Streams

Often a program needs to:

- bring in information from an external source, or
- send out information to an external destination.

The information can be:

- in a file on a disk
- somewhere on the network
- in memory
- in another program

Streams present an abstraction that allows to access (read or write) such information **sequentially**.

Using the streams we can access sources and destinations of information in a unified way no matter where they actually are.

Streams

We distinguish two types of streams:

- **input streams** – programs read from them
- **output streams** – programs write to them

The algorithms for sequentially reading and writing data are basically the same:

- **Reading**

```
open a stream
while more information
    read information
close the stream
```

- **Writing**

```
open a stream
while more information
    write information
close the stream
```

Streams

The package **java.io** contains a collection of stream classes.

The stream classes are divided into two class hierarchies:

- **Byte Streams** – they work on streams of 8-bit bytes (binary data). They are subclasses of (abstract) classes:
  - **InputStream** – input streams
  - **OutputStream** – output streams
- **Character Streams** – they work on streams of 16-bit characters (text files). They are subclasses of (abstract) classes:
  - **Reader** – input streams
  - **Writer** – output streams

Streams

InputStream:

- int read()
- int read(byte[] b)
- int read(byte[] b, int off, int len)

OutputStream:

- void write(int b)
- void write(byte[] b)
- void write(byte[] b, int off, int len)

Reader:

- int read()
- int read(char[] cbuf)
- int read(char[] cbuf, int off, int len)

Writer:

- void write(int c)
- void write(char[] cbuf)
- void write(char[] cbuf, int off, int len)

Streams

There are also other methods. All these classes contain method

- void close()

Note: The method close() can be called either explicitly, or implicitly by the garbage collector.

The classes **InputStream** and **Reader** contain methods

- long skip(long n)
- boolean markSupported()
- void mark(int readAheadLimit)
- void reset()

The classes **OutputStream** and **Writer** contain method

- void flush()

Most of the methods that work with streams can throw **java.io.IOException** (or some of its subclasses).

File Streams

The file streams read or write a file on the file system:

- FileInputStream
- FileOutputStream
- FileReader
- FileWriter

An example of use of **FileReader** and **FileWriter**:

```
Reader in = new FileReader("input.txt");
Writer out = new FileWriter("output.txt");
int c;
while ((c = in.read()) >= 0) {
    out.write(c);
}
in.close();
out.close();
```

File Streams

It is better to read and write bigger chunks of data:

```
InputStream in = new FileInputStream("input.txt");
OutputStream out = new FileOutputStream("output.txt");
final int BUF_LEN = 8192;
byte[] buf = new byte[BUF_LEN];
int l;
while ((l = in.read(buf, 0, BUF_LEN)) >= 0) {
    out.write(buf, 0, l);
}
in.close();
out.close();
```

File Streams

File streams can be created using:

- a file name (class **String**)
- a file object (class **File**)
- a file descriptor (class **FileDescriptor**)

For example, the class **FileReader** contains the following constructors:

- FileReader(String fileName)
- FileReader(File file)
- FileReader(FileDescriptor fd)

Classes **FileOutputStream** and **FileWriter** contain also constructors that allow to specify if an existing file should be overwritten or data should be appended to it:

- FileOutputStream(String name, boolean append)
- FileOutputStream(File file, boolean append)

Class `File`

The instances of the class **java.io.File** represent files on the file system.

It presents an abstract, system-independent view of hierarchical pathnames.

We can create a File object for a file on the file system and query the object for information about the file, such as:

- the full path name
- the name of its parent directory
- if it is directory or a regular file
- if it is an absolute or relative pathname
- if the file exists
- the length of the file
- the access rights (if it can be read and/or written)
- other attributes (time of modification, if it is hidden, ...)

Class `File` (cont.)

We can use an object of class File also for manipulation with the given file. We can for example:

- create the file
- delete the file
- rename the file
- obtain a list of files in the directory
- create a subdirectory
- set time of modification
- create temporary files

Example of a deletion of a file:

```
String filename = "test.txt";
File f = new File(filename);
boolean ok = f.delete();
System.out.println(ok ? "O.K." : "Not deleted");
```

Class `File` (cont.)

