CHAPTER 13

File Input and Output

In this chapter, you will:

- Learn about computer files
- Use the `Path` and `Files` classes
- Learn about file organization, streams, and buffers
- Use Java's IO classes to write to and read from a file
- Create and use sequential data files
- Learn about random access files
- Write records to a random access data file
- Read records from a random access data file

Understanding Computer Files

Data items can be stored on two broad types of storage devices in a computer system:

- **Volatile storage** is temporary; values that are volatile, such as those stored in variables, are lost when a computer loses power. **Random access memory (RAM)** is the temporary storage within a computer. When you write a Java program that stores a value in a variable, you are using RAM. Most computer professionals simply call volatile storage *memory*.
- **Nonvolatile storage** is permanent storage; it is not lost when a computer loses power. When you write a Java program and save it to a disk, you are using permanent storage.

When discussing computer storage, *temporary* and *permanent* refer to volatility, not length of time. For example, a *temporary* variable might exist for several hours in a large program or one that the user forgets to close, but a *permanent* piece of data might be saved and then deleted within a few seconds. In recent years, the distinction between memory and storage has blurred because many systems automatically save data to a nonvolatile device and retrieve it after a power interruption. Because you can erase data from files, some programmers prefer the term "persistent storage" to permanent storage. In other words, you can remove data from a file stored on a device such as a disk drive, so it is not technically permanent. However, the data remains in the file even when the computer loses power; so, unlike with RAM, the data persists, or perseveres.

A **computer file** is a collection of data stored on a nonvolatile device. Files exist on **permanent storage devices**, such as hard disks, Zip disks, USB drives, reels or cassettes of magnetic tape, and compact discs.

You can categorize files by the way they store data:

- **Text files** contain data that can be read in a text editor because the data has been encoded using a scheme such as ASCII or Unicode. (See Appendix B for more information on Unicode.) Some text files are **data files** that contain facts and figures, such as a payroll file that contains employee numbers, names, and salaries; some files are **program files** or **application files** that store software instructions. You have created many such files throughout this book.
- **Binary files** contain data that has not been encoded as text. Their contents are in binary format, which means that you cannot understand them by viewing them in a text editor. Examples include images, music, and the compiled program files with a .class extension that you have created using this book.

Although their contents vary, files have many common characteristics—each file has a size that specifies the space it occupies on a section of disk or other storage device, and each file has a name and a specific time of creation.

Computer files are the electronic equivalent of paper files stored in office file cabinets. When you store a permanent file, you can place it in the main or **root directory** of your storage device. If you compare computer storage to using a file cabinet drawer, saving to the root directory is equivalent to tossing a loose document into the drawer. However, for better organization, most office clerks place documents in folders, and most computer users organize their files into **folders** or **directories**. Users also can create folders within folders to form a hierarchy. A complete list of the disk drive plus the hierarchy of directories in which a file resides is its **path**. For example, the following is the complete path for a Windows file named Data.txt, which is saved on the C drive in a folder named Chapter.13 within a folder named Java:

```
C:\Java\Chapter.13\Data.txt
```

In the Windows operating system, the backslash (\ ) is the **path delimiter**—the character used to separate path components. In the Solaris (UNIX) operating system, a slash (/) is used as the delimiter.

When you work with stored files in an application, you typically perform the following tasks:

- Determining whether and where a path or file exists
- Opening a file
- Writing to a file
- Reading from a file
- Closing a file
- Deleting a file

Java provides built-in classes that contain methods to help you with these tasks.

TWO TRUTHS & A LIE

Understanding Computer Files

1. An advantage of modern computer systems is that both internal computer memory and disk storage are nonvolatile.
2. Data files contain facts and figures; program files store software instructions that might use data files.
3. A complete list of the disk drive plus the hierarchy of directories in which a file resides is the file's path.

The false statement is #1. Internal computer memory (RAM) is volatile storage; disk storage is nonvolatile.

Using the `Path` and `Files` Classes

You can use Java's **`Path` class** to create objects that contain information about files and directories, such as their locations, sizes, creation dates, and whether they even exist. You use the **`Files` class** to perform operations on files and directories, such as deleting them, determining their attributes, and creating input and output streams.

You can include the following statement in a Java program to use both the `Path` and `Files` classes:

```
import java.nio.file.*;
```

The *nio* in java.nio stands for *new input/output* because its classes are "new" in that they were developed long after the first Java versions. The `Path` and `Files` classes are new in Java 7; they replace the functionality of the `File` class used in older Java versions. If you search the Web for Java file-handling programs, you will find many that use the older `File` class.

Creating a Path

To create a `Path`, you first determine the default file system on the host computer by using a statement such as the following:

```
FileSystem fs = FileSystems.getDefault();
```

Notice that this statement creates a `FileSystem` object using the `getDefault()` method in the `FileSystems` class. The statement uses two different classes. The `FileSystem` class, without an ending *s*, is used to instantiate the object. `FileSystems`, with an ending *s*, is a class that contains **factory methods**, which assist in object creation.

After you create a `FileSystem` object, you can define a `Path` using the `getPath()` method with it:

```
Path path = fs.getPath("C:\\Java\\Chapter.13\\Data.txt");
```

Recall that the backslash is used as part of an escape sequence in Java. (For example, `'\n'` represents a newline character.) So, to enter a backslash as a path delimiter within a string, you must type two backslashes to indicate a single backslash. An alternative is to use the `FileSystem` method `getSeparator()`. This method returns the correct separator for the current operating system. For example, you can create a `Path` that is identical to the previous one using a statement such as the following:

```
Path filePath = fs.getPath("C:" + fs.getSeparator() + "Java" +
   fs.getSeparator() + "Chapter.13" + fs.getSeparator()  +
   "Data.txt");
```

Another way to create a `Path` is to use the `Paths` class (notice the name ends with *s*). The `Paths` class is a helper class that eliminates the need to create a `FileSystem` object. The `Paths` class `get()` method calls the `getPath()` method of the default file system without requiring you to instantiate a `FileSystem` object. You can create a `Path` object by using the following statement:

```
Path filePath = Paths.get("C:\\Java\\Chapter.13\\SampleFile.txt");
```

After the `Path` is created, you use its identifier (in this case, `filePath`) to refer to the file and perform operations on it. C:\Java\Chapter.13\SampleFile.txt is the full name of a stored file when the operating system refers to it, but the path is known as `filePath` within the application. The idea of a file having one name when referenced by the operating system and a different name within an application is similar to the way a student known as "Arthur" in school might be "Junior" at home. When an application declares a path and you want to use the application with a different file, you would change only the `String` passed to the instantiating method.

Every `Path` is either absolute or relative. An **absolute path** is a complete path; it does not need any other information to locate a file on a system. A **relative path** depends on other path information. A full path such as C:\Java\Chapter.13\SampleFile.txt is an absolute path. A simple path such as SampleFile.txt is relative. When you work with a path that contains only a filename, the file is assumed to be in the same folder as the program using it. Similarly, when you refer to a relative path such as Chapter.13\SampleFile.txt (without the drive letter or the top-level Java folder), the Chapter.13 folder is assumed to be a subfolder of the current directory, and SampleFile.txt is assumed to be within the folder.

For Microsoft Windows platforms, the prefix of an absolute pathname that contains a disk-drive specifier consists of the drive letter followed by a colon. For UNIX platforms, the prefix of an absolute pathname is always a forward slash.

Retrieving Information About a Path

Table 13-1 summarizes several useful `Path` methods. As you have learned with other classes, the `toString()` method is overridden from the `Object` class; it returns a `String` representation of the `Path`. Basically, this is the list of path elements separated by copies of the default separator for the operating system. The `getFileName()` method returns the last element in a list of pathnames; frequently this is a filename, but it might be a folder name.

Method	Description
`String toString()`	Returns the `String` representation of the `Path`, eliminating double backslashes
`Path getFileName()`	Returns the file or directory denoted by this `Path`; this is the last item in the sequence of name elements
`int getNameCount()`	Returns the number of name elements in the `Path`
`Path getName(int)`	Returns the name in the position of the `Path` specified by the integer parameter

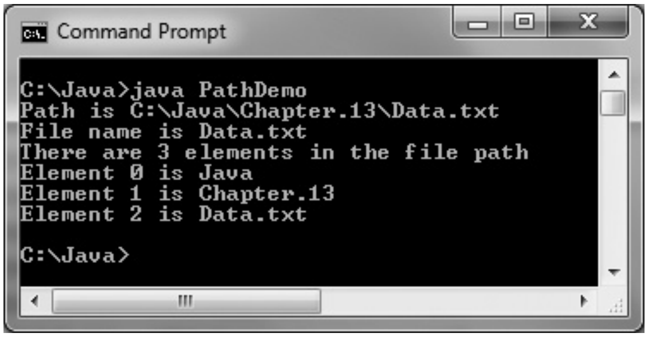
Table 13-1 Selected `Path` class methods

A `Path`'s elements are accessed using an index. The top-level element in the directory structure is located at index 0; the lowest element in the structure is accessed by the `getName()` method and has an index that is one less than the number of items on the list. You can use the `getNameCount()` method to retrieve the number of names in the list and the `getName(int)` method to retrieve the name in the position specified by the argument.

Figure 13-1 shows a demonstration program that creates a `Path` and uses some of the methods in Table 13-1. Figure 13-2 shows the output.

```
import java.nio.file.*;
public class PathDemo
{
   public static void main(String[] args)
   {
      Path filePath =
         Paths.get("C:\\Java\\Chapter.13\\Data.txt");
      int count = filePath.getNameCount();
      System.out.println("Path is " + filePath.toString());
      System.out.println("File name is " + filePath.getFileName());
      System.out.println("There are " + count +
         " elements in the file path");
      for(int x = 0; x < count; ++x)
        System.out.println("Element " + x + " is " +
           filePath.getName(x));
   }
}
```

Figure 13-1 The `PathDemo` class



```
C:\Java>java PathDemo
Path is C:\Java\Chapter.13\Data.txt
File name is Data.txt
There are 3 elements in the file path
Element 0 is Java
Element 1 is Chapter.13
Element 2 is Data.txt

C:\Java>
```

Figure 13-2 Output of the `PathDemo` application

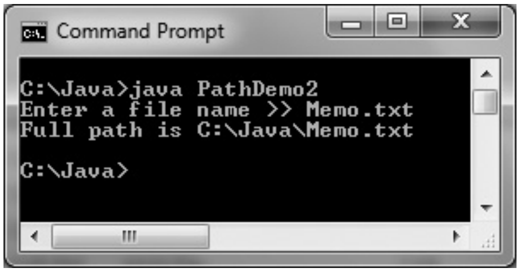
Converting a Relative Path to an Absolute One

The `toAbsolutePath()` method converts a relative path to an absolute path. For example, Figure 13-3 shows a program that asks a user for a filename and converts it to an absolute path, if necessary.

```
import java.util.Scanner;
import java.nio.file.*;
public class PathDemo2
{
   public static void main(String[] args)
   {
       String name;
       Scanner keyboard = new Scanner(System.in);
       System.out.print("Enter a file name >> ");
       name = keyboard.nextLine();
       Path inputPath = Paths.get(name);
       Path fullPath = inputPath.toAbsolutePath();
       System.out.println("Full path is " +  fullPath.toString());
   }
}
```

Figure 13-3 The PathDemo2 class

When the PathDemo2 program executes, the entered filename might represent only a relative path. If the input represents an absolute Path, the program does not modify the input, but if the input represents a relative Path, this program then creates an absolute path by assigning the file to the current directory. Figure 13-4 shows a typical program execution.



```
C:\Java>java PathDemo2
Enter a file name >> Memo.txt
Full path is C:\Java\Memo.txt

C:\Java>
```

Figure 13-4 Output of the PathDemo2 program

Checking File Accessibility

To verify that a file exists and that the program can access it as needed, you can use the checkAccess() method. The following import statement allows you to access constants that can be used as arguments to the method:

```
import static java.nio.file.AccessMode.*;
```

Assuming that you have declared a Path named filePath, the syntax you use with checkAccess() is as follows:

```
filePath.getFileSystem().provider().checkAccess();
```

You can use any of the following as arguments to the `checkAccess()` method:

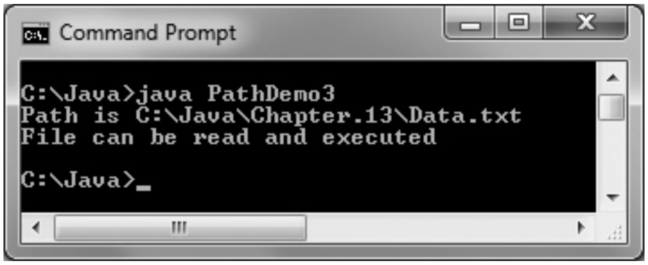
- No argument—Checks that the file exists; as an alternative, you can substitute the `Files.exists()` method and pass it a `Path` argument.
- `READ`—Checks that the file exists and that the program has permission to read the file
- `WRITE`—Checks that the file exists and that the program has permission to write to the file
- `EXECUTE`—Checks that the file exists and that the program has permission to execute the file

Java's **`static import` feature** takes effect when you place the keyword `static` between `import` and the name of the package being imported. This feature allows you to use `static` constants without their class name. For example, if you remove `static` from the `import` statement for `java.nio.file.AccessMode`, you must refer to the `READ` constant by its full name as `AccessMode.READ`; when you use `static`, you can refer to it just as `READ`.

You can use multiple arguments to the `checkAccess()` method, separated by commas. If the file cannot be accessed as indicated in the method call, an `IOException` is thrown. (Notice in Figure 13-5 that the `java.io.IOException` package must be imported because an `IOException` might be instantiated and thrown.) Figure 13-5 shows an application that declares a `Path` and checks whether the file can both be read and executed. Figure 13-6 shows the output.

```
import java.nio.file.*;
import static java.nio.file.AccessMode.*;
import java.io.IOException;
public class PathDemo3
{
   public static void main(String[] args)
   {
      Path filePath =
         Paths.get("C:\\Java\\Chapter.13\\Data.txt");
      System.out.println("Path is " + filePath.toString());
      try
      {
         filePath.getFileSystem().provider().checkAccess(filePath,READ,EXECUTE);
         System.out.println("File can be read and executed");
      }
      catch(IOException e)
      {
         System.out.println
           ("File cannot be used for this application");
      }
   }
}
```

Figure 13-5 The `PathDemo3` class



```
C:\Java>java PathDemo3
Path is C:\Java\Chapter.13\Data.txt
File can be read and executed

C:\Java>_
```

Figure 13-6 Output of the `PathDemo3` application

A program might find a file usable, but then the file might become unusable before it is actually used in a later statement. This type of program bug is called a **TOCTTOU bug** (pronounced *tock too*)—it happens when changes occur from Time Of Check To Time Of Use.

Deleting a Path

The `Files` class `delete()` method accepts a `Path` parameter and deletes the last element (file or directory) in a path or throws an exception if the deletion fails. For example:

- If you try to delete a file that does not exist, a `NoSuchFileException` is thrown.
- A directory cannot be deleted unless it is empty. If you attempt to delete a directory that contains files, a `DirectoryNotEmptyException` is thrown.
- If you try to delete a file but you don't have permission, a `SecurityException` is thrown.
- Other input/output errors cause an `IOException`.

