Contents

Chapter 18 – Data Files and Directories

Introduction to Computer Science with Java

Chapter 18 – Data Files and Directories

This chapter discusses basic data file I/O in Java including the use of I/O streams and buffers for reading and writing text data files and binary data files. The importance of buffering and exception handling is discussed, along with techniques for doing so.

Chapter Learning Outcomes

Upon completion of this chapter students should be able to:

- describe the difference between text data files and binary data files and the advantages and disadvantages of using each.
- describe the nature of path names, and the use of the dot notation and slash and backslash in file path names.
- describe how to use Java's *File* class to identify files and directories, get information about files and directories, and manipulate files and directories.
- describe I/O streams in Java, the concept of buffering, and the importance of buffering data stream I/O.
- describe the use of the *Scanner* and *PrintWriter* classes in Java for text data file I/O and create Java code that reads and writes text data files using them.
- describe the use of the *FileInputStream* and *FileOutputStream* classes for binary data file I/O in Java and create software that uses the classes for binary data file I/O.
- describe how to buffer binary I/O streams using the *BufferedInputStream* and *BufferedOutputStream* classes and write software that uses the classes to implement buffering.
- describe the importance of exception handling as opposed to throwing exceptions in Java data file programming, and create code that uses try-catch-finally blocks to handle I/O exceptions.

18.1 Java Data Files

Java applications typically read and write data files either as text files or binary files.

A **text file** is a data file in which the bits of data are formatted and stored using a character set to represent text data. Java writes text files using the UTF-16 version of the Unicode character set. A person can read the

data in the file using a text editor or any other program that can read and display text data. Text data files are often used as a common format to move data from one application to another, and for files that people will most likely need to read. Java source code files (*.java*) and *HTML* Web page files (*.htm and .html*) are stored as text files. Most other programming languages also store source code files as text files.

A **binary file** is a data file containing raw binary data, not formatted as character data. The format and meaning of the data depends on the software that reads or writes the data file. It usually cannot be read by a text editor. Java *class* files, Java *JAR* files, and object code files for most other programming languages are stored as raw binary files.

The advantage in using text files is that people can read them using a text editor or similar software. The disadvantage in using text files is that the data needs to be converted between the internal data formats of the software using the data and the text format of the file. For example, integer and floating point data needs to be converted to text formats when the data is stored as text data. Binary data files do not need to be converted, so they are more efficient – both in terms of the time it takes to read and write the data and the space taken up by the files. They exhibit better temporal efficiency (processing time) and spatial efficiency (use of storage space).

Software written using various Java APIs often has the ability to work with binary data in specialized formats, such as the WAV, AIFF or AU formats for files written or read by implementations of the Java Sound API representing audio data.

In the remainder of this chapter we will learn about using Java to read and write data files as text files and binary files, but will not discuss specialized formats used by specialized APIs. The APIs will have methods for dealing with any specialized formats they use.

18.2 File Objects

The details of data file operations on each system are platform dependent. Recall that a **computing platform** is a combination of the operating system and the hardware of a computer system. A particular computing platform provides the local environment in which Java software operates, and is sometimes called the *local system* or the *local host*. (In TCP/IP networking the one-word term *localhost* refers to the current local computer.)

Java is a cross-platform programming language, so software written in Java should be as platform *independent* as possible.

The *File* class in Java provides an abstract object that is used for a platform independent implementation of platform dependent functions – managing files and directories on each platform on which Java runs. The particular JVM on the local machine handles the conversion from the abstract *File* class in Java software to the actual file system on the local platform.

The *File* class, and most of the classes used later in this chapter to read and write data files, are in the package *java.io*, so we should use the import statement `import java.io.*;` to access them.

We have seen and used the File class in earlier programming, but only to identify a file. Here we review the use of File class objects to identify files and see what some additional methods in the class can do.

The term **file** refers to an entry in a file system's tree of directories, which could be a data file or a subdirectory of the current directory. This means an instance of the *File* class could refer to a file or a directory.

File System Terminology and Notations

The term *file* in a file system is often used to generally refer to an entry in a file directory, which could be a data file or a sub-directory. The terms *folder* is often used in place of *directory* in a GUI-based operating system.

An *absolute path name* refers to a specific unique entry in the local file system. A *canonical path name* is an absolute path name that does not use any "shortcut" notation, such as dot notation or wildcards. A *relative path name* locates a file relative to another location, usually relative to the current location in the local system's tree of directories and files. Relative path names often use dot notation.

Dot Notation in File Path Names

```
"./datafiles/example.txt"        "../datafiles/example.txt"
```

Most operating systems recognize do notation to refer to current and parent directories. The single dot refers to the current directory, and the double dot refers to the current directory's parent directory.

The Slash vs the Backslash in Windows File Path Names

```
"c:/datafiles/example.txt"
```

Notice that a full context file path names in Java should uses the slash (/) as a separator between elements of the file's path. Originally, Microsoft Windows used the backslash (\) as a separator in file paths, but the POSIX operating system standard calls for the slash to be used. Windows now allows for the use of a slash or a backslash. The UNIX operating system and its derivatives, including Linux, Android, and Apple's OS X and iOS operating systems, all use the slash. In the interest of cross platform compatibility and coding simplicity, the slash should be used in Java source code Strings identifying file paths. Each system's JVM will interpret this correctly for the local environment.

For more about file and directory terminology, see:

http://www.webopedia.com/TERM/D/directory.html

To access a data file (or directory) using the File class, an instance of the class needs to be created and linked to a specific file:

```
File infile = new File("example.txt");
```

This code creates an instance of the File class named *infile* and associates it with the file *example.txt* in the current directory (also called the *working* directory). This only works if the file is in the current directory or a directory identified by a system's *path* variable (*$PATH* on Unix-like systems, *%PATH%* on Microsoft systems).

We could also use a *canonical* absolute file location:

```
File infile = new File("c:/datafiles/example.txt");
```

```
Or, we could use a relative path name using dot notation:
```

```
File infile = new File("./datafiles/example.txt");
```

In this example, the single dot refers to the current directory, so *example.txt* would be in the folder *datatfiles*, a sub-directory of the current directory.

```
File infile = new File("../datafiles/example.txt");
```

In this last example, the double dot refers to the current directory's parent directory, so *example.txt* would be in the folder *datatfiles*, a sub-directory of the current directory's parent, parallel to the current directory.

It is generally better to use a relative path name than an absolute path name, unless the software is being created for a very specific application on a specific system.

The File class in the *java.io* package actually has more than 50 methods that can be used to identify and manipulate files and directories on a local computer system. The table below summarizes some of the more commonly used methods. More detailed information about the File class is available online at: http://docs.oracle.com/javase/7/docs/api/java/io/File.html

java.io.File	
Name and Return Type	**Description**
canRead():boolean	Tests whether the application can read the file denoted by this abstract pathname.
canWrite():boolean	Tests whether the application can modify the file denoted by this abstract pathname.
compareTo(**File** pathname):int	Compares two abstract pathnames lexicographically.
createNewFile():boolean	Atomically creates a new, empty file named by this abstract pathname if and only if a file with this name does not yet exist.
delete():boolean	Deletes the file or directory denoted by this abstract pathname.
deleteOnExit()	Requests that the file or directory denoted by this abstract pathname be deleted when the virtual machine terminates.
exists():boolean	Tests whether the file or directory denoted by this abstract pathname exists.
getAbsolutePath():String	Returns the absolute pathname string of this abstract pathname.
getCanonicalPath(): String	Returns the canonical pathname string of this abstract pathname.
getName(): String	Returns the name of the file or directory denoted by this abstract pathname.
getParent():String	Returns the pathname string of this abstract pathname's parent, or null if this pathname does not name a parent directory.
getParentFile():File	Returns the abstract pathname of this abstract pathname's parent, or null if this pathname does not name a parent directory.
isDirectory():boolean	Tests whether the file denoted by this abstract pathname is a directory.
isFile():boolean	Tests whether the file denoted by this abstract pathname is a normal file.
length():long	Returns the length in bytes of the file denoted by this abstract pathname.
list():String	Returns an array of strings naming the files and directories in the directory denoted by this abstract pathname.
mkdir():boolean	Creates the directory named by this abstract pathname.
renameTo(**File** dest) :boolean	Renames the file denoted by this abstract pathname.
setReadOnly():boolean	Marks the file or directory named by this abstract pathname so that only read operations are allowed.
toString():String	Returns the pathname string of this abstract pathname.

Several of the methods in the File class return boolean values used to test the properties of the file or directory associated with an instance of the class. ***exists()*** returns a true value if the file or directory actually exists, while ***isFile()*** and ***isDirectory()*** and will tell us if the item is a file or a directory. The ***canRead()*** and ***canWrite()*** method will return a true value if we have read access or write access to a specific file or directory.

Some of the methods that manipulate files and directories will return a boolean true value if the method succeeds in carrying out its task and false value if it does not. These include ***createNewFile()***,and ***mkdir()*** methiods used to create a new file or make a new directory; the ***delete()*** method that deletes a file or directory the ***setReadOnly()*** method that can make a file or directory – including on a USB drive – read only; and the ***renameTo()*** method that can be used to change the name of a file.

Only the ***deleteOnExit()*** method does not return a value. It deletes a file or directory when Java software stops running and the associated instance of the Java Virtual Machine terminates.

