Chapter 13:
File Input and Output

Objectives

- Learn about computer files
- Use the `Path` and `Files` class
- 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

- Volatile storage
  - Computer memory or random access memory (RAM)
  - Temporary
- Nonvolatile storage
  - Not lost when computer loses power
  - Permanent
- Computer file
  - Collection of information stored on nonvolatile device in computer system

Understanding Computer Files (cont'd.)

- Permanent storage devices
  - Hard disks
  - Zip disks
  - USB drives
  - Reels or cassettes of magnetic tape
  - Compact discs
- Categories of files by the way they store data
  - **Text files**
  - **Binary files**

Understanding Computer Files (cont'd.)

- **Data files**
  - Contain facts and figures
- **Program files** or **application files**
  - Store software instructions
- **Root directory**
- **Folders** or **directories**
- **Path**
  - A complete list of disk drive plus the hierarchy of directories in which a file resides

Understanding Computer Files (cont'd.)

- When you work with stored files in an application, you perform the following tasks:
  - Determine whether and where a path or file exists
  - Open a file
  - Write information to a file
  - Read data from a file
  - Close a file
  - Delete a file

Using the `Path` and `Files` Classes

- **`Path` class**
  - Use it to create objects that contain information about files or directories
- **`Files` class**
  - Use it to perform operations on files and directories
- `java.nio.file` package
  - Include it to use both the `Path` and `Files` classes

Creating a Path

- First, determine the default file system on the host computer

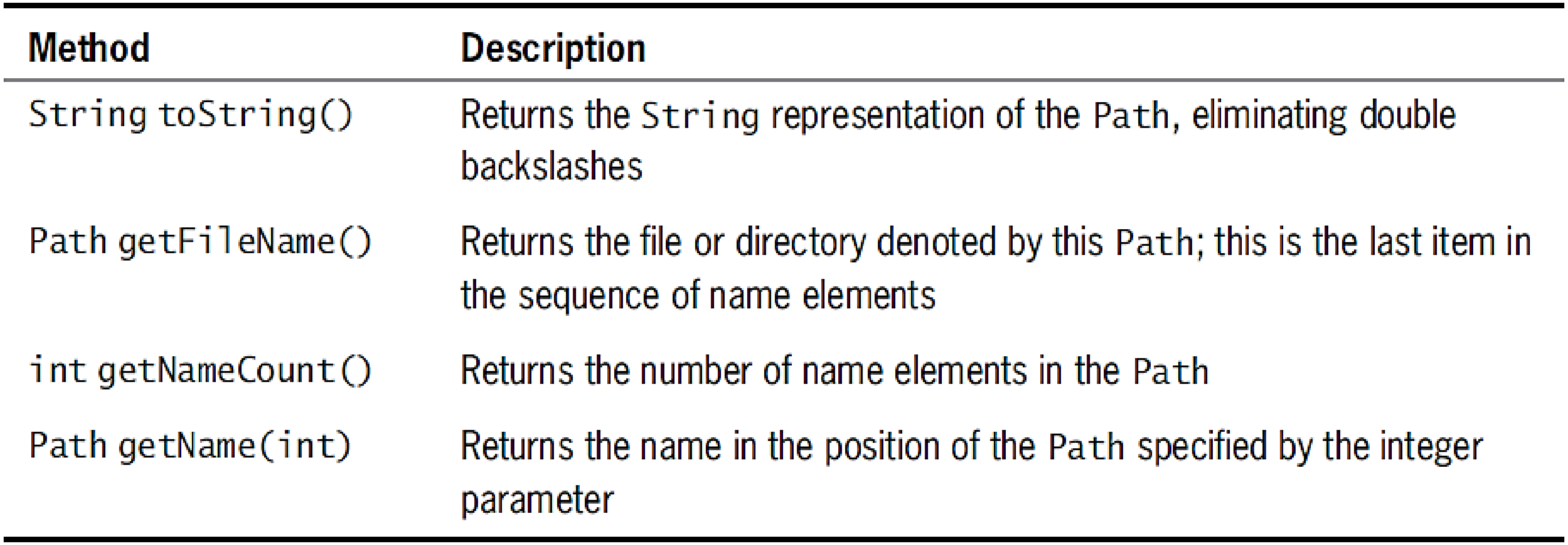
```
FileSystem fs = FileSystems.getDefault();
```

