Files

Working with files is an important part of any program. Files have characteristics – many of which are not visible from Java. In Hack 2.1, you will explore the information Windows provides about files in general. In addition, executable files such as DLLs and EXEs have embedded version information. You will learn how to access and use this information in Hack 2.2.

Files keep changing. They are created, deleted, renamed or modified. Hack 2.3 shows how your program can get notified when these file change events occur.

Wouldn't it be cool if your program could perform file operations using the same animation effects that Windows uses when copying, moving or deleting files? You can leverage the Shell APIs for this, as you'll see in Hack 2.4.

In Hack 2.5, we'll talk about deleting files - you'll see how you can send files to the Recycle Bin, or delete stubborn files that refuse to be deleted.

Integrating your Java program and its data with the Windows Shell can make a key difference between a polished Windows app and another run of the mill Java app. In Hack 2.6, you will see how you can register your own file type with Windows. You'd like your Java program to be associated with the new file type so it is launched automatically when users open your file. Hack 2.7 shows you can do this with a few registry tweaks.

Finally, your Java program need not be restricted to the file types you define. In Hack 2.8, we will show how you can add your command to the context menu of existing file types. This way, your application could work with other registered file types.

Hack 2.1: File Information

Get file type, associated icon, attributes and timestamps for a file.

The Windows shell (Explorer.exe) uses icons to depict files, and categorizes them according to their type. The file properties dialog displays the file icon, type, timestamps, size as well as attributes.

Much of this information is inaccessible to Java. In the Win32 API, we can call two different functions to gather this information. The file attributes and timestamps can be obtained both from Kernel32.GetFileInformationByHandle() and Shell32.SHGetFileInfo(). These functions also provide some different, additional information. SHGetFileInfo(), documented at http://msdn2.microsoft.com/en-us/library/bb762179(VS.85).aspx, provides shell related file information (icon, display name and type) while GetFileInformationByHandle(), documented at http://msdn2.microsoft.com/en-us/library/aa364952.aspx, provides information such as

the file's unique identifier, its volume serial number, timestamps, and the number of links to it. We will use both these functions in this hack.

The File Information Hack application (Figure 2-1) displays file information for a chosen file. In this application, once a file is selected, pressing the 'Get File Information' button displays information for the file. Pressing the Close button closes the application. The application is built with two classes. FileInfoUI.java provides the Swing user interface and uses FileInfo.java to show this information. An instance of FileInfo can be constructed by passing in a file path. On instantiation, the corresponding file information can be obtained using the public fields of FileInfo.

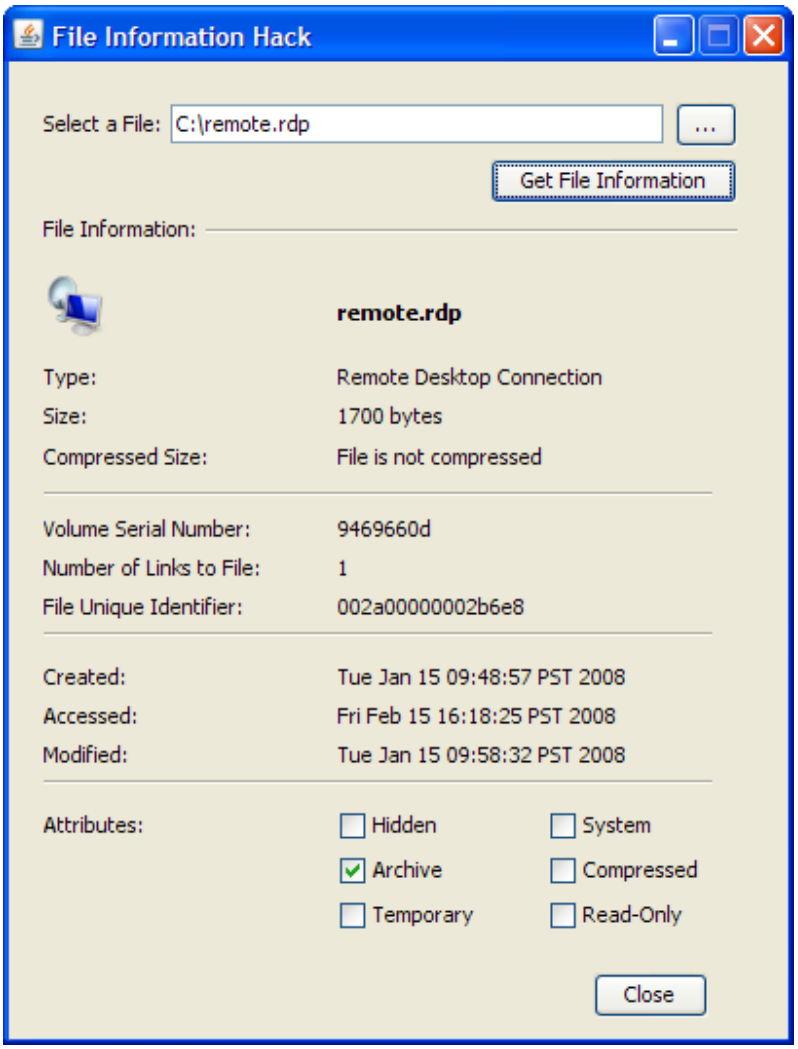


Figure 2-1 The File Information Hack application

The constructor of FileInfo simply delegates the task of collecting the file information to two utility functions:

```
public FileInfo(String filePath) {
      getShellFileInfo(filePath);
```



```
      getKernelFileInfo(filePath);
}
```

Getting Shell related file information

Obtaining the icon, display name and type of a file using the Shell APIs is a simple matter of calling Shell32.SHGetFileInfo() and passing in flags that specify the information we are interested in retrieving.

SHGetFileInfo() populates the ShFileInfo struct with the file's icon, display name and type. The icon is returned in the form of a handle to a Win32 icon (HICON), not directly usable in Java. To be able to use the icon with Java GUI components, we first convert it to a javax.swing.Icon instance using com.jinvoke.Util.getIcon(), a utility function provided by J/Invoke. Once we have converted the icon to Java, we no longer need the HICON. We destroy it using User32.DestroyIcon(hIcon). getShellFileInfo() is thus, just a simple wrapper over Shell32.SHGetFileInfo():

```
// class member to hold the icon
public javax.swing.Icon icon;

private void getShellFileInfo(String filePath) {
   ShFileInfo shInfo = new ShFileInfo();
   int basicShFlags = SHGFI_LARGEICON | SHGFI_ICON | SHGFI_DISPLAYNAME
                      | SHGFI_TYPENAME;

   // call the Shell32 SHGetFileInfo API
   Shell32.SHGetFileInfo(filePath, 0, shInfo,
                  Util.getStructSize(shInfo),
                  basicShFlags);

   // set the file icon
   int hIcon = shInfo.hIcon;
   // convert Win32 HICON to javax.swing.Icon
   icon = Util.getIcon(hIcon);
   User32.DestroyIcon(hIcon);

   // similarly set the display name and file type...
}
```

Getting file information from Kernel32

Kernel32.GetFileInformationByHandle() provides kernel related file information – the file system attributes, creation, access and modification times, etc. This function returns file information through the ByHandleFileInformation struct, documented at http://msdn.microsoft.com/en-us/library/aa363788.aspx. Instead of a file path, GetFileInformationByHandle() needs a file handle that we obtain by opening the file first using Kernel32.CreateFile():

```
// obtains file information using Kernel32.GetFileInformationByHandle
private void getFileInformation(String filePath) {
      // initialize the ByHandleFileInformation struct
```



```
      // this will be populated by the
      // Kernel32.GetFileInformationByHandle method
      ByHandleFileInformation fileInfo = new ByHandleFileInformation();
      fileInfo.ftCreationTime = new FileTime();
      fileInfo.ftLastAccessTime = new FileTime();
      fileInfo.ftLastWriteTime = new FileTime();

      // open the file and get a file handle - to be passed
      // in the Kernel32.GetFileInformationByHandle method
      int fileHandle = Kernel32.CreateFile(filePath, 0,
                        FILE_SHARE_READ | FILE_SHARE_WRITE,
                        null, OPEN_EXISTING, 0, 0);

      // call the GetFileInformationByHandle API
      Kernel32.GetFileInformationByHandle(fileHandle, fileInfo);
```

GetFileInformationByHandle() populates the fileInfo struct, which we can then query to obtain the file size, attributes and time-stamps.

Time stamps are returned as FileTime structs described at http://msdn.microsoft.com/en-us/library/ms724284(VS.85).aspx. FileTime contains two 32-bit values that represents the number of 100 nanosecond intervals that have elapsed since January 1, 1601 based on coordinated universal time (UTC). We convert FileTime to the corresponding Java date using filetimeToDate():

```
// utility method that converts time from
// FileTime format to java's Date format

private Date filetimeToDate(FileTime ftTime) {
      FileTime localFileTime = new FileTime();
      SystemTime sysTime = new SystemTime();

      // convert the filetime to local system time to account
      // for the current time zone and daylight saving time
      Kernel32.FileTimeToLocalFileTime(ftTime, localFileTime);

      // convert the local file time to SystemTime struct
      // this provides us with easy to use fields for constructing
      // GregorianCalendar next
      Kernel32.FileTimeToSystemTime(localFileTime, sysTime);

      GregorianCalendar gc = new GregorianCalendar(sysTime.wYear,
                        sysTime.wMonth - 1, // month is 0-based in Java
                        sysTime.wDay,
                        sysTime.wHour,
                        sysTime.wMinute,
                        sysTime.wSecond);
      // get time in java.util.Date format
      return gc.getTime();
}
```

filetimeToDate() first converts the FileTime to local system time to account for the current time zone and daylight saving time. To be able to easily construct an instance of jav.util.GregorianCalendar, it then converts the local file time to Win32's SystemTime

struct that has the time in terms of hours, minutes and seconds, etc. The Java date is obtained using the GregoarianCalender.getTime() method.

Another interesting piece of information that GetFileInformationByHandle() gives us is the unique identifier of the file, and its volume serial number. Together, they uniquely identify an instance of a file on Windows. The unique identifier is a 64 bit number returned in two parts – the higher order 32 bits are returned in the nFileIndexHigh field, and the lower 32 bits are returned in the nFileIndexLow field. Using Java's bit shifting operator (<<) we merge these numbers into a 64 bit long:

```
// obtain 64-bit unique identifier
long lowpart  = fileInfo.nFileIndexLow;   // the low 32 bits
long highpart = fileInfo.nFileIndexHigh;  // the high 32 bits

// use bit shifting operator to move high part to the high 32 bits
// and add to the low part to obtain the 64 bit unique identifier
uniqueIdentifier = (highpart << 32) + fileInfo.nFileIndexLow;

// obtain volume serial number...
volumeSerialNumber = fileInfo.dwVolumeSerialNumber;
```

The unique identifier can be used to check if two file handles actually point to the same file. It could also be used as a hash when storing and retrieving file related information.

Finally, the program needs to close the file handle using Kernel32.CloseHandle(). Not doing so will keep the file in-use till it terminates.

Hack 2.2: Version Information

Get version information and description for executable files.

Windows executables (EXEs and DLLs) often contain metadata such as description, company name, version and copyright information. Any program that displays these files could take advantage of this information (called 'version information'), and present a meaningful description in addition to the often-cryptic module name. Version information is also useful for installer programs to determine if a particular file needs to be updated to a newer version.

The publisher's name is another important piece of information, especially in this age of malicious software. Of course, this information can be faked – be sure to check if the module is digitally signed by the publisher before placing any trust in this information.

Version information is of two kinds – fixed information such as version number that is language independent, and localized strings such as description, copyright information and vendor name. This information is contained in a "Version Info" resource within the DLL or EXE, much like the manifest contained within a jar file. On Windows Vista, the localized strings for version information are stored in MUI (multilingual user interface) files, while the language independent parts are contained in the executable. Win32

provides the version information family of functions (http://msdn2.microsoft.com/en-us/library/ms646981(VS.85).aspx) in Version.dll to access this information.

The File Version Hack (Figure 2-2) shows the version information for Notepad.exe. After selecting a file, pressing the 'Get Version Information' button populates the version information for it. The Close button can be pressed to close the application. This application also separates the user interface code from the business logic of obtaining the version information similar to Hack 2-1. VersionInfoUI.java provides the Swing user interface and uses VersionInfo.java to obtain this information.

