3 에 정의된 확장 연산을 지원하는 객체들이 선언되어 있다. 이 확장 연산을 수행하기 위해서는 DirContext 대신에 이를 상속하는 LdapContext 인터페이스, InitialDirContext 대신에 이를 상속하는 InitialLdapContext 클래스를 사용해야 한다.

27.1.2.2. LDAP 서버와의 통신

LDAP 서버와의 통신을 위해 LDAP 서버에 다음과 같은 디렉토리를 생성하여 사용할 기본 자료를 입력한다.
LDAP 서버에 데이터를 임포트하거나 익스포트할 때에는 일반적으로 LDAP 데이터 교환 서식(LDAP Data Interchange Format, LDIF 라고 부른다) 규격을 따르는 문서를 사용한다.
testdata.ldif 파일에는 LDIF 형식으로 작성된 기본 데이터가 들어가 있다.
LDAP 서버에 따라 이 파일을 임포트하기 전에 organization 클래스인 JavaDomain 디렉토리를 미리 생성해야 하는 경우도 있다.
이 데이터에는 JavaDomain 이라는 조직을 루트 디렉토리로 하여 People 과 Groups 라는 단위 조직이 있고, People 에는 세 명의 사람이 있으며, Groups 에는 두 개의 그룹이 들어 있다.
예제 LDIF 파일에서 각 엔트리는 dn 으로 시작하여 속성들을 지정하는 형식으로 구성된다.

<예제 2 시작 – testdata.ldif>

```
dn: o=JavaDomain
o: JavaDomain
objectclass: top
objectclass: organization

dn: ou=People,o=JavaDomain
ou: People
objectclass: top
objectclass: organizationalunit

dn: ou=Groups,o=JavaDomain
ou: Groups
objectclass: top
objectclass: organizationalunit
```

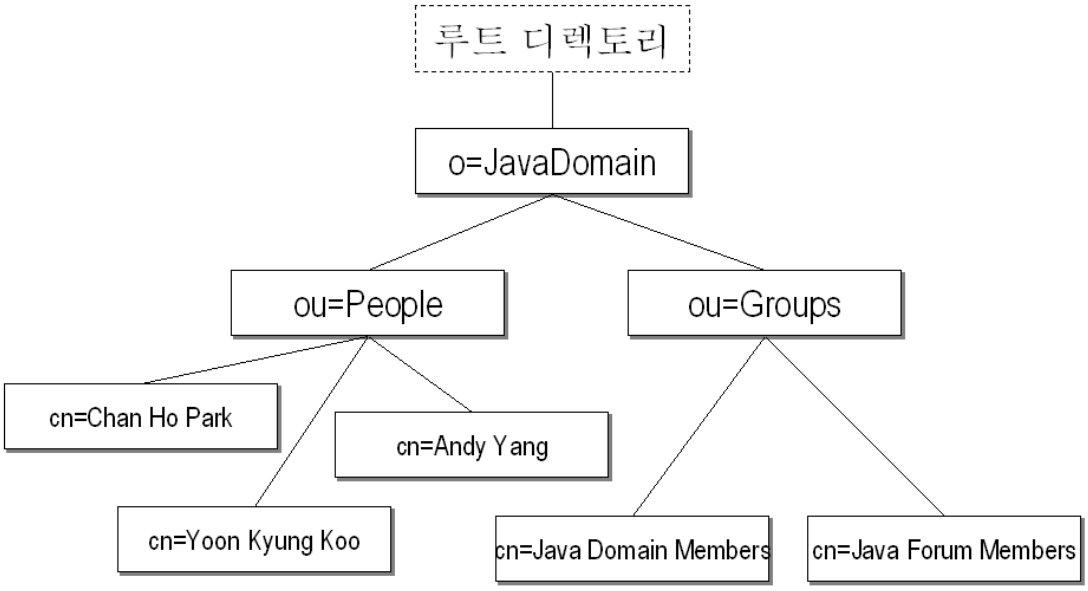
```
dn: cn=Chan Ho Park,ou=People,o=JavaDomain
objectclass: top
objectclass: person
objectclass: organizationalPerson
objectclass: inetOrgPerson
givenName: Chan Ho
sn: Park
cn: Chan Ho Park
mail: park61@yoonforh.com

dn: cn=Andy Yang,ou=People,o=JavaDomain
objectclass: top
objectclass: person
objectclass: organizationalPerson
objectclass: inetOrgPerson
givenName: Andy
sn: Yang
cn: Andy Yang
mail: andy@yoonforh.com

dn: cn=Yoon Kyung Koo,ou=People,o=JavaDomain
objectclass: top
objectclass: person
objectclass: organizationalPerson
objectclass: inetOrgPerson
givenName: Kyung Koo
sn: Yoon
cn: Yoon Kyung Koo
mail: yoonforh@yoonforh.com

dn: cn=Java Domain Members,ou=Groups,o=JavaDomain
objectclass: top
objectclass: groupOfNames
cn: Java Domain Members
member: cn=Yoon Kyung Koo
member: cn=Chan Ho Park
```

```
dn: cn=Java Forum Members,ou=Groups,o=JavaDomain
objectclass: top
objectclass: groupOfNames
cn: Java Forum Members
member: cn=Yoon Kyung Koo
member: cn=Andy Yang
```
<예제 2 끝 – testdata.ldif>

<용어 설명 시작 – DN, RDN, DIT>
LDAP 에서 각 엔트리들을 식별하고 구성하는 방법을 LDAP 네이밍 모델이라고 부른다.
LDAP 네이밍 모델에서 각 디렉토리 엔트리들은 전체가 트리 형태의 계층 구조로 구성되는데 이것을 디렉토리 정보 트리(Directory Information Tree, DIT)라고 부른다.
각 엔트리들은 자신의 식별 이름(Distinguished Name, DN)에 기반하여 DIT 내에 위치된다.
DN 은 하나의 엔트리를 분명하게 구분해주는 고유한 이름이다. DN 은 몇 개의 상대 식별 이름(Relative DN, RDN)으로 구성된다. DN 을 구성하는 각 RDN 은 DIT 의 루트 노드에서 해당 디렉토리 엔트리로 이르는 각 가지에 해당한다.
RDN 은 다음 형식을 가진다.

