

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

Don't Redo It, Reuse It: LDAP JavaBeans

Java is an object-oriented language that lends itself well to encapsulating functionality as components. The JavaBeans specification takes this encapsulation a step further, defining how a component publishes its properties and methods and how other components can discover and access those properties and methods. Many Java development environments support the JavaBeans specification. Because of this support, you can write a component that complies with the specification, and that component can be dropped into the component palette of any of these Java development environments. The component you write can be used again in future projects or by other developers, who need no knowledge about the component's implementation.

Directory SDK for Java includes a few sample JavaBeans. Although each Java Bean does very little, each one provides a single piece of commonly used directory functionality. The JavaBeans also encapsulate some of the implementation details of LDAP, which means that developers can use these JavaBeans without knowing very much about LDAP.

Invisible LDAP JavaBeans

The first set of JavaBeans we'll look at are "invisible Beans." These JavaBeans wrap subsets of the SDK functionality, but they provide no GUI interface.

All these invisible JavaBeans extend the common base class `LDAPBasePropertySupport`, illustrated in Figure 10-1. This class provides standard methods for accessing SDK properties, a method for connecting to the directory, and an event dispatcher for property change events. Property change events are the simplest and probably the most common events used by JavaBeans to notify other components of significant changes. Let's take a look at the implementation of the `LDAPBasePropertySupport` class.

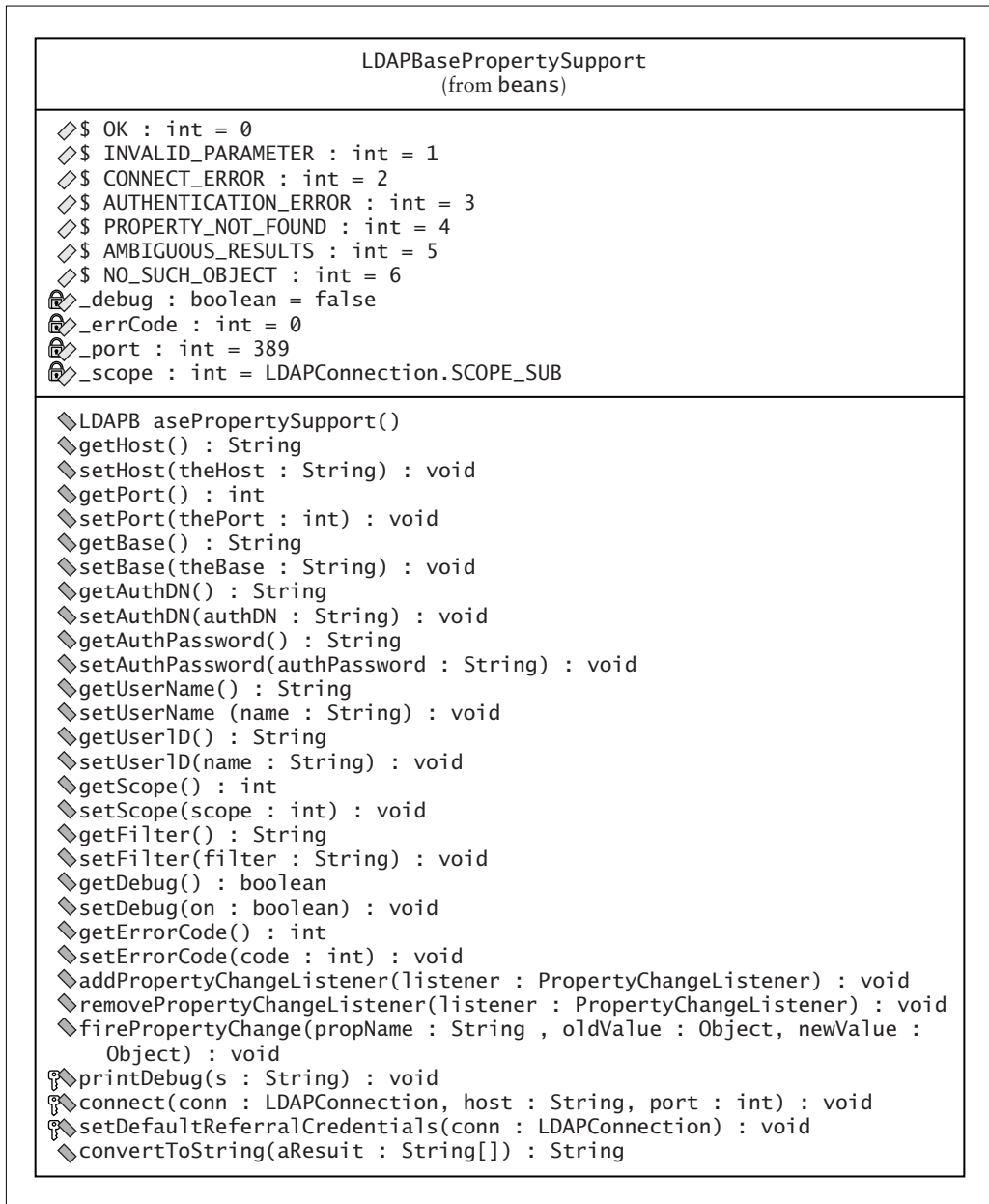


FIGURE 10-1. LDAPBasePropertySupport.

LDAPBasePropertySupport

One of the requirements of a JavaBean is that it must be serializable; that is, the object must be able to be written to disk or transferred over a network and then reconstituted. In addition, the JavaBean must have a public constructor that takes no parameters. These two requirements are related: a JavaBean object can be stored persistently (which requires it to be serializable) and restored later (which requires a constructor with no parameters). The LDAPBasePropertySupport class fulfills both of these requirements:

```
public class LDAPBasePropertySupport implements Serializable {
    /**
     * Constructor with no parameters
     */
    public LDAPBasePropertySupport() {}
}
```

Next, accessor methods are declared for the properties used to connect to the directory and to search the directory. These properties include the host name and port number of the LDAP server, the DN and password used for authentication, and the base DN, scope, and filter used for the search. Only the accessors for the host name property are included in the block of code shown here; the others are similar.

```
/**
 * Returns the host to search at
 * @return DNS name or dotted IP name of host to search at
 */
public String getHost() {
    return _host;
}

/**
 * Sets host string.
 * @param theHost host name
 */
public void setHost( String theHost ) {
    _host = theHost;
}
```

The invisible JavaBeans provided with the SDK use property change events to notify other components of the results of operations. The base class provides support for registering interest in property change events and for “firing” them:

```
/**
 * Add a client to be notified when an authentication result is in
 * @param listener a client to be notified of changes
 */
public void addPropertyChangeListener(
        PropertyChangeListener listener ) {
    System.out.println( "Adding listener " + listener );
    m_propSupport.addPropertyChangeListener( listener );
}

/**
 * Remove a client that had requested notification on authentication
 * @param listener a client not to be notified of changes
 */
public void removePropertyChangeListener(
        PropertyChangeListener listener ) {
    m_propSupport.removePropertyChangeListener( listener );
}

/**
 * Support for bound property notification
 * @param propName name of changed property
 * @param oldValue previous value of property
 * @param newValue new value of property
 */
public void firePropertyChange(
        String propName,
        Object oldValue,
        Object newValue ) {

    if (m_propSupport == null)
        m_propSupport = new PropertyChangeSupport( this );

    m_propSupport.firePropertyChange( propName, oldValue, newValue );
}
```

The LDAPBasePropertySupport class also defines two utility methods, shown in the block of code that follows. The printDebug method prints out debugging information, and the convertToString method converts a String array to a single String

(with line feeds delimiting the fields in the single `String`). The `convertToString` method is useful if you are using this class in JavaScript (through `LiveConnect`). You can use `convertToString` to supply results as a single `String` (JavaScript does not support arrays as return values). This method is not declared static, because JavaScript in earlier browsers doesn't handle inheritance of static methods properly.

```
protected void printDebug( String s ) {
    if ( _debug )
        System.out.println( s );
}

/**
 * Utility method to convert a String array to a single String
 * with line feeds between elements.
 * @param aResult the String array to convert
 * @return a String with the elements separated by line feeds
 */
public String convertToString( String[] aResult ) {
    String sResult = "";
    if ( null != aResult ) {
        for ( int i = 0; i < aResult.length; i++ ) {
            sResult += aResult[i] + "\n";
        }
    }
    return sResult;
}
```

Finally, a method to connect to a directory is defined. It is protected, since it is to be used by derived Beans and not by external components. The `connect` method, shown in the block of code that follows, checks if the Bean is running as an applet in Netscape Navigator, and if so, requests permission to make network connections. We discussed the need to request permission to establish network connections in Chapter 8.

The method also sets up automatic following of referrals using the same credentials that were supplied for the original connection. Referral processing will be discussed in detail in Chapter 16.

```
/**
 * Sets up basic connection privileges for Navigator if necessary,
 * and connects
 * @param host host to connect to
 * @param port port number
 * @exception LDAPException from connect()
 */
```

```
protected void connect( LDAPConnection conn, String host, int port )
    throws LDAPException {
    boolean needsPrivileges = true;
    /* Running stand-alone? */
    SecurityManager sec = System.getSecurityManager();
    printDebug( "Security manager = " + sec );
    if ( sec == null ) {
        printDebug( "No security manager" );
        /* Not an applet; we can do what we want to */
        needsPrivileges = false;
    /* Cannot do instanceof on an abstract class */
    } else if ( sec.toString().startsWith(
        "java.lang.NullSecurityManager" ) ) {
        printDebug( "No security manager" );
        /* Not an applet; we can do what we want to */
        needsPrivileges = false;
    } else if ( sec.toString().startsWith(
        "netscape.security.AppletSecurity" ) ) {

        /* Connecting to the local host? */
        try {
            if ( host.equalsIgnoreCase(
                java.net.InetAddress.getLocalHost().getHostName() ) )
                needsPrivileges = false;
        } catch ( java.net.UnknownHostException e ) {
        }
    }

    if ( needsPrivileges ) {
        /* Running as applet. Is PrivilegeManager around? */
        String mgr = "netscape.security.PrivilegeManager";
        try {
            Class c = Class.forName( mgr );
            java.lang.reflect.Method[] m = c.getMethods();
            if ( m != null ) {
                for( int i = 0; i < m.length; i++ ) {
                    if ( m[i].getName().equals(
                        "enablePrivilege" ) ) {
                        try {
                            Object[] args = new Object[1];
                            args[0] =
                                new String( "UniversalConnect" );
                            m[i].invoke( null, args );
                        }
                    }
                }
            }
        }
    }
}
```

```

        printDebug(
            "UniversalConnect enabled" );
    } catch ( Exception e ) {
        printDebug( "Exception on invoking " +
            "enablePrivilege: " +
            e.toString() );

        break;
    }
    break;
}
}
} catch ( ClassNotFoundException e ) {
    printDebug( "no " + mgr );
}
}

conn.connect( host, port );
setDefaultReferralCredentials( conn );
}

```

The `setDefaultReferralCredentials` method, illustrated in the block of code that follows, creates and configures an object used to handle any referrals encountered during a search. The object holds the DN and password used to authenticate to the original server. When the Bean is “referred” to another LDAP server, the DN and password in the object are used to authenticate to the new server.

```

protected void setDefaultReferralCredentials(
    LDAPConnection conn ) {
    final LDAPConnection m_conn = conn;
    LDAPRebind rebind = new LDAPRebind() {
        public LDAPRebindAuth getRebindAuthentication(
            String host,
            int port ) {
            return new LDAPRebindAuth(
                m_conn.getAuthenticationDN(),
                m_conn.getAuthenticationPassword() );
        }
    };
    LDAPSearchConstraints cons = conn.getSearchConstraints();
    cons.setReferrals( true );
    cons.setRebindProc( rebind );
}

```

In the following block of code, error codes are defined for use by derived classes as well as by clients. Clients do not need to know about the extensive list of error codes in `LDAPException`.

```
/*
 * Variables
 */
/* Error codes from search operations, etc. */
public static final int OK = 0;
public static final int INVALID_PARAMETER = 1;
public static final int CONNECT_ERROR = 2;
public static final int AUTHENTICATION_ERROR = 3;
public static final int PROPERTY_NOT_FOUND = 4;
public static final int AMBIGUOUS_RESULTS = 5;
public static final int NO_SUCH_OBJECT = 6;
```

Finally, variables storing the Bean properties are declared:

```
private boolean _debug = false;
private int _errCode = 0;
private String _host = new String("localhost");
private int _port = 389;
private int _scope = LDAPConnection.SCOPE_SUB;
private String _base = new String("");
private String _filter = new String("");
private String _authDN = new String("");
private String _authPassword = new String("");
private String _userName = new String("");
private String _userID = new String("");
transient private PropertyChangeSupport m_propSupport =
    new PropertyChangeSupport( this );
```

LDAPSimpleAuth

Many applications, particularly server-side applications, use a directory solely to authenticate a user (to determine if a user is who he says he is). We've already looked at authenticating to the directory, starting in Chapter 6. Not much to it. So why do we need a JavaBean to do authentication? The answer will become apparent very soon.

`LDAPSimpleAuth`, illustrated in Figure 10-2, is a JavaBean for LDAP authentication.