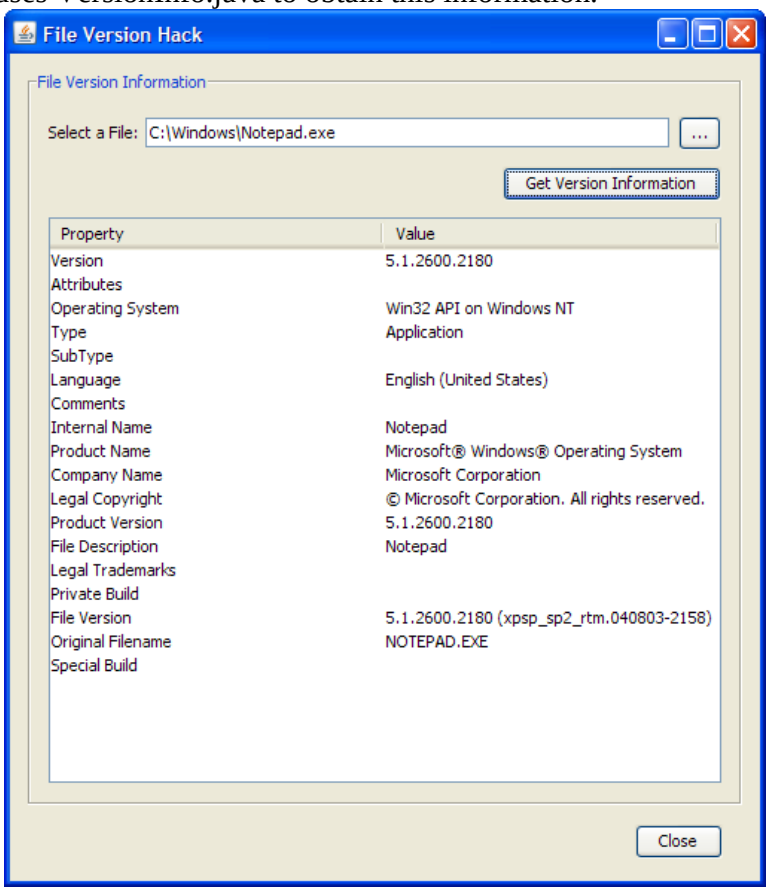


Figure 2-2 Version information for Notepad.exe

The public members of VersionInfo correspond to the fixed and language-dependent version information for a file. To use this class, simply create a VersionInfo object by passing in the file path and query its public members. If all you want to do is query the information, you can use this class. To understand how this information is actually obtained, read on.

The Version Info Resource

Executable files often contain more than just binary code. Icons, bitmaps, dialogs, menus and strings used by the DLL or EXE are often contained as resources within the file. Version information about the executable is contained as a "Version Info" resource.

Tools such as Resource Hacker (free download from http://www.angusj.com/resourcehacker/) are able to peer inside an executable and list the contained resources.

Figure 2-3 shows the Version Info resource contained within Notepad.exe. You will notice that it contains the same information shown by the File Version Hack in Figure 2-2 though in a slightly different format.

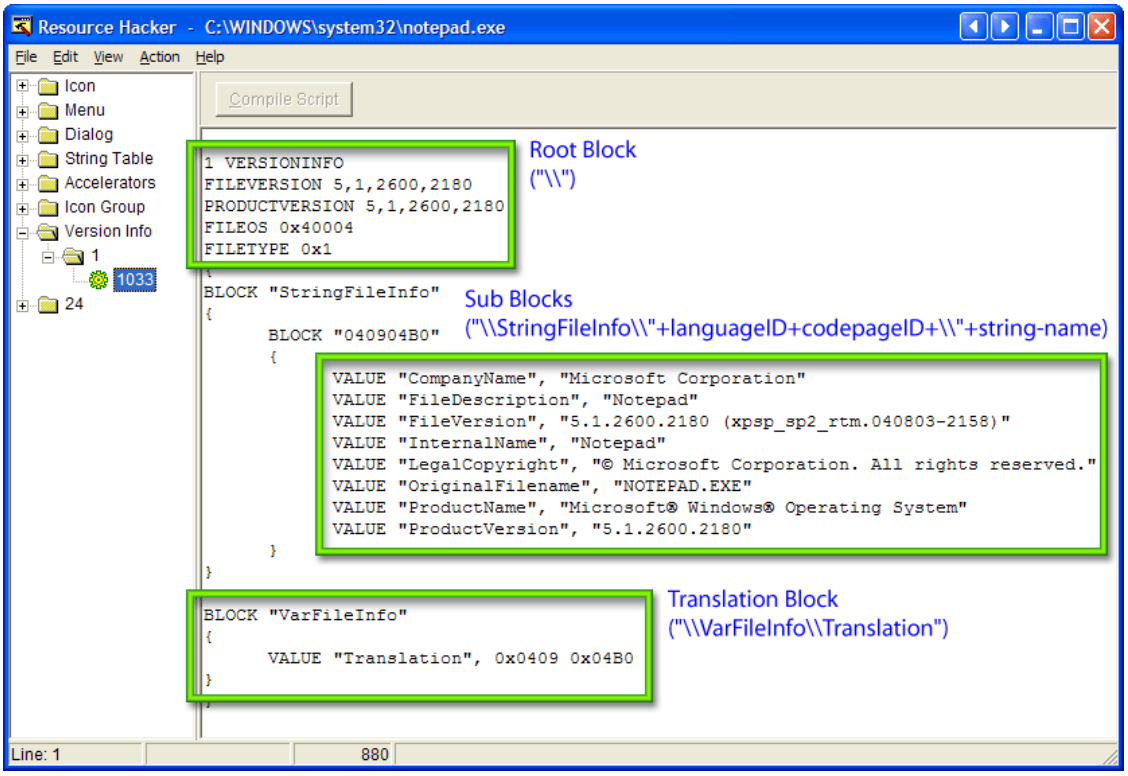


Figure 2-3 Resource Hacker showing Version Info resource in Notepad.exe

The Version Info resource is structured in three blocks as Figure 2-3 shows. The Root Block contains language independent version information such as the file version, product version, supported OS and the file type. There is additional version information in the StringFileInfo sub-blocks that is stored as strings. This information includes the company name, file description, copyright statement and the like. Because this information is localizable, and could be translated to different languages, it is stored separately in StringFileInfo blocks specific to the language and codepage for which the DLL or EXE is built. The language and codepages for which version information is available is stored in the Translation Block. The Version Info resource could contain version information strings for multiple languages and codepages. In that case, there would be several Translation values in the Translation Block, and StringFileInfo sub-blocks corresponding to each language-codepage pair.

To get version information programmatically, we first call Version.GetFileVersionInfoSize() documented at http://msdn.microsoft.com/en-

us/library/ms647005(VS.85).aspx. This function tells us if version information is available for the specified file, and how large of a buffer that information will require. We then allocate a byte array large enough to hold this information, and retrieve it using Version.GetFileVersionInfo() documented at http://msdn.microsoft.com/en-us/library/ms647003(VS.85).aspx:

```
// check if version info is available for this file
// if it is, versionInfoSize will return the size of that information
int versionInfoSize = Version.GetFileVersionInfoSize(filename, handle);

if (versionInfoSize > 0) {
   // version info is available

   // get the version info in a buffer of versionInfoSize bytes
   byte [] data = new byte[versionInfoSize];

   boolean succeeded = Version.GetFileVersionInfo(filename, handle[0],
                                                  versionInfoSize, data);
```

We can then pass this buffer to VerQueryValue() to obtain the fixed as well as localized version information. VerQueryValue(), defined in Version.dll and described at http://msdn2.microsoft.com/en-us/library/ms647464(VS.85).aspx, has the following signature:

```
@NativeImport(library="Version")
static native boolean VerQueryValue (byte[] data, String subBlock,
int[] ptr, int[] asize);
```

VerQueryValue() can be used to obtain the fixed as well as language specific version information. To get the language-specific version information, we need language and code page identifiers – these can also be queried using VerQueryValue().

`data` is the byte array that we obtained using Version.GetFileVersionInfo() and `subBlock` is a string that indicates which piece of information we want to be returned. It could be the root block, the translation block or a StringFileInfo sub-block. The different block strings that can be used are listed at the MSDN documentation for VerQueryValue(). When the method returns, `ptr[0]` contains a pointer to the requested version information. The last argument is used by the function to return the size of the requested data pointed to by the `ptr` argument.

Getting the language-independent version information

As we saw in Figure 2-3, the language-independent version information is contained in the root block. To get this information, we call VerQueryValue() with the root block string as the subBlock parameter:

```
// sub-blocks used by VerQueryValue function
// see http://msdn.microsoft.com/en-us/library/ms647464(VS.85).aspx
// for the list of valid block strings
```



```
private final String ROOT_BLOCK = "\\";
private final String TRANSLATION_BLOCK = "\\VarFileInfo\\Translation";
...

// Get the fixed (language and codepage independent) version info
Version.VerQueryValue(data, ROOT_BLOCK, ptr, bytesReturned);
VS_FixedFileInfo fixedFileInfo = Util.ptrToStruct(ptr[0],
                                    VS_FixedFileInfo.class);

// Set the fixed file version information for this file
setFixedFileInfo(fixedFileInfo);
```

On returning from the function, `ptr[0]` contains a pointer to a VS_FixedFileInfo struct (see http://msdn.microsoft.com/en-us/library/ms646997(VS.85).aspx), which contains language and code page independent version information for the file. J/Invoke provides Util.ptrToStruct() to convert a pointer to a struct. This utility method takes a native memory location pointing to a struct, and the J/Invoke class representing that struct as input and returns the struct initialized with data from that memory location.

We pass on the obtained struct to setFixedFileInfo(), which saves the language-independent version information in VersionInfo's public fields:

```
// class members for storing fixed version information
public int versionMajor;
public int versionMinor;
public int versionBuild;
public int versionPrivatePart;
// other fixed version fields omitted...

// obtain and interpret fixed file information
private void setFixedFileInfo(VS_FixedFileInfo fixedFileInfo) {
   // binary file version
   versionMajor = hiword(fixedFileInfo.dwFileVersionMS);
   versionMinor = loword(fixedFileInfo.dwFileVersionMS);
   versionBuild = hiword(fixedFileInfo.dwFileVersionLS);
   versionPrivatePart = loword(fixedFileInfo.dwFileVersionLS);
   if (versionMajor == 0)
      fileVersionString = "";
   else
      fileVersionString = versionMajor + "."+ versionMinor + "." +
                          versionBuild + "." + versionPrivatePart;

   // similar code to obtain other fixed version information below...
}
```

Getting the language dependent version information

Getting the language dependent version information is a two stage operation. We first need to get the language and code page identifiers for which the language-specific version information strings are available.

For this we use the `TRANSLATION_BLOCK` sub-block. On returning from VerQueryValue(), `ptr[0]` contains a pointer to an array of translation values. Each translation value contains a language and codepage id. We just care about the first pair in this code, but potentially multiple language and code-page pairs could be supported by a file:

```
// Get the translation block - this provides the language and codepage
// identifiers. These are used to obtain the language-specific
// predefined strings later
Version.VerQueryValue(data, TRANSLATION_BLOCK, ptr, bytesReturned);
if (bytesReturned[0] > 0) {
   Translation translation =  Util.ptrToStruct(ptr[0],
                                               Translation.class);
   // convert the language and codepage identifiers to hex strings
   String codepageID = String.format("%1$04x", translation.codepageID);
   String languageID = String.format("%1$04x", translation.languageID);
```

We can now, finally, get the language dependent version information strings by calling VerQueryValue with the following sub-block:

```
"\\StringFileInfo\\" + languageID + codepageID + "\\" + string-name
```

Here, languageID and codepageID are specified as four character wide hex strings, and the string-name is one of the following predefined values as described in the MSDN documentation for VerQueryValue():

Comments	InternalName	ProductName
CompanyName	LegalCopyright	ProductVersion
FileDescription	LegalTrademarks	PrivateBuild
FileVersion	OriginalFilename	SpecialBuild

To get the version information strings for this language-codepage combination, we call VerQueryValue() repeatedly for each predefined string name. The language dependent strings are again stored in public fields of the VersionInfo class.:

```
// Get the version information strings for this language-codepage
// combination
String subBlockPrefix = "\\StringFileInfo\\" + languageID + codepageID
                                                              + "\\";
// set the class fields for language dependent version info strings...
comments     = getPropertyValue(data, subBlockPrefix + "Comments");
internalName = getPropertyValue(data, subBlockPrefix + "InternalName");
productName  = getPropertyValue(data, subBlockPrefix + "ProductName");

// similarly for others...
```

The strings are extracted from the version info resource using getPropertyValue()::


```
private String getPropertyValue(byte[] data, String subBlock) {
    int[] ptr = new int[1];
    int[] bytesReturned = { 0 };
    boolean result = VerQueryValue(data, subBlock, ptr, bytesReturned);
    String value = "";
    if (result == true)
        value = Util.ptrToString(ptr[0]);

    return value;
}
```

getPropertyValue() simply calls VerQueryValue() with the sub-block containing the language and code page identifiers and the predefined string name. If the file contains a language dependent version information for the specified string name, ptr[0] contains a pointer to that string. We use J/Invoke's Util.ptrToString() to read the string from native memory. Finally, VersionInfoUI queries these fields to display them in the user interface.

