Lecture 22

Java

Input/Output (I/O) Streams

Dr. Martin O'Connor
CA166
www.computing.dcu.ie/~moconnor

Topics

- I/O Streams
- Writing to a Stream
- Byte Streams
- Character Streams
- Line-Oriented I/O
- Buffered I/O Streams
- Scanning and Formatting

I/O Streams

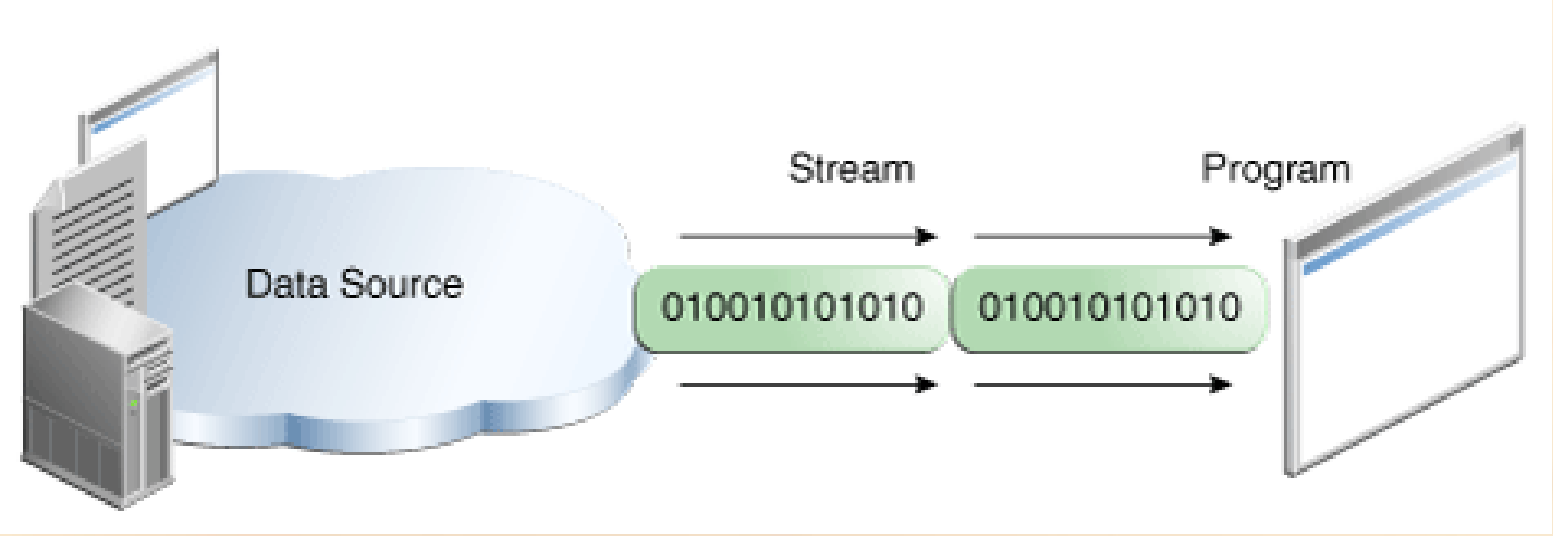
- An *I/O Stream* represents an input source or an output destination
- A stream can represent many different kinds of sources and destinations, including:
  - disk files,
  - peripheral devices,
  - other programs,
  - network sockets

I/O Streams

- Streams support many different kinds of data, including simple bytes, primitive data types, localized characters, and objects
- Some streams simply pass on data; others manipulate and transform the data in useful ways
- No matter how they work internally, all streams present the same simple model to programs that use them

