LDAP PROGRAMMING WITH JAVA™

ROB WELTMAN and TONY DAHBURA

ADDISON–WESLEY

An Imprint of Addison Wesley Longman, Inc.
Reading, Massachusetts • Harlow, England • Menlo Park, California
Berkeley, California Don Mills, Ontario Sydney
Bonn Amsterdam Tokyo Mexico City

CHAPTER 5

Searching with the SDK

As you may recall from Chapter 1, the major feature of an LDAP directory is its ability to return search results on queries rapidly. The SDK provides many flexible methods for obtaining and handling search results from the directory. We will focus in this chapter on building queries using the SDK to retrieve the information we need.

The result set from a search can easily be parsed to return the entry's name and all or a subset of its attributes and values. In our discussion of searches, we will take an example-based approach. Most of the examples here can be run directly from the command-line interface with the `java` command. It is assumed that you have installed or have access to a directory server and have loaded the sample database from the LDIF file that is supplied on the CD-ROM that accompanies this book.

Our First Search

Before you can search an LDAP directory, you need certain information:

- **Host name** of the machine where the directory is installed
- **Port** number of the directory server
- **Base DN** of the directory tree managed by the server
- **Scope** of the search
- Search **filter**
- **Attributes** to request
- Optionally, **search preferences**

Host Name

The host name directs the search to the machine where the directory resides. This parameter is mandatory and is usually of the form `machinename.domain`—for example, `dirhost.acme.com`. If you are at the console on the machine that is running the LDAP server, you can use the host name "localhost" for your test server. You can specify the IP address of the host instead if you wish—for example, `127.0.0.1` for "localhost."

Port

The port is the TCP port of the machine (indicated by the host name) where the directory server is listening for LDAP connections. The standard port for LDAP is port 389 for non-SSL connections. You can use the constant `LDAPConnection.DEFAULT_PORT` for port 389. For SSL-based connections the default port is 636. This is not to say that you cannot have an LDAP server listening on any port you desire, but if you wish to communicate and make your services available to the widest audience, stick to the standard port numbers.

Base DN

The base distinguished name (DN) indicates where in the LDAP directory you wish to begin the search. An LDAP directory is arranged in tree fashion, with a root and various branches off this root. Figure 5-1 depicts a typical architecture. The base DN is used to indicate at which node the search should originate. For example, we could indicate a base of `o=airius.com` for a search that starts at the top and proceeds downward. If instead we specified a base DN of `ou=customers, o=airius.com`, then any entries above this tree level would not be eligible for searching. It is important to specify the base DN correctly to ensure that you receive the anticipated results.

Scope

Scope is the starting point of a search and the depth from the base DN to which the search should occur. There are three options (values) for the scope:

1. **`BASE`**, represented by the constant `LDAPConnection.SCOPE_BASE`, is used to indicate searching only the entry at the base DN, resulting in only that entry being returned (if it also meets the search filter criteria). Figure 5-2 depicts the scope of a base-level search.

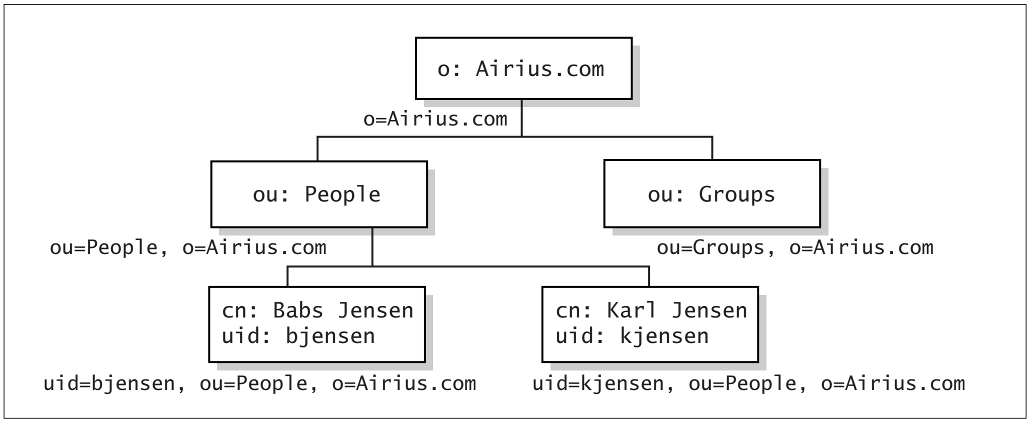


FIGURE 5-1. *Typical directory architecture.*

2. **`ONE`**, represented by the constant `LDAPConnection.SCOPE_ONE`, is used to indicate searching all entries one level under the base DN—but **not** including the base DN. Figure 5-3 depicts the scope of a one level search.

3. **`SUBTREE`**, represented by the constant `LDAPConnection.SCOPE_SUB`, is used to indicate searching of all entries at all levels under and *including* the specified base DN. Figure 5-4 depicts the scope of a subtree search.

The base DN and scope parameters can dramatically affect the number of records returned from a query. It is important to understand what is involved in using these arguments.

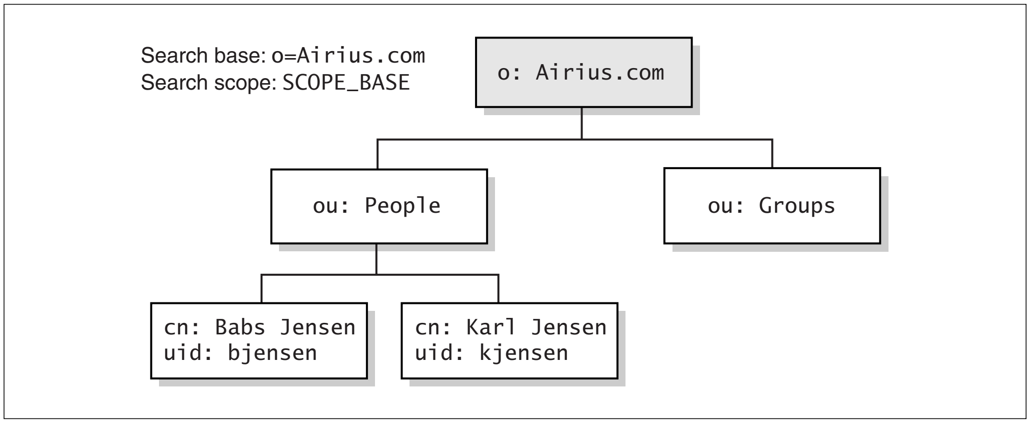


FIGURE 5-2. *Scope* BASE *search.*

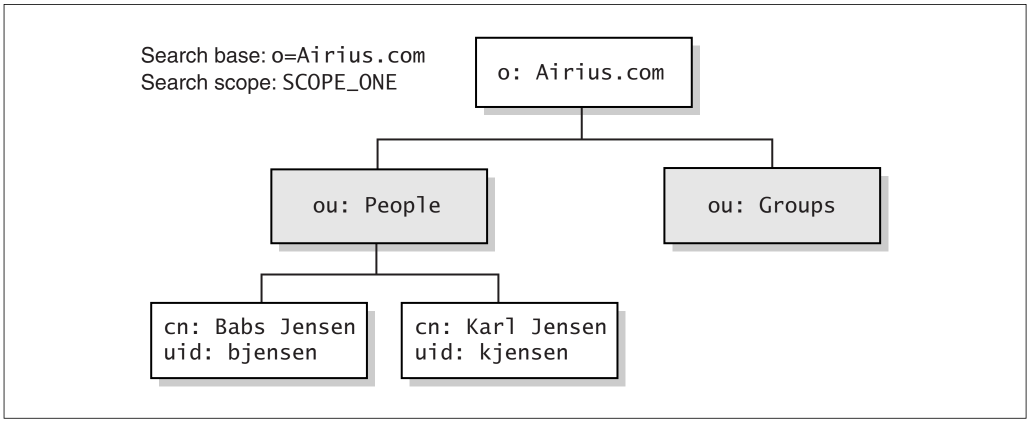


FIGURE 5-3. *Scope* ONE *search.*

Filter

The search filter is the query string. It is used to filter the entries in the directory and produce the desired set of matching records. Filters are built using parentheses and combinations of the symbols &, |, and !, which represent AND, OR, and NOT, respectively. If you wanted to locate all people with "tony" at the beginning of their names, the following filter would do the trick:

```
(&(objectclass=person)(cn=tony*))
```

