

Introduction to Programming - Concepts and Tools

Lecture 10

Reading and Writing Files

Overview

- Writing and reading binary files
- Input and output streams
- Appending to files
 - Random access binary files
 - Copying files
- The `File` class
- Closing streams and exceptions
- Writing and reading text files
- Parsing text files
- Encrypting text files

```
import java.io.*;

class WritingInts {

    public static void main(String[] args) throws IOException {
        DataOutputStream out = new DataOutputStream(
            new FileOutputStream("test.bin"));
        out.writeInt(43210);
        out.writeInt(98765);
        out.close();
    }
}
```

The program writes two `int`'s to a file `test.bin`. If the file doesn't exist it is *created*, otherwise it is *overwritten*.

The file `test.bin` is **binary**, it is not easy displayable in an editor. (Try to open it!). But it can be read by.

```
import java.io.*;

class ReadingInts {

    public static void main(String[] args) throws IOException {
        DataInputStream in = new DataInputStream(
            new FileInputStream("test.bin"));
        System.out.println(in.readInt());
        System.out.println(in.readInt());
        in.close();
    }
}
```

`test.bin` contains a sequence of 8 bytes, 4 for each of the two `int`'s. It has *no type information*, and can be regarded as a sequence of 8 byte's.

```
import java.io.*;

class ReadingBytes {

    public static void main(String[] args) throws IOException {
        DataInputStream in = new DataInputStream(
            new FileInputStream("test.bin"));
        for (int i = 0; i < 8; i++)
            System.out.println(in.readByte());
        in.close();
    }
}
```

java.io

```
import java.io.*;

class WritingInts {

    public static void main(String[] args) throws IOException {
        DataOutputStream out = new DataOutputStream(
            new FileOutputStream("test.bin"));
        out.writeInt(43210);
        out.writeInt(98765);
        out.close();
    }
}
```

`java.io.*` is the package used for input and output.

Streams

All input and output (excluding graphical) is carried out through ***streams***, i.e. a sequence of bytes.

```
public static void main(String[] args) throws IOException {  
    DataOutputStream out = new DataOutputStream(  
        new FileOutputStream("test.bin"));  
    ...  
}
```

Instances of `FileOutputStream` is an output stream for writing bytes to a file. The name of the file may be given as parameter to the constructor.

Streams

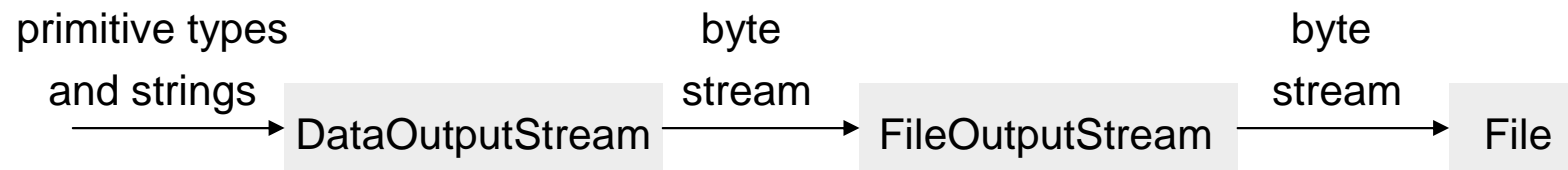
```
public static void main(String[] args) throws IOException {  
    DataOutputStream out = new DataOutputStream(  
        new FileOutputStream("test.bin"));  
    out.writeInt(43210);  
    out.writeInt(98765);  
    ...  
}
```

Instances of `DataOutputStream` writes *primitive data types* and *strings* as bytes to the output stream given as parameter to the constructor.

Methods: `writeBoolean()`, `writeShort()`, ..., `writeLong()`, ..., `writeDouble()`, and `writeUTF()` for writing strings using UTF encoding (explained later).

Streams

The correct assembling of instances of `FileOutputStream` and `DataOutputStream` provides a **pipeline**



Instances of `FileOutputStream` can write bytes directly to a file using `write(byte[])`. But it's more convenient to make use of a pipeline.

Writing to System.out

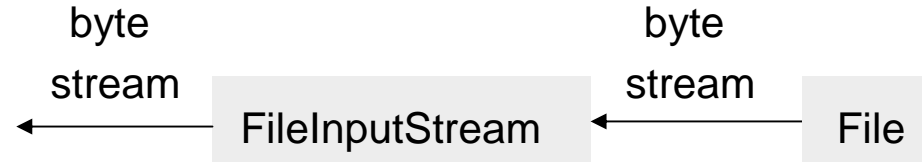
```
class WritingStringsToSystemOut {  
    public static void main(String[] args) throws IOException {  
        DataOutputStream out = new DataOutputStream(System.out);  
        out.writeUTF("Writing ");  
        out.writeUTF("to System.out");  
    }  
}
```

Instances of `DataOutputStream` may write to `System.out` (an output stream writing to "standard" output).

