IN THIS CHAPTER

- Examining the File class
- Understanding command-line parameters
- Introducing the JFileChooser class

Bonus Chapter 1
Working with Files

In this chapter, you discover the ins and outs of working with files and directories. I don't show you how to read or write files (for that info, you should refer to Chapter 2 of the bonus content online)), but you do find out how to find files on the hard drive; create, delete, and rename files; and work with directories. You find out how to use the Swing file-chooser dialog box that lets you add filing capabilities to Swing applications. Finally, you find out how to retrieve parameters from the command line — a useful technique, because command-line parameters are often used to pass file information to console programs.

Beginning with Java 1.7, you have two choices for working with files and directories: You can use the original `File` class or you can use the new `Path` class, which is part of a new file-processing package called NIO.2. (*NIO.2* stands for *New I/O version 2*.) This chapter covers both of these classes.

Using the File Class

The `File` class is your key to processing files and directories. A `File` object represents a single file or directory. Note that the file or directory doesn't actually have to exist on your hard drive. Instead, the `File` object represents a file that may or may not actually exist.

TECHNICAL STUFF

Java uses a single class to represent both files and directories because a directory is actually nothing more than a special type of file. I suppose that the designers of Java could have created a separate `Directory` class to represent directories, but then you'd have to know how to use two classes instead of one.

The `File` class is in the `java.io` package, so any program that uses it should import `java.io.File` or `java.io.*`.

TIP

Java 1.7 introduces a new `Path` class, which is designed to replace the `File` class.

Knowing the class constructors and methods

Table 1-1 lists the main constructors and methods of the `File` class.

TABLE 1-1 The File Class

Constructor	Description
File(String pathname)	Creates a file with the specified pathname.
Field	**Description**
`String separator`	Separates components of a pathname on this system; usually is \ or /.
Method	**Description**
`boolean canRead()`	Determines whether the file can be read.
`boolean canWrite()`	Determines whether the file can be written.
`boolean createNewFile()`	Creates the file on the hard drive if it doesn't exist. The method returns `true` if the file was created or `false` if the file already existed. Throws `IOException`.
`boolean delete()`	Deletes the file or directory. The method returns `true` if the file was deleted successfully.
`boolean exists()`	Returns `true` if the file exists on the hard drive and `false` if the file doesn't exist.
`String getCanonicalPath()`	Returns the complete path to the file, including the drive letter if run on a Windows system; throws `IOException`.
`String getName()`	Gets the name of this file.
`String getParent()`	Gets the name of the parent directory of this file or directory.
`File getParentFile()`	Gets a `File` object representing the parent directory of this file or directory.

`boolean isDirectory()`	Returns `true` if this `File` object is a directory or `false` if it is a file.
`boolean isFile()`	Returns `true` if this `File` object is a file or `false` if it is a directory.
`boolean isHidden()`	Returns `true` if this file or directory is marked by the operating system as hidden.
`long lastModified()`	Returns the time when the file was last modified, expressed in milliseconds since 0:00:00 a.m., January 1, 1970.
`long length()`	Returns the size of the file in bytes.
`String[] list()`	Returns an array of `String` objects with the name of each file and directory in this directory. Each string is a simple filename, not a complete path. If this `File` object is not a directory, the method returns `null`.
`File[] listFiles()`	Returns an array of `File` objects representing each file and directory in this directory. If this `File` object is not a directory, the method returns `null`.
`static File[] listRoots()`	Returns an array containing a `File` object for the root directory of every file system available on the Java runtime. Unix systems usually have just one root, but Windows systems have a root for each drive.
`boolean mkdir()`	Creates a directory on the hard drive from this `File` object. The method returns `true` if the directory was created successfully.
`boolean mkdirs()`	Creates a directory on the hard drive from this `File` object, including any parent directories that are listed in the directory path but don't already exist. The method returns `true` if the directory was created successfully.
`boolean renameTo(File dest)`	Renames the `File` object to the specified destination `File` object. The method returns `true` if the rename was successful.
`boolean setLastModified(long time)`	Sets the last modified time for the `File` object. The method returns `true` if the time was set successfully.
`boolean setReadOnly()`	Marks the file as read-only. The method returns `true` if the file was marked successfully.
`String toString()`	Returns the pathname for this file or directory as a string.