Figure 13-7 shows a program that displays an appropriate message in each of the preceding scenarios after attempting to delete a file.

```
import java.nio.file.*;
import java.io.IOException;
public class PathDemo4
{
   public static void main(String[] args)
   {
      Path filePath =
         Paths.get("C:\\Java\\Chapter.13\\Data.txt");
      try
      {
         Files.delete(filePath)
         System.out.println("File or directory is deleted");
      }
```

Figure 13-7 The `PathDemo4` class *(continues)*

(continued)

```
      catch (NoSuchFileException e)
      {
         System.out.println("No such file or directory");
      }
      catch (DirectoryNotEmptyException e)
      {
         System.out.println("Directory is not empty");
      }
      catch (SecurityException e)
      {
         System.out.println("No permission to delete");
      }
      catch (IOException e)
      {
         System.out.println("IO exception");
      }
   }
}
```

Figure 13-7 The `PathDemo4` class

The `Files` class `deleteIfExists()` method also can be used to delete a file, but if the file does not exist, no exception is thrown.

Determining File Attributes

You can use the `readAttributes()` method of the `Files` class to retrieve useful information about a file. The method takes two arguments—a `Path` object and `BasicFileAttributes.class`—and returns an instance of the `BasicFileAttributes` class. You might create an instance with a statement such as the following:

```
BasicFileAttributes attr =
   Files.readAttributes(filePath, BasicFileAttributes.class);
```

After you have created a `BasicFileAttributes` object, you can use a number of methods for retrieving information about a file. For example, the `size()` method returns the size of a file in bytes. Methods such as `creationTime()` and `lastModifiedTime()` return important file times. Figure 13-8 contains a program that uses these methods.

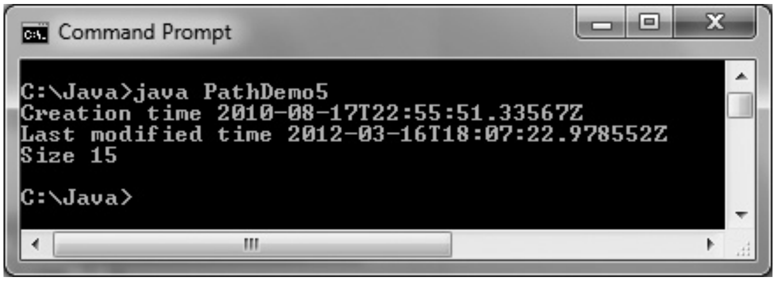
```
import java.nio.file.*;
import java.nio.file.attribute.*;
import java.io.IOException;
public class PathDemo5
{
   public static void main(String[] args)
   {
      Path filePath =
         Paths.get("C:\\Java\\Chapter.13\\Data.txt");
      try
      {
         BasicFileAttributes attr =
            Files.readAttributes(filePath, BasicFileAttributes.class);
         System.out.println("Creation time " + attr.creationTime());
         System.out.println("Last modified time " +
            attr.lastModifiedTime());
         System.out.println("Size " + attr.size());
      }
      catch(IOException e)
      {
          System.out.println("IO Exception");
      }
   }
}
```

Figure 13-8 The PathDemo5 class

The time methods in the `PathDemo5` program each return a `FileTime` object that is converted to a `String` in the `println()` method calls. `FileTime` objects are represented in the following format:

```
yyyy-mm-ddThh:mm:ss
```

In a `FileTime` object, the four-digit year is followed by the two-digit month and two-digit day. Following a *T* for *Time*, the hour, minute, and seconds (including fractions of a second) are separated by colons. You can see from the output in Figure 13-9 that the file was created in August 2010 and last modified in March 2012.



```
C:\Java>java PathDemo5
Creation time 2010-08-17T22:55:51.33567Z
Last modified time 2012-03-16T18:07:22.978552Z
Size 15

C:\Java>
```

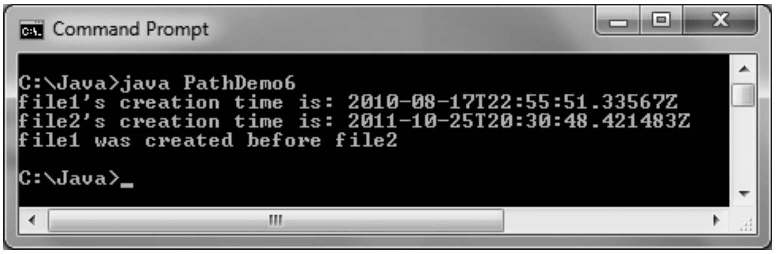
Figure 13-9 Output of the PathDemo5 program

Frequently, you don't care about a file's exact `FileTime` value, but you are interested in comparing two files. You can use the `compareTo()` method to determine the time relationships between files. For example, Figure 13-10 shows how you might compare the creation times of two files. As shown in the shaded statement, the `compareTo()` method returns a value of less than 0 if the first `FileTime` comes before the argument's `FileTime`. The method returns a value of greater than 0 if the first `FileTime` is later than the argument's, and it returns 0 if the `FileTime` values are the same.

```
import java.nio.file.*;
import java.nio.file.attribute.*;
import java.io.IOException;
public class PathDemo6
{
   public static void main(String[] args)
   {
      Path file1 =
         Paths.get("C:\\Java\\Chapter.13\\Data.txt");
      Path file2 =
         Paths.get("C:\\Java\\Chapter.13\\Data2.txt");
      try
      {
         BasicFileAttributes attr1 =
            Files.readAttributes(file1, BasicFileAttributes.class);
         BasicFileAttributes attr2 =
            Files.readAttributes(file2, BasicFileAttributes.class);
         FileTime time1 = attr1.creationTime();
         FileTime time2 = attr2.creationTime();
         System.out.println("file1's creation time is: " + time1);
         System.out.println("file2's creation time is: " + time2);
         if(time1.compareTo(time2) < 0)
            System.out.println("file1 was created before file2");
         else
            if(time1.compareTo(time2) > 0)
                System.out.println("file1 was created after file2");
            else
                System.out.println
                   ("file1 and file2 were created at the same time");
      }
      catch(IOException e)
      {
          System.out.println("IO Exception");
      }
   }
}
```

Figure 13-10 The `PathDemo6` class

Figure 13-11 shows the output of the application in Figure 13-10. The file named file1 was created in August 2010, and file2 was created in October 2011, so the program correctly determines that file1 was created first.



```
C:\Java>java PathDemo6
file1's creation time is: 2010-08-17T22:55:51.33567Z
file2's creation time is: 2011-10-25T20:30:48.421483Z
file1 was created before file2

C:\Java>_
```

Figure 13-11 Output of the `PathDemo6` program

Besides `BasicFileAttributes`, Java supports specialized classes for DOS file attributes used on DOS systems and POSIX file attributes used on systems such as UNIX. For example, DOS files might be *hidden* or *read only* and UNIX files might have a group owner. For more details on specialized file attributes, visit the Java Web site.

Watch the video *Paths and Attributes*.

TWO TRUTHS & A LIE

Using the `Path` and `Files` Classes

1. Java's `Path` class is used to create objects that contain information to specify the location of files or directories.
2. A relative path is a complete path; it does not need any other information to locate a file on a system.
3. You can use the `readAttributes()` method of the `Files` class to retrieve information about a file, such as its size and when it was created.

The false statement is #2. A relative path depends on other path information. An absolute path is a complete path; it does not need any other information to locate a file on a system.

File Organization, Streams, and Buffers

Most businesses generate and use large quantities of data every day. You can store data in variables within a program, but such storage is temporary. When the application ends, the variables no longer exist and the data values are lost. Variables are stored in the computer's main or primary memory (RAM). When you need to retain data for any significant amount of time, you must save the data on a permanent, secondary storage device, such as a disk.

Businesses organize data in a hierarchy, as shown in Figure 13-12. The smallest useful piece of data to most users is the character. A **character** can be any letter, number, or other special symbol (such as a punctuation mark) that makes up data. Characters are made up of bits (the zeros and ones that represent computer circuitry), but people who use data typically do not care whether the internal representation for an *A* is 01000001 or 10111110. Rather, they are concerned with the meaning of *A*—for example, it might represent a grade in a course, a person's initial, or a company code. In computer terminology, a character can be any group of bits, and it does not necessarily represent a letter or number; for example, some "characters" produce a sound or control the display. Also, characters are not necessarily created with a single keystroke; for example, escape sequences are used to create the `'\n'` character, which starts a new line, and `'\\'`, which represents a single backslash. Sometimes, you can think of a character as a unit of information instead of data with a particular appearance. For example, the mathematical character pi (π) and the Greek letter pi look the same, but have two different Unicode values.

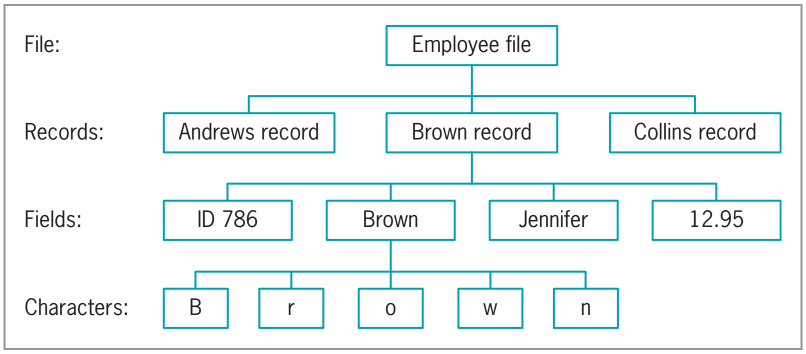


Figure 13-12 Data hierarchy

When businesses use data, they group characters into fields. A **field** is a group of characters that has some meaning. For example, the characters *T*, *o*, and *m* might represent your first name. Other data fields might represent items such as last name, Social Security number, zip code, and salary.

Fields are grouped together to form records. A **record** is a collection of fields that contain data about an entity. For example, a person's first and last names, Social Security number, zip code, and salary represent that person's record. When programming in Java, you have created many classes, such as an `Employee` class or a `Student` class. You can think of the data typically stored in each of these classes as a record. These classes contain individual variables that represent data fields. A business's data records usually represent a person, item, sales transaction, or some other concrete object or event.

Records are grouped to create files. Data files consist of related records, such as a company's personnel file that contains one record for each company employee. Some files have only a few records; perhaps your professor maintains a file for your class with 25 records—one record for each student. Other files contain thousands or even millions of records. For example, a large insurance company maintains a file of policyholders, and a mail-order catalog company maintains a file of available items. A data file can be used as a **sequential access file** when each record is accessed one after another in the order in which it was stored. Most frequently, each record is stored in order based on the value in some field; for example, employees might be stored in Social Security number order, or inventory items might be stored in item number order. When records are not used in sequence, the file is used as a random access file. You will learn more about random access files later in this chapter.

When records are stored in a data file, their fields can be organized one to a line, or a character can be used to separate them. A file of **comma-separated values (CSV)** is one in which each value in a record is separated from the next by a comma; CSV is a widely used format for files used in all sorts of applications, including databases and spreadsheets. Later in this chapter, you will see examples of CSV files.

Before an application can use a data file, it must open the file. A Java application **opens a file** by creating an object and associating a stream of bytes with it. Similarly, when you finish using a file, the program should **close the file**—that is, make it no longer available to your application. If you fail to close an input file—a file from which you are reading data—there usually are no serious consequences; the data still exists in the file. However, if you fail to close an output file—a file to which you are writing data—the data might become inaccessible. You should always close every file you open, and usually you should close the file as soon as you no longer need it. When you leave a file open for no reason, you use computer resources, and your computer's performance suffers. Also, particularly within a network, another program might be waiting to use the file.

Whereas people view a file as a series of records, with each record containing data fields, Java does not automatically attribute such meaning to a file's contents. Instead, Java simply views a file as a series of bytes. When you perform an input operation in an application, you can picture bytes flowing into your program from an input device through a **stream**, which functions as a pipeline or channel. When you perform output, some bytes flow out of your application through another stream to an output device, as shown in Figure 13-13. A stream is an object, and like all objects, streams have data and methods. The methods allow you to perform actions such as opening, closing, reading, and writing.

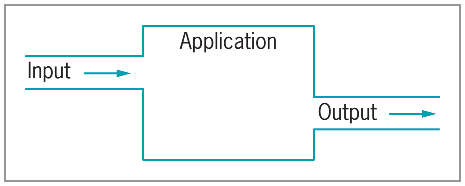


Figure 13-13 File streams

Most streams flow in only one direction; each stream is either an input or output stream. (Random access files use streams that flow in two directions. You will use a random access file later in this chapter.) You might open several streams at once within an application. For example, an application that reads a data disk and separates valid records from invalid ones might require three streams. The data arrives via an input stream; one output stream writes some records to a file of valid records, and another output stream writes other records to a file of invalid records.

Input and output operations are usually the slowest in any computerized system because of limitations imposed by the hardware. For that reason, professional programs often employ buffers. A **buffer** is a memory location where bytes are held after they are logically output but before they are sent to the output device. Using a buffer to accumulate input or output before issuing the actual IO command improves program performance. When you use an output buffer, you sometimes flush it before closing it. **Flushing** clears any bytes that have been sent to a buffer for output but have not yet been output to a hardware device.

Watch the video *File Organization, Streams, and Buffers*.

TWO TRUTHS & A LIE

File Organization, Streams, and Buffers

1. A field is a group of characters that has some meaning; a record is a collection of fields.
2. A data file is used as a sequential access file when the first field for each record is stored first in a file, the second field for each record is stored next, and so on.
3. Java views files as a series of bytes that flow into and out of your applications through a stream.

The false statement is #2. A data file is used as a sequential access file when each record is stored in order based on the value in some field.

Using Java's IO Classes

Figure 13-14 shows a partial hierarchical relationship of some of the classes Java uses for input and output (IO) operations; it shows that `InputStream`, `OutputStream`, and `Reader` are subclasses of the `Object` class. All three of these classes are abstract. As you learned in Chapter 11, abstract classes contain methods that must be overridden in their child classes. The figure also shows the major IO child classes that you will study in this chapter. The capabilities of these classes are summarized in Table 13-2.

```
Object
|
+--InputStream
|   |
|   +--FileInputStream
|   |
|   +--FilterInputStream
|         |
|         +--BufferedInputStream
|
+--OutputStream
|   |
|   +--FileOutputStream
|   |
|   +--FilterOutputStream
|         |
|         +--BufferedOutputStream
|         |
|         +--PrintStream
|
+--Reader
    |
    +--BufferedReader
    |
    +--BufferedWriter
```

Figure 13-14 Relationship of selected IO classes

Class	Description
`InputStream`	Abstract class that contains methods for performing input
`FileInputStream`	Child of `InputStream` that provides the capability to read from disk files
`BufferedInputStream`	Child of `FilterInputStream`, which is a child of `InputStream`; `BufferedInputStream` handles input from a system's standard (or default) input device, usually the keyboard
`OutputStream`	Abstract class that contains methods for performing output
`FileOutputStream`	Child of `OutputStream` that allows you to write to disk files
`BufferedOutputStream`	Child of `FilterOutputStream`, which is a child of `OutputStream`; `BufferedOutputStream` handles input from a system's standard (or default) output device, usually the monitor
`PrintStream`	Child of `FilterOutputStream`, which is a child of `OutputStream`; `System.out` is a `PrintStream` object

Table 13-2 Description of selected classes used for input and output *(continues)*

(continued)

Class	Description
`Reader`	Abstract class for reading character streams; the only methods that a subclass must implement are `read(char[], int, int)` and `close()`
`BufferedReader`	Reads text from a character-input stream, buffering characters to provide for efficient reading of characters, arrays, and lines
`BufferedWriter`	Writes text to a character-output stream, buffering characters to provide for the efficient writing of characters, arrays, and lines

Table 13-2 Description of selected classes used for input and output

As its name implies, the `OutputStream` class can be used to produce output. Table 13-3 shows some of the class's common methods. You can use `OutputStream` to write all or part of an array of bytes. When you finish using an `OutputStream`, you usually want to flush and close it.