- Define a `Path` using the `getPath()` method

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

- Every `Path` is either an **absolute path** or a **relative path**

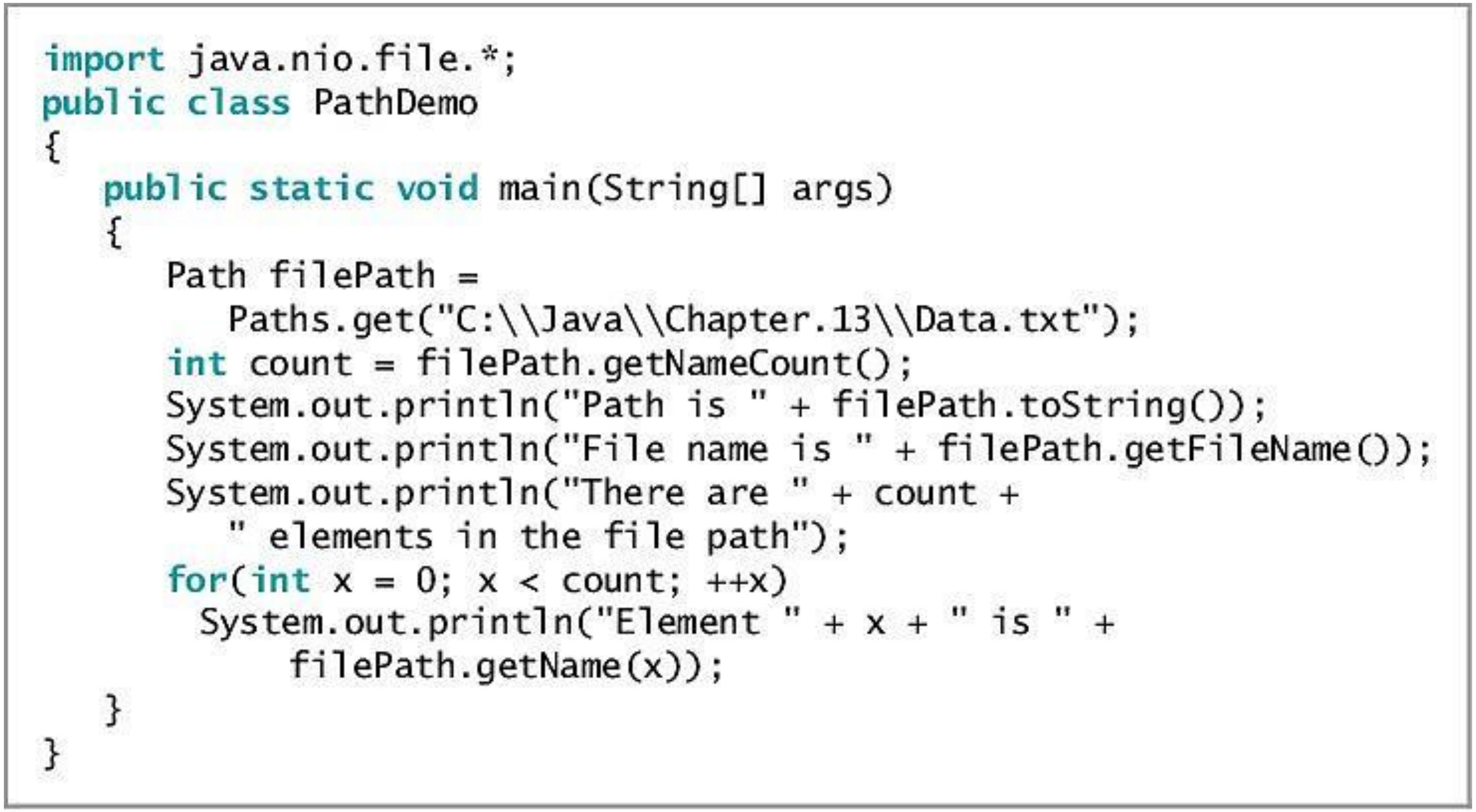
Retrieving Information About a Path



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

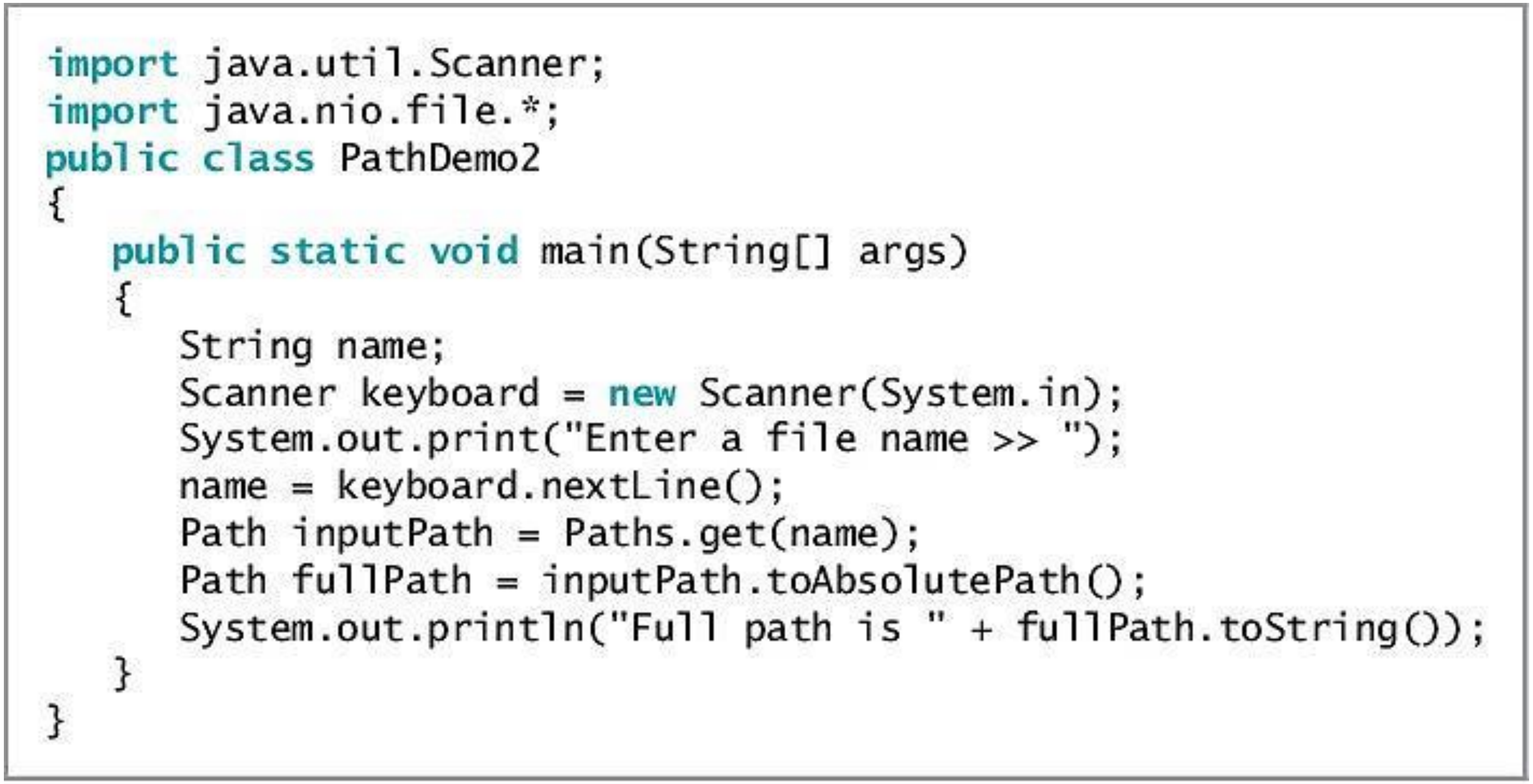
Retrieving Information About a Path (cont'd.)



```
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