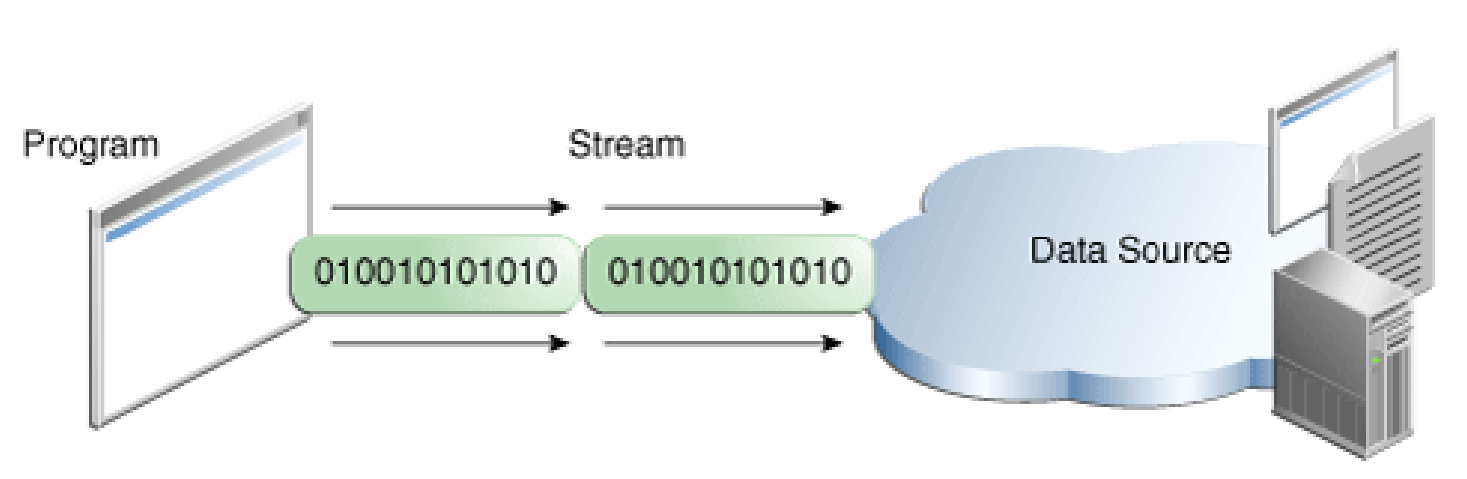
Reading from a Stream

- A stream is a sequence of data
- A program uses an *input stream* to read data from a source, one item at a time



Writing to a Stream

- A program uses an *output stream* to write data to a destination, one item at time



- The data source and data destination can be anything that holds, generates, or consumes data
- These includes disk files, another program, a peripheral device, or a network socket

Byte Streams

- Programs use *byte streams* to perform input and output of 8-bit bytes
- All byte stream classes are descended from the **InputStream** and **OutputStream** classes
- There are many byte stream classes
- We'll focus on the file I/O byte streams, **FileInputStream** and **FileOutputStream**
- Other kinds of byte streams are used in much the same way; they differ mainly in the way they are constructed

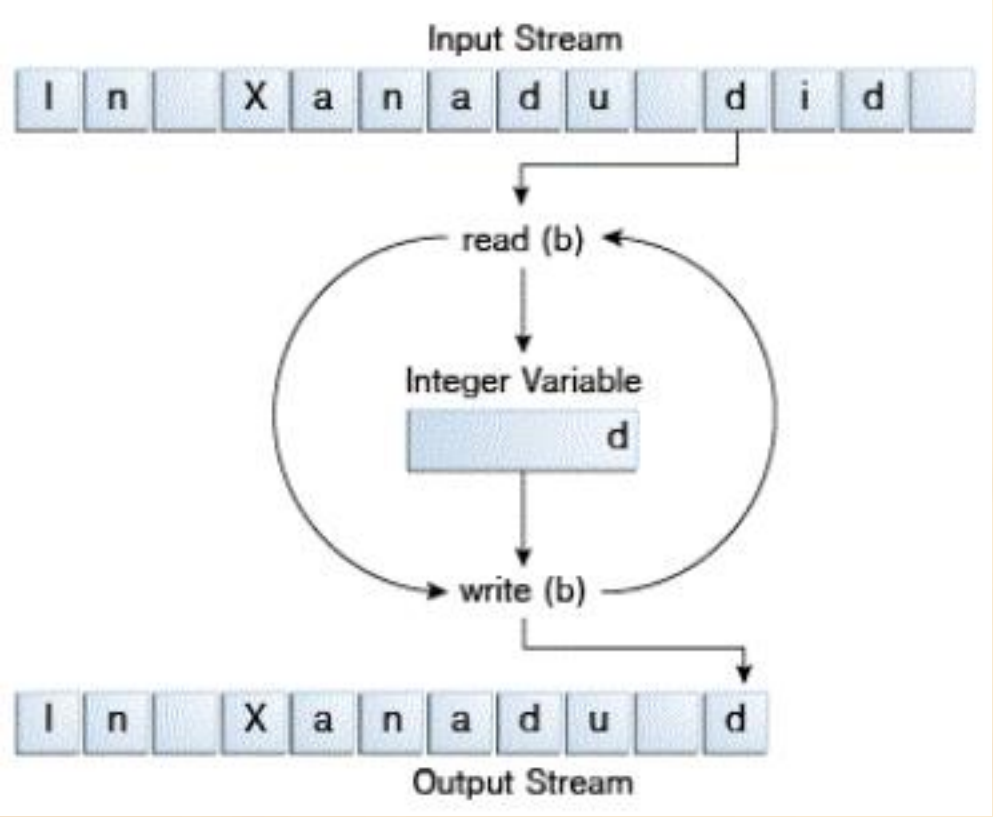
Byte Streams

- A program named CopyBytes which uses byte streams to copy a textfile one byte at a time