Creating a File object

To create a `File` object, you call the `File` constructor, passing a string representing the filename as a parameter. Here's an example:

```
File f = new File("hits.log");
```

Here the file's name is `hits.log`, and it lives in the current directory, which usually is the directory from which the Java Virtual Machine (JVM) was started.

If you don't want the file to live in the current directory, you can supply a complete pathname in the parameter string. You're now entering one of the few areas of Java that becomes system-dependent, however, because the way you write pathnames depends on the operating system you're using. The pathname `c:\logs\hits.log` is valid for Windows systems, for example, but not on Linux or Macintosh systems, which don't use drive letters and use forward slashes instead of backslashes to separate directories. (On a Linux or Mac, use `/logs/hits.log` instead.)

REMEMBER

If you hard-code pathnames as string literals, the backslash character is the escape character for Java strings. Thus you must code two backslashes to get one backslash in the pathname. You must code the path `c:\logs\hits.log` like this:

```
String path = "c:\\logs\\hits.log";
```

Creating a file

Creating a `File` object doesn't create a file on the hard drive. Instead, it creates an in-memory object that represents a file or directory that may or may not actually exist on the hard drive. To find out whether the file or directory exists, you can use the `exists` method, as in this example:

```
File f = new File(path);
if (!f.exists())
    System.out.println
        ("The input file does not exist!");
```

Here an error message is displayed on the console if the file doesn't exist.

To create a new file on the hard drive, create a `File` object with the filename you want to use and then use the `createNewFile` method, like this:

```
File f = new File(path);
if (f.createNewFile())
    System.out.println("File created.");
else
    System.out.println("File could not be created.");
```

Note that the `createNewFile` method returns a boolean that indicates whether the file was created successfully. If the file already exists, `createNewFile` returns `false`, so you don't have to use the `exists` method before you call `createNewFile`.

REMEMBER

When you create a file with the `createNewFile` method, the file doesn't have anything in it. If you want the file to contain data, you can use the classes I describe in Chapter 2 of the bonus content, to write information to the file.

Getting information about a file

Several of the methods of the `File` class simply return information about a file or directory. You can find out whether the `File` object represents a file or directory, for example, by calling its `isDirectory` or `isFile` method. Other methods let you find out whether a file is read-only or hidden, or retrieve the file's age and the time when it was last modified.

You can get the name of the file represented by a `File` object in several popular ways:

- To get just the filename, use the `getName` method. This method returns a string that includes just the filename, not the complete path.
- To get the path that was specified to create the `File` object (such as `\logs\hit.log`), use the `toString` method instead.
- To get the full path for a file — that is, the complete path including the drive letter (for Windows systems) and all the directories and subdirectories leading to the file — use the `getCannonicalPath` method. This method removes any system-dependent oddities such as relative paths, dots (which represent the current directory), and double dots (which represent the parent directory) to get the file's actual path.

Getting the contents of a directory

A *directory* is a file that contains a list of other files or directories. Because a directory is just a special type of file, it's represented by an object of the `File` class. You can tell whether a particular `File` object is a directory by calling its `isDirectory` method. If this method returns `true`, the `File` is a directory. Then you can get an array of all the files contained in the directory by calling the `listFiles` method.