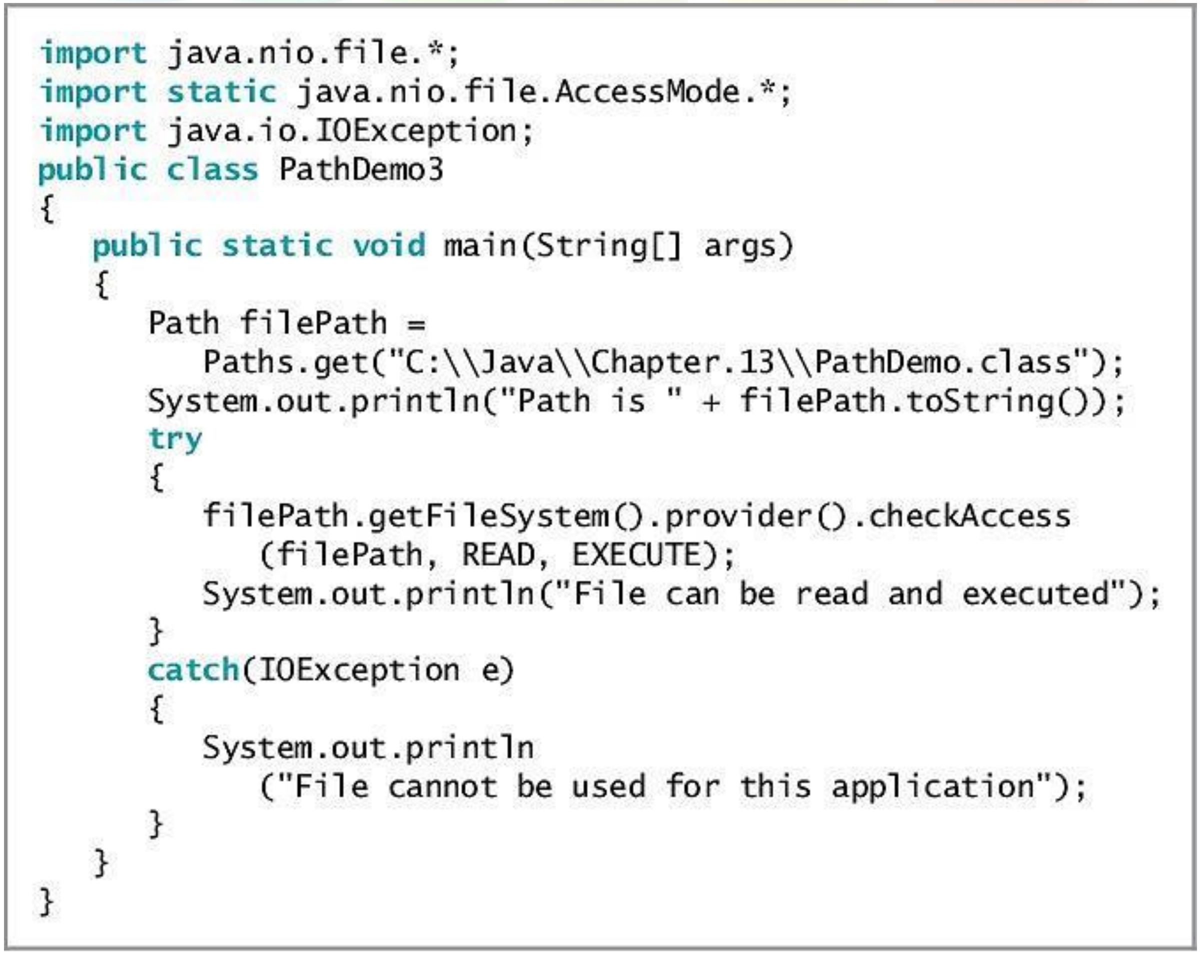
Converting a Relative Path to an Absolute One



```
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

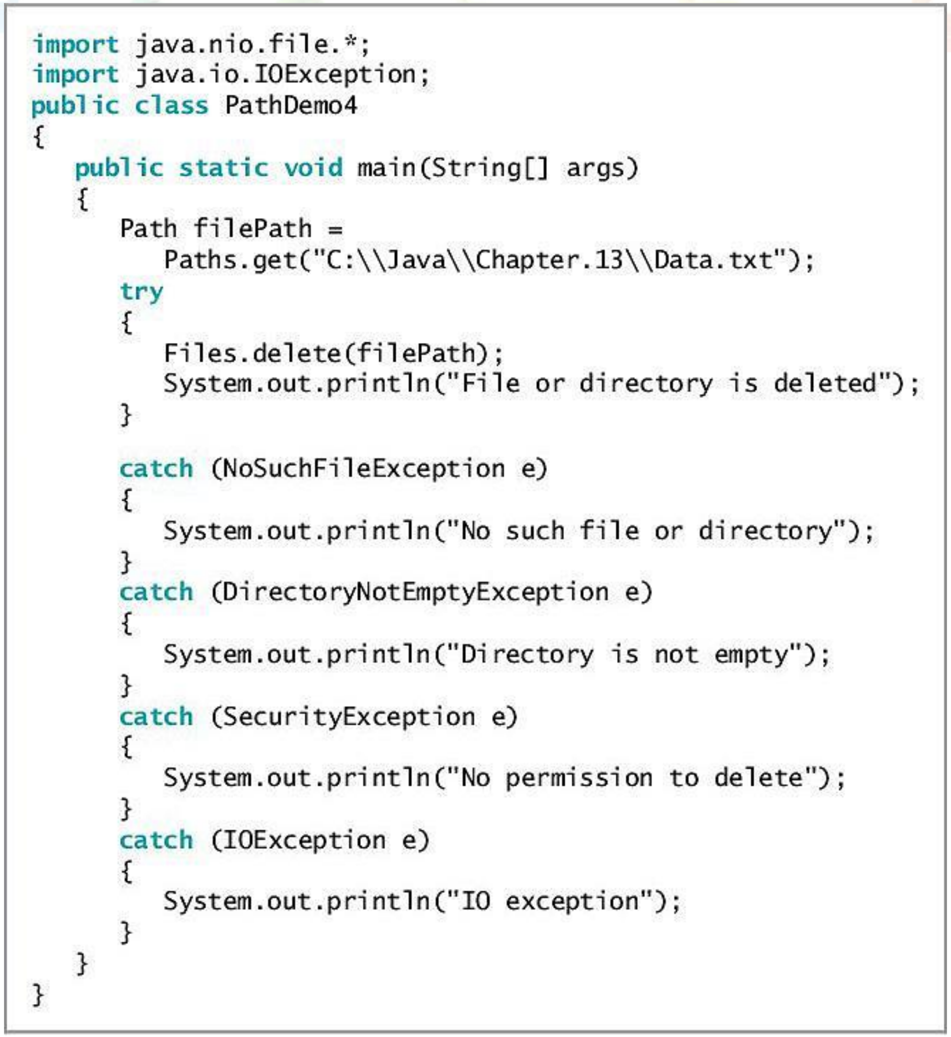
Checking File Accessibility



```
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\\PathDemo.class");
      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

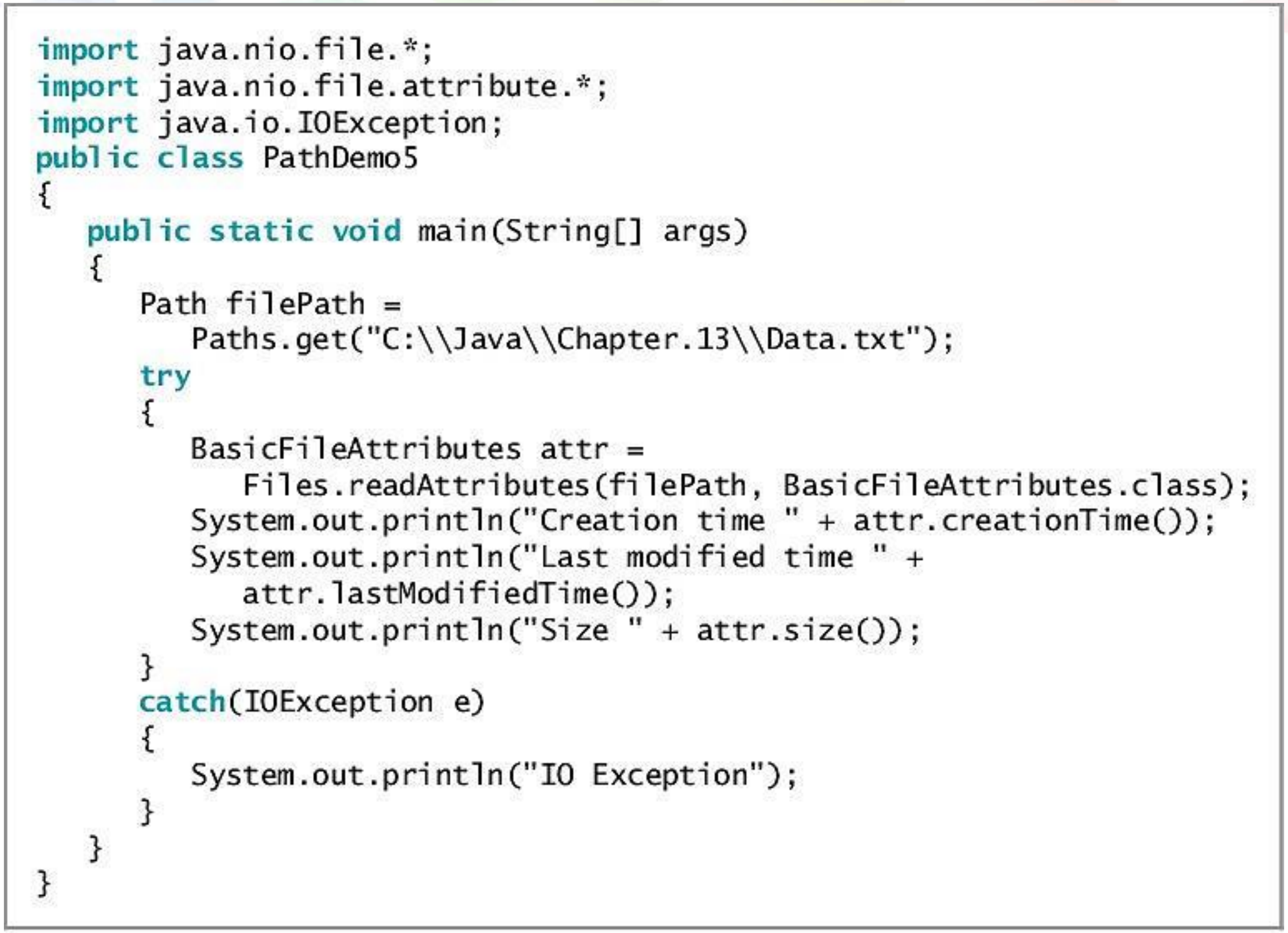
Deleting a Path