`OutputStream` **Method**	**Description**
`void close()`	Closes the output stream and releases any system resources associated with the stream
`void flush()`	Flushes the output stream; if any bytes are buffered, they will be written
`void write(byte[] b)`	Writes all the bytes to the output stream from the specified byte array
`void write(byte[] b, int off, int len)`	Writes bytes to the output stream from the specified byte array starting at offset position `off` for a length of `len` characters

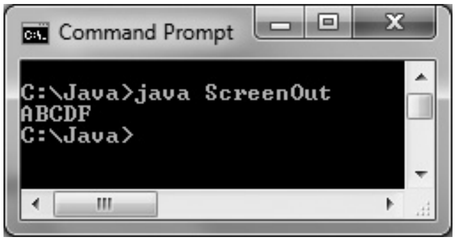
Table 13-3 Selected `OutputStream` methods

Java's `System` class contains a `PrintStream` object named `System.out`; you have used this object extensively in the book, along with its `print()` and `println()` methods. Besides `System.out`, the `System` class defines a `PrintStream` object named `System.err`. The output from `System.err` and `System.out` can go to the same device; in fact, `System.err` and `System.out` are both directed by default to the command line on the monitor. The difference is that `System.err` is usually reserved for error messages, and `System.out` is reserved for valid output. You can direct either `System.err` or `System.out` to a new location, such as a disk file or printer. For example, you might want to keep a hard copy (printed) log of the error messages generated by a program but direct the standard output to a disk file.

Although you usually have no need to do so, you can create your own `OutputStream` object and assign `System.out` to it. Figure 13-15 shows how this works. The application declares a `String` of letter grades allowed in a course. Then, the `getBytes()` method converts the `String` to an array of `bytes`. An `OutputStream` object is declared, and `System.out` is assigned to the `OutputStream` reference in a `try` block. The `write()` method accepts the `byte` array and sends it to the output device, and then the output stream is flushed and closed. Figure 13-16 shows the execution.

```
import java.io.*;
public class ScreenOut
{
   public static void main(String[] args)
   {
      String s = "ABCDF";
      byte[] data = s.getBytes();
      OutputStream output = null;
      try
      {
         output = System.out;
         output.write(data);
         output.flush();
         output.close();
      }
      catch(Exception e)
      {
         System.out.println("Message: " + e);
      }
   }
}
```

Figure 13-15 The `ScreenOut` class



```
C:\Java>java ScreenOut
ABCDF
C:\Java>
```

Figure 13-16 Output of the `ScreenOut` program

Writing to a File

The output in Figure 13-16 is not very impressive. Before you knew about streams, you wrote applications that displayed a string on the monitor by using the automatically created `System.out` object, so the application in Figure 13-15 might seem to contain a lot of unnecessary work at first. However, other output devices can be assigned to `OutputStream` references, allowing your applications to save data to them.

Instead of assigning the standard output device to `OutputStream`, you can assign a file. To accomplish this, you can construct a `BufferedOutputStream` object and assign it to the `OutputStream`. If you want to change an application's output device, you don't have to modify the application except to assign a new object to the `OutputStream`; the rest of the logic remains the same. Java lets you assign a file to a `Stream` object so that screen output and file output work in exactly the same manner.

You can create a writeable file by using the Files class newOutputStream() method. You pass a Path and a StandardOpenOption argument to this method. The method creates a file if it does not already exist, opens the file for writing, and returns an OutputStream that can be used to write bytes to the file. Table 13-4 shows the StandardOpenOption arguments you can use as the second argument to the newOutputStream() method. If you do not specify any options and the file does not exist, a new file is created. If the file exists, it is truncated. In other words, specifying no option is the same as specifying both CREATE and TRUNCATE_EXISTING.

StandardOpenOption	Description
WRITE	Opens the file for writing
APPEND	Appends new data to the end of the file; use this option with WRITE or CREATE
TRUNCATE_EXISTING	Truncates the existing file to 0 bytes so the file contents are replaced; use this option with the WRITE option
CREATE_NEW	Creates a new file only if it does not exist; throws an exception if the file already exists
CREATE	Opens the file if it exists or creates a new file if it does not
DELETE_ON_CLOSE	Deletes the file when the stream is closed; used most often for temporary files that exist only for the duration of the program

Table 13-4 Selected StandardOpenOption constants

Figure 13-17 shows an application that writes a String of bytes to a file. The only differences from the preceding ScreenOut class are shaded in the figure and summarized here:

- Additional import statements are used.
- The class name is changed.
- A Path is declared for a Grades.txt file.
- Instead of assigning System.out to the OutputStream reference, a BufferedOutputStream object is assigned.

```
import java.nio.file.*;
import java.io.*;
import static java.nio.file.StandardOpenOption.*;

public class FileOut
{
   public static void main(String[] args)
   {
      Path file =
         Paths.get("C:\\Java\\Chapter.13\\Grades.txt");
      String s = "ABCDF";
```

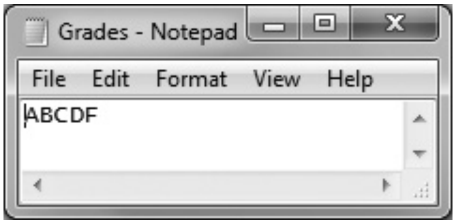
Figure 13-17 The FileOut class *(continues)*

(continued)

```
        byte[] data = s.getBytes();
        OutputStream output = null;
        try
        {
           output = new
             BufferedOutputStream(Files.newOutputStream(file, CREATE));
           output.write(data);
           output.flush();
           output.close();
        }
        catch(Exception e)
        {
           System.out.println("Message: " + e);
        }
     }
  }
```

Figure 13-17 The `FileOut` class

When the `FileOut` program executes, no output appears on the monitor, but a file is created. Figure 13-18 shows the file when it is opened in Notepad.



```
ABCDF
```

Figure 13-18 Contents of the Grades.txt file created by the `FileOut` program

Reading from a File

You use an `InputStream` like you use an `OutputStream`. If you want, you can create an `InputStream`, assign `System.in` to it, and use the class's `read()` method with the created object to retrieve keyboard input. Usually, however, it is more efficient to use the `Scanner` class for keyboard input and to use the `InputStream` class to input data that has been stored in a file.

To open a file for reading, you can use the `Files` class `newInputStream()` method. This method accepts a `Path` parameter and returns a stream that can read bytes from a file. Figure 13-19 shows a `ReadFile` class that reads from the Grades.txt file created earlier. The `Path` is declared, an `InputStream` is declared using the `Path`, and, in the first shaded statement in the figure, a stream is assigned to the `InputStream` reference.

```
import java.nio.file.*;
import java.io.*;
public class ReadFile
{
   public static void main(String[] args)
   {
      Path file = Paths.get("C:\\Java\\Chapter.13\\Grades.txt");
      InputStream input = null;
      try
      {
         input = Files.newInputStream(file);
         BufferedReader reader = new
           BufferedReader(new InputStreamReader(input));
         String s = null;
         s = reader.readLine();
         System.out.println(s);
         input.close();
      }
      catch (IOException e)
      {
         System.out.println(e);
      }
   }
}
```

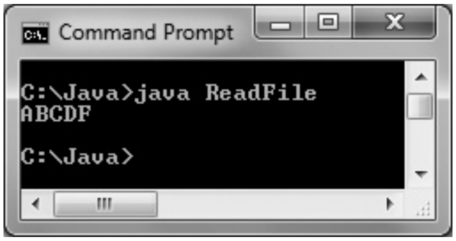
Figure 13-19 The `ReadFile` class

If you needed to read multiple lines from the file in the program in Figure 13-19, you could use a loop such as the following:

```
while(s = reader.readLine() != null)
   System.out.println(s);
```

This loop continuously reads and displays lines from the file until the `readLine()` method returns `null`, indicating that no more data is available.

In the second shaded statement in the `ReadFile` class, a `BufferedReader` is declared. A `BufferedReader` reads a line of text from a character-input stream, buffering characters so that reading is more efficient. Figure 13-20 shows the `ReadFile` program's execution. The `readLine()` method gets the single line of text from the Grades.txt file, and then the line is displayed.



```
C:\Java>java ReadFile
ABCDF

C:\Java>
```

Figure 13-20 Execution of the `ReadFile` program

When you use the `BufferedReader` class, you must import the `java.io` package into your program. Table 13-5 shows some useful `BufferedReader` methods.

`BufferedReader` Method	Description
`close()`	Closes the stream and any resources associated with it
`read()`	Reads a single character
`read(char[] buffer, int off, int len)`	Reads characters into a portion of an array from position `off` for `len` characters
`readLine()`	Reads a line of text
`skip(long n)`	Skips the specified number of characters

Table 13-5 Selected `BufferedReader` methods

TWO TRUTHS & A LIE

Using Java's IO Classes

1. Java's `InputStream`, `OutputStream`, and `Reader` classes are used for handling input and output.
2. You can create your own `OutputStream` object, assign `System.out` to it, and use it for writing output to the screen, or you can use the `Files` class `newOutputStream()` method to create a file and open it for writing.
3. To open a file for reading, you can use the `newOutputStream()` method to get a stream that can read bytes from a file.

The false statement is #3. To open a file for reading, you can use the `newInputStream()` method to get a stream that can read bytes from a file.

Creating and Using Sequential Data Files

Frequently, you want to save more than a single `String` to a file. For example, you might have a data file of personnel records that includes an ID number, name, and pay rate for each employee in your organization. Figure 13-21 shows a program that reads employee ID numbers, names, and pay rates from the keyboard and sends them to a comma-separated file.

```
import java.nio.file.*;
import java.io.*;
import static java.nio.file.StandardOpenOption.*;
import java.util.Scanner;
public class WriteEmployeeFile
{
   public static void main(String[] args)
   {
      Scanner input = new Scanner(System.in);
      Path file =
         Paths.get("C:\\Java\\Chapter.13\\Employees.txt");
      String s = "";
      String delimiter = ",";
      int id;
      String name;
      double payRate;
      final int QUIT = 999;
      try
      {
         OutputStream output = new
            BufferedOutputStream(Files.newOutputStream(file, CREATE));
         BufferedWriter writer = new
            BufferedWriter(new OutputStreamWriter(output));
         System.out.print("Enter employee ID number >> ");
         id = input.nextInt();
         while(id != QUIT)
         {
            System.out.print("Enter name for employee #" +
               id + " >> ");
            input.nextLine();
            name = input.nextLine();
            System.out.print("Enter pay rate >> ");
            payRate = input.nextDouble();
            s = id + delimiter + name + delimiter + payRate;
            writer.write(s, 0, s.length());
            writer.newLine();
            System.out.print("Enter next ID number or " +
               QUIT + " to quit >> ");
            id = input.nextInt();
         }
         writer.close();
      }
      catch(Exception e)
      {
         System.out.println("Message: " + e);
      }
   }
}
```

Figure 13-21 The `WriteEmployeeFile` class

In Figure 13-21, notice the extra `nextLine()` call after the employee's ID number is entered. Recall from Chapter 2 that this extra call is necessary to consume the newline character that remains in the input buffer after the ID number is accepted.

The first shaded statement in the `WriteEmployeeFile` program creates a `BufferedWriter` named `writer`. The `BufferedWriter` class is the counterpart to `BufferedReader`. It writes text to an output stream, buffering the characters. The class has three overloaded `write()` methods that provide for efficient writing of characters, arrays, and strings, respectively. Table 13-6 contains all the methods defined in the `BufferedWriter` class.

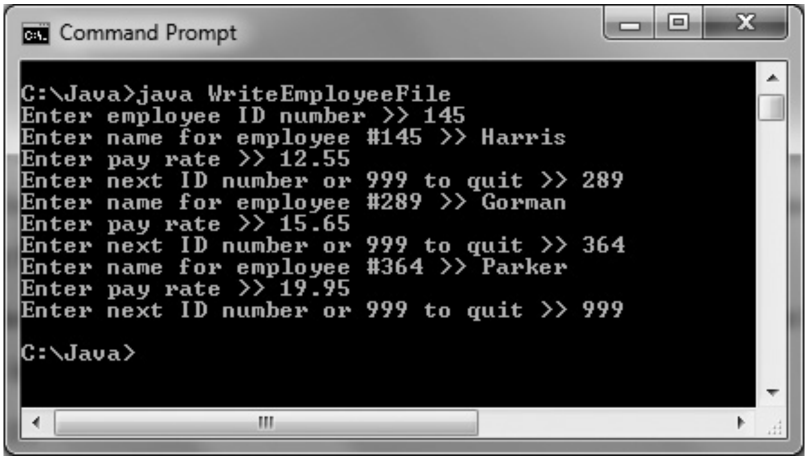
`BufferedWriter` Method	Description
`close()`	Closes the stream, flushing it first
`flush()`	Flushes the stream
`newline()`	Writes a line separator
`write(String s, int off, int len)`	Writes a `String` from position `off` for length `len`
`write(char[] array, int off, int len)`	Writes a character array from position `off` for length `len`
`write(int c)`	Writes a single character

Table 13-6 `BufferedWriter` methods

In the `WriteEmployeeFile` program, `String`s of employee data are constructed within a loop that executes while the user does not enter the `QUIT` value. When a `String` is complete—that is, when it contains an ID number, name, and pay rate separated with commas—the `String` is sent to `writer` in the second shaded statement in the class. The `write()` method accepts the `String` from position 0 for its entire length.

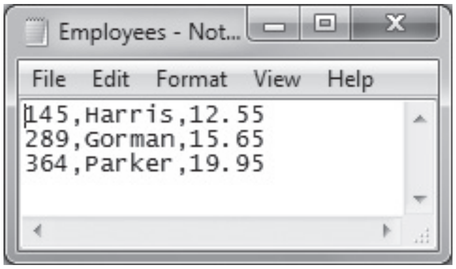
After the `String` is written, the system's newline character is also written. Although a data file would not require a newline character after each record (each new record could be separated with a comma or any other unique character that was not needed as part of the data), placing each record on a new line makes the output file easier for a person to read and interpret. Because not all platforms use `'\n'` to separate lines, the `BufferedWriter` class contains a `newLine()` method that uses the current platform's line separator. Alternatively, you could write the value of `System.getProperty("line.separator ")`. This method call returns the default line separator for a system; the same separator is supplied either way because the `newLine()` method actually calls the `System.getProperty()` method for you.