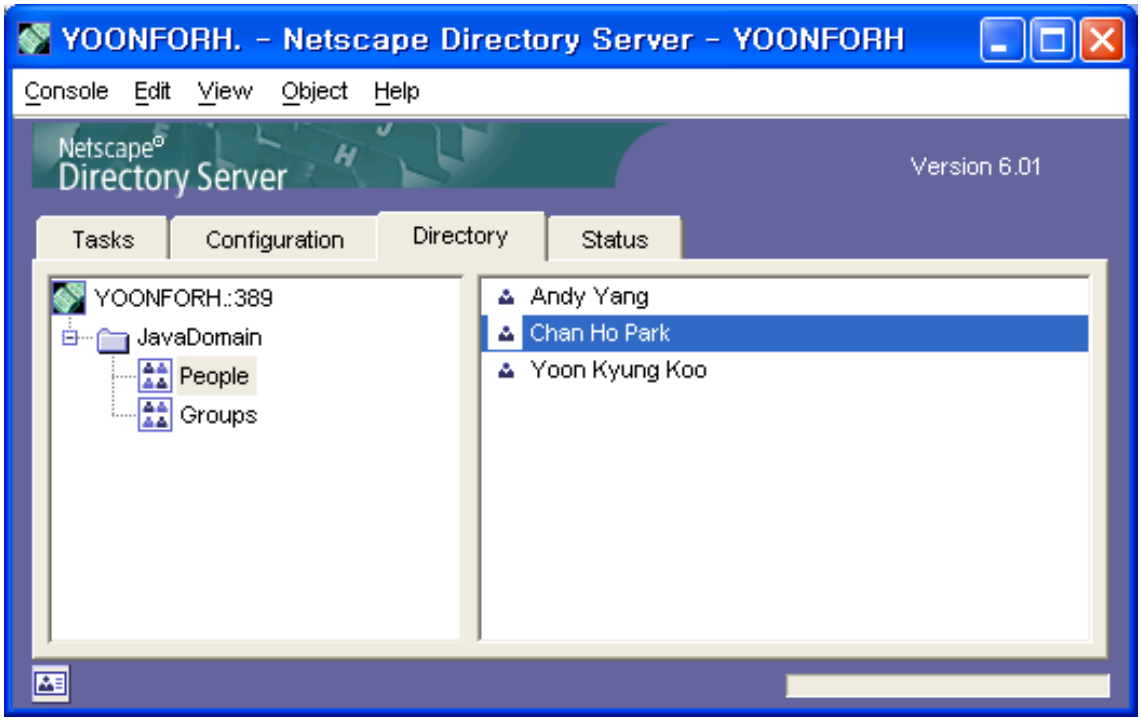
<속성 이름>=<속성 값>

예를 들어, cn=Yoon Kyung Koo,ou=People,o=JavaDomain 이라는 DN 은 쉼표로 구분되는 세 개의 RDN 으로 구성되며, 오른쪽에서 왼쪽으로 갈수록 DIT 트리의 가지 쪽을 향하게 된다.
<용어 설명 끝 – DN, RDN, DIT>



[그림 3 – 테스트 데이터의 디렉토리 정보 트리]

다음 그림은 넷스케이프 디렉토리 서버에 데이터 파일을 임포트한 후의 상태를 보여준다.



[그림 4 – 테스트 데이터를 임포트한 디렉토리 서버 실행 모습]

다음 예제는 JNDI 를 사용하여 디렉토리 서버에 접속한 다음, 등록된 엔트리와 각 속성을 출력하는 프로그램이다.

<예제 3 시작 – DirectoryEx.java>

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

/**
 * LDAP 서버를 사용한 디렉토리 예제
 */

public class DirectoryEx {
  public static void main(String[] args) {
    Hashtable env = new Hashtable();
    env.put(Context.INITIAL_CONTEXT_FACTORY,
        "com.sun.jndi.ldap.LdapCtxFactory");
    // 접속할 LDAP 서버의 URL을 지정한다.
    // 여기에서는 지역 호스트의 389번 포트에
```

```
    // 서버가 실행되고 있다고 가정한다.
    env.put(Context.PROVIDER_URL,
        "ldap://localhost:389");

    try {
      // 디렉토리 문맥을 생성한다.
      DirContext initCtx = new InitialDirContext(env);

      // 루트 노드를 찾는다.
      DirContext ctx = (DirContext) initCtx.lookup("o=JavaDomain");

      // 디렉토리 문맥의 전체 이름을 출력한다.
      System.out.println(ctx.getNameInNamespace());
      // 속성들을 출력한다.
      showAttributes("",
initCtx.getAttributes(ctx.getNameInNamespace()));

      // 디렉토리 문맥의 엔트리를 나열한다.
      showContext("  ", ctx);
    } catch (NamingException e) {
      e.printStackTrace();
    }
  }

  /**
   * 디렉토리 문맥의 내용을 출력
   */
  static void showContext(String indent, DirContext ctx) throws
NamingException {
    // 문맥의 모든 엔트리를 가져온다.
    NamingEnumeration enum = ctx.list("");
    while (enum.hasMore()) {
      NameClassPair entry = (NameClassPair) enum.next();
      System.out.println(indent + entry.getName());
      showAttributes(indent, ctx.getAttributes(entry.getName()));
```

```
      Object obj = ctx.lookup(entry.getName());
      // 디렉토리 문맥일 경우 재귀 호출
      if (obj instanceof DirContext) {
        showContext(indent + "  ", (DirContext) obj);
      }
    }
    enum.close();
  }

  /**
   * 속성 셋의 내용을 출력
   */
  static void showAttributes(String indent, Attributes attrs) throws
NamingException {
    // Attributes 클래스의 getAll() 메소드의
    // 결과인 NamingEnumeration에는
    // Attribute 객체들이 들어 있다.
    NamingEnumeration enum = attrs.getAll();
    while (enum.hasMore()) {
      Attribute attr = (Attribute) enum.next();
      System.out.println(indent + "  attribute : " + attr);
    }
    System.out.println();
    enum.close();
  }
}
```

<예제 3 끝 – DirectoryEx.java>

LDAP 서비스 제공자는 JDK 에 기본 포함되어 있으므로 컴파일하여 실행하면 다음과 같은 결과를 볼 수 있다.

```
javac DirectoryEx.java <엔터>

java -cp . DirectoryEx <엔터>

o=JavaDomain
```

```
  attribute : objectClass: top, organization
  attribute : o: JavaDomain

  ou=People
    attribute : ou: People
    attribute : objectClass: top, organizationalunit

    cn=Chan Ho Park
      attribute : mail: park61@yoonforh.com
      attribute : givenName: Chan Ho
      attribute  :  objectClass:  top,  person,  organizationalPerson,
inetOrgPerson
      attribute : sn: Park
      attribute : cn: Chan Ho Park
   … (생략)
```

27.1.2.3. LDAP 검색

LDAP 검색 기능을 사용할 때에는 검색 필터가 많이 사용된다.
검색 필터에는 검색 필터 연산자와 검색 필터들 간의 대수 연산자를 사용할 수 있다.

연산자	설명	사용 예
=	속성이 같은 엔트리를 찾는다.	cn=Yoon Kyung Koo
>=	알파벳 순서로 같거나 큰 속성 값을 가진 엔트리를 찾는다.	sn>=P 이 경우에는 Park, Yoon 등이 모두 해당된다.
<=	알파벳 순서로 작거나 같은 속성 값을 가진 엔트리를 찾는다.	sn<=Yoon 이 경우에는 Yang, Park 등이 해당된다.
=*	해당하는 속성의 값을 가진 엔트리를 모두 찾는다.	sn=* 이 경우에는 sn 에 값이 지정되면 모두 해당된다.
~=	속성의 값이 비슷한 엔트리를 모두 찾는다.	sn~=Parc 이 경우에는 Parc 와 발음이 비슷한 Park 도 해당된다.

<표 1 – 검색 필터 연산자>

대수 연산자	설명	사용 예
&	AND 연산. 각 검색 필터 항목들을 모두 만족하는 엔트리를 찾는다.	(& (sn=Yoon) (objectClass=person) (mail=*))
|	OR 연산. 각 검색 필터 항목들 중 하나라도 만족하는 엔트리를 찾는다.	(| (sn=Yoon) (& (objectClass=person) (mail>=yoon)))
!	NOT 연산. 해당 검색 필터 항목을 만족하지 않는 엔트리를 찾는다.	(! (sn=Yoon))

<표 2 – 검색 필터 대수 연산자>

SearchControls 클래스는 검색할 범위와 검색 결과의 개수, 시간 제한 등을 지정할 수 있다. 검색할 범위는 특별히 지정하지 않으면 해당 문맥만 찾으며, 하위 문맥까지 모두 찾거나 해당하는 엔트리만 비교하도록 지정할 수 있다.
다음 코드는 LDAP 서버에 접속하여 전체 범위에서 검색하는 예를 보여준다.

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
    "com.sun.jndi.ldap.LdapCtxFactory");
// 접속할 LDAP 서버의 URL을 지정한다.
env.put(Context.PROVIDER_URL,
    "ldap://localhost:389");

try {
  // 디렉토리 문맥을 생성한다.
  DirContext initCtx = new InitialDirContext(env);

  // 루트 노드를 찾는다.
  DirContext ctx = (DirContext) initCtx.lookup("o=JavaDomain");

  SearchControls sc = new SearchControls();
  // 하위 문맥들까지 모두 검색
  sc.setSearchScope(SearchControls.SUBTREE_SCOPE);
```

```
  // 검색 결과를 나타낸다.
  NamingEnumeration enum = initCtx.search(
    ctx.getNameInNamespace(),
    "(&(|(sn~=Parka) (cn>=J)) (!(objectclass=groupOfNames)))",
    sc);
    // search() 메소드의 결과 NamingEnumeration에는
    // SearchResult 객체가 들어 있다.
  while (enum.hasMore()) {
    SearchResult entry = (SearchResult) enum.next();
    System.out.println(entry.getName());
  }
  enum.close();
} catch (NamingException e) {
  e.printStackTrace();
}
```

