

FIT ICT Software Development

Lecture 12

Lecturer : Charles Tyner

OBJECTIVES

In this chapter you'll:

- Create, read, write and update files.
- Retrieve information about files and directories using features of the NIO.2 APIs.
- Learn the differences between text files and binary files.
- Use class **Formatter** to output text to a file.
- Use class **Scanner** to input text from a file.
- Write objects to and read objects from a file using object serialization, interface **Serializable** and classes **ObjectOutputStream** and **ObjectInputStream**.
- Use a **JFileChooser** dialog to allow users to select files or directories on disk.

Introduction

- Data stored in variables and arrays is temporary
 - It's lost when a local variable goes out of scope or when the program terminates
- For long-term retention of data, computers use **files**.
- Computers store files on **secondary storage devices**
 - hard disks, flash drives, DVDs and more.
- Data maintained in files is **persistent data** because it exists beyond the duration of program execution.

Files and Streams

- Java views each file as a sequential **stream of bytes** (Fig. 15.1).
- Every operating system provides a mechanism to determine the end of a file, such as an **end-of-file marker** or a count of the total bytes in the file that is recorded in a system-maintained administrative data structure.
- A Java program simply receives an indication from the operating system when it reaches the end of the stream



Fig. 15.1 | Java's view of a file of n bytes.

Files and Streams (cont.)

- File streams can be used to input and output data as bytes or characters.
 - **Byte-based streams** output and input data in its *binary* format—a `char` is two bytes, an `int` is four bytes, a `double` is eight bytes, etc.
 - **Character-based streams** output and input data as a *sequence of characters* in which every character is two bytes—the number of bytes for a given value depends on the number of characters in that value.
- Files created using byte-based streams are referred to as **binary files**.
- Files created using character-based streams are referred to as **text files**. Text files can be read by text editors.
- Binary files are read by programs that understand the specific content of the file and the ordering of that content.

Files and Streams (cont.)

- A Java program **opens** a file by creating an object and associating a stream of bytes or characters with it.
 - Can also associate streams with different devices.
- Java creates three stream objects when a program begins executing
 - `System.in` (standard input stream) object normally inputs bytes from the keyboard
 - Object `System.out` (the standard output stream object) normally outputs character data to the screen
 - Object `System.err` (the standard error stream object) normally outputs character-based error messages to the screen.
- Class `System` provides methods **`setIn`, `setOut` and `setErr`** to **redirect** the standard input, output and error streams, respectively.

Files and Streams (cont.)

- Java programs perform file processing by using classes from package **java.io** and the subpackages of **java.nio**.
- Character-based input and output can be performed with classes **Scanner** and **Formatter**.
 - Class **Scanner** is used extensively to input data from the keyboard. This class can also read data from a file.
 - Class **Formatter** enables formatted data to be output to any text-based stream in a manner similar to method **System.out.printf**.

Files and Streams (cont.)

- Java SE 8 Adds Another Type of Stream
- JavaSE 8 Lambdas and Streams, introduces a new type of stream that's used to process collections of elements (like arrays and `ArrayLists`), rather than the streams of bytes we discuss in this lectures file-processing examples.

Sequential-Access Text Files

- Sequential-access files store records in order by the record-key field.
- Text files are human-readable files.

Creating a Sequential-Access Text File

- Java imposes no structure on a file
 - Notions such as records do not exist as part of the Java language.
 - You must structure files to meet the requirements of your applications.

Creating a Sequential-Access Text File (cont.)

- **Formatter** outputs formatted **Strings** to the specified stream.
- The constructor with one **String** argument receives the name of the file, including its path.
 - If a path is not specified, the JVM assumes that the file is in the directory from which the program was executed.
- If the file does not exist, it will be created.
- If an existing file is opened, its contents are **truncated**.

```
1 // Fig. 15.3: CreateTextFile.java
2 // Writing data to a sequential text file with class Formatter.
3 import java.io.FileNotFoundException;
4 import java.lang.SecurityException;
5 import java.util.Formatter;
6 import java.util.FormatterClosedException;
7 import java.util.NoSuchElementException;
8 import java.util.Scanner;
9
10 public class CreateTextFile
11 {
12     private static Formatter output; // outputs text to a file
13
14     public static void main(String[] args)
15     {
16         openFile();
17         addRecords();
18         closeFile();
19     }
20 }
```

Fig. 15.3 | Writing data to a sequential text file with class `Formatter`. (Part I of 5.)

```
21 // open file clients.txt
22 public static void openFile()
23 {
24     try
25     {
26         output = new Formatter("clients.txt"); // open the file
27     }
28     catch (SecurityException securityException)
29     {
30         System.err.println("Write permission denied. Terminating.");
31         System.exit(1); // terminate the program
32     }
33     catch (FileNotFoundException fileNotFoundException)
34     {
35         System.err.println("Error opening file. Terminating.");
36         System.exit(1); // terminate the program
37     }
38 }
39 }
```