Several methods return information about a file or directory as a String, including the traditional ***toString()*** method, and the ***getName()***, ***getAbsolutePath()***, ***getCanonicalPath()***, and ***getParent()*** methods. The ***list()*** method returns as an array of Strings the list of files and subdirectories in a particular directory. Each of these has a parallel method that will return a File class object such as ***getParentFile()*** that returns the parent of a file or directory as a File class object.

The File class implements the Comparable interface, with a ***compareTo()*** method returning an integer based on the lexicographical order of the file or path name. On Unix and Unix-like systems, this method is case sensitive because the underlying operating system is case sensitive; on Microsoft Windows systems it is not case sensitive because Microsoft Windows filenames are not case sensitive.

The ***length()*** method also returns an integer – a long integer value that is the size of the file in bytes. It returns a zero if the file does not exist and an unspecified value if the object is a directory.

The following three examples show how to use some of these methods.

Programing Example – Getting File Information

The java application ***FileDataDemo.java*** is shown below and included with the files for this chapter as a NetBeans project. It returns information about a specific file. The program's target file is identified in the code as "*csciLogo.jpg*", a JPEG file for the Community College of Philadelphia logo.

As written, the file must be in the directory for the NetBeans project or in the *dist* folder with the JAR file for the distributed application.

You can rewrite the code to access any file or directory in the local computer's file system. The code checks to see if the target item is a file or a directory. It also returns a message if the target exists but is neither a file nor a directory, which is unlikely, but it could happen on some systems.

It returns the location (absolute path) and file size if the target is a file, returns the location (absolute path) if the target is a directory, and returns a message if the target exists but is neither a file nor directory, or if the target does not exist. Other methods in the File class could be used to return additional information about the target file or directory, or to manipulate it.

```
/* FileDataDemo.java
 * CSCI 112 - Spring 2014
 *
 * The software in this example returns information about a specific file.
 * The file in the is identified in line 25 in the code as  csciLogo.jpg, a copy
 * of the Community College of Philadelphia Computer Science logo.
 *
 * For the code to work as written, the file must be in the the netbeans project folder or
 * in the dist folder with the JAR file for the code.
 *
 * Line 25 can be re-written to identify another file, perhaps using an absolute path.
 * last editied 2/23/2014 by C. Herbert
 */
package filedatademo;
import java.io.*;
import java.util.Scanner;

public class FileDataDemo {

    public static void main(String[] args) {

        // identify the file. An alternative would be to ask the user for the filename.
        String fileName = "csciLogo.jpg";

        // Create a File class object linked to a specific file
        File target = new File(fileName);

        // echo target name and skip a line
        System.out.println(" You requested information about "+ fileName + ".\n");

        // if the identified file exists
        if ( target.exists() ) {

            System.out.println(fileName + " exists.");

            // if the target is a data file
            if ( target.isFile() ) {

                // print data about the file -- the absolute path and length
                System.out.println("The file is: " +target.getAbsolutePath());
                System.out.printf("The file size is %,d bytes",target.length() );
            } // end if ( target.isFile() )
            else if ( target.isDirectory() ) {
                    System.out.println("You have indentified a directory at:");
                    System.out.println(target.getAbsolutePath());
            } // end if ( target.isDirectory() )
            else
                System.out.println("The item exists, but is neither a file nor a directory.");

        } // end if ( target.exists() )
        else
            System.out.println("The item you asked about - " + target + " - does not exist.");

        // got to the next line at the end of the printout
        System.out.println();
    } // main()
} // end class FileDataDemo
```

Here is an example of the output from the program:

```
Output - FileDataDemo (run)
run:
 You requested information about csciLogo.jpg.

csciLogo.jpg exists.
The file is: C:\Users\cherbert\Desktop\NetBeansProjects\FileDataDemo\csciLogo.jpg
The file size is 34,414 bytes
BUILD SUCCESSFUL (total time: 0 seconds)
```

Programing Example – Listing Files in a Directory

The java application ***DirectoryListDemo.java*** is shown below and included with the files for this chapter as a NetBeans project.

the ***list()*** method in the File class returns a String array with of the names of the entries in a directory, if the instance of File class with which it is used is a directory. A for loop can be used to print out the elements in the array. On the right is the directory for the NetBeans project folder ***DirectoryListDemo, and on the left*** along a printout of the names of items in the folder created using the *list()* method.

Name	Date modified	Type	Size
build	2/23/2014 7:42 PM	File folder	
nbproject	2/23/2014 6:01 PM	File folder	
src	2/23/2014 6:01 PM	File folder	
build.xml	2/23/2014 6:01 PM	XML Document	4 KB
csciLogo.JPG	1/12/2014 11:00 AM	JPEG image	34 KB
manifest.mf	2/23/2014 6:01 PM	MF File	1 KB

```
build
build.xml
csciLogo.JPG
manifest.mf
nbproject
src
```

The *list()* method is useful, but the array it creates only contains the names of the entries in a directory. It is difficult to know which entries are for files and which are for sub-directories.

The *listFile()* method returns a similar array with one element for each of the entries in a directory, but as File class objects rather than as Strings. This will allow us to use the other methods in the File class to get additional information about each entry in the directory.

DirectoryListDemo.java displays the size of each file and adds the label *<DIR>* to identify each directory. The code is shown and discussed below. The output from *DirectoryListDemo.java* is shown on the right. In this case, it displays the contents of the root directory of the "c:" disk drive from a Microsoft Windows 7 system.

```
utput - DirectoryListDemo (run)
run:
 You requested information about c:/.

c:/ exists.
Directory of c:\
$RECYCLE.BIN                    <DIR>
APPS                            <DIR>
CARDS                           <DIR>
CSCI.JPG                        22,083
DATAFILES                       <DIR>
DELL                            <DIR>
DELL.SDR                        39,712
DOCUMENTS AND SETTINGS          <DIR>
DRIVERS                         <DIR>
HIBERFIL.SYS                    6,361,395,200
INTEL                           <DIR>
ITS ONLY                        <DIR>
MONOPOLY                        <DIR>
MSOCACHE                        <DIR>
OUT.TXT                         147
PAGEFILE.SYS                    8,481,861,632
PENGUINS.JPG                    777,835
PERFLOGS                        <DIR>
PROGRAM FILES                   <DIR>
PROGRAM FILES (X86)             <DIR>
PROGRAMDATA                     <DIR>
QUARANTINE                      <DIR>
SYSTEM VOLUME INFORMATION       <DIR>
USERS                           <DIR>
WINDOWS                         <DIR>
```

```
/* DirectoryListDemo.java
 * CSCI 112 - Spring 2014
 *
 * The software in this example creates and displays an array with information
 * about entires in a directory.
 *
 * The directory is identified in line 29 of this code as  "c:\", the root directory
 * on most  Microsoft  Windows systems.  changing this to "./" will return
 * information on the current working directory, most likley the folder holding
 * the programsm itself.
 *
 * Line 29 can be changed to identify another directory.  The program can be rewritten
 * to allow the user to input a directory's name.
 *
 * last editied 2/23/2014 by C. Herbert
 */
package directorylistdemo;

import java.io.*;
import java.util.Scanner;

public class DirectoryListDemo {

    public static void main(String[] args) {

        // Identify the directory to be listed
        // This examples uses the root directory on the C: disk drive
        // You can change this a different directory
        String directoryName = "c:/";

        File[] entries;      // an array for the entries in the target directory

        // Create a File class object linked to the target
        File target = new File(directoryName);

        // echo target name and skip a line
        System.out.println(" You requested information about " + directoryName + ".\n");

        // if the identified item exists
        if (target.exists()) {

            // echo target and show absolute path
            System.out.println(directoryName + " exists.");
            System.out.println("Directory of " + target.getAbsolutePath());

            // if the target is a directory
            if (target.isDirectory()) {

                // get the data and load the array
                entries = target.listFiles();

                // iterate the array using an alternate for statement
                for (File entry : entries) {

                    // print the file or directory name
                    System.out.printf("%-32S", entry.getName());

                    // print the file size, or label if it is a directory
                    // left justified, allowing 32 spaces for the name
                    if (entry.isFile())
                        System.out.printf("%,d \n", entry.length());
                    else
                        System.out.println("<DIR>");

                } // end for
            } // end if ( target.isDirectory() )
```

```
            else if (target.isFile()) {
                System.out.println("You have indentified a file at:");
                System.out.println(target.getAbsolutePath());
            } // end else

        } // end if ( target.exists() )

        else {
            System.out.println( target + " - does not exist.");
        }
        // add a next line at the end of the printout
        System.out.println();

    } // end main()
} // end class DirectoryListDemo
```

The code above sets the String variable *directoryName* to "c:/", the root directory in most Windows-based systems. It then creates a File class object name *target*, associated with this directory.

If the target is really a directory, the code uses the File class method *listFiles(),* to create an array of File class objects, with one element for each entry in the target directory. If the target is a file or if the target does not exist, then an appropriate message informs the user.

An *enhanced for loop* is used to iterate the array. The name of each entry is printed, allowing 32 spaces for the name. If an element in the array is a file, then the file size is shown. If an element in the array is a directory, then the message *<DIR>* is shown.

The program can easily be rewritten to ask the user to enter the name of the target directory. If you run the program, notice that it shows all entries in a directory, even hidden entries and systems files that are not normally shown to the user.

The Enhanced *for* Statement

Line 54 in *DirectoryListDemo.java* uses a special version of the *for* statement, called the *enhanced for statement*. The *enhanced for statement* iterates an entire array.

