

Files, Streams and Object Serialization

Chapter 15, Java How to Program, 10/e

15.1 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.

15.2 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

15.2 Files and Streams (cont.)

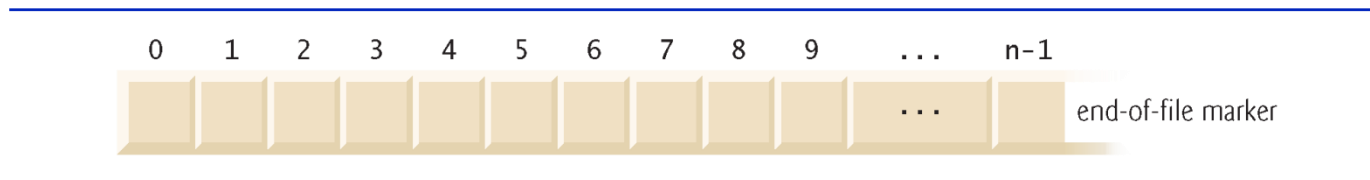
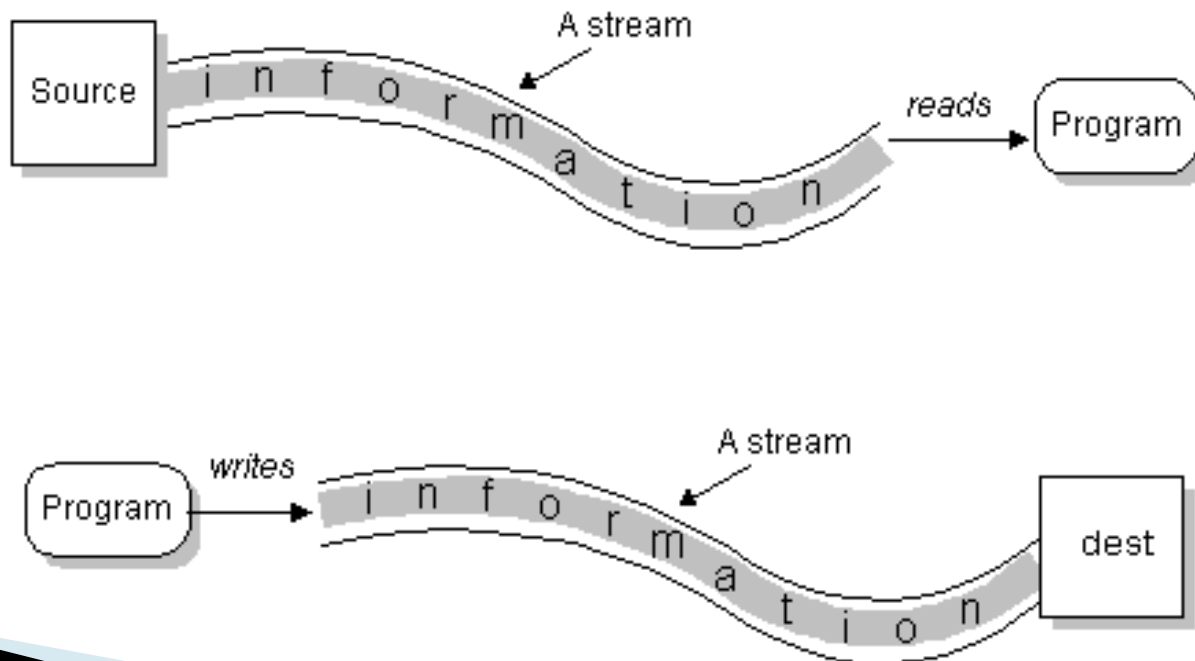


