Based on Chapter 15
Files, Streams and Object Serialization

Java How to Program

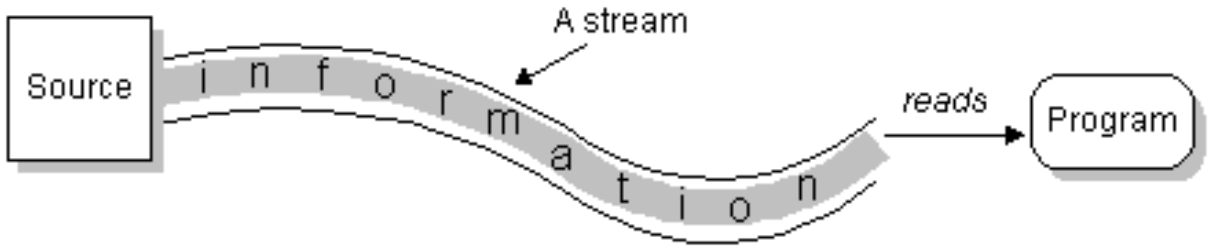
Introduction

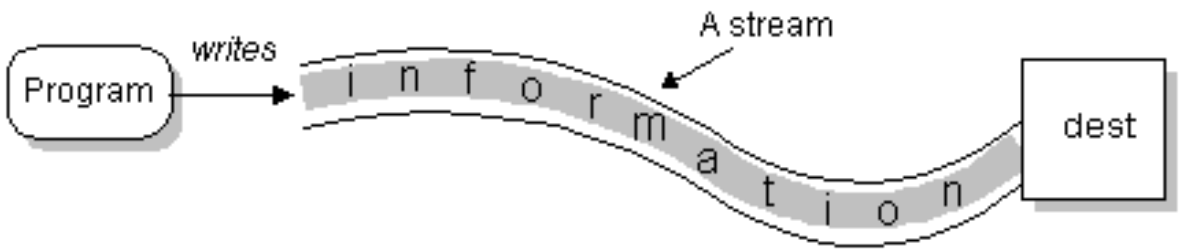
- Data stored in variables and arrays is temporary
  - It's lost when a local variable goes out of scope or when the program terminates
- For long-term retention of data, computers use **files.**
- Computers store files on **secondary storage devices**
  - hard disks, optical disks, flash drives and magnetic tapes.
- 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. 17.1).
- A Java program simply receives an indication from the operating system when it reaches the end of the stream