Fig. 15.3 | Writing data to a sequential text file with class **Formatter**. (Part 2 of 5.)

```
40 // add records to file
41 public static void addRecords()
42 {
43     Scanner input = new Scanner(System.in);
44     System.out.printf("%s%n%s%n? ",
45         "Enter account number, first name, last name and balance.",
46         "Enter end-of-file indicator to end input.");
47
48     while (input.hasNext()) // loop until end-of-file indicator
49     {
50         try
51         {
```

Fig. 15.3 | Writing data to a sequential text file with class `Formatter`. (Part 3 of 5.)

```
52         // output new record to file; assumes valid input
53         output.format("%d %s %s %.2f%n", input.nextInt(),
54                         input.next(), input.next(), input.nextDouble());
55     }
56     catch (FormatterClosedException formatterClosedException)
57     {
58         System.err.println("Error writing to file. Terminating.");
59         break;
60     }
61     catch (NoSuchElementException elementException)
62     {
63         System.err.println("Invalid input. Please try again.");
64         input.nextLine(); // discard input so user can try again
65     }
66
67     System.out.print("? ");
68 } // end while
69 } // end method addRecords
70
```

Fig. 15.3 | Writing data to a sequential text file with class `Formatter`. (Part 4 of 5.)

```
71 // close file
72 public static void closeFile()
73 {
74     if (output != null)
75         output.close();
76 }
77 } // end class CreateTextFile
```

Enter account number, first name, last name and balance.
Enter end-of-file indicator to end input.

```
? 100 Bob Blue 24.98
? 200 Steve Green -345.67
? 300 Pam White 0.00
? 400 Sam Red -42.16
? 500 Sue Yellow 224.62
? ^Z
```

Fig. 15.3 | Writing data to a sequential text file with class Formatter. (Part 5 of 5.)

Creating a Sequential-Access Text File (cont.)

- A **SecurityException** occurs if the user does not have permission to write data to the file.
- A **FileNotFoundException** occurs if the file does not exist and a new file cannot be created.
- **static** method **System.exit** terminates an application.
 - An argument of 0 indicates *successful* program termination.
 - A nonzero value, normally indicates that an error has occurred.
 - The argument is useful if the program is executed from a **batch file** on Windows or a **shell script** on UNIX/Linux/Mac OS X.

Operating system	Key combination
UNIX/Linux/Mac OS X	<i><Enter> <Ctrl> d</i>
Windows	<i><Ctrl> z</i>

Fig. 15.4 | End-of-file key combinations.

Creating a Sequential-Access Text File (cont.)

- Scanner method `hasNext` determines whether the end-of-file key combination has been entered.
- A **NoSuchElementException** occurs if the data being read by a `Scanner` method is in the wrong format or if there is no more data to input.
- Formatter method `format` works like `System.out.printf`
- A **FormatterClosedException** occurs if the `Formatter` is closed when you attempt to output.
- `Formatter` method `close` closes the file.
 - If method `close` is not called explicitly, the operating system normally will close the file when program execution terminates.

Sample data

100	Bob	Blue	24.98
200	Steve	Green	-345.67
300	Pam	White	0.00
400	Sam	Red	-42.16
500	Sue	Yellow	224.62

Fig. 15.5 | Sample data for the program in Fig. 15.3.

Reading Data from a Sequential-Access Text File

- The application (Fig. 15.6) reads records from the file "**clients.txt**" created by the application of Section 15.4.1 and displays the record contents.

```
1 // Fig. 15.6: ReadTextFile.java
2 // This program reads a text file and displays each record.
3 import java.io.IOException;
4 import java.lang.IllegalStateException;
5 import java.nio.file.Files;
6 import java.nio.file.Path;
7 import java.nio.file.Paths;
8 import java.util.NoSuchElementException;
9 import java.util.Scanner;
10
11 public class ReadTextFile
12 {
13     private static Scanner input;
14
15     public static void main(String[] args)
16     {
17         openFile();
18         readRecords();
19         closeFile();
20     }
21 }
```

Fig. 15.6 | Sequential file reading using a Scanner. (Part I of 4.)

```
22 // open file clients.txt
23 public static void openFile()
24 {
25     try
26     {
27         input = new Scanner(Paths.get("clients.txt"));
28     }
29     catch (IOException ioException)
30     {
31         System.err.println("Error opening file. Terminating.");
32         System.exit(1);
33     }
34 }
35
36 // read record from file
37 public static void readRecords()
38 {
39     System.out.printf("%-10s%-12s%-12s%10s%n", "Account",
40                      "First Name", "Last Name", "Balance");
41 }
```

Fig. 15.6 | Sequential file reading using a Scanner. (Part 2 of 4.)

```
42     try
43     {
44         while (input.hasNext()) // while there is more to read
45         {
46             // display record contents
47             System.out.printf("%-10d%-12s%-12s%10.2f%n", input.nextInt(),
48                             input.next(), input.next(), input.nextDouble());
49         }
50     }
51     catch (NoSuchElementException elementException)
52     {
53         System.err.println("File improperly formed. Terminating.");
54     }
55     catch (IllegalStateException stateException)
56     {
57         System.err.println("Error reading from file. Terminating.");
58     }
59 } // end method readRecords
60
```

Fig. 15.6 | Sequential file reading using a Scanner. (Part 3 of 4.)

```
61     // close file and terminate application
62     public static void closeFile()
63     {
64         if (input != null)
65             input.close();
66     }
67 } // end class ReadTextFile
```

Account	First Name	Last Name	Balance
100	Bob	Blue	24.98
200	Steve	Green	-345.67
300	Pam	White	0.00
400	Sam	Red	-42.16
500	Sue	Yellow	224.62

Fig. 15.6 | Sequential file reading using a Scanner. (Part 4 of 4.)