As with all other JavaBeans, `LDAPSimpleAuth` is serializable and has a constructor with no parameters. For convenience, it also has constructors to allow setting

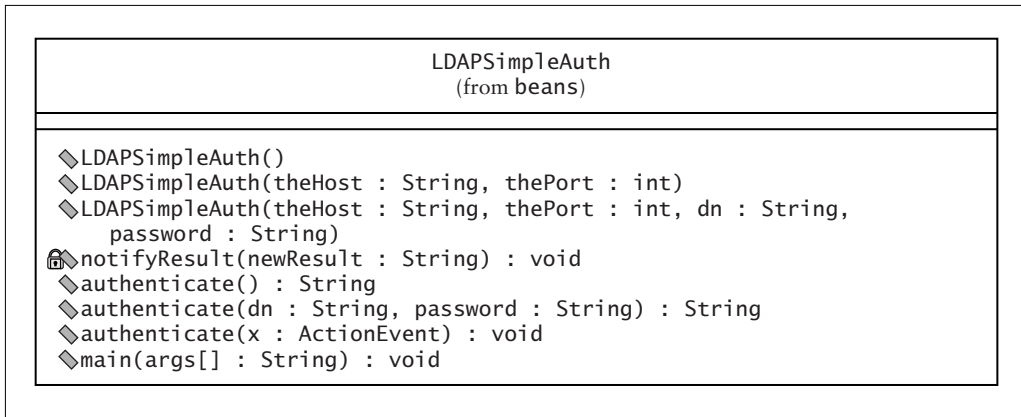


FIGURE 10-2. LDAPSimpleAuth.

some or all parameters in one shot, rather than through the methods for setting individual properties:

```

public class LDAPSimpleAuth extends LDAPBasePropertySupport
    implements Serializable {

    /**
     * Constructor with no parameters
     */
    public LDAPSimpleAuth() {}

    /**
     * Constructor with host and port initializers
     * @param theHost host string
     * @param thePort port number
     */
    public LDAPSimpleAuth( String theHost,
                          int thePort ) {
        setHost( theHost );
        setPort( thePort );
    }

    /**
     * Constructor with all required authentication parameters
     * @param theHost host string
     * @param thePort port number
  
```

```
* @param dn fully qualified distinguished name to authenticate
* @param password password for authenticating the DN
*/
public LDAPSimpleAuth( String theHost,
                      int thePort,
                      String dn,
                      String password ) {
    setHost( theHost );
    setPort( thePort );
    setAuthDN( dn );
    setAuthPassword( password );
}
```

The only public method (aside from a `main` method, which you can use to verify that the JavaBean works) is `authenticate`, which has two signatures. One signature has no parameters and relies on the properties you set by using the accessor methods. The other signature allows you to pass in the DN and the password you want to use for authentication. To facilitate use with JavaScript in a browser or on a Web server, `authenticate` returns a `String`. The `authenticate` method, shown in the following block of code, also notifies clients of results by firing property change events.

```
/**
 * Connect to LDAP server using parameters specified in
 * constructor and/or by setting properties, and attempt to
 * authenticate.
 * @return "Y" on successful authentication, "N" otherwise
 */
public String authenticate() {
    LDAPConnection m_ldc = null;
    String result = "N";
    try {
        m_ldc = new LDAPConnection();
        System.out.println("Connecting to " + getHost() +
                          " " + getPort());
        connect( m_ldc, getHost(), getPort());
    } catch (Exception e) {
        System.out.println( "Failed to connect to " + getHost() +
                            ": " + e.toString() );
    }
    if ( m_ldc.isConnected() ) {
        System.out.println( "Authenticating " + getAuthDN() );
        try {
            m_ldc.authenticate( getAuthDN(), getAuthPassword() );
        }
    }
}
```

```

        result = "Y";
    } catch (Exception e) {
        System.out.println( "Failed to authenticate to " +
            getHost() + ": " + e.toString() );
    }
}

try {
    if ( (m_ldc != null) && m_ldc.isConnected() )
        m_ldc.disconnect();
} catch ( Exception e ) {
}

notifyResult( result );
return result;
}

/**
 * Connect to LDAP server using parameters specified in
 * constructor and/or by setting properties, and attempt to
 * authenticate.
 * @param dn fully qualified distinguished name to authenticate
 * @param password password for authenticating the DN
 * @return "Y" on successful authentication, "N" otherwise
 */
public String authenticate( String dn,
                            String password ) {
    setAuthDN( dn );
    setAuthPassword( password );
    return authenticate();
}

```

The String return value makes it easy to use this Bean in JavaScript. In the following sample section of a Web page, the parameters are taken from HTML text fields in an HTML form called “input,” and the result is displayed in a JavaScript alert dialog box.

```

<SCRIPT LANGUAGE="JavaScript">
function checkAuthentication() {
    auth = new Packages.netscape.ldap.beans.LDAPSimpleAuth();
    auth.setHost( document.input.host.value );
    auth.setPort( parseInt(document.input.port.value) );
    auth.setAuthDN( document.input.dn.value );

```

```
auth.setAuthPassword( document.input.password.value );
// Must request rights to do network connections
netscape.security.PrivilegeManager.enablePrivilege("UniversalConnect");
// And for property reads, to get LDAP error strings
netscape.security.PrivilegeManager.enablePrivilege(
    "UniversalPropertyRead");
result = auth.authenticate();
if ( result == "N" )
    msg = "Incorrect password";
else
    msg = "Successful login";
alert( msg );
}
</SCRIPT>
```

Clients that prefer to listen for a property change event rather than checking the return value of `authenticate` are notified through `notifyResult`:

```
private void notifyResult( String newResult ) {
    firePropertyChange( "result", result, newResult );
    result = newResult;
}
```

LDAPGetEntries

When searching for a user entry in a directory, a typical user might enter part of the name or ID of the desired user and expect to select from a list of matching users. After selecting a name from the list, the user might then expect to see some properties of that entry. The `LDAPGetEntries` JavaBean (Figure 10-3) helps with the first part. It will accept part of a full name (`cn`), part of a user ID (`uid`), or both; search for all matching entries; and return a list of DNs. The full name and/or user ID may contain wild cards—for example, “john*” or “*ramer*.”

As with `LDAPSimpleAuth`, `LDAPGetEntries` has a constructor with no parameters (for deserialization), as well as constructors to allow setting some or all parameters at once, rather than through the methods for setting individual properties:

```
public class LDAPGetEntries extends LDAPBasePropertySupport
    implements Serializable {

    /**
     * Constructor with no parameters
     */
```

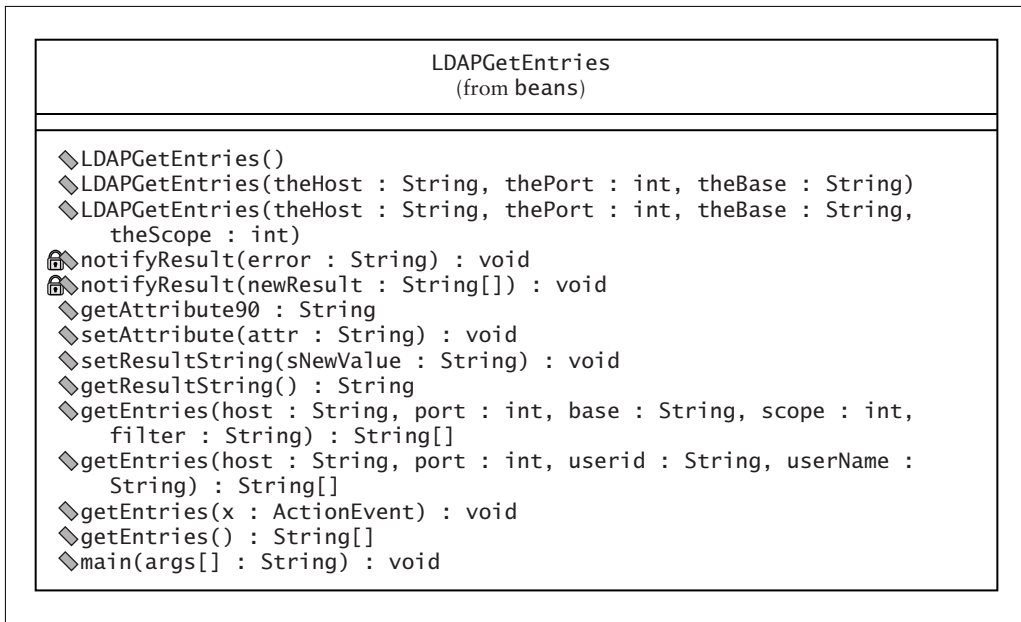


FIGURE 10-3. LDAPGetEntries.

```

public LDAPGetEntries() {
    super();
}

/**
 * Constructor with host, port, and base initializers
 * @param theHost host string
 * @param thePort port number
 * @param theBase directory base string
 */
public LDAPGetEntries( String theHost, int thePort, String theBase ) {
    setHost( theHost );
    setPort( thePort );
    setBase( theBase );
}

/**
 * Constructor with host, port, base, and scope initializers
 * @param theHost host string
 * @param thePort port number
 * @param theBase directory base string

```

```
* @param theScope one of LDAPConnection.SCOPE_BASE,
* LDAPConnection.SCOPE_SUB, LDAPConnection.SCOPE_ONE
*/
public LDAPGetEntries( String theHost,
                      int thePort,
                      String theBase,
                      int theScope ) {
    setHost( theHost );
    setPort( thePort );
    setBase( theBase );
    setScope( theScope );
}
```

As with `LDAPSimpleAuth`, clients can be notified when results are ready with a `PropertyChangeEvent` (delivered by `firePropertyChange`):

```
private void notifyResult( String error ) {
    firePropertyChange( "error", _errorMsg, error );
    _errorMsg = error;
}

private void notifyResult( String[] newResult ) {
    String sNewResult = convertToStrings( newResult );
    firePropertyChange( "result", result, newResult );
    _sResult = sNewResult;
    result = newResult;
}
```

There are accessors for the results:

```
public void setResultString( String sNewValue ) {
    _sResult = sNewValue;
}

public String getResultString() {
    return _sResult;
}
```

The most important method in this JavaBean is `getEntries`, shown in the following block of code. This is the method called by a client to begin the search. Parameters can be supplied with the method, or they can be assumed to be already provided with the constructor or using the accessors.

```
/**
 * Searches and returns values for a specified attribute
 * @param host host string
 * @param port port number
 * @param base directory base string
 * @param scope one of LDAPConnection.SCOPE_BASE,
 * LDAPConnection.SCOPE_SUB, LDAPConnection.SCOPE_ONE
 * @param filter search filter
 * @param attribute name of property to return values for
 * @return array of values for the property
 */
public String[] getEntries( String host,
                           int port,
                           String base,
                           int scope,
                           String filter) {

    setHost( host );
    setPort( port );
    setBase( base );
    setScope( scope );
    setFilter( filter );
    return getEntries();
}
```

```
/**
 * Searches and returns values for a specified attribute
 * @param host host string
 * @param port port number
 * @param base directory base string
 * @param scope one of LDAPConnection.SCOPE_BASE,
 * LDAPConnection.SCOPE_SUB, LDAPConnection.SCOPE_ONE
 * @param userName the user name
 * @param userid the user ID
 * @return array of DNS
 */
public String[] getEntries( String host,
                           int port,
                           String base,
                           int scope,
                           String userid,
                           String userName) {

    setHost( host );
    setPort( port );
```

```
        setBase( base );
        setScope( scope );
        if (userName == null)
            userName = new String("");
        setUsername( userName );
        if (userid == null)
            userid = new String("");
        setUserID( userid );
        return getEntries();
    }

    // Added this method in order to get exposed in BDK
    public void getEntries(ActionEvent x) {
        getEntries();
    }

    /**
     * Searches and returns values of a previously registered property,
     * using previously set parameters
     * @return array of values for the property
     */
    public String[] getEntries() {
        boolean invalid = false;
        if ((getUserName().length() < 1) && (getUserID().length() < 1) &&
            (getFilter().length() < 1)) {
            printDebug("No user name or user ID");
            invalid = true;
        } else if ( (getHost().length() < 1) || (getBase().length() < 1) ) {
            printDebug( "Invalid host name or search base" );
            invalid = true;
        }
        if ( invalid ) {
            setErrorCode( INVALID_PARAMETER );
            notifyResult( (String)null);
            return null;
        }

        if (getFilter().length() < 1) {
            String filter = new String("");
            if ((getUserName().length() > 1) && (getUserID().length() > 1)) {
                filter = "(|(cn="+getUserName()+")(uid="+getUserID()+"))";
            } else if (getUserName().length() > 1) {
                filter = "cn="+getUserName();
            } else if (getUserID().length() > 1) {

```

```
        filter = "uid="+getUserID();
    }
    setFilter(filter);
}

String[] res = null;
LDAPConnection m_ldc = new LDAPConnection();
try {
    try {
        printDebug("Connecting to " + getHost() +
            " " + getPort());
        connect( m_ldc, getHost(), getPort());
    } catch (Exception e) {
        printDebug( "Failed to connect to " + getHost() + ": " +
            e.toString() );
        setErrorCode( CONNECT_ERROR );
        notifyResult( (String)null );
        m_ldc = null;
        throw( new Exception() );
    }

    // Authenticate?
    if ( (!getAuthDN().equals("")) &&
        (!getAuthPassword().equals("")) ) {

        printDebug( "Authenticating " + getAuthDN() );
        try {
            m_ldc.authenticate( getAuthDN(), getAuthPassword() );
        } catch (Exception e) {
            printDebug( "Failed to authenticate: " + e.toString() );
            setErrorCode( AUTHENTICATION_ERROR );
            notifyResult( (String)null );
            throw( new Exception() );
        }
    }

    // Search
    try {
        printDebug("Searching " + getBase() +
            " for " + getFilter() + ", scope = " + getScope());
        String[] attrs = null;
        LDAPSearchResults results = m_ldc.search( getBase(),
            getScope(),
            getFilter(),
```

```
        attrs,
        false);

// Create a vector for the results
Vector v = new Vector();
LDAPEntry entry = null;
while ( results.hasMoreElements() ) {
    try {
        entry = (LDAPEntry)results.next();
    } catch (LDAPReferralException e) {
        if (getDebug()) {
            notifyResult("Referral URLs: ");
            LDAPUrl refUrls[] = e.getURLs();
            for (int i = 0; i < refUrls.length; i++)
                notifyResult(refUrls[i].getUrl());
        }
        continue;
    } catch (LDAPException e) {
        if (getDebug())
            notifyResult(e.toString());
        continue;
    }
    String dn = entry.getDN();
    v.addElement( dn );
    printDebug( "... " + dn );
}
// Pull out the DNs and create a string array
if ( v.size() > 0 ) {
    res = new String[v.size()];
    for( int i = 0; i < v.size(); i++ )
        res[i] = (String)v.elementAt( i );
    setErrorCode( OK );
} else {
    printDebug( "No entries found for " + getFilter() );
    setErrorCode( PROPERTY_NOT_FOUND );
}
} catch (Exception e) {
    printDebug( "Failed to search for " + getFilter() + ": " +
        e.toString() );
    setErrorCode( PROPERTY_NOT_FOUND );
}
} catch (Exception e) {
}
}
```

```

try {
    if ( (m_ldc != null) && m_ldc.isConnected() )
        m_ldc.disconnect();
} catch ( Exception e ) {
}

notifyResult( res );
return res;
}

```

You can verify the functionality of the Bean from the command line as follows:

```

java netscape.ldap.beans.LDAPGetEntries localhost 389 "o=airius.com" sub
"cn=:"
    ou=Directory Administrators, o=mcom.com
    cn=Accounting Managers,ou=groups,o=mcom.com
    cn=HR Managers,ou=groups,o=mcom.com
    cn=QA Managers,ou=groups,o=mcom.com
    cn=PD Managers,ou=groups,o=mcom.com

```

The following is a simple example of using the LDAPGetEntries Bean in a Web page.

```

<FORM NAME=input>
<TABLE WIDTH="300" >
  <TR>
    <TD>Host:</TD>
    <TD><INPUT TYPE=text NAME="host" VALUE="manta" SIZE=40></TD>
  </TR>
  <TR>
    <TD>Port:</TD>
    <TD><INPUT TYPE=text NAME="port" VALUE=389 SIZE=40></TD>
  </TR>
  <TR>
    <TD>Directory base:</TD>
    <TD><INPUT TYPE=text NAME="base" VALUE="o=Airius.com" SIZE=40></TD>
  </TR>
  <TR>
    <TD>Filter:</TD>
    <TD><INPUT TYPE=text NAME="filter" VALUE="objectclass=groupOfUniqueNames"
    SIZE=40></TD>
  </TR>

```

```
</TABLE>
<P><INPUT TYPE=button VALUE="Get entries"
  onClick="getEntries()" ARCHIVE="LDAPGetEntries.jar" ID="2">
</FORM>

<P><FORM NAME=output>

<TEXTAREA NAME="results" ROWS=10 COLS=70></TEXTAREA>

</FORM>

<SCRIPT LANGUAGE="JavaScript" ARCHIVE="LDAPGetEntries.jar" ID="3">
function showError(err) {
  var pkg = netscape.ldap.beans.LDAPGetEntries;
  if ( err == pkg.INVALID_PARAMETER )
    errString = "Invalid parameter";
  else if ( err == pkg.CONNECT_ERROR )
    errString = "Unable to connect to server";
  else if ( err == pkg.AUTHENTICATION_ERROR )
    errString = "Unable to authenticate to server";
  else if ( err == pkg.PROPERTY_NOT_FOUND )
    errString = "Entry or property not found";
  else
    errString = "Unexpected error " + err;

  alert( "Error fetching entries: " + errString );
}

var getter;
function getEntries() {
  if ( document.input.filter.value.length < 1 ) {
    alert( "Must enter a value for Filter" );
  }
  var getter = new netscape.ldap.beans.LDAPGetEntries();
  // Get parameters from form fields
  getter.setHost( document.input.host.value );
  getter.setPort( parseInt(document.input.port.value) );
  getter.setBase( document.input.base.value );
  getter.setFilter( document.input.filter.value );
  // Do the search
  netscape.security.PrivilegeManager.enablePrivilege("UniversalConnect");
  values = getter.getEntries();
  // Display the results, converted to a single string with line feeds
  if ( values != null ) {
    document.output.results.value=getter.convertToString( values );
```

```

    } else {
        var err = getter.getErrorCode();
        showError( err );
    }
}
</SCRIPT>

```

A search for “objectclass=person” in the sample Airius database produces results as in Figure 10-4.