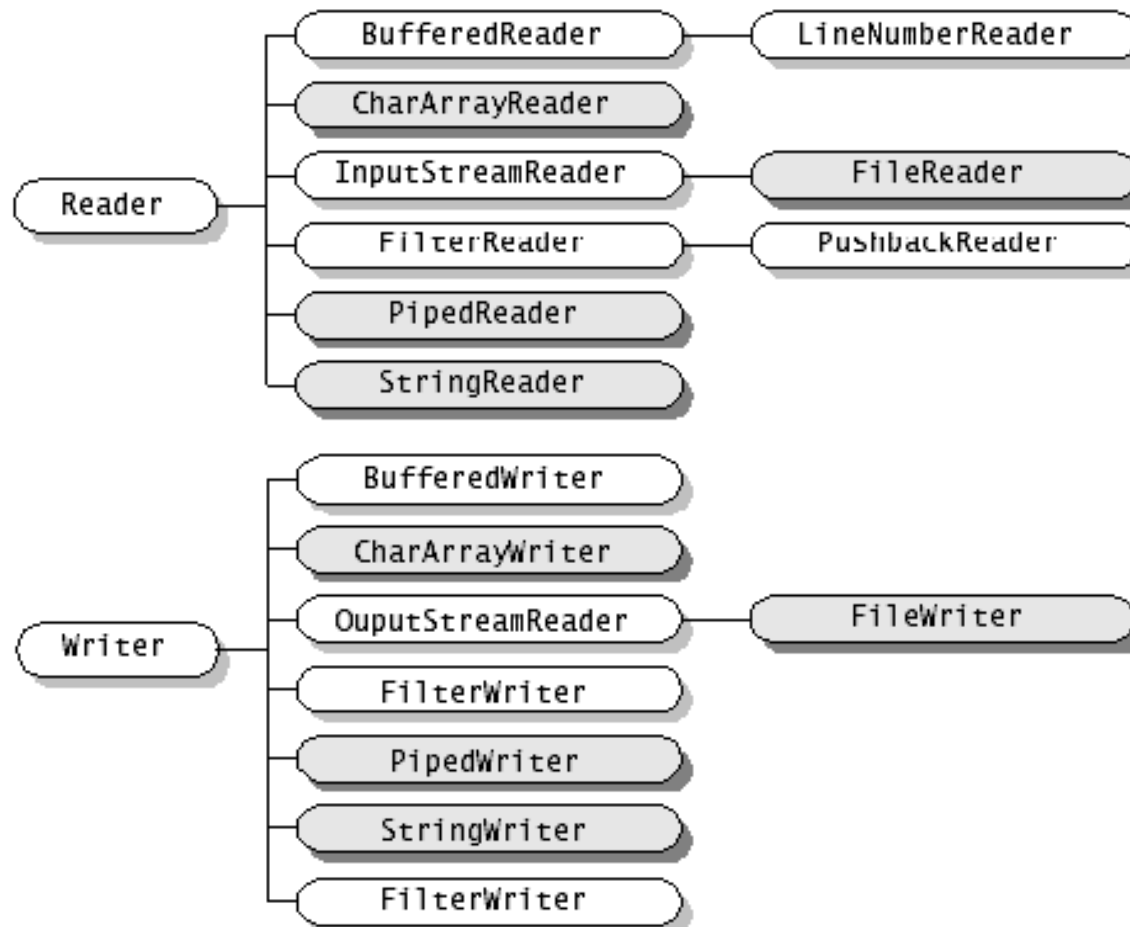
Fig. 15.1 | Java's view of a file of n bytes.