This expression represents a search for all entries with an object class of type `person` in which the common name begins with "tony." Like most other LDAP attri-

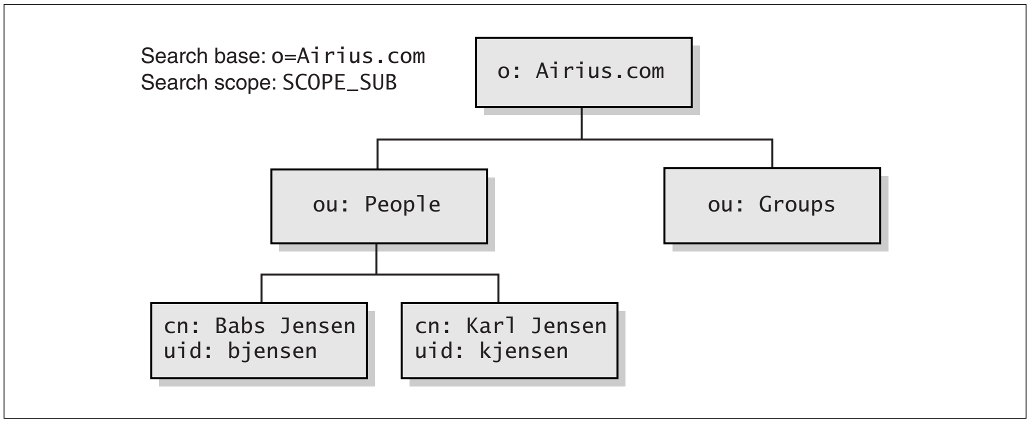


FIGURE 5-4. *Scope* SUBTREE *search.*

butes, the cn attribute has case-insensitive syntax, so replacing tony* with Tony* or TONY* would yield the same results.

Search filters can be nested to any level:

```
(&(objectclass=person)(|(cn=sam carter) (cn=tony*)))
```

This filter says to find all entries with object class person in which the common name is Sam Carter or begins with "tony." Complex filters can be built using the operators and corresponding parentheses. A logical operator should appear before the parenthesis enclosing the group of compares it affects. You can specify the order for operators by nesting parentheses.

Table 5-1 lists all the operators for a search filter. These options can be combined using parentheses, as shown in the examples already given. Wild cards can also be used for filters—for example, (cn=tony*).

Attributes

Among the attributes of an LDAP entry for a person are cn, sn, and givenName. In the LDIF record for Babs Jensen that follows, the attribute names are marked in bold.

TABLE 5-1. *Search filter operators.*

OPERATOR	MEANING
|	OR
&	AND
!	NOT
=	Entry attribute equals value (e.g., cn=John Doe)
>=	Entry attribute is greater than or equal to value (e.g., cn>=John Doe, which would find Tom Doe among other entries)
<=	Entry attribute is less than or equal to value
=*	All entries that have a value for the attribute (e.g., cn=* for all entries with a cn value)
~=	Entries that approximately match the value—a soundex match for values that "sound like" the value (e.g., cn~=olson to match Olson, Olsson, and Oleson)

Each entry can contain numerous attributes—the specific ones determined by the object classes of the entry. Some attributes are optional for a particular object class, and some are required, as discussed in Chapter 2.

```
dn: uid=bjensen, ou=People, o=airius.com
cn: Babs Jensen
sn: Jensen
givenName: Barbara
objectclass: top
objectclass: person
l: Cupertino
uid: bjensen
mail: bjensen@airius.com
telephoneNumber: +1 408 555 1862
roomNumber: 0209
userPassword: hifalutin
```

The search attributes in a search request represent the values to return for records matching the filter, starting at the base DN and progressing through the scope level desired. You should request only attributes that you need. Requesting all attributes for a large result set can significantly increase processing time and memory usage. Note that specifying `null` for the attributes field of the `search` or `read` methods of `LDAPConnection` means to return *all* the attributes associated with each entry. If you wish to retrieve no attributes for an entry, use the constant `LDAPConnection.NO_ATTRS` for the attributes parameter.

LDAP attributes are either user attributes or operational attributes. **User attributes** appear in the directory only if they have been explicitly added to it, by the addition or modification of entries. **Operational attributes** are created by the server itself. Examples of operational attributes are `createTimeStamp` (the time when the entry was created) and `numSubordinates` (the number of direct children of the node). If you specify `null` for attributes in a search, operational attributes are not returned. Each operational attribute to be returned must be specified explicitly in the list of attributes. If you wish to receive operational attributes in addition to all user attributes, use the constant `LDAPConnection.ALL_USER_ATTRS` as one of the attributes—for example:

```
String[] attrs1 = { LDAPConnection.ALL_USER_ATTRS, "createTimeStamp",
                    "numSubordinates"};
String[] attrs2 = { "cn", "objectclass", "createTimeStamp",
                    "numSubordinates"};
```

The `String` array `attrs1` indicates to return all user attributes for this entry, as well as the two operational attributes `createTimeStamp` and `numSubordinates`. The

`String` array `attrs2` is used to return the two user attributes `cn` and `objectclass`, as well as the two operational attributes `createTimeStamp` and `numSubordinates`.

If you want to do client-side sorting of your result sets, you must include the attributes by which you are sorting as attributes to be returned by the server. If you are doing server-side sorting (which we will cover in Chapter 16), including these attributes is not necessary. Note that attribute names are always case-insensitive, so `{ "objectclass" }` is treated the same as "`{ "ObjectClass" }`" when specifying attributes to return. Also, you should not count on the server using the same case for names of attributes it returns.

Search Preferences

You may set certain preferences for a search. These preferences include the amount of time you wish to allow the server to spend on your search, the maximum number of records you will accept, and whether the search should wait (block) until all data is received or should return records as they are available. Search preferences are specified using the `LDAPSearchConstraints` class. Commonly used methods of `LDAPSearchConstraints` include the following:

- **`setBatchSize`** specifies how results are returned during a search. A value of zero indicates to wait until all results are in before returning them; a value of one means to return each result as it becomes available. The second option is useful if you want to populate a list and not make the user wait until everything is back before showing some data. On the other hand, if no data is to be processed until all results have arrived, it is more efficient to specify zero.
- **`setHopLimit`** specifies how many times a returned referral should be followed in finding a real entry. A referral is returned when a server does not contain the data being requested; instead it returns to the caller information on where the data resides. It is said to "refer" the caller to another source for the information.
- **`setMaxResults`** specifies the maximum number of results that should be returned from a search. For no limit on the number of results (unlimited returns), use a value of zero. Note that if this number is higher than the maximum number the server has been configured to return, you will get only the server's maximum, and an exception will be thrown indicating that the server size limit was exceeded.
- **`setReferrals`** specifies whether or not the SDK should follow referrals automatically.

- **`setServerTimeLimit`** specifies the maximum number of seconds for the server to spend on delivering search results.

All options for searching and other LDAP operations are covered in more detail in Chapter 14.

The following examples demonstrate setting these options:

```
// Get the preferences associated with this connection
LDAPSearchConstraints cons = ld.getSearchConstraints();
cons.setBatchSize( 1 );
cons.setHopLimit ( 5 );
cons.setMaxResults( 0 );
cons.setReferrals( true );
cons.setServerTimeLimit( 5 );
```

Our First Search Program

The following program is a command line-based Java search program. It assumes that the `airius.com` sample database was loaded as described in Chapter 4. The program provides a framework to try out different search filters. Let's look at the code for `FilterSearch.java`:

```
import netscape.ldap.*;
import java.util.*;

/**
 * Simple search program to experiment with filters
 */
public class FilterSearch {

    /**
     * Do a subtree search using a specified filter
     *
     * @parm args host, port, authDN, password, baseDN, filter
     */
    public static void main( String[] args ) {

        if ( args.length != 6 ) {
            System.out.println( "Usage: java FilterSearch " +
                                "<host> <port> " +
                                "<authdn> <password> " +
                                "<baseDN> <filter>" );
            System.out.println( "Example:" );
            System.out.println( "    java FilterSearch " +
                                "localhost 389 " +
                                "\"\" \"\" " +
                                "\"o=airius.com\" " +
                                "\"(|(cn=sam*)(cn=b*))\"" );
                System.exit(1);
            }
```

The code declares some needed values, including the host name of the machine and the port on which the LDAP server is listening.

