File I/O

CPSC 1181 – O.O.

Jeremy Hilliker

Summer 2017

Langara.

THE COLLEGE OF HIGHER LEARNING.

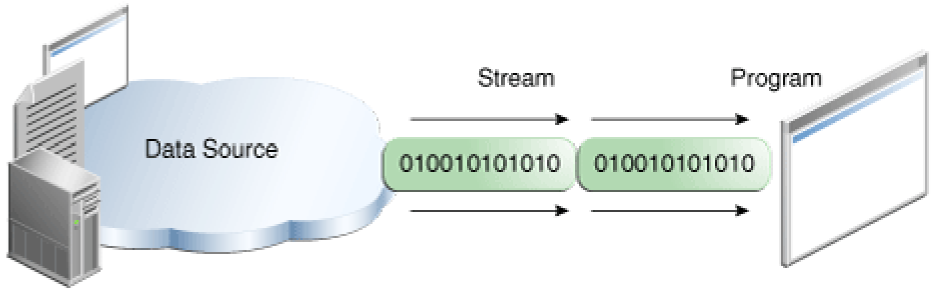
Overview

- java.io
  - I/O streams
  - Binary Streams
  - Character Streams
    - Readers/Writers
  - Wrappers
    - Buffers
    - Scanner
    - PrintWriter
  - Marshalling
    - Data Streams
    - Object Streams
- java.nio
  - Channels & Buffers*
    - Non-blocking I/O
- Stream Streams

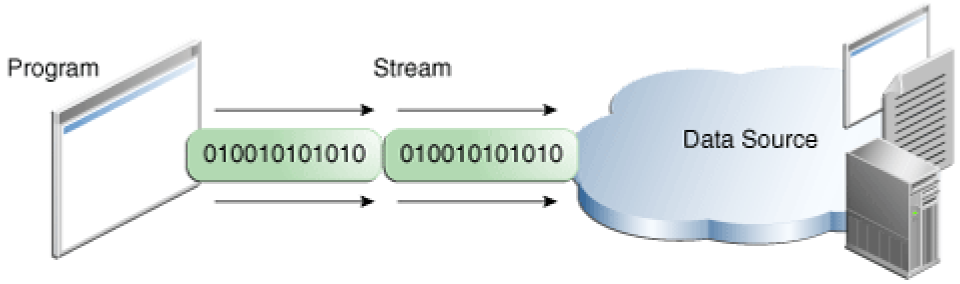
Stream

- Def'n: a ***stream***
  - is a <u>sequence</u> of elements (data / bytes) made available over time
- Conceptually:
  - <u>Continuous</u> stream of data (like a river)
  - One way
  - No delineation of when one set of data begins or ends
    - Except for when the entire stream ends

I/O Streams (java.io.*)



InputStream



OutputStream

https://docs.oracle.com/javase/tutorial/essential/io/index.html

CopyBytes

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

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

        try (InputStream in = new FileInputStream("input.bin");
             OutputStream out = new FileOutputStream("output.bin")) {

             int len;
             byte[] bytes = new byte[4096];
             while ((len = in.read(bytes)) != -1) {
                  out.write(bytes, 0, len);
             }
        }
    }
}
```

Note: always close streams

Character Streams

- java.io.Reader & java.io.Writer
- Used for handling text data for I/O
- Converts Java’s internal format from / to
  - A specified character encoding
  - Or the machine’s default character encoding
  - InputStreamReader
  - OutputStreamWriter

CopyCharacters

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

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

        try (Reader in = new FileReader("input.txt");
             Writer out = new FileWriter("output.txt”) {

            int len;
            char[] chars = new char[4096];
            while ((len = in.read(chars)) != -1) {
                out.write(chars, 0, len);
            }
        }
    }
}
```