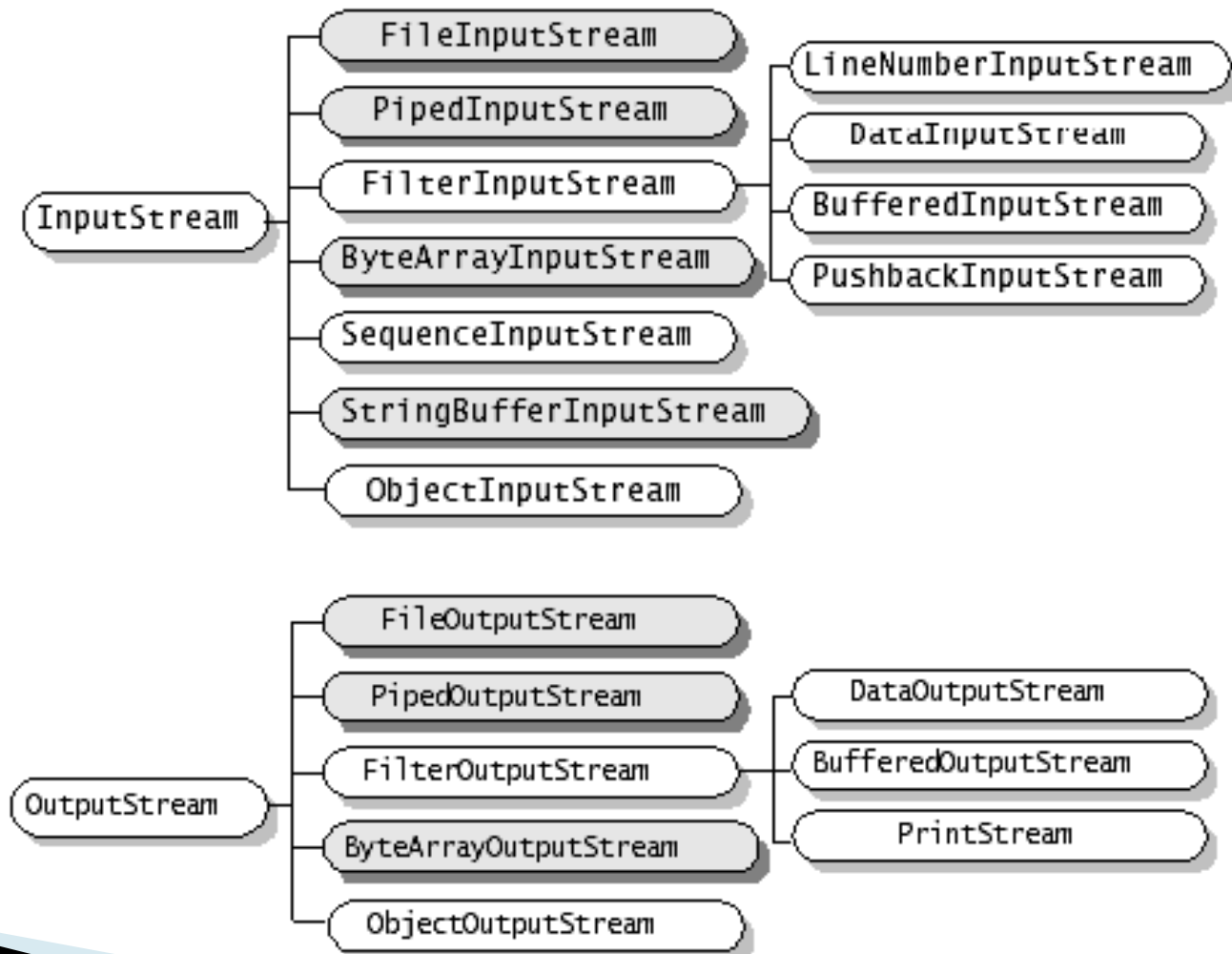
15.2 Files and Streams (cont.)

- ▶ 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.

Character Streams



Byte Streams



15.2 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.

15.2 Files and Streams (cont.)

- ▶ Java programs perform file processing by using classes from package **java.io** and the subpackages of **java.nio**.
- ▶ Includes definitions for stream classes
 - **FileInputStream** (for byte-based input from a file)
 - **FileOutputStream** (for byte-based output to a file)
 - **FileReader** (for character-based input from a file)
 - **FileWriter** (for character-based output to a file)
- ▶ You open a file by creating an object of one these stream classes. The object's constructor opens the file.

Reading Files

```
import java.io.FileReader;
import java.io.IOException;

public class ReadFileTest {

    public static void main(String[] args) throws IOException {
        FileReader fileReader = new FileReader("input.txt");

        int input;
        while ((input = fileReader.read()) != -1)
            System.out.print((char)input);
        fileReader.close();
    }
}
```




Diagram illustrating the end of file (EOF) condition. A box labeled "EOF" is connected by a line to the value `-1` in the `while` loop condition, indicating that the loop terminates when the read operation returns `-1`.

15.2 Files and Streams (cont.)

- ▶ **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`.

