



D06

PROGRAMMING with JAVA

Ch16 – Files and Streams

Chapter Goals

- To be able to read and write **text files**
- To become familiar with the concepts of **text** and **binary** formats
- To understand when to use **sequential** and **random** file access
- To be able to read and write objects using **serialization**

Reading Text Files

- The simplest way for reading text is to use the **Scanner** class
- To read input from a disk file:
 1. Construct a **FileReader** object with the name of the input file
 2. Use the `FileReader` to construct a `Scanner` object

```
FileReader reader = new FileReader("input.txt");  
Scanner in = new Scanner(reader);
```

3. Use the `Scanner` methods (such as `next()`, `nextLine()`, `nextInt()`, and `nextDouble()`) to read data from the input file

Writing Text Files

- To write output to a file:
 1. Construct a **PrintWriter** object with the given file name (if the output file already exists, it is emptied before the new data are written into it; if the file doesn't exist, an empty file is created)
 2. Use the familiar `print()` and `println()` methods to send numbers, objects, and strings to a `PrintWriter` (the `print()` and `println()` methods convert numbers to their decimal string representation and use the `toString()` method to convert objects to strings)
 3. When you are done writing to a `PrintWriter`, be sure to **close** it (if your program exits without closing the `PrintWriter`, not all of the output may be written to the disk file)

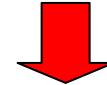
```
PrintWriter out = new PrintWriter("output.txt");
out.println(29.95);
out.println(new Rectangle(5, 10, 15, 25));
out.println("Hello, World!");
out.close();
```

A Sample Program

- The following program reads all lines of an input file and sends them to the output file, preceded by line numbers
- Sample input file:

```
Mary had a little lamb  
Whose fleece was white as snow.  
And everywhere that Mary went,  
The lamb was sure to go!
```

- The program produces the output file:



```
/* 1 */ Mary had a little lamb  
/* 2 */ Whose fleece was white as snow.  
/* 3 */ And everywhere that Mary went,  
/* 4 */ The lamb was sure to go!
```

- This program can be used for numbering Java source files

Continued...

File LineNumberer.java

```
01: import java.io.FileReader;
02: import java.io.IOException;
03: import java.io.PrintWriter;
04: import java.util.Scanner;
05:
06: public class LineNumberer
07: {
08:     public static void main(String[] args)
09:     {
10:         Scanner console = new Scanner(System.in);
11:         System.out.print("Input file: ");
12:         String inputFileNames = console.next();
13:         System.out.print("Output file: ");
14:         String outputFileNames = console.next();
15:
16:         try
17:         {
```

Continued...

File LineNumberer.java

```
18:         FileReader reader = new FileReader(inputFileName);
19:         Scanner in = new Scanner(reader);
20:         PrintWriter out = new PrintWriter(outputFileName);
21:         int lineNumber = 1;
22:
23:         while (in.hasNextLine())
24:         {
25:             String line = in.nextLine();
26:             out.println("/* " + lineNumber + " */ " + line);
27:             lineNumber++;
28:         }
29:
30:         out.close();
31:     }
32:     catch (IOException exception)
33:     {
```

Continued...

File LineNumberer.java

```
34:         System.out.println("Error processing file:"  
                               + exception);  
35:     }  
36: }  
37: }
```

Backslashes in File Names

- When you specify a file name as a constant string, and the name contains backslash characters (as in a Windows file name), you must supply each backslash twice:

```
in = new FileReader("C:\\homework\\input.dat");
```

- When a user supplies a file name to a program, however, the user should not type the backslash twice

Command Line Arguments

- “Invoking the program from the command line” means to start a program by typing its name at a prompt in a terminal or shell window. When you use this method, you can also type any additional information, command line arguments, that the program can use:

```
> java LineNumberer input.txt output.txt
```

- The strings that are typed after the Java program name are placed into the **args** parameter of the `main()` method, so that the `main()` method can process them later

```
args[0] is "input.txt"  
args[1] is "output.txt"
```

- It is customary to interpret strings starting with a hyphen (-) as **options** and other strings as file names

```
> java LineNumberer -c input.txt output.txt
```

Self Check

1. What happens when you supply the same name for the input and output files to the `LineNumberer` program?
2. What happens when you supply the name of a nonexistent input file to the `LineNumberer` program?

Answers

1. When the `PrintWriter` object is created, the output file is emptied. Sadly, that is the same file as the input file. The input file is now empty and the while loop exits immediately.
2. The program catches a `FileNotFoundException`, prints an error message, and terminates.