The following code snippet lists the name of every file in a directory whose pathname is stored in the `String` variable path:

```
File dir = new File(path);
if (dir.isDirectory())
{
    File[] files = dir.listFiles();
```

```
    for (File f : files)
        System.out.println(f.getName());
}
```

The following snippet is a little more selective because it lists only files, not subdirectories, and doesn't list hidden files:

```
File dir = new File(path);
if (dir.isDirectory())
{
    File[] files = dir.listFiles();
    for (File f : files)
    {
        if (f.isFile() && !f.isHidden())
            System.out.println(f.getName());
    }
}
```

TIP

Directory listings are especially well suited to recursive programming, because each `File` object returned by the `listFiles` method may be another directory that itself has a list of files and directories. For an explanation of recursive programming and an example that lists directories recursively, see Book 5, Chapter 3.

Renaming files

You can rename a file by using the `renameTo` method. This method uses another `File` object as a parameter that specifies the file you want to rename the current file to. It returns a `boolean` value that indicates whether the file was renamed successfully.

The following statements change the name of a file named `hits.log` to `savedhits.log`:

```
File f = new File("hits.log");
if (f.renameTo(new File("savedhits.log")))
    System.out.println("File renamed.");
else
    System.out.println("File not renamed.");
```

Depending on the capabilities of the operating system, the `renameTo` method can also move a file from one directory to another. This code moves the file `hits.log` from the folder `logs` to the folder `savedlogs`:

```
File f = new File("logs\\hits.log");
if (f.renameTo(new File("savedlogs\\hits.log")))
```

```
    System.out.println("File moved.");
else
    System.out.println("File not moved.");
```

TIP

Always test the return value of the `renameTo` method to make sure that the file was renamed successfully.

Deleting a file

To delete a file, create a `File` object for the file and then call the `delete` method, as in this example:

```
File f = new File("hits.log");
if (f.delete())
    System.out.println("File deleted.");
else
    System.out.println("File not deleted.");
```

If the file is a directory, the directory must be empty to be deleted.

TIP

With some recursive programming, you can create a method that deletes a non-empty directory — but be sure to heed the upcoming warning. The method looks something like this:

```
private static void deleteFile(File dir)
{
    File[] files = dir.listFiles();
    for (File f : files)
    {
    if (f.isDirectory())
        deleteFile(f);
    else
        f.delete();
    }
    dir.delete();
}
```

Then, to delete a folder named `folder1` along with all its files and subdirectories, call the `deleteFile` method:

```
deleteFile(new File("folder1");
```

WARNING

This feature is extremely dangerous to add to a program. Don't use it without first testing it carefully. If you accidentally delete all the files on your hard drive, don't blame me!

Using Command-Line Parameters

Ever since Book 1, Chapter 1, I've used this construction in every Java program presented so far:

```
public static void main(String[] args)
```

It's high time that you find out what the `args` parameter of the `main` method is used for. The `args` parameter is an array of strings that lets you access any command-line parameters that are specified by the user when he or she runs your program.

Suppose that you run a Java program named `Test` from a command program like this:

```
C:\>java Test the quick brown fox
```

In this case, the Java program is passed four parameters: `the`, `quick`, `brown`, and `fox`. You can access these parameters via the `args` array.

Suppose that the `main` method of the `Test` class is written like this:

```
public static void main(String[] args)
{
    for (String s : args)
        System.out.println(s);
}
```

Then the program displays the following output on the console when run with the command shown a few paragraphs back:

```
the
quick
brown
fox
```

Command-line parameters are useful in Java programs that work with files as a way to pass pathnames to the program. Here's a program that lists all the files in a directory passed to the program as a parameter:

```
import java.io.*;
public class ListDirectory
{
    public static void main(String[] args)
```

```
    {
        if (args.length > 0)
        {
            String path = args[0];
            File dir = new File(path);
            if (dir.isDirectory())
            {
                File[] files = dir.listFiles();
                for (File f : files)
                {
                    System.out.println(f.getName());
                }
            }
            else
                System.out.println("Not a directory.");
        }
    }
}
```

Choosing Files

For the most part, you don't want to mess around with command-line parameters in Swing applications. Instead, you want to use the `JFileChooser` class to let users pick the files they want to work with. This class lets you display Open and Save dialog boxes similar to the ones you've seen in other graphic user interface (GUI) applications with just a few lines of code.

