WebNFS Developer’s Guide

Sun Microsystems, Inc.
2550 Garcia Avenue
Mountain View, CA 94043-1100
U.S.A.

Part No: 806-1067–10
February, 2000

Contents

Preface

The *WebNFS Developer's Guide* provides an overview of the WebNFS™ Software Developer's Kit (SDK), a set of Java™-based Application Programming Interfaces (APIs) that allow programmers to

- Connect to NFS v.2 or v.3 servers using TCP or UDP, and to make use of NFS URLs to access remote filesystems;
- make use of an API that mimics the `java.file.io.*` classes, providing remote file access for multiple filesystem types;
- use a Java Bean™ UI component that allows users to select remote files.

Who Should Use This Book

This book is intended for Java-savvy programmers who want to write networking applications that can access files on different types of remote systems.

Before You Read This Book

You should be familiar with the basics of Java programming. Experience with NFS is helpful. Knowledge of transport protocols such as UDP or TCP is helpful but not necessary.

How This Book Is Organized

Chapter 1 is an overview of the WebNFS SDK.

Chapter 2 describes the Extended Filesystem (XFile) API.

Chapter 3 is an overview of the WebNFS classes used by the XFile API.

Chapter 4 is an overview of the methods of the XFileAccessor interface, which allows the programmer to write filesystem-type-specific accessors.

Chapter 5 describes the XFileChooser Bean and gives examples of how to use it in applications.

Appendix A gives several of the most-frequently asked questions about WebNFS, and their answers.

Appendix B gives some short, simple examples of using the XFile API.

Appendix C is the code for a sample program for a simple editor that makes use of the XFileChooser Bean.

Appendix D is the code for a demo program that makes use of most of the features of the XFileChooser Bean. The chapter includes auxilliary programs.

Related Books

The following books are not prerequisites for this book but may provide useful background.

- *NFS Administration Guide*
- *ONC+ Developer's Guide*

Ordering Sun Documents

The Sun Software Shop stocks select manuals from Sun Microsystems, Inc. You can purchase individual printed manuals and AnswerBook2™ CDs.

For a list of documents and how to order them, visit the Software Shop at `http://www.sun.com/software/shop/`.

Accessing Sun Documentation Online

The docs.sun.com^{SM} Web site enables you to access Sun technical documentation online. You can browse the docs.sun.com archive or search for a specific book title or subject. The URL is `http://docs.sun.com`.

What Typographic Conventions Mean

The following table describes the typographic changes used in this book.

TABLE P–1 Typographic Conventions

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls –a` to list all files. `machine_name% you have mail.`

`AaBbCc123`	What you type, contrasted with on-screen computer output	`machine_name%` **`su`**
		`Password:`

AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type **`rm`** *filename*.
AaBbCc123	Book titles, new words or terms, or words to be emphasized.	Read Chapter 6 in *User's Guide*. These are called *class* options. Do *not* save changes yet.

Shell Prompts in Command Examples

The following table shows the default system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

TABLE P–2 Shell Prompts

Shell	**Prompt**
C shell prompt	`machine_name%`
C shell superuser prompt	`machine_name#`
Bourne shell and Korn shell prompt	`$`
Bourne shell and Korn shell superuser prompt	`#`

CHAPTER 1

Introduction to the WebNFS Software Development Kit

There is much demand for distributed filesystem protocols, such as the WebNFS™ protocol. In particular, experienced Java™ application developers want to be able to access all files on the Internet in the same way. The WebNFS Software Development Kit (WebNFS SDK) includes Java class libraries that provide a way to access files using the same API for local and remote file access.

SDK Features

The SDK is composed of

- WebNFS classes
- Extended Filesystem API (XFile API)
- XFileChooser Bean API

NFS Classes

The WebNFS classes make it possible to establish connections to an NFS™ version 2 or version 3 server, using either TCP or UDP. These classes also provide consistent naming with NFS URLs. Further information about these classes can be found in Chapter 3 and in the WebNFS classes javadoc files included with this release.

The Extended Filesystem API

The Extended Filesystem (XFile) API classes provide a common interface for accessing files of various filesystem types. The classes also allow for dynamic loading of filesystem implementations. The XFile API also provides a means to access file– and filesystem–specific information. Further information about these classes can be found in Chapter 2, and the XFile javadoc files included with this release.

Using the Extended Filesystem API

The XFile API includes classes similar to `java.io.*`'s, which have been extended to include file access. For example, there are equivalents to `java.io`'s `FileInputStream` and `FileWriter` classes, among others, so that using the XFile classes is generally similar to using classes in `java.io`. Additionally, the `XFile` class supports accessing files through URLs, as specified by the NFS URL Scheme.

Sample code showing how to use the XFile API is included with this release.

The XFileChooser Bean

The XFileChooser (WebNFS) Bean API classes provide a high-level component in which users can easily incorporate into their design. The XFileChooser extends an existing Bean provided in JDK 1.2 (JFileChooser) to support access to remote files via the XFile API. Thus the actual code of the WebNFS bean will combine the functionality of the GUI component part of the JFileChooser and the XFile API. For more information, see the Chapter 5 and the javadocs for the XFileChooser which are included with this release.

Using the XFileChooser API

The XFileChooser API provides methods that allow users to get the `XFile` object of a selected file. It inherits most of the properties from the `JFileChooser` and thus methods that are public to `JFileChooser` are available for `XFileChooser`.

Technical References

- RFC1094 NFS™: Network File System Protocol Specification (`http://ds.internic.net/rfc/rfc1094.txt`)
- RFC2224 NFS URL Scheme (`http://ds.internic.net/rfc/rfc2224.txt`)

- RFC1813 NFS Version 3 Protocol Specification (`http://ds.internic.net/rfc/rfc1813.txt`)
- RFC2054 WebNFS Client Specification (`http://ds.internic.net/rfc/rfc2054.txt`)
- RFC2055 WebNFS Server Specification (`http://ds.internic.net/rfc/rfc2055.txt`)
- WebNFS White Paper (`http://www.sun.com/software/webnfs/wp-webnfs`)

CHAPTER 2

Extended Filesystem API

Network computer vendors have realized that network file systems can add a powerful capability to their products. Experienced Java application developers want access to all the files the Internet has to offer in the same way. To meet this demand, several vendors are working on distributed file systems, such as WebNFS, CIFS, and others.

This chapter describes the Extended Filesystem API (XFile API). The Extended Filesystem API provides a common interface for multiple filesystem types. Also, it allows dynamic loading of filesystem implementations. The API includes a set of classes similar to those in the `java.io` API, and therefore provides a familiar programmatic means of accessing files, either locally or remotely. The API also provides a means to access file- and filesystem-specific information.

Features

The Extended Filesystem model:

- is filesystem-neutral
- supports access to multiple filesystems
- is extensible to support custom features of new filesystems
- supports filesystem identification and naming through URLs
- provides `java.io.File`–type access to developers to facilitate migration of existing applications.

Overview

The extended filesystem provides extensibility through the use of:

- An `XFileAccessor` interface that provides a common set of access methods that are required for all filesystems.
- An `XFileExtensionAccessor` abstract class that provides direct access to filesystem-specific features.
- The capability for automatic runtime selection of the file access mechanism.
- Support for multiple file namespaces through URL naming.

The API defines two means of access to network files and filesystems:

- An `XFile` class that supports naming of files with URLs such as the type described in Internet RFC 2224 (`ftp://ftp.isi.edu/in-notes/rfc2224.txt`). Here users specify URLs like `nfs://oaktree.edu/archie`. Developers of other filesystems can register their own URL schemes and implement filesystem accessor classes to support a chosen filesystem type, such as `smb://www.microsoft.com/index.html`.

  The `XFile` class also provides a default "native" URL scheme using `java.io.*` that has a file-naming syntax that varies according to the operating system that supports the Java Runtime Interface, such as Win95, UNIX, and VMS.
- Access to data through the `XFileInputStream`, `XFileOutputStream`, `XRandomAccessFile`, `XFileReader`, and `XFileWriter` classes that are similar to current `java.io.File` classes.

The API supports three categories of applications:

- Web-based applications, specifically browsers that deal with URLs extensively.
- Traditional file-based applications that must extend their access to the network and still use the style of access provided by `java.io.File` classes.
- Mixed-mode applications that use both remote and distributed filesystems.

Architecture

The following picture shows the extended filesystem architecture. The XFile (`com.sun.xfile.*`) layer of the architecture mirrors the standard `java.io.File*` programmatic interfaces. Under this layer is a set of `XFileAccessors`. There is one `XFileAccessor` interface implementation for each filesystem being supported. (Currently only NFS and native filesystems are supported.) The accessors implement the `XFileAccessor` interface. They are responsible for all aspects of file access. The API classes are responsible for multiplexing between the different filesystems as requests are made from the application.

TABLE 2–1 Extended Filesystem Architecture

Java Application			
`com.sun.xfile.*`			
nfs:	cifs:	file:	native:

The WebNFS SDK includes support for dynamic loading of file systems. Please see Chapter 4.

URL Naming

Every filesystem type is identified by a URL of the general form described in RFC 1738 (`ftp://ftp.isi.edu/in-notes/rfc/1738.txt`). The extended filesystem uses the scheme name in the URL to automatically select a filesystem. For example, the NFS filesystem has the scheme name of “nfs.” When presented with a filename string, the extended filesystem checks for a URL scheme followed by a colon at the beginning of the string. If it finds one, it uses the filesystem accessor class associated with that URL scheme. If it does not find one, it defaults to the "native" scheme for access through `java.io.*`.

URL Names and Native Names

Filenames are assumed to be *absolute* URLs or *relative* URLs as determined by a context. If a filename string begins with a valid URL scheme name followed by a colon, then the name is associated with the filesystem type indicated by the URL scheme name. For instance, a filename string of the form `nfs://hostname/path` is associated with the filesystem that has the scheme name of `nfs`. If the filename has no colon-separated prefix, or the scheme is not recognized then the name is assumed to belong to the default "native" scheme, that is, a URL prefix of `native:` is assumed and the name is handled by `java.io.*`. The two-argument constructor for XFile takes an `XFile` object and a filename:

```
XFile(Xfile dir, String filename)
```

The *filename* argument is interpreted according to the context set by the *dir* argument. The *dir* argument is used as a base URL and the name is evaluated as a relative URL. If the *filename* is an absolute URL, then the *dir* is ignored. If the *filename* is a relative URL, then it is combined with the base URL to form the name of the new XFile object. The rules that describe the evaluation of relative URLs are described in RFC 1808 (`ftp://ftp.isi.edu/in-notes/rfc1808.txt`).

Filesystem Selection

Filesystems are selected using these rules:

1. If the `XFile(String filename)` constructor is used and the *filename* has a prepended URL scheme, the filesystem associated with that scheme is used; otherwise, the native filesystem is used.
2. If the `XFile(XFile dir, String filename)` constructor is used, the *filename* is evaluated as a relative URL to the *dir* XFile.

Examples of File Name Usage and Filesystem Selection

The following examples show how the `XFile` constructors may be used:

```
XFile f1 = new XFile("nfs://hostname/directory1");

// NFS file based on an absolute URL

XFile f2 = new XFile("file.txt");

// not a URL, so defaults to the current directory of the "native" filesystem

XFile f4 = new XFile(f1,"text.html");

// evaluated relative to the base URL of f1 - which is "nfs".
```

One point to be made from these examples is that the validity of a constructed `XFile` is made when the `XFile` is used, not when it is constructed. This is consistent with the `java.io.File` class.

Direct Access to Filesystem-Specific Features

A reference to the object implementing a filesystem's specific features can be determined through the class interface. The interface allows developers to directly call these methods. By doing this, you are increasing the risk of writing code that works only for one filesystem type, so its use is discouraged. To obtain access to these features directly, use the public method `XFile.getExtensionAccessor()`. Once you have obtained this handle, you can access its methods directly. Refer to the filesystem implementor's documentation for class description details.

The following example shows how a method in the NFS `XFileExtensionAccessor` is used to set the user's authentication credentials using the NFS PCNFSD protocol:

CODE EXAMPLE 2–1 Using an NFS `XFileExtensionAccessor`

```
import java.io.*;
import com.sun.xfile.*;
import com.sun.nfs.*;

public class nfslogin {

     public static void main(String av[])
     {
          try {
               XFile xf = new XFile(av[0]);
               com.sun.nfsXFileExtensionAccessor nfsx =
               (com.sun.nfsXFileExtensionAccessor) xf.getExtensionAccessor();

               if (! nfsx.loginPCNFSD("pcnfsdsrv", "bob", "-passwd-")) {
                    System.out.println("login failed");
                    return;
               }

               if (xf.canRead())
                    System.out.println("Read permission OK");
               else
                    System.out.println("No Read permission");

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

Class Descriptions

This section describes the *additional* methods that are not currently part of the `java.io` classes. For example, the first entry, called `XFile`, describes the methods that are in the `com.sun.xfile.XFile` class but not in the `java.io.File` class. All `com.sun.xfile` classes that have a counterpart in the `java.io` classes are generally a *superset* of the `java.io` classes in the methods provided, not including constructors. File descriptors are not supported and constructors that use a `java.io.File*` class now use a `com.sun.xfile.XFile*` class. Also shown are the few new interfaces and classes defined by the extended filesystem. A complete description of the API can be found in the XFile javadoc files included in the release.

TABLE 2–2 XFile Classes

Classes	`java.io.File` **Equivalent**	**Methods**
`XFile`	`java.io.File`	`public XFile(String url)` - class constructor `public XFile(XFile dir, String relurl)` - class constructor
`XFileOutputStream`	`FileOutputStream`	`public XFileOutputStream(XFile file)` - class constructor
`XFileInputStream`	`FileInputStream`	`public XFileInputStream(XFile file)` - class constructor
`XRandomAccessFile`	`RandomAccessFile`	`public XRandomAccessFile(XFile file, String mode)` -class constructor
`XFilenameFilter`	`FilenameFilter`	`public abstract boolean accept(XFile dir, String name)` - filter based on file spec
`XFileReader`	`FileReader`	`XFileReader(XFile file)`
`XFileWriter`	`FileWriter`	`XFileWriter(XFile file, String access)`

File Interface Examples

Extending existing programs to network filesystems should be straightforward using the extensions provided. You simply replace declarations of type `java.io.FileXYZ` with the counterpart `java.io.XFileXYZ`. Here are a few simple examples of applications that use the APIs:

CODE EXAMPLE 2–2 Using Streams Interface

```
import java.io.*;
import java.net.*;
import com.sun.xfile.*;

XFile xf = new XFile("file.txt");

if (xf.isFile()) {
     System.out.println("file is file");
}

if (xf.isDirectory()) {
     System.out.println("file is directory");
}

XFileInputStream nfis = null;
nfis = new XFileInputStream(xf);

for (int count = 0; ; count++) {
     int val = (byte) nfis.read();
     if (val == -1)
          break;
     System.out.write(val);
}

System.out.println("read " + count + " bytes ");
```

CODE EXAMPLE 2–3 Using the `XRandomAccessFile` Interface

```
import java.io.*;
import java.net.*;
import com.sun.xfile.*;

     // create connection to host

XFile xf = new XFile("nfs://ian/simple.html");

XRandomAccessFile xraf = new XRandomAccessFile(xf,"r");

int count = 10;

xraf.seek(count);
System.out.println("Value at position " + count
     + " is " + (byte)xraf.read());
```