검색 조건은 sn 속성이 Parka 와 비슷하거나 cn 속성이 J 보다 같거나 크고, objectclass 속성이 groupOfNames 가 아닌 엔트리를 모두 찾도록 지정되었다.
위 코드를 실행한 결과는 다음과 같다.

```
cn=Chan Ho Park,ou=People
cn=Yoon Kyung Koo,ou=People
```

27.1.3. 이벤트 처리

JNDI 는 엔트리가 추가, 삭제, 변경된다거나 혹은 서버로부터의 비동기 메시지를 AWT 이벤트와 같은 방식으로 처리할 수 있는 메커니즘을 제공한다.
LDAP 서버로부터 이벤트를 받으려면 LDAP 서버가 LDAP 버전 3 규격의 퍼시스턴트 검색 컨트롤(persistent search control)을 지원해야 한다. 넷스케이프 디렉토리 서버는 이 규격을 지원한다.
이벤트 처리에 관련된 객체들은 javax.naming.event 패키지에 정의되어 있다.
네이밍 문맥에서 이벤트를 처리하려면 네이밍 문맥을 EventContext 자료형으로 형 변환한 다음, addNamingListener() 메소드를 사용하여 네이밍 리스너를 등록하면 된다.
디렉토리 문맥에서 이벤트를 처리할 경우에는 디렉토리 문맥을 EventContext 의 자식 인터페이스인 EventDirContext 자료형으로 형 변환한 다음, addNamingListener() 메소드의 오버로드된 버전을 사용할 수 있다.
이벤트를 처리할 리스너 인터페이스는 NamingListener 인터페이스이며, 실제로는 이 인터페

이스의 자식 인터페이스인 ObjectChangeListener, NamespaceChangeListener, 혹은 javax.naming.ldap 패키지의 UnsolicitedNotificationListener 를 구현한다.

(1) NamingListener 인터페이스 : 네이밍 이벤트를 처리하는 리스너 인터페이스들의 부모 인터페이스. 특별한 이벤트를 처리하지는 않고, namingExceptionThrown() 메소드에서 이벤트 정보를 생성하는 도중에 발생하는 예외를 처리할 수 있도록 하고 있다. 네이밍 이벤트를 처리하려면 이 인터페이스의 자식 인터페이스를 구현하여야 한다.
(2) NamespaceChangeListener 인터페이스 : 이름 공간을 구성하는 엔트리들이 변경되는 이벤트를 처리하는 인터페이스. 엔트리가 추가, 삭제 혹은 이름이 변경될 때를 처리할 수 있다.
(3) ObjectChangeListener 인터페이스 : 엔트리 자신이 변경되는 이벤트를 처리하는 인터페이스. 엔트리의 속성이 변경된다거나 엔트리가 다른 것으로 대체될 때를 처리할 수 있다.
(4) UnsolicitedNotificationListener 인터페이스 : LDAP 서버가 클라이언트에게 비동기 메시지를 보내는 인터페이스.

JNDI 에서는 이벤트가 발생하면 등록된 네이밍 리스너 객체의 자료형을 검사하여 자료형에 해당하는 이벤트일 경우에 메소드를 호출하는 방식으로 이벤트를 처리한다. 따라서, 여러 종류의 이벤트를 하나의 네이밍 리스너에서 처리하도록 구현이 가능하다.

<따라하기 시작 – 이벤트 처리 예제>
여기에서는 LDAP 서버로부터 객체 변경, 이름 공간 변경 및 비동기 메시지를 모두 받을 수 있는 리스너 인터페이스를 등록하여 이벤트 처리를 한다.

1. 각 이벤트 리스너를 구현하도록 클래스를 선언하고 LDAP 서버에 연결하여 리스너를 등록한다.

DirContext 객체를 EventDirContext 자료형으로 형 변환한 다음 addNamingListener() 메소드를 사용하여 이벤트의 범위와 리스너를 등록할 수 있다.
이벤트를 발생시키기 위해 실제 엔트리를 변경하는 것은 별도의 쓰레드에서 실행될 Modifier 클래스에서 구현한다. 이벤트를 처리하기 위해 반드시 쓰레드를 분리할 필요는 없다.

```
import javax.naming.*;
import javax.naming.directory.*;
import javax.naming.event.*;
import javax.naming.ldap.UnsolicitedNotificationListener;
```

```
import javax.naming.ldap.UnsolicitedNotificationEvent;
import java.util.Hashtable;

/**
 * 네이밍 이벤트를 처리하는 예제
 */

public class NamingEventEx
     implements NamespaceChangeListener,
                ObjectChangeListener,
                UnsolicitedNotificationListener  {
  /**
   * 생성자
   */
  NamingEventEx() {
    Hashtable env = new Hashtable();
     env.put(Context.INITIAL_CONTEXT_FACTORY,
        "com.sun.jndi.ldap.LdapCtxFactory");
    // 접속할 LDAP 서버의 URL을 지정한다.
     env.put(Context.PROVIDER_URL,
        "ldap://localhost:389");

    try {
      // 디렉토리 문맥을 생성한다.
      DirContext initCtx = new InitialDirContext(env);

      // 루트 노드를 찾는다.
      DirContext ctx = (DirContext) initCtx.lookup("o=JavaDomain");

      // 모든 엔트리의 이벤트를 기다리도록 이벤트 리스너를 등록한다.
      ((EventDirContext) ctx).addNamingListener("",
                        EventContext.SUBTREE_SCOPE,
                        this);

      // 별도의 쓰레드에서 변경을 한다.
      new Thread(new Modifier()).start();
```

```
    // 30초 정도 기다려서 이벤트가 처리되기를 기다린다.
    try {
      Thread.sleep(30000L);
    } catch (InterruptedException e) {
      System.out.println("sleep interrupted");
    }
    ctx.close();
  } catch (NamingException e) {
    e.printStackTrace();
  }

}

public static void main(String[] args) {
  new NamingEventEx();
  }
}
```

2. 각 리스너 인터페이스의 메소드들을 구현한다. 여기에서는 간단히 이벤트 내용을 표준 출력으로 내보낸다.

```
/**
 * NamingListener 인터페이스 메소드
 */
public void namingExceptionThrown(NamingExceptionEvent evt) {
  evt.getException().printStackTrace();
}

/**
 * NamespaceChangeListener 인터페이스 메소드
 */
public void objectAdded(NamingEvent evt) {
  System.out.println("object added : " + evt.getNewBinding());
}
```

```
  public void objectRemoved(NamingEvent evt) {
    System.out.println("object removed : " + evt.getOldBinding());
  }

  public void objectRenamed(NamingEvent evt) {
    System.out.println("object renamed from " + evt.getOldBinding()
              + " to " + evt.getNewBinding());
  }

  /**
   * ObjectChangeListener 인터페이스 메소드
   */
  public void objectChanged(NamingEvent evt) {
    System.out.println("object changed from " + evt.getOldBinding()
              + " to " + evt.getNewBinding());
  }

  /**
   * UnsolicitedNotificationListener 인터페이스 메소드
   */
  public void notificationReceived(UnsolicitedNotificationEvent evt) {
    System.out.println("unsolicited event received : " + evt);
  }
```

3. 실제 엔트리를 변경할 Modifier 클래스를 작성한다.