Hack 2.3: File System Watcher

Get notified when a file or directory changes

You may not realize how often files and folders get created and modified when you use the operating system. As a developer, you may be interested in understanding and monitoring file system changes for reasons such as security, software analysis, or just as part of program workflow. For instance, your program may require you to know if a configuration file changed and pick up the configuration changes dynamically. Or, you may want to process a file when it is placed in a particular folder.

Java provides no such file change notification API – writing one in pure Java involves repeatedly checking (polling) the file system for changes. If you poll too often, you lose CPU cycles; if you poll too little, you are not always notified on time. An example of an online Java file system watcher that uses polling can be seen at http://twit88.com/blog/2007/10/02/develop-a-java-file-watcher/

Kernel32.dll provides ReadDirectoryChangesW() just for this. This function lets you register the type of file system changes you are interested in knowing about (new files, updates, deletes, attribute changes, etc), and retrieves information about those changes when they occur.

The File System Watcher application, shown in Figure 2-4, can monitor a folder for changes.

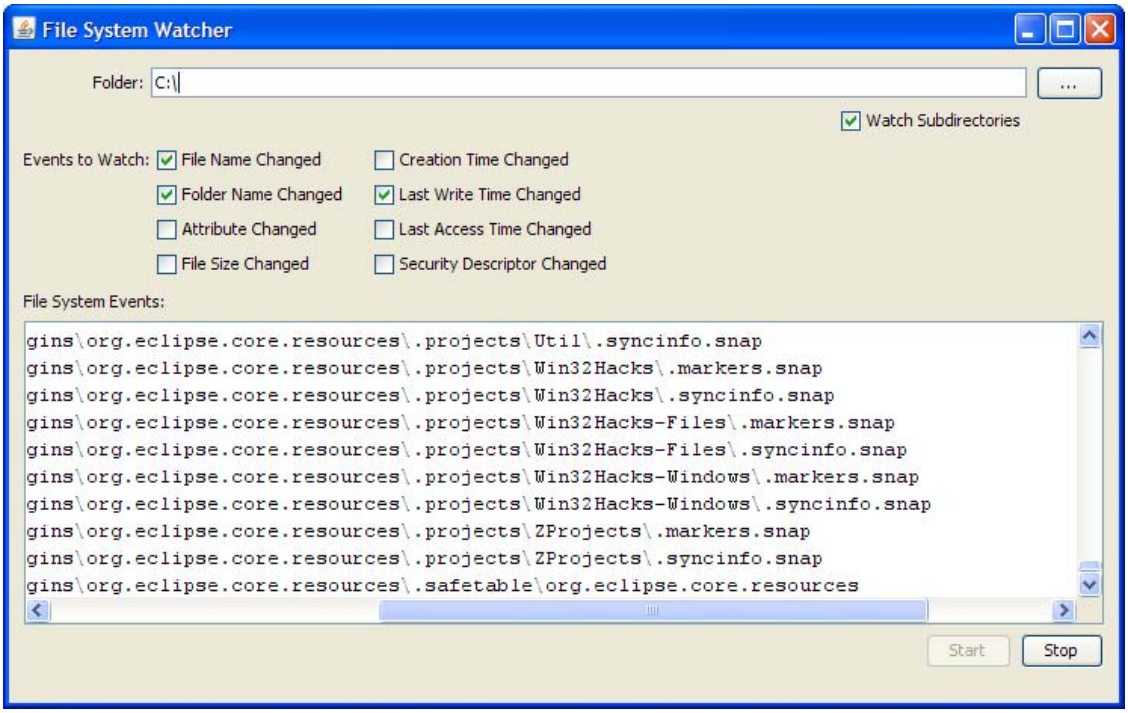


Figure 2-4 File system watcher application showing file system activity

When the user chooses a folder and clicks the Start button, the app starts a new thread to watch for file system changes. The thread calls ReadDirectoryChangesW() to subscribe to the chosen file system events, and updates the output JTextArea when such events occur.

The ReadDirectoryChangesW Function

Let's look at the function signature of this function in com.jinvoke.win32.Kernel32:

```
public static boolean ReadDirectoryChangesW(int hDirectory,
                                            byte[] lpBuffer,
                                            int nBufferLength,
                                            boolean bWatchSubtree,
                                            int dwNotifyFilter,
                                            int[] lpBytesReturned,
                                            Overlapped lpOverlapped,
                        Callback lpCompletionRoutine)
```

The full description of the function and its arguments is available on MSDN at http://msdn2.microsoft.com/en-us/library/aa365465(VS.85).aspx but it's really a simple function that takes in the information needed to watch for file system events, and returns the results in a buffer in a specified format.

We just tell the function what folder to watch (`hDirectory`), whether or not to watch its child folders (`bWatchSubtree`) and the events to watch for (`dwNotifyFilter`). The last two arguments can be used for calling the function asynchronously, but for now we will

just wait for the function to complete. The results are returned in a byte-array buffer that we provide (`lpBuffer` of size `nBufferLength`).

The directory handle is obtained by opening the directory using CreateFile(), and `bWatchSubtree` and `dwNotifyFilter` are derived from the options the user chooses in Figure 2-4.

CreateFile()
The name "CreateFile" is a bit of a misnomer for this function. It doesn't just create files, it opens them too. Well for that matter, it doesn't work with files alone. It works with folders, as well as with Windows interprocess communication mechanisms such as mailslots and named pipes. This versatile and powerful function can also be used to open devices such as the physical hard-disk, LCD screen, COM ports and USB peripherals like we will see in Hack 22.1. Its MSDN documentation at http://msdn2.microsoft.com/en-us/library/aa363858.aspx is worth browsing to get an idea of its capabilities.

ReadDirectoryChangesW() is called in a loop. Every time the function is called, the directory changes that occurred since the previous call are accumulated and stored by Windows in an internal buffer that it maintains. The internal buffer is allocated on the first call to ReadDirectoryChangesW(), and is maintained till the associated directory handle is closed. If there are no changes to the directory, ReadDirectoryChangesW() blocks execution and does not return unless there are changes to report. This saves CPU cycles, as the thread is not kept busy calling it unnecessarily in a tight loop. ReadDirectoryChangesW() also offers an asynchronous mode of operation that we won't cover in this hack. The asynchronous mode requires the use of Overlapped I/O, that we will cover in Hack 6.7 (Can I operate Windows with my foot?).

The directory changes that have accumulated in the buffer are copied to another buffer that we provide when we call the function next. The number of bytes written to the buffer is returned in `lpBytesReturned[0]`. If too many directory changes occur and the internal buffer maintained by Windows overflows, ReadDirectoryChangesW() fails with the ERROR_NOTIFY_ENUM_DIR error code. To prevent this from happening, it is important to call the function frequently enough.

We also need to ensure that the buffer we provide to the function is large enough to hold the expected number of directory change notifications. The buffer size is usually set in multiples of 4K (4096 bytes), to match the page size of the operating system. FileSystemWatcher.java uses a buffer size of 8K. If the buffer size is too large, we use up more memory than necessary. If it is too low, the number of bytes returned will be zero.

So setting the buffer size is somewhat of a trial and error, although somewhere between 8K and 64K should be plenty. This will also depend on the number of notifications your application is likely to receive. If you expect more notifications, the buffer size should be made larger.

FileSystemWatcher accomplishes the above task in watchFolder():

```
public void watchFolder(String directory, int flags) {
      int dirHandle = Kernel32.CreateFile(directory,
         WinConstants.FILE_LIST_DIRECTORY,  // required access right
         WinConstants.FILE_SHARE_READ +     // share for reading
         WinConstants.FILE_SHARE_DELETE +   // share for deleting
         WinConstants.FILE_SHARE_WRITE,     // share for writing
         null,                        // use default security descriptor
         WinConstants.OPEN_EXISTING, // the folder already exists
         WinConstants.FILE_FLAG_BACKUP_SEMANTICS, // needed to get
                                                  // directory handles
         0);

      // some basic error handling for sanity
      if (dirHandle == 0) {
            System.out.println("CreateFile Failed");
            return;
      }

      // create a buffer of 8K to read multiple file change events
      // that occurred in between successive calls to
      // ReadDirectoryChangesW function
      int BUFSIZE = 8 * 1048;
      byte[] buf = new byte[BUFSIZE];
      int[] bytesReturned = new int[1];

      stop = false; // setting the stop flag to false - it will be
                    // set to true when the 'Stop' button is pressed

      // call ReadDirectoryChangesW in a loop
      // till the 'Stop' button is pressed
      while (!stop) {
            if (Kernel32.ReadDirectoryChangesW(dirHandle, buf, BUFSIZE,
                true, flags, bytesReturned, null, null)) {
                  // Read the directory changes here...

                  // ReadDirectoryChangesW blocks until the directory
                  // is changed, so we aren't polling unless the
                  // directory changes
            }
      }
}
```

Okay, that was the easy part. Now, let's move onto reading the contents of the buffer returned by ReadDirectoryChangesW.

Reading the Results

The MSDN documentation at http://msdn.microsoft.com/en-us/library/aa364391(VS.85).aspx graciously tells us that the structure of the buffer is defined by the FILE_NOTIFY_INFORMATION structure:

```
typedef struct _FILE_NOTIFY_INFORMATION {
```



```
  DWORD NextEntryOffset;
  DWORD Action;
  DWORD FileNameLength;
  WCHAR FileName[1];
} FILE_NOTIFY_INFORMATION;
```

It's lying! The first three fields of this structure are indeed three `DWORDs` (or `ints` in Java) but the last member (`FileName`) isn't really a one character array. It's really a variable length field – the size of which is contained in the FileNameLength member.

Declaring an array with one member in a structure is a sleazy technique to keep the compiler happy while describing a variable length array. In reality the structure size is variable. We are thus unable to use J/Invoke's @NativeStruct annotation to describe this structure. Thank you, Win32 - you just made our job a little difficult. But fear not – Java's ByteBuffer class will bail us out here.

The buffer is filled with three pieces of information. The Action member tells what event occurred, and the FileNameLength in conjunction with the variable length FileName member tells us the file or folder name associated with the file change event. The NextEntryOffset member tells us the position of the next file change event entry in the buffer. If it is zero, this is the last entry in the buffer and we are ready to call ReadDirectoryChangesW again. If not, we advance the buffer position by those many bytes to read the next entry.

To make this easy for us, we wrap the byte array buffer in Java's ByteBuffer class, and set it's endianness to LITTLE_ENDIAN, which is the native encoding in Win32. We can then easily read the next entry offset, action and filename length using `getInt()`. To read the file name, we read a byte array long enough to contain the filename using the `get(byte[] dst)` method and convert it to a Java:

```
if (Kernel32.ReadDirectoryChangesW(dirHandle, buf, BUFSIZE,
       true, flags, bytesReturned, null, null)) {
    // Read the directory changes here...
    if (bytesReturned[0] != 0) {
        ByteBuffer bb = ByteBuffer.wrap(buf);
        bb.order(ByteOrder.LITTLE_ENDIAN);

        while (true) {
            // used to compute next entry offset later
            int prevEntryOffset = bb.position();

            // read FILE_NOTIFY_INFORMATION members -
            // NextEntryOffset, Action & FileNameLength

            int nextEntryOffset = bb.getInt();

            // The action variable is declared as final as it
            // is accessed by an anonymous inner class in fileAction()
            // non-final variables cannot be accessed
            // by anonymous inner classes per Java syntax rules
            final int action = bb.getInt();
```



```
            // Read the filename length
            int fileLen = bb.getInt();

            // Read the filename into a byte array
            byte[] stringbytes = new byte[fileLen];
            bb.get(stringbytes);

            try {
                // convert byte-array to Java String
                // the encoding Win32 uses is UTF-16LE
                // once again, filename is declared as final because it
                // is passed to an anonymous inner class in fileAction
                final String filename = new String(stringbytes,
                                                   "UTF-16LE");
                // fire the file event action
                fileAction(action, new String(filename));
            } catch (UnsupportedEncodingException e) {
                e.printStackTrace();
            }

            if (stop || nextEntryOffset == 0) {
                // no more file change events to be reported in
                // this iteration – let's break out of the loop
                // to call ReadDirectoryChangesW again
                break;
            } else {
                // Skip nextEntryOffset from prevEntryOffset to read
                // next file change event entry
                int newposition = prevEntryOffset + nextEntryOffset;
                bb.position(newposition);
            }
        }
    }
}
```

The user interface is actually updated by the fileAction() method, which is a simple switch-case statement to append information to the output JTextArea.