Reading Files (using Scanner)

- ▶ To read from a disk file, construct a `FileReader`
- ▶ Then, use the `FileReader` to construct a `Scanner` object

```
FileReader fileReader = new FileReader("input.txt");  
Scanner myScanner = new Scanner(fileReader);
```

Reading Files (using Scanner)

```
import java.io.FileReader;
import java.io.IOException;
import java.util.Scanner;

public class ReadFileScannerTest {

    public static void main(String[] args) throws IOException {
        FileReader fileReader = new FileReader("input2.txt");
        Scanner inputScanner = new Scanner(fileReader);

        while (inputScanner.hasNext())
            System.out.println(inputScanner.next());
        inputScanner.close();
        fileReader.close();
    }
}
```

Writing to Files

```
import java.io.FileWriter;
import java.io.IOException;

public class WriteFile {

    public static void main(String[] args) throws IOException {
        FileWriter fileWriter = new FileWriter("output.txt");

        fileWriter.write("This is a text! Is it?");

        fileWriter.close();
    }
}
```

Writing to Files (using PrintWriter)

```
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;

public class WriteFilePrintWriter {
    public static void main(String[] args) throws IOException {
        FileWriter fileWriter = new FileWriter("output2.txt");
        PrintWriter out = new PrintWriter(fileWriter);

        out.println("This is some text.");
        out.print("This is the second line.");

        out.close();
        fileWriter.close();
    }
}
```

Binary Read and Write

```
public static void main(String args[]) throws IOException {  
    FileInputStream in = null;  
    FileOutputStream out = null;  
  
    try {  
        in = new FileInputStream("./Pics/java.png");  
        out = new FileOutputStream("./Pics/java1.png");  
  
        int c;  
        while ((c = in.read()) != -1) {  
            out.write(c);  
        }  
    } finally {  
        if (in != null) {  
            in.close();  
        }  
        if (out != null) {  
            out.close();  
        }  
    }  
}
```


Try-with-resource

- ▶ Used for ensuring 'resources' are closed after use.
- ▶ Removes need for explicit closure on both successful and failed control flows.
- ▶ Also known as 'automatic resource management' (ARM).

Try-with-resource

- ▶ See *TryWithResource* class.

```
public static void main(String args[]) {  
    try (FileInputStream in = new FileInputStream("./Pics/java.png");  
        FileOutputStream out = new FileOutputStream("./Pics/java1.jpg"))  
    {  
        int c;  
        while ((c = in.read()) != -1) {  
            out.write(c);  
        }  
        System.out.println("Finished");  
    } catch (IOException e) {  
        e.printStackTrace();  
    }  
}
```

No close() call required in either clause.

15.3 Using NIO Classes and Interfaces to Get File and Directory Information

- ▶ Interfaces `Path` and `DirectoryStream` and classes `Paths` and `Files` (all from package `java.nio.file`) are useful for retrieving information about files and directories on disk:
 - `Path` interface—Objects of classes that implement this interface represent the location of a file or directory. `Path` objects do not open files or provide any file-processing capabilities.
 - `Paths` class—Provides static methods used to get a `Path` object representing a file or directory location.

15.3 Using NIO Classes and Interfaces to Get File and Directory Information (Cont.)

- **Files** class—Provides **static** methods for common file and directory manipulations, such as **copying** files; **creating** and **deleting** files and directories; getting **information** about files and directories; reading the **contents** of files; getting objects that allow you to manipulate the contents of files and directories; and more
- **DirectoryStream** interface—Objects of classes that implement this interface enable a program to iterate through the contents of a directory.

15.3 Using NIO Classes and Interfaces to Get File and Directory Information (Cont.)

- ▶ A file or directory's path specifies its location on disk. The path includes some or all of the directories leading to the file or directory.
- ▶ An **absolute path** contains *all* directories, starting with the **root directory**, that lead to a specific file or directory.
- ▶ Every file or directory on a particular disk drive has the *same* root directory in its path.
- ▶ A **relative path** is “relative” to another directory—for example, a path relative to the directory in which the application began executing. (e.g. `./test/t1.txt`, `./test/../code/c1.bin`)

15.3 Using NIO Classes and Interfaces to Get File and Directory Information (Cont.)

- ▶ An overloaded version of Files static method `get` uses a URI object to locate the file or directory.
- ▶ A **Uniform Resource Identifier (URI)** is a more general form of the **Uniform Resource Locators (URLs)** that are used to locate websites.
- ▶ On Windows platforms, the URI
 - `file://C:/data.txt`
- ▶ identifies the file `data.txt` stored in the root directory of the C: drive. On UNIX/Linux platforms, the URI
 - `file:/home/student/data.txt`
- ▶ identifies the file `data.txt` stored in the home directory of the user student.