Note: always close your readers and writers

Buffered I/O

- The streams so far are un-buffered
- Every read / write operation is passed to the OS
    - Requires a context switch
        - (from User Mode to Kernel Mode & back)
    - Fairly expensive
- If we aren't writing in chunks (we were), would like a abstraction to do buffering for us
    - BufferedInputStream
    - BufferedOutputStream
    - BufferedReader (can read lines)
    - BufferedWriter
- Data is buffered, so may have to flush() a write

java.util.Scanner

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

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

        Scanner s = null;
        double sum = 0;

        try (Scanner s = new Scanner(
               new BufferedReader(new FileReader("usnumbers.txt")))) {
            s.useLocale(Locale.US);
            while (s.hasNext()) {
                if (s.hasNextDouble()) {
                    sum += s.nextDouble();
                } else {
                    s.next();
                }
            }
        }
        System.out.println(sum);
    }
}
```

Scanners can use Regular Expressions

```
String input = "1 fish 2 fish red fish blue fish";
Scanner s = new
Scanner(input).useDelimiter("\\s*fish\\s*");
System.out.println(s.nextInt());
System.out.println(s.nextInt());
System.out.println(s.next());
System.out.println(s.next());
s.close();
```

```
1
2
red
blue
```

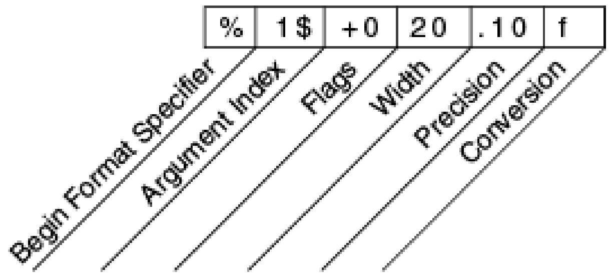
```
String input = "1 fish 2 fish red fish blue fish";
Scanner s = new Scanner(input);
s.findInLine("(\\d+) fish (\\d+) fish (\\w+) fish (\\w+)");
MatchResult result = s.match();
for (int i=1; i<=result.groupCount(); i++)
       System.out.println(result.group(i));
s.close();
```

java.io.PrintWriter : print & println

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

        System.out.print("The square root of ");
        System.out.print(i);
        System.out.print(" is ");
        System.out.print(r);
        System.out.println(".");

        i = 5;
        r = Math.sqrt(i);
        System.out.println("The square root of “
            + i + " is " + r + ".");
    }
}
```