Any of the input or output methods in the `WriteEmployeeFile` program might throw an exception, so all the relevant code in the class is placed in a `try` block. Figure 13-22 shows a typical program execution, and Figure 13-23 shows the output file when it is opened in Notepad.



```
C:\Java>java WriteEmployeeFile
Enter employee ID number >> 145
Enter name for employee #145 >> Harris
Enter pay rate >> 12.55
Enter next ID number or 999 to quit >> 289
Enter name for employee #289 >> Gorman
Enter pay rate >> 15.65
Enter next ID number or 999 to quit >> 364
Enter name for employee #364 >> Parker
Enter pay rate >> 19.95
Enter next ID number or 999 to quit >> 999

C:\Java>
```

Figure 13-22 Typical execution of the `WriteEmployeeFile` program



```
145,Harris,12.55
289,Gorman,15.65
364,Parker,19.95
```

Figure 13-23 Output file following the program execution in Figure 13-22

Figure 13-24 shows a program that reads the Employees.txt file created by the `WriteEmployeeFile` program. The program declares an `InputStream` for the file, then creates a `BufferedReader` using the `InputStream`. The first line is read into a `String`; as long as the `readLine()` method does not return `null`, the `String` is displayed and a new line is read.

```
import java.nio.file.*;
import java.io.*;
public class ReadEmployeeFile
{
   public static void main(String[] args)
   {
      Path file =
         Paths.get("C:\\Java\\Chapter.13\\Employees.txt");
      String s = "";
      try
```

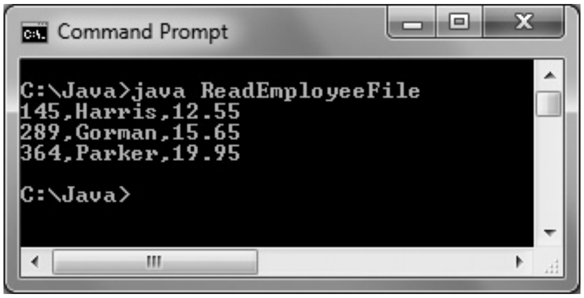
Figure 13-24 The `ReadEmployeeFile` class *(continues)*

(continued)

```
        {
           InputStream input = new
              BufferedInputStream(Files.newInputStream(file));
           BufferedReader reader = new
              BufferedReader(new InputStreamReader(input));
           s = reader.readLine();
           while(s != null)
           {
              System.out.println(s);
              s = reader.readLine();
           }
           reader.close();
        }
        catch(Exception e)
        {
          System.out.println("Message: " + e);
        }
     }
  }
```

Figure 13-24 The `ReadEmployeeFile` class

Figure 13-25 shows the output of the `ReadEmployeeFile` program when it uses the file that was created during the execution in Figure 13-22. Each comma-separated `String` is displayed on its own line.



```
C:\Java>java ReadEmployeeFile
145,Harris,12.55
289,Gorman,15.65
364,Parker,19.95

C:\Java>
```

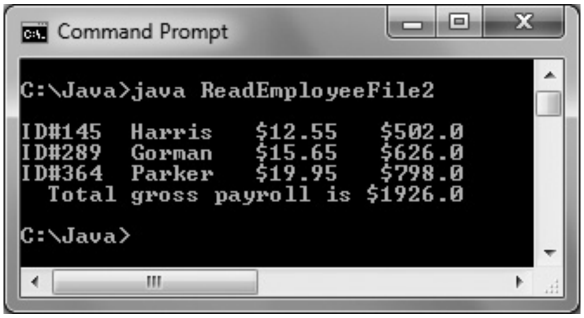
Figure 13-25 Output of the `ReadEmployeeFile` program

Many applications would not want to use the file data only as a `String` like the `ReadEmployeeFile` program does. Figure 13-26 shows a more useful program in which the retrieved file `Strings` are split into usable fields. The `String` class `split()` method accepts an argument that identifies the field delimiter (in this case, a comma) and returns an array of `Strings`. Each array element holds one field. Then methods such as `parseInt()` and `parseDouble()` can be used to reformat the split `Strings` into their respective data types.

```
import java.nio.file.*;
import java.io.*;
public class ReadEmployeeFile2
{
   public static void main(String[] args)
   {
      Path file =
         Paths.get("C:\\Java\\Chapter.13\\Employees.txt");
      String[] array = new String[3];
      String s = "";
      String delimiter = ",";
      int id;
      String name;
      double payRate;
      double gross;
      final double HRS_IN_WEEK = 40;
      double total = 0;
      try
      {
         InputStream input = new
            BufferedInputStream(Files.newInputStream(file));
         BufferedReader reader = new
            BufferedReader(new InputStreamReader(input));
         System.out.println();
         s = reader.readLine();
         while(s != null)
         {
            array = s.split(delimiter);
            id = Integer.parseInt(array[0]);
            name = array[1];
            payRate = Double.parseDouble(array[2]);
            gross = payRate * HRS_IN_WEEK;
            System.out.println("ID#" + id + "   " + name +
               "   $" + payRate + "   $" + gross);
            total += gross;
            s = reader.readLine();
         }
         reader.close();
      }
      catch(Exception e)
      {
        System.out.println("Message: " + e);
      }
      System.out.println("  Total gross payroll is $" + total);
   }
}
```

Figure 13-26 The ReadEmployeeFile2 class

As each record is read and split in the `ReadEmployeeFile2` class, its pay rate field is used to calculate gross pay for the employee based on a 40-hour workweek. Then the gross is accumulated to produce a total gross payroll that is displayed after all the data has been processed. Figure 13-27 shows the program's execution.



```
C:\Java>java ReadEmployeeFile2

ID#145   Harris    $12.55   $502.0
ID#289   Gorman    $15.65   $626.0
ID#364   Parker    $19.95   $798.0
  Total gross payroll is $1926.0

C:\Java>
```

Figure 13-27 Execution of the `ReadEmployeeFile2` program

TWO TRUTHS & A LIE

Creating and Using Sequential Data Files

1. A `BufferedWriter` writes text to an output stream, buffering the characters.
2. A data file does not require a newline character after each record, but adding a newline makes the output file easier for a person to read and interpret.
3. The `String` class `split()` method converts parts of a `String` to `int`s, `double`s, and other data types.

The false statement is #3. The `String` class `split()` method accepts an argument that identifies a field delimiter and returns an array of `String`s in which each array element holds one field. Then you can use methods such as `parseInt()` and `parseDouble()` to convert the split `String`s to other data types.

Learning About Random Access Files

The file examples in the first part of this chapter have been sequential access files, which means that you work with the records in sequential order from beginning to end. For example, in the `ReadEmployeeFile` programs, if you write an employee record with an ID number of 145, and then write a second record with an ID number of 289, the records remain in the original data-entry order when you retrieve them. Businesses store data in sequential order when they use the records for **batch processing**, which involves performing the same tasks with many records, one after the other. For example, when a company produces customer bills, the records for the billing period are gathered in a batch and the bills are calculated and printed in sequence. It really doesn't matter whose bill is produced first because none are distributed to customers until all bills in a group have been printed and mailed.

Besides indicating a system that works with many records, the term *batch processing* can refer to a system in which you issue many operating-system commands as a group.

For many applications, sequential access is inefficient. These applications, known as **real-time** applications, require that a record be accessed immediately while a client is waiting. A program in which the user makes direct requests is an **interactive program**. For example, if a customer telephones a department store with a question about a monthly bill, the customer service representative does not want to access every customer account in sequence. Suppose that the store's database contains tens of thousands of account records to read and that the customer record in question is near the end of the list. It would take too long to access the customer's record if all the records had to be read sequentially. Instead, customer service representatives require **random access files**—files in which records can be retrieved directly in any order. Random files are also called **direct access files** or **instant access files**.

You can use Java's `FileChannel` class to create your own random access files. A **file channel** object is an avenue for reading and writing a file. A file channel is **seekable**, meaning you can search for a specific file location and operations can start at any specified position. Table 13-7 describes some `FileChannel` methods.

`FileChannel` method	**Description**
`FileChannel open(Path file, OpenOption... options)`	Opens or creates a file, returning a file channel to access the file
`long position()`	Returns the channel's file position
`FileChannel position(long newPosition)`	Sets the channel's file position
`int read(ByteBuffer buffer)`	Reads a sequence of bytes from the channel into the buffer
`long size()`	Returns the size of the channel's file
`int write(ByteBuffer buffer)`	Writes a sequence of bytes to the channel from the buffer

Table 13-7 Selected `FileChannel` methods

Several methods in Table 13-7 use a `ByteBuffer` object. As its name describes, a `ByteBuffer` is simply a holding place for bytes waiting to be read or written. An array of bytes can be **wrapped**, or encompassed, into a `ByteBuffer` using the `ByteBuffer wrap()` method. Wrapping a byte array into a buffer causes changes made to the buffer to change the array as well, and causes changes made to the array to change the buffer. Creating a usable `FileChannel` for randomly writing data requires creating a `ByteBuffer` and several other steps:

- You can use the `Files` class `newByteChannel()` method to get a `ByteChannel` for a `Path`. The `newByteChannel()` method accepts `Path` and `StandardOpenOption` arguments that specify how the file will be opened.
- The `ByteChannel` returned by the `newByteChannel()` method can then be cast to a `FileChannel` using a statement similar to the following:

  ```
  FileChannel fc = (FileChannel)Files.newByteChannel(file, READ, WRITE);
  ```

- You can create a `byte` array. For example, a `byte` array can be built from a `String` using the `getBytes()` method as follows:

  ```
  String s = "XYZ";
  byte[] data = s.getBytes();
  ```

- The `byte` array can be wrapped into a `ByteBuffer` as follows:

  ```
  ByteBuffer out = ByteBuffer.wrap(data);
  ```

- Then the filled `ByteBuffer` can be written to the declared `FileChannel` with a statement such as the following:

  ```
  fc.write(out);
  ```

- You can test whether a `ByteBuffer`'s contents have been used up by checking the `hasRemaining()` method.
- After you have written the contents of a `ByteBuffer`, you can write the same `ByteBuffer` contents again by using the `rewind()` method to reposition the `ByteBuffer` to the beginning of the buffer.

Figure 13-28 employs all these steps to declare a file and write some `bytes` in it randomly at positions 0, 22, and 12, in that order.

```
import java.nio.file.*;
import java.io.*;
import java.nio.channels.FileChannel;
import java.nio.ByteBuffer;
import static java.nio.file.StandardOpenOption.*;
public class RandomAccessTest
{
   public static void main(String[] args)
   {
      Path file =
         Paths.get("C:\\Java\\Chapter.13\\Numbers.txt");
      String s = "XYZ";
      byte[] data = s.getBytes();
      ByteBuffer out = ByteBuffer.wrap(data);
      FileChannel fc = null;
      try
```

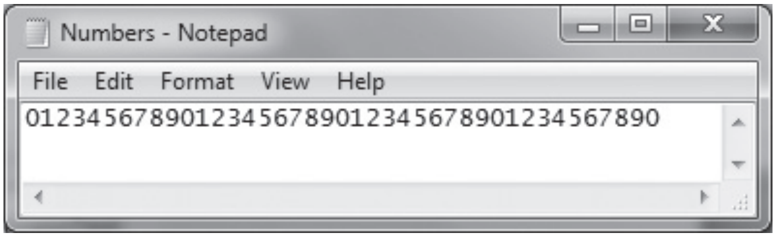
Figure 13-28 The `RandomAccessTest` class *(continues)*

(continued)

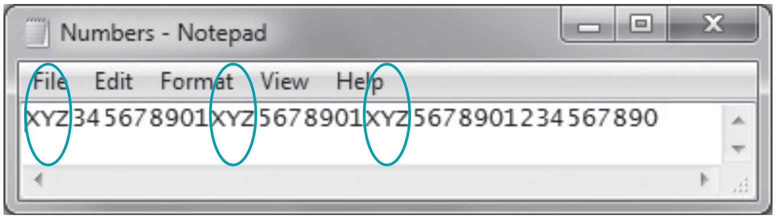
```
         {
            fc = (FileChannel)Files.newByteChannel(file, READ, WRITE);
            fc.position(0);
            while(out.hasRemaining())
               fc.write(out);
            out.rewind();
            fc.position(22);
            while(out.hasRemaining())
               fc.write(out);
            out.rewind();
            fc.position(12);
            while(out.hasRemaining())
               fc.write(out);
            fc.close();
         }
         catch (Exception e)
         {
            System.out.println("Error message: " + e);
         }
      }
   }
```

Figure 13-28 The `RandomAccessTest` class

Figure 13-29 shows the Numbers.txt text file before and after executing the `RandomAccessTest` program in Figure 13-28. The `String` `"XYZ"` has been written at positions 0, 8, and 12.



```
01234567890123456789012345678901234567890
```



```
XYZ345678901XYZ5678901XYZ5678901234567890
```

Figure 13-29 The Numbers.txt file before and after execution of `RandomAccessTest`

TWO TRUTHS & A LIE

Learning About Random Access Files

1. Businesses store data in random order when they use the records for batch processing.
2. Real-time applications are interactive and require using random access data files.
3. A `FileChannel` object is a seekable channel for reading and writing a file.

The false statement is #1. Businesses store data in sequential order when they use the records for batch processing.

Writing Records to a Random Access Data File

Writing characters at random text file locations, as in the `RandomAccessTest` program, is of limited value. When you store records in a file, it is often more useful to be able to access the eighth or 12th record rather than the eighth or 12th byte. In such a case, you multiply each record's size by the position you want to access. For example, if you store records that are 50 bytes long, the first record is at position 0, the second record is at position 50, the third record is at position 100, and so on. In other words, you can access the *n*th record in a `FileChannel` named `fc` using the following statement:

```
fc.position((n - 1) * 50);
```

One approach to writing a random access file is to place records into the file based on a key field. A **key field** is the field in a record that makes the record unique from all others. For example, suppose you want to store employee ID numbers, last names, and pay rates in a random access file. In a file of employees, many records might have the same last name or pay rate, but each record has a unique employee ID number, so that field can act as the key field.

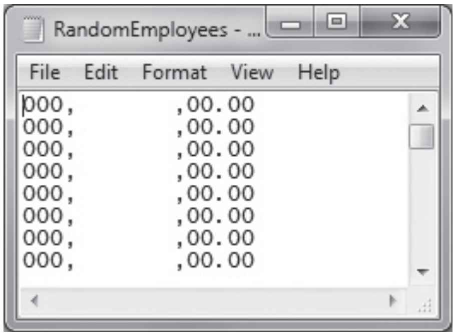
The first step in creating the random access employee file is to create a file that holds default records—for example, using zeroes for the ID numbers and pay rates and blanks for the names. For this example, assume that each employee ID number is three digits; in other words, you cannot have more than 1,000 employees because the ID number cannot surpass 999. Figure 13-30 contains a program that creates 1,000 such records.

```
import java.nio.file.*;
import java.io.*;
import java.nio.ByteBuffer;
import static java.nio.file.StandardOpenOption.*;
public class CreateEmptyEmployeesFile
{
   public static void main(String[] args)
   {
      Path file =
         Paths.get("C:\\Java\\Chapter.13\\RandomEmployees.txt");
      String s = "000,       ,00.00" +
         System.getProperty("line.separator");
      byte[] data = s.getBytes();
      ByteBuffer buffer = ByteBuffer.wrap(data);
      final int NUMRECS = 1000;
      try
      {
         OutputStream output = new
            BufferedOutputStream(Files.newOutputStream(file, CREATE));
         BufferedWriter writer = new
            BufferedWriter(new OutputStreamWriter(output));
         for(int count = 0; count < NUMRECS; ++count)
             writer.write(s, 0, s.length());
         writer.close();
      }
      catch(Exception e)
      {
          System.out.println("Error message: " + e);
      }
   }
}
```

Figure 13-30 The `CreateEmptyEmployeesFile` class

In the first shaded statement in Figure 13-30, a `String` that represents a default record is declared. The three-digit employee number is set to zeros, the name consists of seven blanks, the pay rate is 00.00, and the `String` ends with the system's line separator value. A `byte` array is constructed from the `String` and wrapped into a buffer. Then a file is opened in `CREATE` mode and a `BufferedWriter` is established.