Directory-Based Authentication in JavaScript

LDAPGetEntries and LDAPSimpleAuth can be hooked up to provide simple authentication using a user ID and password. Recall that LDAPSimpleAuth requires a full DN, which users generally do not know (or at least they find difficult to type in). We’ll use

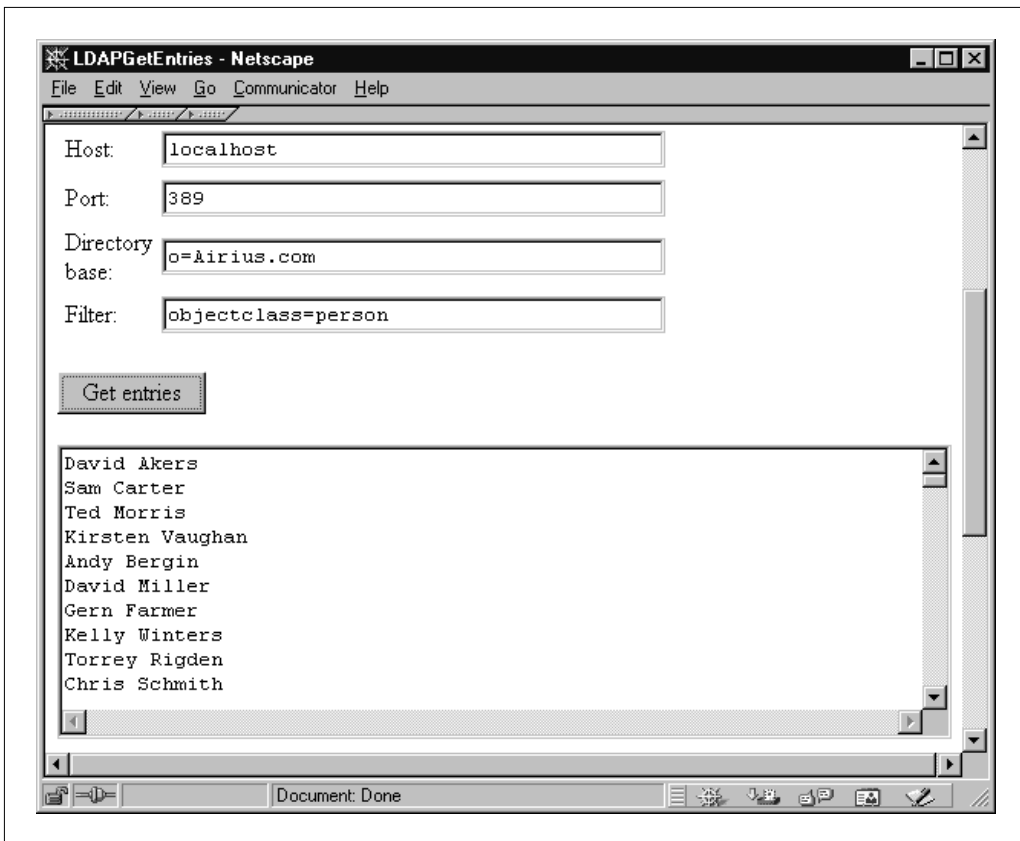


FIGURE 10-4. LDAPGetEntries in a simple Web page.

LDAPGetEntries to find the DN corresponding to the user ID entered by a user, and then LDAPSImpleAuth to validate the password supplied by the user for the DN.

The HTML page `login.html` demonstrates use of the two Beans to validate a user. The validated DN is available in the variable `userDN`:

```
<SCRIPT>
var userDN = "";
function doLogin() {
    // Create an instance of the LDAPGetEntries Bean
    var getter = new netscape.ldap.beans.LDAPGetEntries();
    // Get parameters from form fields
    getter.setHost( document.input.host.value );
    getter.setPort( parseInt(document.input.port.value) );
    getter.setBase( document.input.base.value );
    getter.setFilter( "uid="+document.input.userid.value );
    // Must request rights to do network connections
    netscape.security.PrivilegeManager.enablePrivilege("UniversalConnect");
    // And for property reads, to get LDAP error strings
    netscape.security.PrivilegeManager.enablePrivilege(
        "UniversalPropertyRead");
    // Do the search
    values = getter.getEntries();
    var result;
    // No matching entries?
    if ( (values == null) || (values.length < 1) ) {
        result = getter.getErrorCode();
        if ( result == 0 ) {
            result = getter.NO_SUCH_OBJECT;
        }
    }
    // Too many matching entries?
    } else if ( values.length > 1 ) {
        result = TOO_MANY_MATCHES;
    }
    // Good - just one match
    } else {
        userDN = values[0];
        auth = new Packages.netscape.ldap.beans.LDAPSImpleAuth();
        auth.setHost( document.input.host.value );
        auth.setPort( parseInt(document.input.port.value) );
        auth.setAuthDN( userDN );
        auth.setAuthPassword( document.input.password.value );
        result = auth.authenticate();
    }
    showResult( result );
}
</SCRIPT>
```

Using PropertyChangeEvent Notifications

So far in this chapter our examples have used the LDAP JavaBeans in a synchronous manner—calling a method and receiving the results as the return value of the method. However, JavaBeans are often connected to each other and to a client through event notification, which can be implemented very simply. In the following `TestBean` example, an anonymous object is created inline. This object is set up to be notified when results are available. When the property change event fires, the object prints out the results.

```
public class TestBean {
    public static void main( String[] args ) {
        // Create an instance of the Bean
        LDAPGetEntries getter = new LDAPGetEntries();
        // Create an object that listens for results and prints
        // them out
        getter.addPropertyChangeListener( new PropertyChangeListener() {
            public void propertyChange(PropertyChangeEvent evt) {
                String[] results = (String[])evt.getNewValue();
                for( int i = 0; i < results.length; i++ ) {
                    System.out.println( results[i] );
                }
            }
        } );

        // Do the search
        getter.getEntries( HOST, PORT, BASE, SCOPE, FILTER );
    }

    private static final String HOST = "localhost";
    private static final int PORT = 389;
    private static final String BASE = "o=airius.com";
    private static final int SCOPE = LDAPConnection.SCOPE_SUB;
    private static final String FILTER = "objectclass=groupofuniquenames";
}
```

A more useful and common scenario for JavaBeans is a visual development environment, in which components are combined and connected with a layout tool and no explicit programming is required. `TestBeanApplet` is an applet that contains only a `TextArea`. It has methods to set the font and background and foreground colors either through property accessors or through `PARAM` declarations in the `APPLET` tag, but these methods are omitted here for brevity.

The page `TestBeanApplet.html` demonstrates connecting the `LDAPGetEntries` JavaBean with a `TestBeanApplet` object through property change events. You can use

a visual development environment like Visual JavaScript to lay out, connect, and configure the two components without having to write any code.

`TestBeanApplet.java` is very simple, if we set aside for now the parsing and processing of color and font specifications:

```
public class TestBeanApplet extends Applet
    implements PropertyChangeListener {
    public void init() {
        super.init();
        setLayout(null);
        addNotify();
        int w = Integer.parseInt(getParameter("width"));
        int h = Integer.parseInt(getParameter("height"));
        resize(w+10,h+10);
        textField1 = new java.awt.TextArea(4, 40);
        textField1.setBounds(0,0,w,h);
        parseParameters();
        add(textField1);
    }
    private java.awt.TextArea textField1;
}
```

`TestBeanApplet.html`, shown below, is almost the same as `LDAPGetEntries.html`. Instead of a text area for displaying results, a `TestBeanApplet` is placed on the page and added as a `PropertyChangeListener` to the `LDAPGetEntries` JavaBean.

```
<APPLET code="TestBeanApplet.class" NAME="TestBeanApplet"
    MAYSCRIPT="true" width=450 height=200>
</APPLET>
var getter;
function getEntries() {
    // Get parameters from form fields
    getter.setHost( document.input.host.value );
    getter.setPort( parseInt(document.input.port.value) );
    getter.setBase( document.input.base.value );
    getter.setFilter( document.input.filter.value );
    getter.setAttribute( "cn" );
    // Must request rights to do network connections
    netscape.security.PrivilegeManager.enablePrivilege(
        "UniversalConnect");
    // Do the search
    values = getter.getEntries();
}
```

```

    if ( values == null ) {
        var err = getter.getErrorCode();
        showError( err );
    }
}

// Instantiate the Bean and hook it up to the applet
function doWire() {
    // Create an instance of the Bean
    getter = new netscape.ldap.beans.LDAPGetEntries();
    // Hook it up to the applet
    getter.addPropertyChangeListener( document.TestBeanApplet );
    // Set some interesting colors and font for the applet
    document.TestBeanApplet.setBackgroundColor( "yellow" );
    document.TestBeanApplet.setForegroundColor( "blue" );
    document.TestBeanApplet.setFont( "Helvetica-bolditalic-20" );
}
window.onload=doWire()

```

The Java TextArea can be easily customized with colors and fonts, as in Figure 10-5.

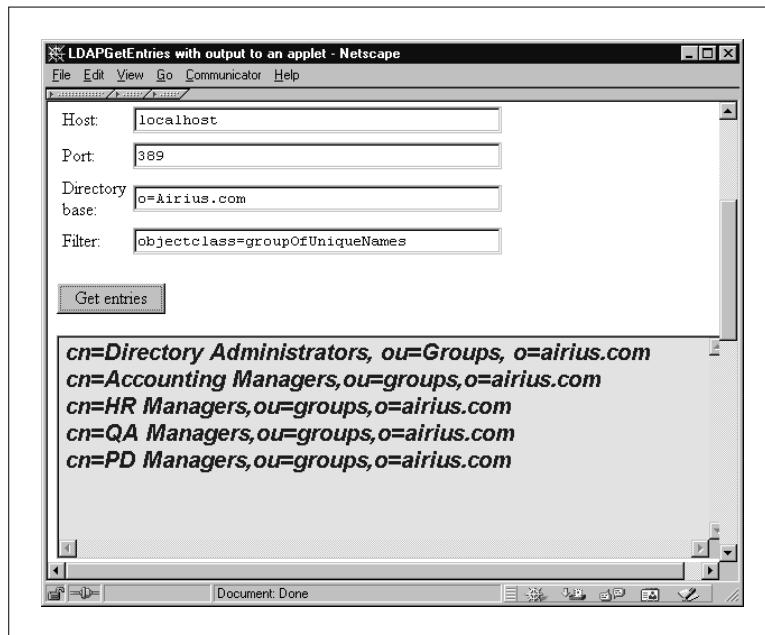


FIGURE 10-5. LDAPGetEntries with JavaBean text area.

Graphical LDAP JavaBeans

Now that we've investigated typical properties of an LDAP JavaBean, we can take a look at graphical components that can be plugged into a graphical user interface (GUI).

A Directory Browser

A component that displays the contents of a directory as a tree can be useful in many ways. It could be used simply to explore the directory. It could be invoked as a pop-up in places in an application where a user must enter a DN. Later in this chapter we'll hook it up to a table component to make a simple directory explorer, similar to Windows Explorer.

To simplify use of the Bean in a JFC (Java Foundation Classes) environment, we'll have it extend `javax.swing.JPanel`.

The `TreePanel` Bean (Figure 10-6) uses several utility classes, which are presented in Appendix C:

- **Debug** for conditional printout of error statements
- **DateUtil** for converting LDAP date strings to a localized date format

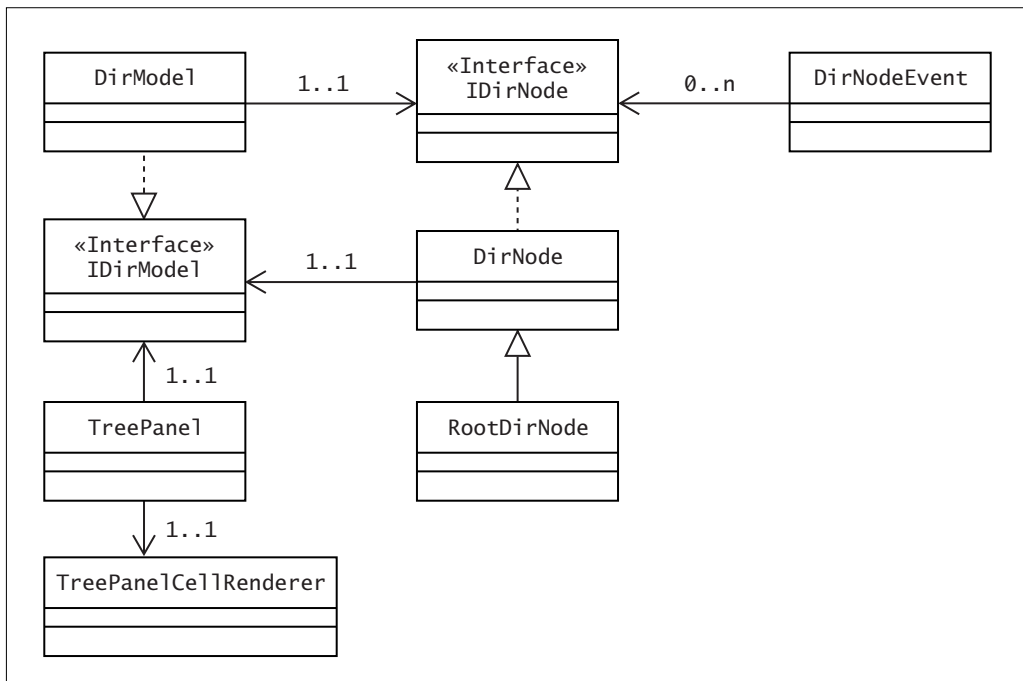


FIGURE 10-6. `TreePanel` and supporting classes.