java.io.PrintWriter : format

```
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);
    }
}

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



Marshalling

- Def’n: Marshalling is
  - The process of transforming the in-memory representation of data into a common format suitable for transmission or storage
  - Not all computer store things in memory the same way
    - Order of bytes: Big-Endian vs Little-Endian
    - Encoding of characters: ASCII vs UTF-8 vs UTF-16
- Part of the presentation* layer

Marshalling

- In general:
  - Use Streams for binary data
  - Wrap them in Reader / Writer for string / character data
- Use higher level abstractions where appropriate
  - Data Streams
  - Object Streams
  - REST

java.io.DataOutput

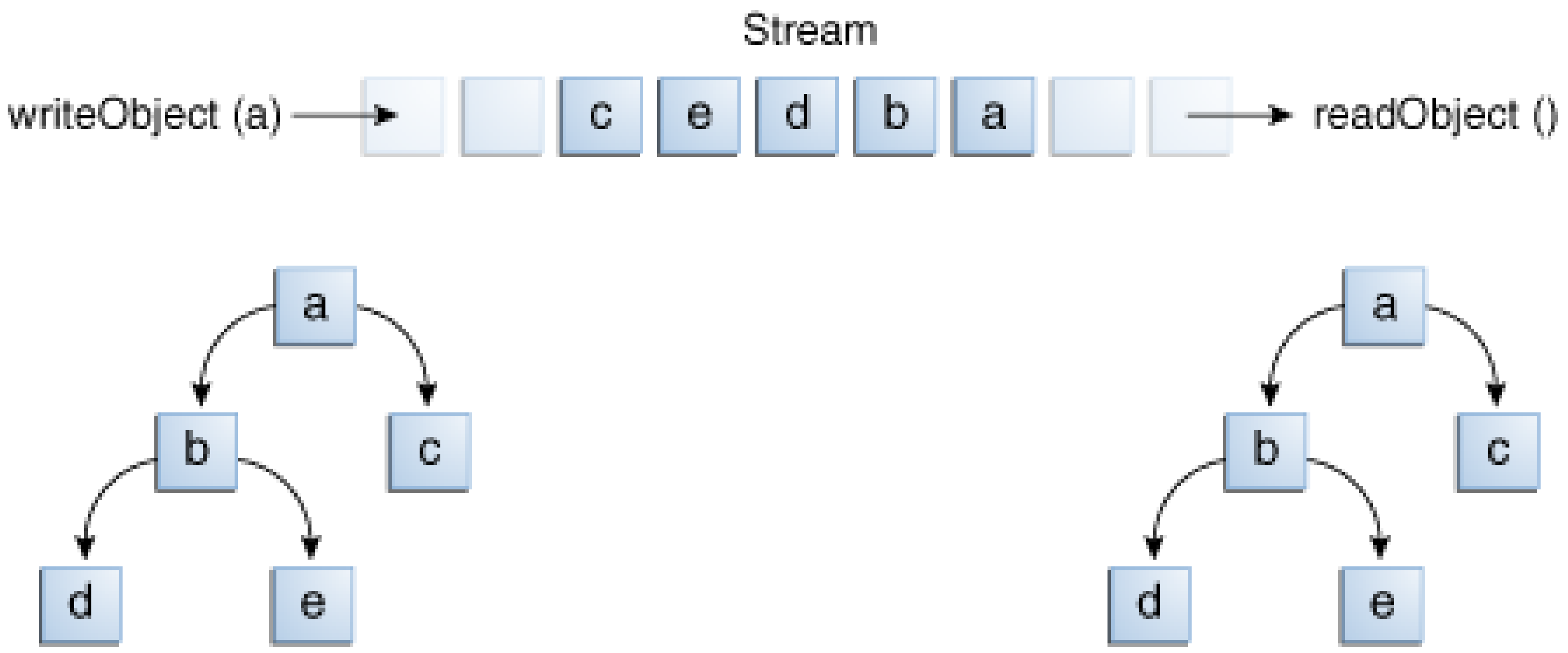
```
public class DataOut {
  static final String dataFile = "invoicedata";
  static final double[] prices = { 19.99, 9.99, 15.99 };
  static final int[] units = { 12, 8, 13 };
  static final String[] descs = { "alice", "bob", "cloe" };
  public static void main(String[] args) throws IOException {
    try (DataOutputStream out = new DataOutputStream(
            new BufferedOutputStream(
            new FileOutputStream(dataFile))) {
       for (int i = 0; i < prices.length; i ++) {
         out.writeDouble(prices[i]);
         out.writeInt(units[i]);
         out.writeUTF(descs[i]);                      // f d s
       } // for
    } // try
  } // main
}
```

java.io.DataInput