15.3 Using NIO Classes and Interfaces to Get File and Directory Information (Cont.)

- ▶ Figure 15.2 prompts the user to enter a file or directory name, then uses classes `Paths`, `Path`, `Files` and `DirectoryStream` to output information about that file or directory.

15.3 Using NIO Classes and Interfaces to Get File and Directory Information (Cont.)

- ▶ A **separator character** is used to separate directories and files in a path.
 - On a Windows computer, the *separator character* is a backslash (\).
 - On a Linux or Mac OS X system, it's a forward slash (/).
- ▶ Java processes both characters identically in a path name.
- ▶ For example, if we were to use the path
 - `c:\Program Files\Java\jdk1.6.0_11\demo/jfc`
- ▶ which employs each separator character, Java would still process the path properly.

15.3 Using NIO Classes and Interfaces to Get File and Directory Information (Cont.)

- ▶ See the *FileAndDirectoryInfo* class.

```
Enter file or directory name:
c:\examples\ch15

ch15 exists
Is a directory
Is an absolute path
Last modified: 2013-11-08T19:50:00.838256Z
Size: 4096
Path: c:\examples\ch15
Absolute path: c:\examples\ch15

Directory contents:
C:\examples\ch15\fig15_02
C:\examples\ch15\fig15_12_13
C:\examples\ch15\SerializationApps
C:\examples\ch15\TextFileApps
```

Fig. 15.2 | File class used to obtain file and directory information. (Part 4 of 5.)

15.3 Using NIO Classes and Interfaces to Get File and Directory Information (Cont.)

- ▶ See the *FileAndDirectoryInfo* class.

```
Enter file or directory name:  
C:\examples\ch15\fig15_02\FileAndDirectoryInfo.java  
  
FileAndDirectoryInfo.java exists  
Is not a directory  
Is an absolute path  
Last modified: 2013-11-08T19:59:01.848255Z  
Size: 2952  
Path: C:\examples\ch15\fig15_02\FileAndDirectoryInfo.java  
Absolute path: C:\examples\ch15\fig15_02\FileAndDirectoryInfo.java
```

Fig. 15.2 | File class used to obtain file and directory information. (Part 5 of 5.)

15.4 Sequential-Access Text Files

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

15.4.1 Creating a Sequential-Access Text File

- ▶ `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**.
- ▶ See *`formattedTextFileIO`* method in the *`FileIO`* class.

15.4.1 Creating a Sequential-Access Text File (cont.)

Enter: first name, last name, student ID, grade

ehsan

edalat

9031066

12.25

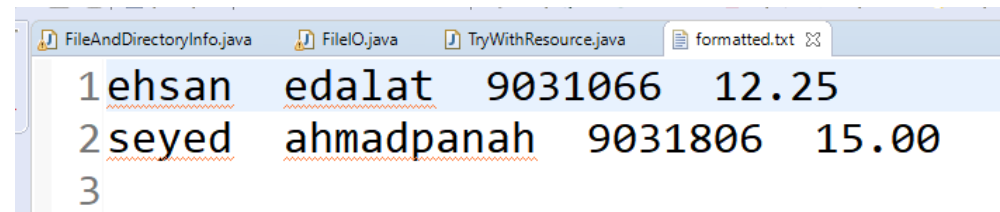
seyed

ahmadpanah

9031806

15

Formatted text read/write done!



```
FileAndDirectoryInfo.java  FileIO.java  TryWithResource.java  formatted.txt  X
1 ehsan edalat 9031066 12.25
2 seyed ahmadpanah 9031806 15.00
3
```

15.4.1 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.

15.4.1 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.