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class CopyBytes {
    public static void main(String[] args) throws IOException {

        FileInputStream in = null;
        FileOutputStream out = null;

        try {
            in = new FileInputStream("xanadu.txt");
            out = new FileOutputStream("outagain.txt");
            int c;

            while ((c = in.read()) != -1) {
                out.write(c);
            }
        } finally {
            if (in != null) {
                in.close();
            }
            if (out != null) {
                out.close();
            }
        }
    }
}
```

Byte Streams

- CopyBytes spends most of its time in a simple loop that reads the input stream and writes the output stream, one byte at a time



Byte Streams

- Notice that read() returns an int value
- If the input is a stream of bytes, why doesn't read() return a byte value?

Byte Streams

- Answer: Using a int as a return type allows read() to use -1 to indicate that it has reached the end of the stream

Common Error Trap

- NOTE: Closing a stream when it's no longer needed is very important
- CopyBytes uses a finally block to guarantee that both streams will be closed even if an error occurs. This practice helps avoid serious resource leaks

Byte Streams

- Byte Streams are a low-level I/O which in practice, you will not use often
- For files with character data, you will use character streams.
- There are also streams for more complicated data types. Thus, Byte streams should only be used for the most primitive I/O
- So why talk about byte streams? Because all other stream types are built on byte streams

Character Streams

- The Java platform stores character values using Unicode conventions
- Character stream I/O automatically translates this internal format to and from the local character set
- In Western locales, the local character set is usually an 8-bit superset of ASCII
- For most applications, I/O with character streams is no more complicated than I/O with byte streams
- If internationalization isn't a priority, you may use the character stream classes without paying much attention to character set issues.

Using Character Streams

- All character stream classes are descended from the **Reader** and **Writer** classes
- As with byte streams, there are character stream classes that specialize in file I/O: **FileReader** and **FileWriter**
- The program CopyCharacters illustrates these classes.

Using Character Streams

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

public class CopyCharacters {
    public static void main(String[] args) throws IOException {

        FileReader inputStream = null;
        FileWriter outputStream = null;

        try {
            inputStream = new FileReader("xanadu.txt");
            outputStream = new FileWriter("characteroutput.txt");

            int c;
            while ((c = inputStream.read()) != -1) {
                outputStream.write(c);
            }
        } finally {
            if (inputStream != null) {
                inputStream.close();
            }
            if (outputStream != null) {
                outputStream.close();
            }
        }
    }
}
```

Using Character Streams

- CopyCharacters is very similar to CopyBytes
- The most important difference is that CopyCharacters uses FileReader and FileWriter for input and output in place of FileInputStream and FileOutputStream
- Notice that both CopyBytes and CopyCharacters use an int variable to read to and write from
- However, in CopyCharacters, the int variable holds a character value in its last 16 bits; in CopyBytes, the int variable holds a byte value in its last 8 bits. Why?

Using Character Streams

- Example: Reading the characters in from a file and printing them to screen.

- Note: in.read() returns an int. To obtain the char value from the int, use a type cast:

```
inputChar = (char) c;
```

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

public class PrintCharactersInFile {
    public static void main(String[] args) throws IOException {

        FileReader in = null;

        try {
            in = new FileReader("xanadu.txt");

            int c;
            char inputChar;
            while ((c = in.read()) != -1) {
                inputChar = (char) c;
                System.out.print(inputChar);
            }
        } finally {
            if (in != null) {
                in.close();
            }
        }
    }
}
```

Using Character Streams

- Character streams are often "wrappers" for byte streams
- The character stream uses the byte stream to perform the physical I/O, while the character stream handles translation between characters and bytes
- **FileReader**, for example, uses **FileInputStream**, while **FileWriter** uses **FileOutputStream**.
- Note: A **FileWriter** object can be instantiated from a File object (An abstract representation of file and directory pathnames)

Using Character Streams

- FileWriter has many constructors

Following syntax creates a FileWriter object given a File object.

```
FileWriter(File file)
```

Following syntax creates a FileWriter object given a File object.