```
public class DataIn {
  public static void main(String[] args) throws IOException {
    try (DataInputStream in = new DataInputStream(
             new BufferedInputStream(
             new FileInputStream(dataFile)))) {
       while(true) {
          double price = in.readDouble();
          int unit = in.readInt();
          String desc = in.readUTF();
System.out.format("You ordered %d" + " units of %s at $%.2f%n",
            unit, desc, price);
       } // while
    } catch (EOFException e) {}
  } // main
}
```

Object Streams

- Java has built-in facilities for serializing objects
- java.io.ObjectInput extends DataInput
  - java.io.ObjectInputStream
  - Object readObject()
- java.io.ObjectOutput extends DataOutput
  - java.io.ObjectOutputStream
  - void writeObject(Object)



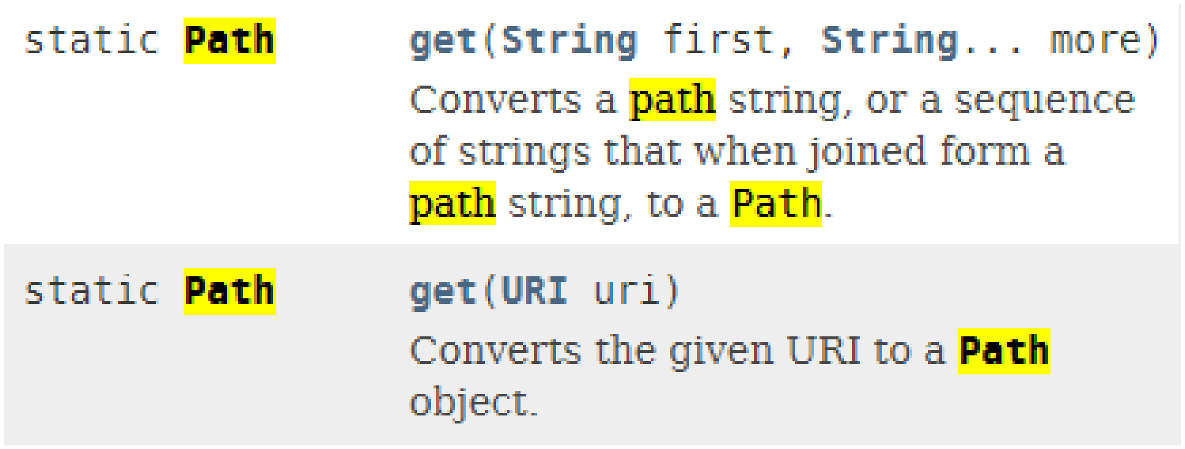
- Objects that can be written implement the "tagging" interface java.io.Serializable
  - Fields are automatically serialized
  - "transient" keyword marks fields which should not be serialized
- private static final long serialVersionUID = 42L;

Automatically handles duplicate references in the same stream

```
try (ObjectOutput out = new ObjectOutputStream(…)) {
  Object ob = new Object();
  out.writeObject(ob);
  out.writeObject(ob);
}

try (ObjectInput in = new ObjectInputStream(…)) {
  Object a = in.readObject();
  Object b = in.readObject();
  assert a == b;
}
```

java.nio.*

- java.nio.file.Paths

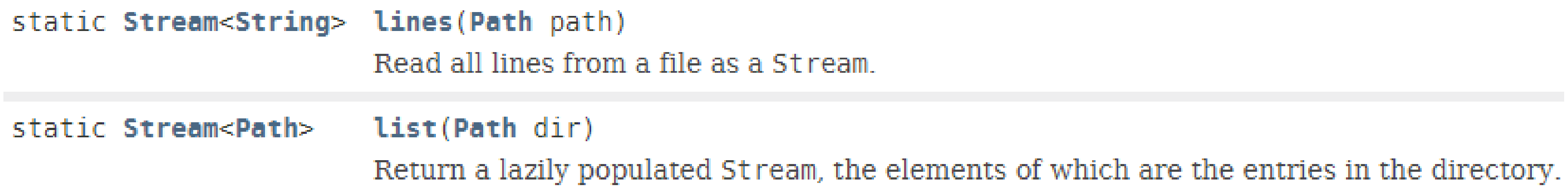


`static Path`	`get(String first, String... more)` Converts a path string, or a sequence of strings that when joined form a path string, to a Path.
`static Path`	`get(URI uri)` Converts the given URI to a Path object.

- java.nio.files.Files
  - File / directory access
  - Helpers to make streams, readers/writers, channels*
- Supports non-blocking I/O

Stream Streams

- java.nio.files.Files



`static Stream<String>`	`lines(Path path)` Read all lines from a file as a `Stream`.
`static Stream<Path>`	`list(Path dir)` Return a lazily populated `Stream`, the elements of which are the entries in the directory.

Recap

- java.io
  - I/O streams
  - Binary Streams
  - Character Streams
    - Readers/Writers
  - Wrappers
    - Buffers
    - Scanner
    - PrintWriter
  - Marshalling
    - Data Streams
    - Object Streams

- java.nio
  - Channels & Buffers*
    - Non-blocking I/O

- Stream Streams