Reading Data from a Sequential- Access Text File

- If a Scanner is closed before data is input, an **IllegalStateException** occurs.

Case Study: A Credit-Inquiry Program

- To retrieve data sequentially from a file, programs start from the beginning of the file and read *all* the data consecutively until the desired information is found.
- It might be necessary to process the file sequentially several times (from the beginning of the file) during the execution of a program.
- Class **Scanner** does *not* allow repositioning to the beginning of the file.
 - The program must *close* the file and *reopen* it.

```
1 // Fig. 15.7: MenuOption.java
2 // enum type for the credit-inquiry program's options.
3
4 public enum MenuOption
5 {
6     // declare contents of enum type
7     ZERO_BALANCE(1),
8     CREDIT_BALANCE(2),
9     DEBIT_BALANCE(3),
10    END(4);
11
12    private final int value; // current menu option
13
14    // constructor
15    private MenuOption(int value)
16    {
17        this.value = value;
18    }
19 } // end enum MenuOption
```

Fig. 15.7 | enum type for the credit-inquiry program's menu options.

```
1 // Fig. 15.8: CreditInquiry.java
2 // This program reads a file sequentially and displays the
3 // contents based on the type of account the user requests
4 // (credit balance, debit balance or zero balance).
5 import java.io.IOException;
6 import java.lang.IllegalStateException;
7 import java.nio.file.Paths;
8 import java.util.NoSuchElementException;
9 import java.util.Scanner;
10
11 public class CreditInquiry
12 {
13     private final static MenuOption[] choices = MenuOption.values();
14
15     public static void main(String[] args)
16     {
17         // get user's request (e.g., zero, credit or debit balance)
18         MenuOption accountType = getRequest();
19     }
}
```

Fig. 15.8 | Credit-inquiry program. (Part I of 8.)

```
20     while (accountType != MenuOption.END)
21     {
22         switch (accountType)
23         {
24             case ZERO_BALANCE:
25                 System.out.printf("%nAccounts with zero balances:%n");
26                 break;
27             case CREDIT_BALANCE:
28                 System.out.printf("%nAccounts with credit balances:%n");
29                 break;
30             case DEBIT_BALANCE:
31                 System.out.printf("%nAccounts with debit balances:%n");
32                 break;
33         }
34
35         readRecords(accountType);
36         accountType = getRequest(); // get user's request
37     }
38 }
39
```

Fig. 15.8 | Credit-inquiry program. (Part 2 of 8.)

```
40 // obtain request from user
41 private static MenuOption getRequest()
42 {
43     int request = 4;
44
45     // display request options
46     System.out.printf("%nEnter request%n%s%n%s%n%s%n%s%n",
47         " 1 - List accounts with zero balances",
48         " 2 - List accounts with credit balances",
49         " 3 - List accounts with debit balances",
50         " 4 - Terminate program");
51
52     try
53     {
54         Scanner input = new Scanner(System.in);
55
56         do // input user request
57         {
58             System.out.printf("%n? ");
59             request = input.nextInt();
60         } while ((request < 1) || (request > 4));
61     }
```

Fig. 15.8 | Credit-inquiry program. (Part 3 of 8.)

```
62     catch (NoSuchElementException noSuchElementException)
63     {
64         System.err.println("Invalid input. Terminating.");
65     }
66
67     return choices[request - 1]; // return enum value for option
68 }
69
70 // read records from file and display only records of appropriate type
71 private static void readRecords(MenuOption accountType)
72 {
73     // open file and process contents
74     try (Scanner input = new Scanner(Paths.get("clients.txt")))
75     {
76         while (input.hasNext()) // more data to read
77         {
78             int accountNumber = input.nextInt();
79             String firstName = input.next();
80             String lastName = input.next();
81             double balance = input.nextDouble();
82         }
83     }
84 }
```

Fig. 15.8 | Credit-inquiry program. (Part 4 of 8.)

```
83         // if proper account type, display record
84         if (shouldDisplay(accountType, balance))
85             System.out.printf("%-10d%-12s%-12s%10.2f%n", accountNumber,
86                               firstName, lastName, balance);
87         else
88             input.nextLine(); // discard the rest of the current record
89     }
90 }
91 catch (NoSuchElementException |
92        IllegalStateException | IOException e)
93 {
94     System.err.println("Error processing file. Terminating.");
95     System.exit(1);
96 }
97 } // end method readRecords
98
```

Fig. 15.8 | Credit-inquiry program. (Part 5 of 8.)

```
99     // use record type to determine if record should be displayed
100    private static boolean shouldDisplay(
101        MenuOption accountType, double balance)
102    {
103        if ((accountType == MenuOption.CREDIT_BALANCE) && (balance < 0))
104            return true;
105        else if ((accountType == MenuOption.DEBIT_BALANCE) && (balance > 0))
106            return true;
107        else if ((accountType == MenuOption.ZERO_BALANCE) && (balance == 0))
108            return true;
109
110        return false;
111    }
112 } // end class CreditInquiry
```

Fig. 15.8 | Credit-inquiry program. (Part 6 of 8.)