LDAP 서버 설정에 따라 다를 수 있지만, 엔트리를 변경하기 위해서는 권한이 있는 사용자로 로그인할 필요가 있다.

Context 클래스에 선언되어 있는 상수들 중 SECURITY_AUTHENTICATION 은 인증 방법을 나타내며, SECURITY_PRINCIPAL 은 로그인할 사람, SECURITY_CREDENTIALS 는 사용자를 증명할 증거를 지정한다. 여기에서는 간단히 사용자 이름과 패스워드 방식으로 인증하는 방법을 사용하였다.

루트 노드를 찾은 후에는 이름 공간을 변경하거나, 엔트리의 속성을 변경하여 이벤트를 발생시킨다.

```
class Modifier implements Runnable {
  public void run() {
```

```
try {
  // 내용을 변경할 수 있는 권한 있는 사용자로 로그인한다.
  Hashtable env = new Hashtable();
  env.put(Context.INITIAL_CONTEXT_FACTORY,
      "com.sun.jndi.ldap.LdapCtxFactory");
  // 접속할 LDAP 서버의 URL을 지정한다.
  env.put(Context.PROVIDER_URL,
      "ldap://localhost:389");
  env.put(Context.SECURITY_AUTHENTICATION, "simple");  // 인증 방법
  env.put(Context.SECURITY_PRINCIPAL,
      "cn=yoonforh"); // 사용자
  env.put(Context.SECURITY_CREDENTIALS,
      "password"); // 패스워드
  // 디렉토리 문맥을 생성한다.
  DirContext initCtx = new InitialDirContext(env);

  // 루트 노드를 찾는다.
  DirContext ctx = (DirContext) initCtx.lookup("o=JavaDomain");

  // 엔트리를 추가한다.
  ctx.bind("cn=Test", "Event test!!!");

  // 엔트리 이름을 변경한다.
  ctx.rename("cn=Test", "cn=RenamedTest");

  // 엔트리를 삭제한다.
  ctx.unbind("cn=RenamedTest");

  Attributes attrs = new BasicAttributes();
  // uid 속성을 추가한다.
  attrs.put("uid", "yoonforh");
  // 속성을 추가한다.
  ctx.modifyAttributes("cn=Yoon Kyung Koo,ou=People",
           DirContext.ADD_ATTRIBUTE,
           attrs);
  // 속성을 삭제한다.
```

```
      ctx.modifyAttributes("cn=Yoon Kyung Koo,ou=People",
                DirContext.REMOVE_ATTRIBUTE,
                attrs);

      // 하위 문맥을 하나 생성한다.
      attrs = new BasicAttributes();
      attrs.put("objectclass", "top");
      attrs.put("objectclass", "person");
      attrs.put("objectclass", "organizationalPerson");
      attrs.put("objectclass", "inetOrgPerson");
      attrs.put("givenName", "Byung Hyun");
      attrs.put("sn", "Kim");
      attrs.put("mail", "bk@yoonforh.com");
      ctx.createSubcontext("cn=Byung Hyun Kim,ou=People", attrs);

      // 다시 지운다.
      ctx.destroySubcontext("cn=Byung Hyun Kim,ou=People");
      ctx.close();
    } catch (NamingException e) {
      e.printStackTrace();
    }
  }
```

4. 컴파일하여 실행해보자.

```
javac NamingEventEx.java <엔터>

java -cp . NamingEventEx <엔터>

object added : cn=Test: …(생략)
object renamed from cn=Test: null:null to cn=RenamedTest: null:null: …
(생략)
object removed : cn=RenamedTest: … (생략)
object changed from cn=Yoon Kyung Koo,ou=People: null:null to cn=Yoon
Kyung Koo,ou=People: … (생략)
object changed from cn=Yoon Kyung Koo,ou=People: null:null to cn=Yoon
```

```
Kyung Koo,ou=People: null:null: … (생략)
object added : cn=Byung Hyun Kim,ou=People: null:null: … (생략)
object removed : cn=Byung Hyun Kim,ou=People: null:null: … (생략)
```

<따라하기 끝 – 이벤트 처리 예제>

27.1.4. JNDI 의 URL

JNDI 에서는 URL 을 사용하여 서비스 제공자를 찾는 방법을 제공한다.
초기 문맥을 생성할 때 별도의 서비스 제공자를 지정하지 않고 URL 을 검색하여 서비스 제공자를 찾을 수 있다.
다음은 초기 문맥을 기본 생성자를 사용하여 단순히 생성한 다음 lookup() 메소드에서 URL 문자열을 지정하여 LDAP 서버의 지정한 엔트리를 찾는 것을 보여준다.

```
try {
  // 디렉토리 문맥을 생성한다.
  DirContext initCtx = new InitialDirContext();

  // 루트 노드를 찾는다.
  DirContext ctx = (DirContext)
initCtx.lookup("ldap://localhost:389/o=JavaDomain");
  System.out.println(ctx.getAttributes(""));
  ctx.close();
} catch (NamingException e) {
  e.printStackTrace();
}
```

JNDI 에서 사용되는 URL 은 URL 입출력 스트림 클래스들과는 상관이 없으며, 지정된 URL 의 규약을 보고, URL 문자열로부터 문맥을 생성하는 문맥 팩토리 클래스를 찾게 된다.
URL 문맥 팩토리 클래스는 <URL 규약>URLContextFactory 라는 이름을 가지며, Context 클래스의 URL_PKG_PREFIXES 라는 상수(실제로는 java.naming.factory.url.pkgs 라는 문자열)를 키로 하는 시스템 속성 값 뒤에 URL 규약을 붙인 값을 패키지로 가진다.
예를 들어, java.naming.factory.url.pkgs 시스템 속성이 com.yoonforh 로 지정되어 있고, URL 규약이 hello 이면 com.yoonforh.hello.helloURLContextFactory 라는 클래스를 찾게 된다.
패키지는 콜론(:)으로 구분하여 몇 개의 값을 지정할 수 있는데 기본값인 com.sun.jndi.url 패키지는 지정된 값들 뒤에 항상 추가된다.
예제의 경우 별도로 시스템 속성이나 해시 테이블로 패키지를 지정하지 않았으므로, 기본값

인 com.sun.jndi.url.ldap.ldapURLContextFactory 클래스에 의해 문맥이 생성된다.

27.2. JNDI 서비스 제공자

27.2.1. DNS 서비스 제공자

JDK 에는 LDAP, DNS, RMI 레지스트리, COS 네이밍을 사용하는 JNDI 서비스 제공자들이 포함되어 있다.

RMI 와 CORBA 에서 사용할 RMI 레지스트리와 COS 네이밍의 JNDI 서비스 제공자는 이 책의 뒷부분에서 RMI 와 CORBA 를 다룰 때 직접 사용하게 될 것이다.

DNS 서비스 제공자는 다음 예제와 같이 사용할 수 있다.