Example of use of the class File:

```
File input = new File("input.txt");
if (!input.exists()) {
   System.err.println("Error: file \"" + input.getName() +
      "\" doesn't exist");
   return;
}

FileReader reader = new FileReader(input);
...
```

Filter Streams

The **java.io** package provides a set of abstract classes that define and partially implement **filter streams**:

- FilterInputStream
- FilterOutputStream
- FilterReader
- FilterWriter

Filter streams allow to combine features of streams and achieve desired functionality.

A filter stream is constructed on another stream (the **underlying** stream):

- The read method reads input from the underlying stream, filters it and passes to the caller.
- The write method filters output and writes the resulting data to the underlying stream.

Buffered Streams

An example of filter streams are **buffered streams**:

- BufferedInputStream
- BufferedOutputStream
- BufferedReader
  - LineNumberReader
- BufferedWriter

An example of use of BufferedReader:

```
BufferedReader reader =
    new BufferedReader(new FileReader("input.txt"));
String s;
while((s = reader.readLine()) != null) {
    System.out.println(s);
}
```

Other Types of Streams

Another type of filter streams are **pushback streams**:

- PushbackInputStream
- PushbackReader

They add to streams the ability to push back or unread bytes or characters.

The are streams for conversion between byte streams and character streams:

- InputStreamReader
- OutputStreamWriter

Note: The character encoding used by these streams can be specified in their constructors.

```
Reader r = new InputStreamReader(
           new FileInputStream("input.txt"));
Writer w = new OutputStreamWriter(
           new FileOutputStream("output.txt"), "iso-8859-2");
```

Print Streams

Print streams allow to print values of different data types in a human readable form:

- PrintStream
- PrintWriter

Unlike other streams the print streams never throw an IOException; instead, exceptional situations merely set an internal flag that can be tested via the checkError() method.

Optionally, they can be created so as to flush automatically after every end of line.

The overloaded methods print() and println() are used to print values of various data types:

- void print(boolean b)
- void print(char c)
- void print(int i)
- ...

Print Streams (cont.)

The methods println() should be used to print line separators instead of using '\n' in printed strings.

In the following example

```
PrintWriter w = new PrintWriter(
                new FileOutputStream("output.txt"));
w.print("Hello\n");
```

it is better to use

```
w.println("Hello");
```

Different platforms use different line separators:

Platform	Decimal	Chars
MS Windows	13 10	"\r\n"
Unix	10	"\n"
MacOS	13	"\r"

Standard Input and Output

Three standard streams are streams are defined in the class **java.lang.System** as static final variables:

- **in** – standard input (InputStream)
- **out** – standard output (PrintStream)
- **err** – standard error output (PrintStream)

All these streams are implicitly opened.

These streams should not be closed.

Standard input stream typically corresponds to keyboard input.

Standard output and error streams typically correspond to display output.

All these streams can be redirected by a user to a file or another program:

```
$ java MyClass < input.txt > output.txt
$ java MyClass < input.txt | less
```

Stream Tokenizer

The StreamTokenizer class takes an input stream and parses it into tokens, allowing the tokens to be read one at a time. The stream tokenizer can recognize identifiers, numbers, quoted strings, and various comment styles.

```
StreamTokenizer s = new StreamTokenizer(
        new InputStreamReader(System.in));
s.eolIsSignificant(true);
loop: while (true) {
    switch (s.nextToken()) {
    case StreamTokenizer.TT_EOF: break loop;
    case StreamTokenizer.TT_WORD:
      System.out.println("a word: " + s.sval); break;
    case StreamTokenizer.TT_NUMBER:
      System.out.println("a number: " + s.nval); break;
    case StreamTokenizer.TT_EOL:
      System.out.println("EOL"); break;
    default:
      System.out.println("other: " + (char)s.ttype);
    }
}
```

Reading from URL

The streams are also used to represent network connections:

```
URL url = new URL("http://java.sun.com/docs");
InputStream in = url.openStream();
OutputStream out = new FileOutputStream("output.txt");
int c;
while ((c = in.read()) >= 0) {
    out.write(c);
}
in.close();
out.close();
```

Note: The class **URL** is from the **java.net** package.