- **DirUtil** to simplify connecting to a server (optionally using SSL)
- **ImageUtil** for remote or local loading of image files
- **ResourceSet** for reading properties from properties files
- **SortUtil** for sorting

TreePanel is supplied with data from an object that implements `IDirModel`. The level of indirection provided by the `IDirModel` interface allows us to substitute other implementations in the future, and it clarifies the interfaces used.

`IDirModel`

`IDirModel` (Figure 10-7) extends `TreeModel`, which is the basic JFC model interface for supporting a `JTree`. It adds methods for getting and setting various additional properties related to how the directory is to be searched.

```
public interface IDirModel extends javax.swing.tree.TreeModel {
    /**
     * Get a connection to the directory instance
     *
     * @return a connection to the server
     */
    public LDAPConnection getLDAPConnection();
}
```

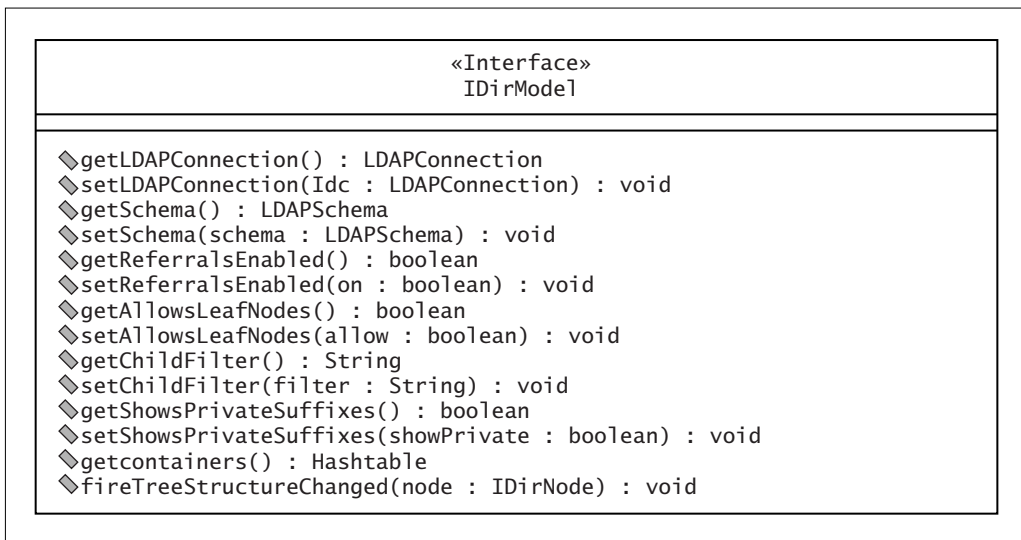


FIGURE 10-7. `IDirModel`.

```
/**
 * Sets the server connection used to populate the tree
 *
 * @param ldc the server connection used to populate the tree
 */
public void setLDAPConnection( LDAPConnection ldc );

/**
 * Get the schema of the directory instance
 *
 * @return a reference to a schema object
 */
public LDAPSchema getSchema();

/**
 * Sets a reference to the schema of the directoryinstance
 *
 * @param schema a reference to a schema object
 */
public void setSchema( LDAPSchema schema );

/**
 * Get the parameter that determines if the
 * ManagedSAIT control is sent with each search. If the
 * control is sent, referral entries are returned as
 * normal entries and not followed.
 *
 * @return true if referrals are to be followed
 */
public boolean getReferralsEnabled();

/**
 * Set a parameter for future searches that determines if the
 * ManagedSAIT control is sent with each search. If referrals are
 * disabled, the control is sent and you will receive the referring
 * entry back.
 *
 * @param on true (the default) if referrals are to be followed
 */
public void setReferralsEnabled( boolean on );

/**
 * Reports if the model is currently configured to show leaf (as well as
 * container) nodes
```

```
*
* @return true if the model is currently configured to show leaf
* (as well as container) nodes
*/
public boolean getAllowsLeafNodes();

/**
* Determines if the model is to show leaf (as well as container) nodes
*
* @param allow true if the model is to show leaf (as well as container)
* nodes
*/
public void setAllowsLeafNodes( boolean allow );

/**
* Used between DirNode and DirModel, to manage the search filter used to
* find children of a node
*
* @return the search filter to be used to find direct children
*/
public String getChildFilter();

/**
* Set the search filter used to find children of a node
*
* @param filter the search filter to be used to find direct children
*/
public void setChildFilter( String filter );

/**
* Report if the model will show private suffixes.
* If true (the default), private suffixes will appear.
*
* @return true if private suffixes are to be displayed in the tree
*/
public boolean getShowsPrivateSuffixes();

/**
* Determines if the model will supply node objects for tree nodes.
* If false (the default), only container nodes will appear.
*
* @param allow true if leaf nodes are to be displayed in the tree
*/
public void setShowsPrivateSuffixes( boolean showPrivate );
```

```
/**
 * Used between DirNode and DirModel, to manage the list of object classes
 * that are to be considered containers
 *
 * @return a hash table containing object classes to be considered
 * containers
 */
public Hashtable getContainers();

/**
 * Informs the tree that a particular node's structure
 * has changed and its view needs to be updated
 *
 * @param node the node that has changed
 */
public void fireTreeStructureChanged( IDirNode node );
}
```

TreePanel

TreePanel (Figure 10-8) catches and handles mouse, key, and tree selection events on the JTree it manages:

```
public class TreePanel extends JPanel
    implements TreeSelectionListener,
               MouseListener,
               KeyListener,
               Serializable {
```

TreePanel uses a special node renderer—TreePanelCellRenderer—to work around bugs in the JFC default implementation and to provide better selection indication. There is a default constructor as well, for deserialization:

```
/**
 * Construct tree using the data model specified
 *
 * @param model a directory model
 */
public TreePanel( IDirModel model ) {
    super();
    _treeRenderer = new TreePanelCellRenderer();
    _model = model;
    setLayout(new BorderLayout());
    setBorder(new EmptyBorder(0, 0, 0, 0));
}
```

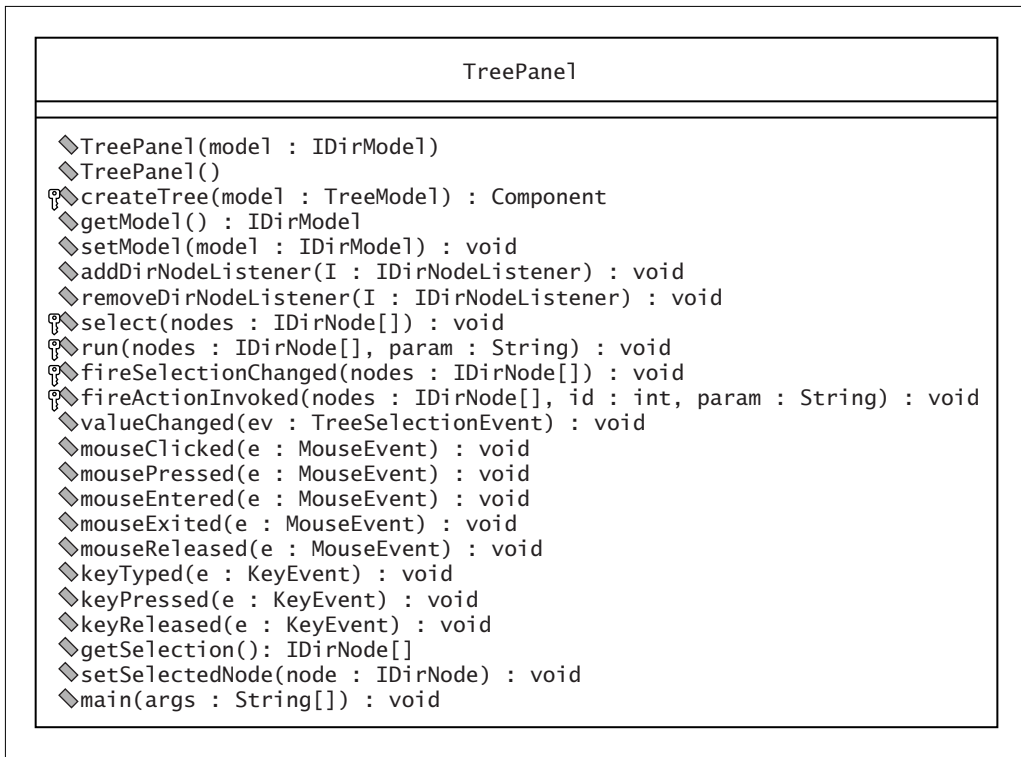


FIGURE 10-8. TreePanel.

```

        Component tree = createTree( model );
        add( tree );
    }

    /**
     * Default constructor for deserialization
     */
    public TreePanel() {
        this( null );
    }

```

The JTree is created and configured:

```

    /**
     * Returns a Component that contains the tree
     *

```

```
* @return a Component that contains the tree
*/
protected Component createTree( TreeModel model ) {
    _tree = new JTree(model);
    // Lines between nodes, as in Windows
    _tree.putClientProperty( "JTree.lineStyle", "Angled" );
    // For now, single selection only
    _tree.getSelectionModel().setSelectionMode(
        TreeSelectionMode.SINGLE_TREE_SELECTION );
    _tree.addFocusListener(new FocusListener() {
        // This causes ALL tree nodes to repaint, which
        // is needed to change colors for selected tree nodes
        public void focusGained(FocusEvent e) {
            JTree tree = (JTree)e.getSource();
            tree.validate();
            tree.repaint();
        }
        public void focusLost(FocusEvent e) {
            JTree tree = (JTree)e.getSource();
            tree.validate();
            tree.repaint();
        }
    });
    // Special cell renderer
    _tree.setCellRenderer(_treeRenderer);
    // Catch all events
    _tree.addTreeSelectionListener(this);
    _tree.addMouseListener(this);
    _tree.addKeyListener(this);

    _treePanel = new JScrollPane();
    _treePanel.getViewport().add(_tree);
    _treePanel.setBorder( new BevelBorder(BevelBorder.LOWERED,
        UIManager.getColor("controlHighlight"),
        UIManager.getColor("control"),
        UIManager.getColor("controlDkShadow"),
        UIManager.getColor("controlShadow")));
    _treePanel.setPreferredSize(new Dimension(200, 200));
    _treePanel.setMinimumSize(new Dimension(1, 1));
    return _treePanel;
}
```

In some cases a client will want to access the model directly:

```

/**
 * Return tree model
 *
 * @return the tree model
 */
public IDirModel getModel() {
    return _model;
}

/**
 * Set the model
 *
 * @param model the model
 */
public void setModel( IDirModel model ) {
    _model = model;
}

```

TreePanel dispatches events to objects implementing the IDirNodeListener interface that have registered an interest:

```

/**
 * Adds a listener that is interested in receiving
 * DirNodeListener events
 *
 * @param l an object interested in receiving DirNodeListener
 * events
 */
public void addDirNodeListener( IDirNodeListener l ) {
    _listenerList.add( IDirNodeListener.class, l );
}

/**
 * Removes a listener that is interested in receiving
 * DirNodeListener events
 *
 * @param l an object interested in receiving DirNodeListener
 * events
 */
public void removeDirNodeListener( IDirNodeListener l ) {
    _listenerList.remove( IDirNodeListener.class, l );
}

```

```
/**
 * Dispatch selection events to listeners
 *
 * @param nodes currently selected nodes
 */
protected void select( IDirNode[] nodes ) {
    fireSelectionChanged( nodes );
}

/**
 * Dispatch "run" event to listeners
 *
 * @param nodes currently selected nodes
 * @param param optional additional event info
 */
protected void run( IDirNode[] nodes, String param ) {
    fireActionInvoked( nodes, DirNodeEvent.RUN, param );
}

/**
 * Dispatch selection events to listeners, using the
 * EventListenerList
 *
 * @param nodes currently selected nodes
 */
protected void fireSelectionChanged( IDirNode[] nodes ) {
    // Guaranteed to return a non-null array
    Object[] listeners = _listenerList.getListenerList();
    // Process the listeners last to first, notifying
    // those that are interested in this event
    for ( int i = listeners.length - 2; i >= 0; i -= 2 ) {
        if ( listeners[i] == IDirNodeListener.class ) {
            IDirNodeListener l = (IDirNodeListener)listeners[i + 1];
            l.selectionChanged( nodes );
        }
    }
}

/**
 * Dispatch events to listeners, using the
 * EventListenerList
 *
 * @param nodes currently selected nodes
```

```

    * @param id identifier of the type of event
    * @param param optional additional event info
    */
protected void fireActionInvoked( IDirNode[] nodes,
                                   int id,
                                   String param ) {
    // Guaranteed to return a non-null array
    Object[] listeners = _listenerList.getListenerList();
    DirNodeEvent e = null;
    // Process the listeners last to first, notifying
    // those that are interested in this event
    for ( int i = listeners.length - 2; i >= 0; i -= 2 ) {
        if ( listeners[i] == IDirNodeListener.class ) {
            // Lazily create the event:
            if ( e == null )
                e = new DirNodeEvent( nodes, id, param );
            IDirNodeListener l = (IDirNodeListener)listeners[i + 1];
            l.actionInvoked( e );
        }
    }
}

```

Then we have the implementations of the mouse, key, and tree selection interfaces. On tree selection events, clients of this object are notified of selection events. On mouse events with double-click or on key events in which the Enter key is pressed, clients are notified of a RUN event:

```

/**
 * Implements TreeSelectionListener. Called when an object is selected
 * in the tree.
 *
 * @param ev event provided by JTree
 */
public void valueChanged(TreeSelectionEvent ev) {
    Debug.println( 9, "Tree.valueChanged: " +
                  ev.getPath().getLastPathComponent() );
    IDirNode[] selection = getSelection();
    if ( selection != null ) {
        Debug.println( 9, "Tree.valueChanged: selection = " +
                      selection[0] );
        select( selection );
    }
}

```

```
/**
 * Implements MouseListener. Called when a mouse button is pressed
 * and released in the tree.
 *
 * @param e mouse event
 */
public void mouseClicked(MouseEvent e) {
    IDirNode[] selection = getSelection();
    if (selection != null) {
        if (e.getClickCount() == 2) { // double click
            run( selection, "" );
        }
    }
}

/**
 * Implements MouseListener. Called when a mouse button is pressed
 * in the tree.
 *
 * @param e mouse event
 */
public void mousePressed(MouseEvent e) {
}

/**
 * Implements MouseListener
 *
 * @param e mouse event
 */
public void mouseEntered(MouseEvent e) {
}

/**
 * Implements MouseListener
 *
 * @param e mouse event
 */
public void mouseExited(MouseEvent e) {
}

/**
 * Implements MouseListener. Called when a mouse button is released
 * in the tree.
 *
```

```

    * @param e mouse event
    */
    public void mouseReleased(MouseEvent e) {
    }

    /**
     * Implements KeyListener
     *
     * @param e key event
     */
    public void keyTyped(KeyEvent e) {
    }

    /**
     * Implements KeyListener. Called when a key is pressed.
     *
     * @param e key event
     */
    public void keyPressed(KeyEvent e) {
        if (e.getKeyCode() == KeyEvent.VK_ENTER) {
            IDirNode[] selection = getSelection();
            if (selection != null) {
                run( selection, "" );
            }
        }
    }

    /**
     * Implements KeyListener. Called when a key is released.
     *
     * @param e key event
     */
    public void keyReleased(KeyEvent e) {
    }

```

Finally, there are two utility methods—`getSelection` and `setSelectedNode`—that other components can use to force selection of a particular node or to find out which nodes are selected at any time:

```

/**
 * Returns array of selected nodes
 *
 * @param return array of selected nodes

```

```
    */
    public IDirNode[] getSelection() {
        IDirNode[] selection = null;
        TreePath path[] = _tree.getSelectionPaths();
        if ((path != null) && (path.length > 0)) {
            selection = new IDirNode[path.length];
            for (int index = 0; index < path.length; index++) {
                selection[index] =
                    (IDirNode)path[index].getLastPathComponent();
            }
        }
        return selection;
    }

    /**
     * Make a node selected and visible
     *
     * @param node node to make visible
     */
    public void setSelectedNode( IDirNode node ) {
        if ( node != null ) {
            TreePath path =
                new TreePath(
                    ((DefaultMutableTreeNode)node).getPath() );
            _tree.expandPath( path );
            _tree.makeVisible( path );
            _tree.scrollPathToVisible( path );
            _tree.repaint();
            _tree.setSelectionPath( path );
        }
    }
}
```

The member variables of `TreePanel` are declared as follows:

```
protected IDirModel _model;
protected JTree _tree = null;
protected TreeCellRenderer _treeRenderer;
protected JScrollPane _treePanel;
// Use an EventListenerList to manage different types of
// listeners
protected EventListenerList _listenerList =
    new EventListenerList();
```

DirModel

DirModel (Figure 10-9) implements the IDirModel interface to provide data for the JTree inside TreePanel. It also implements IDirContentListener so that it can be notified if the directory has changed and the model must be reinitialized.