<예제 4 시작 – DNSEx.java>

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

/**
 * DNS JNDI 서비스 제공자의 사용 예제
 */

public class DNSEx {
  public static void main(String[] args) {
    try {
      Hashtable env = new Hashtable();
      env.put(Context.INITIAL_CONTEXT_FACTORY,
          "com.sun.jndi.dns.DnsContextFactory");
      // 접속할 도메인 네임 서버의 URL을 지정한다.
      env.put(Context.PROVIDER_URL,
          "dns://ns.sun.com");
      // 디렉토리 문맥을 생성한다.
      DirContext initCtx = new InitialDirContext(env);

      // 도메인 호스트 이름을 찾는다.
      DirContext ctx = (DirContext) initCtx.lookup(
        "java.sun.com");

      // 이 이름 공간에서 구별되는 문맥의 이름을 출력한다.
```

```
      System.out.println(ctx.getNameInNamespace());
      // 속성들을 출력한다.
      showAttributes(ctx.getAttributes(""));
      ctx.close();
    } catch (NamingException e) {
      e.printStackTrace();
    }
  }

  /**
   * 속성 셋의 내용을 출력
   */
  static void showAttributes(Attributes attrs) throws NamingException {
    NamingEnumeration enum = attrs.getAll();
    while (enum.hasMore()) {
      Attribute attr = (Attribute) enum.next();
      System.out.println("  attribute : " + attr);
    }
    System.out.println();
    enum.close();
  }

}
```

<예제 4 끝 – DNSEx.java>

예제는 도메인 네임 서버인 ns.sun.com 으로 접속하여 java.sun.com 에 대한 정보를 디렉토리 문맥의 속성으로 출력한다.
컴파일하여 실행하면 다음과 같은 결과를 볼 수 있다.

```
javac DNSEx.java <엔터>

java -cp . DNSEx <엔터>

java.sun.com
  attribute : A: 192.18.97.71
  attribute : MX: 10 mail.javasoft.com
```

27.2.2. JNDI 서비스 제공자 만들기

JNDI 서비스 제공자를 직접 만들 필요가 있을 경우가 있다.
자바 2 기업 환경과 같은 미들웨어에서는 서버의 각 컴포넌트나 데이터베이스 연결 등에 대한 접근을 통일된 방법으로 제공하기 위해 대부분의 경우 JNDI 를 활용하고 있다.
이런 경우와 같이 프로그램 내에서 특별하게 네이밍이나 디렉토리 개념을 지원하는 표준 인터페이스를 위해 JNDI 를 사용하고자 할 때 사용자가 프로그램에 필요한 JNDI 서비스 제공자를 설계할 필요가 생긴다.
JNDI 서비스 제공자를 만들 때 기본적으로 구현해야 할 것들은 다음과 같다.

(1) Context 인터페이스를 구현한 클래스 : 실제 문맥을 나타내는 클래스를 구현해야 한다.
(2) InitialContextFactory 인터페이스를 구현한 클래스 : 초기 문맥을 생성해주는 클래스를 구현해야 한다.
(3) NameParser 인터페이스를 구현한 클래스 : 이름 공간 안의 엔트리 이름들을 해석하는 이름 파서를 구현해야 한다.

이외에 객체를 자체 형식의 데이터로 저장하거나, 저장된 데이터를 다시 객체로 읽어들이는 팩토리 클래스들과 URL 지원 등의 기능들은 필요에 따라 차례로 추가할 수 있다.
서비스 제공자들의 기능을 구현할 때 사용되는 몇 가지 팩토리 인터페이스들과 표준 기능들을 구현한 유틸리티 클래스들이 javax.naming.spi 패키지에 선언되어 있다.

<따라하기 시작 – JNDI 서비스 제공자 만들기>
여기에서 만들 JNDI 서비스 제공자는 메모리의 해시 테이블에 이름과 엔트리의 바인딩 정보를 담고 있다.
구현을 간단하게 하기 위해 문맥의 계층 구조를 지원하지 않는 단일 문맥 구조로 하고, 다른 이름 공간과의 연결을 지원하지 않는다. 즉, CompositeName 도 복합 구성을 지원하지 않고, CompoundName 의 이름도 한 요소만을 가지는 아주 간단한 문맥이다.
이 경우, 이름 파서는 입력된 이름을 계층 구분자 없이 하나의 문맥에 해당하는 이름으로 간주하여 Name 객체를 생성하면 된다.
메모리에 바인딩 정보를 저장할 때, 여러 곳에서 초기 문맥을 생성하여 접근하더라도 동일한 문맥 내용을 가지도록 해시 테이블을 static 변수로 선언하였다. 동일한 프로그램 내에서는 항상 같은 엔트리들을 갖고 있으므로 한번만 데이터베이스 연결 객체의 풀이나 전역 환경 변수 정보 등을 이 문맥에 저장해두면, JNDI 의 표준화된 인터페이스를 사용하여 다른 클래스에서도 필요할 때 쉽게 접근할 수 있어 유용하게 활용할 수 있을 것이다.

먼저 문맥부터 구현해보자.

1. Context 인터페이스를 구현하는 SimpleContext 클래스를 선언하고, 멤버 필드를 정의한다.

필드로는 문맥의 바인딩 정보를 담을 해시 테이블과 환경 변수를 저장할 해시 테이블, 그리고 이 이름 공간의 이름 파서 등이 있다.

문맥 계층 구조를 지원한다면 부모 문맥 필드에 대한 정보도 필드로 가지고 있어야 할 것이다.

cloneContext() 메소드는 문맥을 복사하여 새로운 문맥 객체를 생성하며, getMyNamePart() 메소드는 CompositeName 객체를 분석하여 문맥의 이름 공간에서 처리할 CompoundName 객체만을 가져오는 유틸리티 메소드이다. Context 인터페이스의 여러 메소드들을 구현할 때 사용할 것이다.

```
package yoonforh.jndi.spi;

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

/**
 * 해시 테이블을 사용한 네이밍 문맥.
 * 복합 이름이나 하위 문맥을 지원하지 않는다.
 */

public class SimpleContext implements Context {
  /**
   * 이름 공간의 엔트리들.
   * 일반적으로는 문맥별로 바인딩 정보를 가지고 있어야 하겠지만
   * SimpleContext의 경우 같은 자바 프로그램 내에서
   * 동일한 정보를 공유할 수 있도록 static으로 선언하였다.
   */
  static protected Hashtable bindings = new Hashtable();

  /** 문맥 환경 변수들 */
  protected Hashtable env;
  /** 이름 파서 */
```

```
  protected final static NameParser nparser = new SimpleNameParser();

  /**
   * 생성자.
   */
  SimpleContext(Hashtable env) {
    if (env != null) {
      // 항상 환경 변수 테이블을 복사해서 사용한다.
      this.env = (Hashtable) env.clone();
    }
  }

  /**
   * 문맥 복사
   */
  protected Context cloneContext() {
    return new SimpleContext(env);
  }

  /**
   * 이름 공간에서 처리할 수 있는 이름 부분을 반환한다.
   */
  protected Name getMyNamePart(Name name) throws NamingException {
    if (name instanceof CompositeName) {
      // 복합 이름 공간을 지원하지 않는다.
      if (name.size() > 1) {
        throw new InvalidNameException("this provider doesn't support
federation.");
      }

      return nparser.parse(name.get(0));
    } else { // 단일 이름 공간의 CompoundName
      return name;
    }
  }
}
```

2. lookup() 메소드를 구현한다.

lookup() 메소드를 구현할 때 주의할 점은 인자로 넘어온 이름이 “”이거나 빈 Name 객체이면 문맥 자신을 뜻하므로, 현재 문맥과 같은 정보를 가지도록 문맥을 복사하여 새로운 문맥 객체를 반환한다.

인자로 넘어온 이름이 비어있지 않으면 바인딩 해시 테이블에서 이름을 검색한다.

이때, 해시 테이블에 저장된 객체를 그냥 사용할 수도 있지만, JNDI 에서는 자바 객체를 JNDI 서비스 제공자가 저장하거나, 저장된 값을 자바 객체로 복원할 때에 NamingManager 클래스의 유틸리티 메소드를 사용하여 변환을 하도록 권장하고 있다.

따라서, 해시테이블에 저장할 때에는 getStateToBind(), 해시테이블에서 객체를 가져올 때에는 getObjectInstance() 메소드를 사용하는 것이 서비스 제공자의 확장을 고려할 때 좋은 방법이다.

String 을 인자로 받는 오버로드 버전은 String 인자를 Name 객체로 바꾸어서 원래 메소드를 호출하는 방식으로 구현한다.