In the last shaded statement in Figure 13-30, a loop executes 1,000 times. Within the loop, the default employee string is passed to the `BufferedWriter` object's `write()` method. Figure 13-31 shows a few records from the created file when it is opened in Notepad.



```
000,          ,00.00
000,          ,00.00
000,          ,00.00
000,          ,00.00
000,          ,00.00
000,          ,00.00
000,          ,00.00
000,          ,00.00
```

Figure 13-31 The RandomEmployees.txt file created by the `CreateEmptyEmployeesFile` program

The default fields in the base random access file don't have to be zeros and blanks. For example, if you wanted 000 to be a legitimate employee ID number or you wanted blanks to represent a correct name, you could use different default values such as 999 and "XXXXXXX". The only requirement is that the default records be recognizable as such.

After you create the base default file, you can replace any of its records with data for an actual employee. You can locate the correct position for the new record by performing arithmetic with the record's key field.

For example, the application in Figure 13-32 creates a single employee record defined in the first shaded statement. The record is for employee 002 with a last name of Newmann and a pay rate of 12.25. In the second shaded statement, the length of this string is assigned to `RECSIZE`. (In this case, `RECSIZE` is 19, which includes one character for each character in the sample record string, including the delimiting commas, plus two bytes for the line separator value returned by the `System.getProperty()` method.) After the `FileChannel` is established, the record is written to the file at the position that begins at two times the record size. The value 2 is hard-coded in this demonstration program because the employee's ID number is 002.

```
import java.nio.file.*;
import java.io.*;
import java.nio.channels.FileChannel;
import java.nio.ByteBuffer;
import static java.nio.file.StandardOpenOption.*;
public class CreateOneRandomAccessRecord
{
   public static void main(String[] args)
   {
      Path file =
         Paths.get("C:\\Java\\Chapter.13\\RandomEmployees.txt");
      String s = "002,Newmann,12.25" +
         System.getProperty("line.separator");
```

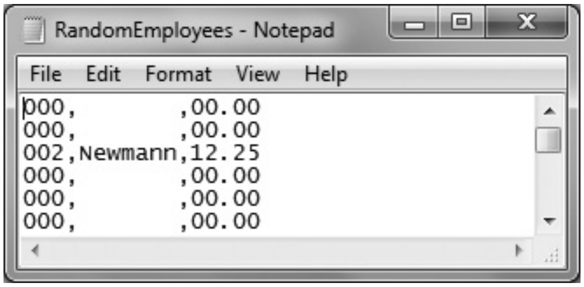
Figure 13-32 The `CreateOneRandomAccessRecord` class *(continues)*

(continued)

```
        final int RECSIZE = s.length();
        byte[] data = s.getBytes();
        ByteBuffer buffer = ByteBuffer.wrap(data);
        FileChannel fc = null;
        try
        {
           fc = (FileChannel)Files.newByteChannel(file, READ, WRITE);
           fc.position(2 * RECSIZE);
           fc.write(buffer);
           fc.close();
        }
        catch (Exception e)
        {
            System.out.println("Error message: " + e);
        }
     }
  }
```

Figure 13-32 The `CreateOneRandomAccessRecord` class

Figure 13-33 shows the RandomEmployees.txt file contents after the `CreateOneRandomAccessRecord` program runs. The employee's data record is correctly placed in the third position in the file. Later, if employees are added that have ID numbers 000 and 001, they can be inserted as the first two records in the file.



```
000,        ,00.00
000,        ,00.00
002,Newmann,12.25
000,        ,00.00
000,        ,00.00
000,        ,00.00
```

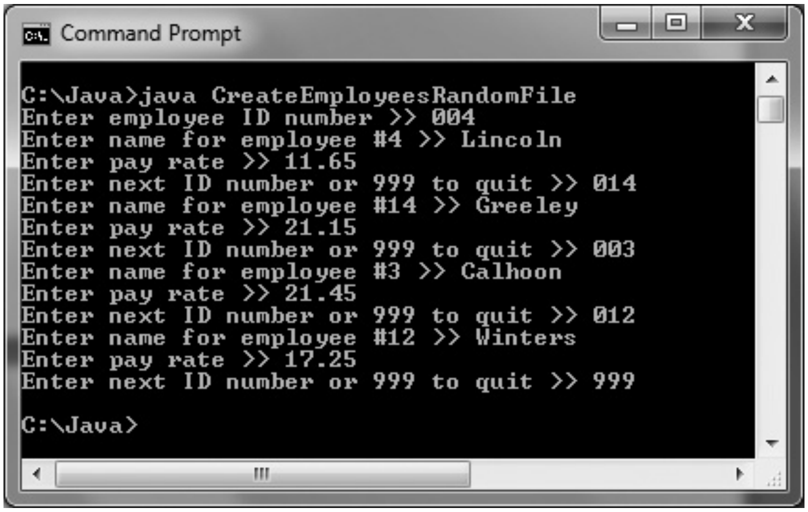
Figure 13-33 The RandomEmployees.txt file after running the `CreateOneRandomAccessRecord` program

A program that inserts one hard-coded employee record into a data file is not very useful. The program in Figure 13-34 accepts any number of records as user input and writes records to a file in a loop. As shown in the first shaded line in the figure, each employee's data value is accepted from the keyboard as a `String` and converted to an integer using the `parseInt()` method. Then, as shown in the second shaded statement, the record's desired position is computed by multiplying the ID number value by the record size.

```
import java.nio.file.*;
import java.io.*;
import java.nio.channels.FileChannel;
import java.nio.ByteBuffer;
import static java.nio.file.StandardOpenOption.*;
import java.util.Scanner;
public class CreateEmployeesRandomFile
{
   public static void main(String[] args)
   {
      Scanner input = new Scanner(System.in);
      Path file =
         Paths.get("C:\\Java\\Chapter.13\\RandomEmployees.txt");
      String s = "000,        ,00.00" +
         System.getProperty("line.separator");
      final int RECSIZE = s.length();
      FileChannel fc = null;
      String delimiter = ",";
      String idString;
      int id;
      String name;
      String payRate;
      final String QUIT = "999";
      try
      {
         fc = (FileChannel)Files.newByteChannel(file, READ, WRITE);
         System.out.print("Enter employee ID number >> ");
         idString = input.nextLine();
         while(!(idString.equals(QUIT)))
         {
            id = Integer.parseInt(idString);
            System.out.print("Enter name for employee #" +
               id + " >> ");
            name = input.nextLine();
            System.out.print("Enter pay rate >> ");
            payRate = input.nextLine();
            s = idString + delimiter + name + delimiter +
               payRate + System.getProperty("line.separator");
            byte[] data = s.getBytes();
            ByteBuffer buffer = ByteBuffer.wrap(data);
            fc.position(id * RECSIZE);
            fc.write(buffer);
            System.out.print("Enter next ID number or " +
               QUIT + " to quit >> ");
            idString = input.nextLine();
         }
         fc.close();
      }
      catch (Exception e)
      {
          System.out.println("Error message: " + e);
      }
   }
}
```

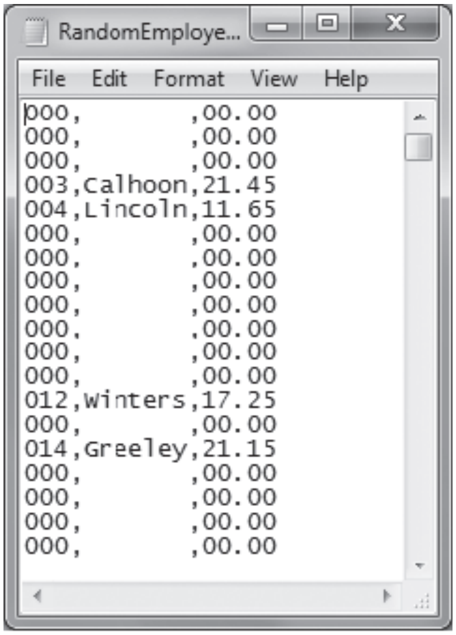
Figure 13-34 The `CreateEmployeesRandomFile` class

Figure 13-35 shows a typical execution of the program, and Figure 13-36 shows the resulting file. (This program was executed after rerunning the `CreateEmptyEmployeesFile` program, so all records started with default values, and the record created by the program shown in Figure 13-33 is not part of the file.) In Figure 13-36, you can see that each employee record is not stored based on the order in which it was entered but is located in the correct spot based on its key field.



```
C:\Java>java CreateEmployeesRandomFile
Enter employee ID number >> 004
Enter name for employee #4 >> Lincoln
Enter pay rate >> 11.65
Enter next ID number or 999 to quit >> 014
Enter name for employee #14 >> Greeley
Enter pay rate >> 21.15
Enter next ID number or 999 to quit >> 003
Enter name for employee #3 >> Calhoon
Enter pay rate >> 21.45
Enter next ID number or 999 to quit >> 012
Enter name for employee #12 >> Winters
Enter pay rate >> 17.25
Enter next ID number or 999 to quit >> 999

C:\Java>
```

Figure 13-35 Typical execution of the `CreateEmployeesRandomFile` program



```
000,        ,00.00
000,        ,00.00
000,        ,00.00
003,Calhoon,21.45
004,Lincoln,11.65
000,        ,00.00
000,        ,00.00
000,        ,00.00
000,        ,00.00
000,        ,00.00
000,        ,00.00
000,        ,00.00
012,Winters,17.25
000,        ,00.00
014,Greeley,21.15
000,        ,00.00
000,        ,00.00
000,        ,00.00
000,        ,00.00
```

Figure 13-36 File created during the execution in Figure 13-35

To keep this example brief and focused on the random access file writing, the `CreateEmployeesFile` application makes several assumptions:

- An employee record contains only an ID number, name, and pay rate. In a real application, each employee would require many more data fields, such as address, phone number, date of hire, and so on.
- Each employee ID number is three digits. In many real applications, ID numbers would be longer to ensure unique values. (Three-digit numbers provide only 1,000 unique combinations.)
- The user will enter valid ID numbers and pay rates. In a real application, this would be a foolhardy assumption because users might type too many digits or type nonnumeric characters. However, to streamline the code and concentrate on the writing of a random access file, error checking for valid ID numbers and pay rates is eliminated from this example.
- The user will not duplicate employee ID numbers. In a real application, a key field should be checked against all existing key fields to ensure that a record is unique before adding it to a file.
- The names entered are all seven characters. This permits each record to be the same size. Only when record sizes are uniform can they be used to arithmetically calculate offset positions. In a real application, you would have to pad shorter names with spaces and truncate longer names to achieve a uniform size.
- Each employee's record is placed in the random access file position that is one less than the employee's ID number. In many real applications, the mathematical computations performed on a key field to determine file placement are more complicated.

TWO TRUTHS & A LIE

Writing Records to a Random Access Data File

1. You can set a `FileChannel`'s reading position based on a key field in a record and the record size.
2. A key field is the field in a record that holds the most sensitive information.
3. A useful technique for creating random access files involves first setting up a file with default records in each position.

The false statement is #2. A key field is the field in a record that makes the record unique from all others.

Reading Records from a Random Access Data File

Just because a file is created as a random access file does not mean it has to be used as one. You can process a random access file either sequentially or randomly.

Accessing a Random Access File Sequentially

The RandomEmployees.txt file created in the previous section contains 1,000 records. However, only four of them contain valuable data. Displaying every record in the file would result in many irrelevant lines of output. It makes more sense to display only those records for which an ID number has been inserted. The application in Figure 13-37 reads through the 1,000-record file sequentially in a `while` loop. The shaded statements check for valid ID numbers. This example assumes that no employee has a valid ID number of 000, so the program displays a record only when the ID is not 000. If 000 could be a valid ID number, then you would want to check for a name that was blank, a pay rate that was 0, or both. Figure 13-38 shows the application's output—a list of the entered records, conveniently in ID number order, which reflects their relative positions within the file.

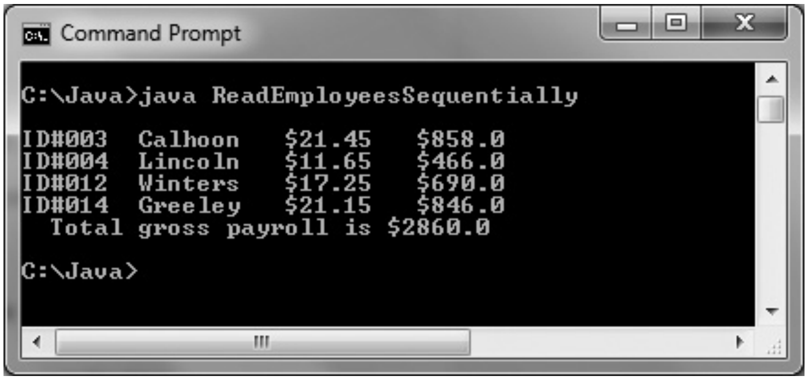
```
import java.nio.file.*;
import java.io.*;
import static java.nio.file.AccessMode.*;
public class ReadEmployeesSequentially
{
   public static void main(String[] args)
   {
      Path file =
         Paths.get("C:\\Java\\Chapter.13\\RandomEmployees.txt");
      String[] array = new String[3];
      String s = "";
      String delimiter = ",";
      int id;
      String stringId;
      String name;
      double payRate;
      double gross;
      final double HRS_IN_WEEK = 40;
      double total = 0;
      try
      {
         InputStream input = new
            BufferedInputStream(Files.newInputStream(file));
         BufferedReader reader = new
            BufferedReader(new InputStreamReader(input));
         System.out.println();
         s = reader.readLine();
```

Figure 13-37 The `ReadEmployeesSequentially` class *(continues)*

(continued)

```
            while(s != null)
            {
               array = s.split(delimiter);
               stringId = array[0];
               id = Integer.parseInt(array[0]);
               if(id !=  0)
               {
                  name = array[1];
                  payRate = Double.parseDouble(array[2]);
                  gross = payRate * HRS_IN_WEEK;
                  System.out.println("ID#" + stringId + "   " +
                     name + "    $" + payRate + "    $" + gross);
                  total += gross;
               }
               s = reader.readLine();
            }
            reader.close();
         }
         catch(Exception e)
         {
            System.out.println("Message: " + e);
         }
         System.out.println("  Total gross payroll is $" + total);
      }
   }
```

Figure 13-37 The `ReadEmployeesSequentially` class



```
C:\Java>java ReadEmployeesSequentially

ID#003  Calhoon   $21.45   $858.0
ID#004  Lincoln   $11.65   $466.0
ID#012  Winters   $17.25   $690.0
ID#014  Greeley   $21.15   $846.0
  Total gross payroll is $2860.0

C:\Java>
```

Figure 13-38 Output of the `ReadEmployeesSequentially` application

Accessing a Random Access File Randomly

If you simply want to display records in order based on their key field, you do not need to create a random access file and waste unneeded storage. Instead, you could sort the records using one of the techniques you learned in Chapter 9. The benefit of using a random access file is the ability to retrieve a specific record from a file directly, without reading through other records to locate the desired one.