```
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");
      }

      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

Determining File Attributes



```
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

File Organization, Streams, and Buffers

- When you need to retain data for any significant amount of time, save it on a permanent, secondary storage device
- Businesses store data in hierarchy
  - **Character**
  - **Field**
  - **Record**
  - Files
- **Sequential access file**
  - Each record is stored in order based on value in some field

File Organization, Streams, and Buffers (cont'd.)

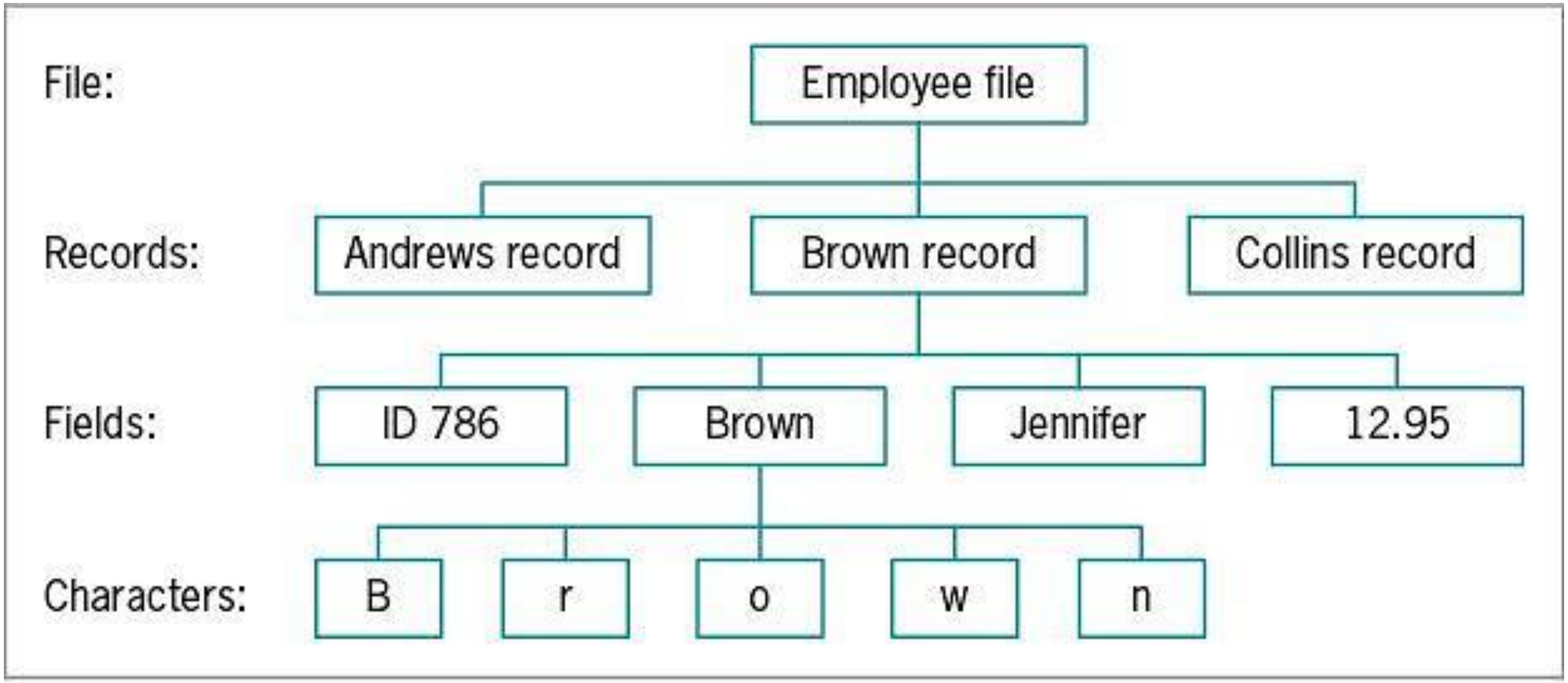


Figure 13-12 Data hierarchy

File Organization, Streams, and Buffers (cont'd.)

- **Open a file**
  - Create object
  - Associate a stream of bytes with it
- **Close the file**
  - Make it no longer available to your application
  - You should always close every file you open
- **Stream**
  - Bytes flow into your program from an input device
  - Bytes flow out of your application to an output device
  - Most streams flow in only one direction

File Organization, Streams, and Buffers (cont'd.)

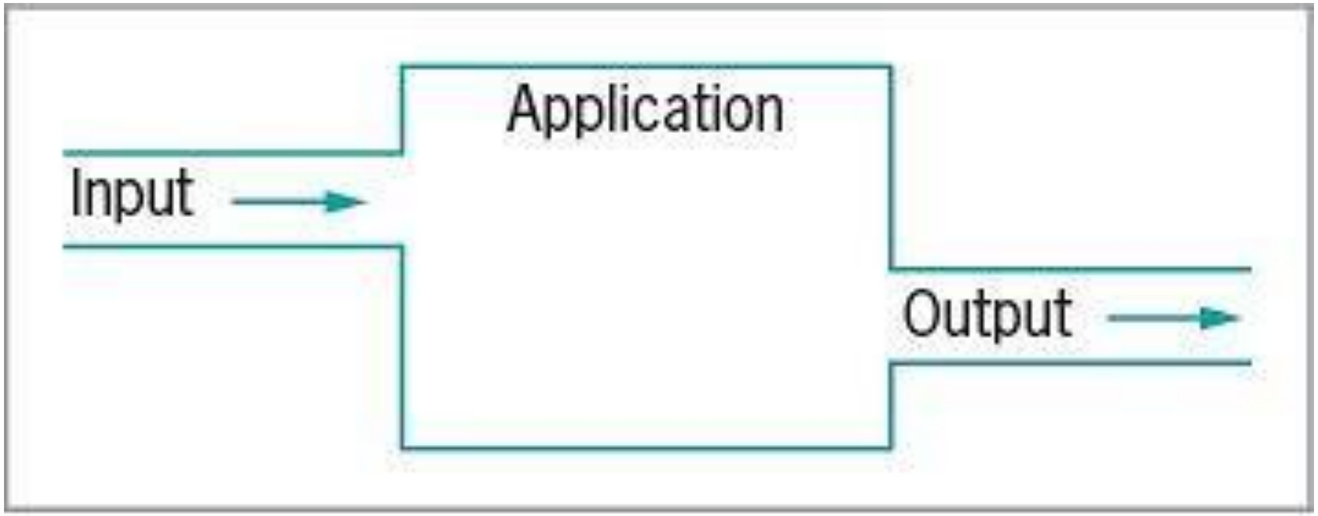


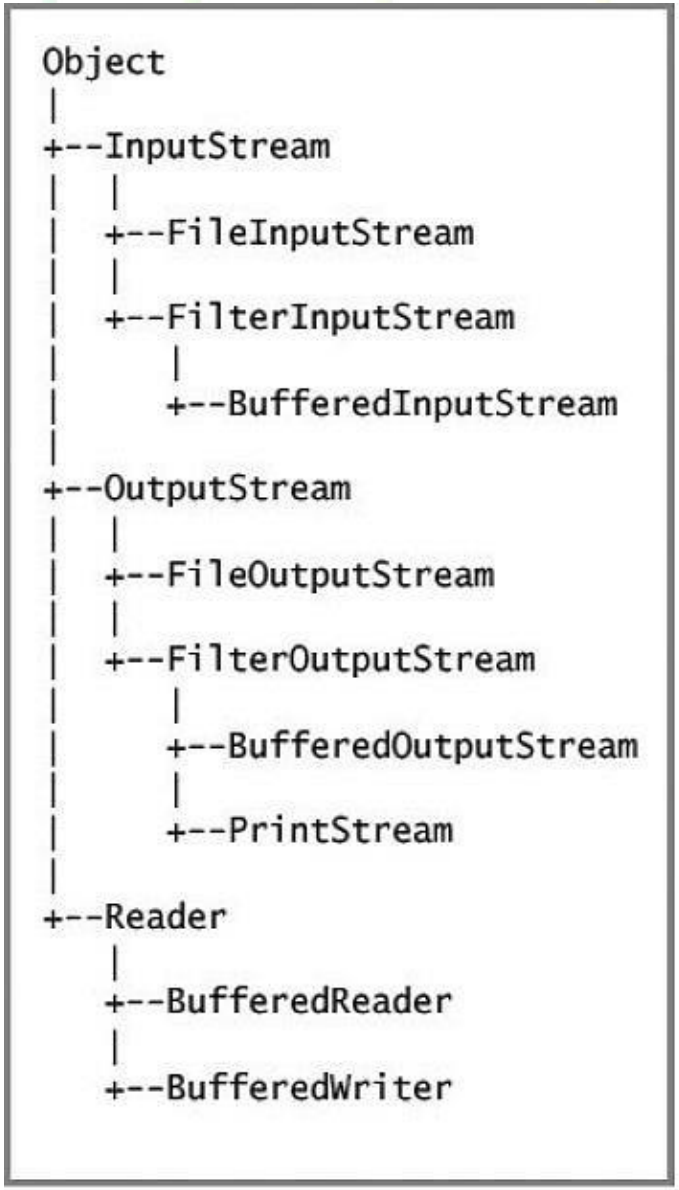
Figure 13-13 File streams

File Organization, Streams, and Buffers (cont'd.)

- **Buffer**
  - Memory location where bytes are held after they are logically output, but before they are sent to the output device
  - Using a buffer improves program performance
- **Flushing**