```
/**
 * 엔트리를 찾는다.
 */
public Object lookup(Name name) throws NamingException {
  if (name.isEmpty()) {
    // 문맥 자신을 찾는 경우이므로
    // 동일한 내용을 가지도록 문맥을 복사한다.
    return cloneContext();
  }

  // 이름 공간에 속하는 이름 부분을 가져온다.
  Name nm = getMyNamePart(name);
  String atom = nm.get(0);

  if (nm.size() == 1) {
    // 해시 테이블에 저장된 객체를 반환한다.
    // 저장된 객체를 반환할 때에는 저장된 객체를 바로 반환하지 않고
    // NamingManager의 getObjectInstance() 메소드를 호출하여
    // 저장 상태를 자바 객체로 변환하는 것이 JNDI의 표준 방법이다.
    try {
      return NamingManager.getObjectInstance(bindings.get(atom), // 저
```

장된 객체

```
                        nm, // 이름
                        this, // 문맥
                        env // 환경
                        );
      } catch (Exception e) {
        e.printStackTrace();
      }
    } else {
      // 하위 문맥을 지원하지 않는다.
      throw new InvalidNameException("this provider does not support
context hierarchy");
    }

    throw new NameNotFoundException("cannot find " + atom);
  }

  /**
   * 오버로드 버전
   */
  public Object lookup(String name) throws NamingException {
    return lookup(new CompositeName(name));
  }
```

3. 마찬가지 방법으로 바인딩에 관련된 bind(), rebind(), unbind() 메소드를 구현한다.
bind() 메소드의 경우에는 주어진 이름에 해당하는 엔트리가 이미 존재할 경우 예외를 던지고, rebind() 메소드의 경우에는 무조건 덮어쓴다는 점이 다르다.
unbind() 메소드의 경우에는 주어진 이름에 해당하는 엔트리가 존재하지 않을 경우에도 에러를 내지 않고 조용히 메소드를 반환해야 한다는 점에 주의한다. 하지만, 이름이 잘못 주어진 경우에는 예외를 던져야 한다.

```
  /**
   * 새 엔트리를 추가한다.
   */
  public void bind(Name name, Object obj) throws NamingException {
    // 빈 이름인지 검사
```

```
    if (name.isEmpty()) {
      throw new InvalidNameException("Cannot bind empty name");
    }

    // 이름 공간에 속하는 이름 부분을 가져온다.
    Name nm = getMyNamePart(name);
    String atom = nm.get(0);

    if (nm.size() == 1) {
      // 해시 테이블에 이름을 가진 엔트리가 있을 경우 에러
      if (bindings.containsKey(atom)) {
        throw new NameAlreadyBoundException();
      }

      // 해시 테이블에 객체를 저장한다.
      // 객체를 바로 저장하기 않고 NamingManager 클래스의 getStateToBind()
      // 메소드를 사용하여 자바 객체를 저장 상태로 변환한 것을 저장하는 것이
      // JNDI의 표준 방법이다.
      bindings.put(atom,
            NamingManager.getStateToBind(obj, // 저장할 객체
                        nm, // 이름
                        this, // 문맥
                        env // 환경
                        ));
    } else { // 하위 문맥을 지원하지 않는다.
      throw new InvalidNameException("this provider does not support
context hierarchy");
    }
  }

  /**
   * 오버로드 버전
   */
  public void bind(String name, Object obj) throws NamingException {
    bind(new CompositeName(name), obj);
  }
```

```
  /**
   * 엔트리가 존재할 경우에도 바인드한다.
   * 엔트리가 미리 존재하는지 검사를 하지 않는다는 점을 제외하면 bind()와 동일하다.
   */
  public void rebind(Name name, Object obj) throws NamingException {
    // 빈 이름인지 검사
    if (name.isEmpty()) {
      throw new InvalidNameException("Cannot rebind empty name");
    }

    // 이름 공간에 속하는 이름 부분을 가져온다.
    Name nm = getMyNamePart(name);
    String atom = nm.get(0);

    if (nm.size() == 1) {
      // 해시 테이블에 객체를 저장한다.
      // 객체를 바로 저장하기 않고 NamingManager 클래스의 getStateToBind()
      // 메소드를 사용하여 자바 객체를 저장 상태로 변환한 것을 저장하는 것이
      // JNDI의 표준 방법이다.
      bindings.put(atom,
            NamingManager.getStateToBind(obj, // 저장할 객체
                        nm, // 이름
                        this, // 문맥
                        env // 환경
                        ));
    } else { // 하위 문맥을 지원하지 않는다.
      throw new InvalidNameException("this provider does not support
context hierarchy");
    }
  }

  /**
   * 오버로드 버전
   */
  public void rebind(String name, Object obj) throws NamingException {
```

```
    rebind(new CompositeName(name), obj);
  }

  /**
   * 엔트리를 삭제한다.
   * 해당하는 이름을 가진 엔트리가 없을 경우에도 에러를 내지 않는다.
   */
  public void unbind(Name name) throws NamingException {
    // 빈 이름인지 검사
    if (name.isEmpty()) {
      throw new InvalidNameException("Cannot unbind empty name");
    }

    // 이름 공간에 속하는 이름 부분을 가져온다.
    Name nm = getMyNamePart(name);
    String atom = nm.get(0);

    if (nm.size() == 1) {
      // unbind는 멱등 연산으로 설계되었으므로,
      // 존재하는지 여부를 검사하지 않고 조용히 삭제한다.
      bindings.remove(atom);
    } else { // 하위 문맥을 지원하지 않는다.
      throw new InvalidNameException("this provider does not support
context hierarchy");
    }
  }

  /**
   * 오버로드 버전
   */
  public void unbind(String name) throws NamingException {
    unbind(new CompositeName(name));
  }
```

4. 마찬가지 방법으로 rename() 메소드를 구현한다.

```
  /**
   * 엔트리의 이름을 변경한다.
   */
  public void rename(Name oldName, Name newName) throws NamingException
{
    // 빈 이름인지 검사
    if (oldName.isEmpty() || newName.isEmpty()) {
      throw new InvalidNameException("Cannot rename empty name");
    }

    // 이름 공간에 속한 이름 부분을 추출한다.
    Name oldnm = getMyNamePart(oldName);
    Name newnm = getMyNamePart(newName);

    if ((oldnm.size() != 1) || (newnm.size() != 1)) {
      // 하위 문맥을 지원하지 않는다.
      throw new InvalidNameException("this provider does not support
context hierarchy");
    }

    String oldatom = oldnm.get(0);
    String newatom = newnm.get(0);

    // 새 이름이 이미 바인드된 경우 에러
    if (bindings.get(newatom) != null) {
      throw new NameAlreadyBoundException(newatom + " is already
bound");
    }

    // 이전 바인딩을 삭제
    Object entry = bindings.remove(oldatom);

    // 이전 이름이 바인드되어 있지 않으면 에러
    if (entry == null) {
      throw new NameNotFoundException("cannot find " + oldatom);
    }
```

```
    // 새 바인딩을 추가
    bindings.put(newatom, entry);
  }

  /**
   * 오버로드 버전
   */
  public void rename(String oldName, String newName) throws
NamingException {
    rename(new CompositeName(oldName), new CompositeName(newName));
  }
```

5. list() 메소드를 구현한다.

list() 메소드는 주어진 이름에 해당하는 문맥의 내용을 NameClassPair 객체로 채워진 NamingEnumeration 을 반환해야 한다.

이를 위해 NamingEnumeration 인터페이스를 구현한 NameList 클래스를 내부 클래스로 선언하였다.

주어진 이름이 문맥이 아닌 경우에는 예외를 던져야 한다.