```
FileWriter(File file, boolean append)
```

Following syntax creates a FileWriter object associated with a file descriptor.

```
FileWriter(FileDescriptor fd)
```

Following syntax creates a FileWriter object given a file name.

```
FileWriter(String fileName)
```

Following syntax creates a FileWriter object given a file name with a boolean indicating whether or not to append the data written.

```
FileWriter(String fileName, boolean append)
```

Line-Oriented I/O

- Character I/O usually occurs in bigger units than single characters
- One common unit is the line: a string of characters with a line terminator at the end.
- A line terminator can be a carriage-return/line-feed sequence ("\r\n"), a single carriage-return ("\r"), or a single line-feed ("\n")
- Supporting all possible line terminators allows programs to read text files created on any of the widely used operating systems

Line-Oriented I/O

- Let's modify the CopyCharacters program to use line-oriented I/O
- we need to use two new classes: **BufferedReader** and **PrintWriter**
- The CopyLines program on the next slide invokes *BufferedReader.readLine* and *PrintWriter.println* to do input and output one line at a time

Line-Oriented I/O

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

public class CopyLines {
    public static void main(String[] args) throws IOException {

        BufferedReader inputStream = null;
        PrintWriter outputStream = null;

        try {
            inputStream = new BufferedReader(new FileReader("xanadu.txt"));
            outputStream = new PrintWriter(new FileWriter("characteroutput.txt"));

            String l;
            while ((l = inputStream.readLine()) != null) {
                outputStream.println(l);
            }
        } finally {
            if (inputStream != null) {
                inputStream.close();
            }
            if (outputStream != null) {
                outputStream.close();
            }
        }
    }
}
```

Line-Oriented I/O

- Invoking inputStream.readLine() reads in a line of text from the input file
- Invoking outputStream.println() writes out a line of text to the output file and appends the line terminator for the current operating system
- NOTE: the line terminator for the current operating system might not be the same line terminator that was used in the input file

Buffered I/O Streams

- Most of the examples we've seen so far use *unbuffered* I/O
- This means each read or write request is handled directly by the underlying OS
- This makes a program much less efficient, since each such request often triggers disk access, network activity, or some other operation that is relatively expensive compared to memory-based processing.
- To reduce this kind of overhead, the Java platform implements *buffered* I/O streams

Buffered I/O Streams

- Buffered input streams read data from a memory area known as a *buffer*; the native input API is called only when the buffer is empty
- Similarly, buffered output streams write data to a buffer, and the native output API is called only when the buffer is full
- A program can convert an unbuffered stream into a buffered stream using the wrapping mechanism we've used several times now, where the unbuffered stream object is passed to the constructor for a buffered stream class

Buffered I/O Streams

- Lets modify the constructor invocations in the CopyCharacters example to use buffered I/O

```
inputStream = new BufferedReader(new FileReader("xanadu.txt"));
outputStream = new BufferedWriter(new FileWriter("characteroutput.txt"));
```

- There are four buffered stream classes used to wrap unbuffered streams:
  - **BufferedInputStream** and **BufferedOutputStream** create buffered byte streams
  - **BufferedReader** and **BufferedWriter** create buffered character streams

Flushing Buffered I/O Streams

- It often makes sense to write out a buffer at critical points, without waiting for it to fill. This is known as *flushing* the buffer
- Some buffered output classes support *autoflush*, specified by an optional constructor argument
- When autoflush is enabled, certain key events cause the buffer to be flushed
- For example, an autoflush **PrintWriter** object flushes the buffer on every invocation of the println or format methods
- To flush a stream manually, invoke its flush method

Scanning and Formatting

- Programming I/O often involves translating to and from the neatly formatted data humans like to work with
- To assist you with these chores, the Java platform provides two APIs
  - The **scanner** API breaks input into individual tokens associated with bits of data
  - The **formatting** API assembles data into nicely formatted, human-readable form.

Scanning

- Objects of type **Scanner** are useful for breaking down formatted input into tokens and translating individual tokens according to their data type

- By default, a scanner uses white space to separate tokens

- White space characters include blanks, tabs, and line terminators. For the full list, refer to the documentation for **Character.isWhitespace**.