15.4.2 Reading Data from a Sequential-Access Text File

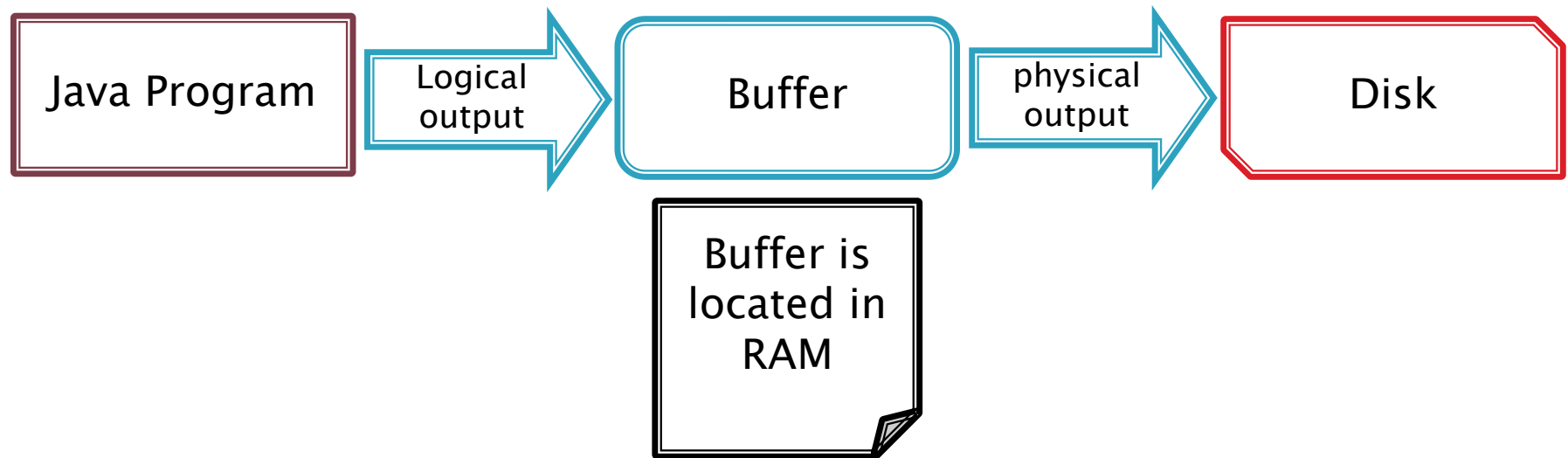
- ▶ The method *scannerTextFileIO* reads records from the file "formatted.txt" created by the method *formattedTextFileIO* and displays the record contents.

```
ehsan    edalat    9031066 12.250000  
seyed    ahmadpanah      9031806 15.000000  
Finished reding from formatted
```


15.7.1 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.

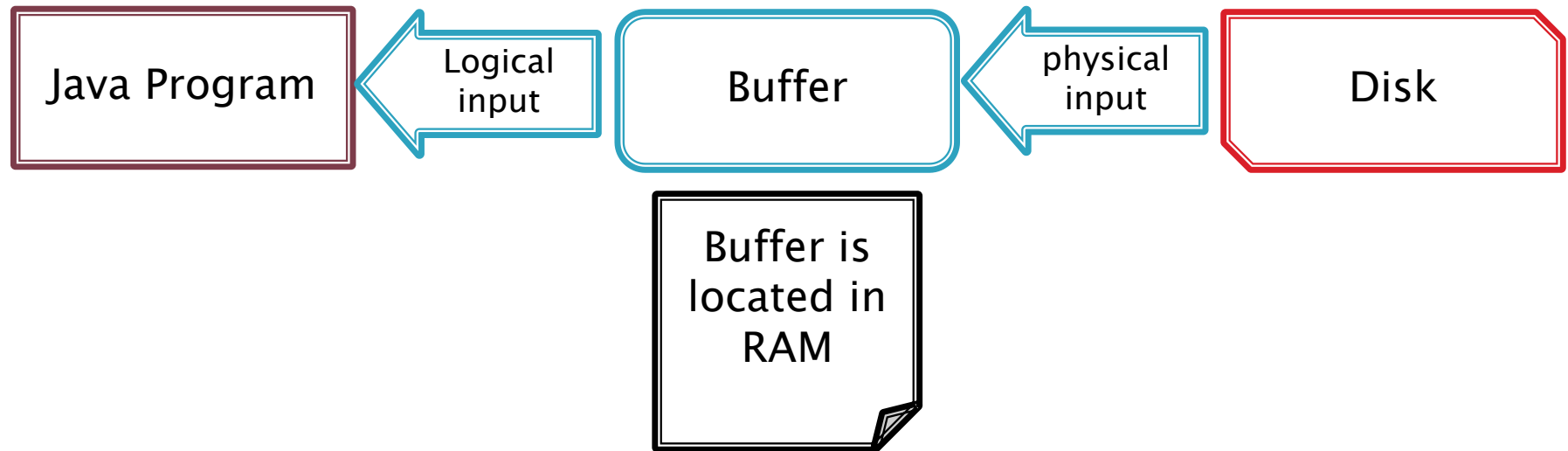
15.7.1 Interfaces and Classes for Byte-Based Input and Output (cont.)