The next section of code sets up our search constraints. The only value we set is to block on one result at a time. This setting will cause our program to get one value and allow us to display it while the next value is being retrieved from the server.

```
            String host = args[0];
            int port = Integer.parseInt( args[1] );
            String authid = args[2];
            String authpw = args[3];
            String base = args[4];
            String filter = args[5];
            String[] ATTRS = {"cn","mail","telephoneNumber"};

            int status = -1;
            LDAPConnection ld = new LDAPConnection();
            try {
                // Connect to server and authenticate
                ld.connect( host, port, authid, authpw );
```

The `getSearchConstraints` method returns a copy of the preferences in the connection. In addition, this program allows connecting with a user DN and password. An LDAPv3 server will assume an anonymous authentication if no user DN and password are specified. If communicating with an LDAPv2 server, you must authenticate, even if binding anonymously (for an anonymous bind, use empty strings for the user DN and password). We will cover authentication in detail in Chapter 6.

The program takes as input a search filter and does a `search` with a scope of `LDAPConnection.SCOPE_SUB`. Recall that `SCOPE_SUB` indicates searching all entries at all levels under and including the specified base DN. We specify the base DN as the top of our tree (`o=airius.com`).

```
                System.out.println( "Search filter=" + filter );
                LDAPSearchResults res = ld.search( base,
                                                  ld.SCOPE_SUB,
                                                  filter,
                                          ATTRS,
                                          false );

                // Loop on results until complete
                while ( res.hasMoreElements() ) {
                    try {
                        // Next directory entry
                        LDAPEntry entry = res.next();
```

Once the search request is issued, we retrieve each eligible record and send it to `prettyPrint` for display on the console, as the following code shows. For now, we will ignore any referrals returned by the server. Referrals are discussed in detail in Chapter 16. If any errors result in an exception (`netscape.ldap.LDAPException`), we just print the error and continue to process any remaining results. All classes in the SDK have a `toString` method, which provides useful information about the state of each object.

```
                        prettyPrint( entry, ATTRS );
                        status = 0;
                    } catch ( LDAPReferralException e ) {
                        // Ignore referrals
                        continue;
                    } catch ( LDAPException e ) {
                        System.out.println( e.toString() );
                        continue;
                    }
                }

            } catch( LDAPException e ) {
              System.out.println( e.toString() );
            }
```

The following block disconnects us from the LDAP server.

```
            // Done, so disconnect
            if ( (ld != null) && ld.isConnected() ) {
                try {
                    ld.disconnect();
                } catch ( LDAPException e ) {
                    System.out.println( e.toString() );
                }
            }

            System.exit( status );
        }
```

The method prettyPrint takes a returned entry and the array of the attributes we requested and pulls the values from the search result:

```
    /**
      * Print names and values of attributes in an entry
      *
      * @param entry entry containing attributes
      * @param attrs array of attribute names to display
      */
     public static void prettyPrint( LDAPEntry entry,
                                     String[] attrs ) {
        System.out.println( "DN: " + entry.getDN() );

         // Use array to pick attributes. We could have
         // enumerated them all using LDAPEntry.getAttributes
         // but this gives us control of the display order.
         for ( int i = 0; i < attrs.length; i++ ) {
             LDAPAttribute attr =
                 entry.getAttribute( attrs[i] );
             if ( attr == null ) {
                 System.out.println( attrs[i] +
                                     " not present" );
                 continue;
             }
             Enumeration enumVals = attr.getStringValues();
             // Enumerate on values for this attribute
             boolean hasVals = false;
             while ( (enumVals != null) &&
                     enumVals.hasMoreElements() ) {
                 String val = (String)enumVals.nextElement();
                 System.out.println( attrs[i] + ": " + val );
                 hasVals = true;
             }
             if ( !hasVals ) {
                 System.out.println( attrs[i] +
                                    " has no values" );
             }
         }
         System.out.println("----------");
     }
 }
```

The value or values for each attribute are obtained with getStringValues, which returns an Enumeration. Most LDAP attributes are strings, but some are binary. Examples of binary attributes are userCertificate;binary and jpegPhoto.

It is up to you, the programmer, to specify if you want the values delivered to you as strings or as binaries. The alternative interface is `getBinaryValues`. There is no way to query the directory to determine whether it is appropriate to call `getStringValues` or `getBinaryValues` on an attribute. As a programmer you must have some understanding of the data type represented in a particular attribute. For standard LDAP attributes the data type is typically a known format that is the same in all applications.

Data from the directory is always returned in binary or UTF8 format, not in any other character set (such as latin-1 or shift-jis). Strings are represented internally in UTF8 format, a form of Unicode, which allows representation of all the world's languages. When you call `getBinaryValues`, the SDK gives you the values exactly as they are stored in the directory. If you call `getStringValues`, the SDK attempts to convert the values into Java `String` objects, which are in UCS2 (another Unicode format), before returning them. If the data cannot be converted, which might be the case with the value of a `jpegPhoto` attribute, for example, then `getStringValues` returns `null`.

Using Search Filters

You should type in the code or load it from the CD-ROM and compile, using the following command:

```
javac FilterSearch.java
```

Let's execute some searches and see what different filters return:

```
java FilterSearch localhost 389 "" "" "o=airius.com" "(cn=sam carter)"
java FilterSearch localhost 389 "" "" "o=airius.com" "(cn=Sam Carter)"
```

These two commands will return the same single record, demonstrating that case does not matter in a search for the common name (which is defined in the LDAP server as a case-insensitive attribute):

```
dn: uid=scarter, ou=People, o=airius.com
cn: Sam Carter
mail: scarter@airius.com
telephoneNumber: +1 408 555 4798
```

Now let's try a more complex search filter. The following request will return a series of results. You will get all members whose names begin with "sam," "tony," or the letter *J*.

```
java FilterSearch localhost 389 "" "" "o=airius.com"
    "(|(cn=sam*)(cn=tony*)(cn=j*))"
```

The next search uses the telephone number field.

```
java FilterSearch localhost 389 "" "" "o=airius.com"
    "(telephoneNumber=650-9*)"
java FilterSearch localhost 389 "" "" "o=airius.com"
    "(telephoneNumber=6509*)"
```

One complication that arises with telephone numbers in many contexts is that some people store them as (XXX) XXX-XXXX, others use the syntax XXX-XXX-XXXX, and some may just store the digits (XXXXXXXXXX). It would be cumbersome if anyone doing a search had to know how each person had entered the telephone number. LDAP defines a special telephone number syntax for the `telephoneNumber` attribute. To enter a phone number to be searched, you can use any of the three formats described here, and the LDAP server strips the expression down to just the numbers before performing the comparison. The LDAP standard document RFC 2252 specifies many different syntaxes for attributes beyond the one most commonly used: case-insensitive string.

The following search is interesting:

```
java FilterSearch localhost 389 "" "" "o=airius.com" "(cn~=brian)"
```

This search uses the "sounds like" operator (see Table 5-1). The following results will be displayed:

```
Search filter=(cn~=brian)
DN: uid=bplante, ou=People, o=airius.com
cn: Brian Plante
mail: bplante@airius.com
telephoneNumber: +1 408 555 3550
----------
DN: uid=jbrown, ou=People, o=airius.com
cn: Judy Brown
mail: jbrown@airius.com
telephoneNumber: +1 408 555 6885
----------
```

The results indicate that "Brown" sounds close to "Brian" according to the syntax rules of the server, and of course Brian was found as well.

Any attribute that is in the directory and is not protected with access control from searches by an anonymous user is eligible to be searched against, as shown here:

```
java FilterSearch localhost 389 "" "" "o=airius.com" "(&(
    (objectclass=person) (cn=t*) (|(telephoneNumber=>650*) (mail=*)))"
```

This search expression indicates that we want all records in which (1) the entry includes the object class `person`, (2) the first name begins with the letter *T*, and (3) the area code of the telephone number is greater than or equal to 650 or the entry has a `mail` attribute. Note the syntax for mail: =*. This syntax indicates that we want every entry that contains a value for this attribute. The asterisk is a presence indicator when used in isolation on the right-hand side of a filter expression.