The *enhanced for statement* has a loop header with the keyword *for*, followed by parentheses that contain a declaration of a variable used to refer to each item in the array and the name of the array. The statement or block of code following the loop header will be executed once for each element in the array.

```
for ( variable declaration : name of an array)
   block of code
```

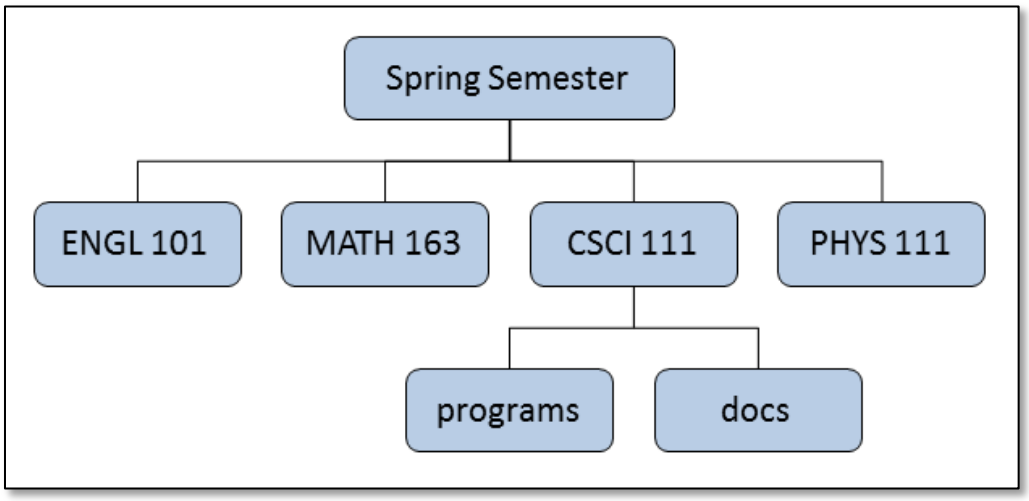
The syntax within an enhanced for loop can be tricky, so be careful. We do not use the name of the array within the loop, only the variable declared in the header. This variable refers to an element of the array, sequentially incremented each time through the loop. See *Chapter 6, page 11*.

Prior to *Java 7*, this only worked using an integer variable. It now works with any data type.

Programing Example – Creating a Set of Directories for an Application

The NetBeans project folder ***CreateDirectoriesDemo*** is included with the files for this. It demonstrates the use of java's File class to create a set of directories. One of the exercises at the end of the chapter deals with copying a directory and all of its subfolders to a new location.

The code in this example creates a set of folders to help a student keep track of homework assignments. The set includes a parent folder for the Spring Semester, and separate folders for each class – *English 101*, *Math 163*, *Art History 105*, and *Computer Science 111*. The *Computer Science 111* folder contains sub-folders for *programs* and *docs* (for course documents*)*.



The code is shown and discussed below.

```
/* CreateDirectoriesDemo.java
 * CSCI 112 - Spring 2014
 *
 * The software in this example creates a set of folders (directories).
 * It is intended as an example for CSCI 112.
 *
 * The directory which will contain the new set of directories is identified in
 * line 23 of this code as  "c:\", the root directory on most Microsoft Windows
 * systems.  This can be changed to place the new set of directories elsewhere.
 *
 * last editied 2/23/2014 by C. Herbert
 */
package createdirectoriesdemo;
import java.io.*;
import java.util.Scanner;

public class CreateDirectoriesDemo {

    public static void main(String[] args) {

        // Establish the location of the parent for the new set of directories.
        // This could be changed to user input.
        String location = "c:/";

        // create a String array of the directories to be created
```

```
        String[] folderPaths  = {
            "/Spring Semester",
            "/Spring Semester/ENGL 101",
            "/Spring Semester/CSCI 111",
            "/Spring Semester/MATH 163",
            "/Spring Semester//PHYS 111",
            "/Spring Semester/CSCI 111/programs",
            "/Spring Semester/CSCI 111/docs"
        };

        // create a File class array for directories to be created
        File[] newFolders = new File[folderPaths.length];

        // create new directories based on the file names in the array
        for (int i = 0 ; i < newFolders.length; i++) {

            // create a File object for this new directory
            // based on the parent location and each new path name
             newFolders[i]  = new File( location + folderPaths[i] ) ;

            // make the new directory
            newFolders[i].mkdir();

        } // end for

    } // end main()
} // end class CreateDirectoriesDemo
```

The code first establishes a String array named *folderPaths* with the names of the new folders to be created, then creates an empty array of abstract File class objects named *newFolders* for the new folders (directories).

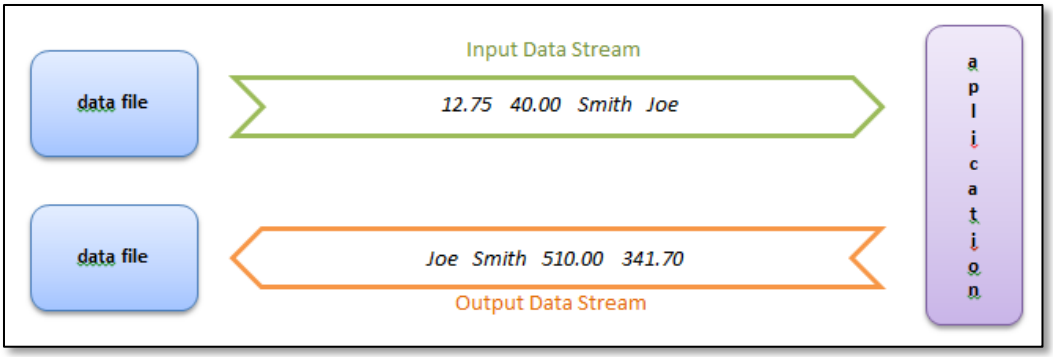
A standard for loop iterates the *newFolders* array, populating the *newFolders* array using the names of the new directories appended to the variable *location*, which was established back at the beginning of the code as the parent location for the entire set of directories. The *mkdir()* method is used to create an actual directory from the abstract File object.

Nothing dramatic will happen when you run the code, but the set of new directories will be created on your system under the root directory of the C: disk drive. Check the system to see if it worked. If access to *c:*\ is denied, then change the value of the location variable to refer to a different location, such as in the documents directory or on a USB flash ROM drive.

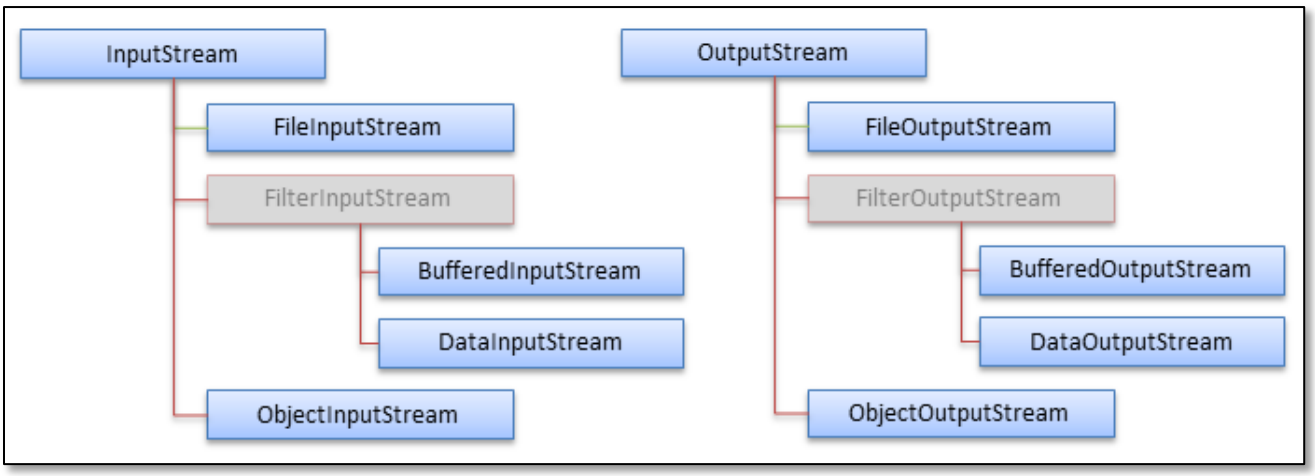
The File class can create and manipulate abstract File objects associated with actual files and directories, but it cannot access the data in a file. To do that, we need to establish input and output streams and connect them to a File object. I/O streams are the primary topic for the remainder of the chapter.

18.3 I/O Streams and Buffers in Java

Java software communicates with files and other data sources and destinations using I/O streams. In CSCI 111, I/O streams were used with the *Scanner* class to read from the keyboard in Chapter 2, then used for simple text file access in Chapter 6, using *The Scanner* and *PrintWiter* classes.



There are several classes in the ***java.io*** package that establish different types of I/O streams for different purposes. The following diagram shows just a few of these:



This is only part of the picture: there are actually dozens of different I/O stream classes in Java. Why so many? Java is a cross-platform programming language used for many different purposes with many different types of data on many different computing systems. The various I/O stream classes exist to be used most efficiently for specific applications.

The *InputStream* class and the *OutputStream* class are the parents of all other I/O stream classes. They read in binary data in a very simple form. The subclasses of *InputStream* and *OutputStream* handle I/O streams in specialized ways. There are also other classes, such as *FileReader* and *FileWriter* that can work with *InputStreams* and *OutPutStreams* (or, in some cases, of their sub-classes) in more specific ways.