Text and Binary Formats

- There are two fundamental ways to store data:
 - **Text format:** data items are represented in human-readable form, as a sequence of characters
 - **Binary format:** data items are represented in bytes (a byte is composed of 8 bits and can denote one of 256 values)
- If you store information in text form, you need to use the **Reader** and **Writer** classes and their subclasses to process input and output:

```
FileReader reader = new FileReader("input.txt");  
FileWriter writer = new FileWriter("output.txt");
```

- If you store information in binary form, you use the **InputStream** and **OutputStream** classes and their subclasses:

```
FileInputStream inputStream = new FileInputStream("input.bin");  
FileOutputStream outputStream = new FileOutputStream("output.bin");
```

Text and Binary Formats

- The **Reader** class has a method, **read()**, to read a single character at a time
- The `read()` method actually returns an `int` so that it can signal either that a character has been read or that the end of input has been reached. At the end of input, `read()` returns `-1`. Otherwise it returns the character (as an integer between 0 and 65,535)
- You should test the return value and, if it is not `-1`, cast it to a `char`:

```
Reader reader = . . . ;
int next = reader.read();
char c;
if (next != -1)
    c = (char) next;
```

- Similarly, the **Writer** class has a **write()** method to write a single character at a time

Text and Binary Formats

- The **InputStream** class has a method, **read()**, to read a single byte at a time
- The `read()` method also returns an `int`, namely either the byte that was input (as an integer between 0 and 255) or the integer -1 if the end of the input stream has been reached)
- You should test the return value and, if it is not -1, cast it to a byte:

```
InputStream in = . . . ;
int next = in.read();
byte b; if
(next != -1)
    b = (byte) next;
```

- Similarly, the **OutputStream** class has a **write()** method to write a single byte at a time

Self Check

3. Suppose you need to read an image file that contains color values for each pixel in the image. Will you use a `Reader` or an `InputStream`?
4. Why do the read methods of the `Reader` and `InputStream` classes return an `int` and not a `char` or `byte`?

Answers

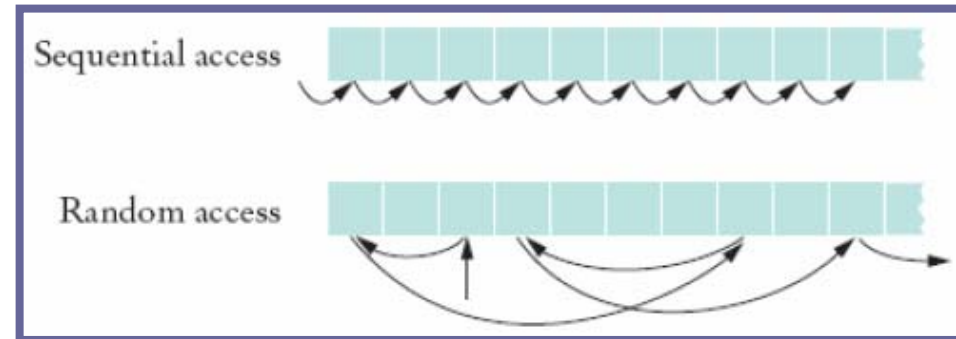
3. Image data is stored in a binary format—try loading an image file into a text editor, and you won't see much text. Therefore, you should use an `InputStream`.
4. They return a special value of -1 to indicate that no more input is available. If the return type had been `char` or `byte`, no special value would have been available that is distinguished from a legal data value.

Random Access vs. Sequential Access

- **Sequential access:**

- To read from a file starting at the beginning and reading the entire contents, one byte at a time, until the end is reached
- It can be inefficient

Figure 4:
Random and Sequential Access



- **Random access:**

- To access specific locations in a file and change only those locations
- Only disk files support random access (`System.in` and `System.out` do not support random access)
- Each disk file has a special **file pointer** position. You can read or write at the position where the pointer is. Normally, the file pointer is at the end of the file, and any output is appended to the end. However, if you move the file pointer to the middle of the file and write to the file, the output overwrites what is already there

RandomAccessFile

- In Java, you use a **RandomAccessFile** object to access a file and move a file pointer. To **open** a random access file, you supply a file name and a string to specify the open mode, “**r**” for reading only or “**rw**” for reading and writing

```
RandomAccessFile f = new RandomAccessFile("bank.dat", "rw");
```