Handling Results

A significant aspect of working with LDAP searches is processing the results after issuing the query. The results from the search are returned as an `LDAPSearchResults` object, which implements `Enumeration`. Note that once you iterate over the result set, it is not available anymore. If you must do multiple passes over the result set, then you must save the values in a store that is internal to your program.

There are two methods for iteration: `nextElement` and `next`. The `nextElement` method returns `Object`, which could be `LDAPEntry`, `LDAPReferralException`, or `LDAPException`. You are responsible for detecting the type of result (using `instanceof`) and taking appropriate action. The `next` method returns `LDAPEntry` and may throw an `LDAPReferralException` or an `LDAPException`. We recommend using `next` in most cases.

When the `next` method of `LDAPSearchResults` is called, there are three possible consequences. The first is that an entry is returned as an `LDAPEntry` object. The second possibility is that you will be passed a referral (search reference) exception. This might happen if there is a referral configured in the directory tree you are searching and you have not set up the SDK to follow referrals automatically. The third possibility is that you will receive an `LDAPException`, which might happen if, for example, the entry specified as the base DN does not exist.

If referrals are followed automatically and if the referral hop limit has not been exceeded, the LDAP Java classes follow the referral and retrieve the entry for you and you will never get a referral exception, even when the classes are creating a new connection to the referred-to server in order to retrieve the entry for you. The default setting in the SDK is to *not* follow referrals automatically, so you might encounter one if you used the FilterSearch code above. You can indicate that you want automatic referral handling with the following code:

```
ld = new LDAPConnection();
ld.getSearchConstraints().setReferrals( true );
```

Referrals are discussed in detail in Chapter 16.

The `next` method of `LDAPSearchResults` returns an `LDAPEntry` object. The `LDAPEntry` class contains the following four methods:

1. **`getDN`** returns the full distinguished name of the entry as a `String` (for example, `uid=scarter, ou=People, o=airius.com`).
2. **`getAttribute( String name )`** takes a `String` argument of the attribute name that we are interested in retrieving and returns an object that represents this attribute. The return type is `LDAPAttribute`. An optional argument that identifies a language subtype can also be specified. Language subtypes (part of the LDAP RFC 2596) can be used to store different values for a single attribute in an environment where clients specify the language in which they want to view directory contents. Such attributes include a semicolon and the language subtype when they are added to the directory. For example:

```
givenName;lang-en: John
givenName;lang-fr: Jean
givenName;lang-sp: Juan
```

If all three values were present in an entry, you could retrieve the third one with the following code:

```
LDAPEntry.getAttribute( "givenName", "lang-sp" );
```

If the specified attribute does not exist in the entry, `null` is returned.

3. **`getAttributeSet`** returns an `LDAPAttributeSet` object that represents all the attributes in this entry. You can then call the `getAttributes` method of the `LDAPAttributeSet` to obtain an `Enumeration` on all the attributes in the entry:

```
LDAPAttributeSet attrs = theEntry.getAttributeSet();
Enumeration enum_attrs = attrs.getAttributes();  //allows iterating
                                                 //each one
```

4. **`toString`** returns the entire entry, including all the attributes retrieved, as a `String`. This method is useful for debugging and is called by the compiler when a conversion to `String` is implied. For example,

```
    System.out.println( "This is what was returned: " + theEntry );
```

Once we have the attributes that are present in the entry, we can obtain the values for these attributes. The `LDAPAttribute` class has many methods for dealing with the attribute and its values. The methods most commonly used are the following:

- **`getStringValues`** returns an `Enumeration` of the values for a particular attribute as `Strings`. Remember that in LDAP, many attributes may have more than one value.
- **`getName`** returns the name of the attribute (for example, `mail` or `cn`).

If we examine the `prettyPrint` method, we can see the calls needed to extract the attribute values from an entry:

```
    public static void prettyPrint( LDAPEntry entry,
                                    String[] attrs ) {
```

The following line displays the DN of an entry, which we get by using the `getDN` method of `LDAPEntry`.

```
        System.out.println("DN: "+theEntry.getDN() );
```

Knowing the DN of an entry is very important to an application developer, because it provides a method to obtain the entry uniquely if we should need to retrieve it again.

To specify the attributes to be extracted from the entry, the next block of code uses the `attrs` array, which contains `{"cn","telephoneNumber","mail"}`.

```
        // Use array to pick attributes. We could have
        // enumerated them all using LDAPEntry.getAttributes,
        // but this gives us control of the display order.
        for ( int i = 0; i < attrs.length; i++ ) {
            LDAPAttribute attr =
                entry.getAttribute( attrs[i] );
            if ( attr == null ) {
                System.out.println( attrs[i] +
                                    " not present" );
                continue;
            }
```

Note that we check if any attribute is `null` for an entry. The value `null` indicates that the attribute is not present in this entry. Any attribute that is not mandatory for an entry may be omitted and will then not be returned during a search. Another popular programming method for handling attributes is just to enumerate over the values for the entry. We will examine a version of `prettyPrint` a bit later that will use this method.

After reaching this point, we know the entry has the attribute, but we do not know if the attribute has a value. Formally, LDAP does not allow attributes with no values, but it does allow attributes with a `null` value. Many attributes in an LDAP directory can have multiple values. For example, the `telephoneNumber` attribute could contain one or more telephone numbers. The following block of code gets the attribute values and handles multivalue situations by calling `LDAPAttribute.getStringValues`, which returns an `Enumeration` of the values:

```
                Enumeration enumVals = attr.getStringValues();
                // Enumerate on values for this attribute
                boolean hasVals = false;
                while ( (enumVals != null) &&
                        enumVals.hasMoreElements() ) {
                    String val = (String)enumVals.nextElement();
                    System.out.println( attrs[i] + ": " + val );
                    hasVals = true;
                }
                if ( !hasVals ) {
                    System.out.println( attrs[i] +
                                      " has no values" );
                }
```

Finally, we mark the end of the output for this record:

```
            }
            System.out.println("----------");
        }
    }
```

Attributes in Detail

One of the three following conditions will be true for an attribute in an entry: the attribute is present but has no value (actually a `null` value), the attribute is not present, or the attribute is present and has one or more values:

1. *Attribute present in entry with no value.* The following abbreviated LDIF record shows that this entry has the `telephoneNumber` attribute, but that the attribute has no value.

```
dn: uid=andy1, ou=People, o=airius.com
ou: People
cn: Andy Jones
...
telephoneNumber:
```

This is a valid condition, and when `prettyPrint` executes, it does not print the attribute in the listing. Code that detects this condition sets the boolean flag `hasVals` inside the enumeration loop:

```
while ( enumVals.hasMoreElements() ) {
    ...
    hasVals = true;
```

The output from `prettyPrint` is as follows:

```
Search filter=(uid=andy*)
DN: uid=andy1, ou=People, o=airius.com
cn: Andy James
mail: andy1@airius.com
telephoneNumber HAS NO VALUES
```

2. *Attribute not present.* As indicated by the following LDIF record, if the attribute is not mandatory for any of the object classes of the entry, then it may or may not be present.

```
dn: uid=andy2, ou=People, o=airius.com
ou: People
objectclass: top
objectclass: person
objectclass: organizationalPerson
objectclass: inetOrgPerson
cn: Andy James
```

The corresponding code in `prettyPrint` that handles this situation is as follows:

```
if (attr == null) {
    System.out.println(ATTRS[i] + " NOT PRESENT");
    continue;
}
```

Keep in mind that just because you requested a particular attribute does not mean that every entry will contain the attribute. If we had used the enumerated set returned by `getAttributes`, the `telephoneNumber` attribute would not have been in the enumeration. If using the enumeration, your program would need to track which attributes were returned for each entry. Otherwise, use the `getAttribute` method to retrieve a specific attribute; if `null` is returned, then the attribute is not present for this particular entry.