During the rest of the semester we will see the five input stream classes and the corresponding five output stream classes highlighted in blue on the above diagram. In this chapter we will focus on:

- text files;
- basic binary data files;
- buffering binary data files.

We will also look at buffered input and output, which can be established by the programmer, or by using the *BufferedInputStream* and *BufferedInputStream* classes for binary files.

First, we will consider the concept of data buffering and its importance.

Data Buffering

When data moves between an application and a file, it is usually really moving between internal memory and an external storage device or some other external I/O device. *External*, in this sense, means that it is not part of the main computer system – usually the chip set containing the CPU and internal memory.

This can be very significant in terms of the efficiency of a computer system. Internal memory is almost always electromagnetic and close to the CPU. *Electromagnetic* means that only electric and magnetic signals move around – there are no moving parts. External devices are often electromechanical. *Electromechanical* devices have moving parts, which means they are much slower than devices that are only electromagnetic – a disk drive , for example, needs to spin and get up to speed and a disk head needs to move to the correct track to write on a disk. This does not happen with internal memory.

Even if the external device – such as a USB flash ROM drive – has no moving parts, it is much farther away from the CPU than the internal memory is and uses slower, less expensive technology.

This all means that using an external storage device is generally about 10,000 times slower than using internal memory. For small amounts of data we probably won't notice the difference because we are talking about hundredths or thousandths of a second instead of millionths or billionths of a second, but for large amounts of data we can actually see the difference.

Consider the following algorithm, which writes an array of 10,000 elements, each with one byte, to a disk drive.

```
byte[]  list = new byte[10000];

open a file on the disk drive;

for i = 1 to 10,000
   write list[i] to a file on the disk drive;

close the file on the disk drive;
```

Unless we specifically tell the computer hardware to do something different, this algorithm will turn on the disk drive, write one byte to the disk drive, then turn off the disk drive. It will do this again and again and again until all 10,000 bytes are written to the disk. If you were near the computer when this happened you would see and hear the disk drive repeatedly turning itself on and off very quickly. This is usually not a good idea. The vibrations from this could potentially damage the disk drive or other parts of the computer system. Imagine repeatedly turning anything mechanical off and on as fast as possible – a household garbage disposal or a vacuum cleaner for example, or in a more dangerous situation, a turbine in an airplane engine or in an electrical generator in a power plant.

For efficiency, and to prevent wear and tear on devices like disk drives, both application software and system software often use a technique known as buffering. A **buffer** is a temporary memory location that holds data moving between processes or between one device and another. Usually the two processes or devices work at

different speeds, or there is some cost of turning a device on and off – such as time or wear and tear on the hardware. In the example above, if we set up a 1 kB (1 kilobyte) disk buffer, then the algorithm would work like this.

```
byte[]  list = new byte[10000];

set up a an array of 1000 bytes to use as a buffer

open a file on the disk drive;

for i = 1 to 10,000
   {
   if the buffer is full
      {
      send all the data in the buffer to the disk drive
      empty the buffer
      }
   write list[i] to the buffer
   }

flush the buffer (make sure it is empty)
close the file on the disk drive;
```

This algorithm is a little more complicated than the previous one, but far more efficient – and it won't beat up our disk drive. It only writes to the disk drive 10 times, not 10,000. With a *5k* buffer it would only write to the drive twice.

Fortunately, most operating systems and most device control software, such as that found in a disk drive's controller chip, uses buffering. Even devices like printers and keyboards, which are much slower than CPU's and internal memory, use buffering.

We can generally rely on system software to protect equipment from damage similar to that described above, but it doesn't always result in maximum efficiency when using external storage and external devices. We can create our own code to set up and use a buffer, or we can use built in features of the *java.io* package that will do this for us.

One of the keys to good buffering is to know what size buffer is most efficient in a given situation. Typically, data moves between the CPU and the RAM through an intermediate memory called a level 2 cache. There may even be several levels of caches to speed up the system. Data usually moves between RAM and disk storage using a disk cache. In each case, the data is usually moved in blocks using a fixed block size.

Determining the optimum size for a buffer depends on knowing the specifics of the hardware and the operating system – things like the Level 2 cache block size, the disk cache block size, the cache and disk latency periods when writing or reading data, and so on.

System software specialists and computer system engineers can work together to analyze a particular situation to optimize the efficiency of the cache size for specific software. This is especially important in widely used software that is I/O intensive, such as the underlying software in database management systems like Oracle and MySQL. Selecting an optimum buffer size is also important in data communications software.

For general purpose software, many programmers suggest using a buffer size of 8K (8,192 bytes). For most purposes this seems to work well, but it may be worth it in some situations to consult with the experts.

In the following sections, as we look at working with text and binary data, we will see how to use buffering in each case. *Computer Science 213 – Computer Organization* covers topics related to buffering in more detail.

18.4 Text File I/O

Most of the I/O that you have seen so far in CSCI 111 and CSCI 112 has been text I/O. Console I/O uses the Scanner class for input and the *System.out.print(), println()*, and *printf()*, methods for output. This is text I/O. Text File I/O ca use the *Scanner* class to read from a file using the same methods used to read from the console, and the *PrintWriter* class for output with *print(), println()*, and *printf()* methods similar to *System.out*.

The *Scanner* class is perhaps the easiest class to use for reading data from text files into primitive variables and Strings in Java. Correspondingly, the *PrintWriter* class is among the easiest to use to send data from an application to a text data file.

Text File Input Using the Scanner Class

The code below shows a method from the program *tutorials* which was used in earlier chapters to demonstrate text file I/0 using Scanner and PrintWriter class objects. This method reads from a file using a Scanner class object. The complete *tutorials* NetBeans project is included with the files for this chapter. The *unsorted.txt* file is in the tutorials NetBeans project folder.

```
    /* This method reads data from the file into the array.
     * We want our array to work with up to 100 elements
     * Each line from the file will be one element in the array.
     *
     * The parameter refers to the array in the main method.
     * The method returns the number of elements it uses.
     */
     public static int readLines(String[] lines) throws Exception
    {
        int count = 0; // number of array elements with data

        // Create a File class object linked to the name of the file to read
        File unsorted = new File("unsorted.txt");

        // Create a Scanner named infile to read the input stream from the file
        Scanner infile = new Scanner(unsorted);

        /* This while loop reads lines of text into an array. it uses a Scanner class
         * boolean function hasNextLine() to see if there another line in the file.
         */

        while ( infile.hasNextLine() )
        {
            // read a line and put it in an array element
            lines[count] = infile.nextLine();
            count ++;  // increment the number of array elements with data

        } // end while
        infile.close();
```

```
        return count;    // returns the number of items used in the array.

    } // end readList()
```

The method creates a File class object named *unsorted* that is associated with the actual data file named *unsorted.txt*. It then creates a Scanner class object named *infile* and associates it with unsorted. The Scanner method *hasNextLine()* is used to determine if there is another line of data in the file. If there is, it is read in using the Scanner class method *nextLine()*, which read a line of data into a String variable. Each line is read into an element in the array *lines[]*. The method returns the number of lines read.

The *FileReader* class could be used in place of the *Scanner* class to read from the file, but it is more primitive than the *Scanner* class. For example, it does not have the *nextFile()* and *nextLine()* methods, so code that is equivalent to these methods would need to be created by the programmer. Detailed information about the *FileReader* class is online at:

http://docs.oracle.com/javase/7/docs/api/java/io/FileReader.html

Detailed information about the *Scanner* class is online at:

http://docs.oracle.com/javase/7/docs/api/java/util/Scanner.html

A summary of the use of the *Scanner* class for console I/O is in chapter 2, section 4 and in chapter 6 section 5 of this text. The chart below, similar to one in Chapter 2, describes some of the methods in the Scanner class.

java.io.Scanner	
hasnext(): Boolean	returns true if this Scanner has another token in its input
next(): String	reads the next token and returns it as a String value
nextInt(): int	reads the next token and returns it as a int value
nextLong(): long	reads the next token and returns it as a long value
nextFloat(): float	reads the next token and returns it as a float value
nextDouble(): double	reads the next token and returns it as a double value
nextLine(): String	reads an entire line and returns it as a String value (a line ends with a line feed or when someone presses *[Enter]*)
close()	closes the I/O stream associated with the Scanner class object

Text File Output Using the PrintWriter Class

The code below shows a method from the program *tutorials* that writes to a file using a *PrintWriter* class object.

```
/* This method writes an array of Strings to a text data file.
 * The first parameter refers to the array in the main method. The second parameter
 * is the number of elements in the array that actually contain data
 */

        public static void  writeLines(String[] lines, int count) throws Exception
        {
            // create a File class object and give the file the name tutorials.txt
            File tut  = new File("tutorials.txt");
            // Create a PrintWriter text output stream and link it to the file x
            PrintWriter outFile  = new PrintWriter(tut);

            int i;  // loop counter

            // iterate the elements actually used
            for ( i=0; i < count; i++)
                outFile.println(lines[i]);

            outfile.flush()
            outFile.close();

        } // end writeTextArray()
```

The code first creates a File class object named *tut* associated with the filename *tutorials.txt.* It then creates a PrintWriter object named *outFile* and associates it with *tut*.

Next, a *for* loop writes each line of text from the array *lines[]* to the actual file using *outFile.println().* The variable *count*, whose value has been passed to the method, determines how many lines are printed. Count is used instead of the length of *lines[]* because not every element in lines contains data.