As this method is called on another thread, it is not safe to update the output JTextPane component directly. We therefore use SwingUtilities.invokeLater to cause the component to be updated from the AWT Event Thread

```
private void fileAction(final int action, final String filename) {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                switch (action) {
                    case FILE_ACTION_ADDED:
                        output.append("\nFile Added: ");
                        break;

                    case FILE_ACTION_MODIFIED:
                        output.append("\nFile Modified: ");
                        break;
```



```
                    case FILE_ACTION_REMOVED:
                        output.append("\nFile Modified: ");
                        break;

                    case FILE_ACTION_RENAMED_NEW_NAME:
                        output.append("\nFile Renamed - New Name: ");
                        break;

                    case FILE_ACTION_RENAMED_OLD_NAME:
                        output.append("\nFile Renamed - Old Name: ");
                        break;

                    default:
                }
                output.append(filename);
            }
        });
    }
```

This method can be modified to do other things – fire Java events, read updated configuration files, or process the updates in another way.

Using ReadDirectoryChangesW() is superior to polling the file system for updates. It blocks execution unless there is a directory change, and notifies us by resuming execution as and when file system changes occur. We don't need to waste CPU cycles repeatedly enumerating directory contents when few or no file change events have occurred. At the same time, if we poll less often, we won't know of changes in a directory for a longer time. Also, we might entirely miss some file system events that occurred in between.

Hack 2.4: Animated File Operations

Copy/move big files with Window's animation effects.

Users are impatient. They begin tapping their fingers the moment they hit Enter. They start squirming in their seats 250 milliseconds later. And they get ready to call you up if the button they clicked doesn't appear to do anything useful in half a second.

The key to great user experience, and hence to happy users, is to give users visual feedback that makes them *think* the computer is busy obeying their command. They'll even grant you an extra five seconds for letting the CPU sleep if you show them some objects flying around the screen, while the cursor is busy spinning frantically.

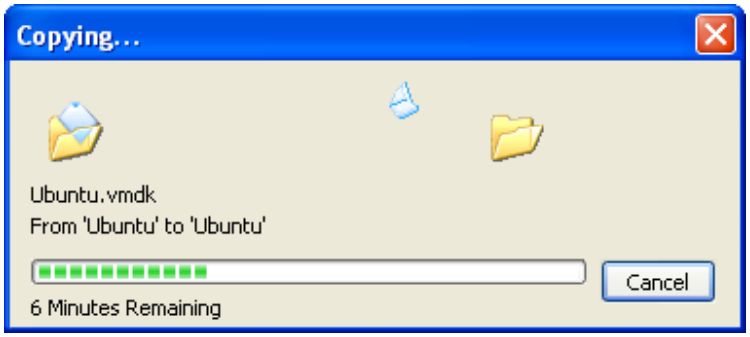


Figure 2-5 A file copy operation in progress

The Windows shell displays a dialog showing an animation of files flying across when copying, moving or deleting files (Figure 2-5). Wouldn't it be nice if you too could use these animations in your Java applications while performing file operations? It will hypnotize users into thinking your app is fast and responsive, and also integrate your application better with Windows.

FileOperationsUI.java is a Java Swing application that copies, moves and renames files or folders using the standard Windows animations, just like Windows Explorer. The application relies on FileOperations.java to show the animations. The application (Figure 2-6) provides options to show the progress dialog box with animations, as well customize the user interface and behavior of the file operation.

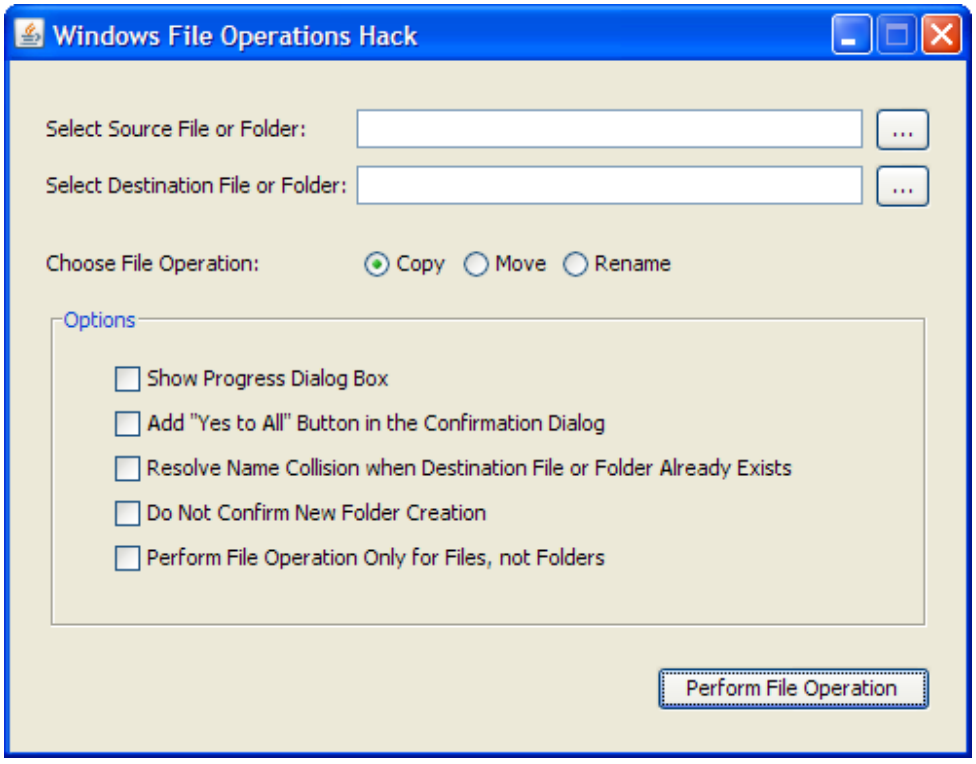


Figure 2-6 Shell file operations application

SHFileOperation() from Shell32.dll performs file operations such as Copy, Move, Delete and Rename while providing feedback to users using progress bars and snazzy animation effects. We will be calling this function from FileOperations.java.

The signature of SHFileOperation(), documented at http://msdn.microsoft.com/en-us/library/bb762164(VS.85).aspx, is deceptively simple:

```
public static int SHFileOperation(ShFileOpStruct lpFileOp);
```

The devil is in the details – namely the innocent looking ShFileOpStruct argument. This struct, detailed at http://msdn.microsoft.com/en-us/library/bb759795(VS.85).aspx is represented in Java as:

```
@NativeStruct
public class ShFileOpStruct {
    public int hwnd;
    public int wFunc;
    public String pFrom;
    public String pTo;
    public short fFlags;
    public boolean fAnyOperationsAborted;
    public int hNameMappings;
    public String lpszProgressTitle;
}
```

ShFileOpStruct has members to specify the file operation to perform (`wFunc`), the source (`pFrom`) and destination (`pTo`) files, and options (`fFlags`) that control whether or not to show the progress dialog box, ask for confirmation, etc.

Now, a few details. `pFrom` and `pTo` need to be double null terminated string. J/Invoke null-terminates Java strings when passing to native code, but you need to add the other null yourself by concatenating the paths with a "\0":

```
// double null terminating pFrom and pTo by appending a null
shFileOp.pFrom = fileOpsUI.txtFieldSource.getText()+ "\0";
shFileOp.pTo =  fileOpsUI.txtFieldDestination.getText() + "\0";
```

This is because the path strings could actually contain multiple paths each separated by nulls for the file operation. This can be used for copying, moving or deleting a bunch of files and folders in one operation.

If you are deleting files, and forget the explicit null terminator ("\0") and there happens to be a "*.*" in native memory right after the path string, it can cause the SHFileOperation API to think you are deleting "*.*" – something you never want to do. This is not just a hypothetical problem – the Shell Revealed blog at http://shellrevealed.com/blogs/shellblog/archive/2006/09/28/Common-Questions-Concerning-the-SHFileOperation-API_3A00_-Part-2.aspx reports that such a thing actually happened with a poorly-written application in the Windows Vista Beta 2 timeframe! Also, the paths need to be full (absolute) paths, not relative paths.

Finally, if this API doesn't work the way that you expect, don't rely on the returned error code. This API is notorious for returning undocumented and incorrect error codes – just follow Microsoft's advice from the API documentation for SHFileOperation:

"You are responsible for validating the input. If you do not validate it, you will experience unexpected results… Do not use GetLastError with the return values of this function."

The Windows Shell team has posted a couple of useful blog posts on "Common Questions Concerning the SHFileOperation API" at the Shell Revealed blog mentioned above. This is definitely worth a read if you'll be using this API anytime soon.

FileOperations.performFileOperation() reads the options from the user interface, converting them to the corresponding flags for SHFileOpStruct, and finally calling SHFileOperation():

```
// import Win32 constants such as FO_COPY, FO_MOVE, etc...
import static com.jinvoke.win32.WinConstants.*;

// performs the file operation using SHFileOperation
// called when the "Perform File Operation" button
// in Fig 2-6 is clicked
public void performFileOperation(int selectedCommand) {

   int fileOpCommand = 0;
   int flags = 0;

   // set the file operation
   switch (selectedCommand) {
       case 0:
         fileOpCommand = FO_COPY;
         break;

       case 1:
         fileOpCommand = FO_MOVE;
         break;

       case 2:
         fileOpCommand = FO_RENAME;
         break;
   }

   // set the flags based on the user interface selections
   // the checkboxes are the ones shown in Fig 2-6
   if (fileOpsUI.chkShowProgress.isSelected()==false)
      flags = flags | FOF_SILENT;
   if (fileOpsUI.chkYesToAll.isSelected()==false)
      flags = flags | FOF_NOCONFIRMATION;
   if (fileOpsUI.chkNameCollisionRename.isSelected())
      flags = flags | FOF_RENAMEONCOLLISION;
   if (fileOpsUI.chkConfirmDirCreation.isSelected())
      flags = flags | FOF_NOCONFIRMMKDIR;
   if (fileOpsUI.chkFilesOnly.isSelected())
```



```
      flags = flags | FOF_FILESONLY;

   // by adding the FOF_ALLOWUNDO flag you can move
   // a file to the Recycle Bin instead of deleting it
   // we'll do that in the RecycleBin hack


   // create a ShFileOpStruct struct and set its members
   ShFileOpStruct shFileOp = new ShFileOpStruct();
   shFileOp.wFunc = fileOpCommand;
   shFileOp.pFrom = fileOpsUI.txtFieldSource.getText()+ "\0";
   shFileOp.pTo =  fileOpsUI.txtFieldDestination.getText() + "\0";
   shFileOp.fFlags = (short) flags;

   // perform the requested file operation
   Shell32.SHFileOperation(shFileOp);
}
```

Hack 2.5: Using the Recycle Bin

Delete files by moving them to the Recycle bin and delete stubborn files at reboot time.

There are as many ways to delete a file as there are to skin a cat. Java's File class provides a delete method that, guess what, deletes files. Permanently. Windows Explorer is more forgiving – by default deleted files are moved to the Recycle Bin from which they can be restored if needed. Of course, the option to delete files permanently is present too. Some files are stubborn – they refuse to be deleted. This can happen if they are 'in use'. Kernel32.dll provides MoveFileEx() to mark such files for deletion when the computer reboots.

Using the Recycle Bin

The Recycle Bin Hack shown in Figure 2-7 uses the Recycle Bin programmatically. The user interface code is contained in RecycleBinUI.java in the associated code for this hack. This class uses RecycleBin.java that has methods for deleting files and for emptying and querying the recycle bin.

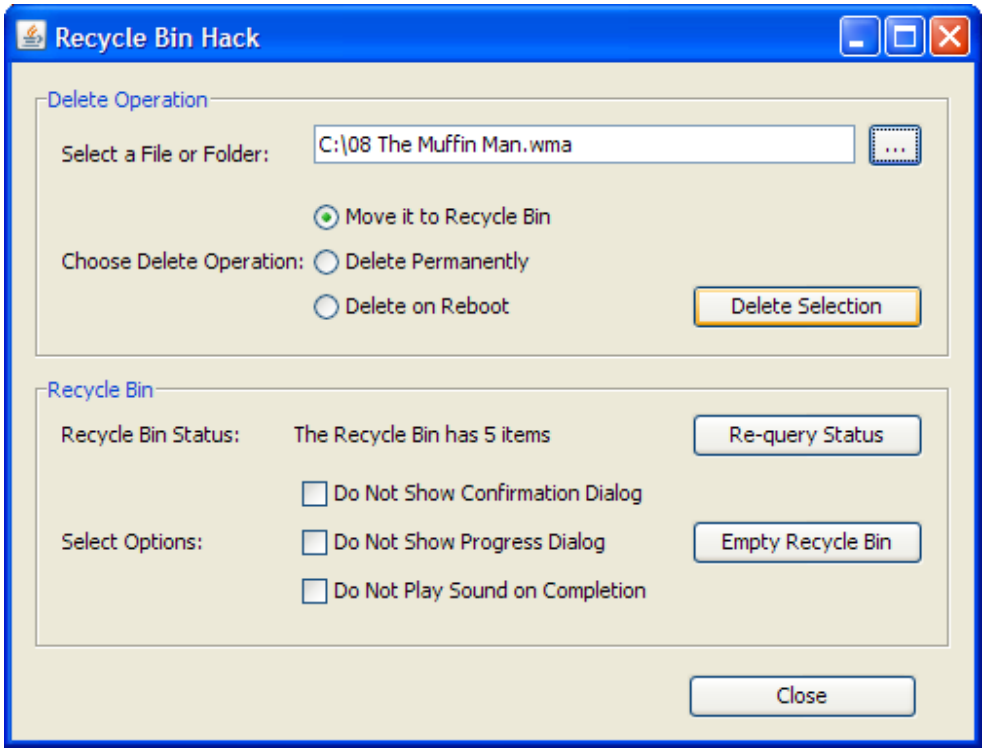


Figure 2-7 Recycle Bin Hack application

To delete a file for good, call Shell32.SHFileOperation() described in the previous hack (Hack 2.4: Animated file operations) with FO_DELETE as the file operation. By using the FOF_ALLOWUNDO flag, you can move the file to the Recycle Bin instead of deleting it.