  - Clears any bytes that have been sent to a buffer for output, but have not yet been output to a hardware device

Using Java’s IO Classes

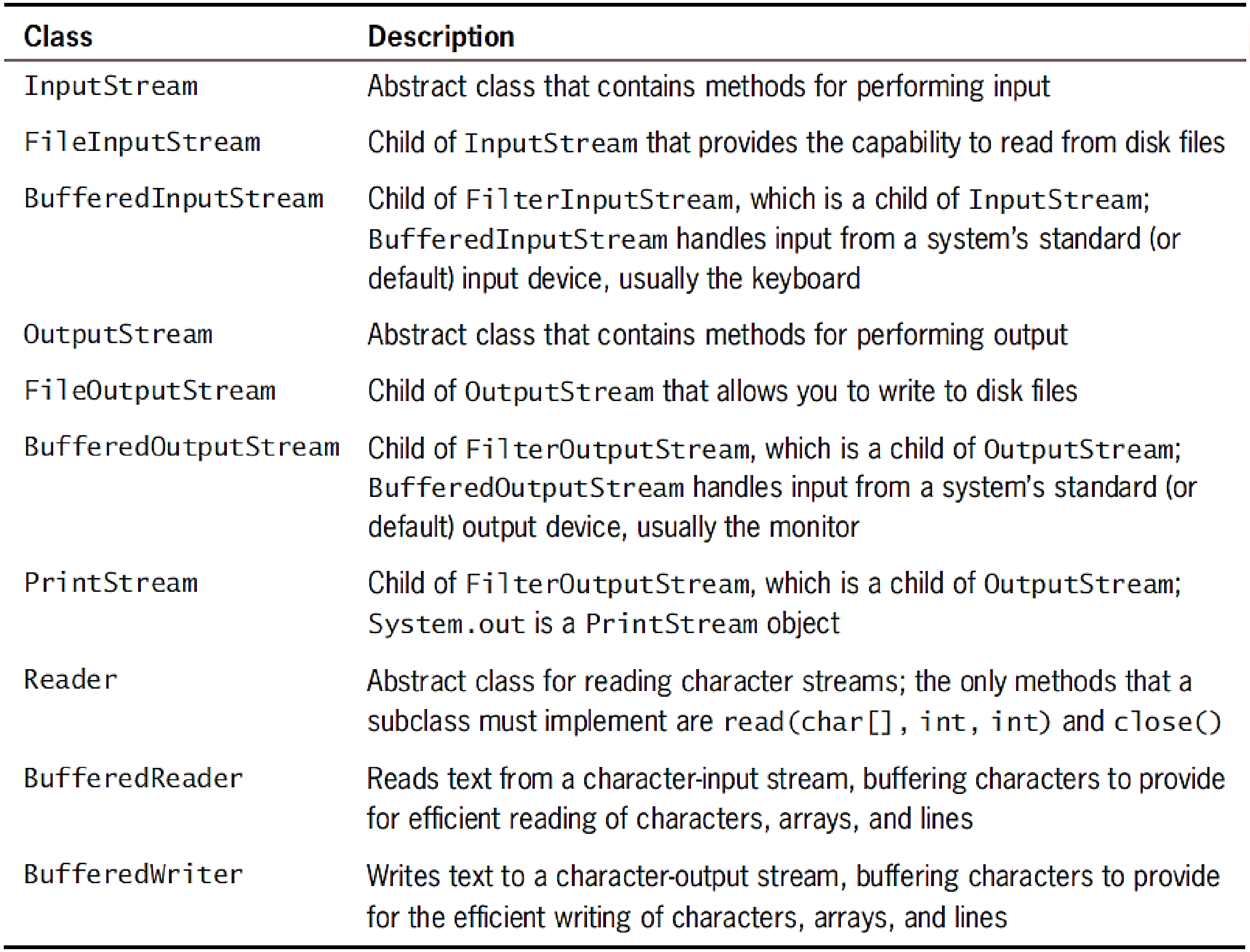
- `InputStream`, `OutputStream`, and `Reader`
  - Abstract classes that contain methods for performing input and output
- `System.out`
  - `PrintStream` object
  - Defined in `System` class
- `System.err`
  - Usually reserved for error messages



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

Figure 13-14 Relationship of selected IO classes

Using Java’s IO Classes (cont'd.)



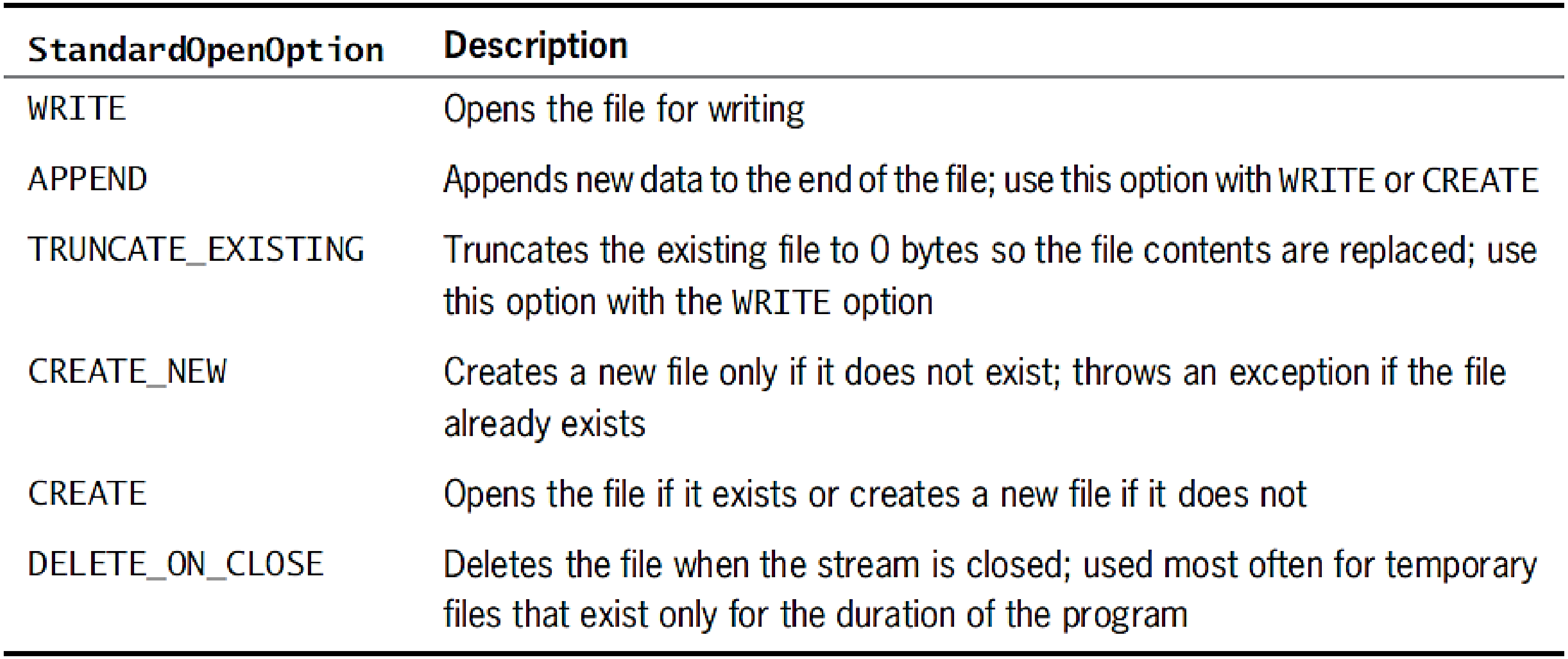
Class	Description
InputStream	Abstract class that contains methods for performing input
FileInputStream	Child of InputStream that provides the capability to read from disk files
BufferedInputStream	Child of FilterInputStream, which is a child of InputStream; BufferedInputStream handles input from a system's 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
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 Selected classes used for input and output

Writing to a File

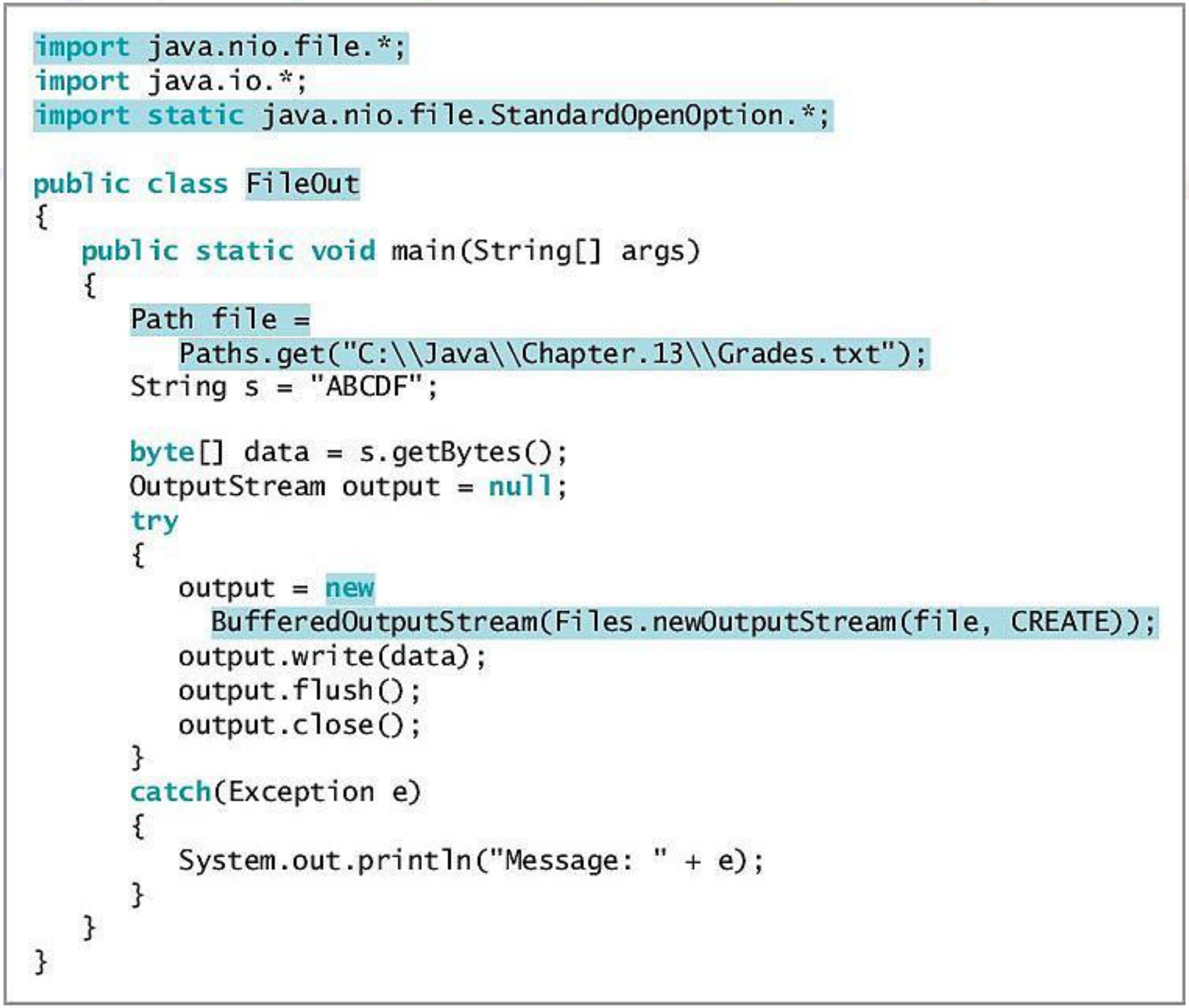
- Assign file to the `OutputStream`
  - Construct a `BufferedOutputStream` object
  - Assign it to the `OutputStream`
- Create a writeable file by using the `Path` class `newOutputStream()` method
  - Creates a file if it does not already exist
  - Opens the file for writing and returns an `OutputStream` that can be used to write bytes to the file

Writing to a File (cont'd.)



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

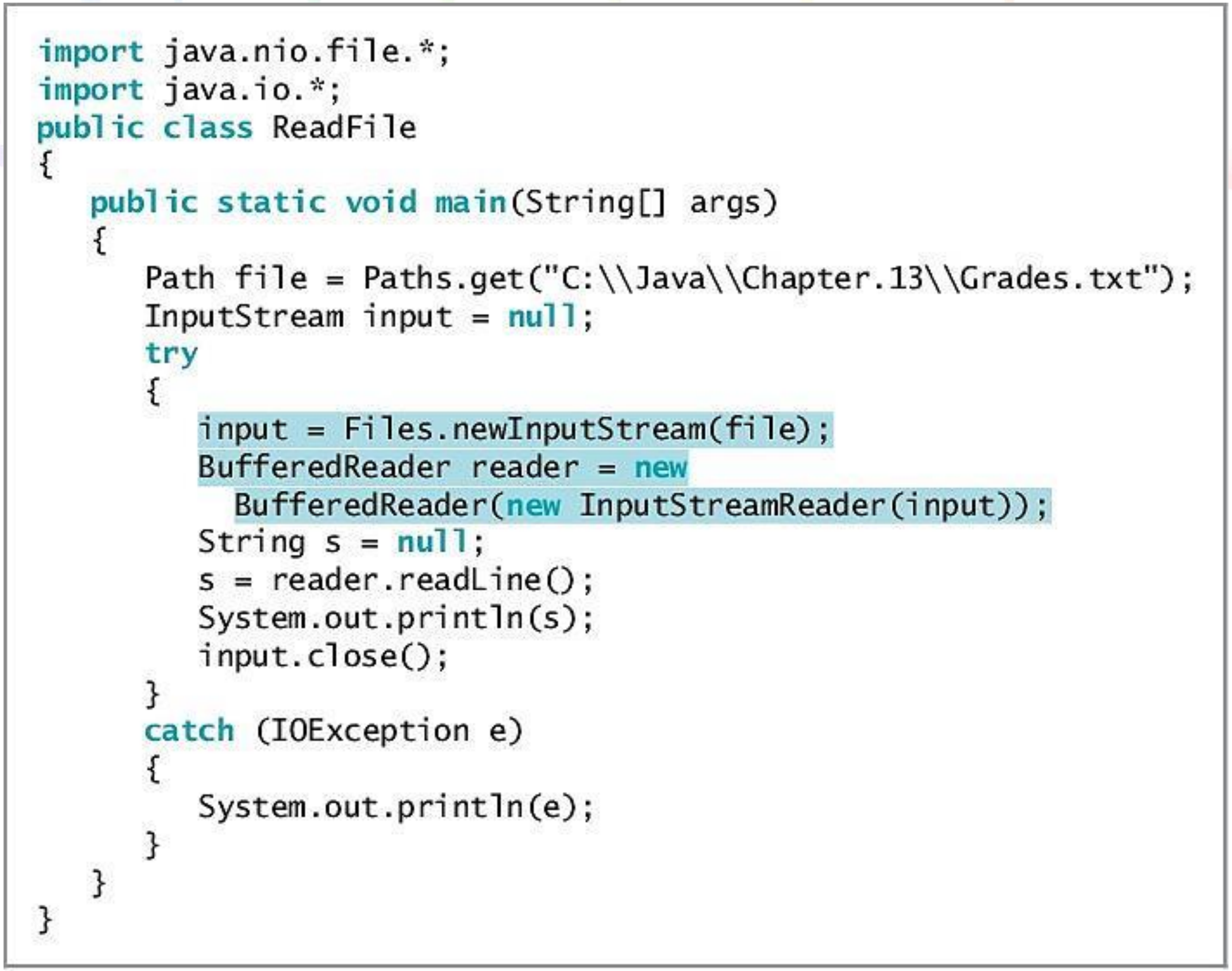