The output from `prettyPrint` is as follows:

```
DN: uid=andy2, ou=People, o=airius.com
cn: Andy Jones
mail: andy2@airius.com
telephoneNumber NOT PRESENT
```

3. *Attribute present and has one or more values*. This condition is the most common for most searches. The following LDIF record has multiple values for the `telephoneNumber` attribute. Often programmers are interested in only one value, but they must be prepared for the occurrence of multiple values.

```
dn: uid=andy3, ou=People, o=airius.com
ou: People
...
cn: Andy Stevens
telephoneNumber: 650-555-1212
telephoneNumber: 650-555-1213
```

The output from `prettyPrint` is as follows:

```
DN: uid=andy3, ou=People, o=airius.com
cn: Andy Stevens
mail: andy3@airius.com
telephoneNumber: 650-555-1212
telephoneNumber: 650-555-1213
```

I Want Only One Record and I Have the DN

The DN uniquely identifies a single entry in the directory. In some situations you have the DN (for example, because you saved it or because information is provided to allow you to build it), and you want to retrieve the single record corresponding to the DN. The `read` method of the `LDAPConnection` class provides this functionality. Although not as flexible as issuing a full search, this method provides the benefit of retrieving the single uniquely identified record with few parameters required. Within the SDK the method is just a search with the scope set to SCOPE_BASE and the filter to objectclass=*.

The following code is a modification of FilterSearch that takes a DN as argument instead of a search base and filter.

```
public class EntryRead {

    /**
     * Read an entry from the directory and display the contents
     *
     * @param args host, port, authDN, password, dn
     */
    public static void main( String[] args ) {

        if ( args.length != 5 ) {
            System.out.println( "Usage: java EntryRead " +
                                "<host> <port> " +
                                "<authdn> <password> " +
                                "<dn>" );
            System.out.println( "Example:" );
            System.out.println( "    java EntryRead " +
                            "localhost 389 " +
                            "\"\" \"\" " +
                            "\"uid=scarter, ou=People, " +
                                "o=airius.com\"" );
            System.exit(1);
        }

        String host = args[0];
        int port = Integer.parseInt( args[1] );
        String authid = args[2];
        String authpw = args[3];
        String base = args[4];
        String[] ATTRS = { "cn","mail","telephoneNumber" };

        int status = -1;
        LDAPConnection ld = new LDAPConnection();
        try {
            // Connect to server and authenticate
            ld.connect( host, port, authid, authpw );

            LDAPEntry entry = ld.read( base, ATTRS );
            prettyPrint( entry, ATTRS );
```

```
                status = 0;
            } catch ( LDAPReferralException e ) {
                // Ignore referrals
            } catch ( LDAPException e ) {
                System.out.println( e.toString() );
            }

            // Done, so disconnect
            if ( (ld != null) && ld.isConnected() ) {
             try {
                    ld.disconnect();
                } catch ( LDAPException e ) {
                    System.out.println( e.toString() );
                }
            }
            System.exit( status );
        }
    }
```

The method `prettyPrint` here is the same as in `FilterSearch`.

Executing the program with the following command:

```
java EntryRead localhost 389 "" "" "uid=scarter, ou=People, o=airius.com"
```

will result in the following output:

```
DN: uid=scarter, ou=People, o=airius.com
cn: Sam Carter
mail: scarter@airius.com
telephoneNumber: +1 408 555 4798
```

Searching and Comparing

The LDAP SDK allows you to compare a value in memory to the value of an attribute of an entry without actually retrieving the entry. This is called a **`compare`** operation. In many ways a `compare` can be simulated using a `search` by setting the scope of the `search` to `SCOPE_BASE` and providing a search filter with which the value can be compared; if an entry is returned, the `compare` operation was successful. A `compare`, however, may improve performance because the return data from the LDAP server is a small packet that says either that the value is the same or that it is different.

Let's examine a small piece of code to compare a specific record and see if the `l` attribute (the LDAP attribute for location) has the value `Santa Clara`. The following code is abbreviated to show simply how to call the `compare` method of `LDAPConnection`.

```
String ENTRYDN = "uid=scarter, ou=People, o=Airius.com";
LDAPAttribute attr = new LDAPAttribute( "l","Santa Clara" );
try {
    LDAPConnection ld = new LDAPConnection();
    ld.connect(HOST, PORT); //connect to server
    boolean ok = ld.compare( ENTRYDN, attr );
    if (ok) {
        System.out.println("Values matched!");
    } else {
        System.out.println("No Match!");
    }
} catch (LDAPException e) {
    System.out.println( "Error: " + e.toString() );
}
```

The `compare` method takes a DN, an `LDAPAttribute` object representing the attribute you wish to compare and the value to compare, and an optional `LDAPSearchConstraints` object. The method returns `true` if the entry has the attribute and specified value and `false` if the entry does not have the value or the attribute. An exception is thrown if the entry does not exist.

More on Filters

In the previous sections we spent a great deal of time on filters and the syntax to build them. Although filters are incredibly powerful, the power is magnified tenfold if we can build them dynamically (at run time) using templates. Take the case of providing a user with a text field in which to type some search data. How can we form a search using this data? The user might type in a phone number, or she might type in "tony." We should choose a search filter that includes `sn` in one case but `telephoneNumber` in the other case. It would also be wasteful to use a search of the form `(|(cn=searchstring)(telephoneNumber=searchstring))`, since the server is comparing against extra fields that are not needed and it might return unwanted results.

A filter configuration file allows building rules that can present the filter based on input at run time. The filter configuration file has the following format:

Tag					
	pattern	delimiters	filter1-1	description	(optional scope)
			filter1-n	description	(optional scope)
	pattern2	delimiters	filter2-1	description	(optional scope)

As an example, let's look at a filter configuration file we will be using:

```
"search"
          "="                 " "    "%v"                              "arbitrary"
          "^[0-9][0-9-]*$"    " "    "(telephoneNumber=*%v*)"          "phone number"
          "@"                 " "    "(mail=%v)"                       "email address"
                                     "(mail=%v*)"                      "start of email"
          "^.[. _].*"         ". _"  "(cn=%v1* %v2-)"                  "first initial"
          ".*[. _].$"         ". _"  "(cn=%v1-*)"                      "last initial"
          "[. _]"             ". _"  "(|(sn=%v1-)(cn=%v1-))"           "exact"
                                     "(|(sn~=%v1-)(cn~=%v1-))"         "approximate"
          "*"                 "."    "(|(cn=%v1)(sn=%v1)(uid=%v1))"    "exact"
                                     "(|(cn~=%v1)(sn~=%v1))"           "approximate"
```

The tag is used to identify a block of patterns, allowing the mixing of multiple patterns in a single filter configuration file. The filter configuration file shown here has only one tag, named `search`. The patterns are regular expressions that are applied to the search string entered by the user. The first pattern—"="—indicates that if the search string entered by the user contains "=" anywhere, then the designated filter, in this case `%v`, should be applied.

Filters are built using the text entered by the user and static text from the configuration file. The filter `%v` indicates a variable substitution. By itself, `%v` means the whole string entered by the user. If we used the template `(mail=%v)` and the user entered "tony@abc.com," a filter string of `(mail=tony@abc.com)` would be built.

The filter `%v` has a series of different modifiers. Assume for the examples shown in Table 5-2 that the search string is "this is a test." Words are determined and split on the basis of the characters entered in the second column of the configuration file—the delimiter column.

Returning to our configuration file example, if the user enters a string of the form "cn=tony" (in the pattern "="), then the first rule will be used: return a filter consisting of the whole string entered by the user. This pattern allows advanced users to directly build their own filters at run time. The next pattern—`^[0-9][0-9-]*$`—is used to detect if a phone number has been entered. The character `^` indicates to start at

TABLE 5-2. *"This is a test" search.*

SYNTAX	ENTITY REPRESENTED	SAMPLE	RESULT
`%v`	Whole value entered	`%v`	`"this is a test"`
`%vN`	Word N	`%v2`	`"is"`
`%vN-`	Word N and all words following	`%v2-`	`"is a test"`
`%vN-M`	Words N through M	`%v3-4`	`"a test"`
`%v$`	Last word	`%v$`	`"test"`

the beginning and use the filter if the search string contains one or more digits: if the user types one or more digits, then return the filter `(telephoneNumber=*digits*)`.