- Methods that can be used with file pointers:
 - **seek()** → to move the file pointer to a specific byte n (counted from the beginning of the file)
 - **getFilePointer()** → to get the current position of the file pointer
 - **length()** → to determine the number of bytes in a file

```
f.seek(n);
```

```
long fileLength = f.length();
```

```
long n = f.getFilePointer(); // type "long" (files can be very large)
```

A Sample Program

- Use a random access file to store a set of bank accounts. The program lets you pick an account and deposit money into it
- To manipulate a data set in a file, pay special attention to data formatting:
 - Suppose we store the data as text. Say account 1001 has a balance of \$900, and account 1015 has a balance of 0:

1 0 0 1 9 0 0 1 0 1 5 0

- We want to deposit \$100 into account 1001

1 0 0 1 9 0 0 1 0 1 5 0



- If we now simply write out the new value, the result is

1 0 0 1 1 0 0 0 1 0 1 5 0



Continued...

A Sample Program

- A better way to manipulate a data set in a file:
 - Give each value a fixed size that is sufficiently large
 - Every record has the same size
 - Easy to skip quickly to a given record
 - To store numbers, it is easier to store them in binary format
- **RandomAccessFile** class stores binary data
- `readInt()` and `writeInt()` read/write integers as four-byte quantities
- `readDouble()` and `writeDouble()` use 8 bytes

```
double x = f.readDouble();  
f.writeDouble(x);
```

A Sample Program

- To find out how many bank accounts are in the file:

```
public int size() throws IOException
{
    return (int) (file.length() / RECORD_SIZE);
    // RECORD_SIZE is 12 bytes:
    // 4 bytes for the account number and 8 bytes for the balance }
}
```

- To read the nth account in the file:

```
public BankAccount read(int n)
    throws IOException
{
    file.seek(n * RECORD_SIZE);
    int accountNumber = file.readInt();
    double balance = file.readDouble();
    return new BankAccount(accountNumber, balance);
}
```

- To write the nth account in the file:

```
public void write(int n, BankAccount account)
    throws IOException
{
    file.seek(n * RECORD_SIZE);
    file.writeInt(account.getAccountNumber());
    file.writeDouble(account.getBalance());
}
```

Continued...

File BankDataTester.java

```
01: import java.io.IOException;
02: import java.io.RandomAccessFile;
03: import java.util.Scanner;
04:
05: /**
06:     This program tests random access. You can access existing
07:     accounts and deposit money, or create new accounts. The
08:     accounts are saved in a random access file.
09: */
10: public class BankDataTester
11: {
12:     public static void main(String[] args)
13:         throws IOException
14:     {
15:         Scanner in = new Scanner(System.in);
16:         BankData data = new BankData();
17:         try
```

Continued...

File BankDatatester.java

```
18:     {
19:         data.open("bank.dat");
20:
21:         boolean done = false;
22:         while (!done)
23:         {
24:             System.out.print("Account number: ");
25:             int accountNumber = in.nextInt();
26:             System.out.print("Amount to deposit: ");
27:             double amount = in.nextDouble();
28:
29:             int position = data.find(accountNumber);
30:             BankAccount account;
31:             if (position >= 0)
32:             {
33:                 account = data.read(position);
34:                 account.deposit(amount);
```

Continued...

File BankDatatester.java

```
35:         System.out.println("new balance="
36:             + account.getBalance());
37:     }
38:     else // Add account
39:     {
40:         account = new BankAccount(accountNumber,
41:             amount);
42:         position = data.size();
43:         System.out.println("adding new account");
44:     }
45:     data.write(position, account);
46:
47:     System.out.print("Done? (Y/N) ");
48:     String input = in.next();
49:     if (input.equalsIgnoreCase("Y")) done = true;
50: }
51: }
```

Continued...

File BankDatatester.java

```
52:         finally
53:         {
54:             data.close();
55:         }
56:     }
57: }
58:
59:
60:
61:
62:
63:
64:
65:
66:
67:
68:
69:
```

File BankData.java

```
001: import java.io.IOException;
002: import java.io.RandomAccessFile;
003:
004: /**
005:     This class is a conduit to a random access file
006:     containing savings account data.
007: */
008: public class BankData
009: {
010:     /**
011:         Constructs a BankData object that is not associated
012:         with a file.
013:     */
014:     public BankData()
015:     {
016:         file = null;
017:     }
```

Continued...

File BankData.java

```
018:
019:     /**
020:         Opens the data file.
021:         @param filename the name of the file containing savings
022:         account information
023:     */
024:     public void open(String filename)
025:         throws IOException
026:     {
027:         if (file != null) file.close();
028:         file = new RandomAccessFile(filename, "rw");
029:     }
030:
031:     /**
032:         Gets the number of accounts in the file.
033:         @return the number of accounts
034:     */
```

Continued...