Streams

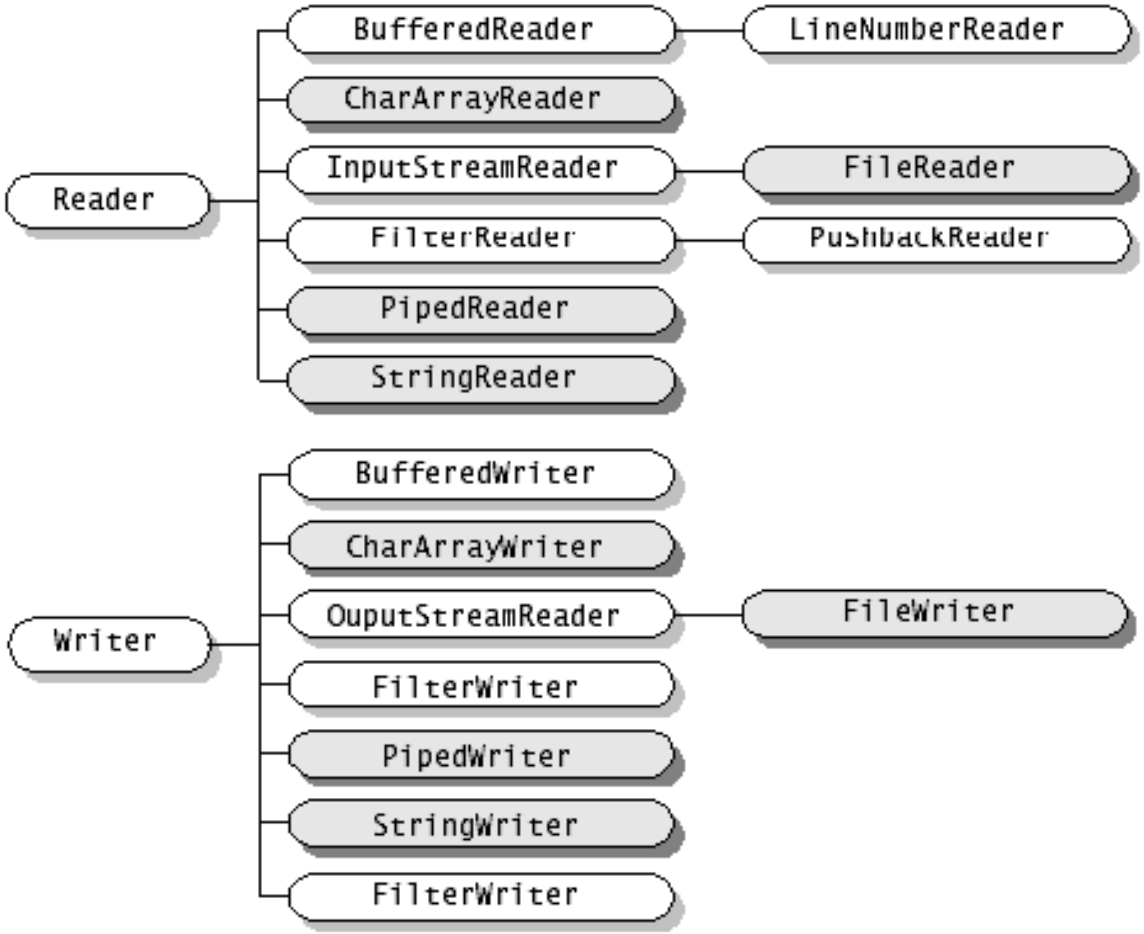

A stream
Source
i n f o r m a t i o n
reads
Program


A stream
Program
writes
i n f o r m a t i o n
dest

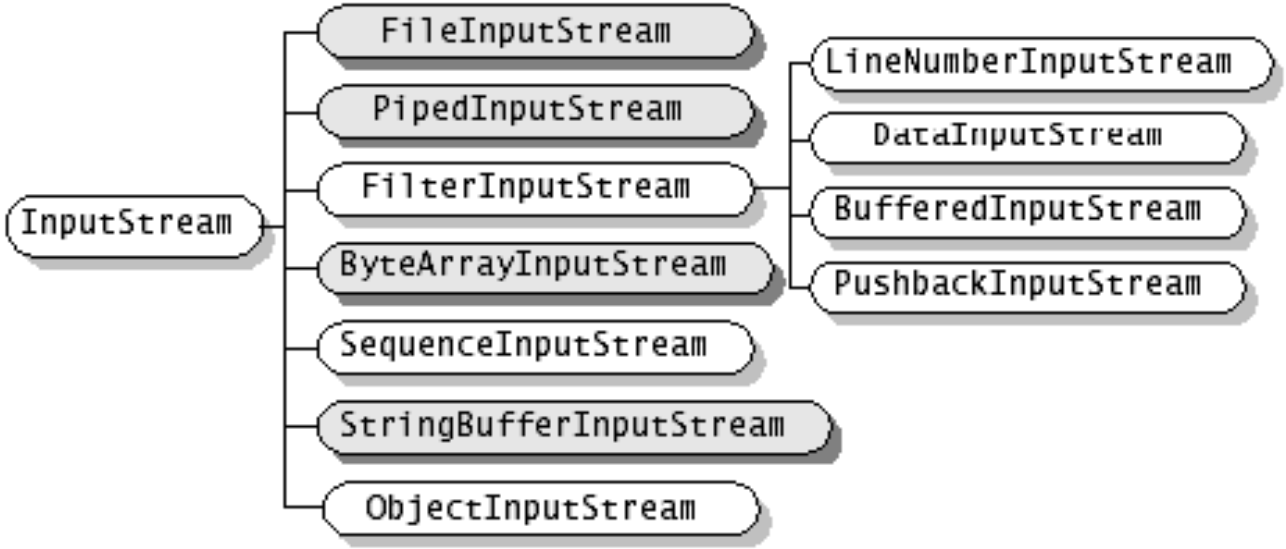
Files and Streams (cont.)