The Shell API has functions for working with the Recycle Bin. SHQueryRecycleBin() can query the number of items present in the Recycle Bin and their cumulative size as described at http://msdn.microsoft.com/en-us/library/bb762241(VS.85).aspx. The function is passed a ShQueryRBInfo struct. ShQueryRBInfo is documented at http://msdn.microsoft.com/en-us/library/bb759803(VS.85).aspx and is defined using C preprocessor directives such as #if, #else, #endif:

```
typedef struct _SHQUERYRBINFO{
    DWORD cbSize;
    #if !defined(_MAC) || defined (_MAC_INT_64)
        __int64 i64Size;
        __int64 i64NumItems;
    #else
        DWORDLONG i64Size;
        DWORDLONG i64NumItems;
    #endif
} SHQUERYRBINFO, *LPSHQUERYRBINFO;
```

To convert such a struct to its equivalent Java class, we'll have to don the cap of the C preprocessor, and simplify the struct definition ourselves. We can safely assume that

_MAC is not defined on the Win32 platform, so "`#if !defined(_MAC)`" will evaluate to true. We can thus delete the block of code in the `#else` block, and arrive at the following simpler definition:

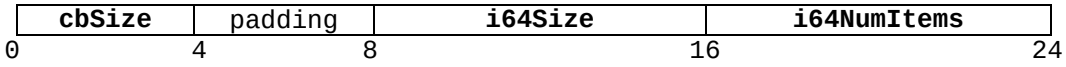
```
typedef struct _SHQUERYRBINFO{
    DWORD cbSize;
    __int64 i64Size;
    __int64 i64NumItems;
} SHQUERYRBINFO, *LPSHQUERYRBINFO;
```

This is definitely more palatable, and using the data type conversions table from Chapter 1, we can substitute Java types for the native types to arrive at the equivalent J/Invoke declaration:

```
@NativeStruct
class ShQueryRBInfo {
    int cbSize;       // the size of the struct, use Util.getStructSize
    long i64Size;     // the total size of the items in the Recycle Bin
    long i64NumItems; // number of items in the Recycle Bin
}
```

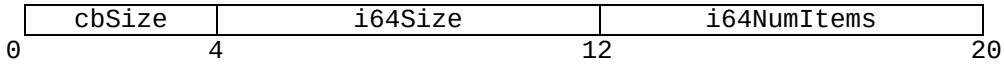
One more thing… There is a slight problem with this structure definition that you'll run into if you use the definition above. Instead of the default 8 byte packing, it so happens that this struct uses 1 byte packing.

Normally, when a struct is laid out in memory, each member is aligned at 8 byte offsets, with gaps being filled with empty space (called padding):



The numbers below the struct members indicate their offset from the beginning of the struct. Here `cbSize` starts at the beginning of the struct and occupies 4 bytes, but is followed by a padding of another 4 bytes, before the next two members (`i64Size` and `i64NumItems` of 8 bytes each) are stored. This is done for performance reasons, as the CPU is able to most efficiently access data that is kept in a memory location that is a multiple of its size in bytes. With an 8 byte alignment , which is the default in Win32, bytes (1 byte), shorts(2 bytes), ints(4 bytes), and longs(8 bytes) are automatically aligned at their preferred memory address.

However, the Windows Shell uses 1-byte alignment for space-efficiency. When the shell APIs were defined in the early Win32 days, memory was a scarce resource and thus 1 byte packing was used for the shell structs. In the ShellAPI.h header file, the alignment is set to one byte through the use of compiler directives. This causes the ShQueryRBInfo structure to be laid out in memory as:



With 1 byte packing, there is no padding between `cbSize`, `i64Size` and `i64NumItems` and the struct occupies lesser space in memory. The size of this struct is 20 bytes as opposed to 24 bytes with the default 8 byte packing. However, the CPU will lose some efficiency in accessing `i64Size` and `i64NumItems` as they do not start at a memory address that is a multiple of their data type size (i.e. 8 bytes). You can read more about data alignment at `http://msdn.microsoft.com/en-us/library/ms253949(VS.80).aspx`.

The `@NativeStruct` annotation provides a way to specify the packing size used for aligning structure fields. By setting **`packing=1`** for the J/Invoke declaration of ShQueryRBInfo, we arrive at the correct definition:

```
@NativeStruct(packing=1)
class ShQueryRBInfo {
    int cbSize;       // the size of the struct, use Util.getStructSize
    long i64Size;     // the total size of the items in the Recycle Bin
    long i64NumItems; // number of items in the Recycle Bin
}
```

Thankfully, this struct is already defined in the com.jinvoke.win32.structs package, so we didn't need to do all this ourselves. The above discussion will be useful when you encounter structs that are not defined in the J/Invoke packages and use non-default packing.

SHQueryRecycleBin() populates the struct with information on the number of items in the Recycle Bin, and their cumulative size:

```
// queries the number of files in the Recycle Bin
public String queryRecycleBin() {
    ShQueryRBInfo shQBInfo = new ShQueryRBInfo();

    shQBInfo.cbSize = Util.getStructSize(ShQueryRBInfo.class);

    Shell32.SHQueryRecycleBin("C:\\", shQBInfo);

    String result =  "";
    if (shQBInfo.i64NumItems !=0)
        result = "The Recycle Bin has " + shQBInfo.i64NumItems +
                                                          " items";
    else
        result = "The Recycle Bin is empty";
    recycleBinUI.lblRecycleBin.setText(result);
    return result;
}
```

SHEmptyRecycleBin() can be used to programmatically empty the recycle bin. You can use the SHERB_NOCONFIRMATION and SHERB_NOPROGRESSUI flags to not show the confirmation and progress dialogs respectively, and SHERB_NOSOUND to suppress the completion sound. The function and associated flags are described at http://msdn.microsoft.com/en-us/library/bb762160(VS.85).aspx. RecycleBin.emptyRecycleBin() uses this function to empty the recycle bin:


```
public void emptyRecycleBin(StringBuffer binDrive) {
      int flags = 0;
      if (binDrive.toString().equals(""))
              binDrive=null;

      // use user interface options to set flags
      // the checkboxes referenced below are shown in Fig 2-7
      if (recycleBinUI.chkNoConfirmationDialog.isSelected())
          flags = SHERB_NOCONFIRMATION;
      if (recycleBinUI.chkNoProgressDialog.isSelected())
          flags = flags | SHERB_NOPROGRESSUI;
      if (recycleBinUI.chkNoSound.isSelected())
          flags = flags | SHERB_NOSOUND;

      Shell32.SHEmptyRecycleBin(Util.getWindowHandle(recycleBinUI),
                                                    binDrive, flags);
}
```

Deleting Files at Reboot

To delete an in-use file that refuses to be deleted, the hack uses Kernel32.MoveFileEx(), passing the MOVEFILE_DELAY_UNTIL_REBOOT flag:

```
// mark the file for deletion.
// The file is deleted when the system reboots.
boolean retVal = Kernel32.MoveFileEx(
  recycleBinUI.txtFieldSource.getText(), // file to delete from Fig 2-7
  null, MOVEFILE_DELAY_UNTIL_REBOOT);
```

MoveFileEx() and flags it works with are described at http://msdn.microsoft.com/en-us/library/aa365240(VS.85).aspx. It marks files for deletion by storing the filename in the registry (Figure 2-8).

The Windows Registry
If you already didn't know, the Windows registry is a central database used to store and retrieve information needed by the applications and system components to configure the system features for users, applications and hardware devices. You can view the registry keys, subkeys and their values using the registry viewer. Registry viewer can be opened by typing 'regedit'from the run command in the start menu, or from the command prompt. For more information on Windows registry, you can read the MSDN documentation at http://msdn2.microsoft.com/en-us/library/ms724871.aspx.