After the loop is finished, the *flush()* merthod ensures no data was left in the buffer and thethe file is closed with the statement `outFile.close();`

Since Java version 7, there is a *PrintWriter* constructor that will work with a filename directly, as in this example:

```
java.io.PrintWriter outfile = new java.io.PrintWriter("tutorials.txt");
```

This replaces the two lines:

```
java.io.File tut = new java.io.File("tutorials.txt");
java.io.PrintWriter outfile = new java.io.PrintWriter(tut);
```

However, it does not create a File class object for the file; so many programmers are in the habit of using the two lines instead of one. In most cases, either technique is acceptable.

The PrintWriter class has methods to write to a file, similar to those used by *System.out* for console output. it also has *write()* and *flush()* methods, described below. The printWriter class has several additional methods not shown here, as does the Scanner class. Complete documentation for the PrintWriter class is online at:

http://docs.oracle.com/javase/7/docs/api/java/io/PrintWriter.html

java.io.PrintWriter	
print(x)	a polymorphic method that will write Strings or primitive data types to a file
println(x)	a polymorphic method that will write Strings or primitive data types to a file then send a newline character
write(x)	writes a sinlge character, an array of characters or a String to a file, similar to print()
flush()	makes sure all data in the stream has been sent, but keeps the stream open
close()	closes the I/O stream associated with the PrintWriter class object

The *FileWriter* class could be used in place of the *PrintWriter* class to write to a file, but it is more primitive than the *PrintWriter* class -- it has no *print()* or *println()* methods. Detailed information about the *FileWriter* class is online at:

http://docs.oracle.com/javase/7/docs/api/java/io/FileWriter.html

Buffering Text File I/O

Since *Java 6*, methods in the *Scanner* class and the *PrintWriter* class automatically use buffers for file I/O, but neither *FileReader* nor *FileWriter*, use buffers.

The *BufferedReader* and *BufferedWriter* classes can be used as wrapper classes to create buffered character streams for text I/O when using *FileReader* or *FileWriter.*

Here is a way to set up buffered output with FileWriter:

```
FileWriter outFile = new FileWriter("filename.txt");
BufferedWriter outBuffered = new BufferedWriter(outFile);
```

Here is a way to set up buffered input using FileReader:

```
FileReader inFile = new FileReader("filename.txt");
BufferedReader inBuffered = new BufferedReader(inFile);
```

Remember, buffering increases the efficiency of applications and decreases the wear and tear on electromechanical I/O device, such as disk drives. Operating systems might automatically provide buffering whether or not our code does, but we cannot rely on this in all cases, especially with embedded processors in simple systems.

18.5 Binary File I/O

Recall from section 1 of this chapter that a binary file is a data file containing raw binary data, not formatted as character data. The format and meaning of the data depend on the software that reads or writes the data file. Three sets of methods are commonly used for binary file I/O in Java applications:

- *FileInputStream* and *FileOutputStream* for raw binary I/O;
- *DataInputStream* and *DataOutputStream* for moving data between binary files and primitive data types (and Strings);
- *ObjectInputStream* and *ObjectOutputStream* for moving data between binary files and serialized objects, including arrays;

In the remainder of this chapter, we will examine the use of *FileInputStream* and *FileOutputStream* for raw binary I/O. Later in the semester we will see the use of the other classes for moving primitive data and objects between memory and data files.

FileInputStream and FileOutputStream

The *FileInputStream* and *FileOutputStream* classes move bytes of data between binary files and Java applications without regard to the format or meaning of the data. They provide a simple fast way to move data to and from files. Within an application, the data can be parsed using methods from other classes, or it can be passed on to other files, other devices, or other applications. However, if the data represents primitive data types, then *DataInputStream* and *DataOutputStream* can be used, and if the data represents objects, then *ObjectInputStream* and *ObjectOutputStream* can be used. So, *FileInputStream* and *FileOutputStream* are normally only used when the application is handing the data but not using the data itself in any meaningful way.

The *FileInputStream* and *FileOutputStream* classes each have less than a dozen methods. *FileInputStream* has a ***read()*** method that reads a byte of data and returns the byte as an unsigned integer (0 to 255), or as -1 if there is no more data in the file to read. *FileOutputStream* has a corresponding ***write()*** method that writes a byte as an integer to a data file. Both classes have other read and write methods to move an array of bytes and ***close()*** and ***flush()*** methods to manage the I/O streams.

Detailed specifications for *FileInputStream* can be found online at:
http://docs.oracle.com/javase/7/docs/api/java/io/FileInputStream.html

Detailed specifications for *FileOutputStream* can be found online at:
http://docs.oracle.com/javase/7/docs/api/java/io/FileOutputStream.html

The code below for ***copyFileDemoA*** shows the use of *FileInputStream* and *FileOutputStream* to copy a data file. It does so without regard to the meaning or format of the data in the file.

The *FileInputStream* and *FileOutputStream* classes do not use buffering. ***copyFileDemoA*** does not use buffering. The example following this one – ***copyFileDemoB*** – does use buffering. Both ***copyFileDemoA*** and ***copyFileDemoB*** are included with the files for this chapter zipped as NetBeans projects.

```
/* CopyFileDemoA.java
 * CSCI 112 - Spring 2014
 *
 * The software in this example copies a data file using FileInputStream and FileOutputStream
 * for simple, quick binary file I/O.
 *
 * It has no buffferng. The program CopyFileDemoA.java is a similar program,
 * using BufferedReader and BufferedWrier for buffering.
 *
 * For the code to work as written, the file CSCI.jpg must be in the
 * the NetBeans project folder or  in the dist folder with the JAR file for the code.
 *
 * Lines 29 and 30 can be re-written to copy another file.
 * The copy method can be used in other software.
 *
 * last editied 2/23/2014 by C. Herbert
 */
package copyfiledemoa;
import java.io.*;

public class CopyFileDemoA {

    public static void main(String[] args) throws Exception
    {

        /* The names of the files are hardcoded into this demo.
         * The code could be rewritten to get them from the user.
         */

        String sourceFile = "CSCI.jpg";             // the name of the source file
        String destinationFile = "CSCIcopy.jpg";    // the name of the destination file

        // call the method that copies the source to the destination
        copyFile(sourceFile, destinationFile);

    } //end main
    //*********************************************************************

    /* This method copies a file.  The parameters are strings with the names
     * of the source and destination files.
     *
     * WARNING: This method does not check to see if the destination file exists.
     *          It will overwrite existing files.
     */
     public static void copyFile(String source, String destination) throws Exception
     {
        // Create file objects for the source and destination files
        File sourceFile = new File(source);
        File destFile = new File(destination);

        // create file streams for the source and destination files
        FileInputStream sourceStream = new FileInputStream(sourceFile);
        FileOutputStream destStream = new FileOutputStream(destFile);

        // use an integer to transfer data between files
        int transfer;

        // tell the user what is happening
```

```
        System.out.println("begining file copy:");
        System.out.println("\tcopying " + source);
        System.out.println("\tto      " + destination);

        // read a byte, checking for end of file (-1 is returned by read at EOF)
        while ((transfer = sourceStream.read()) != -1) {

            // write a byte
            destStream.write(transfer);

        } // end while

        // close file streams
        sourceStream.close();
        destStream.close();

        System.out.println("File copy complete.");

    } // end copyFile
} // end class  CopyFileDemoA
```

The main method uses two Strings with the names of the source and destination files hardcoded in the program. This is only for demonstration puposes; the code could easily be re-written to ask the user for the file names.

The *main()* method calls the copyFile() method with the names of the source and destination files as parameters. You may be tempted to use the copyFile() method in other software. If you do so, don't use the copyFile() method from ***CopyFileDemoB***, use the method from ***CopyFileDemoB*****,** which works much better.

The *copyFile()* method first sets up two File class objects, *sourceFile* and *destFile* using the names of the source and destination files:

```
File sourceFile = new File(source);
File destFile = new File(destination);
```

The method then creates an I/O stream associated with each File object:

```
FileInputStream sourceStream = new FileInputStream(sourceFile);
FileOutputStream destStream = new FileOutputStream(destFile);
```

The integer variable *transfer* is used to capture the input from *sourceStream* and write the data to *destStream*. Remember, the *FileInputStream read()* method returns a byte as an integer and the *FileOutputStream write()* method writes a byte from an integer.

Each byte flows like this:

sourceFile → sourceStream → transfer → destStream → destFile

The *while* statement sets this up as a loop that repeats until the byte being read into the variable *transfer* from the *sourceStream* equals -1, which indicates the end of the file. The *FileInputStream's read()* instruction returns an integer value in the *while* loop condition, which will immediately write the value back out to *destStream,* then repeat the process until a -1 value terminates the loop.

After the loop terminates the streams are closed, which will do two things – it will make sure no data is left in the stream because a stream will be emptied before closing, and it will free any system resources used for the read and write operations. It is important to close I/O streams when we are finished with them to prevent data from the end of the file from being lost.