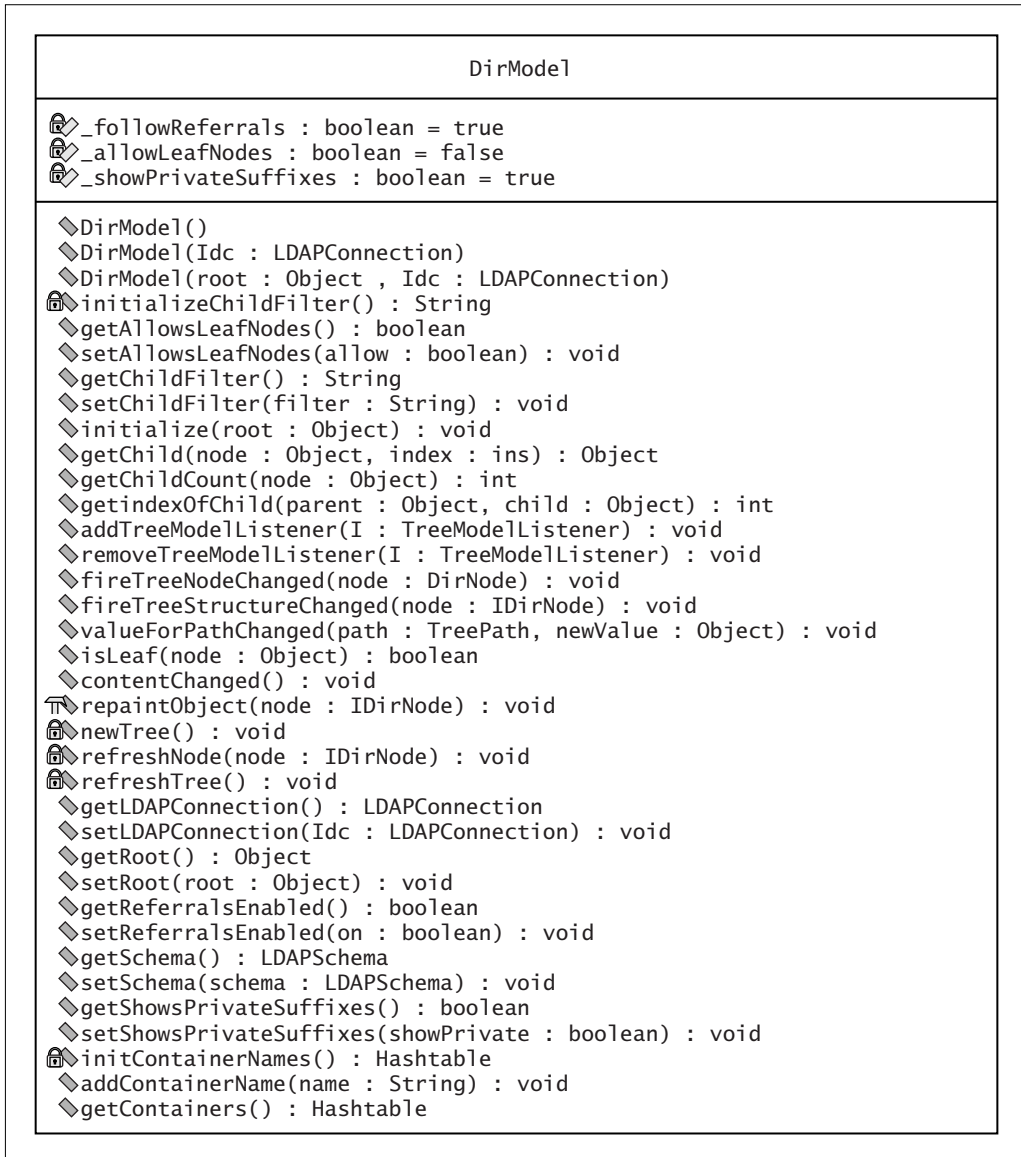


FIGURE 10-9. DirModel.

```
public class DirModel implements IDirModel,
                                IDirContentListener,
                                Serializable {

    /**
     * Default constructor for deserialization
     */
    public DirModel() {
        setChildFilter( initializeChildFilter() );
    }

    /**
     * Constructor of the model that doesn't populate the tree. You must
     * call initialize() to populate it.
     *
     * @param ldc connection to LDAP server
     */
    public DirModel( LDAPConnection ldc ) {
        this();
        setLDAPConnection( ldc );
    }

    /**
     * Constructor of the model with the root object passed in.
     * Suffix nodes are retrieved and added to the tree.
     *
     * @param root root object
     * @param ldc connection to LDAP server
     */
    public DirModel( Object root, LDAPConnection ldc ) {
        this( ldc );
        initialize( root );
    }
}
```

The child filter is the search filter used to find children of a node. It is initialized to search for all entries that contain any object classes considered to be containers, as well as any entries that have children of their own. There are methods to switch between this mode and the mode in which all children are searched:

```
/**
 * Set default filter, which causes only container nodes to be
 * displayed in the tree
 */
```

```

private String initializeChildFilter() {
    Hashtable containers = getContainers();
    /* "numSubordinates>=1" is not indexed, but
       "&(numSubordinates=*)(numSubordinates>=1)" is, in DS 4.0
    */
    String filter = "(|(&(numSubordinates=*)(numSubordinates>=1)))";
    Enumeration e = containers.keys();
    while( e.hasMoreElements() )
        filter += "(objectclass=" + (String)e.nextElement() + ")";
    filter += ")";
    Debug.println( "DirModel.initializeChildFilter: " +
                   filter );
    return filter;
}

/**
 * Report if the model will supply node objects for tree nodes.
 * If false (the default), only container nodes will appear.
 *
 * @return true if leaf nodes are to be displayed in the tree
 */
public boolean getAllowsLeafNodes() {
    return _allowLeafNodes;
}

/**
 * Determines if the model will supply node objects for tree nodes.
 * If false (the default), only container nodes will appear.
 *
 * @param allow true if leaf nodes are to be displayed in the tree
 */
public void setAllowsLeafNodes( boolean allow ) {
    if ( allow ) {
        setChildFilter( "objectclass=*" );
    } else {
        setChildFilter( initializeChildFilter() );
    }
    _allowLeafNodes = allow;
    contentChanged();
}

/**
 * Used between DirNode and DirModel, to manage the search
 * filter used to find children of a node

```

```
*
 * @return the search filter to be used to find direct children
 */
public String getChildFilter() {
    return _childFilter;
}

/**
 * Set the search filter used to find children of a node
 *
 * @param filter the search filter to be used to find
 * direct children
 */
public void setChildFilter( String filter ) {
    _childFilter = filter;
}
```

The `initialize` method creates the root node, which fetches all public and optionally all private suffixes and adds them as children. It is necessary to explicitly enumerate the suffixes because the root entry in LDAP has no children. Extra work is required to figure out what the public and private suffixes of the directory are, in order to retrieve them and present them as children.

```
/**
 * Create root node and a node for each suffix
 *
 * @param root root node for the tree. If null, a new root node
 * will be created and the root entry will be searched for suffixes.
 */
public void initialize( Object root ) {
    if ( root == null ) {
        root = new RootDirNode( this, getShowsPrivateSuffixes() );
        Debug.println(9, "DirModel.initialize: new root");
    } else {
        ((DirNode)root).setModel( this );
        Debug.println(9, "DirModel.initialize: old root=" +
            root);
    }
    setRoot( root );
}
```

Many methods are called by—or call back to—the `JTree` object. Most of these methods delegate the responsibility to the `DirNode` involved:

```
/**
 * Get a child node of a node
 *
 * @param node parent node
 * @param index position of the child
 * @return the child of the specified node
 */
public Object getChild(Object node, int index) {
    IDirNode sn = (IDirNode) node;
    return sn.getChildAt(index);
}

/**
 * Return the number of children
 *
 * @param node node to be checked
 * @return number of children of the specified node
 */
public int getChildCount(Object node) {
    IDirNode sn = (IDirNode) node;
    return sn.getChildCount();
}

/**
 * Returns the index of a particular child node
 * @param parent parent node
 * @param child child node
 * @return position of the child
 */
public int getIndexOfChild(Object parent, Object child) {
    return ((IDirNode) parent).getIndex(
        (IDirNode) child);
}

/**
 * Check whether the node is a leaf node or not
 *
 * @param node node to be checked
 * @return true if the node is leaf, false otherwise
 */
public boolean isLeaf( Object node ) {
    IDirNode sn = (IDirNode) node;
    return ( sn.isLeaf() );
}
```

The model provides the expected methods for managing listeners:

```
/**
 * Adds a listener that is interested in receiving TreeModelListener
 * events. Called by JTree.
 *
 * @param l an object interested in receiving TreeModelListener
 * events
 */
public void addTreeModelListener(TreeModelListener l) {
    _listenerList.add(TreeModelListener.class, l);
}

/**
 * Removes a listener that is interested in receiving
 * TreeModelListener events. Called by JTree.
 *
 * @param l an object interested in receiving TreeModelListener
 * events
 */
public void removeTreeModelListener(TreeModelListener l) {
    _listenerList.remove(TreeModelListener.class, l);
}

/**
 * Informs the tree that a particular node has changed
 *
 * @param node the node that changed
 * @see EventListenerList
 */
public void fireTreeNodeChanged( DirNode node ) {
    // Guaranteed to return a non-null array
    Object[] listeners = _listenerList.getListenerList();
    TreeModelEvent e = null;
    // Process the listeners last to first, notifying
    // those that are interested in this event
    for ( int i = listeners.length - 2; i >= 0; i -= 2 ) {
        if ( listeners[i] == TreeModelListener.class ) {
            // Lazily create the event:
            if ( e == null )
                e = new TreeModelEvent( this, node.getPath() );
            ((TreeModelListener)listeners[i + 1]).treeNodesChanged(e);
        }
    }
}
```

```

/**
 * Informs tree that a particular node's structure has changed
 * and its view needs to be updated.
 * @param node the node at the root of the changes
 * @see EventListenerList
 */
public void fireTreeStructureChanged( IDirNode node ) {
    // Guaranteed to return a non-null array
    Object[] listeners = _listenerList.getListenerList();
    TreeModelEvent e = null;
    // Process the listeners last to first, notifying
    // those that are interested in this event
    for ( int i = listeners.length - 2; i >= 0; i -= 2 ) {
        if ( listeners[i] == TreeModelListener.class ) {
            // Lazily create the event:
            if ( e == null )
                e = new TreeModelEvent(this, ((DirNode)node).getPath());
            TreeModelListener l = (TreeModelListener)listeners[i + 1];
            l.treeStructureChanged( e );
        }
    }
}

/**
 * Called when user has altered the value for the item identified
 * by path to newValue. Called by JTree.
 *
 * @param path path to the changed node
 * @param newValue new value of the node
 */
public void valueForPathChanged(TreePath path,
                                Object newValue) {
}

```

The method `contentChanged` is called when another component with which `DirModel` has registered an interest in changes in directory contents wishes to notify `DirModel`. A few helper methods cause regeneration of part or all of the tree:

```

/**
 * Called when the tree structure has changed radically; read a new
 * tree from the server
 */
public void contentChanged() {
    newTree();
}

```

```
void repaintObject( IDirNode node ) {
    Debug.println( "DirModel.repaintObject: " +
        node );
    fireTreeStructureChanged( (DirNode)node );
}

private void newTree() {
    DirNode root = new RootDirNode( this, "",
        getShowsPrivateSuffixes() );
    Debug.println(9, "DirModel.newTree: new root");
    setRoot( root );
    refreshTree();
}

private void refreshNode( IDirNode node ) {
    node.load();
    repaintObject( node );
}

private void refreshTree() {
    refreshNode( (IDirNode)getRoot() );
}
```

All nodes share the LDAP connection of the model, which they access through `getLDAPConnection`, as the following block of code shows. It is also possible to set the connection after object instantiation—for example, on deserialization.

```
/**
 * Returns the server connection used to populate the tree
 *
 * @return the server connection used to populate the tree
 */
public LDAPConnection getLDAPConnection() {
    return _ldc;
}

/**
 * Sets the server connection used to populate the tree
 *
 * @param ldc the server connection used to populate the tree
 */
public void setLDAPConnection( LDAPConnection ldc ) {
    _ldc = ldc;
}
```