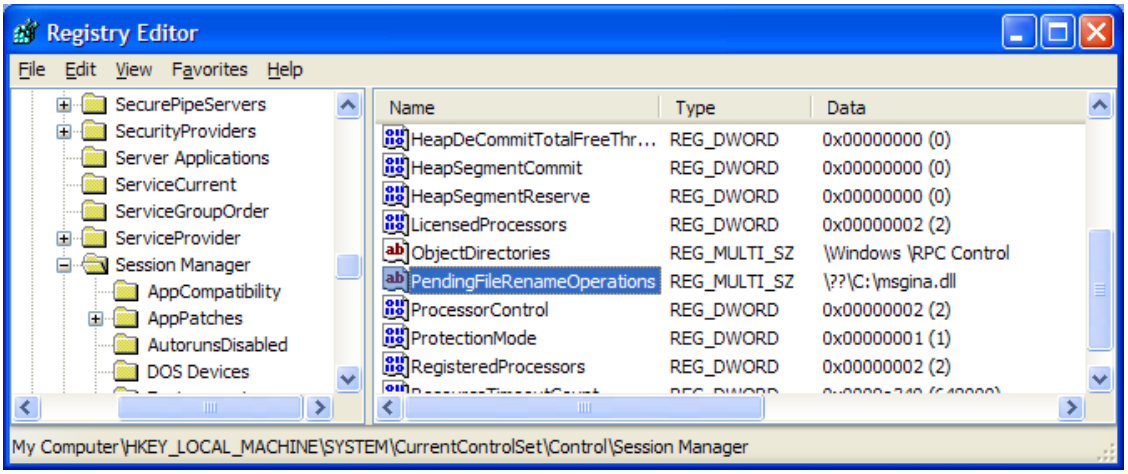


Figure 2-8 File marked for deletion in registry

The old and new (if any) filename is stored in the registry at:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\
                          Session Manager\PendingFileRenameOperations
```

When Windows reboots, the kernel reads this registry key and deletes such files. This function can come in handy when uninstalling applications. The uninstaller program can mark itself for deletion to ensure a clean uninstall. For MoveFileEx() to work, you must be an administrator, and have delete permissions on the file being deleted.

Hack 2.6: A new file type

Register a new file type and icon.

You've seen how the Windows shell categorizes files by types, and how you can query this information using SHGetFileInfo() in Hack 2.2. Now, let's talk about how you can create your own file type

Windows uses file extensions to determine file types. Files belonging to a particular type are shown with an icon representing that file type (see Figure 2-7). File types can have an associated application – when you double click a file, the associated application is launched and opens the file. Custom menu entries can also be registered for file types. When you right-click a file with a registered file type, these custom menu entries show up, and allow the user to easily Open, Preview, Print or do something else with the file.

For example, files that end with the ".doc" extension are treated as Microsoft Word documents. They are shown with a file icon representing Microsoft Word documents and double clicking these files opens them with Microsoft Word.

Windows knows about many standard file types. Those that it does not know about (like foofile.foo in Figure 2-9) are shown with the "Unknown file type" icon. They do not have

an associated application, and double clicking such a file opens up an 'unknown file type' dialog instead of opening the file.

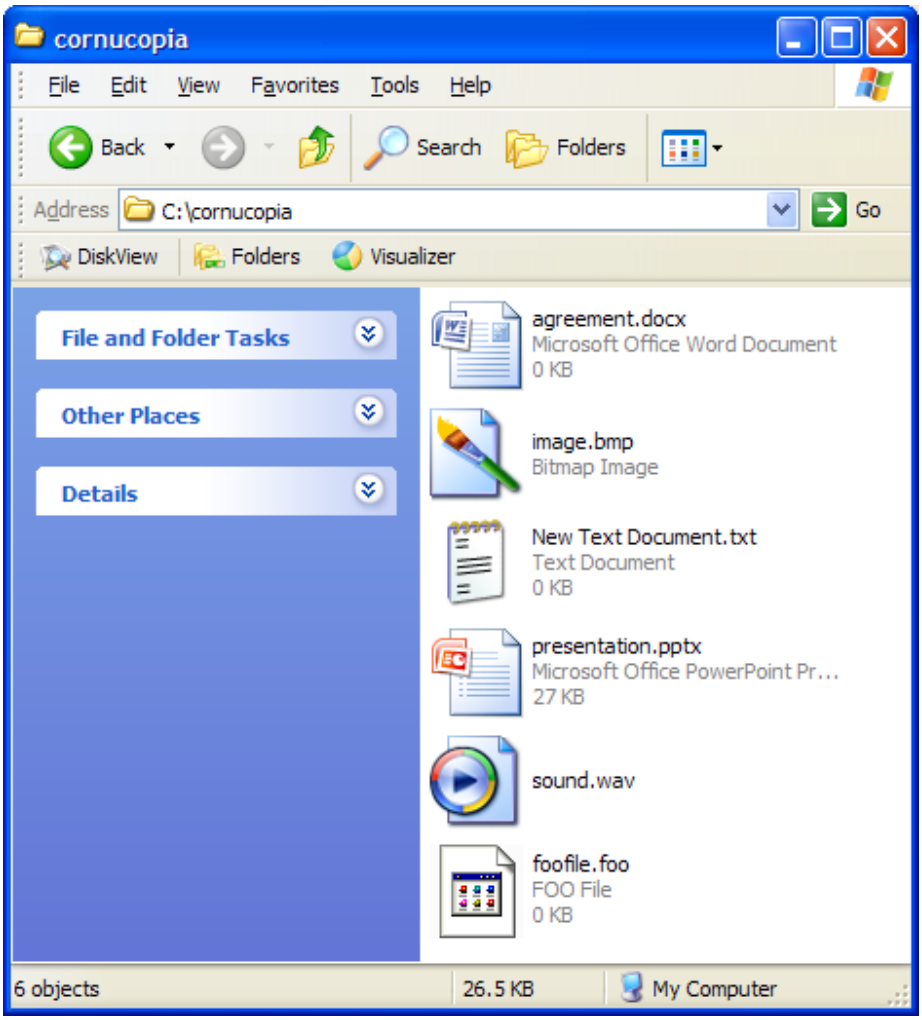


Figure 2-9 Files of different file types. Note that foofile.foo is shown with an unknown file icon.

Registering a new file type with Windows is easy. You can do it by adding a few registry keys. Let's say your program creates a files with the ".foo" extension. To create a new file type for Foo files, add a new registry key under HKEY_LOCAL_MACHINE\Software\Classes with the file extension as its name, i.e. ".foo" and a program identifier (ProgID) as it's default value. Figure 2-10 shows the registry editor after adding the ".foo" key with "Win32Hacks.FooApp.1" as its default value.

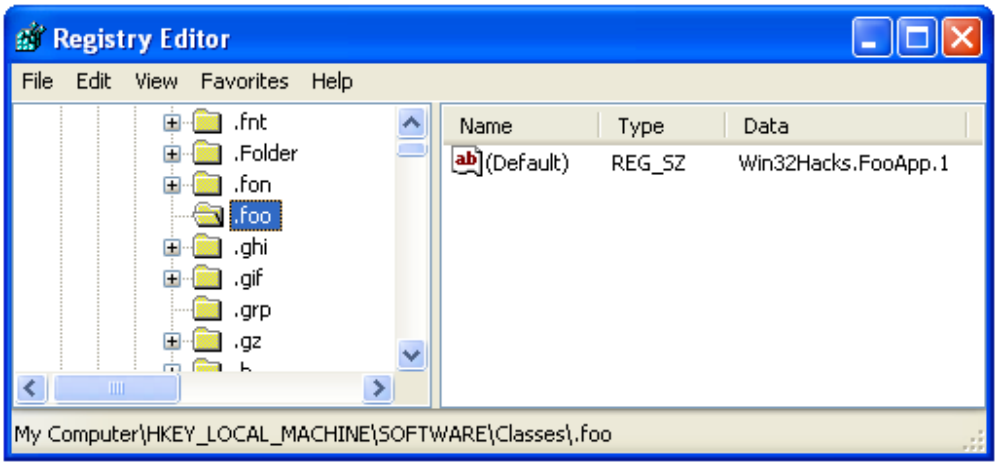


Figure 2-10 Adding a new file type for .foo with Win32Hacks.FooApp.1 as progid

The identifier can be any string, but Microsoft recommends that you name it in the form *CompanyName.ProductName.VersionNumber.* We'll use Win32Hacks.FooApp.1 as the identifier for our file type.

We also need to add another registry key with the identifier as the name and a descriptive string as its default value. To do this, create a key called "Win32Hacks.FooApp.1" with a default value of "My Foo File", as shown in Figure 2-11.

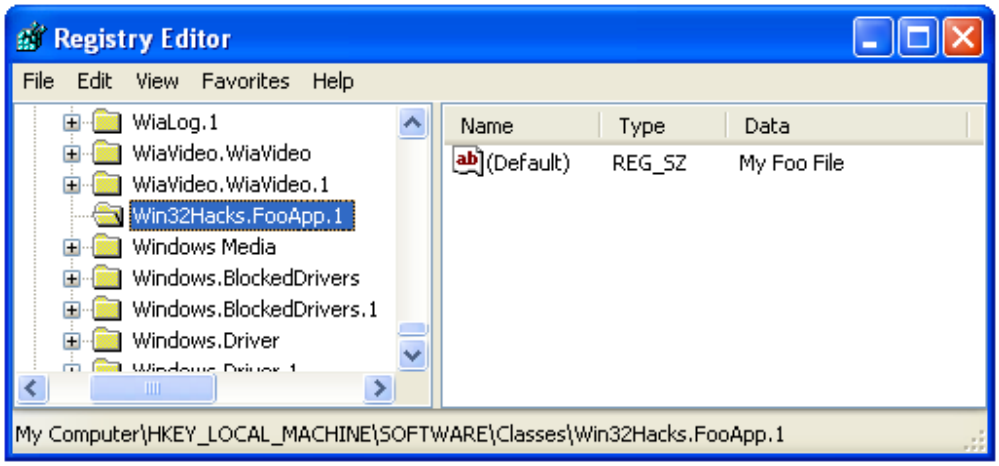


Figure 2-11 ProgID key (Win32Hacks.FooApp.1) with description as default value

To show files belonging to this file with a particular icon, we need to add a DefaultIcon sub-key with the path to an icon file as its default value. Figure 2-12 shows the DefaultIcon sub-key added to Win32Hacks.FooApp.1 with the icon path set to "C:\foo.ico".

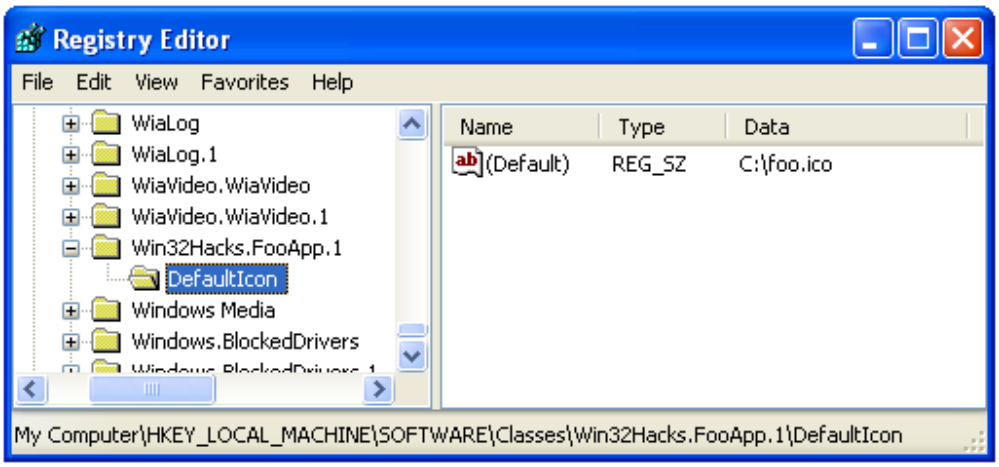


Figure 2-12 DefaultIcon sub key to specify the icon to represent .foo files

Although icon files (.ICO extension) can be created using Microsoft Paint, you'll probably use a software such as Axialis IconWorkshop (http://www.axialis.com/iconworkshop/) or Aha-soft's ArtIcons(http://www.aha-soft.com/articons/) for a professionally designed icon in multiple sizes and bit-depths.

This is all that is required to register a new file type with its own icon. You can control some aspects of the file type behavior using other related registry keys. These are described at http://msdn2.microsoft.com/en-us/library/bb776870(VS.85).aspx.

The New File Type Hack application (Figure 2-11) adds these registry keys programmatically. The task of adding the needed registry keys to register a new file type is performed by NewFileType.java, while NewFileTypeUI.java provides the GUI front-end.

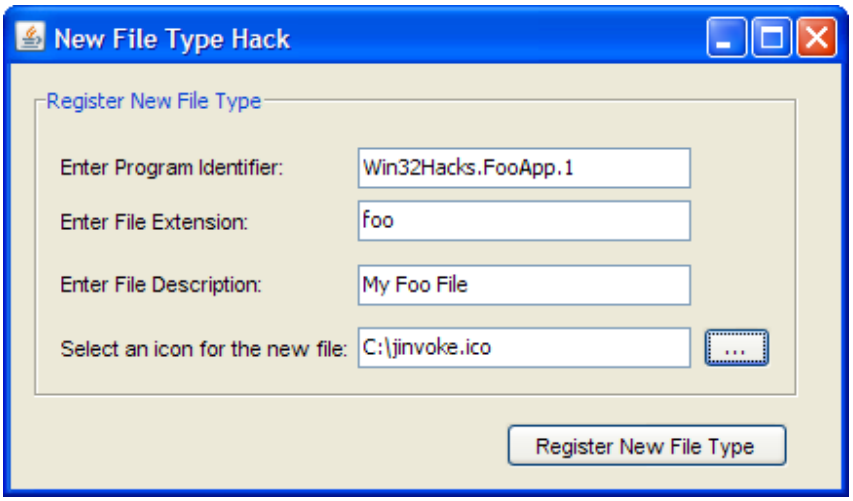


Figure 2-13 Adding a new file type using the New File Type hack application

NewFileType uses registerNewFileType() to create the new file type:


```
public void registerNewFileType(String fileExtension,
                                 String progID,
                                 String fileDescription,
                                 String iconPath) {
    // create HKEY_LOCAL_MACHINE\.foo with progID as default value
    // this adds the registry key shown in Figure 2-10
    String keyName = "." + fileExtension;
    String keyValue = progID;
    createKey(keyName, keyValue);

    // create HKEY_LOCAL_MACHINE\progID with description as value
    // this adds the registry key shown in Figure 2-11
    keyName = progID;
    keyValue = fileDescription;
    createKey(keyName, keyValue);

    // create HKEY_LOCAL_MACHINE\progID\DefaultIcon with path to icon
    // this adds the registry key shown in Figure 2-12
    keyName = progID + "\\DefaultIcon";
    keyValue = iconPath;
    createKey(keyName, keyValue);

    // notify the Windows shell of the new file type
    Shell32.SHChangeNotify(SHCNE_ASSOCCHANGED, SHCNF_IDLIST, 0, 0);
}
```

The task of creating the keys and setting their value is delegated to NewFileType.createKey(), which uses RegCreateKeyEx() and RegSetValue() from Advapi32.dll. These functions, along with others from the Registry API are documented at http://msdn.microsoft.com/en-us/library/ms724875(VS.85).aspx.