```
System.out.println("Current file position is "
     + xraf.getFilePointer());

xraf.seek(0);

for (count = 0 ; count < 100 ; count++) {
     int val = xraf.read();
     if (val == -1)
          break;
     System.out.print(val);
}
```

CHAPTER **3**

NFS Classes for the Extended Filesystem

This is the first implementation of remote file system access for Java applications that provides 100% Pure Java compatibility. Although there are already Java applications that access remote files through the URLConnection class of `java.net`, the access is generally read-only and limited to whole file transfer as supported by the underlying protocols: HTTP and FTP. There is no provision for the same kind of file access that applications currently enjoy through java.io classes.

Through the Extended File system API and the NFS client, applications can perform the full range of file operations on a remote NFS server.

NFS URL

The NFS URL is a global, universal name for naming files or directories on any NFS server. The general form is similar to that of other URL schemes: `nfs://server/path` where the scheme name is `nfs` and the server name is the name of any NFS server. The path is a slash-separated path that names an NFS-exported file on the server. The words "global" and "universal" are not simply buzzwords: "global" means that the URL can refer to a file or directory on any NFS server in the world whether the server is an IBM mainframe, a UNIX server, or a Windows NT™ server running NFS. The word "universal" means that the URL always has the same syntax whether the client is a Macintosh™, Windows 95™ laptop, or VAX VMS™ Minicomputer. These properties are important for Java applications; Java applications must have global access via the Internet and to be "100% Pure Java™" they must run unchanged across all Java platforms.

The NFS URL Scheme is described in RFC 2224 (`ftp://ftp.isi.edu/in-notes/rfc2224txt`).

Connecting to the Server

When an application first tries to access a file named by an NFS URL through the extended file system, the NFS client will negotiate a connection with the server. First the client will attempt to make a TCP connection since TCP is the preferred transport protocol on the Internet for conveying large volumes of data. In addition, corporate firewalls can be configured to allow NFS TCP connections to Internet servers on port 2049. If the server rejects the TCP connection, the client will then use the UDP protocol. UDP performs as well as TCP on local area networks but is not as well suited to Wide Area Networks and the Internet.

The NFS client incorporates the latest WebNFS technology described in `RFC 2054`. It connects directly with the NFS server and attempts to locate the requested file or directory using a “public filehandle” and “multi-component lookup.” This is a very efficient way to connect to the server since it avoids additional steps needed by conventional NFS clients to "mount" the file system using the NFS MOUNT protocol.

If the NFS server does not support WebNFS connection, the client will attempt to connect using the MOUNT protocol. Since the MOUNT protocol does not use a well-known network port, it cannot easily transit a firewall to connect to Internet servers, but it can be used to connect to local Intranet NFS servers. When accessing files on a server that does not support WebNFS connections, the URL may need an extra slash in the pathname; for instance, if UNIX NFS clients normally use the name `server:/path` in their mount command, then the equivalent URL for the NFS client will be `nfs://server//path`. Whether or not the extra slash needs to be used is not an issue for the user to figure out. It’s the responsibility of whoever distributes the URL to determine the correct path depending on whether the server is a WebNFS server or a MOUNT-only server.

The server administrator has little to do since the NFS client has much in common with other NFS clients. If the server supports the WebNFS service then the public filehandle may be associated with a particular directory by including the "public" option in the share command. The pathname in the NFS URL is relative to the directory with the public filehandle; for instance, if the server exports the directory `/export` with the "public" option then the file `/export/this/file` is named by the NFS URL `nfs://server/this/file`.

On servers running Solaris 2.6 and above, the public filehandle has a default location at the system root;, from this location an NFS URL can name a file or directory on any of its exported file systems.

NFS Versions

The NFS client implements both versions 2 and 3 of the NFS protocol. Version 2 is by far the most widely supported version of NFS. Version 3 features many improvements particularly in performance. To an end-user or application developer the version of NFS that is being used is unimportant, the NFS client negotiates the NFS version automatically with the server. Because version 3 has many performance improvements the client will exercise a preference for that version of the protocol. It will use version 2 only if the server does not support version 3.

TCP or UDP ?

The NFS protocol is transport-independent. It will run over a stream-oriented protocol like TCP or a datagram protocol like UDP. Historically, there are more UDP implementations of the NFS protocol because early implementations of TCP were notoriously slow. In recent years the performance of UDP and TCP implementations on local area networks is comparable and on wide area networks and the Internet there is a distinct preference for the congestion control and reliability of TCP. Additionally, it is much easier to control TCP connections at a firewall than UDP.

The NFS client will first attempt to use TCP to connect to any NFS server whether it be a local server or across the world on the Internet. Only if the server rejects the TCP connection attempt will the NFS client use UDP.

Reliability

More than any other distributed file system protocol, the NFS protocol is known for its reliability and data safety. The NFS version 2 protocol was notorious for slow write speed. The NFS guarantee of data safety required the server to store the data from each write request to disk before replying to the client. Although the "synchronous write" requirement imposed a performance trade-off, the client was assured that a server crash would never lose its data. An NFS server crash or network outage will never result in corrupted or half-written files. Version 3 preserves the data safety guarantee while allowing the server to improve write performance through asynchronous writes. The NFS client uses the improved write technique of version 3 to deliver excellent write performance with data safety.

If an NFS server crashes or the network connection is lost, the NFS client will persist in attempting to restore the connection and continue where it left off. If a TCP connection is broken for any reason, the client will re-establish the connection. If a UDP request or reply is lost, the client will re-transmit the request until the operation succeeds using an exponential backoff on the timeout to avoid overloading the server with retransmissions. Since NFS servers are designed to be stateless, the server need do nothing on recovery other than serve new NFS requests that may include retransmitted requests that were not completed before the crash.

This reliability has tangible benefits for users who are used to the low bandwidth access to busy Internet servers. With protocols like HTTP or FTP a lost connection usually means that a file transfer must begin over and applications that use these protocols will receive an error. These problems are transparent to applications that use the NFS classes through the extended file system. Any pending read or write will block until it completes successfully. In the case of a file transfer over NFS it means that the file transfer will resume automatically from where it left off. This blocking behavior and persistence need not be inconvenient to an interactive user with limited patience; a Java application can implement a "stop" or "abort" button that kills a blocked thread.

In short, the NFS protocol and the NFS client implementation of it are very reliable.

Security

NFS servers currently control access to their files using "trusted host" security. The server trusts that the client machines will construct a valid credential that correctly identifies the individual requesting file access. Generally the server identifies the list of trusted client machines with an access list that explicitly lists the names of the client machines, or identifies a "netgroup" that lists the names of trusted clients or other netgroups. If the client machine is "trusted" by the server then the NFS client will be able to access the server like any other NFS client.

Since the Java Runtime has no concept of the user's identity, the NFS client cannot automatically construct a credential that identifies the user with a UID or GID value, since these values may be meaningless on platforms that do not support them (Macintosh and Windows machines do not require the user to log in). Instead, the client constructs a nobody credential that uses a UID and GID value of 60001 which the server identifies as user nobody. Since file access permissions on public or Internet archive servers are often liberal in allowing read access to anyone, the nobody identity is of no consequence. However, if the user wishes to access private files in his or her own home directory or to create files with his or her identity assigned to the owner of the file then a credential with the user's UID and GID must be used.

The NFS client uses the same technique to acquire a valid user credential as PC-NFS clients — it uses the network PCNFSD service. The PCNFSD service is commonly installed wherever PC-NFS clients are present. It need not be installed on every NFS server, only one server in the network is sufficient. The NFS client provides a `loginPCNFSD()` method in that can be called through the NFS `XFileExtensionAccessor` class. This method takes the user's login name, password and the name of a host running the PCNFSD service and passes this information to the PCNFSD server (the password is not encrypted but it is obscured by exclusive OR-ing with a bit pattern so that the password is not disclosed to casual network sniffers). If the login name and password are valid, the PCNFSD server returns the user's UID and GID, which the NFS classes then use in subsequent requests to the NFS server. Here is an example of the a Java application setting the user's credential:

CODE EXAMPLE 3–1

```
import sun.xfile.*;
import sun.nfs.*;

public class pcnfsd {

     public static void main(String av[])
     {
          try {
               XFile xf = new XFile(av[0]);

               com.sun.nfsXFileExtensionAccessor nfsx =
                    (com.sun.nfsXFileExtensionAccessor) xf.getExtensionAccessor();

               if (! nfsx.loginPCNFSD("pcnfsdsrv", "jane", "-passwd-")) {
                    System.out.println("login failed");
                    return;
               }

               if (xf.canRead())
                    System.out.println("Read permission OK");
               else
                    System.out.println("No Read permission");

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

Of course a more realistic application would likely obtain the parameters for the call to `loginPCNFSD()` from a dialog box with a protected text field for the user's password.

The "trusted host" security has several shortcomings, not the least of which is that it is not particularly secure. A determined network sniffer could monitor PCNFSD calls and obtain passwords, a malicious client could masquerade as a "trusted" host by spoofing the source IP address, or the trusted host itself could be compromised, in which case an intruder could spoof the credential of anyone. In addition, on an Internet scale it would be impractical to maintain access lists for all client machines. Even then, it is common for users to access the server from unpredictable IP addresses either because the client is a "kiosk" machine used by many different users, or because the source IP address is dynamically allocated by a DHCP server.

For this reason there is work underway in the IETF to define more secure NFS authentication. The ONC RPC working group has devised a new framework for secure RPC called "RPCSEC_GSS" (`http://ds.internic.net/rfc/rfc2203.txt`), which is based on the IETF's GSS-API (`http://ds.internic.net/rfc/rfc2708`), an open-ended framework that facilitates "plug-in" security mechanisms that can provide secure authentication, data integrity and data privacy for NFS traffic on the network. (The RPCSEC_GSS and GSS_API are bundled with WebNFS, but are not currently exposed. The Java WebNFS security framework is the Java implementation of the RPCSEC_GSS protocol specified in RFC 2203.) The RPCSEC_GSS mechanism supports public key-based security schemes that are more suitable for Internet use. Sun also offers SEAM™, a Solaris plug-in for Kerberos v5.

Network Performance

A common criticism of Java applications is that they are "too slow." The often-quoted figure is "20 to 40 times" slower than equivalent compiled C or C++ applications. However, it is strongly associated with networks and in a network context Java applications suffer zero speed penalty. In just the last ten years CPU speed has increased tenfold, yet the speed of network access through modems has barely doubled and many of us are still using the same 10 Mb/sec Ethernets we were using 10 years ago (though now switched). The consequence of this is that machines are much faster than the networks they communicate with. Java applications that make use of the network spend most of their runtime waiting for the network.

The NFS client uses several techniques to use the network efficiently. First it caches NFS file attributes and data in memory to avoid unnecessary calls to the server. When the client connects to an NFS version 3 server it eliminates the 8Kb read and write limitation of version 2 and reads or writes data in large 32Kb blocks. Version 3 also allows the client to use safe, asynchronous writes that are several times faster than the synchronous writes required by version 2. Finally, the NFS client utilizes Java threads to implement read-ahead and write-behind. If the client detects that the application is reading a file sequentially then it will asynchronously issue read requests ahead of the current block of data in anticipation of the application's need. The use of a single block read-ahead can double the client's file reading throughput. Similarly, write-behind allows the client application to write blocks of data to the server without waiting for confirmation for each block of data. This has a similar effect on file write throughput.

Our experience with the NFS client is that it can read and write data at over 1 Mb per second across a 10Mb Ethernet from a Sun Ultra 1 client.

Package Size

Implementations of the NFS protocol are assumed to be large. This misconception is perhaps due to NFS clients being integrated with the operating system along with their attendant administrative mount commands, automounters and the like. The NFS classes contain 70Kb of Java bytecode which reduces to 38Kb when delivered in a zip file. The classes that comprise the API contain 55Kb of bytecode (29Kb zipped). The additional XFileChooser Bean classes and GSS-API classes bring the total package size to 124 Kb.

Using NFS Classes through the Extended Filesystem

Although the intent of the API is to hide details of file system implementation behind a common API, there are some features of the underlying file system that should be considered.

Caching

The NFS client uses the same cache model as other NFS client implementations. File data and attributes are cached for a time interval depending on the frequency of update. If the file is updated frequently then the cache time will be as short as 3 seconds. However if the file is updated infrequently then the cache time may be as long as 60 seconds. If the file changes on the server during the time in which the cached copy is considered valid, then the change will not be noticed until the cache time has expired. This caching behavior should be considered when Java applications running in different Java virtual machines or on multiple network clients need to coordinate their activity around a common set of files.

The NFS client caches aggressively for good network performance. Currently the cache is open-ended: as new NFS files or directories are encountered, their filehandles and file attributes are cached. These cache entries are never released, so if a Java application touches a large number of files perhaps in a file tree walk, then the application may encounter an `OutOfMemoryError`. A future version of the client will manage the cache within a bounded amount of memory.

Buffering

When using the `java.io` classes either directly or as a "native" file system under the extended file system, there is no attempt to buffer data explicitly unless the `BufferedInputStream` or `BufferedOutputStream` classes are used. Data is buffered implicitly by the underlying file system.

The NFS classes access the network directly through the Java Socket interface. To provide acceptable performance the NFS classes buffer all reads and writes. The buffer size varies from 8k for NFS version 2 servers to 32 for NFS version 3 servers though the actual buffer size is invisible to the Java application itself. The Java application must be diligent in calling the `close()` method when writing to an `OutputStream` or `XRandomAccessFile` to ensure that a partially filled buffer is written to the NFS server before the application exits. There is nothing in the Java Runtime that will cause partially written buffers to be flushed automatically at application exit.

Symbolic Links

Although the NFS protocol supports the use of symbolic links, neither the java.io classes nor the API acknowledge their existence. The NFS classes make symbolic links transparent to the application by following links automatically in the same way that PC-NFS clients do. If a Java application attempts to access an NFS URL that references a symbolic link, the NFS classes will follow the link (and any further links) and return the attributes of the file or directory that is referenced by the link. A Java application cannot create a symbolic link through the API, though a future release of the NFS `XFileExtensionAccessor` will support this feature.

File Attributes

The `java.io` and the Extended File system APIs support a limited set of file attributes that are considered common across many file system types. For instance all file systems are assumed to support some notion of file size, modification time, access control for read or write and filetypes with common behaviors like directories and "flat" files. The NFS protocol supports some file attributes that are not in the "common" set like the file owner and group (UID and GID), creation time, last access time, and UNIX-style permission bits. While these attributes are not accessible through the XFile API they are accessible through the `XFileExtensionAccessor` classes.

File Accessibility

The XFile methods `canRead()` and `canWrite()` are implemented within the NFS classes with code that checks the permission bits of the file attributes against the user's UID. While this technique is used by all NFS version 2 clients, it may sometimes return misleading result if file access is controlled on the server with an Access Control List (ACL) since the effects of the ACL cannot always be represented by an equivalent set of permission bits. Version 3 supports an ACCESS request that allows the client to request the server do the access check subject to the ACL. The result from ACCESS is always an accurate indication of the file access permitted the user. The NFS client does not currently implement the ACCESS check but will support it in the next release.

File Truncation

The NFS protocol allows a client to control the size of a file by setting the file size attribute. For instance a file can be truncated by setting the file size to zero. The API allows Java applications to obtain the file size through `XFile.length()` but there is no method that allows the length to be set. This may be a future addition.

NFS Exception Information

Many of the I/O methods in the API will throw an IOException for any condition that causes the operation to fail. The underlying NFS classes throw a subclass of IOException called NFSException that conveys more detailed information about the failure in an error code within the Exception object. For instance, the NFS exception will indicate whether a write failed because the file system is exported read-only, or because there is no space left on the disk. While an application that knows it is accessing an NFS file can cast the IOException to an NFSException and obtain the NFS-specific error code, there is yet no file system independent mechanism that will allow an application to obtain a common set of error codes for all file system types.