Figure 1-1 shows an Open dialog box created with just these two lines of code:

```
JFileChooser fc = new JFileChooser();
int result = fc.showOpenDialog(this);
```

This code appears in a frame class that extends the `JFrame` class, so the `this` keyword in the `showOpenDialog` call refers to the parent frame.

The result returned by the `showOpenDialog` method indicates whether the user chose to open a file or click Cancel, and the `JFileChooser` class provides a handy `getSelectedFile` method that you can use to get a `File` object for the file selected by the user.

REMEMBER

The important thing to know about the `JFileChooser` class is that it doesn't actually open or save the file selected by the user; instead, it returns a `File` object for the file the user selects. Your program has the task of opening or saving the file.

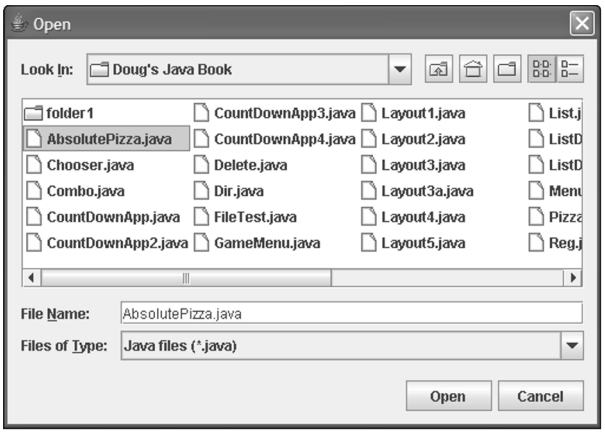

FIGURE 1-1: An Open dialog box displayed by the JFileChooser class.

The `JFileChooser` class has many additional methods that you can use to tailor its appearance and behavior in just about any way imaginable. Table 1-2 lists the commonly used constructors and methods of this powerful class.

TABLE 1-2 The JFileChooser Class

Constructor	Description
`JFileChooser()`	Creates a file chooser that begins at the user's default directory. On Windows systems, this directory is usually `My Documents`.
`JFileChooser(File file)`	Creates a file chooser that begins at the location indicated by the file parameter.
`JFileChooser(String path)`	Creates a file chooser that begins at the location indicated by the path string.
Method	**Description**
`void addChoosableFileFilter (FileFilter filter)`	Adds a file filter to the chooser.
`File getSelectedFile()`	Returns a `File` object for the file selected by the user.
`File[] getSelectedFiles()`	Returns an array of `File` objects for the files selected by the user if the file chooser allows multiple selections.
`void setAcceptAllFileFilterUsed(bo olean value)`	If `false`, removes the All Files filter from the file chooser.
`void setApproveButtonText(String text)`	Sets the text for the Approve button.

`void setDialogTitle(String title)`	Sets the title displayed by the file-chooser dialog box.
`void setFileHidingEnabled(boolean value)`	Doesn't show hidden files if `true`.
`void setMultiSelectionEnabled(boolean value)`	Allows the user to select more than one file if `true`.
`int showDialog(Component parent, String text)`	Displays a custom dialog box with the specified text for the Accept button. The return values are `JFileChooser.CANCEL_OPTION`, `APPROVE_OPTION`, and `ERROR_OPTION`.
`void setFileSelectionMode(int mode)`	Determines whether the user can select files, directories, or both. The parameter can be specified as `JFileChooser.FILES_ONLY`, `DIRECTORIES_ONLY`, or `FILES_AND_DIRECTORIES`.
`int showOpenDialog(Component parent)`	Displays an Open dialog box. The return values are the same as for the `showDialog` method.
`int showSaveDialog(Component parent)`	Displays a Save dialog box. The return values are the same as for the `showDialog` method.