```
// import Win32 constants used in createKey()
import static com.jinvoke.win32.WinConstants.*;

private void createKey(String keyName, String keyValue) {
    // keyHandle is a single element array to hold the handle to the
    // newly created sub key. keyHandle[0] is set by RegCreateKeyEx
    // and used to set the key's default value using RegSetValue
    int[] keyHandle = { 0 };

    // If you write keys to a key under HKEY_CLASSES_ROOT, the system
    // stores the information under HKEY_LOCAL_MACHINE\Software\Classes
    Advapi32.RegCreateKeyEx(
       HKEY_CLASSES_ROOT,  // handle to parent key
       keyName,            // sub key to be created
       0,                  // reserved, must be 0
       null,               // reserved, must be null
       REG_OPTION_NON_VOLATILE,  // save the key
       KEY_ALL_ACCESS,     // standard rights required
       null,               // use default security descriptor
       keyHandle,          // used to get handle to newly created key
       null);              // we are not interested in disposition

    // set the default value – using empty string ("")
    Advapi32.RegSetValueEx(keyHandle[0],  // registry key handle
```



```
        "",                         // set the default value
        0,                          // reserved, must be 0
        REG_SZ,                     // set a String value
        keyValue,                   // string to set the value to
        (keyValue.length()+1)*2);   // length of keyValue in bytes
                                    // (+1 char for terminating null, and
}                                   //  *2 to account for wide characters)
```

After adding a new file type or changing an existing one, the system needs to be informed of the change. This is done by calling SHChangeNotify(), described at http://msdn.microsoft.com/en-us/library/bb762118(VS.85).aspx:

```
Shell32.SHChangeNotify(SHCNE_ASSOCCHANGED, SHCNF_IDLIST, 0, 0)
```

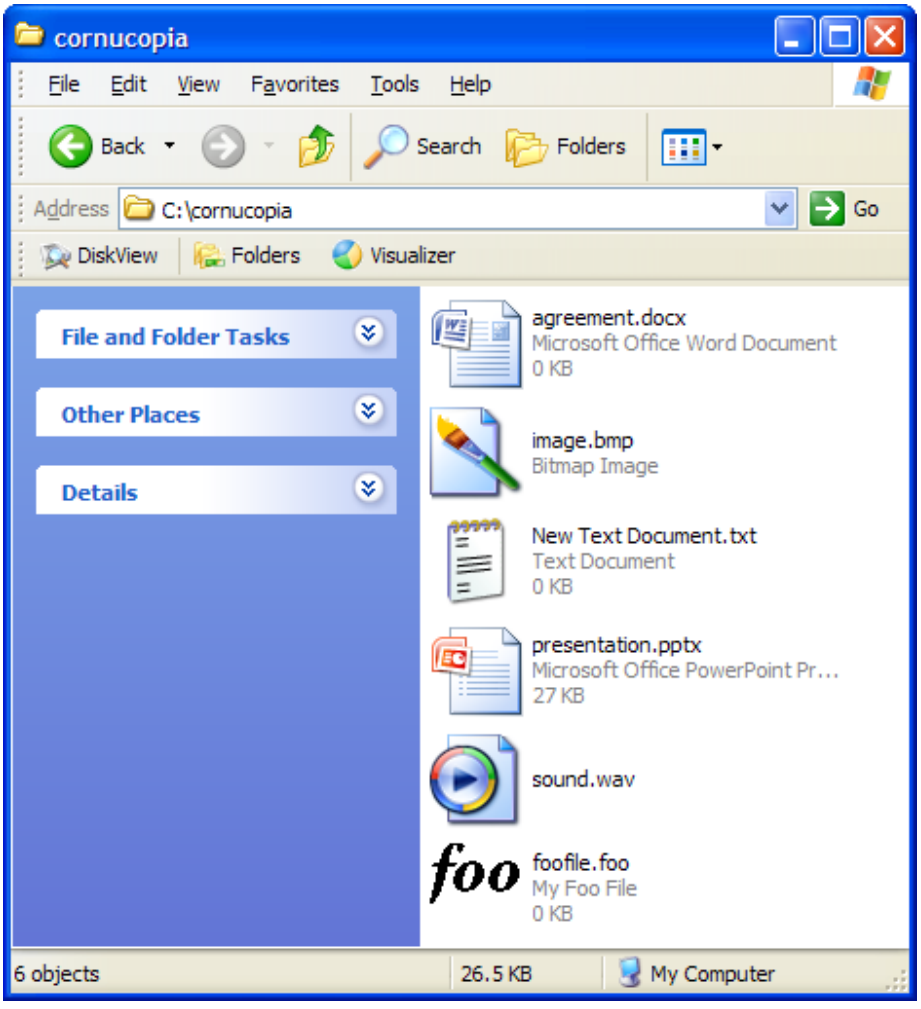


Figure 2-14 The foo type is now recognized by Windows Explorer

On calling this function, the new file type is recognized by Windows Explorer. As Figure 2-12 shows, .foo files are shown with a distinctive icon, and their type is based on the description we set.

Hack 2.7: Associated application

Associate a java application to open your file.

You've added a new file type for your file. Now, how can you associate it with your java application?

When a user double-clicks a file, the Windows shell performs a default action for that file type. The action is specified by the default value of the Shell sub-key of the file type's identifier. If the Shell key does not have a default value, the Open action is performed. The default value of the Command sub key specifies the command to be executed when file of that type is opened.

Continuing the example from the previous hack, the Shell, Open and Command keys shown in Figure 2-15 need to be added to associate an application with the Foo file type.

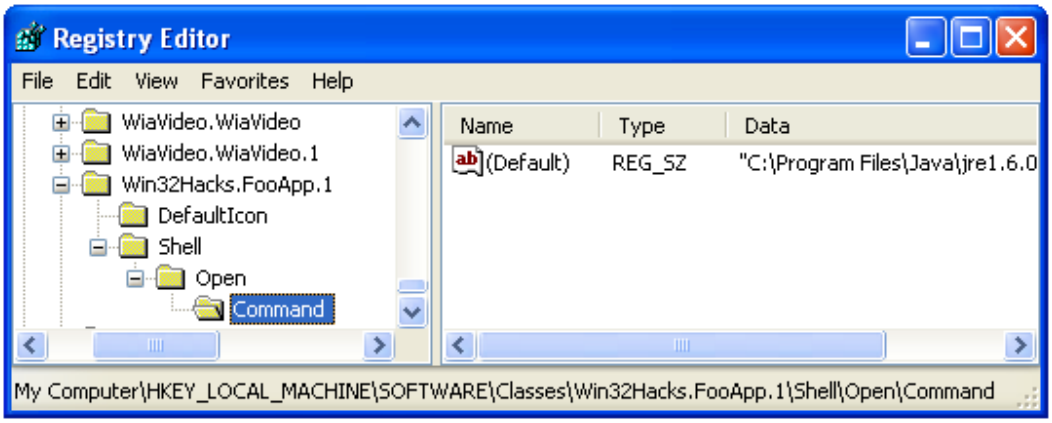


Figure 2-15 Adding a default application to open Foo files

When a file of our newly added type is double-clicked, we want our java application to be executed and passed the file-path as its first argument. If our java program is in a jar file at `"C:\Program Files\Win32Hacks\app.jar"`, we'll have to invoke the following command to accomplish this:

```
"C:\Program Files\Java\jre1.6.0_05\bin\javaw.exe" -jar "C:\Program
Files\Win32Hacks\app.jar" "%1"
```

Here `"%1"` is the path to the file to be opened – the shell fills this in before invoking the command. Java is installed in `C:\Program Files\Java\jre1.6.0_05` on my computer, so this is the command I need to give. However, Java maybe installed at a different path on another computer, so the path to javaw.exe needs to be determined first. The next section will explain how to obtain it.

Alert readers would have noticed that we're using javaw to launch the program, not java. There is a subtle but important difference among them. Java.exe is used when you are debugging a java program or running one that interacts with users through the standard

input and output. It runs within a command window. For GUI applications, we don't want users to see the ugly black command window to pop up just to start our program. We can do this using the javaw command – it is identical to java, except that a command prompt window is not shown.

Getting the path to Java

The Java Runtime installer adds registry keys that make it possible to get the path to the JRE. To get the path, we first need to find the version of the JRE installed on the system. This can be obtained from the CurrentVersion value of the HKLM\SOFTWARE\JavaSoft\Java Runtime Environment key. As Figure 2-16 shows, the JRE version on my computer is 1.6.

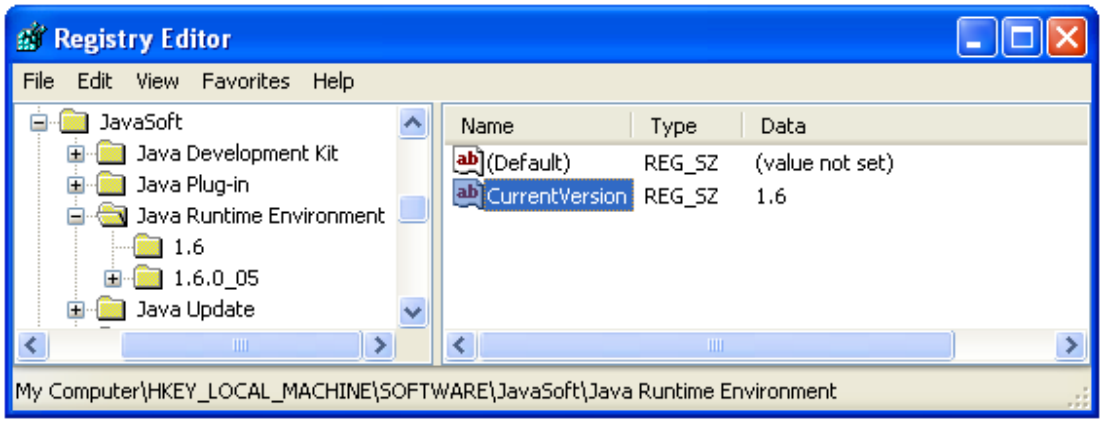


Figure 2-16 Registry keys showing the current JRE version

The JRE path is stored in a sub-key of the Java Runtime Environment key. We can obtain the path by querying the sub-key with the same name as the current version. The JavaHome value of the version sub-key provides the JVM path. Figure 2-17 shows that the JVM on my computer is at C:\Program Files\Java\jre1.6.0_05.

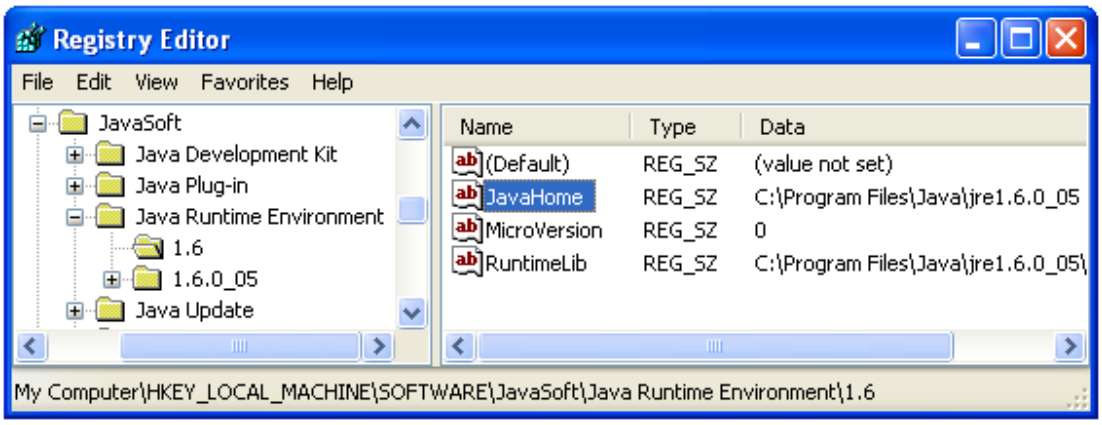


Figure 2-17 JVM path is present in JavaHome value of the *version* (1.6) sub key

To query the registry, we use another set of registry functions from Advapi32.dll. RegOpenKeyEx(), described at http://msdn.microsoft.com/en-us/library/ms724897(VS.85).aspx opens the named registry key and provides us with its handle, and RegQueryValueEx() retrieves its value as explained at http://msdn.microsoft.com/en-us/library/ms724911(VS.85).aspx. Once we are done using a registry key, we should close it using RegCloseKey(). This function is detailed at http://msdn.microsoft.com/en-us/library/ms724837(VS.85).aspx.

Fortunately, J/Invoke provides these functions in the com.jinvoke.win32.Advapi32 class, which we use to obtain the JRE path in getJRE():

```
// static import for Win32 constants referenced in getJREPath()
import static com.jinvoke.win32.WinConstants.*;

// registry keys and values to query for getting the JRE path
private static final String REGISTRY_JRE_KEYNAME =
                        "SOFTWARE\\JavaSoft\\Java Runtime Environment";
private static final String REGISTRY_JRE_JAVAHOME_VALUENAME =
                        "JavaHome";

// this method queries the JRE path using Registry lookups
private static String getJREPath(){
      // single element array to hold registry key handle
      int[] hkey = { 0 };

      // open the Java Runtime Environment key for read access
      if (Advapi32.RegOpenKeyEx(HKEY_LOCAL_MACHINE,
            REGISTRY_JRE_KEYNAME, 0, KEY_READ, hkey) != ERROR_SUCCESS)
            return null;

      int[] lpType = { 0 };
      int[] lpcbData= { MAX_PATH };
      // byte array to read in the CurrentVersion value
      byte[] sCurrentVersion = new byte[MAX_PATH];

      // query the CurrentVersion value from the
      // Java Runtime Environment key as shown in Figure 2-16
      if (Advapi32.RegQueryValueEx(hkey[0], "CurrentVersion",
            null, lpType, sCurrentVersion, lpcbData) != ERROR_SUCCESS){
            // in case of error, close the key and bail out
            Advapi32.RegCloseKey(hkey[0]);
            return null;
      }