Enter request

- 1 - List accounts with zero balances
- 2 - List accounts with credit balances
- 3 - List accounts with debit balances
- 4 - Terminate program

? 1

Accounts with zero balances:

300	Pam	White	0.00
-----	-----	-------	------

Enter request

- 1 - List accounts with zero balances
- 2 - List accounts with credit balances
- 3 - List accounts with debit balances
- 4 - Terminate program

? 2

Accounts with credit balances:

200	Steve	Green	-345.67
400	Sam	Red	-42.16

Fig. 15.8 | Credit-inquiry program. (Part 7 of 8.)

Enter request

- 1 - List accounts with zero balances
- 2 - List accounts with credit balances
- 3 - List accounts with debit balances
- 4 - Terminate program

? 3

Accounts with debit balances:

100	Bob	Blue	24.98
500	Sue	Yellow	224.62

Enter request

- 1 - List accounts with zero balances
- 2 - List accounts with credit balances
- 3 - List accounts with debit balances
- 4 - Terminate program

? 4

Fig. 15.8 | Credit-inquiry program. (Part 8 of 8.)

Updating Sequential-Access Files

- The data in many sequential files cannot be modified without the risk of destroying other data in the file.
- If the name “**white**” needed to be changed to “**worthington**,” the old name cannot simply be overwritten, because the new name requires more space.
- Fields in a text file—and hence records—can vary in size.
- Records in a sequential-access file are not usually updated in place. Instead, the entire file is rewritten.
- Rewriting the entire file is uneconomical to update just one record, but reasonable if a substantial number of records need to be updated.

Object Serialization

- To read an entire object from or write an entire object to a file, Java provides **object serialization**.
- A **serialized object** is represented as a sequence of bytes that includes the object's data and its type information.
- After a serialized object has been written into a file, it can be read from the file and **deserialized** to recreate the object in memory.

Object Serialization (cont.)

- Classes `ObjectInputStream` and `ObjectOutputStream` (package `java.io`), which respectively implement the **ObjectInput** and **ObjectOutput** interfaces, enable entire objects to be read from or written to a stream.
- To use serialization with files, initialize `ObjectInputStream` and `ObjectOutputStream` objects that read from and write to files.

Object Serialization (cont.)

- `ObjectOutput` interface method **writeObject** takes an `Object` as an argument and writes its information to an `OutputStream`.
- A class that implements `ObjectOutput` (such as `ObjectOutputStream`) declares this method and ensures that the object being output implements `Serializable`.
- `ObjectInput` interface method **readObject** reads and returns a reference to an `Object` from an `InputStream`.
 - After an object has been read, its reference can be cast to the object's actual type.

Creating a Sequential-Access File Using Object Serialization

- Objects of classes that implement interface **Serializable** can be *serialized* and *deserialized* with `ObjectOutputStreams` and `ObjectInputStreams`.
- Interface **Serializable** is a **tagging interface**.
 - It does not contain methods.
- A class that implements **Serializable** is *tagged* as being a **Serializable** object.
- An `ObjectOutputStream` will not output an object unless it *is a Serializable* object.

```
1 // Fig. 15.9: Account.java
2 // Serializable Account class for storing records as objects.
3 import java.io.Serializable;
4
5 public class Account implements Serializable
6 {
7     private int account;
8     private String firstName;
9     private String lastName;
10    private double balance;
11
12    // initializes an Account with default values
13    public Account()
14    {
15        this(0, "", "", 0.0); // call other constructor
16    }
17}
```

Fig. 15.9 | Account class for serializable objects. (Part I of 4.)

```
18 // initializes an Account with provided values
19 public Account(int account, String firstName,
20                 String lastName, double balance)
21 {
22     this.account = account;
23     this.firstName = firstName;
24     this.lastName = lastName;
25     this.balance = balance;
26 }
27
28 // set account number
29 public void setAccount(int acct)
30 {
31     this.account = account;
32 }
33
34 // get account number
35 public int getAccount()
36 {
37     return account;
38 }
39
```

Fig. 15.9 | Account class for serializable objects. (Part 2 of 4.)

```
40 // set first name
41 public void setFirstName(String firstName)
42 {
43     this.firstName = firstName;
44 }
45
46 // get first name
47 public String getFirstName()
48 {
49     return firstName;
50 }
51
52 // set last name
53 public void setLastName(String lastName)
54 {
55     this.lastName = lastName;
56 }
57
58 // get last name
59 public String getLastName()
60 {
61     return lastName;
62 }
63
```

Fig. 15.9 | Account class for serializable objects. (Part 3 of 4.)

```
64 // set balance
65 public void setBalance(double balance)
66 {
67     this.balance = balance;
68 }
69
70 // get balance
71 public double getBalance()
72 {
73     return balance;
74 }
75 } // end class Account
```

Fig. 15.9 | Account class for serializable objects. (Part 4 of 4.)