The following Java command-line program will allow you to try out filters and `search` commands. The program presents all the filters that match a search string and builds a filter expression that can be issued to an LDAP search.

```
import netscape.ldap.*;
import netscape.ldap.util.*;
import java.util.*;

// Class to experiment with filter configuration files
public class CreateFilter {

    public static void main( String[] args ) {

        if ( args.length != 2 ) {
            System.out.println( "Usage: java CreateFilter " +
                                "<filterfile> <search " +
                                "expression> ");
            System.out.println( "Example:" );
            System.out.println( "    java CreateFilter " +
                                "tryfilt.conf \"*peter*\"" );
            System.exit( 1 );
        }

        LDAPFilterDescriptor filterDesc = null;
        LDAPFilterList filtlist = null;
        String srchfilter = "";
        int numfilts = 0;
        try {
            // Read a filter configuration file
            filterDesc = new LDAPFilterDescriptor( args[0] );
        } catch (Exception e) {
            System.out.println( "Cannot load file: " +
                                args[0] );
            System.exit(0);
        }

        try {
            // Construct filters from the parsed configuration
            // file and the search expression from the command
            // line
            filtlist =
                filterDesc.getFilters( "search", args[1] );
            numfilts = filtlist.numFilters();
            if ( numfilts > 1 ) {
                srchfilter += "(|";
            }
            // Iterate through constructed expressions
            while ( filtlist.hasMoreElements() ) {
                LDAPFilter fline = filtlist.next();
                String fstr = fline.getFilter();
                System.out.println( "Filter = " + fstr );
                // Concatenate the individual matches
                srchfilter += fstr;
            }
            if ( numfilts > 1 ) {
                srchfilter += ")";
            }
        } catch ( Exception e ) {
            System.out.println( "Filter error: " +
                                e.toString() );
        }
        System.out.println( "Search filter = " +
                            srchfilter );
    }

}
```

Some sample runs and corresponding output follow:

```
java CreateFilter tryfilt.conf 213
filter=(telephoneNumber=*213*)
search string=(|(telephoneNumber=*213*))
```

The above response indicates that a number was detected and that the filter for `telephoneNumber` was built. The value typed by the user—213—was inserted into a search string. Note that the program builds a search string by prepending it with `"(|"` and appending it with `")"`. The search string is built in this way to handle the case in which multiple filters may be returned, as in the next example:

```
java CreateFilter tryfilt.conf tony
filter=(|(cn=tony)(sn=tony)(uid=tony))
filter=(|(cn~=tony)(sn~=tony))
search string=(|(|(cn=tony)(sn=tony)(uid=tony))(|(cn~=tony)
(sn~=tony)))
```

Let's focus on the code used to handle the input and return these filters. Before anything can occur with a filter configuration, an `LDAPFilterDescriptor` object needs to be created. The constructor for an `LDAPFilterDescriptor` can take a file name, a `StringBuffer` containing the filter configuration information, or a URL to the filter file (allowing the file to exist anywhere on the Web).

The following instruction will read the file.

```
LDAPFilterDescriptor filtdesc = new LDAPFilterDescriptor("filename");
```

The next step is to call the `getFilters` method, passing in your search string and the tag for the section to use as an example:

```
LDAPFilterList filtlist = filltdesc.getFilters("tag","search string");
```

This method will return an enumerated list that can be iterated over to retrieve each filter and other information.

The following code fragment shows how to enumerate the filters.

```
while (filtlist.hasMoreElements()) {
    LDAPFilter fline = filtlist.next();
    System.out.println("description:"+ fline.getDescription());
    System.out.println("filter="+ fline.getFilter());
}
```

The primary information we need to retrieve is the filter, which is obtained with the `getFilter` method.

There are many advantages to using filter configuration files. They eliminate the need to predefine searches in the code, and they provide flexibility at run time for dynamically tailoring a query based on information provided by a user. The CD-ROM for this book contains a graphical Java application, a screen shot of which is shown in Figure 5-5. The application takes a search string entered by a user and issues a query against the directory. The results are displayed in a scrollable text box. The code demonstrates use of a filter configuration file, the searching functions of the LDAP SDK, and some AWT (Abstract Windows Toolkit) user interface code as well. The code is presented here for review, and it will be extended in the next section in our discussion of client-side sorting.

```
import java.lang.*;
import java.awt.*;
import java.awt.event.*;
import netscape.ldap.*;
import netscape.ldap.util.*;
import java.util.*;

/**
 * Frame to select filters from a filter file and do searches
 */

public class FilterSearchDialog extends Frame {

    /**
     * Launch a frame to do searches using a filter file
     *
     * @param args host, port, authDN, password, base
     */
    public static void main( String[] args ) {
        if ( (args.length != 4) &&
             (args.length != 6 ) ) {
            System.out.println( "Usage: java " +
                                "FilterSearchDialog " +
                                "<host> <port> " +
                                "<filterfile> <baseDN> " +
                                "[<authdn> <password>]" );
            System.out.println( "Example:" );
            System.out.println( "    java " +
                               "FilterSearchDialog " +
                               "localhost 389 " +
                               "filter.conf \"o=airius.com\"" );
            System.exit(1);
        }

        String host = args[0];
        int port = Integer.parseInt( args[1] );
        String conf = args[2];
        String base = args[3];
        String authid = "";
        String authpw = "";
        if ( args.length > 4 ) {
            authid = args[4];
            authpw = args[5];
        }

        Frame f = new FilterSearchDialog(
            "Graphical LDAP Search", host, port,
            authid, authpw, conf, base );
        f.setSize( 430,280 );
        f.show();
    }

    /**
     * Standard Frame constructor, plus connection parameters
     *
     * @param title window title
     * @param host host to search
     * @param port port number of server
     * @param authid DN to authenticate as (may be "")
     * @param authpw password for authentication (may be "")
     * @param conf name of filter configuration file
     * @param base base DN for subtree search
     */
    public FilterSearchDialog( String title,
                               String host, int port,
                               String authdn, String authpw,
                               String conf, String base ) {
        super( title );

        this.host = host;
        this.port = port;
        this.authdn = authdn;
        this.authpw = authpw;
        this.conf = conf;
        this.base = base;

        setLayout(null);
```

The following block of code handles disconnecting from the LDAP server when the user closes the window.

```
this.addWindowListener(new WindowAdapter() {
    public void windowClosing( WindowEvent e ) {
        // Disconnect from server
        if ( (ld != null) && ld.isConnected() ) {
            try {
                ld.disconnect();
            } catch ( LDAPException le ) {
                System.out.println( le.toString() );
            }
        }
        System.exit(0);
    }
});
```

The following code creates the GUI components and places them on the frame.

```
Label lbl1 = new Label( "Search for:" );
lbl1.setBounds( 10,36,75,26 );
add( lbl1 );
srch = new TextField();
Font font = new Font("Monospaced",Font.PLAIN,12);
srch.setFont( font );
srch.setBounds( 90,36,230,26 );
add( srch );

searchb = new Button( "Search" );
searchb.setBounds( 340,36,80,26 );
add( searchb );

output = new TextArea(12,3);
output.setFont( font );
output.setEditable( false );
output.setBounds( 10,70,410,200 );
add( output );
```

An action is associated with the Search button:

```
searchb.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        String srchstr = srch.getText();
        // If there is a search string, do a search
        if ( srchstr.length() > 0 ) {
            searchLDAP( srchstr );
        }
```

```
            }
        });
```

The following code handles reading the filter description file and creating a filter descriptor. If the file cannot be located, the program aborts, reporting an error to the console.

```
        // Read the filter description file
        // If not found, exit and report error to the console
        try {
            filterDesc =
                new LDAPFilterDescriptor(conf);
        } catch ( Exception e ) {
            System.out.println( "Cannot load " + conf +
                                " file" );
            System.out.println( "Exiting..." );
            System.exit( 1) ;
        }
    }
```