Listed below is the code for ***CopyFileDemoB.*** It is identical to ***CopyFileDemoA***, except that it uses *BufferedInputStream* and *BufferedOutputStream* as wrapper classes for buffering *FileInputStream* and *FileOutputStream*.

```
/* CopyFileDemoB.java
 * CSCI 112 - Spring 2014
 *
 * The software in this example copies a data file using FileInputStream and FileOutputStream
 * for simple, quick binary file I/O.
 *
 * The code uses BufferedInputStream and BufferedOutputStream to buffer the I/O.
 * CopyFileDemoB is a similar program without buffering.  You should try both
 * programs with a file of 1 Mb or more and compare their effficiency.
 * Buffering is much more efficient.
 *
 * For the code to work as written, the file CSCI.jpg must be in the
 * the NetBeans project folder or  in the dist folder with the JAR file for the code.
 *
 * Lines 32 and 33 can be re-written to copy another file.
 * The copy method can be used in other software.
 *
 * last editied 2/23/2014 by C. Herbert
 */
package copyfiledemob;
import java.io.*;

public class CopyFileDemoB {

    public static void main(String[] args) throws Exception
    {

        /* The names of the files are hardcoded into this demo.
         * The code could be rewritten to get them from the user.
         */

        String sourceFile = "CSCI.jpg";             // the name of the source file
        String destinationFile = "CSCIcopy.jpg";    // the name of the destination file

        // call the method that copies the source to the destination
        copyFile(sourceFile, destinationFile);

    } //end main()
    //*************************************************************************

    /* This method copies a file.  The parameters are strings with the names
     * of the source and destination files.
     *
     * WARNING: This method does not check to see if the destination file exists.
     *          It will overwrite existing files.
     */
```

```
    public static void copyFile(String source, String destination) throws Exception
    {
        // Create file objects for the source and destination files
        File sourceFile = new File(source);
        File destFile = new File(destination);

        // create file streams for the source and destination files
        FileInputStream sourceStream = new FileInputStream(sourceFile);
        FileOutputStream destStream = new FileOutputStream(destFile);

        // buffer the file streams -- set the buffer sizes to 8K
        BufferedInputStream bufferedSource = new BufferedInputStream(sourceStream, 8182);
        BufferedOutputStream bufferedDestination  = new BufferedOutputStream(destStream, 6192);

        // use an integer to transfer data between files
        int transfer;

        // tell the user what is happening
        System.out.println("beginning file copy:");
        System.out.println("\tcopying " + source);
        System.out.println("\tto      " + destination);

        // read a byte, checking for end of file (-1 is returned by read at EOF)
        while ((transfer = bufferedSource.read()) != -1) {

            // write a byte
            bufferedDestination.write(transfer);

        } // end while

        // close file streams
        bufferedSource.close();
        bufferedDestination.close();

        System.out.println("File copy complete.");

    } // end copyFile
} // end class  CopyFileDemoA
```

As mentioned above, the code in *CopyFileDemoB* is identical to the code for CopyFileDemoA, except that *CopyFileDemoB* uses *BufferedInputStream* for buffering incoming data and *BufferedOutputStream* for buffering outgoing data.

two instructions set up the buffering:

```
BufferedInputStream bufferedSource = new BufferedInputStream(sourceStream, 8182);
BufferedOutputStream bufferedDestination  = new BufferedOutputStream(destStream, 6192);
```

BufferedInputStream has a constructor that takes an *InputStream* and the size of the buffer it should use as parameters. It also has a constructor that takes just an *InputStream* as a constructor and uses a default buffer size. As the diagram back near the beginning of section 3 of this chapter shows us, a *FileInputStream* is a sub-class of *InputStream*, so the constructors will work with a *FileInputStream*.

BufferedInputStream constructors works the same way, using *OutputStream*, or its subclass *FileOuputStream.*

The read and write methods in the remainder of the code in the code in *CopyFileDemoB* is similar to the code in *CopyFileDemoA*, except that the *bufferedSource* and *bufferedDestination* streams are used instead of the unbuffered *sourceStream* and *destStream* for ther read, write, and close operations.

In the following exerecise you will use the two projects *CopyFileDemoA* and *CopyFileDemoB* to see the importance of buffering stream I/O, especially for longer files.

Exercise: The Importance of Buffering Binary File I/O

As we have seen, CopyFileDemoA copies a data file using *FileInputStream* and *FileOutputStream* for simple, quick binary file I/O. However, it has no buffering. *CopyFileDemoB* is a similar program but it uses *BufferedInputStream* and *BufferedOutputStream* as wrapper classes for buffering *FileInputStream* and *FileOutputStream*.

Buffering is much more efficient. How efficient? Consider the following test results:

- *CopyFileDemoA,* without buffering took 4 minutes and 10 seconds to copy a 35 Mb PDF file to the same directory as the source file on a 500Gb hard drive. The hard drive began to heat up.
- *CopyFileDemoB,* with 8K buffering, performed the same operation on the same file on the same system in less than 3 seconds.
- *CopyFileDemoB,* with 1K buffering, performed the same operation on the same file on the same system in less than 5 seconds.

CopyFileDemoA and *CopyFileDemoB* are identical except for buffering.

Buffering is not necessary, but it is a really good idea to use buffering with all file I/O. *Scanner* and *PrintWriter* for text I/O now include buffering. If you are using *FileInputStream* and *FileOutputStream* for quick binary file I/O, it won't really be quick unless you add buffering.

Buffering also results in fewer disk operations – once for each buffer full of data, as opposed to once for each byte of data. This means less wear and tear on the hard drive and a small reduction in the use of electricity, which both save money. The savings in electricity over millions of computers with thousands of read and write operations each year add up. Not only does buffering save time and money, but it also is energy efficient, potentially helping to reduce greenhouse gasses, which potentially reduces the effects of climate change, which potentially diminishes the number of snow days interfering with computer courses each year at CCP.

In the following exercise you will do your own experiment with buffered and unbuffered binary file I/O.

STEP 1.

Download and unzip the NetBeans projects *CopyFileDemoA*.

STEP 2.

Open the project in NetBeans and examine the code to see how it works. As written, it will make a copy of the 22K file *CSCI.jpg* included in the project folder. The code to copy the file is the *copyFile()* method that is invoked from the main method.

STEP 3.

Look in the project folder to see what's there. You should see both the *CSCI.jpg* file that has the CCP Computer Science logo and the file *AppendixA.pdf*, a 1.1 megabyte file with a copy of Appendix A from this textbook.

STEP 4.

Run the project and pay attention to the output and how long the project takes to run. Look in the project folder again. Now you should see the file *CSCcopyI.jpg*, a copy of the *CSCI.jpg* file.

STEP 5.

Change lines 32 and 33 in the source code's *main()* method to make a copy of the larger file *AppendixA.pdf*.

Lines 32 and 33 were originally:

```
String sourceFile = "CSCI.jpg";              // the name of the source file
String destinationFile = "CSCIcopy.jpg";     // the name of the destination file
```

You should change them to:

```
String sourceFile = "AppendixA.pdf";              // the name of the source file
String destinationFile = "AppendixAcopy.pdf";     // the name of the destination file
```

STEP 6.

Run the project again and pay attention to how long it takes this time. It usually takes 8 to 15 seconds on most systems, but the actual time is machine dependent – based on the hardware and the operating system.

STEP 7.

Run the project another time without changing anything. Notice that the program operates without warning us that the destination file, *AppendixAcopy.pdf* already exists. It simply overwrites the file and the older copy of *AppendixAcopy.pdf* is lost. One of the exercises at the end of this chapter addresses this issue.

STEP 8.

Open the PDF file *AppendixAcopy.pdf.* Then, while the PDF file is still open, run the project again. This time you should see a message telling us an exception has been generated. This happens because PDF files lock when open (on most systems) and we cannot write to an open PDF file. The exception is *thrown* by our application; we do not have exception handling in the code. Now you can close the PDF file.

STEP 9.

Download and unzip the NetBeans projects *CopyFileDemoB*.

STEP 10.

Open the project and examine the code. It should be identical to *CopyFileDemoB* except for the use of the *BufferedInputStream* and *BufferedOutputStream* for buffering in the *copyFile()* method. Lines 58 and 59 in the method set up *bufferedSource* and *bufferedDestination* as buffered I/O streams based on the first parameter in each method call, the unbuffered *FileInputStream* and *FileOutputStream objects – sourceStream* and *destStream*. The second parameter in each of the two method calls is the size of the buffer in bytes. The methods will work without this parameter, but the JVM and the operating system will set a default buffer size.

```
BufferedInputStream bufferedSource = new BufferedInputStream(sourceStream, 8182);
BufferedOutputStream bufferedDestination  = new BufferedOutputStream(destStream, 6192);
```

STEP 11.

Run the *CopyFileDemoB* project and pay attention to the output and how long it takes this software to run. There should be little visible difference between running this application and the running *CopyFileDemoA* to copy the CSCI.jpg file because of the small file size.

STEP 12.

The pdf file *AppendixAcopy.pdf* is also in the project folder for *CopyFileDemoB,* just as it was in the *CopyFileDemoA* project. Change the code in lines 32 and 33 in the *main()* method in *CopyFileDemoB* to make a copy of the longer file *AppendixAcopy.pdf*, just as you did in step 5 above for *CopyFileDemoA*, then run the program again. You should see a significant difference in performance between the two applications when they copy *longer files*, because of the buffering in *CopyFileDemoB.*

You can run each again several times to see that the difference in performance is consistent. You could also copy a longer file – 5 to 10 Megabytes, to the folders and change the programs so you can see the performance differences with these files. It will be even more significant. This is because the unbuffered version of *copyFile()* is repeatedly writing 1 byte at a time to the disk drive, whereas the buffered version is writing 8,192 bytes at a time to the disk drive.