```
  /**
   * 문맥의 내용을 NameClassPair 클래스 객체로 나열
   */
  public NamingEnumeration list(Name name) throws NamingException {
    if (name.isEmpty()) {
      // 빈 이름일 경우 문맥 전체 내용을 나열
      return new NameList(bindings.keys());
    }

    // name이 문맥을 가리키는지 확인
    Object target = lookup(name);
    if (target instanceof Context) {
      return ((Context) target).list("");
    }

    throw new NotContextException("cannot list " + name);
```

```
}

/**
 * 오버로드 버전
 */
public NamingEnumeration list(String name) throws NamingException {
  return list(new CompositeName(name));
}

/**
 * Enumeration 내용을 NameClassPair 객체로 바꾸어 넣는 내부 클래스
 */
class NameList implements NamingEnumeration {
  protected Enumeration names;

  /**
   * 생성자
   */
  NameList(Enumeration names) {
    this.names = names;
  }

  public boolean hasMoreElements() {
    try {
      return hasMore();
    } catch (NamingException e) {
      return false;
    }
  }

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

  /**
   * Enumeration의 내용을 NameClassPair 객체로 만든다.
```

```
     */
    public Object next() throws NamingException {
      String name = (String) names.nextElement();
      String className = bindings.get(name).getClass().getName();
      return new NameClassPair(name, className);
    }

    public Object nextElement() {
      try {
        return next();
      } catch (NamingException e) {
        NoSuchElementException ne = new NoSuchElementException();
        ne.initCause(e);
        throw ne;
      }
    }

    public void close() {}
  }
```

6. listBindings() 메소드를 구현한다.

listBindings() 메소드는 list() 메소드와 비슷하나 해당하는 문맥의 내용을 NameClassPair 객체 대신에 Binding 객체로 채워진 NamingEnumeration 을 반환한다는 점에서 다르다. 이를 위해 NameList 클래스를 상속하는 BindingList 클래스를 내부 클래스로 선언하였다. Binding 객체를 생성할 때에는 해시테이블에 저장된 데이터를 자바 객체로 바꾸기 위하여 NamingManager 클래스의 getObjectInstance() 메소드를 사용하였다.

```
  /**
   * 문맥의 내용을 Binding 클래스 객체로 나열
   */
  public NamingEnumeration listBindings(Name name) throws
NamingException {
    if (name.isEmpty()) {
      // 빈 이름일 경우 이 문맥 전체 내용을 나열
      return new BindingList(bindings.keys());
    }
```

```
    // name이 문맥을 가리키는지 확인
    Object target = lookup(name);
    if (target instanceof Context) {
      return ((Context) target).listBindings("");
    }

    throw new NotContextException("cannot list " + name);
  }

  /**
   * 오버로드 버전
   */
  public NamingEnumeration listBindings(String name) throws
NamingException {
    return listBindings(new CompositeName(name));
  }

  /**
   * Enumeration 내용을 Binding 객체로 바꾸어 넣는 내부 클래스
   */
  class BindingList extends NameList {
    /**
     * 생성자
     */
    BindingList(Enumeration names) {
      super(names);
    }

    /**
     * Enumeration의 내용을 Binding 객체로 만든다.
     */
    public Object next() throws NamingException {
      String name = (String) names.nextElement();
      Object obj = bindings.get(name);
```

```
      try {
        obj = NamingManager.getObjectInstance(obj,
                            new CompositeName().add(name),
                            SimpleContext.this,
                            SimpleContext.this.env);
      } catch (Exception e) {
        NamingException ne = new NamingException();
        ne.setRootCause(e);
        throw ne;
      }

      return new Binding(name, obj);
    }
  }
```

7. 하위 문맥을 생성하거나 삭제하는 메소드를 구현한다. 여기에서는 문맥의 계층 구조를 지원하지 않으므로, 간단하게 OperationNotSupportedException 예외를 던진다.

```
  /**
   * 하위 문맥 삭제. 여기서는 지원하지 않는다.
   */
  public void destroySubcontext(Name name) throws NamingException {
    throw new OperationNotSupportedException("this provider does not
support subcontexts");
  }

  /**
   * 오버로드 버전
   */
  public void destroySubcontext(String name) throws NamingException {
    destroySubcontext(new CompositeName(name));
  }

  /**
   * 하위 문맥 생성해서 바인드한다.
   * 여기서는 지원하지 않는다.
```

```
   */
  public Context createSubcontext(Name name) throws NamingException {
    throw new OperationNotSupportedException("this provider does not
support subcontexts");
  }

  /**
   * 오버로드 버전
   */
  public Context createSubcontext(String name) throws NamingException {
    return createSubcontext(new CompositeName(name));
  }
```

8. lookupLink() 메소드는 여기에서는 링크 개념을 지원하지 않으므로 그냥 lookup() 메소드를 호출하도록 구현한다.

```
  /**
   * 링크를 찾는다. 여기에서는 링크 개념을 지원하지 않으므로, 그냥 lookup()을 호출
한다.
   */
  public Object lookupLink(Name name) throws NamingException {
    return lookup(name);
  }

  /**
   * 오버로드 버전
   */
  public Object lookupLink(String name) throws NamingException {
    return lookupLink(new CompositeName(name));
  }
```

9. 이름 파서와 이름에 관련된 몇 가지 메소드를 구현한다.

```
/**
   * 해당하는 이름의 이름 파서 객체를 반환한다. 여기에서는 항상 같다.
   */
```

```
  public NameParser getNameParser(Name name) throws NamingException {
    return nparser;
  }

  /**
   * 오버로드 버전
   */
  public NameParser getNameParser(String name) throws NamingException {
    return getNameParser(new CompositeName(name));
  }

  /**
   * prefix 이름에 대한 상대 이름으로 name을 간주하여 이름을 구성한다.
   */
  public Name composeName(Name name, Name prefix) throws
NamingException {
    Name result;

    // 복합 이름이 아니면, 상대 이름으로 간주하여
    // Name 객체를 생성한다.
    if (!(name instanceof CompositeName)
      && !(prefix instanceof CompositeName)) {
      result = (Name) (prefix.clone());
      result.addAll(name);
      return result;
    }

    // 복합 이름 공간을 지원하지 않는다.
    throw new OperationNotSupportedException("this provider doesn't
support federation.");
  }

  /**
   * 오버로드 버전
   */
  public String composeName(String name, String prefix)
```

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

/**
 * 이름 공간에서 문맥의 전체 이름을 반환한다.
 * 계층 구조를 지원하지 않으므로 항상 이 문맥은 초기 문맥과 동일하며 이름은 ""이다.
 */
public String getNameInNamespace() throws NamingException {
  return "";
}
```

10. 기타 환경 변수 관련된 메소드와 close() 메소드를 구현한다.

```
/**
 * 환경 변수를 추가한다.
 */
public Object addToEnvironment(String propName, Object propVal)
     throws NamingException {
  // 환경 변수 해시 테이블에 지정된 속성을 추가한다.
  if (env == null) {
    env = new Hashtable();
  }

  return env.put(propName, propVal);
}

/**
 * 환경 변수를 삭제한다.
 */
public Object removeFromEnvironment(String propName)
  throws NamingException {
  // 환경 변수 해시 테이블에서 속성을 삭제한다.
  if (env == null) {
```

```
      return null;
    }

    return env.remove(propName);
  }

  /**
   * 환경 변수 내용을 반환한다.
   */
  public Hashtable getEnvironment() throws NamingException {
    if (env == null) {
      // 널이 아닌 해시 테이블을 반환해야 한다.
      return new Hashtable();
    } else {
      return (Hashtable) env.clone();
    }
  }

  /**
   * 문맥에 관련된 리소스를 해지한다.
   */
  public void close() throws NamingException {
    // 해지할 리소스가 없다.
  }
```

11. 문맥 클래스의 구현이 끝났으므로 이름 파서를 구현한다.