The searchLDAP method takes the search string from the user and builds a filter with the aid of the filter descriptor. Once the proper search string has been built, an LDAP search is executed:

```
    protected void searchLDAP( String srchString ) {
        int status = -1;
        String appendmsg = "";
        LDAPSearchResults res = null;
        LDAPFilterList filtlist = null;
        String  srchfilter = "(|";

        // Check if we are connected first
        if ( (ld == null) || (!ld.isConnected()) ) {
            connectServer();
        }

        // Use the filter descriptor to build a
        // search filter
        try {
            filtlist =
                filterDesc.getFilters("search", srchString);

            while ( filtlist.hasMoreElements() ) {
                LDAPFilter fline = filtlist.next();
                // The actual filter is next
                String fstr = fline.getFilter();
                srchfilter += fstr;
            }
            srchfilter += ")";
            if ( srchfilter.length() == 3 ) {
                // No filters found
                return;
            }

            // Now do the search
            res = ld.search( base,
                             scope,
                             srchfilter,
                             ATTRS,
                             false );

            // Display search filter
            String outres = "Filter=" + srchfilter + "\n" +
                            "-------\n";

            // Loop on results building each line
            while ( res.hasMoreElements() ) {
                try {
                    // Next directory entry
                    LDAPEntry entry = res.next();
                    outres += format( entry );
                    status = 0;
                } catch ( LDAPReferralException e ) {
                    // Ignore referrals
                    continue;
                } catch (LDAPException le) {
                    int rc = le.getLDAPResultCode();
                    if ( rc == le.SIZE_LIMIT_EXCEEDED ) {
                        appendmsg =
                            "\nExceeded size limit";
                    } else if ( rc ==
                                le.TIME_LIMIT_EXCEEDED ) {
                        appendmsg =
                            "\nExceeded time limit";
                    } else {
                        appendmsg = le.toString();
                    }
                }
            }
```

```
            outres += appendmsg;
            output.setText( outres ); // Display in text area

        } catch( Exception e ) {
            System.out.println( "Search error: " +
                                 e.toString() );
        }
     }
```

The connectServer method is called whenever we need to establish a connection to the LDAP server:

```
    protected void connectServer() {
        // Connect to the LDAP server
        if ( (ld == null) || (!ld.isConnected()) ) {
            try {
                ld = new LDAPConnection();
                ld.connect( host, port, authdn, authpw );
            } catch( LDAPException e ) {
                 System.out.println( "Connect error: " +
                                     e.toString() );
                System.exit( 1 );
            }
        }
    }
```

We are binding with the specified command-line credential information. If none is supplied, we simply bind anonymously.

The following two methods return a display string for the text box. They handle situations in which no value exists by substituting a dash for the value. The format method returns a String with each matching entry in tab-delimited format. The returned String is directly appended to the text box by the calling method.

```
    /**
     * Format a string with attribute values from an entry,
     * separated by tabs
     *
     * @param entry LDAP entry containing cn, telephoneNumber,
     * and mail
     */
    public String format( LDAPEntry entry ) {
        String outstr = "";

        // Get the data - hard-coded attribute names here!
        String name = getValue( entry, "cn" );
        String phone = getValue( entry, "telephoneNumber" );
        String email = getValue( entry, "mail" );

        // Limit the full name to 15 characters
        if ( name.length() > 15 ) {
            name = name.substring( 0, 15 );
        }

        outstr = name + "\t" + phone + "\t" + email + "\n";

        return outstr;
    }

    /**
     * Get first string value of an attribute from an entry
     * or '-' if not present
     *
     * @param entry LDAP entry containing the attribute
     * @param attrName name of attribute to retrieve
     * @return first value of attribute or '-'
     */
    protected String getValue( LDAPEntry entry,
                               String attrName ) {
        LDAPAttribute attr = entry.getAttribute( attrName );
        if ( attr == null ) {
            return "-";
        }
        Enumeration enumVals = attr.getStringValues();
        // Enumerate on values for this attribute
        boolean hasVals = false;
        if ( (enumVals == null) ||
             !enumVals.hasMoreElements() ) {
            return "-";
        }

        return (String)enumVals.nextElement();
    }

    private LDAPConnection ld = null;
    private String host;
    private int port;
    private String conf;
    private String base;
    private String authdn;
    private String authpw;
    private int scope = LDAPConnection.SCOPE_SUB;
    private TextArea output;
    private TextField srch;
    private Button searchb;
    // Attributes to display for each entry found
    private static final String[] ATTRS =
        {"cn","mail","telephoneNumber"};
    // Filter configuration file object
    private LDAPFilterDescriptor filterDesc = null;
}
```

This program is useful as an example of working with the results from a search and presenting them in a graphical environment. When the program is first started, a Connect button is displayed. Clicking this button will open a connection to the LDAP server and rename the button as Search. After a search string is entered, clicking the Search button will cause the search string to be parsed using a filter configuration file, and then the search will be submitted to the server. Search filter configurations provide other functionality as well, such as filter prefixing and suffixing.

Check the reference section of this book for details on these features. It should be evident that this functionality can make your code more dynamic in response to user input.

To execute the program, use the following command-line option:

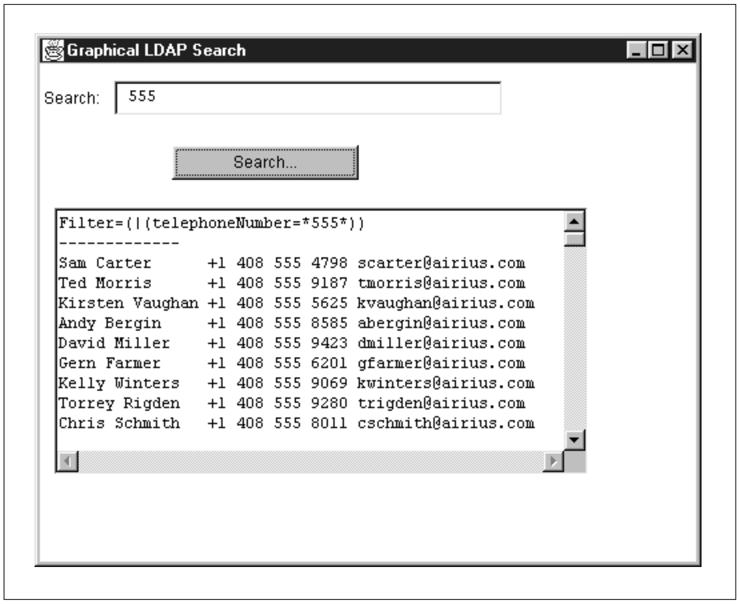
```
java FilterSearchDialog localhost 389 filter.conf "o=airius.com"
```

This command will present a search screen, where you may issue searches against the directory. Issuing a search of 555 against the sample data file included on the CD-ROM that accompanies this book will result in the display shown in Figure 5-5.

Sorting

We now turn our attention to sorting the results returned from the server. As you may have noticed, the LDAP server does not always return results in a natural order. Human beings generally prefer to have information ordered so that it can be reviewed or browsed easily. The LDAP SDK provides two methods for sorting results: client-side sorting and server-side sorting. **Server-side sorting** is an LDAPv3 enhancement and is supported on many servers, including Netscape Directory Server. We will cover server-side sorting in detail in Chapter 16. **Client-side sorting** is the option to retrieve the data and sort it on the client machine before working with the results.

Client-side sorting has a couple of restrictions. First, the attributes on which you wish to sort must be among the attributes you request in your search results. You can-



```
Filter=(|(telephoneNumber=*555*))
-------------
Sam Carter      +1 408 555 4798 scarter@airius.com
Ted Morris      +1 408 555 9187 tmorris@airius.com
Kirsten Vaughan +1 408 555 5625 kvaughan@airius.com
Andy Bergin     +1 408 555 8585 abergin@airius.com
David Miller    +1 408 555 9423 dmiller@airius.com
Gern Farmer     +1 408 555 6201 gfarmer@airius.com
Kelly Winters   +1 408 555 9069 kwinters@airius.com
Torrey Rigden   +1 408 555 9280 trigden@airius.com
Chris Schmith   +1 408 555 8011 cschmith@airius.com
```

FIGURE 5-5. *Results of sample query with* `FilterSearchDialog`.