The ReadEmployeesRandomly application in Figure 13-39 allows the user to enter an employee's ID number. The application calculates the correct record position in the data file (one less than the ID number) and positions the file pointer at the correct location to begin reading. The user is then prompted for an ID number, which is converted to an integer with the parseInt() method. (To keep this example brief, the application does not check for a valid ID number, so the parseInt() method might throw an exception to the operating system, ending the execution of the application.) In the shaded portion of the application in Figure 13-39, while the user does not enter 999 to quit, the position of the sought-after record is calculated by multiplying the ID number by the record size and then positioning the file pointer at the desired location. (Again, to keep the example short, the ID number is not checked to ensure that it is 999 or less.) The employee record is retrieved from the data file and displayed, and then the user is prompted for the next desired ID number. Figure 13-40 shows a typical execution.

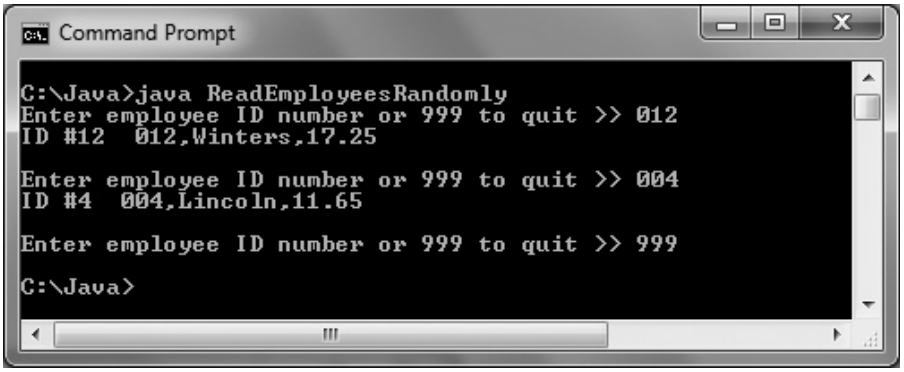
```
import java.nio.file.*;
import java.io.*;
import java.nio.channels.FileChannel;
import java.nio.ByteBuffer;
import static java.nio.file.StandardOpenOption.*;
import java.util.Scanner;
public class ReadEmployeesRandomly
{
   public static void main(String[] args)
   {
      Scanner keyBoard = new Scanner(System.in);
      Path file =
         Paths.get("C:\\Java\\Chapter.13\\RandomEmployees.txt");
      String s = "000,       ,00.00" +
         System.getProperty("line.separator");
      final int RECSIZE = s.length();
      byte[] data = s.getBytes();
      ByteBuffer buffer = ByteBuffer.wrap(data);
      FileChannel fc = null;
      String idString;
      int id;
      final String QUIT = "999";
      try
      {
         fc = (FileChannel)Files.newByteChannel(file, READ, WRITE);
         System.out.print("Enter employee ID number or " +
            QUIT + " to quit >> ");
         idString = keyBoard.nextLine();
```

Figure 13-39 The ReadEmployeesRandomly class *(continues)*

(continued)

```
            while(!idString.equals(QUIT))
            {
               id = Integer.parseInt(idString);
               buffer= ByteBuffer.wrap(data);
               fc.position(id * RECSIZE);
               fc.read(buffer);
               s = new String(data);
               System.out.println("ID #" + id + "   " + s);
               System.out.print("Enter employee ID number or " +
                  QUIT + " to quit >> ");
               idString = keyBoard.nextLine();
            }
            fc.close();
         }
         catch (Exception e)
         {
             System.out.println("Error message: " + e);
         }
      }
   }
```

Figure 13-39 The `ReadEmployeesRandomly` class



```
C:\Java>java ReadEmployeesRandomly
Enter employee ID number or 999 to quit >> 012
ID #12   012,Winters,17.25

Enter employee ID number or 999 to quit >> 004
ID #4   004,Lincoln,11.65

Enter employee ID number or 999 to quit >> 999

C:\Java>
```

Figure 13-40 Typical execution of the `ReadEmployeesRandomly` program

Watch the video *Random Access Data Files.*

TWO TRUTHS & A LIE

Reading Records from a Random Access Data File

1. When a file is created as a random access file, you also must read it randomly.
2. The benefit of using a random access file is the ability to retrieve a specific record from a file directly, without reading through other records to locate the desired one.
3. When you access a record from a random access file, you usually calculate its position based on a key.

The false statement is #1. Just because a file is created as a random access file does not mean it has to be used as one. You can process the file sequentially or randomly.

You Do It

Creating Multiple Random Access Files

In this section, you write a class that prompts the user for customer data and assigns the data to one of two files depending on the customer's state of residence. This program assumes that Wisconsin (WI) records are assigned to an in-state file and that all other records are assigned to an out-of-state file. First you will create empty files to store the records, and then you will write the code that places each record in the correct file.

1. Open a new file in your text editor, and type the following required `import` statements:

```
import java.nio.file.*;
import java.io.*;
import java.nio.channels.FileChannel;
import java.nio.ByteBuffer;
import static java.nio.file.StandardOpenOption.*;
import java.util.Scanner;
import java.text.*;
```

2. Enter the beginning lines of the program, which include a `Scanner` class object to accept user input:

```
public class CreateFilesBasedOnState
{
   public static void main(String[] args)
   {
      Scanner input = new Scanner(System.in);
```

(continues)

(continued)

3. This program uses two `Path` objects to hold records for in-state and out-of-state customers. You can use a different `String` value for your `Paths` based on your `System` and the location where you want to save your files.

```
Path inStateFile =
   Paths.get("C:\\Java\\Chapter.13\\InStateCusts.txt");
Path outOfStateFile =
   Paths.get("C:\\Java\\Chapter.13\\OutOfStateCusts.txt");
```

4. Build a `String` that can be used to format the empty files that are created before any actual customer data is written. Include constants for the format of the account number (three digits), the customer name (10 spaces), the customer's state, and the customer's balance (up to 9999.99). After defining the field delimiter (a comma), you can build a generic customer string by assembling the pieces. The record size is then calculated from the dummy record. A consistent record size is important so it can be used to calculate a record's position when the files are accessed randomly.

```
final String ID_FORMAT = "000";
final String NAME_FORMAT = "          ";
final int NAME_LENGTH = NAME_FORMAT.length();
final String HOME_STATE = "WI";
final String BALANCE_FORMAT = "0000.00";
String delimiter = ",";
String s = ID_FORMAT + delimiter + NAME_FORMAT +
   delimiter + HOME_STATE + delimiter + BALANCE_FORMAT +
   System.getProperty("line.separator");
final int RECSIZE = s.length();
```

5. The last declarations are for two `FileChannel` references; `String` and integer representations of the customer's account number; the customer's `name`, `state`, and `balance` fields; and a `QUIT` constant that identifies the end of data entry.

```
FileChannel fcIn = null;
FileChannel fcOut = null;
String idString;
int id;
String name;
String state;
double balance;
final String QUIT = "999";
```

6. Next, you call a method that creates the empty files into which the randomly placed data records can eventually be written. The method accepts the `Path` for a file and the `String` that defines the record format.

```
createEmptyFile(inStateFile, s);
createEmptyFile(outOfStateFile, s);
```

(continues)

(continued)

7. Add closing curly braces for the `main()` method and the class. Then save the file as **CreateFilesBasedOnState.java** and compile it. Correct any errors before proceeding.

Writing a Method to Create an Empty File

In this section, you write the method that creates empty files using the default record format string. The method will create 1,000 records with an account number of 000.

1. Just before the closing curly brace of the `CreateFilesBasedOnState` class, insert the header and opening brace for a method that will create an empty file to hold random access records. The method accepts a `Path` argument and the default record `String`.

```
public static void createEmptyFile(Path file, String s)
{
```

2. Define a constant for the number of records to be written:

```
    final int NUMRECS = 1000;
```

3. In a `try` block, declare a new `OutputStream` using the method's `Path` parameter. Then create a `BufferedWriter` using the `OutputStream`.

```
    try
    {
       OutputStream outputStr = new
          BufferedOutputStream(Files.newOutputStream(file,CREATE));
       BufferedWriter writer = new BufferedWriter(new
          OutputStreamWriter(outputStr));
```

4. Use a `for` loop to write 1,000 default records using the parameter `String`. Then close the `BufferedWriter`, and add a closing brace for the `try` block.

```
       for(int count = 0; count < NUMRECS; ++count)
          writer.write(s, 0, s.length());
       writer.close();
    }
```

5. Add a `catch` block to handle any `Exception` thrown from the `try` block, and add a closing curly brace for the method.

```
    catch(Exception e)
    {
       System.out.println("Error message: " + e);
    }
}
```

6. Save the file and compile it. Correct any errors.

(continues)

(continued)

Adding Data Entry Capability to the Program

In these steps, you add the code that accepts data from the keyboard and writes it to the correct location (based on the customer's account number) within the correct file (based on the customer's state).

1. After the calls to the `createEmptyFile()` method, but before the method header, start a `try` block that will handle all the data entry and file writing for customer records:

```
try
{
```

2. Set up the `FileChannel` references for both the in-state and out-of-state files.

```
fcIn = (FileChannel)Files.newByteChannel(inStateFile, CREATE, WRITE);
fcOut = (FileChannel)Files.newByteChannel(outOfStateFile, CREATE, WRITE);
```

3. Prompt the user for a customer account number, and accept it from the keyboard. Then start a loop that will continue as long as the user does not enter the `QUIT` value. Next, convert the entered account number to an integer so it can be used to calculate the file position for the entered record. In a full-blown application, you would add code to ensure that the account number is three digits, but to keep this example shorter, this program assumes that the user will enter valid account numbers.

```
System.out.print("Enter customer account number >> ");
idString = input.nextLine();
while(!(idString.equals(QUIT)))
{
   id = Integer.parseInt(idString);
```

4. Prompt the user for and accept the customer's name. To ensure that entered names are stored using a uniform length, assign the name to a `StringBuilder` object, and set the length to the standard length. Then assign the newly sized `StringBuilder` back to the `String`.

```
System.out.print("Enter name for customer >> ");
name = input.nextLine();
StringBuilder sb = new StringBuilder(name);
sb.setLength(NAME_LENGTH);
name = sb.toString();
```

5. Prompt the user for and accept the customer's state of residence. (In a fully developed program, you would check the entered state against a list of valid states, but this step is omitted to keep the program shorter.)

```
System.out.print("Enter state >> ");
state = input.nextLine();
```

(continues)

(continued)

6. Prompt the user for and accept the customer's balance. Because you use the `nextDouble()` method to retrieve the balance, you follow it with a call to `nextLine()` to absorb the Enter key value left in the input stream. Then you can use the `DecimalFormat` class to ensure that the balance meets the format requirements of the file. Because the `BALANCE_FORMAT String`'s value is 0000.00, zeros will be added to the front or back of any `double` that would not otherwise meet the standard. For example, 200.99 will be stored as 0200.99 and 0.1 will be stored as 0001.00. Appendix C contains more information on the `DecimalFormat` class and describes other potential formats.

```
System.out.print("Enter balance >> ");
balance = input.nextDouble();
input.nextLine();
DecimalFormat df = new DecimalFormat(BALANCE_FORMAT);
```

7. Construct the `String` to be written to the file by concatenating the entered fields with the comma delimiter and the line separator.

```
s = idString + delimiter + name + delimiter +
   state + delimiter + df.format(balance)  +
   System.getProperty("line.separator");
```

8. Convert the constructed `String` to an array of bytes, and wrap the array into a `ByteBuffer`.

```
byte data[] = s.getBytes();
ByteBuffer buffer = ByteBuffer.wrap(data);
```

9. Depending on the customer's state, use the in-state or out-of-state `FileChannel`. Position the file pointer to start writing a record in the correct position based on the account number, and write the data `String`.

```
if(state.equals(HOME_STATE))
{
   fcIn.position(id * RECSIZE);
   fcIn.write(buffer);
}
else
{
   fcOut.position(id * RECSIZE);
   fcOut.write(buffer);
}
```

10. Prompt the user for the next customer account number, and add a closing curly brace for the `while` loop.

```
   System.out.print("Enter next customer account number or " +
      QUIT + " to quit >> ");
   idString = input.nextLine();
}
```

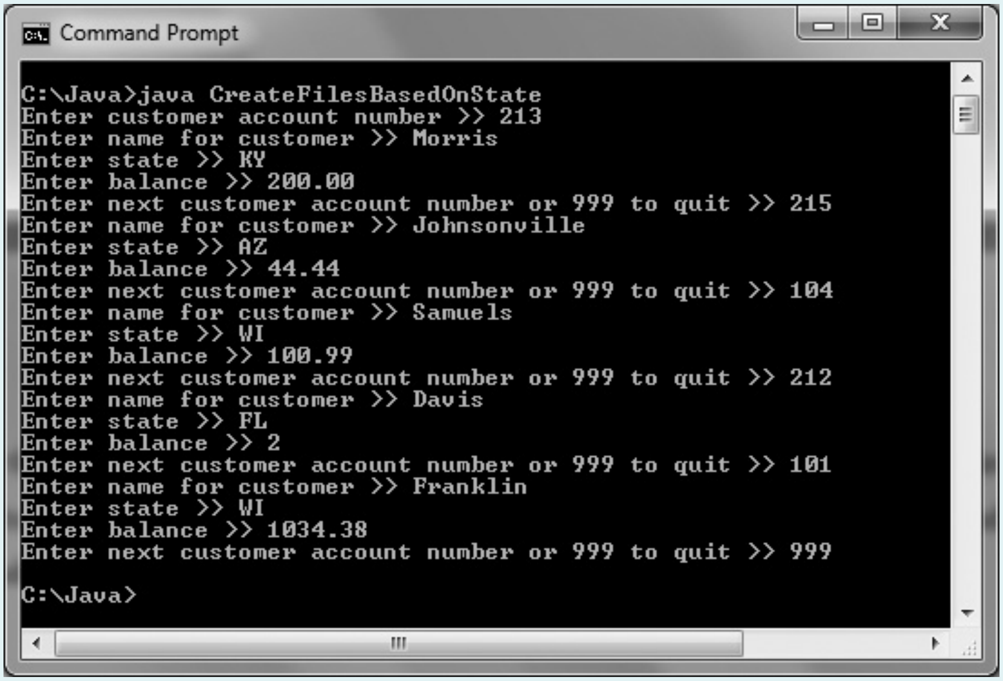
(continues)

(continued)

11. Close the `FileChannels`, and add a closing curly brace for the class.

```
    fcIn.close();
    fcOut.close();
}
```

12. Save the file and compile it. Execute the program, and enter several records. Make sure to include names that are longer and shorter than 10 characters and to include a variety of balance values. Figure 13-41 shows a typical execution.