Creating an Open dialog box

As you've just seen, you can create an Open dialog box with just a few lines of code. First, you call the `JFileChooser` constructor to create a `JFileChooser` instance; then you call the `showOpenDialog` method to display an Open dialog box.

If you don't pass a parameter to the constructor, the file chooser starts in the user's default directory, which on most systems is the operating system's current directory. If you want to start in some other directory, you have two options:

- Create a `File` object for the directory and then pass the `File` object to the constructor.
- Pass the pathname for the directory where you want to start to the constructor.

The `JFileChooser` class also includes methods that let you control the appearance of the chooser dialog box. You can use the `setDialogTitle` method to set the title (the default is `Open`), for example, and you can use the `setFileHidingEnabled` method to control whether hidden files are shown. If you want to allow the user to select more than one file, use the `setMultiSelectionEnabled` method.

A `setFileSelectionMode` method lets you specify whether users can select files, directories, or both. The options for this method need a little explanation:

- `JFileChooser.FILES_ONLY`: With this option (which is the default), the user can choose files only with the file-chooser dialog box. The user can navigate directories in the file-chooser dialog box but can't actually select a directory.
- `JFileChooser.DIRECTORIES_ONLY`: With this option, the user can select only directories, not files. One common use for this option is to let the user choose a default location for files used by your application without actually opening a file.
- `JFileChooser.FILES_AND_DIRECTORIES`: This option lets the user select either a file or a directory. For most applications, you want the user to pick either a file or a directory, but not both, so you probably won't use this option much.

TIP

In addition to an Open dialog box, you can display a Save dialog box by calling the `showSaveDialog` method. A Save dialog box is similar to an Open dialog box but has different default values for the title and the text shown on the Approve button. Otherwise these dialog boxes work pretty much the same way.

Getting the selected file

The file-chooser dialog box is a *modal* dialog box, which means that after you call the `showOpenDialog` method, your application is tied up until the user closes the file-chooser dialog box by clicking the Open or Cancel button.

You can find out which button the user clicked by inspecting the value returned by the `showOpenDialog` method:

- If the user clicked Open, the return value is `JFileChooser.APPROVE_OPTION`.
- If the user clicked Cancel, the return value is `JFileChooser.CANCEL_OPTION`.
- If an I/O (input/output) or other error occurred, the return value is `JFileChooser.ERROR_OPTION`.

Assuming that the `showOpenDialog` method returns `APPROVE_OPTION`, you can use the `getSelectedFile` method to get a `File` object for the file selected by the user. Then you can use this `File` object elsewhere in the program to read or write data.

Putting it all together, then, here's a method that displays a file-chooser dialog box and returns a `File` object for the file selected by the user. If the user cancels or an error occurs, the method returns `null`.

```
private File getFile()
{
    JFileChooser fc = new JFileChooser();
    int result = fc.showOpenDialog(null);
    File file = null;
    if (result == JFileChooser.APPROVE_OPTION)
        file = fc.getSelectedFile();
    return file;
}
```

You can call this method from an action event handler when the user clicks a button, selects a menu command, or otherwise indicates that he or she wants to open a file.

Using file filters

The file-chooser dialog box includes a Files of Type drop-down list filter that the user can use to control what types of files are displayed by the chooser. By default, the only item available in this drop-down list is All Files, which doesn't filter the files at all. If you want to add another filter to this list, you must create a class that extends the `FileFilter` abstract class and then pass an instance of this class to the `addChoosableFileFilter` method.

Table 1-3 lists the methods of the `FileFilter` class. Fortunately, it has only two methods that you need to implement. This class is in the `javax.swing.filechooser` package.

TABLE 1-3 The FileFilter Class

Method	Description
`public boolean abstract accept(File f)`	You must implement this method to return `true` if you want the file to be displayed in the chooser or `false` if you don't want the file to be displayed.
`public String abstract getDescription()`	You must implement this method to return the description string that is displayed in the Files of Type drop-down list in the chooser dialog box.