References

- T. Berners-Lee, L. Masinter, M. McCahill, "Uniform Resource Locators (URL)," RFC-1738, December 1994.
  `http://ds.internic.net/rfc/rfc1738.txt`
- B. Callaghan, "NFS URL Specification," RFC-2244, October 1996.
  `http://ds.internic.net/rfc/rfc2224.txt`

- B. Callaghan, "WebNFS Client Specification," RFC-2054, October 1996.
`http://ds.internic.net/rfc/rfc2054.txt`
- B. Callaghan, "WebNFS Server Specification," RFC-2055, October 1996.
`http://ds.internic.net/rfc/rfc2055.txt`
- A. Chiu, "Authentication Mechanisms for ONC RPC," Internet-Draft: draft-ietf-oncrpc-auth-04.txt, October 1997.
`ftp://ftp.ietf.org/internet-drafts/ draf-ietf-oncrpc-auth-04.txt`
- M. Eisler, A. Chiu, L. Ling, "RPCSEC_GSS Protocol Specification," RFC-2203, September 1997.
`http://ds.internic.net/rfc/rfc2203.txt`
- J. Linn, “Generic Security Services Application Programming Interface, version 2”
`http://ds.internic.net/rfc/rfc2078.txt`
- R. Fielding, "Relative Uniform Resource Locators," RFC-1808, June 1995
`http://ds.internic.net/rfc/rfc1808.txt`
- Sandberg, R., D. Goldberg, S. Kleiman, D. Walsh, B. Lyon, "Design and Implementation of the Sun Network Filesystem," USENIX Conference Proceedings, USENIX Association, Berkeley, CA, Summer 1985. The basic paper describing the SunOS implementation of the NFS version 2 protocol, and discusses the goals, protocol specification and trade-offs. R. Srinivasan, "Binding Protocols for ONC RPC Version 2," RFC-1833, August 1995.
`http://ds.internic.net/rfc/rfc1833.txt`
- R. Srinivasan, "RPC: Remote Procedure Call Protocol Specification Version 2," RFC-1831, August 1995.
`http://ds.internic.net/rfc/rfc1831.txt`
- R. Srinivasan, "XDR: External Data Representation Standard," RFC-1832, August 1995.
`http://ds.internic.net/rfc/rfc1832.txt`
- Sun Microsystems, Inc.®, "Network Filesystem Specification," RFC-1094, DDN Network Information Center, SRI International, Menlo Park, CA. NFS version 2 protocol specification.
`http://ds.internic.net/rfc/rfc1094.txt`
- Sun Microsystems, Inc., "NFS Version 3 Protocol Specification," RFC-1813, DDN Network Information Center, SRI International, Menlo Park, CA.
`http://ds.internic.net/rfc/rfc1813.txt`
- X/Open Company, Ltd.®, X/Open CAE Specification: Protocols for X/Open Internetworking: (PC)NFS, Developer’s Specification, X/Open Company, Ltd., Apex Plaza, Forbury Road, Reading Berkshire, RG1 1AX, United Kingdom, 1991. This is a reference for PC-NFS and accompanying protocols.
- X/Open Company, Ltd., X/Open CAE Specification: Protocols for X/Open Internetworking: XNFS, X/Open Company, Ltd., Apex Plaza, Forbury Road, Reading Berkshire, RG1 1AX, United Kingdom, 1991. This is an indispensable

reference for the NFS and accompanying protocols, including the Lock Manager and the Portmapper.

CHAPTER 4

The XFileAccessor Interface Class and Methods

Overview

The WebNFS Extended File (XFile) API provides a common multiple-filesystem-type access method for Java developers. The XFileAccessor interface, which allows filesystem developers to supply pluggable filesystem implementations in Java, is the back-end counterpart to the XFile API. The XFileAccessor interface is implemented by filesystems that need to be accessed via the XFile API. The XFileAccessor interface features are as follows:

- Support for dynamic filesystem loading
- Existing NFS and native filesystems now conform to the XFileAccessor interface
- Separate implementations of Java and NFS may now be delivered

The XFileAccessor interface strengthens the XFile API strategy, encouraging the growth of filesystem implementations from software partners.

The `XFileAccessor` Interface

The `XFileAccessor` interface is implemented by filesystems that need to be accessed by way of the XFile API. Classes that implement this interface must be associated with a URL scheme that is structured according to the Common Internet Scheme syntax described in `RFC 1738`, an optional location part followed by a hierarchical set of slash-separated directories.

A class file that implements this interface must be named `XFileAccessor` and be installed in a directory named after the URL scheme that it implements. For instance, an `XFileAccessor` that provides file access through the HTTP protocol would be associated with the http URL and its class file would be called:

```
http.XFileAccessor
```

A class prefix is added to this name. The default prefix is `com.sun` and this composite name is located by the `classLoader` using the CLASSPATH. For instance, Sun's NFS `XFileAccessor` is installed as:

```
com.sun.nfs.XFileAccessor
```

The default class prefix `com.sun` can be changed by setting the system property java.protocols.xfile to any desired prefix or list of prefixes separated by vertical bars. Each prefix in the list will be used to construct a package name and the `classLoader` will attempt to load that package by way of the CLASSPATH. This process will continue until the `XFileAccessor` is successfully loaded.

Example

For instance, if you want to use the ftp `XFileAccessor` from Acme, Inc. and the NFS `XFileAccessor` from ABC Inc., then you can set the java.protocols.xfile system property as follows:

```
java.protocols.xfile=com.acme|com.abc
```

When an ftp URL is used, the following package names will be constructed:

```
com.acme.ftp.XFileAccessor
com.abc.ftp.XFileAccessor
com.sun.ftp.XFileAccessor
```

(The default `com.sun prefix` is automatically added to the end of the property list.)

The `classLoader` attempts to load each of the constructed package names in turn relative to the CLASSPATH until it is successful.

A subsequent reference to an NFS URL will result in the following list of candidate package names:

```
com.acme.nfs.XFileAccessor
com.abc.nfs.XFileAccessor
com.sun.nfs.XFileAccessor
```

In this case, the NFS `XFileAccessor` from ABC, Inc. will be loaded in preference to Sun's NFS.

For the `XFile` interface documentation, refer to the XFile javadocs included with this release.

Methods

- `public abstract boolean canRead()`

  Verifies that the application can read from the specified file. Returns true if the file specified by this object exists and the application can read the file; false otherwise.

- `public abstract boolean canWrite()`

  Verifies that the application can write to this file. Returns true if the application is allowed to write to a file specified by this object; false otherwise.

- `public abstract void close() throws IOException`

  Closes this file and releases any system resources associated with the file. After the file is closed, further I/O operations may throw `IOException`. Throws `IOException` if an I/O error has occurred.

- `public abstract boolean delete()`

  Deletes the file specified by this object. If the target file to be deleted is a directory, it must be empty for deletion to succeed. Returns true if the file is successfully deleted; false otherwise.

- `public abstract boolean exists()`

  Verifies that this `XFile` object exists. Returns true if the file specified by this object exists; false otherwise.

- `public abstract void flush() throws IOException`

  Forces any buffered output bytes to be written out. Throws `IOException` if an I/O error has occurred.

- `public abstract XFile getXFile()`

  Returns the `XFile` for this Accessor.

- `public abstract boolean isDirectory()`

  Verifies that the file represented by this `XFileAccessor` object is a directory. Returns true if this `XFileAccessor` object exists and is a directory; false otherwise.

- `public abstract boolean isFile()`

  Verifies that the file represented by this object is a "normal" file. A "normal" file is one that is not a directory and, in addition, satisfies other system-dependent criteria. Any non-directory file created by a Java application is guaranteed to be a normal file. Returns true if the file specified by this object exists and is a "normal" file; false otherwise.

- `public abstract long lastModified()`

  Returns the time that the file represented by this `XFile` object was last modified. It is measured as the time in milliseconds since midnight, January 1, 1970 UTC. Returns the time the file specified by this object was last modified, or 0L if the specified file does not exist.

- `public abstract long length()`

  Returns the length, in bytes, of the file specified by this object, or 0L if the specified file does not exist.

- `public abstract String[] list()`

  Returns a list of the files in the directory specified by this `XFileAccessor` object. Returns an array of file names in the specified directory. This list does not include the current directory or the parent directory ("." and ".." on UNIX systems).

- `public abstract boolean mkdir()`

  Creates a directory, the pathname of which is specified by this `XFileAccessor` object. Returns true if the directory could be created; false otherwise.

- `public abstract boolean mkfile()`

  Creates an empty file, the pathname of which is specified by this `XFileAccessor` object. Returns true if the file could be created; false otherwise.

- `public abstract booleanopen(XFile xf, boolean serial, boolean readOnly)`

  Opens a file in this filesystem. This method is called before any other method. It may be used to open the real file.

  Parameters:

  - *xf* — The `XFile` for the file to be accessed. The URL will be of the form *proto*`://`*location*`/`*path* where *proto* is the name of the filesystem (for example, NFS) and *location* is the filesystem location. For NFS, this is the network name of a server. The *path* is a pathname that locates the file within *location*. As required by RFC 1738, the component delimiters in the pathname are as for URL syntax: forward slashes (`/`) only.
  - *serial* — true if serial access; false if random access.
  - *readOnly* — true if read only; false if read / write.

- `public abstract int read(byte b[], int off, int len, long foff) throws IOException`

  Reads a sub-array as a sequence of bytes. Throws `IOException` if an I / O error has occurred.

  Parameters:

    - *b* — the buffer into which the data is read.
    - *off* — the start offset in the data buffer.
    - *len* — the maximum number of bytes to be read.
    - *foff* — the offset into the file.

    Returns number of bytes read - zero if none.

- `public abstract boolean renameTo(XFile dest)`

  Renames the file specified by this `XFileAccessor` object with the pathname given by the `XFileAccessor` object argument. The destination `XFile` object will be of the same URL scheme as this object. The change of name must not affect the existence or accessibility of this object. Returns true if the renaming succeeds; false otherwise.

  Parameters:

    - *dest* — the new filename.

- `public abstract void write(byte b[],int off, int len, long foff) throws IOException`

  Writes a sub-array as a sequence of bytes. Throws `IOException` if an I/O error has occurred.

  Parameters:

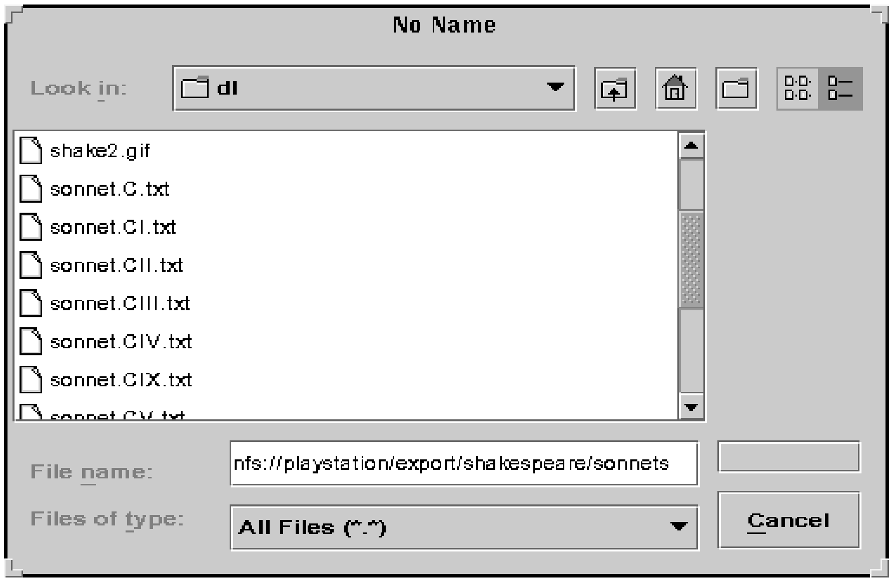
    - *b* — the data to be written.
    - *off* — the start offset in the data in the buffer.
    - *len* — the number of bytes that are written.
    - *foff* — the offset into the file.

CHAPTER 5

XFileChooser

Introduction

A *file chooser* is a standalone graphical component for selecting files. Typically, it's brought up when the user chooses Open or Save from a menu or presses an Open or Save button. A typical file chooser is shown in Figure 5–1.



File name: nfs://playstation/export/shakespeare/sonnets

Figure 5–1 The Default XFileChooser

The Java Swing package provides a file chooser by means of the `JFileChooser` class. Although `JFileChooser` is typically used to allow a user to open or save files, an application may perform a customized action with the selected files instead. The `JFileChooser` does *not* itself perform any action on any file, including opening or saving; instead, it only provides the user with a window for selecting files to be worked with. For each file the user selects, `JFileChooser` returns a `File` object to the calling application, which can then perform the usual file operations on it.

Note - For more on `JFileChooser`, see the `javadocs` for the `JFileChooser` and the section `How to Use File Choosers` in the `Java Tutorial`.

`JFileChooser`, however, is limited to local filesystems. To allow an application to select files on distributed filesystems such as NFS, another file chooser class is required: `XFileChooser`.

The `XFileChooser` class extends the `JFileChooser` class; however, it utilizes the Extended File Application Programming Interface (the XFile API, for short) for its file operations instead of `java.File`. Specifically, it uses `XFile` objects in place of the `File` objects used by `JFileChooser`. Since `XFile` objects include files distributed on NFS networks, an XFileChooser can access exported files anywhere on an NFS network. So, while `JFileChooser` allows you to select files on your own local (or "native") system, for example:

```
/home/marlowe/plays
```

`XFileChooser` can access not only a local file but any exported filesystem on an NFS server, using the syntax specified by the NFS classes used by the Extended File API:

```
nfs://playstation/export/shakespeare/sonnets
```

Note - Although the XFile API theoretically provides access to other kinds of distributed filesystems, including HTTP and CIFS, currently only NFS access is implemented (along with native access).

Because `XFileChooser` is an extension of `JFileChooser`, it inherits the public methods and variables of that class. An application uses `XFileChooser` in virtually the same way it uses `JFileChooser`; in most cases "porting" an application that uses `JFileChooser` to using `XFileChooser` requires only changing the names of methods (for example, from `getSelectedFile()` to `getSelectedXFile()`) and replacing `File` objects with `XFile` objects. In many cases this may be only a few minutes' work.

`XFileChooser` is a Bean

XFileChooser is a Java Bean™. This means that

- It can listen for events from other Beans, and notify them of its own events. See "Events" on page 36 for more information.
- The `XFileChooser` class can be incorporated into a Bean-aware editor, such as `BeanBox`™, by importing the `xfilechooser.jar` file included with this release. The XFileChooser icon will appear on the palette of the editor:

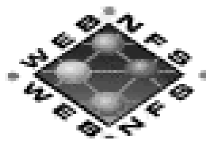


Figure 5–2 The XFileChooser Bean Icon

Here's what the XFileChooserBean looks like when loaded into BeanBox. Note that the Bean suppors a property sheet that allows users to customize the Bean:

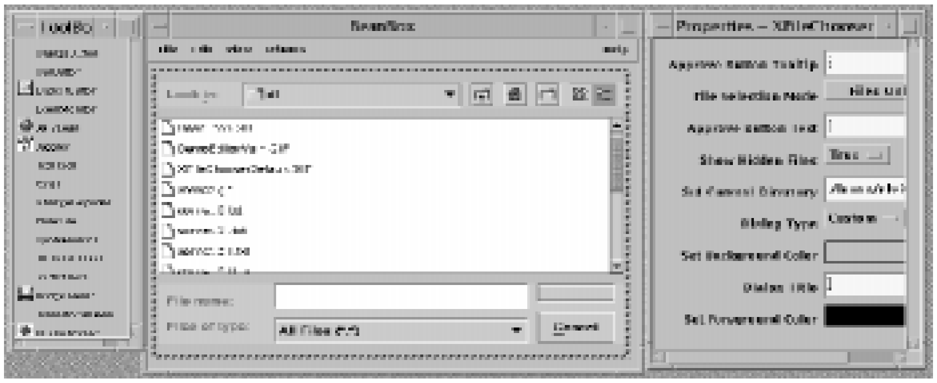


Figure 5–3 The XFileChooser Bean in the BeanBox Editor

Properties that can be modified from the property sheet include: the dialogue type; the dialogue title; the file-selection mode; hidden-file display; text for the "approve" button; mnemonic for the "approve" action; tooltip; look-and-feel; and current directory. See "Properties" on page 36 for a list of `XFileChooser`'s properties.