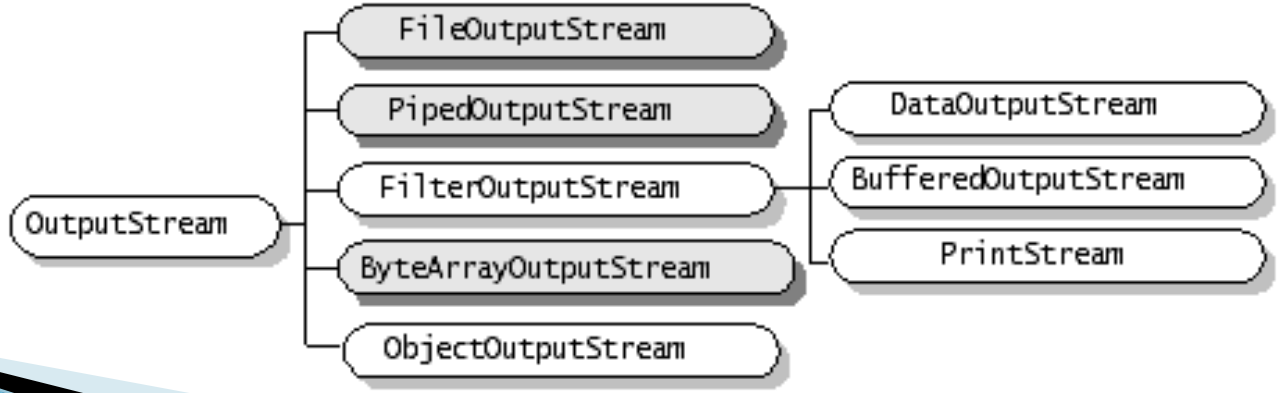
- File streams can be used to input and output data as bytes or characters.
- Streams that input and output bytes are known as **byte-based streams**, representing data in its binary format.
- Streams that input and output characters are known as **character-based streams**, representing data as a sequence of characters.
- Files that are 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





Files and Streams (cont.)

- Java programs perform file processing by using classes from package **java.io.**
- 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();
    }
}
```

Files and Streams (cont.)

- Character-based input and output can be performed with class **Formatter**.
  - Class `Formatter` enables formatted data to be output to any text-based stream in a manner similar to method `System.out.printf`.
  - See formattedTextFileIO method in the code.

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.print(inputScanner.next());
        inputScanner.close();
        fileReader.close();
    }
}
```

Interfaces and Classes for Byte-Based Input and Output

- **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.

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.

Class `File`

- Class `File` provides four constructors.
- The one with a `String` argument specifies the `name` of a file or directory to associate with the `File` object.
  - The `name` can contain **path information** as well as a file or directory name.
  - A file or directory's path specifies its location on disk.
  - An **absolute path** contains all the directories, starting with the **root directory**, that lead to a specific file or directory.
  - A **relative path** normally starts from the directory in which the application began executing and is therefore "relative" to the current directory.

Method	Description
`boolean canRead()`	Returns `true` if a file is readable by the current application; `false` otherwise.
`boolean canWrite()`	Returns `true` if a file is writable by the current application; `false` otherwise.
`boolean exists()`	Returns `true` if the file or directory represented by the `File` object exists; `false` otherwise.
`boolean isFile()`	Returns `true` if the name specified as the argument to the `File` constructor is a file; `false` otherwise.
`boolean isDirectory()`	Returns `true` if the name specified as the argument to the `File` constructor is a directory; `false` otherwise.
`boolean isAbsolute()`	Returns `true` if the arguments specified to the `File` constructor indicate an absolute path to a file or directory; `false` otherwise.

Method	Description
`String getAbsolutePath()`	Returns a `String` with the absolute path of the file or directory.
`String getName()`	Returns a `String` with the name of the file or directory.
`String getPath()`	Returns a `String` with the path of the file or directory.
`String getParent()`	Returns a `String` with the parent directory of the file or directory (i.e., the directory in which the file or directory is located).
`long length()`	Returns the length of the file, in bytes. If the `File` object represents a directory, an unspecified value is returned.
`long lastModified()`	Returns a platform-dependent representation of the time at which the file or directory was last modified. The value returned is useful only for comparison with other values returned by this method.
`String[] list()`	Returns an array of `String`s representing a directory's contents. Returns `null` if the `File` object does not represent a directory.

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`, 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 with `FileInputStream` and `FileOutputStream` objects.

Object Serialization (cont.)

- `ObjectOutput` interface method **writeObject** takes an `Object` as an argument and writes its information to an `OutputStream`.
- A class that implements `ObjectOuput` (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 `ObjectOutputStream`s and `ObjectInputStream`s.
- 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.

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`.