The `getDescription` method simply returns the text displayed in the Files of Type drop-down list. You usually implement it with a single return statement that returns the description. Here's an example:

```
public String getDescription()
{
    return "Java files (*.java)";
}
```

Here the string `Java files (*.java)` is displayed in the Files of Type drop-down list.

The `accept` method does the work of a file filter. The file chooser calls this method for every file it displays. The file is passed as a parameter. The `accept` method returns a boolean that indicates whether the file is displayed.

The `accept` method can use any criteria it wants to decide which files to accept and which files to reject. Most filters do this based on the file-extension part of the filename. Unfortunately, the `File` class doesn't have a method that returns the file extension, but you can get the name with the `getName` method and then use the `matches` method with a regular expression to determine whether the file is of the type you're looking for. Here's an `if` statement that determines whether the filename in the `name` variable is a `.java` file:

```
if (name.matches(".*\\.java\\z"))
```

Here the regular expression matches strings that begin with any sequence of characters and end with `.java`. (For more information about regular expressions, refer to Book 5, Chapter 2.)

Here's a `FileFilter` class that displays files with the extension `.java`:

```
private class javaFilter
    extends javax.swing.filechooser.FileFilter
{
    public boolean accept(File f)
    {
        if (f.isDirectory())
           return true;
        String name = f.getName();
        if (name.matches(".*\\.java\\z"))
           return true;
        else
           return false;
    }
    public String getDescription()
    {
        return "Java files (*.java)";
    }
}
```

After you create a class that implements a file filter, you can add the file filter to the file chooser by calling the `addChoosableFileFilter` method, passing a new instance of the `FileFilter` class, like this:

```
fc.setChoosableFileFilter(new JavaFilter());
```

If you want, you can remove the All Files filter by calling the method `setAcceptAllFileFilterUsed`, like this:

```
fc.setAcceptAllFileFilterUsed(false);
```

Then only the file filters that you add to the file chooser appear in the Files of Type drop-down list.

Using Path, Paths, and Files

The original Java `File` class has been around since the beginning. With Java 1.7, several newer classes were introduced to address some of the weaknesses of the original `File` class — most notably, that the `File` class doesn't provide detailed error information when problems occur. For example, the `delete` method of the `File` class returns a boolean value to indicate whether the file was deleted, but if the file could not be deleted, there's no way to find out why.

Note that `Path` is not an interface, not a class. That means you can't directly create a `Path` object by calling a constructor. Instead, you use various static methods of two classes — `Paths` and `Files` — to create `Path` objects. The naming is confusing — `Path` versus `Paths`, `File` versus `Files` — but after working with the `Paths` class to create `Path` objects, you get used to the distinction.

`Path`, `Paths`, and `Files` are in the `java.nio.file` package, so any program that uses them should import `java.nio.file.*`.

Creating a Path

Because it's an interface and not a class, `Path` has no constructor. So, the only way to create a `Path` is to get one via the return value of a method that returns a `Path`. There are plenty to choose from — many of the methods of the `Files` class return a `Path`). However, if all you want to do is create a `Path` from a String that represents the `Path`, you have two choices:

- Use the static `get` method of the `Paths` class, like this:

```
Path p = Paths.get("c:\\data\\movies.txt");
```

- Use the static `of` method from the `Path` class, like this:

```
Path p = Path.of("c:\\data\\movies.txt");
```

Either way, the result is the same: a `Path` object is created from the path `c:\data\movies.txt`.

`Paths.get` and `Path.of` throw `InvalidPathException`, so you should enclose them in a `try...catch` block to handle the exception if it's thrown.

Using the Files class

With a `Path` in hand, you can use a plethora of methods from the `Files` class to manipulate files and directories. A useful assortment of those methods are listed in Table 1-4.