18.6 Exception Handling when Accessing Data Files

None of the software to work with data files that we have examined so far in this chapter handles exceptions. All of it throws exceptions ultimately back to the JVM and the operating system. The error message generated during the previous exercise while attempting to write to a locked PDF file did not come from our software.

Reacall from chapter 12 that a Java method can do one of three things when the JVM detects an exceptional event and sends an exception object back to the method:

- It can **handle the exception**. For this to work, an exception handler must be built into the code. using **try-catch-finally blocks**.
- It can **throw the exception**. This means that the current method will send the exception back to the method that invoked the current method, the invoking method could throw it back further, and so on, back to the operating system,
- A method can simply **ignore the exception**. This won't work for checked exceptions, because a Java compiler will not compile code that could generate a checked exception and ignores the possibility.

The exceptions that can be generated during file access are subclasses of the *IOException* class, which are all checked exceptions. All Java code that accesses files must throw or handle these exceptions.

The details of the IOException class with a listing of its subclasses and links to the subclass descriptions is available online at:

http://docs.oracle.com/javase/7/docs/api/java/io/IOException.html

An example of one of these exception classes is *FileNotFoundException*. According to its documentation, a *FileNotFoundException* will be thrown back to our software:

> *"by the* FileInputStream*,* FileOutputStream*, and* RandomAccessFile *constructors when a file with the specified pathname does not exist. It will also be thrown by these constructors if the file does exist but for some reason is inaccessible, for example when an attempt is made to open a read-only file for writing."* *-- see http://docs.oracle.com/javase/7/docs/api/java/io/FileNotFoundException.html*

Other examples of I/O exceptions that could occur during file access are *AccessDeniedException*, *EOFException*, *FileSystemException*, *NoSuchFileException*, and *NotDirectoryException*. These are just a few of the many IO Exceptions that could be generated during file I/O.

The simplest way to handle this, and what our code does so far in this chapter, is to throw the exception back to the calling method, and so on, all the way back through a *main()* to the JVM and the operating system. (This is sometimes referred to as the "*hot potato*" technique for handling exceptions.) They will eventually send out an error message, but there is one major flaw with this approach: **The software generating the exception will stop running, which means the program will crash**. Files, file streams, buffers, and so on, will often be left just as they are when a program accessing them simply stops running.

This means data could be left in a buffer, files could be corrupted, a directory could be corrupted, and in some cases, the file system on the storage device could be corrupted. This is not a good thing. It means lost data and lost time and energy repairing the damage. If it happens on a critical server, there is the possibility that an entire network could be shut down – in some rare cases even part of the Internet. So, in a simple case on one device, the device could crash and data could be lost. In rarer but much more damaging cases, entire networks could shut down.

But the reality of crashing software because of not handling exceptions gets even worse. Several times in CSCI 111 and 112 we've heard that there are more JVMs running java on the earth than there are people. Consider some of the important applications that are based on Java software: financial systems, vehicles, hospital equipment, and so on. In some cases people can actually die if software crashes. In short, things can bad, very bad, if Java software simply crashes for things like a file not being available pr a directory name being spelled wrong.

To properly handle file I/O exceptions, code that accesses files should be enclosed in tray-catch-finally blocks. If we can find a way to recover from an error gracefully, then the catch block should do that. At a minimum the catch block should send the user or calling program information about the error that caused the exception. But the most important part of this is the finally block. Code to close files or flush buffers, or other code that should be executed whether or not an error occurs, should be placed in a finally block. Then, whether or not the error occurs, the program will perform those functions rather thansimply throwing an error because it can't access a file properly.

Ther is however a problem that could occur when trying to compile code that uses try-catch-finally blocks, which has to do with declaring File and stream variables. Recall from the quote above that FileInputStream and FileOutputStream constructors will generate an exception if a file does not exist, or if access to a file is denied, such as attempting to write to a read-only file. So, we should put statements with *FileInputStream* and *FileOutputStream* constructors in try blocks to catch these exceptions. How we do that is a bit tricky. Consider the following code :

```
try {

    // Create file objects for the source and destination files
    File sourceFile = new File(source);
    File destFile = new File(destination);

    // create file streams for the source and destination files
    FileInputStream sourceStream = new FileInputStream(sourceFile);
    FileOutputStream destStream = new FileOutputStream(destFile);

    // buffer the file streams -- set the buffer sizes to 8K
    BufferedInputStream bufferedSource = new BufferedInputStream(sourceStream, 8182);
    BufferedOutputStream bufferedDestination  = new BufferedOutputStream(destStream, 6192);

      . . .    not all of the code is not shown

    } catch (IOException e) {
        e.printStackTrace();
        System.out.println(" an unexpected I/O error occured.");

    } finally {
        // close file streams
        bufferedSource.close();
        bufferedDestination.close();
        System.out.println("Files closed. Copy complete.");
    }

      . . .    not all of the code is not shown
```

This code won't work, because it won't compile properly. The problem has to do with the scope of variables.

The declarations for the File objects, and for our I/O stream objects (FileInputStream, FileOutputStream, BufferedOutputStream, and BufferedOutputStream) are all in the try block, while code that uses them, namely the close() methods is in a different block – the finally block. You may recall from CSCI 111 that variables only exist within the block of code in which they are declared. So the variables such as *sourceFile, sourceStream*, and so on, cannot be used within the *catch* block, the *finally* block or any other place outside of the *try* block.

This seems to be a bit of Catch-22 situation (Or, a *Try-Catch-Finally-22* situation. Who knew Joseph Heller had a connection to Java?). We need to put the stream constructor in a try block and the *close()* method call that refers to it in a *finally* block, but putting the variable declaration in the try block means the *close()* statement in the *finally* block won't recognize it. What are we to do?

Fortunately there is a simple solution. Split the each instantiation into two parts. Instead of one line like this:

```
FileInputStream sourceStream = new FileInputStream(sourceFile);
```

write it as two lines:

```
FileInputStream sourceStream = null;
sourceStream = new FileInputStream(sourceFile);
```

The first part - the variable declaration, can be placed before the *try* block. Then it will work anywhere in the method, including in the *try-catch-finally* blocks. The second part – which references the *FileInputStream* constructor, can be put in the *try* block, so that any exceptions it generates will be caught.

Also, notice that the variable declaration `FileInputStream sourceStream = null;` sets the initial values of the variable to *null*. This helps to avoid a "*variable might not have been initialized*" error when compiling – another *try-catch-finally-22* situation.

The code to make this work looks like this:

```
// declare file and stream variables
File sourceFile = null;
File destFile= null;

FileInputStream sourceStream = null;
FileOutputStream destStream = null;

BufferedInputStream bufferedSource = null;
BufferedOutputStream bufferedDestination = null;


try {

// Create file objects for the source and destination files
sourceFile = new File(source);
destFile = new File(destination);

// create file streams for the source and destination files
sourceStream = new FileInputStream(sourceFile);
destStream = new FileOutputStream(destFile);

// buffer the file streams -- set the buffer sizes to 8K
bufferedSource = new BufferedInputStream(sourceStream, 8182);
bufferedDestination  = new BufferedOutputStream(destStream, 6192);

  . . .    not all of the code is not shown

} catch (IOException e) {
    e.printStackTrace();
    System.out.println(" an unexpected I/O error occured.");

} finally {
    // close file streams
    bufferedSource.close();
    bufferedDestination.close();
    System.out.println("Files closed. Copy complete.");
}

  . . .    not all of the code is not shown
```

Now, we only have one more file I/I problem to solve. If an error occurs and a stream never opens, then trying to close that stream will generate more errors, but this time in the finally block. To avoid ridiculous nested try-catch-finally blocks, we can simple use an if command to see if a stream is open before we try to close it.

Instead of:

```
// close file streams
bufferedSource.close();
```

our code should look like this:

```
// close file streams that are open (not null)
if (bufferedSource != null)
    bufferedSource.close();
```

That's it, *CopyFileDemoE* which has buffering and IO handling is included with the files for this chapter as a NetBeans project, along with *CopyFileDemoA* that has no buffering and no exception handling, and *CopyFileDemoB* which has buffering but no exceptions handling. If you are looking for a good copyFile() method to use in other software, you should use the copyFile() method from *CopyFileDemoE.*

The Top Ten Things You Should Know about Data File I/O in Java

So to summarize, here are the top ten things you should know about data file I/O in Java:

1. We should use buffering for data file I/O.
2. Scanner and PrintWriter can be used for text file I/O. They use buffering on their own.
3. *FileInputStream* and *FileOutputStream* can be used for quick, raw binary I/O, but they don't use buffering. That's a problem.
4. We can use *BufferedInputStream* and *BufferedOutputStream* as wrapper classes for buffering *FileInputStream* and *FileOutputStream*. Problem solved.
5. We should handle data file I/O exceptions with try-catch-finally blocks because file I/O errors can cause exceptions that will otherwise crash our software.
6. FileInputStream and FileOutputStream constructors can cause I/O exceptionsand shuld be in a try block.
7. Streams should always be closed, so the code to do so goes in the finally block.
8. The *try-catch-finally 22* situation arises – the stream variable declarations and their close statements can't be in mutually exclusive blocks, like *try-catch-finally* blocks.
9. But there is a solution – put File and stream variable declarations before the *try* block, then use the constructors for them in the *try* block. Don't forget to initialize the variables when they are declared.
10. We should check to see if streams are closed (maybe they never opened) before trying to close them. This can be done with a simple *if* statement.