```java
import java.nio.file.*;
import java.io.*;
import static java.nio.file.StandardOpenOption.*;

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

      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

Reading from a File

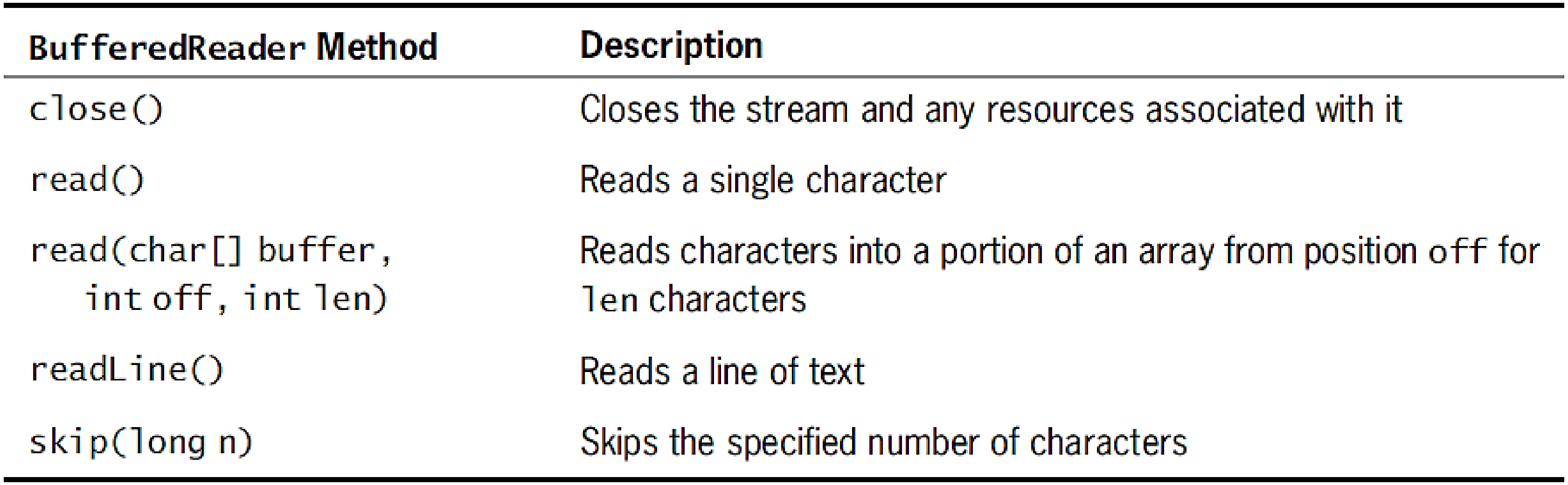
- Use an `InputStream` as you would use an `OutputStream`
- Open a file for reading with the `newInputStream()` method
  - Returns a stream that can read bytes from a file



```
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

Reading from a File (cont'd.)

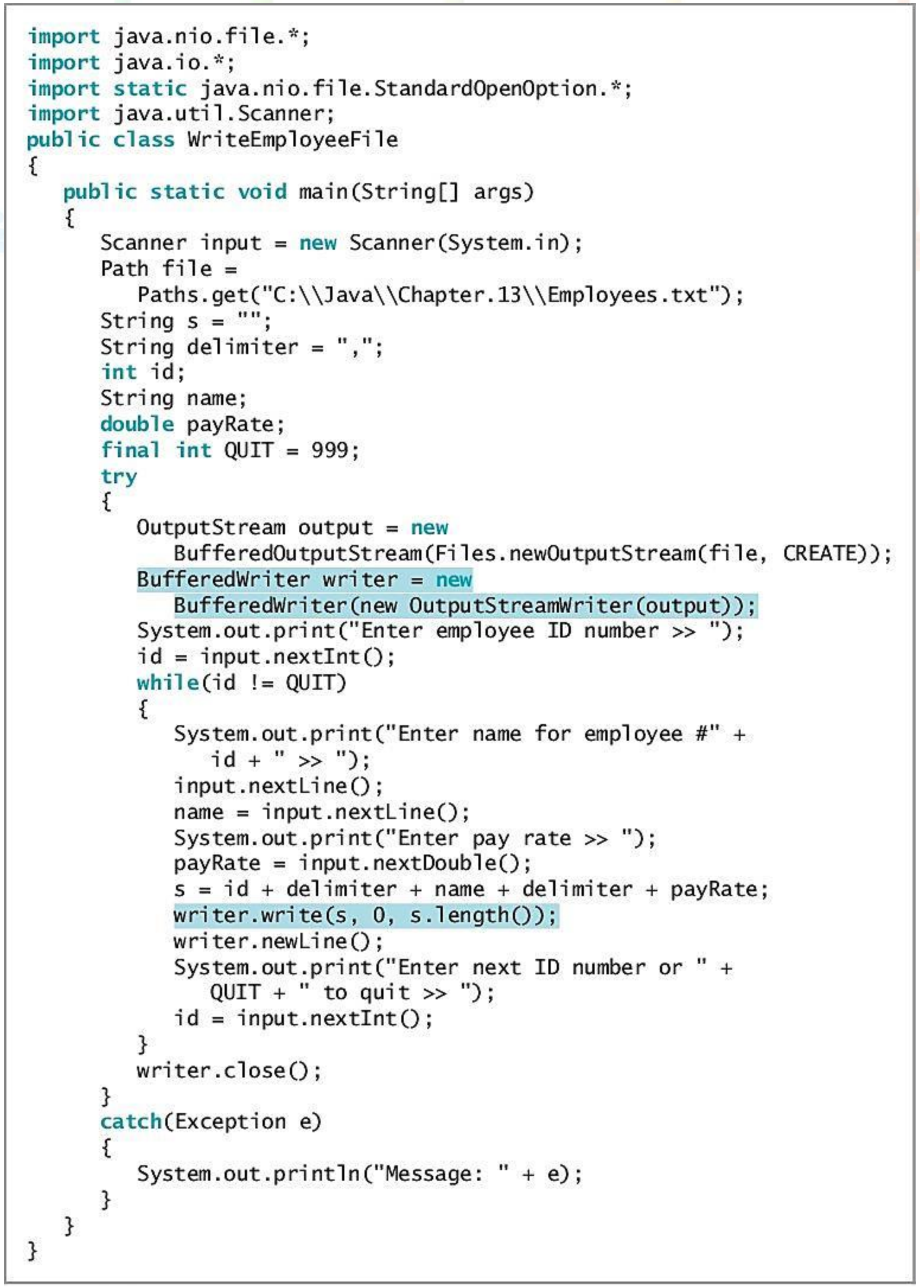


BufferedReader Method	Description
close()	Closes the stream and any resources associated with it
read()	Reads a single character
read(char[] buffer, int off, int len)	Reads characters into a portion of an array from position off for len characters
readLine()	Reads a line of text
skip(long n)	Skips the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

Table 13-5 Selected BufferedReader methods

Creating and Using Sequential Data Files

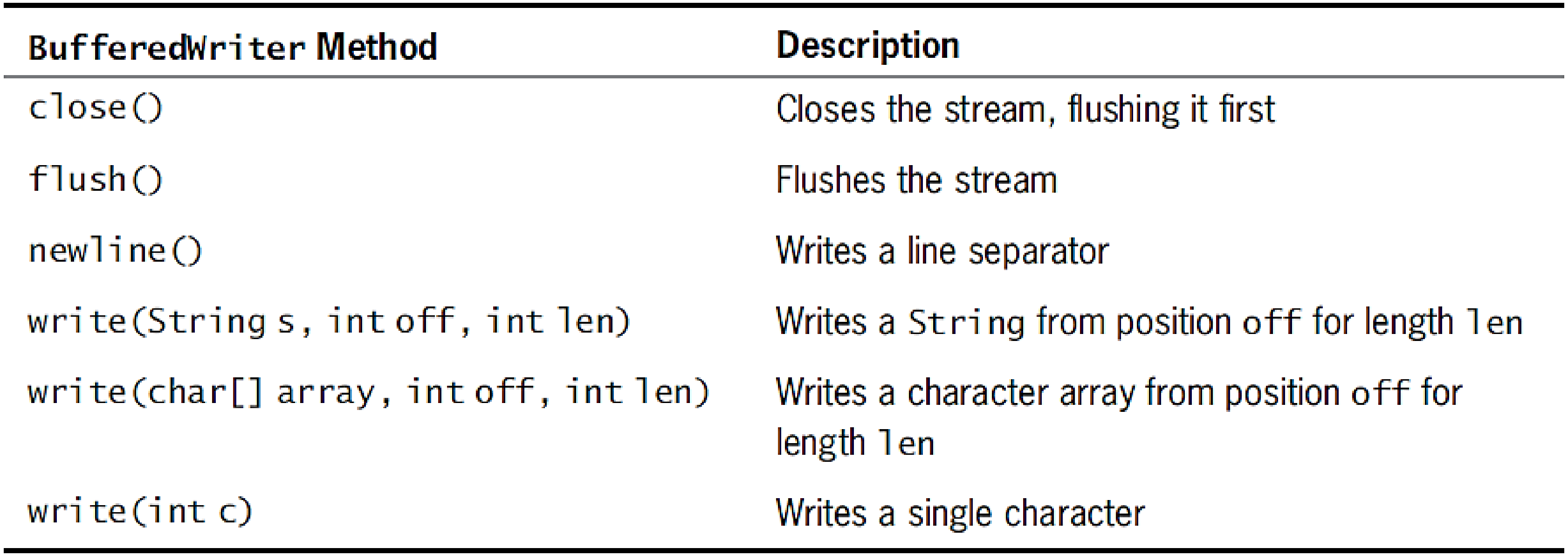
- `BufferedWriter` class
  - Counterpart to `BufferedReader`
  - Writes text to an output stream, buffering the characters
  - The class has three overloaded `write()` methods that provide for efficient writing of characters, arrays, and strings, respectively