Creating a Sequential-Access File Using Object Serialization (cont.)

- In a class that implements `Serializable`, every variable must be `Serializable`.
- Any one that is not must be declared **`transient`** so it will be ignored during the serialization process.
- *All primitive-type variables are serializable.*
- For reference-type variables, check the class's documentation (and possibly its superclasses) to ensure that the type is `Serializable`.

```
1 // Fig. 15.10: CreateSequentialFile.java
2 // Writing objects sequentially to a file with class ObjectOutputStream.
3 import java.io.IOException;
4 import java.io.ObjectOutputStream;
5 import java.nio.file.Files;
6 import java.nio.file.Paths;
7 import java.util.NoSuchElementException;
8 import java.util.Scanner;
9
10 public class CreateSequentialFile
11 {
12     private static ObjectOutputStream output; // outputs data to file
13
14     public static void main(String[] args)
15     {
16         openFile();
17         addRecords();
18         closeFile();
19     }
20 }
```

Fig. 15.10 | Sequential file created using ObjectOutputStream. (Part 1 of 5.)

```
21 // open file clients.ser
22 public static void openFile()
23 {
24     try
25     {
26         output = new ObjectOutputStream(
27             Files.newOutputStream(Paths.get("clients.ser")));
28     }
29     catch (IOException ioException)
30     {
31         System.err.println("Error opening file. Terminating.");
32         System.exit(1); // terminate the program
33     }
34 }
35
36 // add records to file
37 public static void addRecords()
38 {
39     Scanner input = new Scanner(System.in);
40
41     System.out.printf("%s%n%s%n? ",
42         "Enter account number, first name, last name and balance.",
43         "Enter end-of-file indicator to end input.");
44 }
```

Fig. 15.10 | Sequential file created using ObjectOutputStream. (Part 2 of 5.)

```
45     while (input.hasNext()) // loop until end-of-file indicator
46     {
47         try
48         {
49             // create new record; this example assumes valid input
50             Account record = new Account(input.nextInt(),
51                 input.next(), input.next(), input.nextDouble());
52
53             // serialize record object into file
54             output.writeObject(record);
55         }
56         catch (NoSuchElementException elementException)
57         {
58             System.err.println("Invalid input. Please try again.");
59             input.nextLine(); // discard input so user can try again
60         }
61         catch (IOException ioException)
62         {
63             System.err.println("Error writing to file. Terminating.");
64             break;
65         }
66     }
67 }
```

Fig. 15.10 | Sequential file created using `ObjectOutputStream`. (Part 3 of 5.)

```
66
67         System.out.print("? ");
68     }
69 }
70
71 // close file and terminate application
72 public static void closeFile()
73 {
74     try
75     {
76         if (output != null)
77             output.close();
78     }
79     catch (IOException ioException)
80     {
81         System.err.println("Error closing file. Terminating.");
82     }
83 }
84 } // end class CreateSequentialFile
```

Fig. 15.10 | Sequential file created using ObjectOutputStream. (Part 4 of 5.)

```
Enter account number, first name, last name and balance.  
Enter end-of-file indicator to end input.  
? 100 Bob Blue 24.98  
? 200 Steve Green -345.67  
? 300 Pam White 0.00  
? 400 Sam Red -42.16  
? 500 Sue Yellow 224.62  
? ^Z
```

Fig. 15.10 | Sequential file created using ObjectOutputStream. (Part 5 of 5.)

Reading and Deserializing Data from a Sequential-Access File

- The program in Fig. 15.11 reads records from a file created by the program in Section 15.5.1 and displays the contents.

```
1 // Fig. 15.11: ReadSequentialFile.java
2 // Reading a file of objects sequentially with ObjectInputStream
3 // and displaying each record.
4 import java.io.EOFException;
5 import java.io.IOException;
6 import java.io.ObjectInputStream;
7 import java.nio.file.Files;
8 import java.nio.file.Paths;
9
10 public class ReadSequentialFile
11 {
12     private static ObjectInputStream input;
13
14     public static void main(String[] args)
15     {
16         openFile();
17         readRecords();
18         closeFile();
19     }
20 }
```

Fig. 15.11 | Reading a file of objects sequentially with `ObjectInputStream` and displaying each record. (Part 1 of 6.)

```
21 // enable user to select file to open
22 public static void openFile()
23 {
24     try // open file
25     {
26         input = new ObjectInputStream(
27             Files.newInputStream(Paths.get("clients.ser")));
28     }
29     catch (IOException ioException)
30     {
31         System.err.println("Error opening file.");
32         System.exit(1);
33     }
34 }
35
```