Architecture

`XFileChooser` inherits from `JFileChooser` in the following way:

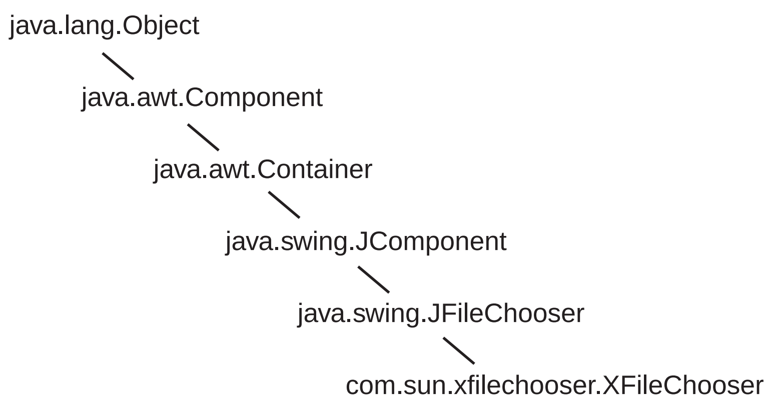


Figure 5–4 The `XFileChooser` Class Hierarchy

The relationship between `XFileChooser`, `JFileChooser`, and the Extended File API is shown in Figure 5–5:

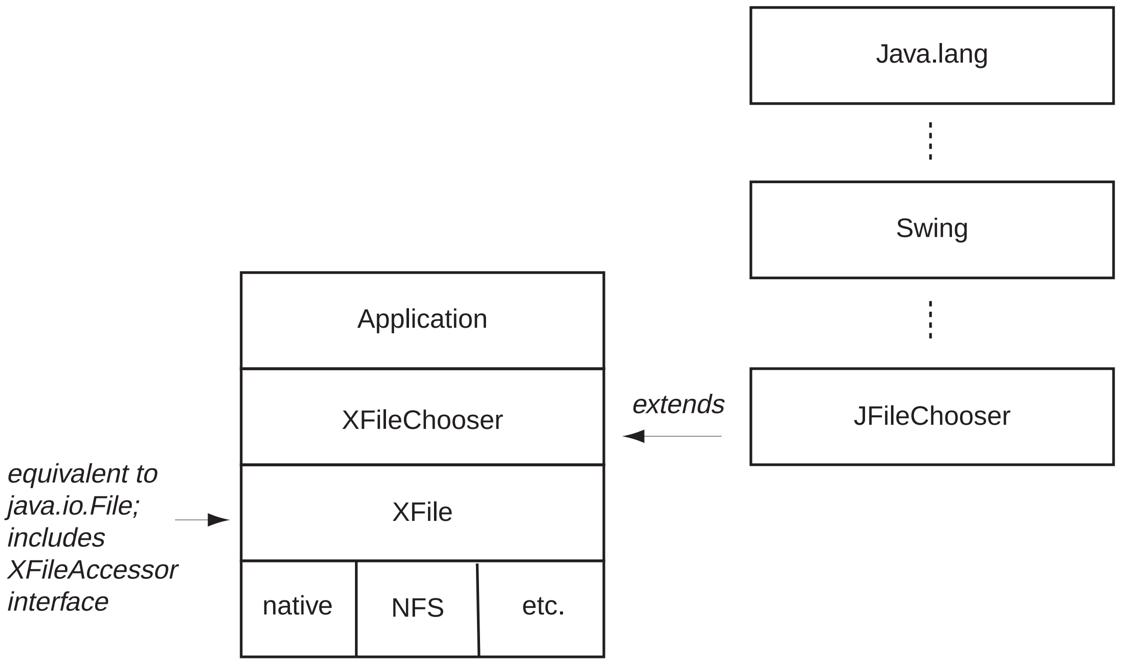


Figure 5–5 `XFileChooser`, `JFileChooser`, and `XFile`

Requirements

The following are required to use the XFileChooser Bean:

Basic Requirements

You must have the Extended File API (from the WebNFS SDK 1.2).

To make use of the entire WebNFS 1.2 FCS release, you must have JDK 1.2 or later installed on your system. Therefore we recommend you install JDK 1.2 if you have not already. It may be found at `http://www.java.sun.com/products/index.html`.

If you are using JDK 1.1.6, you must also have the Swing 1.1 set (available at `http://java.sun.com/products/jfc/index.html#download-swing`) in order to use the XFileChooser. You can use the other WebNFS components without Swing 1.1.

Events

An file chooser can notify other Beans of changes. Changes can be of two types:

- Action events. These alert other Beans if the file chooser's action buttons (Open, Save, Cancel, or a custom button) have been pushed.
- Property changes. These reflect the current state of the XFileChooser Bean. Other Beans can find out if, for example, the file chooser has a newly selected file or if its title bar has changed.

Action Events

`XFileChooser` has two actions for which other Beans can register to listen with `addActionListener()`: `CANCEL_SELECTION`, if the Cancel button is pushed, and `APPROVE_SELECTION`, if the Open or Save (or custom) button is pushed.

Rather than listen for events on these buttons, however, an application can simply bring up a file chooser and then check its return status using one of `XFileChooser`'s properties:

```
Xfilechooser xfc = new XFileChooser;
retval = xfc.show{Open,Save}Dialog(this);
if (retval == xfc.APPROVE_OPTION)
     // open  or save or whatever ...
```

Properties

Other Beans can listen to events in the file chooser by using the `addPropertyChangeListener()` method.

```
public MyListener(XFileChooser xfc) {
     xfc.addPropertyChangeListener(this);
     new PropertyChangeEvent e;
     if e.getPropertyName() == xfc.SELECTED_XFILE_PROPERTY_CHANGED {
          do_something();
...
```

Most of the properties in `XFileChooser` are inherited from `JFileChooser`, so you should refer to the`javadocs` of the `JFileChooser` for a complete list of `JFileChooser` fields. Properties that can be changed through the XFileChooser Bean's property sheet (in a Bean-aware editor) are listed in "`XFileChooser` is a Bean" on page 32.

`XFileChooser` does have three variables unique to itself, however:

- `SELECTED_XFILE_CHANGED_PROPERTY`

  Equivalent to `SELECTED_FILE_CHANGED_PROPERTY` in `JFileChooser`. Indicates that a user has changed a single-file selection.

- `SELECTED_XFILES_CHANGED_PROPERTY`

  Equivalent to `SELECTED_FILES_CHANGED_PROPERTY`. Indicates that a user has changed the selection of multiple files.

- `XDIRECTORY_CHANGED_PROPERTY`

  Equivalent to `DIRECTORY_CHANGED_PROPERTY`. Indicates that the user has changed the working directory.

Classes

As mentioned earlier, `XFileChooser` inherits from `JFileChooser`; check the `javadocs` of the `JFileChooser` for a list of the methods and fields in that class.

The following is a list of the methods unique to `XFileChooser`. For a list of variables, see "Properties" on page 36.

See also the `XFileChooser` javadocs.

Constructors

In addition to creating the file chooser and pointing it to an initial directory, these constructors initialize some of the variables needed for using a Bean editor.

- **XFileChooser()**

  Creates an `XFileChooser` pointing to the user's home directory.

- **XFileChooser(String currentDirectoryPath)**

  Creates an `XFileChooser` pointing to the path specified by *currentDirectoryPath*. If *currentDirectoryPath* is null, the file chooser points to the user's home directory. *currentDirectoryPath* may be an NFS URL as specified by the Extended File API.

- **XFileChooser(Xfile currentDirectory)**

Creates a file chooser that points to the directory specified by the XFile object currentDirectory.

Other Methods

- **ensureFileIsVisible(Xfile xfile)**

  Makes sure that the passed `XFile` shows up in the file list.

- **getCurrentXDirectory()**

  Returns an `XFile` object of the current directory.

- **getSelectedXFile()**

  Returns the `XFile` object of the selected file. The selected file can be set either programmatically, with `setSelectedXfile( )`, or by the user, either by typing the filename in or choosing the file from a list.

- **getSelectedXFileInputStream()**

  Returns an `XFileInputStream` for the currently selected file.

- **getSelectedXFileOutputStream()**

  Returns an XFileOutputStream for the selected file.

- **getSelectedXFiles()**

  Returns an array of `Xfile` objects corresponding to the selected files.

- **propertyChange(PropertyChange e)**

  This method gets called when certain bound properties have changed on an associated `XFileChooser`. The properties it listens to include `XDIRECTORY_CHANGED_PROPERTY`, `SELECTED_XFILE_CHANGED_PROPERTY`, and `SELECTED_XFILES_CHANGED_PROPERTY`. This method is needed to create the corresponding `XFile` for the `File` object of the selected file or directory.

- **setCurrentXDirectory(Xfile currentDirectory)**

  Sets the current directory. Passing a null value sets the file chooser to point to the user's home directory. If currentDirectory is not actually a directory, its parent directory will be used as the current directory. If the parent is not traversable, then it will walk up the parent tree until it finds a traversable directory, or hits the root of the filesystem.

- **setSelectedXFile(XFile xfile)**

  Sets the XFile object of the selected file. If the file's parent directory is not the current directory, it changes the current directory to the parent directory (if it is traversable). If the parent is not traversable, then it will walk up the parent tree until it finds a traversable directory, or hits the root of the filesystem.

- **setSelectedXFiles(XFile selectedFiles[])**

Same as `setSelectedXFile()`, except that it sets a list of files, if the file chooser is set to allow multiple-file selection.

Customizing the File Chooser

A file chooser can be customized in a number of ways. (Several of these modifications are shown in Figure 5–7.) Most of the modifications are of the sort that one normally sees with GUI components, such as changing the string in the file chooser's title bar or setting button mnemonics. Certain customizations are more specific to the file chooser, however, and deserve special mention.

Performing a Unique Action

Although the most common use for a file chooser is to open or save files, `XFileChooser` allows you to select a file for any other action you specify. To this end, a "generic" method for bringing up a file chooser window, `showDialog()`, is provided in addition to `showOpenDialog()` and `showSaveDialog()`. (Since the file chooser itself doesn't affect files, the only difference between a custom dialog and the others is the title on the dialog window and the label on the "accept" button.)

File Filtering

As with the JFileChooser, three types of file filtering are available to the application in bringing up a file chooser:

- Built-in filtering. Currently this is limited to showing hidden files or not (using the `setFileHidingEnabled()` method). Since the underlying Extended File API does not handle invisible files, this filter only checks to see if the filename begins with a '.' (period), signifying a hidden file on UNIX systems.
- Application-controlled filtering, whereby the file chooser shows only files that the filter — some instance of `FileFilter`^{api}) — allows. For example, you may want your application to only open files with a particular suffix.
- User-choosable filters. In this scenario, the file chooser presents the user with selection of filters to apply to the file chooser. An example of this is shown in"Filtering Files" on page 45.

Setting the FileView

The FileView is the way each file is shown in the file chooser's list of files. By default a small icon and the file's name appear; this can be customized, for example, to show a unique icon next to each filename. An example of this is shown in "Modifying the FileView" on page 46.

Setting the Accessory

An *accessory* is a component that is invoked in the file chooser when a file is selected. One example of an accessory might be a panel with controls in it. Another might be a small preview area in the file chooser that displays a thumbnail of a selected graphic file. An example of this kind of customization is given in "Setting the Accessory" on page 47.

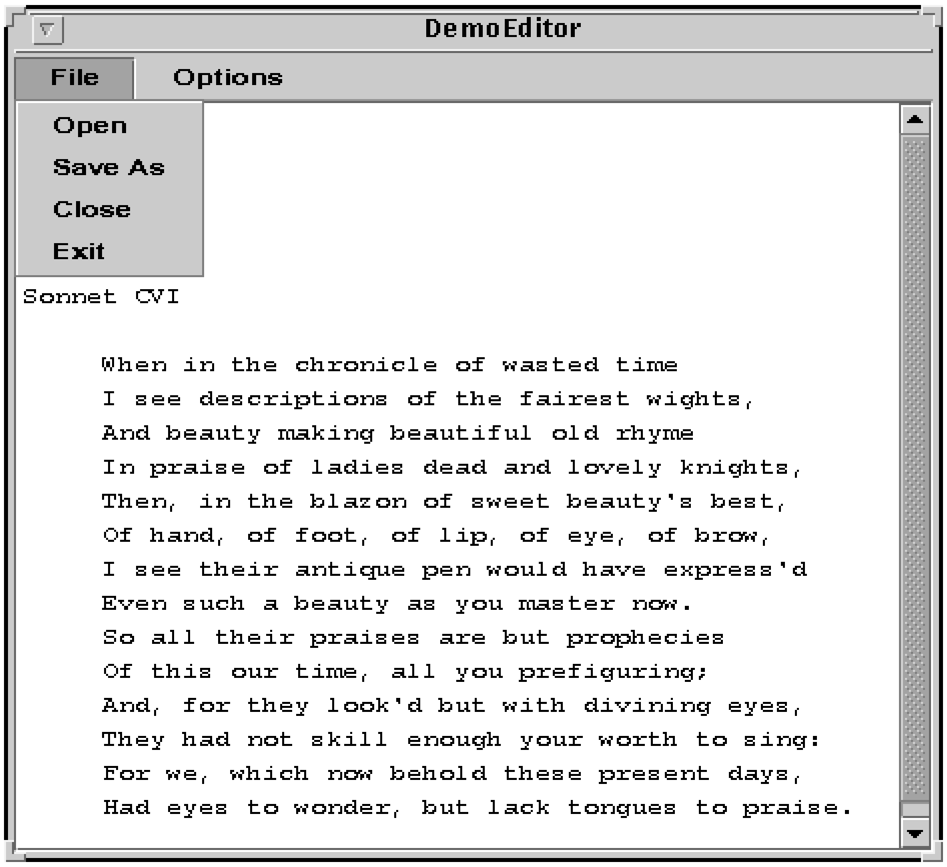
Sample Programs

Two sample programs that make use of `XFileChooser` are included with this release:

- The first, `DemoEditor`, found in Appendix C, is a simple file editor that uses the `XFileChooser` in selecting a file for editing.
- The second, `XFileChooserDemo`, found in Appendix D, is a mock application that shows how a file chooser may be customized.

The `DemoEditor` Sample Program

The DemoEditor program allows a user to open a text file and perform simple edits on it (Cut, Paste, etc.). The file can then be saved to another name.



```
Sonnet CVI

     When in the chronicle of wasted time
     I see descriptions of the fairest wights,
     And beauty making beautiful old rhyme
     In praise of ladies dead and lovely knights,
     Then, in the blazon of sweet beauty's best,
     Of hand, of foot, of lip, of eye, of brow,
     I see their antique pen would have express'd
     Even such a beauty as you master now.
     So all their praises are but prophecies
     Of this our time, all you prefiguring;
     And, for they look'd but with divining eyes,
     They had not skill enough your worth to sing:
     For we, which now behold these present days,
     Had eyes to wonder, but lack tongues to praise.
```

Figure 5–6 The `DemoEditor` Program

As part of the initialization of the application,

1. menu items are created for opening, closing, and saving a file, and for exiting;
2. listeners are added to the menu items;
3. the menu items are attached to the File menu.