```
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);
      }
   }
}
```

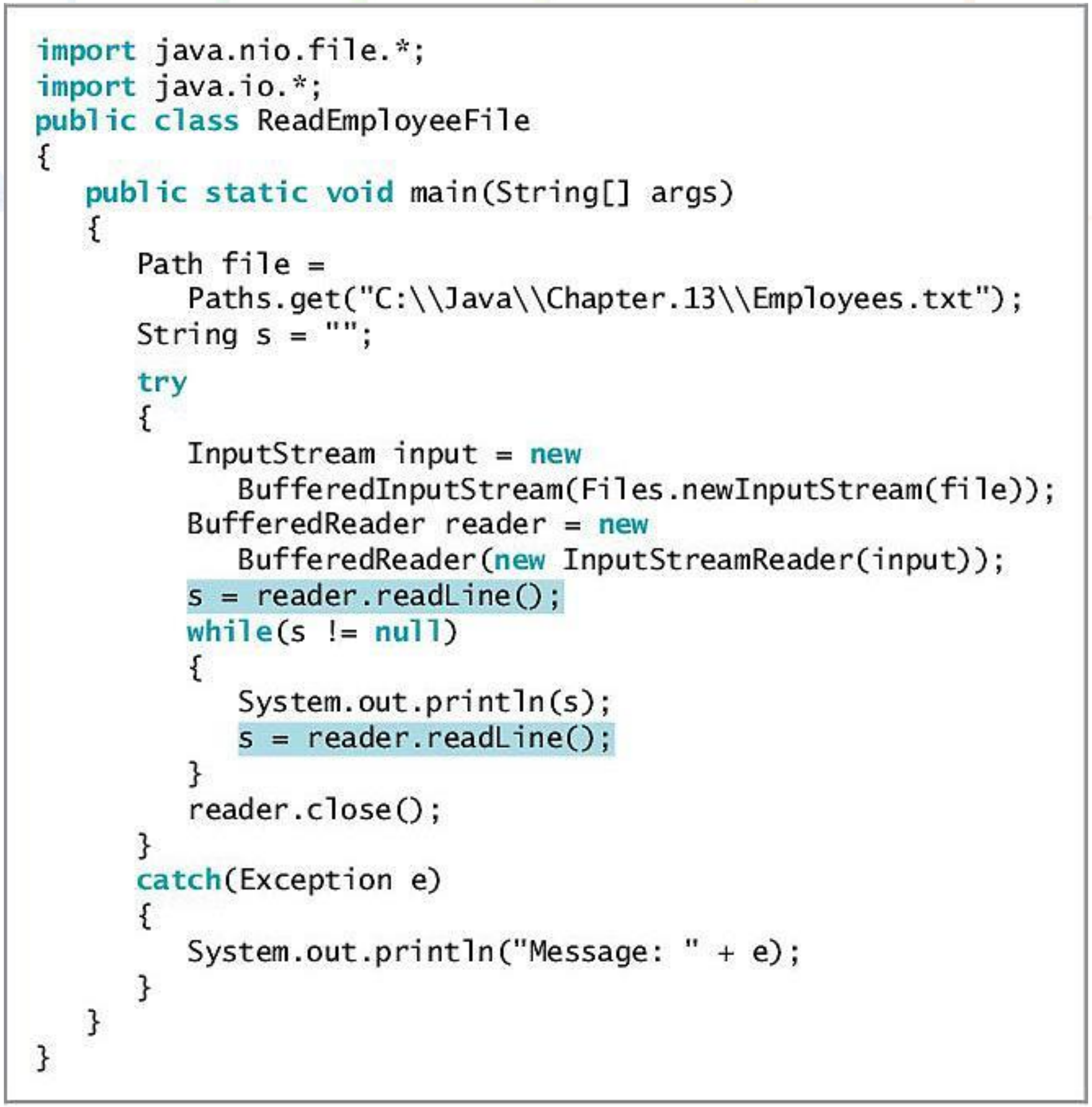
Figure 13-21 The WriteEmployeeFile class

Creating and Using Sequential Data Files (cont'd.)



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



```
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
      {
         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

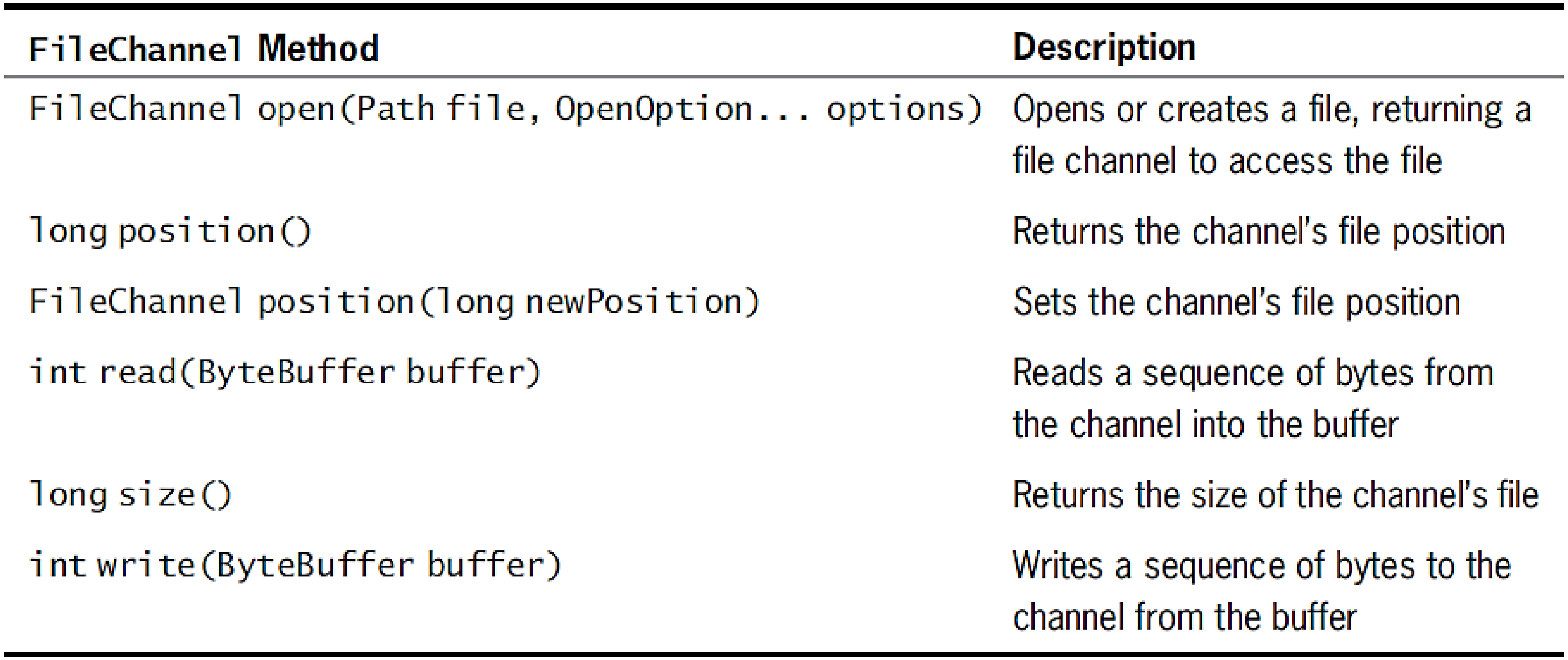
Learning About Random Access Files

- Sequential access files
  - Access records sequentially from beginning to end
  - Good for **batch processing**
    - Same tasks with many records one after the other
  - Inefficient for many applications
- **Real-time** applications
  - Require immediate record access while client waits

Learning About Random Access Files (cont'd.)

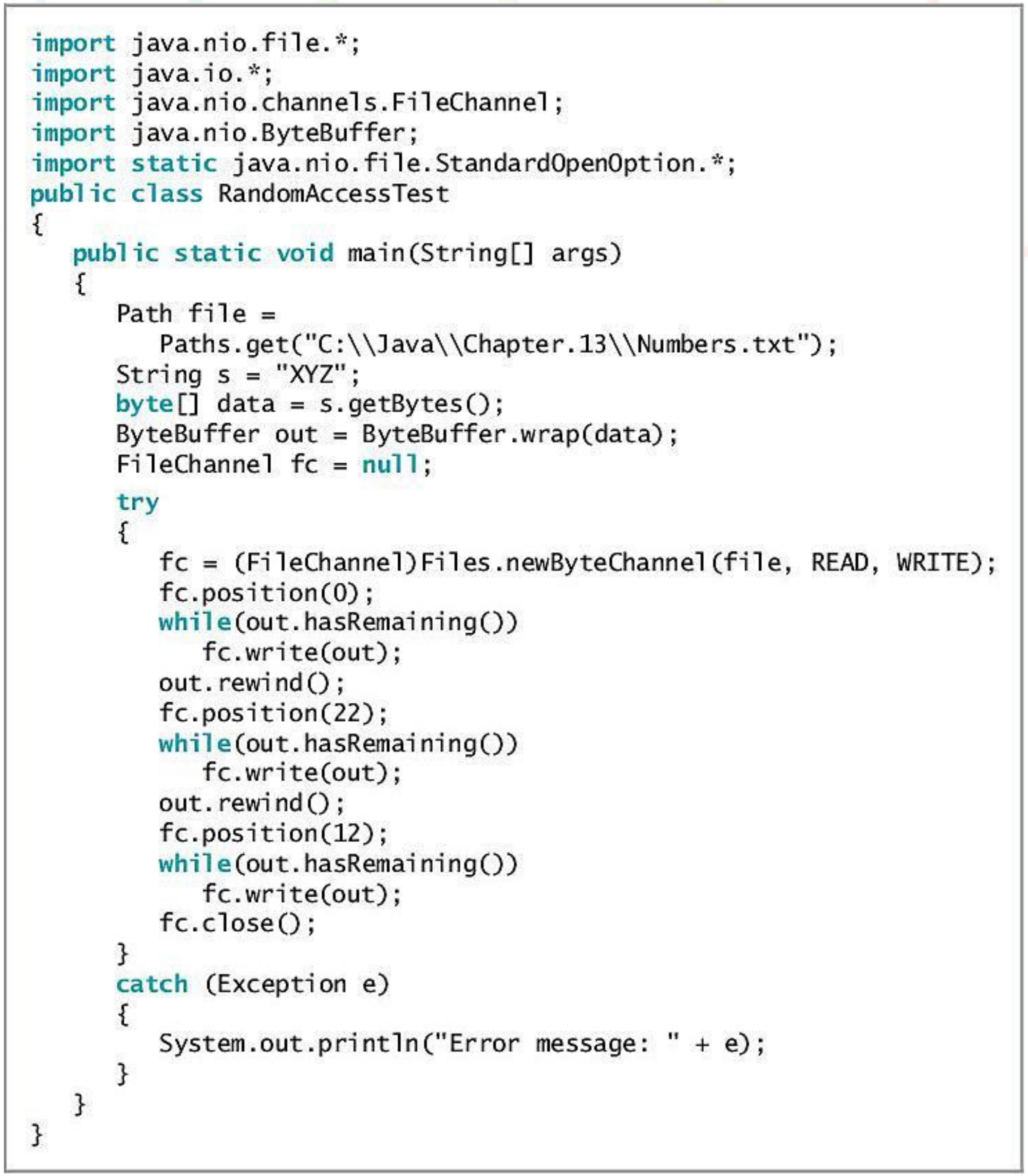
- **Random access files**
  - Records can be located in any order
  - Also called **direct access files** or **instant access files**
- **File channel** object
  - An avenue for reading and writing a file
  - You can search for a specific file location, and operations can start at any specified position
- `ByteBuffer wrap()` method
  - Encompasses an array of bytes into a `ByteBuffer`

Learning About Random Access Files (cont'd.)