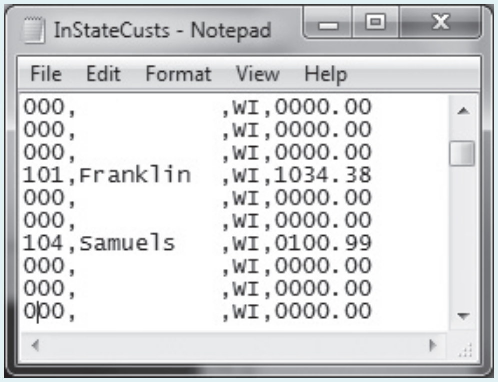
```
C:\Java>java CreateFilesBasedOnState
Enter customer account number >> 213
Enter name for customer >> Morris
Enter state >> KY
Enter balance >> 200.00
Enter next customer account number or 999 to quit >> 215
Enter name for customer >> Johnsonville
Enter state >> AZ
Enter balance >> 44.44
Enter next customer account number or 999 to quit >> 104
Enter name for customer >> Samuels
Enter state >> WI
Enter balance >> 100.99
Enter next customer account number or 999 to quit >> 212
Enter name for customer >> Davis
Enter state >> FL
Enter balance >> 2
Enter next customer account number or 999 to quit >> 101
Enter name for customer >> Franklin
Enter state >> WI
Enter balance >> 1034.38
Enter next customer account number or 999 to quit >> 999

C:\Java>
```

Figure 13-41 Typical execution of the `CreateFilesBasedOnState` program

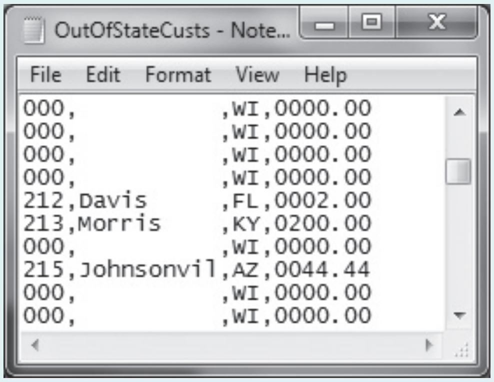
13. Locate and open the **InStateCusts.txt** and **OutOfStateCusts.txt** files. Scroll through the files until you find the records you created. Figure 13-42 shows part of both files that contains the records added using the execution in Figure 13-41. Confirm that each record is placed in the correct file location, that each name and balance is in the correct format, and that the records with a `state` value of "WI" are placed in one file while all the other records are placed in the other file.

(continues)

(continued)



```
000,          ,WI,0000.00
000,          ,WI,0000.00
000,          ,WI,0000.00
101,Franklin  ,WI,1034.38
000,          ,WI,0000.00
000,          ,WI,0000.00
104,Samuels   ,WI,0100.99
000,          ,WI,0000.00
000,          ,WI,0000.00
000,          ,WI,0000.00
```



```
000,          ,WI,0000.00
000,          ,WI,0000.00
000,          ,WI,0000.00
000,          ,WI,0000.00
212,Davis     ,FL,0002.00
213,Morris    ,KY,0200.00
000,          ,WI,0000.00
215,Johnsonvil,AZ,0044.44
000,          ,WI,0000.00
000,          ,WI,0000.00
```

Figure 13-42 Contents of the files created by the program execution in Figure 13-41

Setting Up a Program to Read the Created Files

Now, you can write a program that can use either of the files you just created. The program has four parts:

- The program will prompt the user to enter the filename to be used and set up all necessary variables and constants.
- A few statistics about the file will be displayed.
- The nondefault contents of the file will be displayed sequentially.
- A selected record from the file will be accessed directly.

1. Open a new file in your text editor. Enter all the required import statements and the class header for the `ReadStateFile` application.

```
import java.nio.file.*;
import java.io.*;
import java.nio.file.attribute.*;
import static java.nio.file.StandardOpenOption.*;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;
import java.util.Scanner;
public class ReadStateFile
{
```

2. Declare a `Scanner` object to handle keyboard input. Then declare a `String` that will hold the name of the file the program will use. Prompt the user for the filename, concatenate it with the correct path, and create a `Path` object.

(continues)

(continued)

```
Scanner kb = new Scanner(System.in);
String fileName;
System.out.print("Enter name of file to use >> ");
fileName = kb.nextLine();
fileName = "C:\\Java\\Chapter.13\\" + fileName;
Path file = Paths.get(fileName);
```

3. Add the `String` formatting constants and build a sample record `String` so that you can determine the record size. To save time, you can copy these declarations from the `CreateFilesBasedOnState` program.

```
final String ID_FORMAT = "000";
final String NAME_FORMAT = "          ";
final int NAME_LENGTH = NAME_FORMAT.length();
final String HOME_STATE = "WI";
final String BALANCE_FORMAT = "0000.00";
String delimiter = ",";
String s = ID_FORMAT + delimiter + NAME_FORMAT + delimiter +
   HOME_STATE + delimiter + BALANCE_FORMAT +
   System.getProperty("line.separator");
final int RECSIZE = s.length();
```

4. The last set of declarations includes a byte array that you will use with a `ByteBuffer` later in the program, a `String` that represents the account number in an empty account, and an array of strings that can hold the pieces of a split record after it is read from the input file. Add a variable for the numeric customer balance, which will be converted from the `String` stored in the file. Also, declare a total and initialize it to 0 so the total customer balance due value can be accumulated.

```
byte data[] = s.getBytes();
final String EMPTY_ACCT = "000";
String[] array = new String[4];
double balance;
double total = 0;
```

5. Add two closing curly braces for the method and the class. Save the file as **ReadStateFile.java**. Compile the file and correct any errors.

Displaying File Statistics

In the next section of the program, you display the creation time and size of the file.

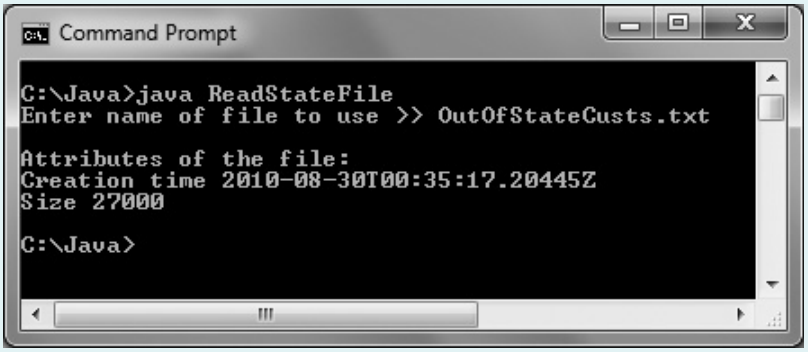
1. Just before the two closing curly braces you just added to the program, insert a `try` block in which you declare a `BasicFileAttributes` object. Then add statements to display the file's creation time and size. Include a `catch` block to handle any thrown exceptions.

(continues)

(continued)

```
try
{
   BasicFileAttributes attr =
      Files.readAttributes(file, BasicFileAttributes.class);
   System.out.println("\nAttributes of the file:");
   System.out.println("Creation time " + attr.creationTime());
   System.out.println("Size " + attr.size());
}
catch(IOException e)
{
   System.out.println("IO Exception");
}
```

2. Save the file, then compile and execute it. When prompted, you can type the name of either the **InStateCusts.txt** file or the **OutOfStateCusts.txt** file. Figure 13-43 shows a typical execution.



```
C:\Java>java ReadStateFile
Enter name of file to use >> OutOfStateCusts.txt

Attributes of the file:
Creation time 2010-08-30T00:35:17.20445Z
Size 27000

C:\Java>
```

Figure 13-43 Typical execution of the `ReadStateFile` program

Reading a File Sequentially

In these steps, you display all the entered records in the file of the user's choice.

1. Start a new `try…catch` pair after the first one ends but before the two closing curly braces in the program. Declare an `InputStream` and `BufferedReader` to handle reading the file.

```
try
{
   InputStream iStream = new
     BufferedInputStream(Files.newInputStream(file));
   BufferedReader reader = new
     BufferedReader(new InputStreamReader(iStream));
```

(continues)

(continued)

2. Display a heading, and then read the first record from the file into a `String`.

```
System.out.println("\nAll non-default records:\n");
s = reader.readLine();
```

3. In a loop that continues while there is more data to read, split the `String` using the comma delimiter. Test the first split element, the account number, and proceed only if it is not "000". If the record was entered in the previous program, display the split `String` elements. Add the balance to a running total. As the last action in the loop, read the next record.

```
while(s != null)
{
   array = s.split(delimiter);
   if(!array[0].equals(EMPTY_ACCT))
   {
       balance = Double.parseDouble(array[3]);
       System.out.println("ID #" + array[0] + "  " +
          array[1] + array[2] + "  $" + array[3]);
       total += balance;
   }
  s = reader.readLine();
}
```

4. After all the records have been processed, display the total and close the reader. Add a closing curly brace for the `try` block.

```
   System.out.println("Total of all balances is $" + total);
   reader.close();
}
```

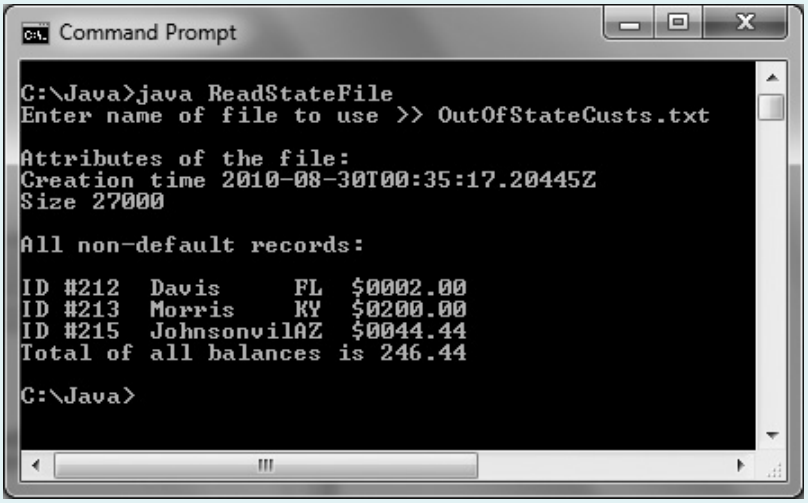
5. Create a `catch` block to handle any thrown exceptions.

```
catch(Exception e)
{
  System.out.println("Message: " + e);
}
```

6. Save the file, and then compile and execute it. Enter the name of either data file. Figure 13-44 shows a typical execution. After the file statistics are displayed, the records added earlier are also displayed, and the balances are accumulated.

(continues)

(continued)



```
C:\Java>java ReadStateFile
Enter name of file to use >> OutOfStateCusts.txt

Attributes of the file:
Creation time 2010-08-30T00:35:17.20445Z
Size 27000

All non-default records:

ID #212  Davis     FL  $0002.00
ID #213  Morris    KY  $0200.00
ID #215  JohnsonvilAZ  $0044.44
Total of all balances is 246.44

C:\Java>
```

Figure 13-44 Typical execution of the `ReadStateFile` program after code has been added to display records

Reading a File Randomly

In the last part of the program, you ask the user to enter an account number and then display the record directly by repositioning the file pointer.

1. After the closing brace of the last `catch` block, but before the two final closing braces in the class, add a new `try` block that declares a `FileChannel` and `ByteBuffer` and then prompts the user for and accepts an account to search for in the file.

```
try
{
   FileChannel fc = (FileChannel)Files.newByteChannel(file, READ);
   ByteBuffer buffer = ByteBuffer.wrap(data);
   int findAcct;
   System.out.print("\nEnter account to seek >> ");
   findAcct = kb.nextInt();
```

2. Calculate the position of the sought-after record in the file by multiplying the record number by the file size. Read the selected record into the `ByteBuffer`, and convert the associated `byte` array to a `String` that you can display. Add a closing curly brace for the `try` block.

```
   fc.position(findAcct * RECSIZE);
   fc.read(buffer);
   s = new String(data);
   System.out.println("Desired record: " + s);
}
```

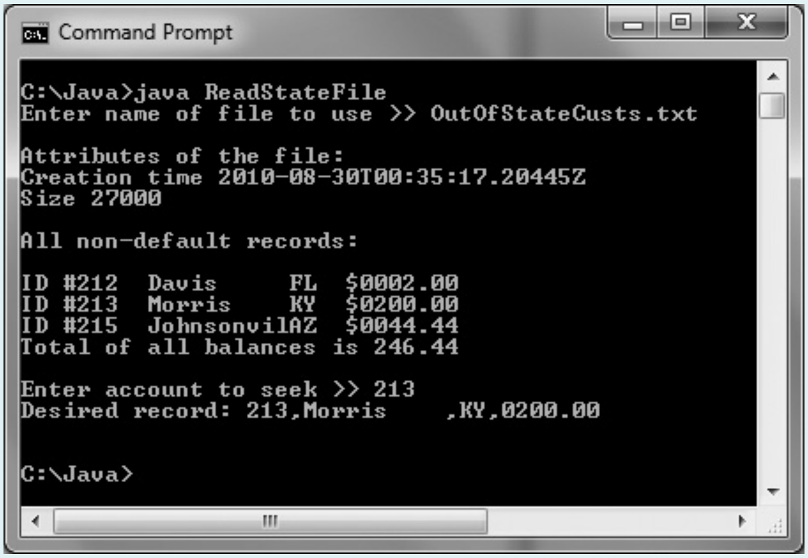
(continues)

(continued)

3. Add a `catch` block to handle any exceptions.

```
catch(Exception e)
{
   System.out.println("Message: " + e);
}
```

4. Save the file, and then compile and execute it. Figure 13-45 shows a typical execution. First, the file attributes are displayed, then all the records are displayed, and then a record selected by the user is displayed.



```
C:\Java>java ReadStateFile
Enter name of file to use >> OutOfStateCusts.txt

Attributes of the file:
Creation time 2010-08-30T00:35:17.20445Z
Size 27000

All non-default records:

ID #212  Davis     FL  $0002.00
ID #213  Morris    KY  $0200.00
ID #215  JohnsonvilAZ  $0044.44
Total of all balances is 246.44

Enter account to seek >> 213
Desired record: 213,Morris     ,KY,0200.00

C:\Java>
```

Figure 13-45 Typical execution of the `ReadStateFile` program after code has been completed

Don't Do It

- Don't forget that a `Path` name might be relative and that you might need to make the `Path` absolute before accessing it.
- Don't forget that the backslash character starts the escape sequence in Java, so you must use two backslashes in a string that describes a `Path` in the DOS operating system.

Key Terms

Volatile storage is temporary storage that is lost when a computer loses power.

Random access memory (RAM) is the temporary storage within a computer.

Nonvolatile storage is permanent storage; it is not lost when a computer loses power.

A **computer file** is a collection of data stored on a nonvolatile device in a computer system.

Permanent storage devices, such as hard disks, Zip disks, USB drives, reels or cassettes of magnetic tape, and compact discs, are nonvolatile and hold files.