TABLE 1-4 Files Methods

Method	Description
`static long copy(Path source, Path target)`	Copies the specified source file to the specified target file. Returns the number of bytes written.
`static Path moveTo(Path target)`	Moves the file to the target path and deletes the original file.
`Path createDirectory(Path dir)`	Creates the directory on the hard drive if it doesn't already exist.
`Path createFile(Path path)`	Creates the file on the hard drive if it doesn't already exist.
`void delete(Path path)`	Deletes the file or directory. The method throws an exception if the file or directory doesn't exist or couldn't be deleted.
`void deleteIfExists(Path path)`	Deletes the file or directory if it exists. The method doesn't throw an exception if the file or directory doesn't exist.
`boolean exists()`	Returns `true` if the file exists on the hard drive or `false` if the file doesn't exist on the hard drive.
`boolean notExists()`	Returns `true` if the file doesn't exist on the hard drive or `false` if the file does exist on the hard drive.
`DirectoryStream newDirectoryStream(Path dir)`	Gets a `DirectoryStream` object that you can use to read the contents of the directory.
`DirectoryStream newDirectoryStream(Path dir, String filter)`	Gets a `DirectoryStream` object that's filtered by the filter string, which can contain wildcards such as `*.txt` to retrieve just `.txt` files.
`String toString()`	Returns the pathname for this file or directory as a string.

A `Path` object represents a file that may or may not actually exist on the hard drive. You can test to see whether a path exists by using the `Files.exists` or `Files.notExists` method, like this:

```
Path p = Paths.get(path);
if (Files.notExists(p))
    System.out.println
        ("The input file does not exist!");
```

To create a new file, use the `createFile` method, like this:

```
Path p = Paths.get("c:\\data\\test.txt");
try
{
    Files.createFile(p);
    System.out.println ("File created!");
}
catch (Exception e)
{
    System.out.println ("Error: " + e.getMessage());
}
```

Note that the `createFile` method throws an exception if the file couldn't be created. The `getMessage` method of this exception returns a message that explains why the file couldn't be created.

Getting the contents of a directory

One of the weaknesses of the `File` class is that it doesn't deal well with large directories. In particular, methods such as `listFiles` that allow you to access the contents of a directory return an array of `File` objects. That's fine if the directory contains a few dozen or even a few hundred files, but what if the directory contains thousands or tens of thousands of files? In short, the `File` class is not scalable.

The `Path` class remedies this deficiency by letting you access the contents of a directory via a stream object defined by `DirectoryStream`. I say more about working with streams in Chapter 2 of the bonus content, but for now, suffice it to say that a stream provides a simple way to access a large number of data items

one at a time. You can retrieve the items in a directory stream easily by using an enhanced `for` statement, as in this example:

```
Path c = Paths.get("C:\\");
try
{
    DirectoryStream<Path> stream = Files.newDirectoryStream(c);
    for (Path entry: stream)
        System.out.println(entry.toString());
}
catch (Exception e)
{
    System.out.println("Error: " + e.getMessage());
}
```

This example displays a listing of the contents of the `C:\` directory, much as typing **DIR C:** at a command prompt would.

You could change the preceding example to list just the text files (files with the extension `.txt`) by changing the first statement after the `try` statement to this:

```
DirectoryStream<Path> stream = Files.newDirectoryStream(c, "*.txt");
```

Using a File Visitor to Walk a File Tree

In the section "Getting the contents of a directory," earlier in this chapter, I mention that processing subdirectories of a main directory by using the `File` class requires fancy recursive programming. Fortunately, you can create a special class called a *file visitor* to handle the tricky part of the recursive programming. It does this by *walking* the file tree — visiting every file in the tree and calling one or more methods defined by your file visitor. Here are the basic details of how this magic works:

1. **Create a file-visitor class, which is extended from `FileVisitor` or, more often, `SimpleFileVisitor`.**

   The `SimpleFileVisitor` class defines several methods that are called for every file in the file tree, as shown in Table 1-5.