There are several accessors for properties of the model:

```
/**
 * Returns root node of the tree.
 *
 * return Root node of the tree
 */
public Object getRoot() {
    return _root;
}

/**
 * Sets root node of the tree
 *
 * @param root root node for the tree
 */
public void setRoot(Object root) {
    _root = (IDirNode) root;
}

/**
 * Get the parameter that determines if the
 * ManageDsaIT control is sent with each search
 *
 * @returns true if referrals are to be followed
 */
public boolean getReferralsEnabled() {
    return _followReferrals;
}

/**
 * Set a parameter for future searches that determines if the
 * ManageDsaIT control is sent with each search. If referrals are
 * disabled, the control is sent and you will receive the referring
 * entry back.
 *
 * @param on true (the default) if referrals are to be followed
 */
public void setReferralsEnabled( boolean on ) {
    _followReferrals = on;
}

/**
 * Get the schema of the directory instance
```

```
*
* @return a reference to a schema object
*/
public LDAPSchema getSchema() {
    if ( _schema == null ) {
        _schema = new LDAPSchema();
        try {
            _schema.fetchSchema( getLDAPConnection() );
        } catch ( LDAPException e ) {
            Debug.println( "DirModel.getSchema: " + e );
            _schema = null;
        }
    }
    return _schema;
}

/**
 * Sets a reference to the schema of the directory instance
 *
 * @param schema a reference to a schema object
 */
public void setSchema( LDAPSchema schema ) {
    _schema = schema;
}

/**
 * Report if the model will show private suffixes.
 * If true (the default), private suffixes will appear.
 *
 * @return true if private suffixes are to be displayed in the tree
 */
public boolean getShowsPrivateSuffixes() {
    return _showPrivateSuffixes;
}

/**
 * Determines if the model will supply node objects for tree nodes.
 * If false (the default), only container nodes will appear.
 *
 * @param allow true if leaf nodes are to be displayed in the tree
 */
public void setShowsPrivateSuffixes( boolean showPrivate ) {
    _showPrivateSuffixes = showPrivate;
    contentChanged();
}
}
```

Some methods manage the definitions of the object classes that are to be considered containers:

```

/**
 * Get object classes that are to be considered containers
 * from a properties file
 *
 */
private static Hashtable initContainerNames() {
    Hashtable h = new Hashtable();
    String items = _resource.getString( _section, "containers" );
    Debug.println( "DirModel.initContainerNames" );
    if ( items != null ) {
        StringTokenizer st = new StringTokenizer( items, " " );
        int i = 0;
        while ( st.hasMoreTokens() ) {
            String name = st.nextToken().toLowerCase();
            Debug.println( " added container type " + name );
            h.put( name, name );
        }
    }
    return h;
}

/**
 * Add the name of an object class to be considered a container
 *
 * @param name name of an object class to be considered a container
 */
public void addContainerName( String name ) {
    _cContainers.put( name, name );
}

/**
 * Used between DirNode and DirModel, to manage the list of
 * object classes that are to be considered containers
 *
 * @return a hash table containing object classes to be
 * considered containers
 */
public Hashtable getContainers() {
    if ( _cContainers == null )
        _cContainers = initContainerNames();
    return _cContainers;
}

```

Finally, member variables are declared:

```
// Properties for this component (strings)
static ResourceSet _resource =
    new ResourceSet( "dirtree" );
// Section of the properties file to use
private static final String _section = "EntryObject";

// Active connection to directory
private LDAPConnection _ldc;
// Schema definitions
private LDAPSchema _schema = null;
// Control to use if referrals are not to be followed
private static LDAPControl _manageDSAITControl =
    new LDAPControl( LDAPControl.MANAGEDSAIT, true, null );
// Root node of the tree
private IDirNode _root = null;
private boolean _followReferrals = true;
private boolean _allowLeafNodes = false;
// List of possible container object classes
private Hashtable _cContainers = null;
// Filter string to search for immediate children
private String _childFilter;
// Helper object to manager event listeners
protected EventListenerList _listenerList =
    new EventListenerList();
// Set this to false to NOT show private suffixes
private boolean _showPrivateSuffixes = true;
```

IDirNode

DirModel delegates much of its work to DirNode (Figure 10-10), which implements the IDirNode interface (in addition to extending DefaultMutableTreeNode).

```
public interface IDirNode extends javax.swing.tree.TreeNode,
    javax.swing.tree.MutableTreeNode {

    /**
     * Specifies the name for this object, displayed in tree, right of icon
     *
     * @return a string representing the object's name
     */
    public String getName();
```

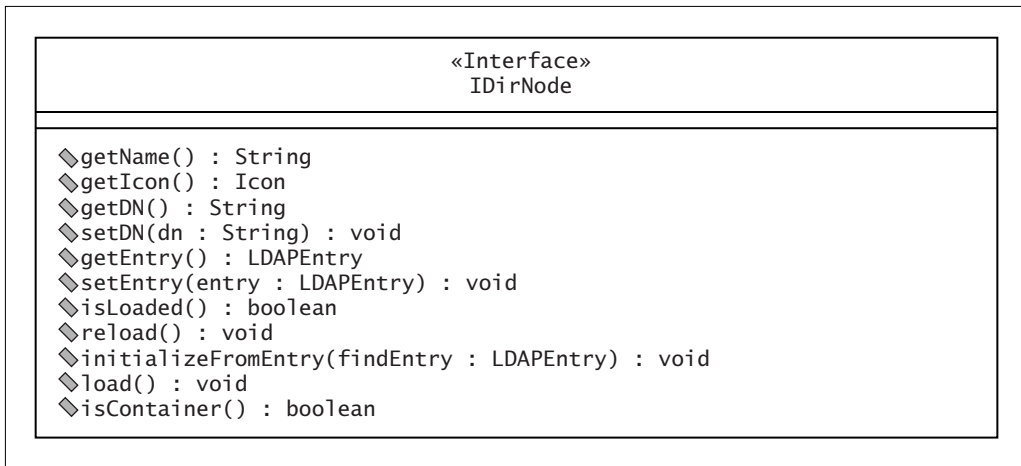


FIGURE 10-10. IDirNode.

```

/**
 * Specifies an icon for this object, displayed in tree, left of name.
 * The recommended size for this icon is 16_16 pixels.
 *
 * @return an icon representing the object's icon
 */
public Icon getIcon();

/**
 * Get the DN of the entry corresponding to this node
 *
 * @return the DN of the node
 */
public String getDN();

/**
 * Set the DN of the node
 *
 * @param dn the new DN of the node
 */
public void setDN( String dn );

/**
 * Report the entry associated with this node. If the entry has not been
  
```

```
* retrieved from the directory yet, it is done now.
*
* @return the entry associated with this node. Only a few attributes are
* retrieved in the entry.
*/
public LDAPEntry getEntry();

/**
 * Set the entry for this node
 *
 * @param entry the new entry. May be null to force reinitialization.
 */
public void setEntry( LDAPEntry entry );

/**
 * Returns true if the node has read its entry from the directory
 *
 * @return true if the node has read its entry from the directory
 */
public boolean isLoaded();

/**
 * Create all the one-level child nodes
 */
public void reload();

/**
 * Initialize the node from data in an entry
 *
 * @param entry an entry initialized with data
 */
public void initializeFromEntry( LDAPEntry findEntry );

/**
 * Check if there are children to this node
 */
public void load();

/**
 * Report if this node is considered a container. This is true if it is
 * one of a defined list of object classes, or if it has children.
 *
 * @return true if the node is considered a container
```

```

    */
    public boolean isContainer();
}

```

DirNode

`DirNode` (Figure 10-11) is a fairly large class. We won't go into its complete definition here, but we'll discuss the main things that distinguish it from a standard JFC `DefaultMutableTreeNode`.

`RootDirNode` (Figure 10-12) is derived from `DirNode`. The main difference is that the root node in a directory—called the root DSE (the special entry with the empty DN)—returns nothing if you do a one-level search on it. To get the public and private suffixes of the directory, you have to look elsewhere. `RootDirNode` overrides the method to get children of itself, and it is set up to retrieve the suffixes on instantiation.

`DirNode` maintains its own image and display label. Appropriate images for various object classes are defined in a properties file, and additional images can be added there. The properties file entries are as follows:

```

EntryObject-person-icon=alluser16n.gif
EntryObject-organization-icon=folder.gif
EntryObject-organizationalunit-icon=ou16.gif
EntryObject-groupofuniquenames-icon=allgroup16n.gif
EntryObject-default-icon=genobject.gif
EntryObject-default-folder-icon=folder.gif

```

When a node is initialized from a directory entry, an appropriate image and label are selected:

```

/**
 * Initialize the node from data in an entry
 *
 * @param entry an entry initialized with data
 */
public void initializeFromEntry( LDAPEntry entry ) {
    _fLoaded = ( entry.getAttribute( SUBORDINATE_ATTR ) != null );
    _objectClasses = checkObjectClasses( entry );
    _fContainer = checkIfContainer();
    setIcon( checkIcon( _objectClasses, !isContainer() ) );
    _sCn = checkCn( entry );
    if ( _sCn != null ) {
        setName( _sCn );
    }
}
}

```

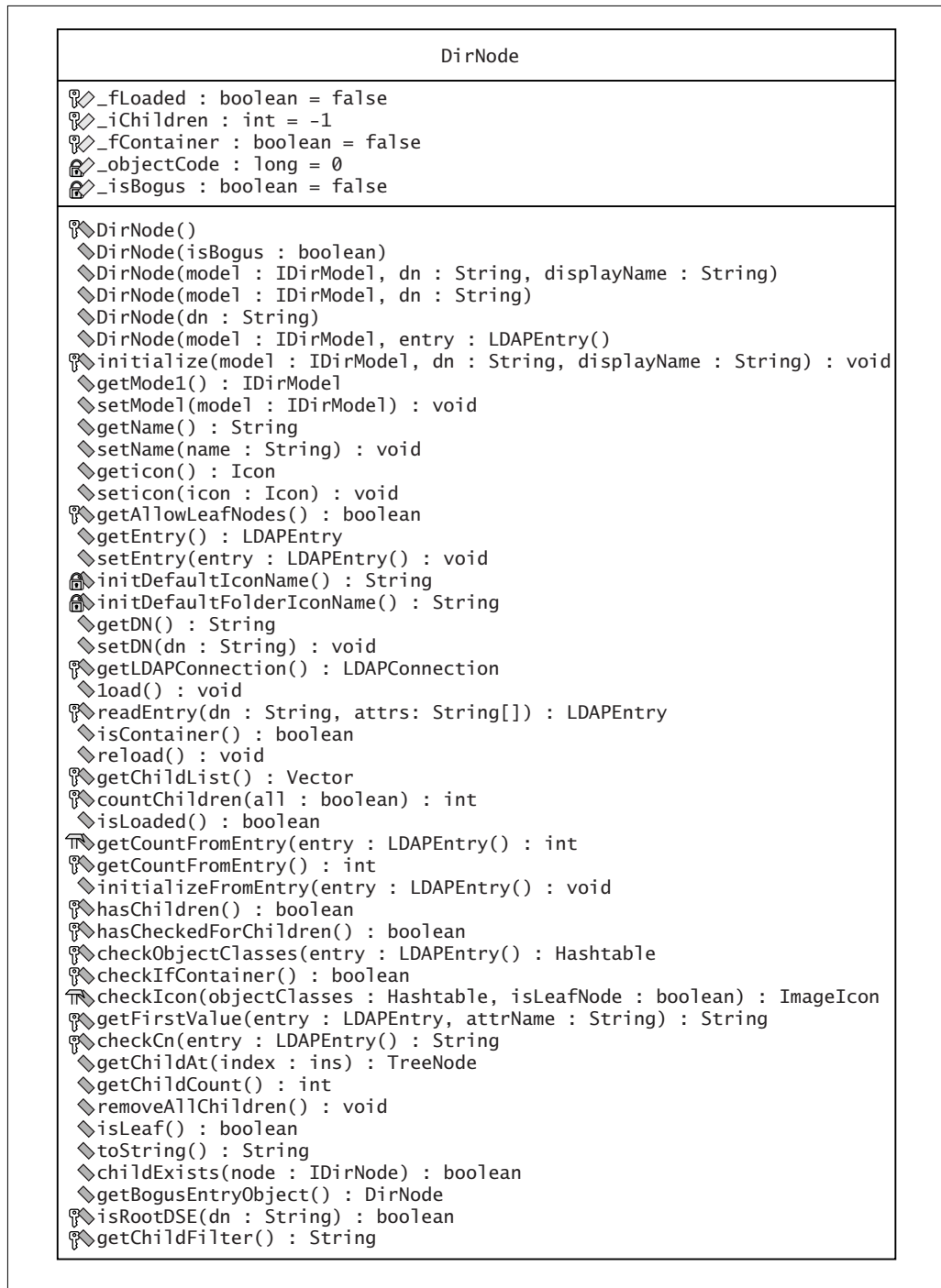


FIGURE 10-11. DirNode.

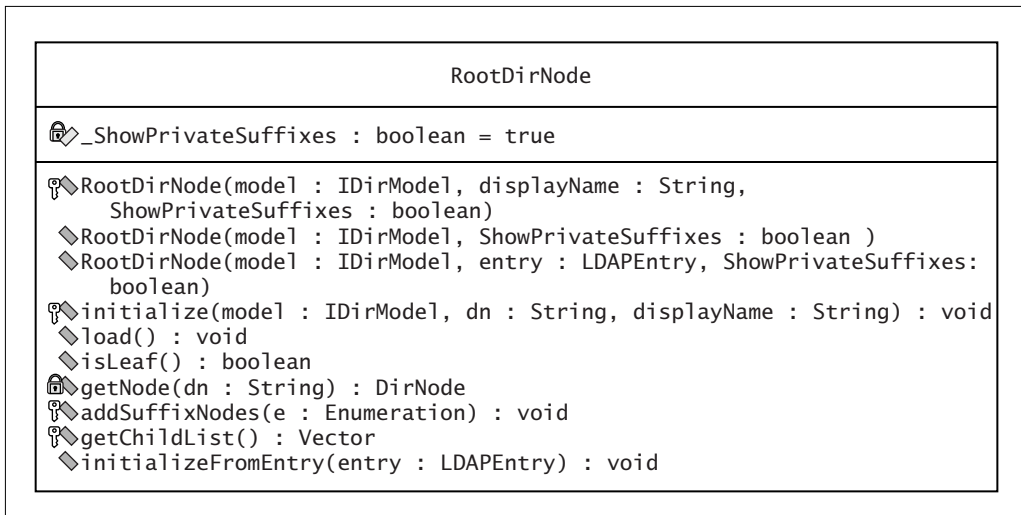


FIGURE 10-12. RootDirNode.

```

/**
 * Create hash table of object classes from the entry
 *
 * @param entry entry containing at least object classes
 * @return a hash table of the object classes
 */
protected Hashtable checkObjectClasses( LDAPEntry entry ) {
    if ( _objectClasses != null )
        return _objectClasses;
    Hashtable objectClasses = new Hashtable();
    LDAPAttribute attr = entry.getAttribute(
        "objectclass" );
    String[] names = { "top" };
    /* attr should never be null, but there is a bug in
       "cn=monitor,cn=ldbm" */
    if ( attr != null ) {
        Enumeration e = attr.getStringValues();
        while ( e.hasMoreElements() ) {
            String name = (String)e.nextElement();
            objectClasses.put( name.toLowerCase(), name );
        }
    }
    return objectClasses;
}
}

```

```
/**
 * Report if this node is to be considered a container
 *
 * @return true if the node has children or is of container
 * type
 */
protected boolean checkIfContainer() {
    if ( getCountFromEntry() > 0 ) {
        return true;
    }
    Hashtable containers = getModel().getContainers();
    Enumeration e = _objectClasses.elements();
    while ( e.hasMoreElements() ) {
        String s = (String)e.nextElement();
        if ( containers.get( s ) != null ) {
            return true;
        }
    }
    return false;
}

/**
 * Find an appropriate image for a node, based on the
 * object classes specified and whether or not it is a
 * leaf node
 *
 * @param objectClasses hash table containing object classes
 * for which to look for an icon
 * @param isLeafNode true if this is for a leaf node
 * @return an appropriate image
 */
static ImageIcon checkIcon( Hashtable objectClasses,
                            boolean isLeafNode ) {
    String iconName = "";
    Enumeration e = objectClasses.keys();
    while ( e.hasMoreElements() ) {
        String s = ((String)e.nextElement()).toLowerCase();
        iconName = (String)_icons.get( s );
        if ( iconName == null ) {
            iconName = _resource.getString( _section,
                                           s+"-icon" );

            if ( iconName == null )
                iconName = "";
            _icons.put( s, iconName );
        }
    }
}
```

```

        if ( !iconName.equals( "" ) )
            break;
    }
    if ( iconName.equals( "" ) ) {
        if ( isLeafNode )
            iconName = _defaultImageName;
        else
            iconName = _defaultFolderImageName;
    }
    return ImageUtil.getPackageImage( iconName );
}

```

`DirModel` delegates most messages from `JTree` to `DirNode`. Of special interest is how `DirNode` handles the `isLeaf`, `getChildCount`, and `getChildAt` methods. We don't want to populate the whole tree when it is instantiated. That might take a long time and use a lot of memory. We want to read only as much as is necessary to render the parts of the tree that the user has traversed.