Fig. 15.11 | Reading a file of objects sequentially with `ObjectInputStream` and displaying each record. (Part 2 of 6.)

```
36 // read record from file
37 public static void readRecords()
38 {
39     System.out.printf("%-10s%-12s%-12s%10s%n", "Account",
40                      "First Name", "Last Name", "Balance");
41
42     try
43     {
44         while (true) // loop until there is an EOFException
45         {
46             Account record = (Account) input.readObject();
47
48             // display record contents
49             System.out.printf("%-10d%-12s%-12s%10.2f%n",
50                               record.getAccount(), record.getFirstName(),
51                               record.getLastName(), record.getBalance());
52         }
53     }
```

Fig. 15.11 | Reading a file of objects sequentially with `ObjectInputStream` and displaying each record. (Part 3 of 6.)

```
54     catch (EOFException endOfFileException)
55     {
56         System.out.printf("%No more records%n");
57     }
58     catch (ClassNotFoundException classNotFoundException)
59     {
60         System.err.println("Invalid object type. Terminating.");
61     }
62     catch (IOException ioException)
63     {
64         System.err.println("Error reading from file. Terminating.");
65     }
66 } // end method readRecords
67
```

Fig. 15.11 | Reading a file of objects sequentially with `ObjectInputStream` and displaying each record. (Part 4 of 6.)

```
68 // close file and terminate application
69 public static void closeFile()
70 {
71     try
72     {
73         if (input != null)
74             input.close();
75     }
76     catch (IOException ioException)
77     {
78         System.err.println("Error closing file. Terminating.");
79         System.exit(1);
80     }
81 }
82 } // end class ReadSequentialFile
```

Fig. 15.11 | Reading a file of objects sequentially with `ObjectInputStream` and displaying each record. (Part 5 of 6.)

Account	First Name	Last Name	Balance
100	Bob	Blue	24.98
200	Steve	Green	-345.67
300	Pam	White	0.00
400	Sam	Red	-42.16
500	Sue	Yellow	224.62

No more records

Fig. 15.11 | Reading a file of objects sequentially with `ObjectInputStream` and displaying each record. (Part 6 of 6.)

Reading and De-serialising Data from a Sequential-Access File (cont.)

- `ObjectInputStream` method `readObject` reads an `Object` from a file.
- Method `readObject` throws an **EOFException** if an attempt is made to read beyond the end of the file.
- Method `readObject` throws a `ClassNotFoundException` if the class for the object being read cannot be located.



Software Engineering Observation 15.1

This section introduced object serialization and demonstrated basic serialization techniques.

Serialization is a deep subject with many traps and pitfalls. Before implementing object serialization in industrial-strength applications, carefully read the online Java documentation for object serialization.

Opening Files with JFileChooser

- Class **JFileChooser** displays a dialog that enables the user to easily select files or directories.
- To demonstrate **JFileChooser**, we enhance the example in Section 15.3, as shown in Figs. 15.12–15.13.
- Call method **setFileSelectionMode** specifies what the user can select from the **fileChooser**. For this program, we use **JFileChooser** static constant **FILES_AND_DIRECTORIES** to indicate that files and directories can be selected. Other static constants include **FILES_ONLY** (the default) and **DIRECTORIES_ONLY**.

```
1 // Fig. 15.12: JFileChooserDemo.java
2 // Demonstrating JFileChooser.
3 import java.io.IOException;
4 import java.nio.file.DirectoryStream;
5 import java.nio.file.Files;
6 import java.nio.file.Path;
7 import java.nio.file.Paths;
8 import javax.swing.JFileChooser;
9 import javax.swing.JFrame;
10 import javax.swing.JOptionPane;
11 import javax.swing.JScrollPane;
12 import javax.swing.JTextArea;
13
14 public class JFileChooserDemo extends JFrame
15 {
16     private final JTextArea outputArea; // displays file contents
17
18     // set up GUI
19     public JFileChooserDemo() throws IOException
20     {
21         super("JFileChooser Demo");
22         outputArea = new JTextArea();
23         add(new JScrollPane(outputArea)); // outputArea is scrollable
24         analyzePath(); // get Path from user and display info
25     }
}
```

Fig. 15.12 | Demonstrating JFileChooser. (Part I of 4.)

```
26
27 // display information about file or directory user specifies
28 public void analyzePath() throws IOException
29 {
30     // get Path to user-selected file or directory
31     Path path = getFileOrDirectoryPath();
32
33     if (path != null && Files.exists(path)) // if exists, display info
34     {
35         // gather file (or directory) information
36         StringBuilder builder = new StringBuilder();
37         builder.append(String.format("%s:%n", path.getFileName()));
38         builder.append(String.format("%s a directory%n",
39             Files.isDirectory(path) ? "Is" : "Is not"));
40         builder.append(String.format("%s an absolute path%n",
41             path.isAbsolute() ? "Is" : "Is not"));
42         builder.append(String.format("Last modified: %s%n",
43             Files.getLastModifiedTime(path)));
44         builder.append(String.format("Size: %s%n", Files.size(path)));
45         builder.append(String.format("Path: %s%n", path));
46         builder.append(String.format("Absolute path: %s%n",
47             path.toAbsolutePath()));
48     }
```