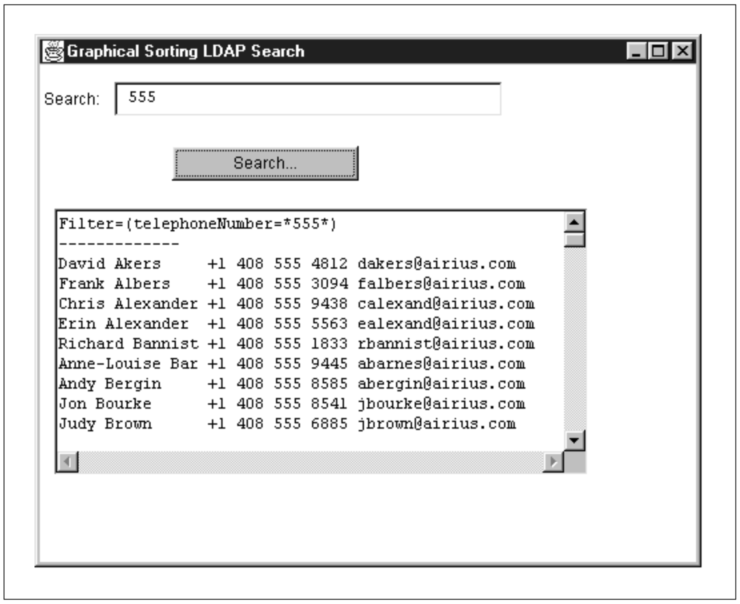
not, for instance, request just the `uid` and `telephoneNumber` for your search, and then try to sort by `cn`. Second, in client-side sorting, the sort will block until all records have been retrieved from the server. With these restrictions in mind, let's look at how we can add client-side sorting to the `FilterSearchDialog` program.

The client-side sort routine needs two arrays or two single values that indicate the attributes to sort and a flag for ascending or descending order. The following code snippet shows the modifications to the `FilterSearchDialog` code that are needed to sort by `cn` in ascending order.

```
res = ld.search( BASE, SCOPE, srchfilter, ATTRS, false;
// Since we are sorting by only one field, we do not need an array
res.sort ( new LDAPCompareAttrNames("cn",true) );
```

The `LDAPCompareAttrNames` constructor creates a comparator that looks at LDAP string values in the entries for sorting purposes. The `LDAPCompareAttrNames` constructor also takes a form with two arrays. For instance, to sort on both the `cn` and `telephoneNumber` attributes, the code would look like this:

```
String[] sortattr = {"cn", "telephoneNumber"};
boolean[] ascend = {true, true};
res.sort new ( LDAPCompareAttrNames(sortattr,ascend) );
```



```
Filter=(telephoneNumber=*555*)
-------------
David Akers     +1 408 555 4812 dakers@airius.com
Frank Albers    +1 408 555 3094 falbers@airius.com
Chris Alexander +1 408 555 9438 calexand@airius.com
Erin Alexander  +1 408 555 5563 ealexand@airius.com
Richard Bannist +1 408 555 1833 rbannist@airius.com
Anne-Louise Bar +1 408 555 9445 abarnes@airius.com
Andy Bergin     +1 408 555 8585 abergin@airius.com
Jon Bourke      +1 408 555 8541 jbourke@airius.com
Judy Brown      +1 408 555 6885 jbrown@airius.com
```

FIGURE 5-6. *Results with client-side sorting.*

The output from `FilterSearchDialogSort` (the sorting version of `FilterSearchDialog`) with the same search as earlier looks like Figure 5-6.

Authenticating for Searches

None of the examples presented so far in this chapter have involved authenticating to the directory. All connections have been anonymous (not using a DN or password). Most LDAP directories are configured to allow anonymous searching of at least some of the information in the system, but some attributes may have access control configured to prevent access. For instance, the corporate directory at Netscape Communications Corporation does not allow anonymous connections to retrieve the JPEG photo of an employee. Only security personnel or the employee corresponding to the entry may retrieve this attribute. The same usually is true for the `userPassword` attribute as well.

If you ask for one of the required or commonly used attributes and it is not returned, more than likely access control has been configured to prevent you from retrieving it. We will cover authenticating to the directory in Chapter 6, when we will be modifying data, but to peek ahead—for those who need to authenticate for a search or to retrieve a specific attribute—the additional method to call is shown here (indicated by bold):

```
ld = new LDAPConnection();
ld.connect( HOST, PORT ); // Connect to server
ld.authenticate( 3, authid, authpw );  // Bind by DN and password
```

The connect and authenticate steps can be combined:

```
ld.connect( 3, HOST, PORT, authid, authpw );
```

The optional first numerical parameter is the requested LDAP protocol version. You must specify `3` to take advantage of controls and other new features of LDAPv3. On the other hand, if the server supports only LDAPv2 and you specify `3` when authenticating, the server will refuse the connection. The default in the SDK is 2.

The value used to authenticate to the directory must be a DN of an existing entry. One DN in the server—the root DN (like the root user on UNIX)—has unlimited privileges and does not correspond to a physical entry. In Netscape Directory Server, the default root DN is `cn=Directory Manager`. The root DN is also often called the Directory Manager.

Typically the DN used to authenticate will be that of a user needing to perform an operation on his own entry. For example, if Sam Carter wanted to bind and retrieve his photograph, he would authenticate as follows:

```
String bindDN = "uid=scarter, ou=People, o=airius.com";
// Bind password is passed to us
if (bindpwd.length() > 0) {
    ld.authenticate(bindDN,bindpwd);
}
```

When authenticating, always validate that the password is not a blank string (`""`) or `null`. If a blank string is passed as the password, there will be no exception thrown to indicate an invalid authentication. Instead the operation will succeed but the connection will be anonymous. Later, when the program attempts to modify an entry, an exception may be thrown because anonymous users do not have the right to make modifications. We will cover authenticating in detail in Chapter 6.

Improving Directory Search Performance

As an application developer you can increase the performance of your search operations, reduce memory usage, and reduce the load on the server by observing a few rules of thumb:

- Use indexed attributes
- Specify an object class in your filter to get only entries of the desired type
- Retrieve only attributes you need
- Keep the DN handy
- Use `compare` where it makes sense

Use Indexed Attributes

The most significant way to get good performance from the directory when searching is to use only indexed attributes in your search requests. As a programmer you may need to work with your directory administrator to determine which attributes are indexed or to request that additional attributes be indexed. If you find you need to perform searches frequently on unindexed attributes, then it may make sense to index the particular attributes. With Netscape Directory Server, you can view the access logs and determine if searches are occurring against unindexed fields. The following is a sample of the access log. The text "notes=U" marks a search against an unindexed attribute.

```
[03/May/1999:09:24:29 -0400] conn=19 op=6 SRCH
base="ou=tony.home,o=NetscapeRoot" scope=2 filter="(objectclass=NsHost)"
     .
     .
[03/May/1999:09:24:29 -0400] conn=19 op=6 RESULT err=0 tag=101 nentries=1
etime=0 notes=U
```

Specify an Object Class to Get Only Entries of the Desired Type

If your application is working with particular types of records (for example, person records), it makes sense to include in your filter the object class you need. For instance, for all the records of people whose names begin with "barbara," use a filter such as `(&(objectclass=person)(cn=Barbara*))`. You can use a filter configuration file to set up a tag for finding entries that represent people. Include the filter component `objectclass=person` in the tag. The result may be that fewer records are returned to the client, and consequently that performance is improved, particularly if your directory stores many entries for objects other than people.

Retrieve Only Attributes You Need

Many programmers pass `null` as the attributes field for a `search` operation. The result is that all attributes for the indicated records are returned. For a large potential result set, the performance of both server and client can be severely affected. If you need only the name and phone number, then specify these in your search request. Keep in mind that if you are doing client-side sorting, you will also need to request the attributes by which you wish to sort.

Keep the DN Handy

If you are going to do anything else with a retrieved record, keep the DN. The DN can be used to find a record uniquely within the directory without invoking a new search. For instance, suppose you are displaying a list of names and want to allow the user to click on a name and get all the information about that person. Store the DN for each record in a nonvisible variable and use it to look up the record when the user clicks on it.

Use `compare` Where It Makes Sense

If you are interested only in whether an attribute exists and has a certain value, use `compare` rather than `search`. A `compare` is a lightweight transaction with very little client and server overhead. When entries are returned, access control must be evaluated for each attribute, and client memory usage increases according to the size of the entries returned.

Conclusion

The major use of an LDAP directory is to retrieve information. In this chapter we have presented samples of code to direct searches to LDAP directories. We have also covered the details of processing the results, along with many tips to make the most efficient use of the SDK. One of the key pieces of information to maintain during processing is the DN of the retrieved records. With this information, any other data can be obtained rapidly from the directory. We have also included in this chapter a discussion of techniques to minimize impact on the directory, the client application, and the network through efficient use of the SDK.