With Netscape Directory Server or MessagingDirect LDAP Server, you can determine if a particular node has children and how many children it has by reading the `numSubordinates` operational attribute. For the case in which all nodes are to be displayed in the tree, the `numSubordinates` operational attribute is checked. If only container nodes are to be displayed, it is necessary to do a one-level search of the directory to see if there are any child entries of the container type. However, this search has been optimized in the following block of code: if `numSubordinates` is 0, then the number of container children must also be 0, and the search is not performed.

Netscape Directory Server also provides the `hasSubordinates` operational attribute, which indicates whether or not a node has child entries, but not how many. Other LDAP servers may provide similar attributes with other names.

```

/**
 * Create a vector of all the one-level subnodes. The nodes
 * are also added as subnodes to this node.
 *
 * @return a Vector of all direct child nodes
 */
protected Vector getChildList() {
    String dn = getDN();
    Debug.println(9, "DirNode.getChildList: <" + dn +
        ">, " + getChildFilter() );
    Vector v = null;
    removeAllChildren();

    try {
        LDAPConnection ldc = getLDAPConnection();

```

```
if ( ldc == null ) {
    Debug.println( "DirNode.getChildList: " +
        "no LDAP connection" );
    return new Vector();
}
LDAPSearchConstraints cons =
    ldc.getSearchConstraints();
// Unlimited search results
cons.setMaxResults( 0 );
LDAPControl[] controls;
if ( !getModel().getReferralsEnabled() ) {
    // If not following referrals, send the
    // managedDSAIT control, which tells the server
    // to return referral entries as ordinary
    // entries
    controls = new LDAPControl[2];
    controls[0] = _manageDSAITControl;
} else {
    controls = new LDAPControl[1];
}
// Ask the server to sort the results, by
// specifying a sort control
String[] sortOrder =
    { "sn", "givenName", "cn", "ou", "o" };
LDAPSortKey[] keys =
    new LDAPSortKey[sortOrder.length];
for( int i = 0; i < sortOrder.length; i++ ) {
    keys[i] = new LDAPSortKey( sortOrder[i] );
}
controls[controls.length-1] =
    new LDAPSortControl( keys, false );
cons.setServerControls( controls );
// Search for immediate children
LDAPSearchResults result =
    ldc.search( dn, ldc.SCOPE_ONE,
        getChildFilter(),
        _baseAttrs, false, cons );
Debug.println(9, "DirNode.getChildList: <" + dn +
    "> searching" );

int found = 0;
while ( result.hasMoreElements() ) {
    try {
        // Add each entry found to the tree
        LDAPEntry entry = result.next();
```

```

        Debug.println(7, "DirNode.getChildList: " +
            "adding <" +
            entry.getDN() + ">" );
        DirNode node =
            new DirNode( getModel(), entry );
        insert( node, super.getChildCount() );
        found++;
    } catch (LDAPException e) {
        Debug.println( "DirNode.getChildList: " +
            "<" + dn + ">: " + e );
    }
    if ( (found % 100) == 0 ) {
        Debug.println(5, "DirNode.getChildList: " +
            "added " + found );
    }
}
_iChildren = super.getChildCount();
Debug.println(9, "DirNode.getChildList: <" +
    dn + "> found " + found);
} catch (LDAPException e) {
    Debug.println( "DirNode.getChildList: " +
        "<" + dn + "> " + e );
}
return children;
}

/**
 * Count the number of children of this node that are containers
 *
 * @return the number of children that are containers
 */
protected int countContainerChildren() {
    String dn = getDN();
    int count = 0;
    try {
        LDAPConnection ldc = getLDAPConnection();
        if ( ldc == null ) {
            Debug.println(
                "DirNode.countChildren: " +
                "no LDAP connection" );
            return count;
        }
        LDAPSearchConstraints cons =
            ldc.getSearchConstraints();
        cons.setMaxResults( 0 );
    }
}

```

```
        if ( !getModel().getReferralsEnabled() ) {
            cons.setServerControls( _manageDSAITControl );
        }
        String[] attrs = { "dn" }; // Pseudo-attribute
        String filter = (all) ? _allFilter :
            getChildFilter();
        Debug.println(9, "DirNode.countChildren: " +
            "<" + dn + "> , " + filter );
        LDAPSearchResults result =
            ldc.search( dn, ldc.SCOPE_ONE,
                filter,
                attrs, false, cons );

        while ( result.hasMoreElements() ) {
            try {
                LDAPEntry entry = result.next();
                Debug.println(9, "DirNode.countChildren: " +
                    "<" + entry.getDN() + ">" );
                count++;
            } catch (LDAPException e) {
                // This is for inline exceptions and
                // referrals
                Debug.println( "DirNode.countChildren: " +
                    "<" + dn + "> " + e );
            }
        }
    } catch (LDAPException e) {
        // This is for exceptions on the search request
        Debug.println( "DirNode.countChildren: " +
            "<" + dn + ">: " + e );
    }
    return count;
}

/**
 * Return the value of the numSubordinates attribute in an
 * entry, or -1 if the attribute is not present
 *
 * @param entry the entry containing the attribute
 * @return the number of children, or -1
 */
static int getCountFromEntry( LDAPEntry entry ) {
    String s = getFirstValue( entry, SUBORDINATE_ATTR );
    if ( s != null ) {
```

```

        int count = Integer.parseInt( s );
        if ( _verbose ) {
            Debug.println( "DirNode.getCountFromEntry: <" +
                entry.getDN() + "> = " + count );
        }
        return count;
    }
    return -1;
}

/**
 * Return the value of the numSubordinates attribute in the
 * entry of this node, or -1 if the attribute is not present
 *
 * @return the number of children, or -1
 */
protected int getCountFromEntry() {
    return getCountFromEntry( getEntry() );
}

/**
 * Report the number of children (containers only) of this
 * node
 *
 * @return the number of container nodes that are children
 * of this node
 */
public int getChildCount() {
    /* If there are no children at all, ... */
    if ( !hasCheckedForChildren() ) {
        if ( !isLoading() ) {
            load();
        }
        int count = getCountFromEntry();
        if ( count < 1 ) {
            _iChildren = 0;
        }
    }

    if ( _iChildren < 0 ) {
        if ( getModel().getAllowsLeafNodes() ) {
            _iChildren = getCountFromEntry();
        } else {
            _iChildren = countContainerChildren();
        }
    }
}

```

```
    }
    return _iChildren;
}

/**
 * Check whether or not the node is a leaf node. Since this
 * is used by JTree to determine whether or not to put an
 * expander on the tree, return true if the node currently
 * has no children.
 *
 * @return true if the node is leaf, false otherwise
 */
public boolean isLeaf() {
    int count = getChildCount();
    Debug.println( 9, "DirNode.isLeaf: <" + getDN() +
        "> : " + count +
        " children" );
    return ( count == 0 );
}

/**
 * Return a specific child of this node, by index. This
 * currently assumes that all nodes in the tree are
 * explicitly managed by JFC (and not by a virtual tree
 * where we supply the contents).
 *
 * @param index zero-based index of child to return
 * @return the node at the requested index, or null
 */
public TreeNode getChildAt( int index ) {
    TreeNode node = null;
    Debug.println( 9, "DirNode.getChildAt: <" +
        getDN() + "> index " + index );

    /* Search for and collect all children, if not already done */
    int count = getChildCount();
    if ( count > super.getChildCount() ) {
        reload();
    }

    try {
        node = super.getChildAt( index );
    } catch ( Exception e ) {
        // Request for node outside of range
        Debug.println( "DirNode.getChildAt: " + count +
            " children " +
```

```

        "available, number " + index +
        " requested: " + e );
    }
    Debug.println( 9, "DirNode.getChildAt: found <" +
        ((DirNode)node).getDN() + ">" );
    return node;
}

```

Put It All Together, What Do You Get?

Using the Bean is as simple as

```

LDAPConnection ldc = DirUtil.getLDAPConnection( "manta.mcom.com",
                                                389,
                                                "cn=directory manager",
                                                "password" );
TreePanel tree = new TreePanel( new DirModel( ldc ) );

```

Figure 10-13 shows `TreePanel` in the mode in which only container entries are displayed. `TreePanel` is displayed in the application `TestTree`, which allows selection of various options from the command line:

- Display leaf nodes or only container nodes
- Show private suffixes or only public suffixes

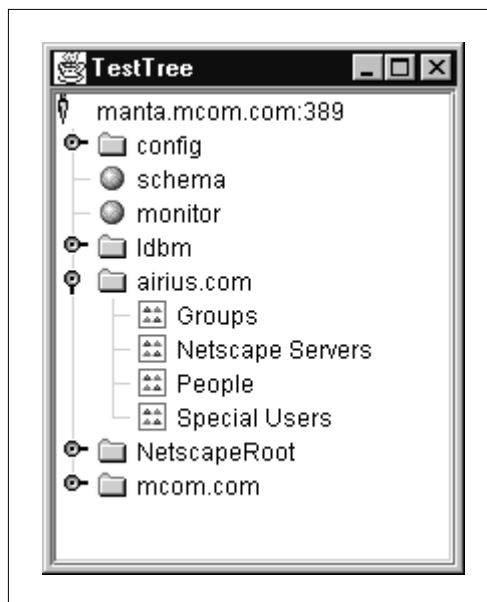


FIGURE 10-13. `TreePanel` with only containers.

- Follow referrals or not
- Show only the tree or the tree together with a table of child entries

If we want to display all nodes, including leaf nodes, then instantiation and initialization have to be done separately:

```
LDAPConnection ldc = DirUtil.getLDAPConnection( "manta.mcom.com",  
                                                389,  
                                                "cn=directory manager",  
                                                "password" );  
  
DirModel model = new DirModel( ldc );  
model.setAllowsLeafNodes( true );  
model.initialize( null );  
TreePanel tree = new TreePanel( model );
```

In Figure 10-14, both container entries and leaf entries are displayed.

To compile the code examples, you will need to have the bin directory of your Java 2 installation in your PATH, and the LDAP JAR file `ldapjdk.jar`, as well as the

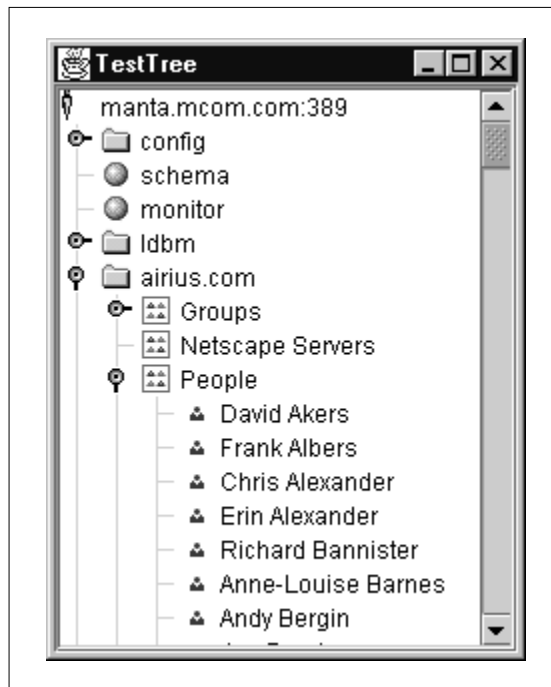


FIGURE 10-14. `TreePanel` with all nodes.

current working directory, in your CLASSPATH, as described in Chapter 3. You can compile the examples in the source code directory for Chapter 10 using the following command:

```
javac *.java
```

You can run `TreePanel` as an application using the command

```
java TreePanel localhost 389 "cn=directory manager" password
```

Substitute the host name of the machine where the directory is installed (if it is not on the same machine where you are running the application), the port number of the directory, a valid distinguished name, and a password for that DN. You can use any valid DN and password, including anonymous, but a nonprivileged user will not be able to see any of the private suffixes. The command looks like this for anonymous access:

```
java TreePanel localhost 389 "" ""
```

Figure 10-15 illustrates `TreePanel` with only public suffixes displayed (because it is executed as a nonprivileged user).

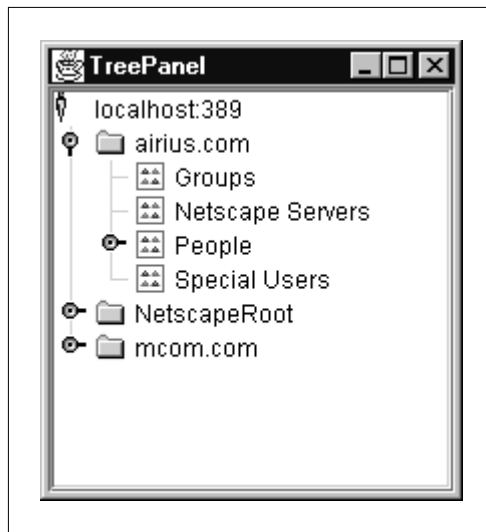


FIGURE 10-15. *Tree viewed by anonymous user.*

A Directory Lister

Displaying all the nodes of the directory in a tree has some disadvantages:

- It is hard to find a particular entry if a particular node contains many entries.
- You may end up with a large number of entries in memory when the user expands a node with many children.
- If you want to display attributes of entries, and not just the DNs or names, you can display only one at a time.

It is common to display only container nodes in a tree, and then show children of a selected node in a list or table. The list or table may display several attributes of each child. The next JavaBean (`SimpleTable`) is a table with sorting, and an adapter to hook up the `TreePanel` Bean with the table Bean.