- The following program ScanXan reads the individual words of a textfile and prints them out, one per line

Scanning

- ScanXan.java

```
import java.io.*;
import java.util.Scanner;

public class ScanXan {
    public static void main(String[] args) throws IOException {

        Scanner s = null;

        try {
            s = new Scanner(new BufferedReader(new FileReader("xanadu.txt")));

            while (s.hasNext()) {
                System.out.println(s.next());
            }
        } finally {
            if (s != null) {
                s.close();
            }
        }
    }
}
```

Scanning

- Notice that ScanXan invokes Scanner's close method when it is done with the scanner object

- Even though a scanner is not a stream, you need to close it to indicate that you're done with its underlying stream

- Input file: xanadu.txt

```
In Xanadu did Kubla Khan
A stately pleasure-dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
Down to a sunless sea.
```

Output to Screen

```
In
Xanadu
did
Kubla
Khan
A
stately
pleasure-dome
...
```

Scanning

- To use a different token separator, invoke useDelimiter(), specifying a regular expression

- For example, suppose you wanted the token separator to be a comma, optionally followed by white space. You would invoke

```
s.useDelimiter(",\\s*");
```

- Scanner also supports tokens for all of the Java language's primitive types (except for char), as well as BigInteger and BigDecimal

Formatting

- Recall: the formatting API assembles data into nicely formatted, human-readable form.

- Stream objects that implement formatting are instances of either **PrintWriter**, a character stream class, or **PrintStream**, a byte stream class

- **Note:** The only PrintStream objects you are likely to need are **System.out** and **System.err**.

- When you need to create a formatted output stream, instantiate **PrintWriter**, not **PrintStream**. Why?

Formatting

- Both **PrintStream** and **PrintWriter** implement the same set of methods for converting internal data into formatted output
- Two levels of formatting are provided:
  - print and println format individual values in a standard way
  - format formats almost any number of values based on a format string, with many options for precise formatting

Formatting

- Invoking print or println outputs a single value after converting the value using the appropriate toString method
- The format method formats multiple arguments based on a *format string*
- The format string consists of static text embedded with *format specifiers*;
- With the exception of the format specifiers, the format string is unchanged in the output (an example on next slide)
- Format strings support many features, here we cover just a few
- For a complete description: visit:
  http://docs.oracle.com/javase/7/docs/api/java/util/Formatter.html#syntax

Formatting

- An Example of a formatted string:

```
public class Root2 {
    public static void main(String[] args) {
        int i = 2;
        double r = Math.sqrt(i);

        System.out.format("The square root of %d is %f.%n", i, r);
    }
}
```

Here is the output:

```
The square root of 2 is 1.414214.
```

- All format specifiers begin with a % and end with a 1- or 2-character *conversion* that specifies the kind of formatted output being generated
  - *d* formats an integer value as a decimal value
  - *f* formats a floating point value as a decimal value
  - *n* outputs a platform-specific line terminator

Formatting

- Some other conversions
  - *s* formats any value as a string
  - *x* formats an integer as a hexadecimal value
  - *tB* formats an integer as a locale-specific month name

- **Note:** Except for %% and %n, all format specifiers must match an argument. If they don't, an exception is thrown

Formatting

- In addition to the conversion, a format specifier can contain several additional elements that further customize the formatted output.
- Here's an example, **Format**, that uses every possible kind of element

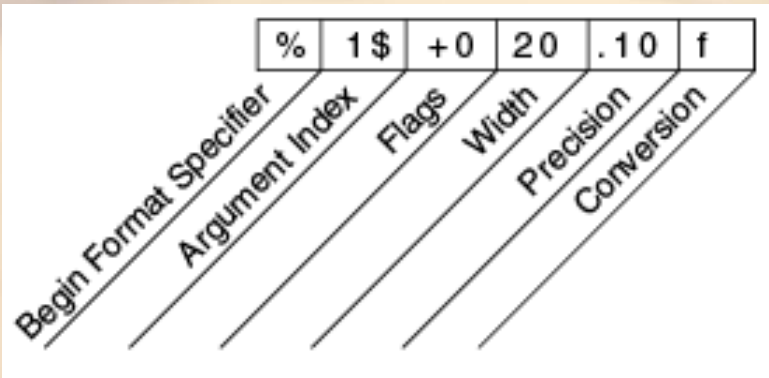
```
public class Format {
    public static void main(String[] args) {
        System.out.format("%f, %1$+020.10f %n", Math.PI);
    }
}
```

Here's the output:

```
3.141593, +00000003.1415926536
```

Formatting

- An explanation of the longer specifier is as follows:



- A complete description of the above is provided at:
  http://docs.oracle.com/javase/tutorial/essential/io/formatting.html

Acknowledgement

- The slides are based (in part) on the **I/O Streams** lesson of the Online Oracle Java Tutorial

  http://docs.oracle.com/javase/tutorial/essential/io/streams.html