File BankData.java

```
035:     public int size()
036:         throws IOException
037:     {
038:         return (int) (file.length() / RECORD_SIZE);
039:     }
040:
041:     /**
042:      * Closes the data file.
043:      */
044:     public void close()
045:         throws IOException
046:     {
047:         if (file != null) file.close();
048:         file = null;
049:     }
050:
```

Continued...

File BankData.java

```
051:    /**
052:        Reads a savings account record.
053:        @param n the index of the account in the data file
054:        @return a savings account object initialized with
055:            // the file data
056:    */
057:    public BankAccount read(int n)
058:        throws IOException
059:    {
060:        file.seek(n * RECORD_SIZE);
061:        int accountNumber = file.readInt();
062:        double balance = file.readDouble();
063:        return new BankAccount(accountNumber, balance);
064:    }
065:    /**
066:        Finds the position of a bank account with a given
067:        // number
```

Continued...

File BankData.java

```
067:         @param accountNumber the number to find
068:         @return the position of the account with the given
           // number,
069:         or -1 if there is no such account
070:     */
071:     public int find(int accountNumber)
072:         throws IOException
073:     {
074:         for (int i = 0; i < size(); i++)
075:         {
076:             file.seek(i * RECORD_SIZE);
077:             int a = file.readInt();
078:             if (a == accountNumber) // Found a match
079:                 return i;
080:         }
081:         return -1; // No match in the entire file
082:     }
```

Continued...

File BankData.java

```
083:
084:     /**
085:         Writes a savings account record to the data file
086:         @param n the index of the account in the data file
087:         @param account the account to write
088:     */
089:     public void write(int n, BankAccount account)
090:         throws IOException
091:     {
092:         file.seek(n * RECORD_SIZE);
093:         file.writeInt(account.getAccountNumber());
094:         file.writeDouble(account.getBalance());
095:     }
096:
097:     private RandomAccessFile file;
098:
```

Continued...

File BankData.java

```
099:     public static final int INT_SIZE = 4;
100:     public static final int DOUBLE_SIZE = 8;
101:     public static final int RECORD_SIZE
102:         = INT_SIZE + DOUBLE_SIZE;
103: }
```

Output

```
Account number: 1001
Amount to deposit: 100
adding new account
Done? (Y/N) N
Account number: 1018
Amount to deposit: 200
adding new account
Done? (Y/N) N
Account number: 1001
Amount to deposit: 1000
new balance=1100.0
Done? (Y/N) Y
```

Self Check

7. Why doesn't `System.out` support random access?
8. What is the advantage of the binary format for storing numbers? What is the disadvantage?

Answers

7. Suppose you print something, and then you call `seek(0)`, and print again to the same location. It would be difficult to reflect that behavior in the console window.
8. Advantage: The numbers use a fixed amount of storage space, making it possible to change their values without affecting surrounding data. Disadvantage: You cannot read a binary file with a text editor.

Object Streams

- The `ObjectOutputStream` class can save entire objects out to disk
- The `ObjectInputStream` class can read them back in
- Objects are saved in binary format; hence, you use streams and not writers

Writing and Reading a BankAccount Object to a File

- The **ObjectOutputStream** automatically saves all instance variables of the object to the stream
- When reading the object back in, you use the **readObject()** method of the **ObjectInputStream** class. That method returns an `Object` reference, so you need to remember the types of the objects that you saved and use a cast:

```
BankAccount b = . . . ;  
ObjectOutputStream out = new ObjectOutputStream(  
    new FileOutputStream("bank.dat"));  
out.writeObject(b);
```

```
ObjectInputStream in = new ObjectInputStream(  
    new FileInputStream("bank.dat"));  
BankAccount b = (BankAccount) in.readObject();
```

- The **readObject()** method can throw a **ClassNotFoundException** –it is a checked exception, so you need to catch or declare it

Write and Read an `ArrayList` to a File

- You can store a whole bunch of objects in an array list or array, or inside another object, and then save that object:
- With one instruction, you can save the array list and all the objects that it references

```
ArrayList<BankAccount> a = new ArrayList<BankAccount>();  
// Now add many BankAccount objects into a  
out.writeObject(a);
```

- You can read all of them back with one instruction:

```
ArrayList<BankAccount> a = (ArrayList<BankAccount>)  
    in.readObject();
```