      // convert the byte array representation of the CurrentVersion
      // value to a Java string
      String currVersion = "";
      try {
            // Win32 uses UTF16 Little Endian encoding (UTF-16LE)
            currVersion = new String(sCurrentVersion, "UTF-16LE");
      } catch (UnsupportedEncodingException e) {
            e.printStackTrace();
      }
```



```
      // As we are using the Unicode character set, the number of bytes
      // copied into sCurrentVersion is twice the length of the string
      // (including a trailing null character). To get the string
      // length in Java, divide by size of a wchar_t (i.e. 2
      // bytes)and subtract one character for trailing null,

      int strlen = lpcbData[0]/2 – 1;

      // trim the null characters from the end of the string
      currVersion = currVersion.substring(0, strlen);

      // open the version sub-key for read access
      if (Advapi32.RegOpenKeyEx(HKEY_LOCAL_MACHINE,
            REGISTRY_JRE_KEYNAME+"\\"+currVersion, 0, KEY_READ, hkey)
            != ERROR_SUCCESS) {
            return null;
      }

      // query the JVM path from the JavaHome value of the sub-key
      // (programmatically obtain the value shown in Figure 2-17)

      // each character is 2 bytes in Unicode and the maximum path
      // could be MAX_PATH characters long, (i.e. MAX_PATH*2 byes)
      lpcbData[0] = MAX_PATH*2;

      // byte array to read in the JVM path
      byte[] sJREJavaHome = new byte[MAX_PATH*2];
      if (Advapi32.RegQueryValueEx(hkey[0],
                  REGISTRY_JRE_JAVAHOME_VALUENAME, null, lpType,
                  sJREJavaHome, lpcbData) != ERROR_SUCCESS) {
            // in case of error, close the key and bail out
            Advapi32.RegCloseKey(hkey[0]);
            return null;
      }

      // convert the byte array representation of path to a Java string
      String JREHome = "";
      try {
            JREHome = new String(sJREJavaHome, "UTF-16LE");
      } catch (UnsupportedEncodingException e) {
            e.printStackTrace();
      }
      JREHome = JREHome.substring(0, lpcbData[0]/2 - 1); // as before

      return JREHome;
}
```

Given the path to the JRE and to the application jar file, it is easy to construct the command string needed to be open the file path with the associated application:

```
private static String getCommand(String jarPath) {
      String jrehome = getJREPath();
      // append "\bin\javaw.exe" to JavaHome
      String javawPath = jrehome + File.separator + "bin" +
                                   File.separator + "javaw.exe";
```



```
        // construct the command string with a trailing "%1"
        String command = "\"" + javawPath + "\" -jar \"" +
                                 jarPath + "\" \"%1\"";
        return command;
}
```

As discussed before, the Windows shell fills in `"%1"` with the path to the file to be opened. The associated java application can thus get the file that was double clicked and perform some operation on it.

Once we have the command string ready, we can associate the file type with our Java application by creating the <ProgID>\shell\open\command registry key and setting the command string as its default value. The associated java application for this hack enhances the New File Type application from the previous hack(Hack 2.7) by creating this additional key from within registerNewFileType() :

```
public void registerNewFileType(String fileExtension, String progID,
          String fileDescription, String iconPath, String jarPath) {
   //...skipping code to register new file type from Hack 2-6

   // associate the executable jar at jarPath with this file type
   keyName = progID + "\\Shell\\Open\\Command";
   keyValue = getCommand(jarPath);
   createKey(keyName, keyValue);

   // notify the Windows shell of the new file type
   Shell32.SHChangeNotify(SHCNE_ASSOCCHANGED, SHCNF_IDLIST, 0, 0);
}
```

This adds the registry keys shown in Figure 2-15. When the new .foo file type is associated with the executable jar for File Information (Hack 2-1), double clicking any .foo file shows File information for that file.

Hack 2.8: Customize file context menu

Add a command to a file's context menu.

We've seen how we can create our own file types, and associate our Java application with them. We can also add custom commands to ours as well as existing file types just as easily.

Fig. 2-18 shows a custom context-menu ("Check Version") added to exe files. When selected, this menu invokes the "File Version Hack" app from Hack 2.2 (Version Information – Figure 2-2). We also see context menus for "7-Zip" and "Eluent Tools". These have been added by other third-party applications to integrate better with the Windows Shell.

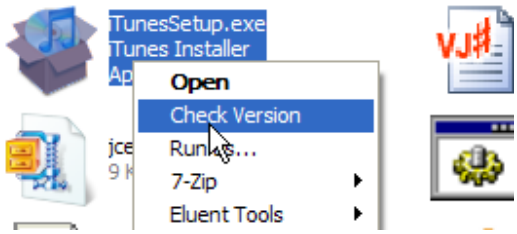


Figure 2-18 Custom menu to 'Check Version' for exe files

AddContextMenu.java, in the associated code for this hack, adds a custom context menu to all exe files.

The default value of the HKEY_CLASSES_ROOT\.exe key indicates that the program identifier for .exe extension is exefile (Figure 2-19). Thus, we'll have to create registry keys under the HKEY_CLASSES_ROOT\exefile key to customize the context menu for exe files.

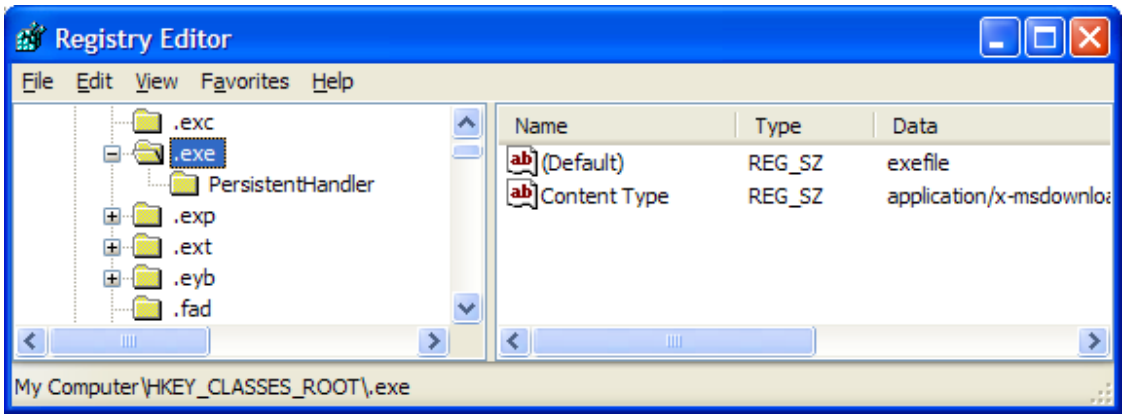


Figure 2-19 Program identifier for .exe is exefile

We need to create a new key under the shell sub-key for our custom action. The name of the key can be any unique name and its default value is shown in the file context menu. Figure 2-20 shows the checkversion key with a default value of "Check Version" added to the HKEY_CLASSES_ROOT\exefile\shell key. It's worth noting that the shell key already contains open and runas subkeys. In fact, these keys are part of the default Windows installation and are responsible for the "Open" and "Run As…" menu items seen in Figure 2-18.

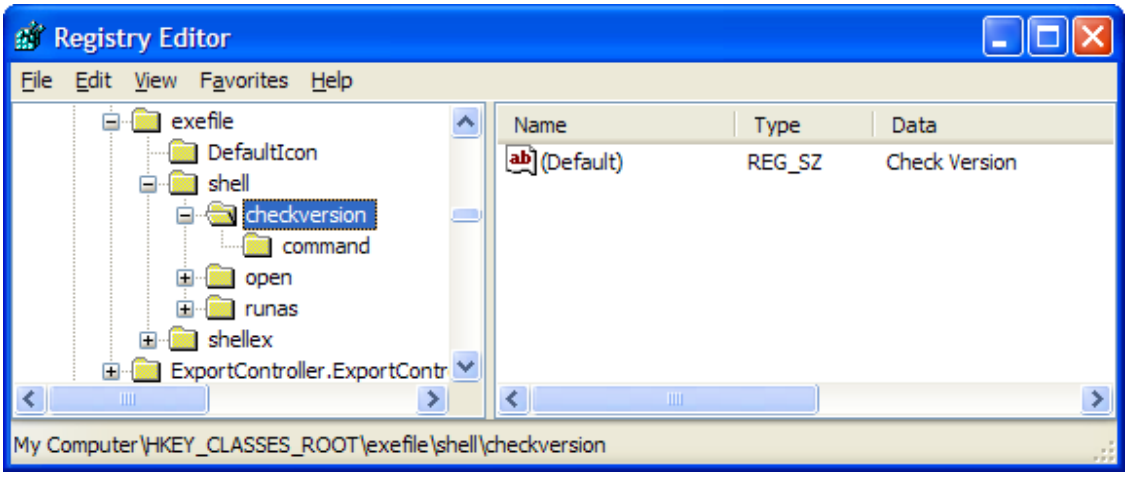


Figure 2-20 checkversion key added to HKEY_CLASSES_ROOT\exefile\shell

The command to be invoked when the menu item is selected is under the command sub-key. As Figure 2-21 shows, it's the full path to java.exe followed by arguments to invoke the version information hack from Hack 2-2:

```
"C:\Program Files\Java\jre1.6.0_05\bin\javaw.exe" -jar
"C:\versioninfo.jar" "%1"
```

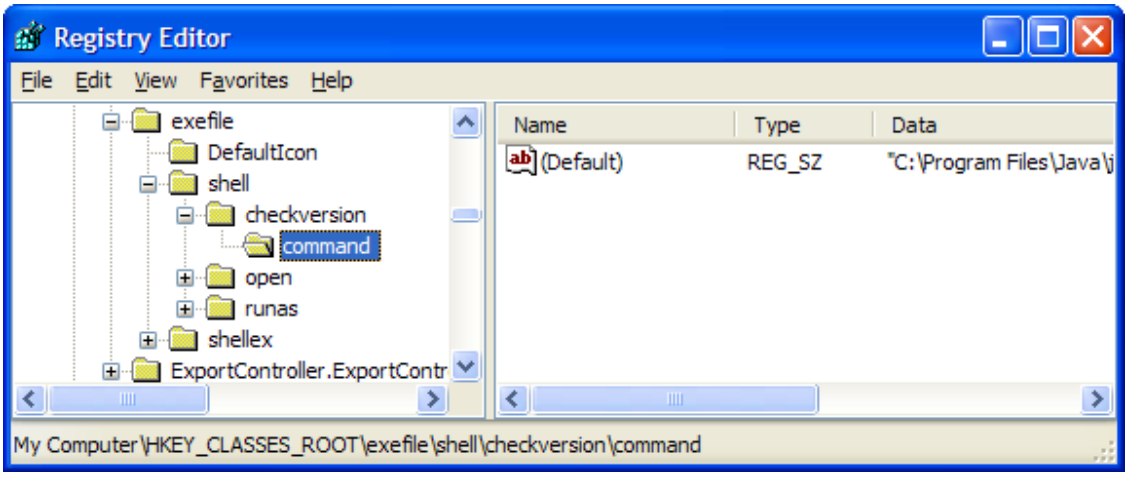


Figure 2-21 command subkey contains the command to be invoked

The Java code to add a custom menu to exe files that runs a jar file is thus quite simple:

```
public static void addContextMenu(String menuText, String jarPath) {
      // specify the menu text to be shown to the user
      String keyName = "exefile\\shell\\checkversion";
      String keyValue = menuText;
      // adds the checkversion key shown in Figure 2-20
      createKey(keyName, keyValue);

      // specify the command to be invoked
```



```
      keyName = "exefile\\shell\\checkversion\\command";
      keyValue = getCommand(jarPath);
      // adds the command key shown in Figure 2-21
      createKey(keyName, keyValue);

      // notify the Windows shell of the change
      Shell32.SHChangeNotify(SHCNE_ASSOCCHANGED, SHCNF_IDLIST, 0, 0);
}
```

The command to run the executable jar file can be derived using getCommand()described in Hack 2.7 (Associated Application) and we use createKey() discussed in Hack 2.6 (New File Type) to add the needed registry keys. addContextMenu() is called from the main method of AddContextMenu.java:

```
public static void main(String[] args) {
      String menuText = "Check Version";
      String jarPath  = "C:\\versioninfo.jar";
      if (args.length == 2) {
            menuText = args[0];
            jarPath  = args[1];
      }
      addContextMenu(menuText, jarPath);
      System.out.println("Added context menu for \"" + menuText +
                                    "\" to run \"" + jarPath + "\".");
}
```

The main method in AddContextMenu.java uses command line arguments to get the menu text and the jar path, and calls addContextMenu() to add the needed registry keys. The Windows Shell reads the keys and shows the context menu to invoke our java application.