The code for *CopyFileDemoE* with exception handling is included below. More specific exception handling is addressed in the exercises at the end of this chapter and in later chapters for specific situations dealing with data File I/O.

```
/* CopyFileDemoE.java
 * CSCI 112 - Spring 2014
 *
 * The software in this example copies a data file using FileInputStream and FileOutputStream
 * for simple, quick binary file I/O.
 *
 * The code uses BufferedInputStream and BufferedOutputStream to buffer the I/O.
 * It includes basic error handling. CopyFileDemoB is a similar program with buffering, but it
 * throws errors instead of handling them.  This is a safer method to use because it closes
 * streams in the finally block if an I/O error would otherwise leave them open.
 *
 * For the code to work as written, the file CSCI.jpg must be in the
 * the NetBeans project folder or in the dist folder with the JAR file for the code.
 *
 * Lines 32 and 33 can be re-written to copy another file.
 * The copy method can be used in other software.
 *
 * last editied 2/23/2014 by C. Herbert
 */
package copyfiledemoe;
import java.io.*;

public class CopyFileDemoE {

    public static void main(String[] args) throws Exception
    {

        /* The names of the files are hardcoded into this demo.
         * The code could be rewritten to get them from the user.
         */

        String sourceFile = "CSCI.jpg";             // the name of the source file
        String destinationFile = "CSCIcopy.jpg";    // the name of the destination file

        // call the method that copies the source to the destination
        copyFile(sourceFile, destinationFile);

    } //end main()
    //*************************************************************************

    /* This method copies a file.  The parameters are strings with the names
     * of the source and destination files.
     *
     * WARNING: This method does not check to see if the destination file exists.
     *          It will overwrite existing files.
     */
    public static void copyFile(String source, String destination) throws Exception {

        // declare File
        File sourceFile = null;
        File destFile = null;

        // declare stream variables
        FileInputStream sourceStream = null;
        FileOutputStream destStream = null;
```

```
     // declare buffering variables
        BufferedInputStream bufferedSource = null;
        BufferedOutputStream bufferedDestination = null;

        try {

            // Create file objects for the source and destination files
            sourceFile = new File(source);
            destFile = new File(destination);

            // create file streams for the source and destination files
            sourceStream = new FileInputStream(sourceFile);
            destStream = new FileOutputStream(destFile);

            // buffer the file streams -- set the buffer sizes to 8K
            bufferedSource = new BufferedInputStream(sourceStream, 8182);
            bufferedDestination = new BufferedOutputStream(destStream, 6192);

            // use an integer to transfer data between files
            int transfer;

            // tell the user what is happening
            System.out.println("begining file copy:");
            System.out.println("\tcopying " + source);
            System.out.println("\tto      " + destination);

            // read a byte, checking for end of file (-1 is returned by read at EOF)
            while ((transfer = bufferedSource.read()) != -1) {

                // write a byte
                bufferedDestination.write(transfer);

            } // end while

        } catch (IOException e) {

            e.printStackTrace();
            System.out.println(" An unexpected I/O error occurred.");

        } finally {

            // close file streams
           if (bufferedSource != null)
               bufferedSource.close();

           if (bufferedDestination != null)
            bufferedDestination.close();

           System.out.println("Files closed. Copy complete.");

        } // end finally

    } // end copyFile
} // end class CopyFileDemoE
```

Chapter Review

Section 1 of this chapter described the nature of text data files and binary data files and the advantages and disadvantages in using each type of file.

Section 2 discussed File class objects in Java and the use of File class methods to identify files and directories, get information about files and directories, and manipulate files and directories. Use of the enhanced *for* statement was reviewed, and examples showed how to get file information, list files in a directory, and create a set of directories for an application, all using File class objects.

Section 3 discussed I/O streams in Java, the concept of buffering, and the importance of buffering.

Section 4 reviewed Text File I/O in Java, focusing on the *Scanner* and *Printwriter* classes, which both automatically buffer input and output.

Section 5 introduced the use of the *FileInputStream* and *FileOutputStream* classes for binary data file I/O in Java and buffering binary I/O streams using the *BufferedInputStream* and *BufferedOutputStream* classes. An exercise using similar programs to copy data files, one without buffering and one with buffering, helped illustrate the importance of buffering.

Section 6 discussed the need for basic exception handling with try-catch-finally blocks in data file programming to prevent code from crashing with file streams still open. A summary at the end of the section listed the top ten things you should know about data file programming. A version of the file copy program that includes exception handling was shown.

Chapter Questions

1. What is the difference between a text data file and a binary data file? What are the advantages and disadvantages of using each of these?
2. What is a computing platform? How does a cross platform language like java implement platform dependent file operations? What package does java software need to import for file handling and reading and writing data files?
3. What does the dot notation on file paths mean? How are slashes and backslashes used in file path names by different operating systems? Which should we use in Java code?
4. What do we need to do to access a file or directory in our java code? What is the difference between a file's absolute path and a relative path? How is this related to a canonical path? Which are better to use on java code and why?
5. What methods can we use in the File class to get information about a data file? What values do they return and what are the return types?
6. What methods in the File class can be used tell if a file exists? What methods can be used to distinguish between a file and a directory? What valuesdo these methodsa return and what do the values mean?
7. What does the File class *list()* method do? What can it be used for? How does the *listFile()* method differ from the *list()* method?

8. How can java's *enhanced for statement* be used to help us list the contents of a directory? How does the statement work?
9. How does a java application communicate with a data file?
10. What classes are the parent classes of all other I/O stream classes in java? Which classes are commonly used for binary I/O? which for text I/O, which for object I/O, which for buffering binary I/O?
11. What is data buffering? Why do applications software and systems software use buffering? What can happen to storage devices if we routinely read from and write to large data files without buffering?
12. How does the speed of using external storage compare to using internal memory? Why is this so?
13. How is the optimum size of an I/O buffer determined? What size buffer do many programmers suggest for general purpose software? Which course at CCP studies topics related to determining optimum buffer size?
14. What method class makes reading text data file similar to reading console I/O? What class makes sending data to text data files similar to sending output to the console? What methods does the class have that we can use for this?
15. Which classes have built in data buffering for I/O with text data files? Which methods read and write text data files without buffering, and what methods can we use to set up buffered input with these methods?
16. What I/O stream methods can be used to build an application that quickly copies data from one file to another? Why is binary I/O faster than text I/O for such a purpose?
17. How fast is buffered binary data file I/O compared to unbuffered binary data file I/O for files of various sizes?
18. How and why is can an integer variable be used to transfer data between an input stream and an output stream in a java method to copy a binary file? How can we use the value of this variable to tell that we have reached the end of a data file that is being read?
19. Why is it important to include exception handling instead of just throwing exceptions in java code that accesses data files? What usually can happen if a file is not closed properly when an I/O exception occurs during a file operation?
20. Where should the close statement be in data file I/O programming that uses exception handling? What is the catch-22 problem that appears when we try to set up exception handling with a *FileInputStream or FileOutputStream* in Java, and what can we do to avoid the problem?

Chapter Exercises

1. File Access Information

Create a java application that will ask the user for a directory, then display three lists – a list of all files in the directory to which we have write access, a list of all file in the directory to which we have read access to but not write access and a list of all files in the directory to which we have neither read nor write access.

2. Creating New Directories

The Netbeans project ***CreateDirectoriesDemo*** is included with the files for this chapter as a zipped file. rewrite the program so that it asks the user for the location where the new directories are to be created, and then asks the user to enter, one at a time, the relative path names of the directories it should create.

3. Determining File Types

Files names in most modern operating systems have two parts – the main part of the name and an extension, separated by a period. Write a java application that asks the user for a directory and a file extension, then displays all of the files in the directory that have that file extension. Your solution should return the list of files in the directory as a String array, then use String class methods to determine which files to display.

4. Adding File I/O Exceptions

None of the java programming examples in section 2 of this chapter handle I/O exceptions, they only all throw exceptions. Rewrite the code to handle I/O exceptions instead.

5. Copying a Directory and its Contents

A *File* class object may refer to a data file or a directory. The Netbeans project *DirectoryListDemo*, discussed starting on page 8, shows how get information about the contents of a directory using File class methods *list()* and *listFIle()*. The Netbeans project *CreateDirectoriesDemo*, starting on page 10 shows how to make a set of directories using the File class method *mkdir()*. The NetBeans project *CopyFileDemoE* has a method to copy a file.

Your task is to use techniques from all three of these examples to create software that can be used to copy a directory and all of its contents, including subdirectories and their contents. Your program should work recursively.

Your software should:

1. Ask the user for the source directory and a destination. The source is the directory to be copied; the destination is the directory that will be the parent of the new copy.
2. First your program should make a new directory in the new location with the same name as the source directory. (You may need to do something special for root directories if you are copying an entire disk. A root directory has no parent directory, and often, no name.)
3. Then your program should create an array with File class objects for each item in the contents of the source directory, similar to what was done in *DirectoryListDemo.*
4. Next , it should iterate the array, and for each item in the array,
    a. if it is a file, copy the file to the new directory using the *copyFile()* method taken from *CopyFileDemoE.*
    b. if it is a directory, recursively call this method to copy the directory and all of its contents.

The finished program will be a very useful utility. For example, a user could put an old flash ROM drive in one USB port, and a new flash ROM Drive in another USB port, and then, assuming that the two ports are the E: and F: drives, the user could run your program and tell it to copy E:/ to F:/, which would make a copy of the USB drive.

You should submit the NetBeans project folder with your software along with a lab report.