Serializable

- To place objects of a particular class into an object stream, the class must implement the **Serializable** interface. That interface has no methods, so there is no effort involved in implementing it:

```
class BankAccount implements Serializable
{
    . . .
}
```

- The process of saving objects to a stream is called **serialization** because each object is assigned a serial number on the stream
- If the same object is saved twice, only the serial number is written out the second time. When the objects are read back in, duplicate serial numbers are restored as references to the same object

Use Object Streams

- **Object streams** have a huge advantage over other data file formats:
 - You don't have to come up with a way of breaking objects up into numbers and strings when writing a file
 - You don't have to come up with a way of combining numbers and strings back into objects when reading a file
 - The **serialization** mechanism takes care of this automatically. You simply write and read objects
- For this to work, you need to have each of your classes implement the **Serializable** interface
- To save your **data** to disk, it is best to put them all into one large object (such as an array list) and save that object. When you need your data back, read that object back in (it is easy for you to retrieve data from an object that it is to search for them in a file)

File Serialtester.java

```
01: import java.io.File;
02: import java.io.IOException;
03: import java.io.FileInputStream;
04: import java.io.FileOutputStream;
05: import java.io.ObjectInputStream;
06: import java.io.ObjectOutputStream;
07:
08: /**
09:     This program tests serialization of a Bank object.
10:     If a file with serialized data exists, then it is
11:     loaded. Otherwise the program starts with a new bank.
12:     Bank accounts are added to the bank. Then the bank
13:     object is saved.
14: */
15: public class SerialTester
16: {
```

Continued...

File Serialtester.java

```
17:     public static void main(String[] args)
18:         throws IOException, ClassNotFoundException
19:     {
20:         Bank firstBankOfJava;
21:
22:         File f = new File("bank.dat");
23:         if (f.exists())
24:         {
25:             ObjectInputStream in = new ObjectInputStream
26:                 (new FileInputStream(f));
27:             firstBankOfJava = (Bank) in.readObject();
28:             in.close();
29:         }
30:         else
31:         {
32:             firstBankOfJava = new Bank();
33:             firstBankOfJava.addAccount(new
                BankAccount(1001, 20000));
```

Continued...

File Serialtester.java

```
34:         firstBankOfJava.addAccount(new
           BankAccount(1015, 10000));
35:     }
36:
37:     // Deposit some money
38:     BankAccount a = firstBankOfJava.find(1001);
39:     a.deposit(100);
40:     System.out.println(a.getAccountNumber()
           + ":" + a.getBalance());
41:     a = firstBankOfJava.find(1015);
42:     System.out.println(a.getAccountNumber()
           + ":" + a.getBalance());
43:
44:     ObjectOutputStream out = new ObjectOutputStream
45:         (new FileOutputStream(f));
46:     out.writeObject(firstBankOfJava);
47:     out.close();
48: }
49: }
```

Continued...

Output

First Program Run

```
1001:20100.0  
1015:10000.0
```

Second Program Run

```
1001:20200.0  
1015:10000.0
```

Self Check

9. Why is it easier to save an object with an `ObjectOutputStream` than a `RandomAccessFile`?
10. What do you have to do to the `Coin` class so that its objects can be saved in an `ObjectOutputStream`?

Answers

9. You can save the entire object with a single `writeObject` call. With a `RandomAccessFile`, you have to save each field separately.
10. Add `implements Serializable` to the class definition.

Chapter Summary

- When reading text files, use the `Scanner` class
- When writing text files, use the `PrintWriter` class and the `print()` or `println()` methods
- You must **close** all files when you are done processing them
- A `File` object describes a file or directory
- You can pass a `File` object to the constructor of a file reader, writer, or stream
- When you launch a program from the command line, you can specify arguments after the program name. The program can access these strings by processing the `args` parameter of the `main()` method
- **Streams** access sequences of bytes. **Readers** and **writers** access sequences of characters

Continued

Chapter Summary

- Use `FileReader`, `FileWriter`, `FileInputStream`, and `FileOutputStream` classes to read and write disk files
- The `read()` method returns an integer, either -1, at the end of the file, or another value, which you need to cast to a `char` or `byte`
- In **sequential** file access, a file is processed one byte at a time. **Random** access allows access at arbitrary locations in the file, without first reading the bytes preceding the access location
- A **file pointer** is a position in a random access file. Because files can be very large, the file pointer is of type `long`
- Use **object streams** to save and restore all instance fields of an object automatically
- Objects saved to an object stream must belong to classes that implement the `Serializable` interface