For brevity's sake, only the code for opening files is shown below:

```
JMenu filemenu = new JMenu(‘‘File’’);
openItem = new JMenuItem(‘‘Open’’);
openItem.addActionListener(this);
fileMenu.add(openItem);
```

If an item on the File menu is selected, an *actionEvent* is fired off which causes the `actionPerformed()` method to be called. `actionPerformed()` checks the type of menu item selected and acts accordingly: it creates an `XFileChooser` object, and, if Open or Save is selected from the menu, brings up a file chooser.

```
public void actionPerformed(ActionEvent ae) {
   int retval;
   XFileChooser chooser = new XFileChooser();
   if (ae.getSource == closeItem)
     closeDocument();
   } else if (ae.getSource() == openItem) {
      retval = chooser.showOpenDialog(this);
      if (retval == XFileChooser.APPROVE_OPTION) {
        XFile theFile = chooser.getSelectedXFile();
   // make sure we're not already editing a file!
        closeDocument();
        if (theFile != null)
          readFromFile(theFile);
     }
   } else if (ae.getSource() == saveItem) {
     retval = chooser.showSaveDialog(this);
     if (retval == XFileChooser.APPROVE_OPTION) {
        XFile theFile = chooser.getSelectedXFile();
        if (theFile != null)
          writeToFile(theFile);
        }
   } else if (ae.getSource() == exitItem) {
     System.exit(0);
   }
}
```

Some things to note about the foregoing:

- All the file I/O involves `XFile` objects, not `File` objects. The `XFileChooser` method `getSelectedXFile()` is used in place of the `JFileChooser` method `getSelectedFile()`.

  `readFromFile()`, which reads from a file into the Editor, takes an `XFile` as an argument, and in turn calls `XFileInputStream()`, which is the Extended File API's equivalent to the `java.lang.io` method `FileInputStream()`:

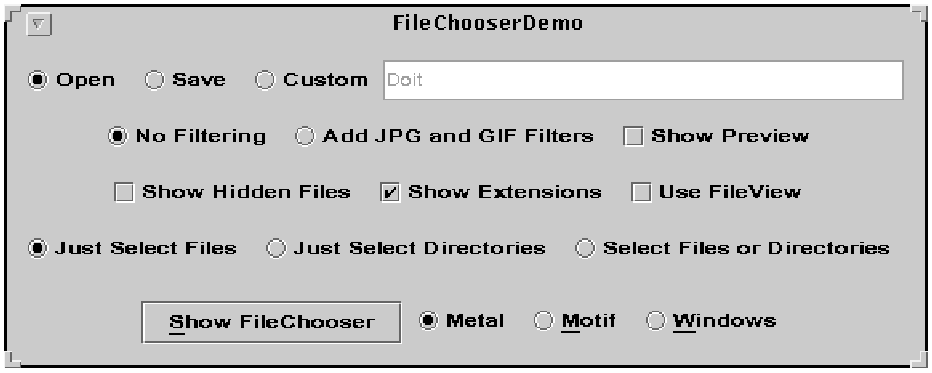
```
public void readFromFile(XFile file) {
     Document doc = jtp.getDocument();
     try {
          XFileInputStream is = new XFileInputStream(file);
          String ins = retrieveAsString(is);
          doc.insertString(0, ins, stdoutSet);
...
```

- The same file chooser is instantiated for both Open and Save As. This has two advantages: first, the chooser remembers the current directory between uses, so that the Open and Save As dialogue windows automatically share the same directory. Second, if you choose to customize the file chooser, you only have to customize it once; your customizations apply to both the Open and Save versions of it.
- The actual dialogue box (shown in Figure 5–1) is brought up by the call to either `showOpenDialog()` or `showSaveDialog()`, both of which are inherited from `JFileChooser`. (*APPROVE_OPTION* is also inherited from `JFileChooser`.)

The `XFileChooserDemo` Sample Program

The `XFileChooserDemo` program is a mock application designed to show the different ways an XFileChooser can be customized.

Note - Since this program is just a modification of the sample program `FileChooserDemo2` shown in the Java Tutorial, much of the description has been left out. Please see `How To Use File Choosers` for a description of that program.



FileChooserDemo

[✓ Open] Save Custom Doit

[✓ No Filtering] Add JPG and GIF Filters Show Preview

Show Hidden Files [✓ Show Extensions] Use FileView

[✓ Just Select Files] Just Select Directories Select Files or Directories

Show FileChooser [✓ Metal] Motif Windows

Figure 5–7 The `XFileChooserDemo` Sample Program

When the settings are as shown in Figure 5–7, a default file chooser (shown in Figure 5–1), will pop up when the Show File Chooser button is pressed.

This sample program differs from `DemoEditor` in several aspects:

First, the file chooser that comes up has been customized in three different ways:

- A user-configurable filter has been added; that is, the user can choose to filter the file list shown in the file chooser.
- The FileView — the way that each file appears in the list, depending on its type — has been modified to display a custom icon next to each file name.
- An Accessory component has been provided for the files displayed. In this case, it displays a preview of a selected file if the file is a graphic file.

(See “Customizing the File Chooser” on page 39 for information on other ways a file chooser may be customized.)

The second way that this program differs from `DemoEditor` is that the file chooser is invoked not from a menu but from a single button, the Show File Chooser button. Which kind of file chooser is brought up — Open, Save, or a custom operation — depends upon the settings in the main window.

Finally, unlike `DemoEditor`, the look-and-feel of this sample program is changeable by the user. The file chooser inherits the look-and-feel of its parent application.

Determining Which File Chooser to Bring Up

The DemoEditor program had separate menu items for opening and saving a file, so that a different file chooser would appear, depending on the item selected. (Remember, however, that the same XFileChooser object is instantiated in both cases.) Things are different with the XFileChooserDemo program: There's a single button, Show File Chooser, and which dialogue this button brings up depends on which radio button (Open, Save, or Custom) in the main window has been selected. The code for using the Open button is shown below.

```
// Create an 'Open' radio button
openButton = new JRadioButton(''Open'')
...
openButton.addActionListener(optionListener);
// Create 'Show File Chooser' button
button = new JButton(''Show File Chooser'');
button.addActionListener(this);
```

When a radio button is selected, OptionListener( ) is called. If the Open radio button is selected, the OptionListener( ) sets the file chooser to be an Open file chooser with the setDialogType( ) method:

```
class OptionListener implements ActionListener {
   public void actionPerformed(ActionEvent e) {
     JComponent c = (JComponent) e.getSource();
     if (c == openButton) {
        chooser.setDialogType(XFileChooser.OPEN_DIALOG);
...
```

OptionListener( ) does the same sort of thing when Save is chosen, except that the dialogue type is set to SAVE_DIALOG. Note, by the way, that setDialogType( ) and the OPEN_DIALOG field are inherited from JFileChooser.

In the DemoEditor program, the actionPerformed( ) method chose between bringing up a file chooser with showOpenDialog( ) or showSaveDialog( ), depending on whether Open or Save was selected off the File menu. Since the dialog type is already set here, it is not necessary to differentiate between them in actionPerformed( ). Instead, a "generic" method, showDialog( ), is used to bring up the file chooser:

```
public void actionPerformed(ActionEvent e) {
   int retval = chooser.showDialog(frame, null);
...
```

Another difference to note is that the XFileChooser is not instantiated in actionPerformed( ), as it was in the DemoEditor program. Instead, it's instantiated at the class level so that the dialog type can be set in OptionListener( ). In both programs, however, a single file chooser is instantiated for all operations (Open, Save, or custom).

Filtering Files

As mentioned in “Customizing the File Chooser” on page 39, three different file filters can be applied to a file chooser. In this program, a filter chosen by the user is enabled. If the Add JPEG and GIF Filters radio button is selected, the resulting file chooser presents the user with several filters to choose from, as shown in Figure 5–8:

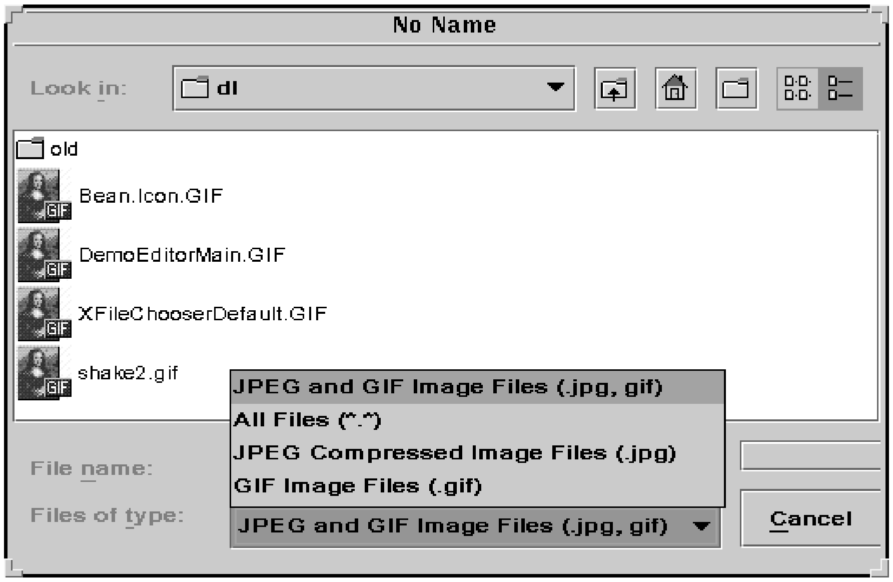


Figure 5–8 Filtering Files

The file filters used by `XFileChooserDemo` are described in the class `ExampleFileFilter`, found in `ExampleFileFilter.java` (shown in “ExampleFileFilter” on page 77). `ExampleFileFilter` takes two arguments: a file extension (for example, `.jpg`) and a description of the file, and filters files appropriately.

Here is where `XFileChooserDemo` instantiates the various filters:

```
jpegFilter = new ExampleFileFilter(‘‘jpg’’, ‘‘JPEG
Compressed Image Files’’);
gifFilter = new ExampleFileFilter(‘‘gif’’, ‘‘GIF
Image Files’’);
bothFilter = new ExampleFileFilter(new String[] ‘‘‘‘jpg’’, ‘‘gif’’}, ‘‘JPEG
and GIF Image Files’’);
```

The program’s `OptionListener()` checks to see if the radio buttons governing filters have been changed. If the user has selected the Add JPEG and GIFs button, `OptionListener()` adds the various filters provided by ExampleFileFilter, using the `addChoosableFileFilter()` method inherited from `JFileChooser`:

```
class OptionListener implements ActionListener {
   public void actionPerformed(ActionEvent e) {
      JComponent c = (JComponent) (e.getsource();
...
        else if (c == addFiltersButton) {
           chooser.addChoosableFileFilter(bothFilter);
              chooser.addChoosableFileFilter(jpgFilter);
           chooser.addChoosableFileFilter(gifFilter);
        }
```

(The program could have used finer control in determining which filters to add to the file chooser, but the approach used here minimizes the number of radio buttons cluttering the main window.)

When the user chooses a new filter in the file chooser, a `CHOOSABLE_FILE_FILTER_CHANGED_PROPERTY` is fired off, and the panel displaying the list of files gets updated accordingly, to show only the filtered files.

Modifying the FileView

The *FileView* is the way a file in a list is presented. By default, a file chooser shows each file with a small icon showing whether the file is a directory of a flat file. An example of a modified FileView is seen in Figure 5–8; compare the icon shown next to the file `shakes2.gif` to the icon that appears in next to it in Figure 5–1.

Setting the FileView works in the same way as filtering files does. Here's how it's done in `XFileChooserDemo`:

1. Create a custom subclass of the class `FileView`api, overriding any of its abstract methods you need. For this sample program, one has been created called `ExampleFileView` (see "ExampleFileView" on page 80), which looks at a file's extension (for example,`.jpg`) and returns the file's name and, if the extension indicates that the file is a graphic file, an icon representing that type of file.
2. Create a `FileView` object.

```
fileView = new ExampleFileView();
```

3. Add a checkbox for changing the FileView, and add a listener to it.

```
useFileViewButton = new JCheckbox(''Use FileView'');
useFileViewButton.addActionListener(optionListener);
```

4. In `optionListener( )`, call `setFileView( )` when the Use FileView checkbox gets selected.

```
if (c == useFileViewButton) {
     if (useFileViewButton.isSelected() {
          chooser.setFileView(fileView);
     else
                chooser.setFileView(null);
```

Now when a file chooser is brought up `ExampleFileView` will be used to display the files in the file list.

Setting the Accessory

The XFileChooserDemo's *accessory component* is a small panel on the right side of the file chooser that displays a preview of GIF and JPEG graphics files. In Figure 5–9 it's the picture of William Shakespeare:

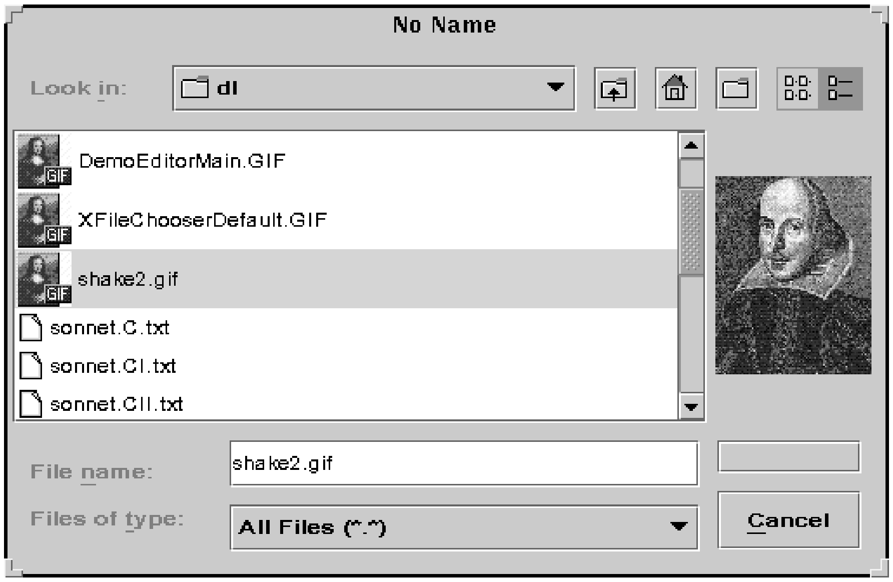


Figure 5–9 The XFileChooserDemo's Accessory

Another use for an accessory is a panel with more controls in it (say, checkboxes to toggle certain features).

Two aspects of accessories are important:

- An accessory can be any object that inherits from JComponent.
- The accessory should implement either paint( ) or paintComponent( ).
- The component should have a preferred size that looks good in a file chooser.

The XFileChooserDemo program begins by creating the file chooser; the accessory (here called the previewer), which is declared to be a FilePreviewer object; and a related checkbox. It adds a listener (ActionListener) to the checkbox, as we've seen with the file view and file filtering:

```
chooser = new XFileChooser;
...
previewer = new FilePreviewer(chooser);
...
accessoryButton = new JCheckBox(‘‘Show Preview’’);
accessoryButton.addActionListener(optionListener);
```

The OptionListener( ) checks to see if the Show Preview checkbox is checked; if it is, it calls setAccessory( ):

```
if (c == accessoryButton) {
     if (AccessoryButton.isSelected()) {
          chooser.setAccessory(previewer);
     else
          chooser.setAccessory(null);
```

The previewer itself just sets its preferred size and adds a `PropertyChangeListener()` that listens for changes to the file chooser's state.