15.7.1 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.

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



Reading and Writing with bufferedInput/Output Streams

- ▶ See the method *copyFile* which demonstrates efficient binary read/write operations for files.

Reading and Writing with bufferedReader/Writer

- ▶ See the method *bufferedTextFileIO* which demonstrates efficient read/write operations for large text files.

15.4.4 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 “ali” needed to be changed to “ehsan” 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.

Any problem?

- ▶ Java can perform input and output of objects or variables of primitive data types without having to worry about the details of converting such values to byte format.
- ▶ To perform such input and output, objects of classes **ObjectInputStream** and **ObjectOutputStream** can be used together with the **byte-based file stream classes** **FileInputStream** and **FileOutputStream**.

15.5 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.

15.5 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.

15.5 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.

15.5.1 Creating a Sequential-Access File Using Object Serialization

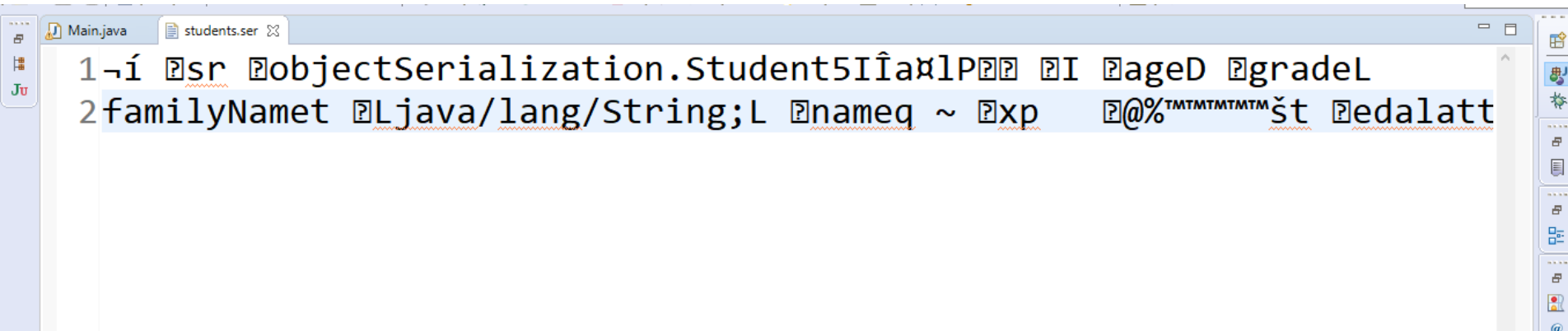
- ▶ Objects of classes that implement interface **Serializable** can be *serialized* and *deserialized* with `ObjectOutputStreams` and `ObjectInputStreams`.
- ▶ Interface `Serializable` is a **tagging interface**.
 - It does not contain methods.
- ▶ A class that implements `Serializable` is *tagged*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

15.5.1 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`.

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

- ▶ See code in *objectSerialization* package.
- ▶ File content is not human readable!



```
1-í sr objectSerialization.Student5IÎalP I ageD gradeL
2familyName Ljava/lang/String;L nameq ~ xp @%TMTMTMTMTMš edalatt
```