Text files contain data that can be read in a text editor because the data has been encoded using a scheme such as ASCII or Unicode.

Data files contain facts and figures, such as a payroll file that contains employee numbers, names, and salaries.

Program files or **application files** store software instructions.

Binary files contain data that has not been encoded as text; their contents are in binary format.

The **root directory** of a storage device is the main directory.

Folders or **directories** are used to organize stored files.

A **path** is the complete list of the disk drive plus the hierarchy of directories in which a file resides.

A **path delimiter** is the character used to separate path components.

You can use Java's **`Path` class** to create objects that contain information about files and directories, such as their locations, sizes, creation dates, and whether they even exist.

You use the **`Files` class** to perform operations on files and directories, such as deleting them, determining their attributes, and creating input and output streams.

Factory methods are methods that assist in object creation.

An **absolute path** is a complete path; it does not need any other information to locate a file on a system.

A **relative path** is one that depends on other path information.

Java's **`static import` feature** allows you to use `static` constants without their class name.

A **TOCTTOU bug** is an error that occurs when changes take place from Time Of Check To Time Of Use.

A **character** can be any letter, number, or other special symbol (such as a punctuation mark) that constitutes data.

A **field** is a group of characters that has some meaning.

A **record** is a collection of fields that contain data about an entity.

In a **sequential access file**, each record is accessed one after another in the order in which it was stored.

Comma-separated values (CSV) are fields that are separated by a comma.

To **open a file** is to create an object and associate a stream of bytes with it.

To **close the file** is to make it no longer available to an application.

A **stream** is a data pipeline or channel.

A **buffer** is a memory location where bytes are held after they are logically output but before they are sent to the output device.

Flushing clears any bytes that have been sent to a buffer for output but that have not yet been output to a hardware device.

Batch processing involves performing the same tasks with many records, one after the other.

Real-time applications require that a record be accessed immediately while a client is waiting.

In an **interactive program**, a user makes direct requests to a system.

Random access files are files in which records can be retrieved directly in any order.

Direct access files and **instant access files** are alternate names for *random access files*.

A **file channel** object is an avenue for reading and writing a file.

Seekable describes a file channel in which you can search for a specific file location and in which operations can start at any specified position.

To be **wrapped** is to be encompassed in another type.

A **key field** is the field in a record that makes the record unique from all others.

Chapter Summary

- Data items can be stored on two broad types of storage devices—temporary, volatile storage, or permanent, nonvolatile storage. A computer file is a collection of data stored on a nonvolatile device. Files can be text files or binary files, but all files share characteristics, such as a size, name, and time of creation.
- Java's `Path` class is used to gather file information, such as its location, size, and creation date. You can use the `Files` class to perform operations on files and directories, such as deleting them, determining their attributes, and creating input and output streams.

- Businesses organize data in a hierarchy of character, field, record, and file. When a program performs input and output operations, bytes flow into a program stream, which functions as a pipeline or channel. A buffer is a memory location where bytes are held after they are logically output but before they are sent to the output device. Using a buffer to accumulate input or output improves program performance. Flushing clears any bytes that have been sent to a buffer for output but that have not yet been output to a hardware device.
- `InputStream`, `OutputStream`, and `Reader` are subclasses of the `Object` class that are used for input and output. Output devices can be assigned to `OutputStream` references, allowing applications to save data to them. You can create a file and write to it by using the `Files` class `newOutputStream()` method. To open a file for reading, you can use the `newInputStream()` method.
- The `BufferedWriter` class contains `write()` methods that are used to create data files. Files can be read using the `BufferedReader` class. The `String` class `split()` method accepts an argument that identifies a field delimiter and returns an array of `Strings` in which each array element holds one field.
- Businesses store data in sequential order when they use the records for batch processing. Real-time applications require interactive processing with random access files. Java's `FileChannel` class creates random access files. A file channel is seekable, meaning you can search for a specific file location and operations can start at any specified position.
- One approach to writing a random file is to place records into the file based on a key field that makes a record unique from all others. The first step in creating the random access file is to create a file that holds default records. Then you can replace any default record with actual data by setting the file channel position.
- You can process a random access file either sequentially or randomly. The benefit of using a random access file is the ability to retrieve a specific record from a file directly, without reading through other records to locate the desired one.

Review Questions

1. Which of the following statements is true?
   a. Volatile storage lasts only a few seconds.
   b. Volatile storage is lost when a computer loses power.
   c. Computer disks are volatile storage devices.
   d. All of the above are true.

2. A collection of data stored on a nonvolatile device in a computer system is ____________.
   a. a file
   b. an application
   c. volatile
   d. a type of binary file

3. A complete list of the disk drive plus the hierarchy of directories in which a file resides is its ________.

   a. directory
   b. folder
   c. delimiter
   d. path

4. Which of the following statements creates a `Path` named `p` to a `FileStream` named `f`?

   a. `Path p = new Path("C:\\Java\\MyFile.txt");`
   b. `Path p = f("C:\\Java\\MyFile.txt");`
   c. `Path p = f.getPath("C:\\Java\\MyFile.txt");`
   d. `Path p = getPath(new f("C:\\Java\\MyFile.txt"));`

5. A path that needs no other information to locate a file on a system is ________.

   a. an absolute path
   b. a relative path
   c. a final path
   d. a constant path

6. The `Path` class `getFileName()` method returns ________.

   a. the `String` representation of a `Path`
   b. an absolute `Path`
   c. the first item in a `Path`'s list of name elements
   d. the last item in a `Path`'s list of name elements

7. Which of the following statements always returns the same value as `Files.exists(file)`?

   a. `file.checkAccess()`
   b. `file.checkAccess(EXISTS)`
   c. `file.checkAccess(READ, WRITE)`
   d. `file.checkAccess(file.exists())`

8. You cannot delete a `Path` ________.

   a. under any circumstances
   b. if it represents a directory
   c. if it represents a directory that is not empty
   d. if it represents more than five levels

9. The data hierarchy occurs in the following order from the smallest to largest piece of data: ________.

   a. character, field, record, file
   b. character, file, record, field
   c. character, record, field, file
   d. record, character, field, file

10. When records are accessed one after the other in the order in which they were stored, their file is being used as a ________ access file.

   a. random
   b. binary
   c. chronological
   d. sequential

11. If you fail to close an output file, ____________.
    a. there are usually no serious consequences
    b. you might lose access to the written data
    c. Java will close it for you automatically
    d. Two of the above are correct.

12. Which of the following is true of streams?
    a. Streams are channels through which bytes flow.
    b. Streams always flow in two directions.
    c. Only one stream can be open in a program at a time.
    d. All of the above are true.

13. A buffer ____________.
    a. holds bytes that are scheduled for input or output
    b. deteriorates program performance
    c. cannot be flushed in Java
    d. All of the above are true.

14. `InputStream` is ____________.
    a. a child of `OutputStream`
    b. an abstract class
    c. used for screen output as opposed to file output
    d. All of the above are true.

15. Java's `print()` and `println()` methods are defined in the ____________ class.
    a. `BufferedOutputStream`
    b. `System`
    c. `PrintStream`
    d. `Print`

16. The `newOutputStream()` method ____________.
    a. is defined in the `Files` class
    b. creates a file if it does not already exist
    c. opens a file for writing
    d. All of the above are true.

17. Which of the following does the same thing as the `BufferedWriter` class `newLine()` method?
    a. `System.getProperty("line.separator ")`
    b. `Path.getProperty("line.separator ")`
    c. `System.out.println()`
    d. `System.out.print("\n")`

18. Which of the following systems is most likely to use batch processing?
    a. an airline reservation system
    b. payroll
    c. point-of-sale credit checking
    d. an e-mail application

19. Real-time applications ___________.
    a. use sequential access files
    b. use batch processing
    c. use random access files
    d. seldom are interactive

20. A file channel ___________.
    a. can be read from
    b. can be written to
    c. is seekable
    d. All of the above are true.

Exercises

Programming Exercises

1. Create a file using any word-processing program or text editor. Write an application that displays the file's name, containing folder, size, and time of last modification. Save the file as **FileStatistics.java**.
2. Create two files using any word-processing program or text editor. Write an application that determines whether the two files are located in the same folder. Save the file as **SameFolder.java**.
3. Create a file that contains your favorite movie quote. Use a text editor such as Notepad, and save the file as **quote.txt**. Copy the file contents, and paste them into a word-processing program such as Word. Save the file as **quote.docx**. Write an application that displays the sizes of the two files as well as the ratio of their sizes to each other. Save the file as **FileStatistics2.java**.
4. Write an application that determines which, if any, of the following files are stored in the folder where you have saved the exercises created in this chapter: autoexec.bat, SameFolder.java, FileStatistics.class, and Hello.doc. Save the file as **FindSelectedFiles.java**.
5. a. Create a program that accepts a series of employee ID numbers, first names, and last names from the keyboard and saves the data to a file. Save the program as **WriteEmployeeList.java**. When you execute the program, be sure to enter multiple records that have the same first name because you will search for repeated first names in part d of this exercise.
   b. Write an application that reads the file created by the `WriteEmployeeList` application and displays the records. Save the file as **DisplaySavedEmployeeList.java**.

c. Write an application that allows you to enter any ID number and displays the first and last name for the record stored in the employee file with the given ID number. Display an appropriate message if the ID number cannot be found in the input file. Save the file as **DisplaySelectedIDNumbers.java**.

d. Write an application that allows you to enter any first name and displays all the ID numbers and last names for any records stored in the employee file with the given first name. Display an appropriate message if the first name cannot be found in the input file. Save the file as **DisplaySelectedFirstNames.java**.

6. Using a text editor, create a file that contains a list of at least 10 six-digit account numbers. Read in each account number and display whether it is valid. An account number is valid only if the last digit is equal to the sum of the first five digits divided by 10. For example, the number 223355 is valid because the sum of the first five digits is 15, the remainder when 15 is divided by 10 is 5, and the last digit is 5. Write only valid account numbers to an output file, each on its own line. Save the application as **ValidateCheckDigits.java**.

7. a. Write an application that allows a user to enter a filename and an integer representing a file position. Assume that the file is in the same folder as your executing program. Access the requested position within the file, and display the next 20 characters there. Save the file as **SeekPosition.java**.

   b. Modify the `SeekPosition` application so that instead of displaying 20 characters, the user enters the number of characters to display, beginning with the requested position. Save the file as **SeekPosition2.java**.

8. a. Create an application that allows you to enter salesperson data that consists of an ID number, first name, last name, and current month sales in whole dollars. Depending on whether the salesperson's sales value exceeds $1,000, output each record either to a high-performers file or a low-performers file. Save the program as **HighAndLowSales.java**.

   b. Create an application that displays each record in the two files created in the `HighAndLowSales` application in Exercise 8a. Display a heading to introduce the list produced from each file. For each record, display the ID number, first name, last name, sales value, and the amount by which the sales value exceeds or falls short of the $1,000 cutoff. Save the program as **HighAndLowSalesDisplay.java**.

9. a. The Rochester Bank maintains customer records in a random access file. Write an application that creates 10,000 blank records and then allows the user to enter customer account information, including an account number that is 9999 or less, a last name, and a balance. Insert each new record into a data file at a location that is equal to the account number. Assume that the user will not enter invalid account numbers. Force each name to eight characters, padding

it with spaces or truncating it if necessary. Also assume that the user will not enter a bank balance greater than 99,000.00. Save the file as **CreateBankFile.java**.

b. Create an application that uses the file created by the user in Exercise 9a and displays all existing accounts in account-number order. Save the file as **ReadBankAccountsSequentially.java**.

c. Create an application that uses the file created by the user in Exercise 9a and allows the user to enter an account number to view the account balance. Allow the user to view additional account balances until entering an application-terminating value. Save the file as **ReadBankAccountsRandomly.java**.

10. a. Write a program that allows you to create a file of customers for a company. The first part of the program should create an empty file suitable for writing a three-digit ID number, six-character last name, and five-digit zip code for each customer. The second half of the program accepts user input to populate the file. For this exercise, assume that the user will correctly enter ID numbers and zip codes, but force the customer name to seven characters if it is too long or too short. Issue an error message, and do not save the records if the user tries to save a record with an ID number that has already been used. Save the program as **CreateCustomerFile.java**.

b. Write a program that creates a file of items carried by the company. Include a three-digit item number and up to a 20-character description for each item. Issue an error message if the user tries to store an item number that has already been used. Save the program as **CreateItemFile.java**.

c. Write an application that takes customer orders. Allow a user to enter a customer number and item ordered. Display an error message if the customer number does not exist in the customer file or the item does not exist in the item file; otherwise, display all the customer information and item information. Save the program as **CustomerItemOrder.java**.

Debugging Exercises

1. Each of the following files in the Chapter13 folder of your downloadable student files has syntax and/or logic errors. In each case, determine the problem and fix the program. After you correct the errors, save each file using the same filename preceded with *Fix*. For example, DebugThirteen1.java will become FixDebugThirteen1.java.

   a. DebugThirteen1.java
   b. DebugThirteen2.java
   c. DebugThirteen3.java
   d. DebugThirteen4.java

   The Chapter13 folder contains four additional data files named DebugData1.txt, DebugData2.txt, DebugData3.txt, and DebugData4.txt. These files are used by the Debug programs.

Game Zone

1. In several Game Zone assignments earlier in this book, you created games similar to Hangman in which the user guesses a secret phrase by selecting a series of letters. These versions had limited appeal because each contained only a few possible phrases to guess; after playing the games a few times, the user would have memorized all the phrases. Now create a version in which any number of secret phrases can be saved to a file before the game is played. Use a text editor such as Notepad to type any number of phrases into a file, one per line. Save the file as **Phrases.txt**. Then, create a game that randomly selects a phrase from the file and allows a user to guess the phrase letter by letter. Save the game as **SecretPhraseUsingFile.java**.

2. In Chapter 8, you created a game named `Quiz` in which the user could respond to multiple-choice questions. Modify the game so that it stores the player's highest score from any previous game in a file and displays the high score at the start of each new game. (The first time you play the game, the high score is 0.) Save the game as **QuizUsingFile.java**.

3. Use a text editor to create a comma-delimited file of user IDs and passwords. Revise any one of the games you have created throughout this book so the user must first enter a correct ID and its associated password before playing. Save the program as **GameWithPassword.java**.

Case Problems

1. a. In Chapter 12, you created an interactive `StaffDinnerEvent` class that obtains all the data for a dinner event for Carly's Catering, including details about the event and all the staff members required to work at the event. Now, modify the program to prompt the user for data for three dinner events and to create a data file that contains each event number, event type code, number of guests, and price. Save the program as **StaffDinnerEventAndCreateFile.java**.

   b. Write a program that displays the data saved in the file created in part 1a. Save the program as **DisplayDinnerEventFile.java**.

2. a. In Chapter 12, you created an interactive `RentalDemo` class that obtains all the data for four rentals from Sammy's Seashore Rentals, including details about the contract number, length of the rental, and equipment type. Now, modify the program to create a data file that contains each contract number, rental time in hours and minutes, equipment type code and name, and price. Save the program as **RentalDemoAndCreateFile.java**.

   b. Write a program that displays the data saved in the file created in part 2a. Save the program as **DisplayRentalFile.java**.