This is its only method, its constructor:

```
public FilePreviewr(XFileChooser fc) {
     setPreferredSize(new Dimension(100, 50));
     fc.addPropertyChangeListener(this);
```

If a property is changed, `addPropertyChangeListener()` calls the abstract method `PropertyChange()`. Here's how `PropertyChange()` defined for the `XFileChooserDemo` program. It first checks to see what property has changed; if a new file has been selected, then a new preview is required.

```
public void propertyChange(PropertyChangeEvent e) {
     String prop = e.getPropertyName();
     if (prop.equals(XFileChooser.SELECTED_XFILE_CHANGED_PROPERTY)) {
          f = (File) e.getNewValue()
          if (isShowing()) {
               loadImage();
               repaint();
          }
     }
}
```

APPENDIX **A**

Frequently Asked Questions

Overview

This appendix covers the following questions:

The WebNFS and Extended Filesystem API FAQ

Answers to some commonly asked questions about WebNFS:

Why Do I Get "Permission Denied" Errors When I Try to Read My Own Files?

The Java NFS client defaults the user's credential to `nobody`,that is, a user-id and group-id of 60001. This identity is fine if the user is accessing files in a public archive, but if the files have access restrictions enforced by the file permissions then access will be denied unless a valid credential is used. The class `XFileExtensionAccessor` in `com.sun.nfs` supports a method called `loginPCNFSD()` that passes the user's login name and password to the `PCNFSD` service on a server that is running the `PCNFSD` daemon. If the loginname and password are correct then a valid credential with the user's UID and GID is constructed and used in all future NFS calls.

When I Create a File Or Directory over NFS, Why Is It Owned by `nobody`?

You are using the default credential. You need to obtain a valid credential for your loginname and password so that the server can identify you as the owner of the file. (See previous question.)

Why Won’t My NFS URL Work with a Non-WebNFS Server?

If the Java NFS client cannot evaluate the URL using the WebNFS technique, it will fall back to using the MOUNT protocol. While WebNFS paths are evaluated relative to the directory associated with a public filehandle, MOUNT paths are absolute. For UNIX servers this means that the path must start with a slash. The slash in the NFS URL that separates the server name from the pathname is not considered to be a part of the pathname. To have the pathname begin with a slash you need to add a second slash, e.g. if mounting /export/home from the non-WebNFS server dopey you use the following path:

```
nfs://dopey//export/home
```

Can I Use an NFS Proxy Server?

No, not directly. The HTTP protocol is designed for proxy use, but protocols like FTP and NFS cannot be proxied directly; that is. you cannot send an NFS request to a proxy server with instructions to forward the request to (and reply from) another NFS server. However, as with FTP, NFS can be proxied by an NFS-capable HTTP proxy server if you need to be able to browse directories graphically or transfer files.

How Do I Access an Internet NFS Server Through My Firewall?

If the NFS server supports WebNFS and TCP connections for NFS, then have your firewall administrator provide access for outgoing connections to port 2049. If the server does not support WebNFS, then access will be much difficult since the client will have to use the MOUNT protocol to obtain an initial filehandle; however, this protocol does not use a “well-known” port. This makes it almost impossible to configure the firewall to allow NFS mounts unless the firewall software supports some kind of portmap protocol snooping.

Why Can’t I Access Files on a Digital UNIX Server?

If the server doesn’t support WebNFS then the client will use the MOUNT protocol. Some mount daemons will reject mount requests if they come from a client that cannot be resolved through a reverse DNS lookup or if the client is using an unprivileged source port (not less than or equal to1024). Digital Unix server administrators should check the settings of the mountd’s –i and –n options.

How Do I Browse the Exported Filesystems of a Server?

If you have a URL to an exported filesystem, you can browse the exported filesystem by listing directories and subdirectories. However, if you want to discover the exported filesystems on an NFS server, only the `MOUNT` protocol can do that. On UNIX clients you can use the `showmount` or `dfshare`" commands to obtain a list of exported filesystems from an NFS server.

How Can I Tell If a File Is "Hidden"?

`XFile` has no method to determine the "hiddenness: of a file; that is, there is no `isHidden()` method. It is difficult to implement for native filesystems because `java.io.*` provides no access to the "hidden" bit on DOS/Windows filesystems. NFS clients typically recognize hidden files using the UNIX convention of an initial dot in the name, though this is just a convention and not supported directly by the NFS protocol. In summary, there is no way to tell if a DOS/Windows file is hidden, but an initial dot is a clue to the "hiddenness" of a UNIX file.

What Kind of Cacheing Does the Java NFS Client Use?

Its cache model is quite similar to other NFS client implementations. File and directory data are cached from between 3 and 30 seconds, depending on the frequency of file or directory modification. That is, if the file is being changed every 10 seconds then the cache time will be set to 10 seconds, but if the file or directory hasn't changed for 6 months then it'll be cached for up to 30 seconds. When the cache time expires, the client will revalidate the cached data via a `GETATTR` request to the server. If the file or directory on the server is changed, then the cached copy is invalidated. The client uses the attributes returned by NFS v3 `READ`, `WRITE`, and `LOOKUP` calls avoiding unnecessary `GETATTR` revalidations.

To avoid hogging the JVM heap, the client avoids aggressive caching of file data. Only the current, read-ahead or write-behind buffers are cached. This has no impact on sequential file access, but could cause performance problems for applications that frequently re-read files exceeding a buffer in size (8k for v2, 32k for v3). The list of names in an NFS directory is cached when the directory is first read. If the server is read using NFS version 3 then the client will use the `READDIRPLUS` request and obtain the filehandles and attributes of all the directory entries. Note that a file tree walk or other procedure that touches lots of NFS files will eventually run the client out of memory. This memory limitation will be fixed in future versions of the client.

Using `XFileOutputStream`, I Write Data, But It Never Appears in the File. Why?

This is probably due to the write buffering. When you write data to an NFS server using `XRandomAccessFile` or `XFileOutputStream` the writes are buffered. When the buffer becomes full (NFS v2 8k, NFS v3 32k) the data is written to the server. If the Java application exits without calling the `close()` method then the last incomplete buffer will not be written to the server. It is important to call `close()` when you have finished writing data to a file!

How Do I Create or Read Symbolic Links?

The XFile API does not support symbolic links and the NFS `ExtensionAccessor` does not yet provide any methods that allow symbolic links to be read or created. The Java NFS client automatically follows symbolic links so that a URL that references a symbolic link will eventually result in a reference to a file or directory. The Java NFS client will correctly evaluate a symbolic link that contain an NFS URL. Note that the client does not yet detect symbolic link loops - a stack overflow will result if a symlink loop is entered.

Can I Detect a Symbolic Link?

No. Any reference to a symbolic link will cause it to be followed and the attributes of the link destination will be returned. A future release of the NFS client will add the ability to detect symbolic links to the NFS `ExtensionAccessor`.

Can I Control the Client's Read Size and Write Size?

Most client implementations of NFS allow the configuration of read and write sizes. Normally an NFS version 2 client reads and writes files in 8k chunks, however some routers may drop UDP packets of this size and a smaller transfer size will yield better results. The Java NFS client provides no interface for setting transfer size. Transfer size is negotiated with the server.

How Do I Lock a File Using the Extended Filesystem API?

The Extended Filesystem API mirrors the file access functions available through the Java I/O package java.io. Neither package supports the ability to lock files. NFS clients generally lock files using the companion NLM (Network Lock Manager) protocol but this is not currently implemented in the Java NFS client. Where locking is not supported directly, applications can use a lock file as a substitute.

Where Can I Find Out More about NFS?

Version 2 and version 3 of the NFS protocol are described in RFC 1094 (version 2) (ftp://ftp.isi.edu/in-notes/rfc1094.txt) and RFC 1813 (version 3) (ftp://ftp.isi.edu/in-notes/rfc1813.txt). The WebNFS method of binding to the server is described in RFC 2054 (ftp://ftp.isi.edu/in-notes/rfc2054.txt) (client side) and RFC 2055 (ftp://ftp.isi.edu/in-notes/rfc2055.txt) (server side). The NFS URL scheme is specified in RFC 2224 (ftp://ftp.isi.edu/in-notes/rfc2224.txt). The X/Open Specification for NFS including descriptions of the MOUNT protocol and Network Lock Manager protocol can be ordered from The Open Group (`http://www.opengroup.org/publications/catalog/c525.htm`). If you are interested in comparing the performance of NFS servers from various vendors, take a look at the SPEC SFS Suite benchmark (`http://www.specbench.org/osg/sfs`). NFS vendors also test their client and server implementations for interoperability at Connectathon. For information on the administration of NFS on Solaris clients and servers, check the *NFS Administration Guide*. If you would like to communicate with other NFS users, administrators, or developers, try the Newsgroup `comp.protocols.nfs`.

How Do I Obtain the File Separator Character Using XFile?

The path separator for all URL schemes except “native” is a forward slash (“/”). If you are using native paths, the separator can be obtained via System properties. For example:

```
String s = System.getProperties().getProperty(‘‘file.separator’’)
```

Do I Have to Type a URL to Access Any Non-Native File?

There is some concern that URL naming could be awkward due to having to type the colons and multiple slashes typical of URL syntax. However, in an application context, as with Web browsers, users can operate within a naming context and use relative names. For instance, most URLs embedded in HTML documents use short, relative URLs. Similarly, when a user is selecting a file from a dialog window that contains the list of files in a directory, the "base" URL is the directory being displayed and the user can type, or click on, a relative name. Relative paths can also be used to name files in any subdirectory and the use of ".." components can be used to name files in parent directories. A full URL must be entered only to select a file on a different server or to name a file with a different URL scheme. For a description of relative naming using URLs see RFC 1808.

What Happens If a Relative Name Looks Like a URL?

An ambiguity could arise if a relative name looks like a URL. For instance, if the current directory of the Java Runtime contained a directory `nfs:` then a reference to a sub-directory `nfs:/xxx/yyy` might be intended, but the path would be evaluated as an NFS URL. If a filesystem that implements the URL scheme is currently loaded then the evaluation will fail and an error will be returned. To avoid this ambiguity the file must be identified by its absolute path,that is, the complete URL for the file. A short reference to the "native" scheme is `.` (period), so that a path beginning with `.:` is identified as a URL for a native path. If the apparent URL cannot be identified (no implementation available) then the name will be evaluated correctly within the directory indicated by the current URL context. Fortunately, file names with an appended colon are not popular. The only exception is the DOS disk letter naming, though most URL schemes use an identifier longer than a single letter — so ambiguity is unlikely to be a problem.

How Do I Get or Set Attributes with XFile?

XFile supports access to the file attributes supported by `java.io.File`, for instance: `canRead`, `canWrite`, `isFile`, `isDirectory`, `lastModified`, and `length`; however, as yet there is no mechanism for obtaining filesystem-specific attributes, for instance, the UID of an NFS file. Access to these attributes will be provided in future releases within the `XFile ExtensionAccessor` classes.

How Do I Write an Accessor Class for a "Plug-in" Filesystem?

Refer to the WebNFS XFileAcessor Interface Reference which describes how to write an accessor class to support any filesystem scheme.

Why Isn't `XFile` a Subclass of `File`?

This would seem to make sense, particularly where a `File` object is used as an argument in the constructor for instances of `FileInputStream()` or `RandomAccessFile()`. Unfortunately, nearly all the methods in the `File` class interact directly with the JNI - it is not possible to interpose `XFile` as a filesystem-independent layer. In addition, classes that take a `File` in the constructor also work directly with the JNI. Eventually we expect `XFile` to be subsumed by an Extended Filesystem API in a future release of the Java JDK.

APPENDIX **B**

Simple XFile Sample Programs

Overview

Note that it is not important whether the source and destination files for these programs are local or remote.

Copying a File

This program does a simple file copy.

Usage:

% **java xcopy** ***src dest***

CODE EXAMPLE B–1

```
import java.io.*;
import com.sun.xfile.*;
class xcopy {
     public static void main(String av[]) {
          try {
               String srcFile = av[0];
               String dstFile = av[1];

               XFileInputStream  in  = new XFileInputStream(srcFile)
               XFileOutputStream out = new XFileOutputStream(dstFile);
```

(continued)

(Continuation)

```
                int c;
                byte[] buf = new byte[32768];
                long elapsedtime = System.currentTimeMillis();
                int filesz = 0;

                while ((c = in.read(buf)) > 0) {
                     filesz += c;
                     out.write(buf, 0, c);
                     System.out.print("" + filesz);
                }

                System.out.println();
                in.close();
                out.close();

                elapsedtime = System.currentTimeMillis() - elapsedtime;
                int rate = (int) (filesz / (elapsedtime / 1000.0) / 1024);
                System.err.println(filesz + " bytes copied @ " + rate + "Kb/sec");
           } catch (IOException e) {
                     System.err.println(e);
           }
      }
}
```

Recursive Copy

This program takes a file or a directory hierarchy and copies it somewhere else.

Usage:

% **java rcopy** ***src dest***

CODE EXAMPLE B–2

```
import java.io.*;
import com.sun.xfile.*;
class rcopy {
     static void copy(XFile src, XFile dst) throws IOException {
          if (src.isDirectory()) {
               dst.mkdir();
```

(continued)

(Continuation)

```
                String[] dirList = src.list();
                for (int i = 0; i < dirList.length; i++)
                     String entry = dirList[i];
                     copy(new XFile(src, entry), new XFile(dst, entry));
                }
           } else {                  // assume it’s a file
                XFileInputStream in  = new XFileInputStream(src);
                XFileOutputStream out = new XFileOutputStream(dst);

                int c;
                byte[] buf = new byte[32768];

                while ((c = in.read(buf)) > 0)
                     out.write(buf, 0, c);

                in.close();
                out.close();
           }
      }

      public static void main(String av[]) {
           try {
                copy(new XFile(av[0]), new XFile(av[1]));
           } catch (IOException e) {
                System.err.println(e);
                     e.printStackTrace();
           }
      }
}
```

APPENDIX **C**

The DemoEditor Sample Program

The DemoEditor Sample Program

The DemoEditor demonstrates a use of the `XFileChooser` object. It brings up a window with a File Menu and Option Menu. The File Menu allows you to open, save, and close a file. The `XFileChooser` object comes up when you do an Open or a Save. Figure 5–6 shows what the DemoEditor program looks like; “The `DemoEditor` Sample Program” on page 40 explains this program at length.