NameParser 인터페이스는 parse() 메소드 하나를 선언하고 있으므로, 이 메소드만 구현하면 된다.

일반적으로 문자열을 CompoundName 객체로 파싱하여 구성하기 위해서는 문맥 계층을 구분하는 구분 문자, 계층을 나타내는 좌우 방향, 대소문자 구분 등의 정보가 필요하다. CompoundName 클래스는 이러한 정보를 Properties 객체로 생성자에서 넘겨받는다.

여기에서는 계층 구조를 지원하지 않으므로, 단순하게 기본값만으로 CompoundName 을 생성하게 하였다. CompoundName 에 지정할 수 있는 여러 속성들에 대해서는 API 문서를 참조하자.

```
package yoonforh.jndi.spi;
```

```
import javax.naming.*;

/**
 * SimpleContext의 이름 파서
 */

public class SimpleNameParser implements NameParser {
  /**
   * 이름을 단일한 이름 공간의 단일한 계층 이름으로 간주하여 Name 객체를 생성한다.
   */
  public Name parse(String name) throws NamingException {
    // 빈 Properties 객체를 반환하는 것은 모두 기본값을 사용한다는 뜻.
    // Properties에 들어갈 수 있는 키에 대해서는 CompoundName 클래스
    // 문서를 참조.
    return new CompoundName(name, new java.util.Properties());
  }
}
```

12. 마지막으로 초기 문맥 팩토리 클래스를 구현한다.
초기 문맥을 구현할 때 환경 변수를 참조할 수 있지만, 여기에서는 환경 변수와 관련이 없으므로, 단순히 문맥을 생성하는 역할만 한다.

```
package yoonforh.jndi.spi;

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

/**
 * SimpleContext의 초기 문맥을 생성한다.
 */

public class SimpleContextFactory implements InitialContextFactory {
  /**
```

```
   * 초기 문맥을 생성한다.
   */
  public Context getInitialContext(Hashtable environment)
       throws NamingException {
    return new SimpleContext(environment);
  }

}
```

13. 문맥, 이름 파서, 초기 문맥 팩토리 클래스들의 구현이 끝났으므로 컴파일한다.

```
javac yoonforh\jndi\spi\*.java <엔터>
```

14. 성공적으로 컴파일이 되었으면 테스트 프로그램을 작성하여 의도대로 동작하는지 보도록 하자.

다음은 간단한 테스트 프로그램이다.

초기 문맥을 생성할 때 초기 문맥 팩토리 클래스 이름을 여기에서 작성한 SimpleContextFactory 로 지정한 다음 JNDI 의 일반적인 사용법에 따라 테스트하면 된다.

초기 문맥을 두 번 생성하여 테스트하는 이유는 SimpleContext 의 경우 동일한 클래스 로더에서는 모두 같은 바인딩 정보를 가지도록 설계하였으므로, 이것을 확인하기 위한 것이다.

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

/**
 * Simple JNDI 서비스 제공자 사용 예제
 */

public class SimpleSPITest {
  public static void main(String[] args) {
    try {
      Hashtable env = new Hashtable();
      env.put(Context.INITIAL_CONTEXT_FACTORY,
          "yoonforh.jndi.spi.SimpleContextFactory");
```

```
    // 초기 문맥을 생성한다.
    Context initCtx = new InitialContext(env);

    // 루트 노드를 찾는다.
    Context ctx = (Context) initCtx.lookup("");

    // 엔트리를 추가한다.
    ctx.bind("hello", "Hello, world!!!");
    System.out.println("hello : " + ctx.lookup("hello"));

    // 엔트리 이름을 변경한다.
    ctx.rename("hello", "hi");
    System.out.println("hi : " + ctx.lookup("hi"));

    // 엔트리에 File 객체를 추가한다.
    ctx.bind("file", new java.io.File("test.txt"));
    System.out.println("file : " + ctx.lookup("file"));

    // 문맥의 엔트리를 나열한다.
    showEntries(ctx);
    ctx.close();

    // 다른 초기 문맥을 생성한다.
    Context initCtx2 = new InitialContext(env);
    // 새로 생성한 문맥의 엔트리를 나열한다.
    // SimpleContext는 엔트리들을 모두 공유하므로 위와 동일하다.
    showEntries(initCtx2);
  } catch (NamingException e) {
    e.printStackTrace();
  }

}

 /**
  * 문맥에 바인드된 엔트리들을 보여준다.
  */
```

```
  static void showEntries(Context ctx) throws NamingException {
    NamingEnumeration enum = ctx.list("");
    System.out.println("current entry list ---");
    while (enum.hasMore()) {
      NameClassPair entry = (NameClassPair) enum.next();
      System.out.println("  " + entry);
    }
    enum.close();
  }
}
```

15. 테스트 프로그램을 컴파일하여 실행해보자.
실행할 때 서비스 제공자 클래스 파일들도 함께 클래스 경로에 포함되어야 한다.

```
javac SimpleSPITest.java <엔터>

java -cp . SimpleSPITest <엔터>

hello : Hello, world!!!
hi : Hello, world!!!
file : test.txt
current entry list ---
  hi: java.lang.String
  file: java.io.File
current entry list ---
  hi: java.lang.String
  file: java.io.File
```

문맥 엔트리가 동일하게 출력됨을 볼 수 있다.

<따라하기 끝 – JNDI 서비스 제공자 만들기>

맺음말

JNDI 는 LDAP 과 같은 디렉토리 서버를 사용하기에 편리한 인터페이스일 뿐만 아니라, 다양한 네이밍과 디렉토리에 적용할 수 있으며, 또 필요한 서비스 제공자를 만들어 사용할 수 있는 강력하고 확장성 높은 프레임웍이다.

특히, 서버 환경에서는 동일한 인터페이스를 사용하여 리소스에 접근할 수 있기 때문에 광범위하게 활용되고 있다.
예를 들어, 데이터베이스 연결 리소스를 사용할 때, javax.sql 패키지의 PooledConnection 인터페이스를 구현한 클래스 객체를 JNDI 를 사용하여 바인딩할 수 있다. 프로그램이 시작되는 초기에 바인딩을 하고 나면, 프로그램의 다른 부분에서 쉽게 JNDI 인터페이스를 사용하여 이 PooledConnection 객체에 접근할 수 있고, 현재 사용 가능한 데이터베이스 연결을 쉽게 이용할 수 있다. 필요할 때마다 직접 데이터베이스 연결을 하는 것에 비해 훨씬 효율적이고, 이해하기 쉬우며, 또 관리하기도 쉽다.
리소스 뿐만 아니라 전역 환경 설정 정보들 같은 경우에도 JNDI 서비스 제공자에 저장되어 있으면 손쉽게 정보를 가져올 수 있을 것이다.