`SimpleTable` (Figure 10-16) is a `JPanel` containing a simple extension of `JTable`. It adds support for sorting by clicking on the column headers, and for easily changing the headers and data dynamically. It is not LDAP-aware; that's the purpose of the adapter. It is also not suitable for very large numbers of entries. In Chapter 16 we will discuss the use of Virtual List View to handle the display of large databases.

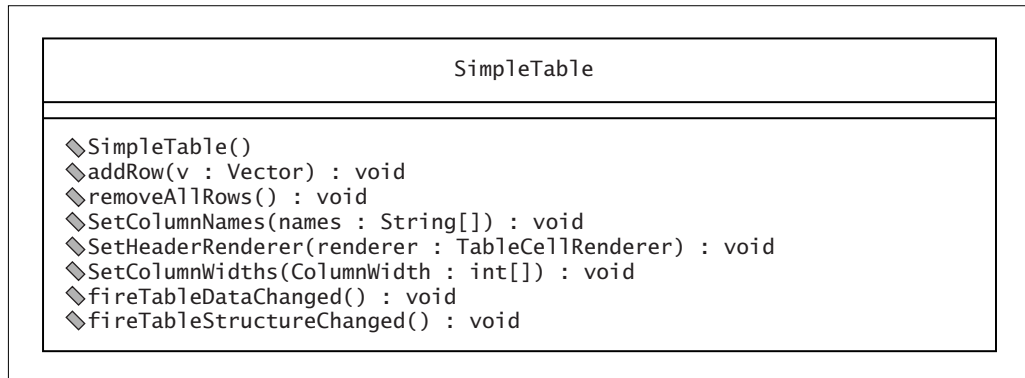


FIGURE 10-16. `SimpleTable`.

SimpleTable offers the following methods of interest, dispatching to the table:

```

/**
 * Add a row to the table model without triggering an event to notify
 * the table to update itself. After finishing adding all rows,
 * fireTableDataChanged should be called to notify the table.
 *
 * @param v a row to add
 */
public void addRow( Vector v ) {
    _tableModel.getDataVector().addElement( v );
}

/**
 * Remove all rows from the model and notify the table
 */
public void removeAllRows() {
    Vector v = _tableModel.getDataVector();
    v.removeAllElements();
    _tableModel.fireTableDataChanged();
}

/**
 * Set the column header labels and attach the sorting header
 * renderer
 *
 * @param names array of one label for each column
 */
public void setColumnNames( String[] names ) {
    _tableModel.setColumnIdentifiers( names );
    setHeaderRenderer( _renderer );
}

/**
 * Attach the sorting header renderer to each column
 */
public void setHeaderRenderer( TableCellRenderer renderer ) {
    TableColumnModel model = _table.getColumnModel();
    for ( int i = model.getColumnCount() - 1; i >= 0; i-- ) {
        model.getColumn(i).setHeaderRenderer( renderer );
    }
}

/**
 * Set the column widths of the table

```

```
*
* @param columnWidth array of one width for each column
*/
public void setColumnWidths( int[] columnWidth ) {
    TableColumnModel model = _table.getColumnModel();
    for (int i = 0; i < columnWidth.length; i++) {
        model.getColumn(i).setPreferredWidth(columnWidth[i]);
    }
    _table.sizeColumnsToFit( -1 );
}

/**
* Notify the table that all the data has changed
*/
public void fireTableStructureChanged() {
    _tableModel.fireTableStructureChanged();
    _tableModel.getIndexes();
}
}
```

EntryListAdapter (Figure 10-17) acts on selection events from TreePanel and searches the directory to provide data for SimpleTable:

```
/**
* The selection changed
*
* @param nodes array of selected tree nodes
*/
public void selectionChanged( IDirNode[] nodes ) {
    String dn = nodes[0].getDN();
    Debug.println( "EntryListAdapter.selectionChanged: " + dn );
    _table.removeAllRows();
    try {
        LDAPSearchConstraints cons =
            (LDAPSearchConstraints)_ldc.getSearchConstraints().clone();
        cons.setMaxResults( 0 );
        LDAPControl[] controls = new LDAPControl[1];
        String[] sortOrder = { "sn", "givenName", "cn", "ou", "o" };
        LDAPSortKey[] keys = new LDAPSortKey[sortOrder.length];
        for( int i = 0; i < sortOrder.length; i++ ) {
            keys[i] = new LDAPSortKey( sortOrder[i] );
        }
        controls[controls.length-1] =
            new LDAPSortControl( keys, true );
        cons.setServerControls( controls );
        LDAPSearchResults result =
            _ldc.search( dn, _ldc.SCOPE_ONE, "objectclass=*",
```

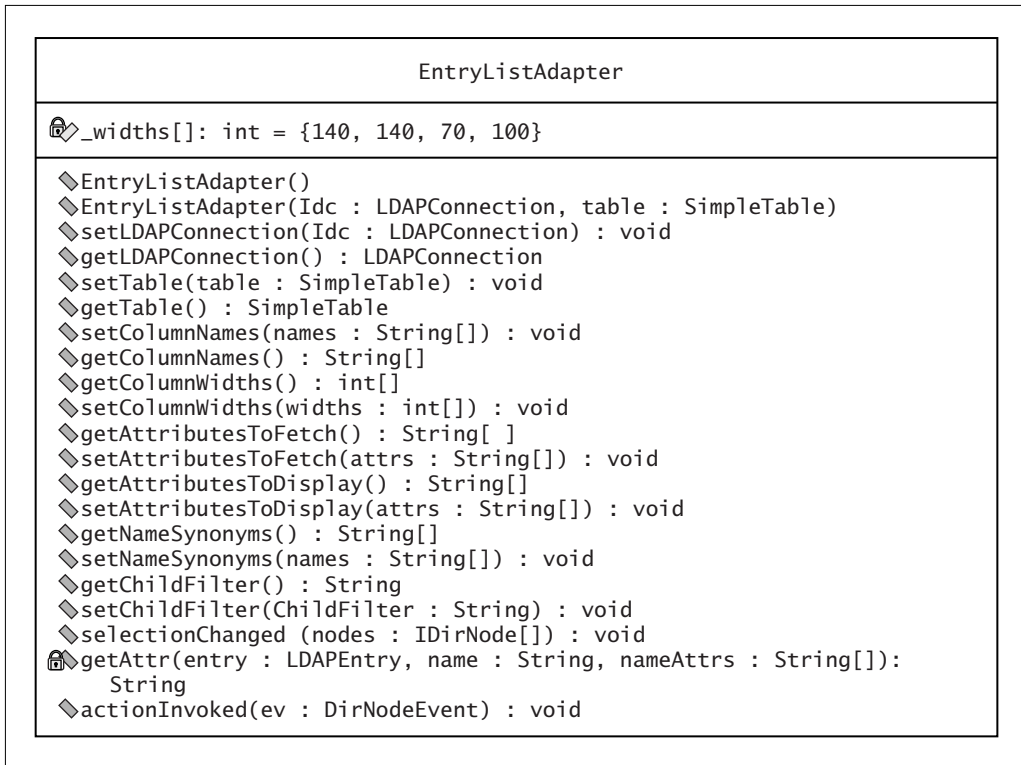


FIGURE 10-17. EntryListAdapter.

```

        getAttributesToFetch(), false, cons );
LDAPEntry entry;
Vector all = new Vector();
String[] displayAttrs = getAttributesToDisplay();
String[] nameAttrs = getNameSynonyms();
while ( result.hasMoreElements() ) {
    Vector v = new Vector();
    entry = result.next();
    v.removeAllElements();
    for( int i = 0; i < displayAttrs.length; i++ ) {
        v.addElement( getAttr( entry, displayAttrs[i],
            nameAttrs ) );
    }
    _table.addRow( v );
}
_table.fireTableStructureChanged();
_table.setColumnNames( getColumnNames() );
_table.setColumnWidths( getColumnWidths() );

```

```
    } catch (LDAPException e) {
        System.err.println( "EntryLister.selectionChanged" +
            " cannot get entry <" + dn + ">" );
    }
}

private String getAttr( LDAPEntry entry, String name,
    String[] nameAttrs ) {
    String value = " ";
    LDAPAttribute attr = null;
    // For the special cn case, check several possible
    // attributes
    if ( name.equals( "cn" ) ) {
        for( int i = 0;
            (attr == null) && (i < nameAttrs.length); i++ ) {
            attr = entry.getAttribute( nameAttrs[i] );
        }
    } else {
        attr = entry.getAttribute( name );
    }
    if ( attr != null ) {
        Enumeration en = attr.getStringValues();
        if ( (en != null) && (en.hasMoreElements()) ) {
            value = (String)en.nextElement();
        }
    }
    return value;
}
```

EntryListAdapter has accessor methods to configure what attributes are displayed, how the columns should be labeled, and so on:

```
/**
 * Headings to display on columns. The default is:
 * { "Name", "Email", "User ID", "Phone" }
 *
 * @return headings to display on columns
 */
public String[] getColumnNames() {
    return _names;
}

public void setColumnNames( String[] names ) {
    _names = names;
}
```

```
/**
 * Widths of columns. The default is:
 * { 140, 140, 60, 100 }
 *
 * @return widths of columns
 */
public int[] getColumnWidths() {
    return _widths;
}
public void setColumnWidths( int[] widths ) {
    _widths = widths;
}

/**
 * Attributes to search for. The default is:
 * { "cn", "mail", "uid", "telephoneNumber", "ou",
 *   "o", "displayName" }
 *
 * @return attributes to search for
 */
public String[] getAttributesToFetch() {
    return _attrs;
}
public void setAttributesToFetch( String[] attrs ) {
    _attrs = attrs;
}

/**
 * Attributes to display. The default is: { "cn", "mail",
 * "uid", "telephoneNumber" }
 *
 * @return attributes to display
 */
public String[] getAttributesToDisplay() {
    return _displayAttrs;
}
public void setAttributesToDisplay( String[] attrs ) {
    _displayAttrs = attrs;
}

/**
 * Attributes that may be used to display in the cn column,
 * in priority order. The default is:
 * { "displayName", "cn", "ou", "o" }.
 */
```

```
    *
    * @return attributes that may be used to display in the cn
column
    */
    public String[] getNameSynonyms() {
        return _nameAttrs;
    }
    public void setNameSynonyms( String[] names ) {
        _nameAttrs = names;
    }
}
```

TreePanel and SimpleTable Join Forces

It doesn't take much code to hook up the two Beans:

```
LDAPConnection ldc = DirUtil.getLDAPConnection( "manta.mcom.com",
                                                389,
                                                "cn=directory manager",
                                                "secretdog" );

TreePanel tree = new TreePanel( new DirModel( ldc ) );
SimpleTable table = new SimpleTable();
EntryListAdapter lister = new EntryListAdapter( ldc,
                                                tree,
                                                table );

// EntryListAdapter could add itself as a listener, but we'll
// do it here just to demonstrate
tree.addDirNodeListener( lister );
JSplitPane splitPane = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT,
                                       sp, table);
splitPane.setBorder( new EmptyBorder(6, 6, 6, 6) );
frame.getContentPane().add( splitPane );
```

Figure 10-18 shows `TreePanel` and `SimpleTable` connected and functioning as a directory browser.

Figure 10-19 demonstrates that the data can be sorted by any column and in either ascending or descending order.

`SimpleTable` also offers events on selection, so it can be used to trigger other actions—for example, to launch a user editor for a selected user, or to delete the selected user.

You can run the `TestTree` application using the following command:

```
java TestTree localhost 389 "cn=directory manager" password
```

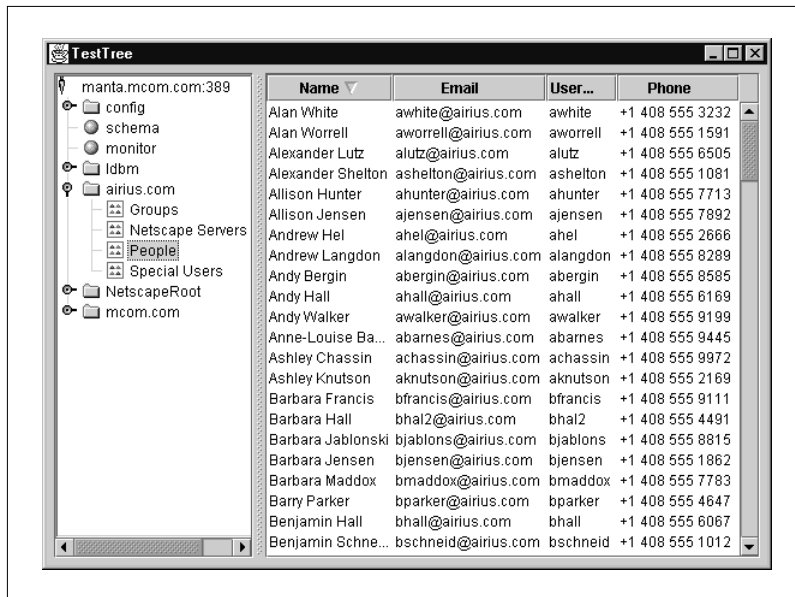


FIGURE 10-18. TreePanel and SimpleTable.

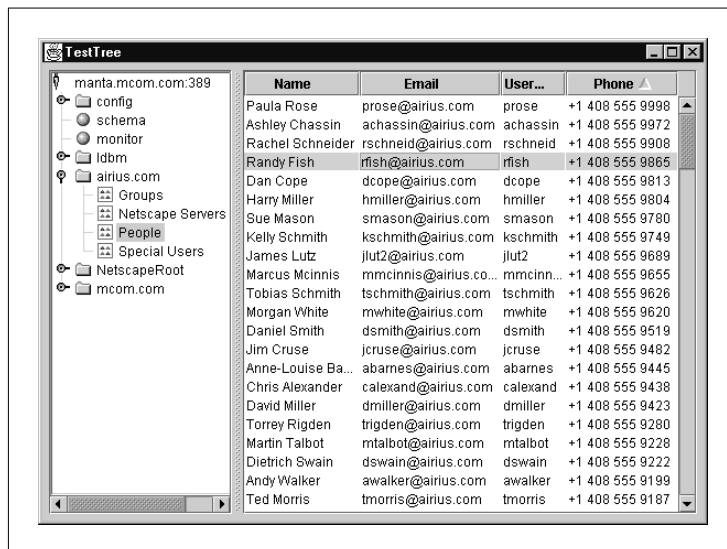


FIGURE 10-19. Selecting a different sort column and direction.

Substitute the host name of the machine where the directory is installed (if it is not on the same machine where you are running the application), the port number of the directory, a valid distinguished name, and a password for that DN.

Conclusion

In this chapter we have looked at several JavaBeans for LDAP programming. Some are invisible and simply encapsulate subsets of the functionality of the class library; others capture and render directory contents in graphical form and allow the user to interact with them. LDAP JavaBeans can be combined easily, using a small amount of Java or JavaScript glue code. The Beans can be configured and the glue code generated in an IDE (integrated development environment) that understands JavaBeans, without additional coding. A complete application or applet can be composed of one or more LDAP JavaBeans that communicate through a listener interface.