To compile the DemoEditor demo on Solaris, type as follows:

```
% javac -classpath ../../xfilechooser.jar DemoEditor.java
```

To run the compiled bytecode:

```
% java -classpath ../../xfilechooser.jar:. DemoEditor
```

If compiling or running on Windows, use backslashes in the path, and semicolon as separator, e.g.:

```
% java -classpath ..\..\xfilechooser.jar;. DemoEditor
```

Note - You must be using the java in JDK1.2,; thus set your path accordingly. If you are using a Java2 release prior to JDK 1.2.1 you may need the JFileChooser patch to fix bug 4169763. **The XFileChooser release notes describe how to use the the Xbootclasspath to include the patch.**

CODE EXAMPLE C–1 The DemoEditor Sample Program

```
/*
 * @(#)DemoEditor.java 1.2 99/04/18
 *
 * Copyright 1999 by Sun Microsystems, Inc.,
 * 901 San Antonio Road, Palo Alto, California, 94303, U.S.A.
 * All rights reserved.
 *
 * This software is the confidential and proprietary information
 * of Sun Microsystems, Inc. ("Confidential Information").  You
 * shall not disclose such Confidential Information and shall use
 * it only in accordance with the terms of the license agreement
 * you entered into with Sun.
 */

/*
 * DemoEditor.java
 * A text editor which demonstrates how to use the XFileChooser.
 * The Editor can save and restore a text file full of simple styles
 * and have some simple options (e.g. cut, paste, copy)
 */

import java.awt.*;
import java.awt.event.*;
import java.io.*;
import javax.swing.*;
import javax.swing.text.*;
import com.sun.xfile.*;
import com.sun.xfilechooser.*;

public class DemoEditor extends JFrame implements ActionListener {

    JTextPane jtp = new JTextPane();
    JScrollPane jsp = new JScrollPane(jtp);
    ColoredAttributeSet stdoutSet;
    JMenuItem saveItem, openItem, exitItem, closeItem;
    JMenuItem cutItem, pasteItem, copyItem;

    public DemoEditor() {
 super("DemoEditor");

 addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
  System.exit(0);
     }});

 setSize(600,500);

 jtp.setCaretColor(Color.magenta);

 JMenuBar jmb = new JMenuBar();
 JMenu fileMenu = new JMenu("File");
 saveItem = new JMenuItem("Save As");
 saveItem.addActionListener(this);
 openItem = new JMenuItem("Open");
 openItem.addActionListener(this);
 exitItem = new JMenuItem("Exit");
 exitItem.addActionListener(this);
 closeItem = new JMenuItem("Close");
 closeItem.addActionListener(this);


 fileMenu.add(openItem);
 fileMenu.add(saveItem);
 fileMenu.add(closeItem);
 fileMenu.add(exitItem);

 OptionListener optL = new OptionListener();
 JMenu optMenu = new JMenu("Options");
 cutItem = new JMenuItem("Cut");
 cutItem.addActionListener(optL);
 pasteItem = new JMenuItem("Paste");
 pasteItem.addActionListener(optL);
 copyItem = new JMenuItem("Copy");
 copyItem.addActionListener(optL);

 optMenu.add(cutItem);
 optMenu.add(pasteItem);
 optMenu.add(copyItem);

 jmb.add(fileMenu);
 jmb.add(optMenu);
 setJMenuBar(jmb);
 Container contentPane = getContentPane();
 contentPane.add(jsp);
 // set attribute style
 stdoutSet = new ColoredAttributeSet(Color.black);

    }

    public void init() {
 Document doc = jtp.getDocument();
    }

    public String retrieveAsString(XFileInputStream is) throws IOException {
 BufferedReader br = new BufferedReader(new InputStreamReader(is));
 StringBuffer buffer = new StringBuffer(4096);
 String line;

 while((line = br.readLine()) != null) {
     buffer.append(line);
     buffer.append('\n');
 }

 br.close();
 return buffer.toString();
    }

    public void readFromFile(XFile file) {
 Document doc = jtp.getDocument();
 try {
     XFileInputStream is = new XFileInputStream(file);
     String ins = retrieveAsString(is);
     doc.insertString(0, ins, stdoutSet);
     jtp.setCaretPosition(0);
     is.close();
 } catch (Exception e) {}
    }
    public void writeToFile(XFile file) {
 Document doc = jtp.getDocument();
 String s = null;

 try {
     XFileOutputStream os = new XFileOutputStream(file);
     s = doc.getText(0, doc.getLength());
     os.write(s.getBytes(), 0, doc.getLength());
     os.close();
 } catch (Exception e) {}

    }

    public void closeDocument() {
 Document doc = jtp.getDocument();
 try {
     doc.remove(0, doc.getLength());
 } catch (Exception e) {}
    }


    public void actionPerformed(ActionEvent ae) {
 int retval;
 XFileChooser chooser = new XFileChooser();
       if (ae.getSource() == closeItem) {
     closeDocument();
 } else if (ae.getSource() == openItem) {
     retval = chooser.showOpenDialog(this);
     if(retval == XFileChooser.APPROVE_OPTION) {
  XFile theFile = chooser.getSelectedXFile();
  closeDocument();
  if(theFile != null)
      readFromFile(theFile);
     }
 } else if (ae.getSource() == saveItem) {
     retval = chooser.showSaveDialog(this);
     if(retval == XFileChooser.APPROVE_OPTION) {
  XFile theFile = chooser.getSelectedXFile();
  if(theFile != null)
      writeToFile(theFile);
     }
 } else if (ae.getSource() == exitItem) {
     System.exit(0);
 }
    }

    public static void main(String args[]) {
 DemoEditor tc = new DemoEditor();
 tc.init();
 tc.setVisible(true);
    }

    public class ColoredAttributeSet extends SimpleAttributeSet {
 public ColoredAttributeSet(Color c) {
     super();
     StyleConstants.setForeground(this, c);
 }
    }

    public class OptionListener implements ActionListener {
```

```
 public void actionPerformed(ActionEvent ae) {
     Object src = ae.getSource();
     if (src == cutItem)
  jtp.cut();
     else if (src == copyItem)
  jtp.copy();
     else if (src == pasteItem)
  jtp.paste();
 }
    }

}
```

APPENDIX **D**

The XFileChooser Sample Program

Overview

This demo is based on the JFileChooser demo of the JDK1.2. This XFileChooserDemo has been modified to support the `XFileChooser` object. The source JFileChooserDemo.java has been changed by replacing references to J`FileChooser` objects with `XFileChooser` objects and `File` with `XFile`. This program is described in detail in “The `XFileChooserDemo` Sample Program” on page 43.

FileChooserDemo demonstrates some of the capabilities of the Swing `JFileChooser` object. It brings up a window displaying several configuration controls that allow you to play with the JFileChooser options dynamically.

To compile the FileChooserDemo demo on 1.2 on Solaris:

```
% javac -classpath ../../xfilechooser.jar  \        ExampleFileFilter.java FileChooserDemo.java ExampleFileView.java
```

To run the compiled bytecode:

```
% java -classpath ../../xfilechooser.jar:. FileChooserDemo
```

If compiling or running on Windows, use backslashes in the path, and semicolon as separator, e.g.:

```
% java -classpath ..\..\xfilechooser.jar;. FileChooserDemo
```

Note - You must be using the java in JDK1.2, thus set your path accordingly. If you are using a Java2 release prior to JDK 1.2.1 you may need the JFileChooser patch to fix bug 4169763. **The XFileChooser release notes describe how to use the the Xbootclasspath to include the patch.**

The XFileChooserDemo Program

This is the main program. It makes use of "ExampleFileFilter" on page 77 and "ExampleFileView" on page 80 to bring up an file chooser with numerous options.

CODE EXAMPLE D–1 The XFileChooser Demo Program

```
/*
 * @(#)FileChooserDemo.java 1.2 99/04/18
 *
 * Copyright 1999 by Sun Microsystems, Inc.,
 * 901 San Antonio Road, Palo Alto, California, 94303, U.S.A.
 * All rights reserved.
 *
 * This software is the confidential and proprietary information
 * of Sun Microsystems, Inc. ("Confidential Information").  You
 * shall not disclose such Confidential Information and shall use
 * it only in accordance with the terms of the license agreement
 * you entered into with Sun.
 */

import javax.swing.*;
import javax.swing.filechooser.*;

import java.awt.*;
import java.io.File;
import java.awt.event.*;
import java.beans.*;
import com.sun.xfile.*;
import com.sun.xfilechooser.*;

/**
 *
 * A demo which makes extensive use of the file chooser.
 *
 * 1.7 08/26/98
 * @author Jeff Dinkins
 */
public class FileChooserDemo extends JPanel implements ActionListener {
    static JFrame frame;

    static String metal= "Metal";
```

(continued)

```
    static String metalClassName = "javax.swing.plaf.metal.MetalLookAndFeel";

    static String motif = "Motif";
    static String motifClassName = "com.sun.java.swing.plaf.motif.MotifLookAndFeel";

    static String windows = "Windows";
    static String windowsClassName = "com.sun.java.swing.plaf.windows.WindowsLookAndFeel";


    JButton button;
    JCheckBox useFileViewButton, accessoryButton, hiddenButton, showFullDescriptionButton;
    JRadioButton noFilterButton, addFiltersButton;
    JRadioButton openButton, saveButton, customButton;
    JRadioButton metalButton, motifButton, windowsButton;
    JRadioButton justFilesButton, justDirectoriesButton, bothFilesAndDirectoriesButton;

    JTextField customField;

    ExampleFileFilter jpgFilter, gifFilter, bothFilter;
    ExampleFileView fileView;

    JPanel buttonPanel;

    public final static Dimension hpad10 = new Dimension(10,1);
    public final static Dimension vpad10 = new Dimension(1,10);

    FilePreviewer previewer;
    XFileChooser chooser;

    public FileChooserDemo() {
 setLayout(new BoxLayout(this, BoxLayout.Y_AXIS));

 chooser = new XFileChooser();
 previewer = new FilePreviewer(chooser);
 chooser.setAccessory(previewer);

 jpgFilter = new ExampleFileFilter("jpg", "JPEG Compressed Image Files");
 gifFilter = new ExampleFileFilter("gif", "GIF Image Files");
 bothFilter = new ExampleFileFilter(new String[] {"jpg", "gif"}, "JPEG and GIF Image Files");

 fileView = new ExampleFileView();
 fileView.putIcon("jpg", new ImageIcon("images/jpgIcon.jpg"));
 fileView.putIcon("gif", new ImageIcon("images/gifIcon.gif"));

 chooser.setAccessory(previewer);
 chooser.setFileView(fileView);

 // create a radio listener to listen to option changes
 OptionListener optionListener = new OptionListener();

 // Create options
 openButton = new JRadioButton("Open");
 openButton.setSelected(true);
 openButton.addActionListener(optionListener);
```

(continued)

(Continuation)

```
saveButton = new JRadioButton("Save");
saveButton.addActionListener(optionListener);

customButton = new JRadioButton("Custom");
customButton.addActionListener(optionListener);

customField = new JTextField("Doit");
customField.setAlignmentY(JComponent.TOP_ALIGNMENT);
customField.setEnabled(false);
customField.addActionListener(optionListener);

ButtonGroup group1 = new ButtonGroup();
group1.add(openButton);
group1.add(saveButton);
group1.add(customButton);

// filter buttons
noFilterButton = new JRadioButton("No Filtering");
noFilterButton.setSelected(true);
noFilterButton.addActionListener(optionListener);

addFiltersButton = new JRadioButton("Add JPG and GIF Filters");
addFiltersButton.addActionListener(optionListener);

ButtonGroup group2 = new ButtonGroup();
group2.add(noFilterButton);
group2.add(addFiltersButton);

accessoryButton = new JCheckBox("Show Preview");
accessoryButton.addActionListener(optionListener);
accessoryButton.setSelected(true);

// more options
hiddenButton = new JCheckBox("Show Hidden Files");
hiddenButton.addActionListener(optionListener);

showFullDescriptionButton = new JCheckBox("Show Extensions");
showFullDescriptionButton.addActionListener(optionListener);
showFullDescriptionButton.setSelected(true);

useFileViewButton = new JCheckBox("Use FileView");
useFileViewButton.addActionListener(optionListener);
useFileViewButton.setSelected(true);

// File or Directory chooser options
ButtonGroup group3 = new ButtonGroup();
justFilesButton = new JRadioButton("Just Select Files");
justFilesButton.setSelected(true);
group3.add(justFilesButton);
justFilesButton.addActionListener(optionListener);

justDirectoriesButton = new JRadioButton("Just Select Directories");
group3.add(justDirectoriesButton);
```

(continued)

```
justDirectoriesButton.addActionListener(optionListener);

bothFilesAndDirectoriesButton = new JRadioButton("Select Files or Directories");
group3.add(bothFilesAndDirectoriesButton);
bothFilesAndDirectoriesButton.addActionListener(optionListener);

// Create show button
button = new JButton("Show FileChooser");
button.addActionListener(this);
       button.setMnemonic('s');

// Create laf buttons.
metalButton = new JRadioButton(metal);
       metalButton.setMnemonic('o');
metalButton.setActionCommand(metalClassName);

motifButton = new JRadioButton(motif);
       motifButton.setMnemonic('m');
motifButton.setActionCommand(motifClassName);

windowsButton = new JRadioButton(windows);
       windowsButton.setMnemonic('w');
windowsButton.setActionCommand(windowsClassName);

ButtonGroup group4 = new ButtonGroup();
group4.add(metalButton);
group4.add(motifButton);
group4.add(windowsButton);

       // Register a listener for the laf buttons.
metalButton.addActionListener(optionListener);
motifButton.addActionListener(optionListener);
windowsButton.addActionListener(optionListener);

JPanel control1 = new JPanel();
control1.setLayout(new BoxLayout(control1, BoxLayout.X_AXIS));
control1.add(Box.createRigidArea(hpad10));
control1.add(openButton);
control1.add(Box.createRigidArea(hpad10));
control1.add(saveButton);
control1.add(Box.createRigidArea(hpad10));
control1.add(customButton);
control1.add(customField);
control1.add(Box.createRigidArea(hpad10));

JPanel control2 = new JPanel();
control2.setLayout(new BoxLayout(control2, BoxLayout.X_AXIS));
control2.add(Box.createRigidArea(hpad10));
control2.add(noFilterButton);
control2.add(Box.createRigidArea(hpad10));
control2.add(addFiltersButton);
control2.add(Box.createRigidArea(hpad10));
control2.add(accessoryButton);
control2.add(Box.createRigidArea(hpad10));
```

(continued)

(Continuation)

```
JPanel control3 = new JPanel();
control3.setLayout(new BoxLayout(control3, BoxLayout.X_AXIS));
control3.add(Box.createRigidArea(hpad10));
control3.add(hiddenButton);
control3.add(Box.createRigidArea(hpad10));
control3.add(showFullDescriptionButton);
control3.add(Box.createRigidArea(hpad10));
control3.add(useFileViewButton);
control3.add(Box.createRigidArea(hpad10));

JPanel control4 = new JPanel();
control4.setLayout(new BoxLayout(control4, BoxLayout.X_AXIS));
control4.add(Box.createRigidArea(hpad10));
control4.add(justFilesButton);
control4.add(Box.createRigidArea(hpad10));
control4.add(justDirectoriesButton);
control4.add(Box.createRigidArea(hpad10));
control4.add(bothFilesAndDirectoriesButton);
control4.add(Box.createRigidArea(hpad10));

JPanel panel = new JPanel();
panel.setLayout(new BoxLayout(panel, BoxLayout.X_AXIS));
panel.add(Box.createRigidArea(hpad10));
panel.add(button);
panel.add(Box.createRigidArea(hpad10));
panel.add(metalButton);
panel.add(Box.createRigidArea(hpad10));
panel.add(motifButton);
panel.add(Box.createRigidArea(hpad10));
panel.add(windowsButton);
panel.add(Box.createRigidArea(hpad10));

add(Box.createRigidArea(vpad10));
add(control1);
add(Box.createRigidArea(vpad10));
add(control2);
add(Box.createRigidArea(vpad10));
add(control3);
add(Box.createRigidArea(vpad10));
add(control4);
add(Box.createRigidArea(vpad10));
add(Box.createRigidArea(vpad10));
add(panel);
add(Box.createRigidArea(vpad10));
   }

   public void actionPerformed(ActionEvent e) {
int retval = chooser.showDialog(frame, null);
if(retval == XFileChooser.APPROVE_OPTION) {
    XFile theFile = chooser.getSelectedXFile();
    if(theFile != null) {
 if(theFile.isDirectory()) {
     JOptionPane.showMessageDialog(
```

(continued)