2. **In the file visitor, override one or more methods that are defined by the `SimpleFileVisitor` class.**

These methods are where you write the code that you want to execute for every file visited when the directory tree is walked. You always want to override at least three of the methods listed in Table 1-5:

- `visitFile`, which is called for every file in the file tree
- `visitFileFailed`, which is called for every file that can't be accessed (usually due to permissions issues)
- `preVisitDirectoryFailed`, which is called for every directory that couldn't be accessed

3. **Create a `Path` object that indicates the starting point (that is, the root) of the file tree you want to walk.**

   If you want to visit all the files on your `C:` drive, for example, this path should point to `C:\`.

4. **Call the `walkFileTree` method of the static `Files` class to process the files.**

   This method takes just two arguments: the `Path` object (which identifies the root of your file tree) and an instance of your file-visitor class.

TABLE 1-5 The SimpleFileVisitor Class

Method	Description
`FileVisitResult visitFile(T file, BasicFileAttributes attr)`	Called once for every file in the file tree.
`FileVisitResult visitFileFailed(T file, IOException e)`	Called if the file couldn't be accessed.
`FileVisitResult preVisitDirectory(T dir)`	Called once for every directory in the file tree. This method is called before any of the files in the directory are visited.
`FileVisitResult postVisitDirectory(T dir)`	Called once for every directory in the file tree. This method is called after all the files in the directory are visited.

Confusing enough? An example should clear things up. Listing 1-1 shows just about the simplest example of a file visitor I can come up with. This program lists all the files in `C:\Windows\System32`, including all its subfolders.

LISTING 1-1: Simple File Visitor

```
public class FileVisitorDemo
{
    public static void main(String[] args)
    {
        Path start = Paths.get("c:\\Windows\\System32");                    →5
        MyFileVisitor visitor = new MyFileVisitor();                        →6
        try
        {
            Files.walkFileTree(start, visitor);                             →9
        }
        catch (Exception e)
        {
            System.out.println(e.getMessage());
        }
    }

    private static class MyFileVisitor extends SimpleFileVisitor <Path>     →17
    {
        @Override
        public FileVisitResult visitFile(Path file,                         →20
            BasicFileAttributes attr)
        {
            System.out.println(file.toString());
            return FileVisitResult.CONTINUE;                                →24
        }

        @Override
        public FileVisitResult visitFileFailed(Path file, IOException e)    →28
        {
            System.out.println(file.toString() + " COULD NOT ACCESS!");
            return FileVisitResult.CONTINUE;
        }

    }
}
```

Here are the highlights of how this program works:

- →5 Creates an instance of the `Path` class that starts the file tree at `C:\Windows\System32`. You could substitute any directory you want for this path.
- →6 Creates an instance of the `MyFileVisitor` class, which is defined later in the program (at line 21).
- →9 Walks the file tree, starting at the directory indicated by `start`, using the `MyFileVisitor` object created in line 10.

- →17 Defines the `MyFileVisitor` class, which extends the `SimpleFileVisitor` class. `SimpleFileVisitor` is a generic class, so you must specify a type. Usually you specify the `Path` type so that `SimpleFileVisitor` processes `Path` objects.
- →20 Overrides the `visitFile` method, which is called once for each file that is accessed as the file tree is walked. This method simply prints the name of the file to the console. In a more realistic program, you would perform some more significant action on the file, such as copying it or opening the file and reading its contents.
- →24 Produces the return value of the `visitFile` method, which is of type `FileVisitResult`. The most commonly returned value is `CONTINUE`, which indicates that the file-tree walker should continue processing files. Other options include `TERMINATE`, `SKIP_SIBLINGS`, and `SKIP_SUBTREE`.
- →28 Overrides the `visitFileFailed` method, which is called whenever a file can't be accessed. In this program, the `visitFileFailed` method simply prints an error message.