Data Streams

There are input and output streams for reading and writing primitive data types in a **binary** (but portable) form at:

- DataInputStream
- DataOutputStream

The class **DataInputStream** contains methods such as:

- void readFully(byte[] b)
- void readFully(byte[] b, int off, int len)
- boolean readBoolean()
- byte readByte()
- int readUnsignedByte()
- short readShort()
- int readUnsignedShort()
- int readInt()
- String readUTF()
- . . .

Data Streams (cont.)

The class **DataOutputStream** contains methods such as:

- void writeBoolean(boolean v)
- void writeByte(int v)
- void writeChar(int v)
- void writeInt(int v)
- void writeLong(long v)
- void writeFloat(float v)
- void writeDouble(double v)
- void writeBytes(String s)
- void writeChars(String s)
- void writeUTF(String str)

All these methods for reading and writing binary data are declared in interfaces:

- DataInput
- DataOutput

Serialization

Java's **object serialization** allows to take any object that implements the **java.io.Serializable** interface and turn it into a sequence of bytes that can later be fully restored to regenerate the original object.

The following classes are used to read and write objects:

- ObjectInputStream
- ObjectOutputStream

It is possible to use these classes to read and write primitive data types since they implement interfaces DataInput and DataOutput.

> **Note:** Instance variables defined as **transient** and static variables are prevented from serialization.

The interface **java.io.Serializable** does not declare any methods.

Serialization (cont.)

Writing into an object stream:

```
FileOutputStream fos = new FileOutputStream("t.tmp");
ObjectOutputStream oos = new ObjectOutputStream(fos);
oos.writeInt(12345);
oos.writeObject("Today");
oos.writeObject(new Date());
oos.close();
```

Reading from an object stream:

```
FileInputStream fis = new FileInputStream("t.tmp");
ObjectInputStream ois = new ObjectInputStream(fis);
int i = ois.readInt();
String today = (String) ois.readObject();
Date date = (Date) ois.readObject();
ois.close();
```

Serialization (cont.)

Classes that require special handling during the serialization and deserialization process must implement two special methods with the given signatures:

```
private void writeObject(ObjectOutputStream s)
        throws IOException {
    s.defaultWriteObject();
    // customized serialization code
}
```

```
private void readObject(ObjectInputStream s)
        throws IOException, ClassNotFoundException {
    s.defaultReadObject();
    // customized deserialization code
    //   . . .
    // followed by code to update the object, if necessary
}
```

Serialization (cont.)

For complete, explicit control of the serialization process, a class must implement the **java.io.Externalizable** interface.

For Externalizable objects, only the identity of the object's class is automatically saved by the stream. The class is responsible for writing and reading its contents.

```
package java.io;

public interface Externalizable extends Serializable {
    public void writeExternal(ObjectOutput out)
        throws IOException;

    public void readExternal(ObjectInput in)
        throws IOException, java.lang.ClassNotFoundException;
}
```

Note: Default constructor of a deserialized object implementing Externalizable is always invoked. Thus the constructor must be **public**.

Random Access Files

The input and output streams are **sequential access streams**. **Random access files** permit nonsequential, or random, access to a file's contents.

The **RandomAccessFile** class in the **java.io** package implements a random access file.

Note: The **RandomAccessFile** class is not part of class hierarchy of streams, but it implements **DataInput** and **DataOutput** interfaces.

It is possible to open a random access file only for reading:

```
new RandomAccessFile("file.txt", "r");
```

And also for reading and writing:

```
new RandomAccessFile("file.txt", "rw");
```

After the file has been opened, the common methods read() and write() can be used for reading and writing.

Random Access Files (cont.)

The class RandomAccessFile supports the notion of a **file pointer** that indicates the current location in the file.

- When the file is opened, the file pointer is set to 0 (to the beginning of the file).
- Calls to the read() and write() methods adjust the file pointer by the number of bytes read or written.

The RandomAccessFile class contains three methods for explicitly manipulating the file pointer:

- int skipBytes(int n) – moves the file pointer forward the specified number of bytes
- void seek(long pos) – positions the file pointer just before the specified byte
- long getFilePointer() – returns the current byte location of the file pointer

Random Access Files (cont.)

The RandomAccessFile class contains also methods for manipulation with the length of the file:

- long length() – returns the length of the file
- void setLength(long newLength) – sets the length of the file