(Continuation)

```
    frame, "You chose this directory: " +
    chooser.getSelectedXFile().getAbsolutePath()
       );
   } else {
       JOptionPane.showMessageDialog(
    frame, "You chose this file: " +
    chooser.getSelectedXFile().getAbsolutePath()
       );
   }
   return;
      }
  }
  JOptionPane.showMessageDialog(frame, "No file was chosen.");
     }

     /** An ActionListener that listens to the radio buttons. */
     class OptionListener implements ActionListener {
  public void actionPerformed(ActionEvent e) {
      JComponent c = (JComponent) e.getSource();
      if(c == openButton) {
   chooser.setDialogType(XFileChooser.OPEN_DIALOG);
   customField.setEnabled(false);
   repaint();
      } else if (c == saveButton) {
   chooser.setDialogType(XFileChooser.SAVE_DIALOG);
   customField.setEnabled(false);
   repaint();
      } else if (c == customButton || c == customField) {
   customField.setEnabled(true);
   chooser.setDialogType(XFileChooser.CUSTOM_DIALOG);
   chooser.setApproveButtonText(customField.getText());
   repaint();
      } else if(c == noFilterButton) {
   chooser.resetChoosableFileFilters();
      } else if(c == addFiltersButton) {
   chooser.addChoosableFileFilter(bothFilter);
   chooser.addChoosableFileFilter(jpgFilter);
   chooser.addChoosableFileFilter(gifFilter);
      } else if(c == hiddenButton) {
   chooser.setFileHidingEnabled(!hiddenButton.isSelected());
      } else if(c == accessoryButton) {
   if(accessoryButton.isSelected()) {
       chooser.setAccessory(previewer);
   } else {
       chooser.setAccessory(null);
   }
      } else if(c == useFileViewButton) {
   if(useFileViewButton.isSelected()) {
       chooser.setFileView(fileView);
   } else {
       chooser.setFileView(null);
   }
      } else if(c == showFullDescriptionButton) {
   jpgFilter.setExtensionListInDescription(showFullDescriptionButton.isSelected());
```

(continued)

(Continuation)

```
 gifFilter.setExtensionListInDescription(showFullDescriptionButton.isSelected());
 bothFilter.setExtensionListInDescription(showFullDescriptionButton.isSelected());
 if(addFiltersButton.isSelected()) {
     chooser.resetChoosableFileFilters();
     chooser.addChoosableFileFilter(bothFilter);
     chooser.addChoosableFileFilter(jpgFilter);
     chooser.setFileFilter(gifFilter);
 }
    } else if(c == justFilesButton) {
 chooser.setFileSelectionMode(XFileChooser.FILES_ONLY);
    } else if(c == justDirectoriesButton) {
 chooser.setFileSelectionMode(XFileChooser.DIRECTORIES_ONLY);
    } else if(c == bothFilesAndDirectoriesButton) {
 chooser.setFileSelectionMode(XFileChooser.FILES_AND_DIRECTORIES);
    } else {
 String lnfName = e.getActionCommand();

 try {
     UIManager.setLookAndFeel(lnfName);
     SwingUtilities.updateComponentTreeUI(frame);
     if(chooser != null) {
  SwingUtilities.updateComponentTreeUI(chooser);
     }
     frame.pack();
 } catch (UnsupportedLookAndFeelException exc) {
     System.out.println("Unsupported L&F Error:" + exc);
     JRadioButton button = (JRadioButton)e.getSource();
     button.setEnabled(false);
     updateState();
 } catch (IllegalAccessException exc) {
     System.out.println("IllegalAccessException Error:" + exc);
 } catch (ClassNotFoundException exc) {
     System.out.println("ClassNotFoundException Error:" + exc);
 } catch (InstantiationException exc) {
     System.out.println("InstantiateException Error:" + exc);
 }
    }

}
   }

   public void updateState() {
String lnfName = UIManager.getLookAndFeel().getClass().getName();
if (lnfName.indexOf(metal) >= 0) {
    metalButton.setSelected(true);
} else if (lnfName.indexOf(windows) >= 0) {
    windowsButton.setSelected(true);
} else if (lnfName.indexOf(motif) >= 0) {
    motifButton.setSelected(true);
} else {
    System.err.println("FileChooserDemo if using an unknown L&F: " + lnfName);
}
   }
```

(continued)

(Continuation)

```
   class FilePreviewer extends JComponent implements PropertyChangeListener {
ImageIcon thumbnail = null;
File f = null;

public FilePreviewer(XFileChooser fc) {
    setPreferredSize(new Dimension(100, 50));
    fc.addPropertyChangeListener(this);
}

public void loadImage() {
    if(f != null) {
 ImageIcon tmpIcon = new ImageIcon(f.getPath());
 if(tmpIcon.getIconWidth() > 90) {
     thumbnail = new ImageIcon(
  tmpIcon.getImage().getScaledInstance(90, -1, Image.SCALE_DEFAULT));
 } else {
     thumbnail = tmpIcon;
 }
    }
}

public void propertyChange(PropertyChangeEvent e) {
    String prop = e.getPropertyName();
    if(prop == XFileChooser.SELECTED_FILE_CHANGED_PROPERTY) {
 f = (File) e.getNewValue();
 if(isShowing()) {
     loadImage();
     repaint();
 }
    }
}

public void paint(Graphics g) {
    if(thumbnail == null) {
 loadImage();
    }
    if(thumbnail != null) {
 int x = getWidth()/2 - thumbnail.getIconWidth()/2;
 int y = getHeight()/2 - thumbnail.getIconHeight()/2;
 if(y < 0) {
     y = 0;
 }

 if(x < 5) {
     x = 5;
 }
 thumbnail.paintIcon(this, g, x, y);
    }
}
   }

   public static void main(String s[]) {
/*
   NOTE: By default, the look and feel will be set to the
```

(continued)

```
      Cross Platform Look and Feel (which is currently Metal).
      The user may someday be able to override the default
      via a system property. If you as the developer want to
      be sure that a particular L&F is set, you can do so
      by calling UIManager.setLookAndFeel(). For example, the
      first code snippet below forcibly sets the UI to be the
      System Look and Feel. The second code snippet forcibly
      sets the look and feel to the Cross Platform Look & Feel.

      Snippet 1:
      try {
         UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
      } catch (Exception exc) {
         System.err.println("Error loading L&F: " + exc);
      }

      Snippet 2:
      try {
         UIManager.setLookAndFeel(UIManager.getCrossPlatformLookAndFeelClassName());
      } catch (Exception exc) {
         System.err.println("Error loading L&F: " + exc);
      }
   */

   try {
       // UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
       UIManager.setLookAndFeel(UIManager.getCrossPlatformLookAndFeelClassName());
   } catch (Exception exc) {
       System.err.println("Error loading L&F: " + exc);
   }

   FileChooserDemo panel = new FileChooserDemo();

   frame = new JFrame("FileChooserDemo");
   frame.addWindowListener(new WindowAdapter() {
       public void windowClosing(WindowEvent e) {System.exit(0);}
   });
   frame.getContentPane().add("Center", panel);
   frame.pack();
   frame.setVisible(true);

   panel.updateState();
      }
  }
```

ExampleFileFilter

This program is used by the `XFileChooser` demo program to filter files for display.

CODE EXAMPLE D–2 ExampleFileFilter

```
/*
 * @(#)ExampleFileFilter.java 1.2 99/04/18
 *
 * Copyright 1999 by Sun Microsystems, Inc.,
 * 901 San Antonio Road, Palo Alto, California, 94303, U.S.A.
 * All rights reserved.
 *
 * This software is the confidential and proprietary information
 * of Sun Microsystems, Inc. ("Confidential Information").  You
 * shall not disclose such Confidential Information and shall use
 * it only in accordance with the terms of the license agreement
 * you entered into with Sun.
 */


import java.io.File;
import java.util.Hashtable;
import java.util.Enumeration;
import javax.swing.*;
import javax.swing.filechooser.*;

/**
 * A convenience implementation of FileFilter that filters out
 * all files except for those type extensions that it knows about.
 *
 * Extensions are of the type ".foo", which is typically found on
 * Windows and Unix boxes, but not on Macinthosh. Case is ignored.
 *
 * Example - create a new filter that filerts out all files
 * but gif and jpg image files:
 *
 *     JFileChooser chooser = new JFileChooser();
 *     ExampleFileFilter filter = new ExampleFileFilter(
 *                   new String{"gif", "jpg"}, "JPEG & GIF Images")
 *     chooser.addChoosableFileFilter(filter);
 *     chooser.showOpenDialog(this);
 *
 * @version 1.8 08/26/98
 * @author Jeff Dinkins
 */
public class ExampleFileFilter extends FileFilter {

    private static String TYPE_UNKNOWN = "Type Unknown";
    private static String HIDDEN_FILE = "Hidden File";

    private Hashtable filters = null;
```

(continued)

(Continuation)

```
    private String description = null;
    private String fullDescription = null;
    private boolean useExtensionsInDescription = true;

    /**
     * Creates a file filter. If no filters are added, then all
     * files are accepted.
     *
     * @see #addExtension
     */
    public ExampleFileFilter() {
 this.filters = new Hashtable();
    }

    /**
     * Creates a file filter that accepts files with the given extension.
     * Example: new ExampleFileFilter("jpg");
     *
     * @see #addExtension
     */
    public ExampleFileFilter(String extension) {
 this(extension,null);
    }

    /**
     * Creates a file filter that accepts the given file type.
     * Example: new ExampleFileFilter("jpg", "JPEG Image Images");
     *
     * Note that the "." before the extension is not needed. If
     * provided, it will be ignored.
     *
     * @see #addExtension
     */
    public ExampleFileFilter(String extension, String description) {
 this();
 if(extension!=null) addExtension(extension);
  if(description!=null) setDescription(description);
    }

    /**
     * Creates a file filter from the given string array.
     * Example: new ExampleFileFilter(String {"gif", "jpg"});
     *
     * Note that the "." before the extension is not needed adn
     * will be ignored.
     *
     * @see #addExtension
     */
    public ExampleFileFilter(String[] filters) {
 this(filters, null);
    }

    /**
     * Creates a file filter from the given string array and description.
```

(continued)

```
    * Example: new ExampleFileFilter(String {"gif", "jpg"}, "Gif and JPG Images");
    *
    * Note that the "." before the extension is not needed and will be ignored.
    *
    * @see #addExtension
    */
   public ExampleFileFilter(String[] filters, String description) {
this();
for (int i = 0; i < filters.length; i++) {
    // add filters one by one
    addExtension(filters[i]);
}
 if(description!=null) setDescription(description);
   }

   /**
    * Return true if this file should be shown in the directory pane,
    * false if it shouldn't.
    *
    * Files that begin with "." are ignored.
    *
    * @see #getExtension
    * @see FileFilter#accepts
    */
   public boolean accept(File f) {
if(f != null) {
    if(f.isDirectory()) {
 return true;
    }
    String extension = getExtension(f);
    if(extension != null && filters.get(getExtension(f)) != null) {
 return true;
    };
}
return false;
   }

   /**
    * Return the extension portion of the file's name .
    *
    * @see #getExtension
    * @see FileFilter#accept
    */
    public String getExtension(File f) {
if(f != null) {
    String filename = f.getName();
    int i = filename.lastIndexOf('.');
    if(i>0 && i
    if(i>0 && i
```

ExampleFileView

This program is used by the XFileChooserDemo program to set the way a file in a file list is displayed by the file chooser.

CODE EXAMPLE D–3 ExampleFileView

```
/*
 * @(#)ExampleFileView.java 1.2 99/04/18
 *
 * Copyright 1999 by Sun Microsystems, Inc.,
 * 901 San Antonio Road, Palo Alto, California, 94303, U.S.A.
 * All rights reserved.
 *
 * This software is the confidential and proprietary information
 * of Sun Microsystems, Inc. ("Confidential Information").  You
 * shall not disclose such Confidential Information and shall use
 * it only in accordance with the terms of the license agreement
 * you entered into with Sun.
 */

import javax.swing.*;
import javax.swing.filechooser.*;

import java.io.File;
import java.util.Hashtable;

/**
 * A convenience implementation of the FileView interface that
 * manages name, icon, traversable, and file type information.
 *
 * This this implemention will work well with file systems that use
 * "dot" extensions to indicate file type. For example: "picture.gif"
 * as a gif image.
 *
 * If the java.io.File ever contains some of this information, such as
 * file type, icon, and hidden file inforation, this implementation may
 * become obsolete. At minimum, it should be rewritten at that time to
 * use any new type information provided by java.io.File
 *
 * Example:
 *    JFileChooser chooser = new JFileChooser();
 *    fileView = new ExampleFileView();
 *    fileView.putIcon("jpg", new ImageIcon("images/jpgIcon.jpg"));
 *    fileView.putIcon("gif", new ImageIcon("images/gifIcon.gif"));
 *    chooser.setFileView(fileView);
 *
 * @version 1.7 08/26/98
 * @author Jeff Dinkins
 */
public class ExampleFileView extends FileView {
    private Hashtable icons = new Hashtable(5);
```

(continued)

```
   private Hashtable fileDescriptions = new Hashtable(5);
   private Hashtable typeDescriptions = new Hashtable(5);

   /**
    * The name of the file.  Do nothing special here. Let
    * the system file view handle this.
    * @see #setName
    * @see FileView#getName
    */
   public String getName(File f) {
return null;
   }

   /**
    * Adds a human readable description of the file.
    */
   public void putDescription(File f, String fileDescription) {
fileDescriptions.put(fileDescription, f);
   }

   /**
    * A human readable description of the file.
    *
    * @see FileView#getDescription
    */
   public String getDescription(File f) {
return (String) fileDescriptions.get(f);
   };

   /**
    * Adds a human readable type description for files. Based on "dot"
    * extension strings, e.g: ".gif". Case is ignored.
    */
   public void putTypeDescription(String extension, String typeDescription) {
typeDescriptions.put(typeDescription, extension);
   }

   /**
    * Adds a human readable type description for files of the type of
    * the passed in file. Based on "dot" extension strings, e.g: ".gif".
    * Case is ignored.
    */
   public void putTypeDescription(File f, String typeDescription) {
putTypeDescription(getExtension(f), typeDescription);
   }

   /**
    * A human readable description of the type of the file.
    *
    * @see FileView#getTypeDescription
    */
   public String getTypeDescription(File f) {
return (String) typeDescriptions.get(getExtension(f));
   }
```

(continued)

```
   /**
    * Conveinience method that returnsa the "dot" extension for the
    * given file.
    */
   public String getExtension(File f) {
String name = f.getName();
if(name != null) {
    int extensionIndex = name.lastIndexOf('.');
    if(extensionIndex < 0) {
 return null;
    }
    return name.substring(extensionIndex+1).toLowerCase();
}
return null;
   }

   /**
    * Adds an icon based on the file type "dot" extension
    * string, e.g: ".gif". Case is ignored.
    */
   public void putIcon(String extension, Icon icon) {
icons.put(extension, icon);
   }

   /**
    * Icon that reperesents this file. Default implementation returns
    * null. You might want to override this to return something more
    * interesting.
    *
    * @see FileView#getIcon
    */
   public Icon getIcon(File f) {
Icon icon = null;
String extension = getExtension(f);
if(extension != null) {
    icon = (Icon) icons.get(extension);
}
return icon;
   }

   /**
    * Whether the file is hidden or not. This implementation returns
    * true if the filename starts with a "."
    *
    * @see FileView#isHidden
    */
   public Boolean isHidden(File f) {
String name = f.getName();
if(name != null && !name.equals("") && name.charAt(0) == '.') {
    return Boolean.TRUE;
} else {
    return Boolean.FALSE;
}
```

(continued)

(Continuation)

```
    };

    /**
     * Whether the directory is traversable or not. Generic implementation
     * returns true for all directories.
     *
     * You might want to subtype ExampleFileView to do somethimg more interesting,
     * such as recognize compound documents directories; in such a case you might
     * return a special icon for the diretory that makes it look like a regular
     * document, and return false for isTraversable to not allow users to
     * descend into the directory.
     *
     * @see FileView#isTraversable
     */
    public Boolean isTraversable(File f) {
 if(f.isDirectory()) {
     return Boolean.TRUE;
 } else {
     return Boolean.FALSE;
 }
    };

}
```