Fig. 15.12 | Demonstrating JFileChooser. (Part 2 of 4.)

```
49     if (Files.isDirectory(path)) // output directory listing
50     {
51         builder.append(String.format("%nDirectory contents:%n"));
52
53         // object for iterating through a directory's contents
54         DirectoryStream<Path> directoryStream =
55             Files.newDirectoryStream(path);
56
57         for (Path p : directoryStream)
58             builder.append(String.format("%s%n", p));
59     }
60
61     outputArea.setText(builder.toString()); // display String content
62 }
63 else // Path does not exist
64 {
65     JOptionPane.showMessageDialog(this, path.getFileName() +
66         " does not exist.", "ERROR", JOptionPane.ERROR_MESSAGE);
67 }
68 } // end method analyzePath
69
```

Fig. 15.12 | Demonstrating JFileChooser. (Part 3 of 4.)

```
70 // allow user to specify file or directory name
71 private Path getFileOrDirectoryPath()
72 {
73     // configure dialog allowing selection of a file or directory
74     JFileChooser fileChooser = new JFileChooser();
75     fileChooser.setFileSelectionMode(
76         JFileChooser.FILES_AND_DIRECTORIES);
77     int result = fileChooser.showOpenDialog(this);
78
79     // if user clicked Cancel button on dialog, return
80     if (result == JFileChooser.CANCEL_OPTION)
81         System.exit(1);
82
83     // return Path representing the selected file
84     return fileChooser.getSelectedFile().toPath();
85 }
86 } // end class JFileChooserDemo
```

Fig. 15.12 | Demonstrating JFileChooser. (Part 4 of 4.)

```
1 // Fig. 15.13: JFileChooserTest.java
2 // Tests class JFileChooserDemo.
3 import java.io.IOException;
4 import javax.swing.JFrame;
5
6 public class JFileChooserTest
7 {
8     public static void main(String[] args) throws IOException
9     {
10         JFileChooserDemo application = new JFileChooserDemo();
11         application.setSize(400, 400);
12         application.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13         application.setVisible(true);
14     }
15 } // end class JFileChooserTest
```

Fig. 15.13 | Testing class FileDemonstration. (Part I of 3.)

a) Use this dialog to locate and select a file or directory

Files and directories are displayed here

Click **Open** to submit file or directory name to program

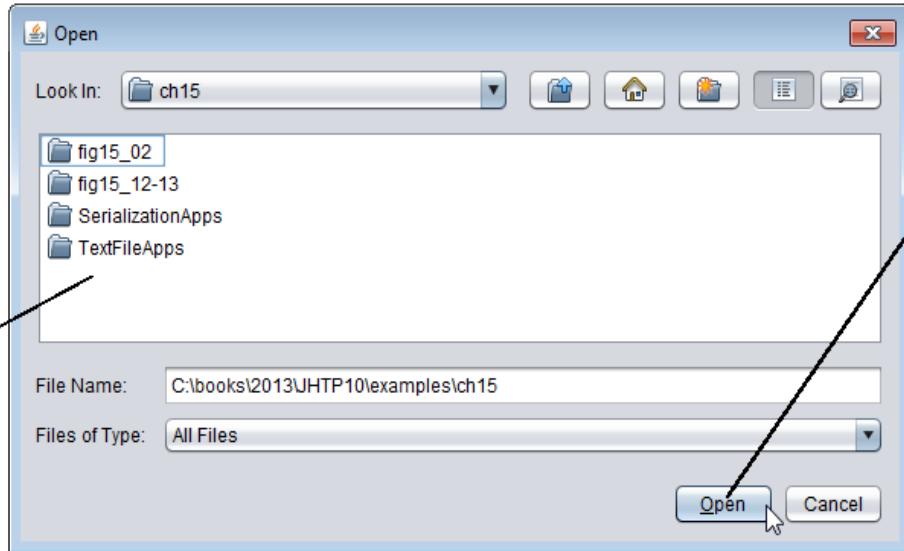


Fig. 15.13 | Testing class `FileDemonstration`. (Part 2 of 3.)

b) Selected file's or directory's information;
if it's a directory, the
contents of that
directory are displayed

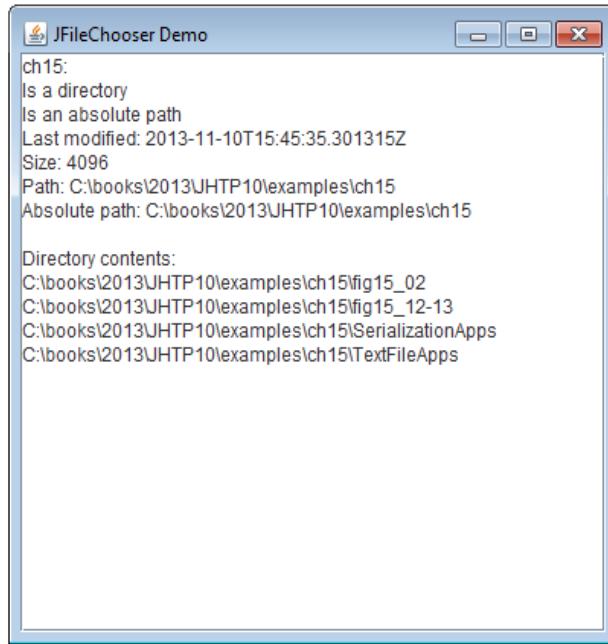


Fig. 15.13 | Testing class `FileDemonstration`. (Part 3 of 3.)

