```
ThreadAccount
ThreadDemo
UnsafeAccount
UnsafeBlock
UnsafePointer
```

Files and Streams

Programming languages have undergone an evolution in how they deal with the important topic of input/output (I/O). Early languages, such as FORTRAN, COBOL, and the original BASIC, had I/O statements built into the language. Later languages have tended not to have I/O built into the language, but instead rely on a standard library for performing I/O, such as the **<stdio.h>** library in C. The library in languages like C works directly with files.

Still later languages, such as C++ and Java, introduced a further abstraction called a *stream*. A stream serves as an intermediary between the program and the file. Read and write operations are done to the stream, which is tied to a file. This architecture is very flexible, because the same kind of read and write operations can apply not only to a file, but to other kinds of I/O, such as network sockets. This added flexibility introduces a slight additional complexity in writing programs, because you have to deal not only with files but also with streams, and there exists a considerable variety of stream classes. But the added complexity is well worth the effort, and C# strikes a nice balance, with classes that make performing common operations quite simple.

As with directories, the **System.IO** namespace contains two classes for working with files. The **File** class has all static methods, and the **FileInfo** class has instance methods. The program **FileDemo** extends the **DirectoryDemo** example program to illustrate reading and writing text files. We will illustrate binary file I/O later in this chapter, when we discuss serialization. The directory commands are retained so that you can easily exercise the program on different directories. The two new commands are "read" and "write." The "read" command illustrates using the **File** class. The "dir" command, already present in the **DirectoryDemo** program, illustrates using the **FileInfo** class.

Here is the code for the "read" command. The user is prompted for a file name. The static **OpenText** method returns a **StreamReader** object, which is used for the actual reading. There is a **ReadLine** method for reading a line of text, similar to the **ReadLine** method of the **Console** class. A **null** reference is returned by **ReadLine** when at end of file. Our program simply displays the contents of the file at the console. When done, we close the **StreamReader**.

```
...
else if (cmd.Equals("read"))
{
   string fileName = iw.getString("file name: ");
   StreamReader reader = File.OpenText(fileName);
   string str;
   str = reader.ReadLine();
   while (str != null)
   {
      Console.WriteLine(str);
      str = reader.ReadLine();
   }
   reader.Close();
}
...
```

Here is the code for the "write" command. Again we prompt for a file name. This time we also prompt for whether or not to append to the file. There is a special constructor for the **StreamWriter** class that will directly return a **StreamWriter** without first getting a file object. The first parameter is the name of the file, and the second a **bool** flag specifying the append mode. If **true**, the writes will append to the end of an already existing file. If **false**, the writes will overwrite an existing file. In both cases, if a file of the specified name does not exist, a new file will be created.

```
...
else if (cmd.Equals("write"))
{
   fileName = iw.getString("file name: ");
   string strAppend = iw.getString("append (yes/no): ");
   bool append = (strAppend == "yes" ? true : false);
   StreamWriter writer = new StreamWriter(fileName, append);
   Console.WriteLine("Enter text, blank line to terminate");
   string str = iw.getString(">>");
   while (str != "")
   {
      writer.WriteLine(str);
      str = iw.getString(">>");
   }
   writer.Close();
}
...
```

Here is a sample run of the program. We first obtain a listing of existing files in the current directory. We then create a new text file, **one.txt**, and enter a couple of lines of text data. We again do "dir", and our new file shows up. We try out the "read" command. You could also open up the file in a text editor to verify that it has been created and has the desired data. Next we write out another line of text to this same file, this time saying "yes"