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



```
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
      {
         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);
      }
   }
}
```

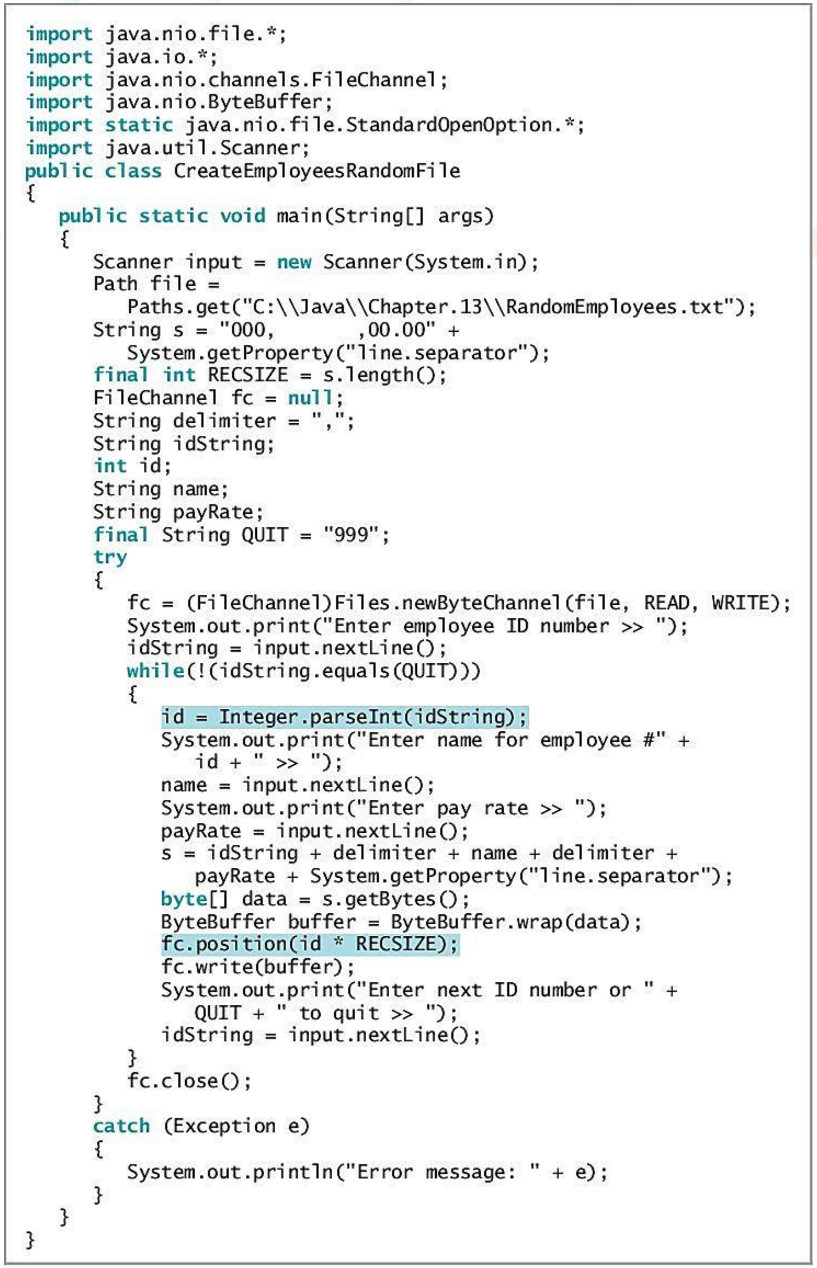
Figure 13-28 The RandomAccessTest class

Writing Records to a Random Access Data File

- Access a particular record

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

- Place records into the file based on a key field
- **Key field**
  - A field that makes a record unique from all others



```java
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

Reading Records from a Random Access File

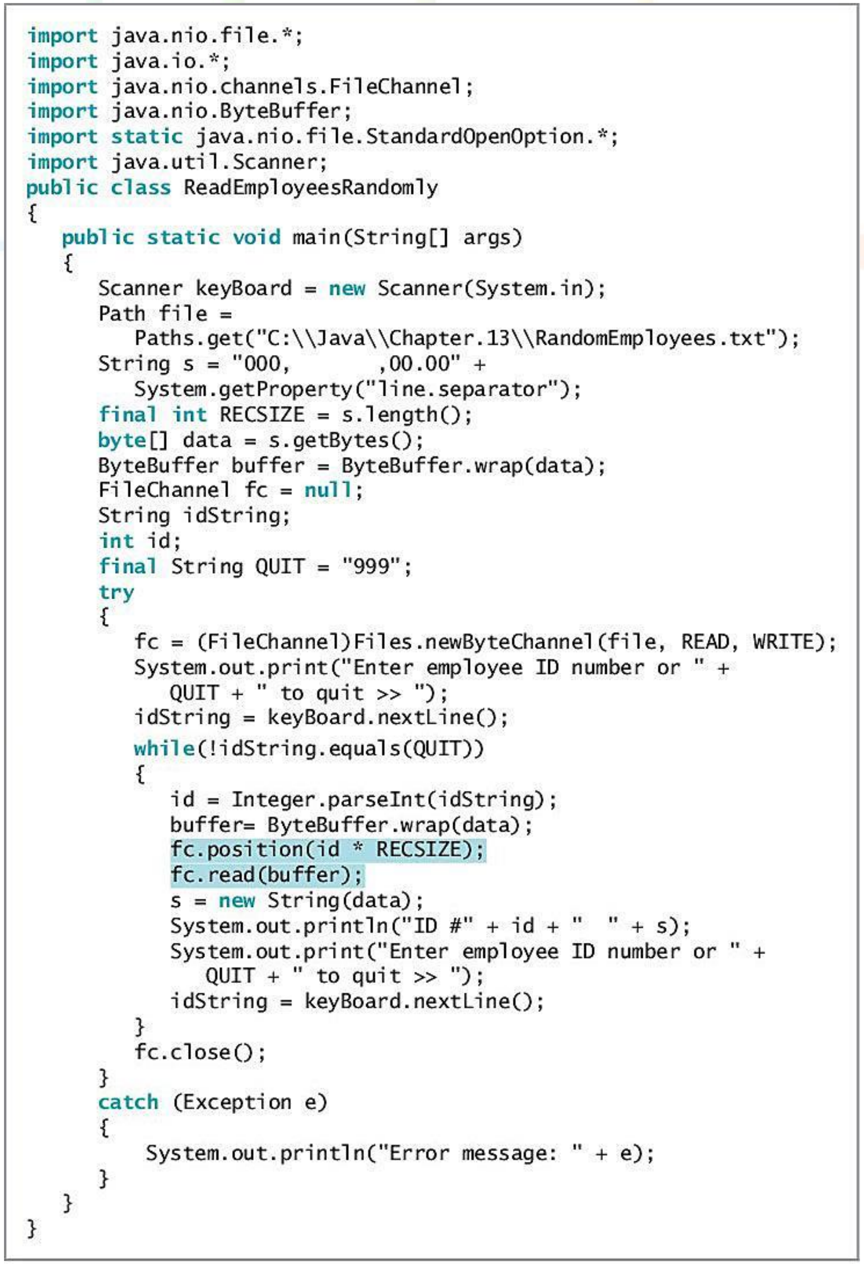
- You can process a random access file either sequentially or randomly

Accessing a Random Access File Sequentially

- `ReadEmployeesSequentially` application
  - Reads through 1,000-record RandomEmployees.txt file sequentially in a `for` loop (shaded)
  - When ID number value is 0:
    - No user-entered records are stored at that point
    - The application does not bother to print it

Accessing a Random Access File Randomly

- To display records in order based on the key field, you do not need to create a random access file and waste unneeded storage
  - Instead, sort the records
- By using a random access file, you retrieve specific record from the file directly without reading through other records



```
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();
         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

You Do It

- Creating Multiple Random Access Files
  - Writing a Method to Create an Empty File
  - Adding Data-Entry Capability to the Program
  - Setting Up a Program to Read the Created Files
  - Displaying File Statistics
  - Reading a File Sequentially
  - Reading a File Randomly

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
  – You must use two backslashes in a string that describes a `Path` in the DOS operating system

Summary

- Files
  - Objects stored on nonvolatile, permanent storage
- `File` and `Files` class
  - Gather file information
- Java views file as a series of bytes
  - Views a stream as an object through which input and output data flows
- `DataOutputStream` class
  - Accomplishes formatted output

Summary (cont'd.)

- `DataInputStream` objects
  - Read binary data from `InputStream`
- Random access files
  - Records can be located in any order
  - `RandomAccessFile` class
- Write objects to files if they implement `Serializable` interface