(Optional) Additional `java.io` Classes

- This section overviews additional interfaces and classes (from package `java.io`).

Interfaces and Classes for Byte- Based Input and Output

- **InputStream** and **OutputStream** are abstract classes that declare methods for performing byte-based input and output, respectively.
- **Pipes** are synchronized communication channels between threads.
 - **PipedOutputStream** (a subclass of **OutputStream**) and **PipedInputStream** (a subclass of **InputStream**) establish pipes between two threads in a program.
 - One thread sends data to another by writing to a **PipedOutputStream**.
 - The target thread reads information from the pipe via a **PipedInputStream**.

Interfaces and Classes for Byte- Based Input and Output (cont.)

- A **FilterInputStream** filters an **InputStream**, and a **FilterOutputStream** filters an **OutputStream**.
- **Filtering** means simply that the filter stream provides additional functionality, such as aggregating bytes into meaningful primitive-type units.
- **FilterInputStream** and **FilterOutputStream** are typically used as superclasses, so some of their filtering capabilities are provided by their subclasses.

Interfaces and Classes for Byte- Based Input and Output (cont.)

- A **PrintStream** (a subclass of **FilterOutputStream**) performs text output to the specified stream.
- **System.out** and **System.err** are **PrintStream** objects.

Interfaces and Classes for Byte- Based Input and Output (cont.)

- Usually, programs read data as aggregates of bytes that form `ints`, `floats`, `doubles` and so on.
- Java programs can use several classes to input and output data in aggregate form.
- Interface `DataInput` describes methods for reading primitive types from an input stream.
- Classes `DataInputStream` and `RandomAccessFile` each implement this interface to read sets of bytes and process them as primitive-type values.

Interfaces and Classes for Byte- Based Input and Output (cont.)

- Interface **DataOutput** describes a set of methods for writing primitive types to an output stream.
- Classes **DataOutputStream** (a subclass of **FilterOutputStream**) and **RandomAccessFile** each implement this interface to write primitive-type values as bytes.

Interfaces and Classes for Byte- Based Input and Output (cont.)

- **Buffering** is an I/O-performance-enhancement technique.
- With a **BufferedOutputStream**, each output operation is directed to a **buffer**
 - holds the data of many output operations
- Transfer to the output device is performed in one large **physical output operation** each time the buffer fills.
- The output operations directed to the output buffer in memory are often called **logical output operations**.
- A partially filled buffer can be forced out to the device at any time by invoking the stream object's **flush** method.
- Using buffering can greatly increase the performance of an application.



Performance Tip 15.1

Buffered I/O can yield significant performance improvements over unbuffered I/O.

Interfaces and Classes for Byte- Based Input and Output (cont.)

- With a **BufferedInputStream**, many “logical” chunks of data from a file are read as one large **physical input operation** into a memory buffer.
- As a program requests each new chunk of data, it’s taken from the buffer.
- This procedure is sometimes referred to as a **logical input operation**.
- When the buffer is empty, the next actual physical input operation from the input device is performed.

Interfaces and Classes for Byte- Based Input and Output (cont.)

- Java stream I/O includes capabilities for inputting from `byte` arrays in memory and outputting to `byte` arrays in memory.
- A `ByteArrayInputStream` (a subclass of `InputStream`) reads from a `byte` array in memory.
- A `ByteArrayOutputStream` (a subclass of `OutputStream`) outputs to a `byte` array in memory.

Interfaces and Classes for Byte- Based Input and Output (cont.)

- A `SequenceInputStream` (a subclass of `InputStream`) logically concatenates several `InputStreams`
- The program sees the group as one continuous `InputStream`.
- When the program reaches the end of one input stream, that stream closes, and the next stream in the sequence opens.

Interfaces and Classes for Character- Based Input and Output

- The **Reader** and **Writer** abstract classes are Unicode two-byte, character-based streams.
- Most of the byte-based streams have corresponding character-based concrete **Reader** or **Writer** classes.

Interfaces and Classes for Character- Based Input and Output (cont.)

- Classes **BufferedReader** (a subclass of abstract class **Reader**) and **BufferedWriter** (a subclass of abstract class **Writer**) enable buffering for character-based streams.
- Classes **CharArrayReader** and **CharArrayWriter** read and write, respectively, a stream of characters to a **char** array.
- A **LineNumberReader** (a subclass of **BufferedReader**) is a buffered character stream that keeps track of the number of lines read.

Interfaces and Classes for Character- Based Input and Output (cont.)

- An `InputStream` can be converted to a `Reader` via class **InputStreamReader**.
- An `OuputStream` can be converted to a `Writer` via class **OutputStreamWriter**.
- Class `File-Reader` and class `Filewriter` read characters from and write characters to a file.
- Class **PipedReader** and class **Pipedwriter** implement piped-character streams for transferring data between threads.
- Class **StringReader** b**Stringwriter** read characters from and write characters to `Strings`.
- A `Printwriter` writes characters to a stream.