UTF encoding allows Unicode strings consisting entirely of ASCII characters to be encoded with 1 byte pr. character. (The first two bytes tells the length of the encoded output.)

Files are read using *input* streams.

```
class ReadingInts {  
    public static void main(String[] args) throws IOException {  
        DataInputStream in = new DataInputStream(  
            new FileInputStream("test.bin"));  
        ...  
    }  
}
```

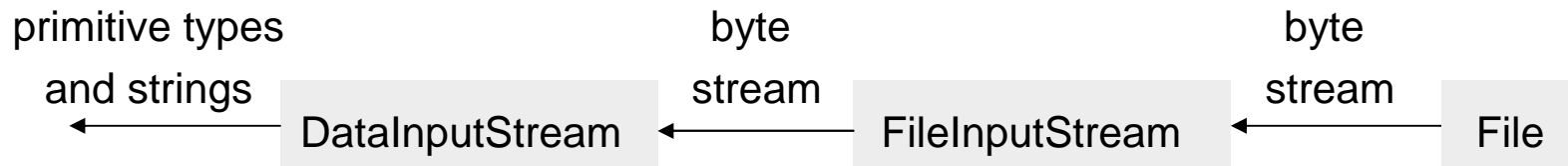


Instances of `FileInputStream` (the dual of `FileOutputStream`) reads a stream of bytes from a file.

```

public static void main(String[] args) throws IOException {
    DataInputStream in = new DataInputStream(
        new FileInputStream("test.bin"));
    System.out.println(in.readInt());
    System.out.println(in.readInt());
    ...
}

```



Instances of `DataInputStream` (the dual of `DataOutputStream`) reads *primitive data types* and *strings* from the byte input stream given as parameter to the constructor.

Methods: `readBoolean()`, `readShort()`, ... , `readLong()`, ... , `writelnDouble()`, and `readUTF()`.

Reading from System.in

```
class ReadingFromSystemIn {  
  
    public static void main(String[] args)  
        throws IOException {  
        DataInputStream in =  
            new DataInputStream(System.in);  
        System.out.println(in.readInt());  
    }  
}
```

The program will not give the expected output because each byte read is regarded as 1 byte of a 4-byte integer.

Output and input streams targeting the same source may occur in the same program

```
class WritingReadingInts {  
  
    public static void main(String[] args) throws IOException {  
  
        DataOutputStream out = new DataOutputStream(  
            new FileOutputStream("test.bin"));  
  
        out.writeInt(43210);  
        out.writeInt(98765);  
        out.close();  
  
        DataInputStream in = new DataInputStream(  
            new FileInputStream("test.bin"));  
        System.out.println(in.readInt());  
        System.out.println(in.readInt());  
        in.close();  
  
    }  
}
```

Appending to Files

Writing to an existing file will overwrite it. We may use `RandomAccessFile` to append to a binary file.

A ***random access file*** is like a large array of bytes. There is an index (the ***file pointer***) into the implied array.

Input operations read bytes starting at the file pointer and advance the file pointer past the bytes read.

Output operations write bytes starting at the file pointer and advance the file pointer past the bytes written

```
RandomAccessFile app = new RandomAccessFile("test.bin","rw");  
//opens a stream to read and write the file  
app.seek(app.length());  
//moves the file pointer to the end of the file  
app.writeInt(99999);  
app.close();
```

Appending to files by copying

```
DataInputStream in = new DataInputStream(  
    new FileInputStream("test.bin"));  
DataOutputStream tmp = new DataOutputStream(  
    new FileOutputStream("tmp.bin"));  
//create new temporary file tmp.bin  
tmp.writeInt(in.readInt());  
tmp.writeInt(in.readInt());  
//copy from test.bin  
tmp.writeInt(99999); //add new int to tmp  
in.close();  
tmp.close();  
  
File testFile = new File("test.bin");  
//An instance of File is an system independent representation of  
    a file name  
testFile.delete(); //delete test.bin  
File tmpFile = new File("tmp.bin");  
tmpFile.renameTo(testFile); // rename tmp.bin to test.bin
```

Instances of `File` can be used to work with files and directories

```
String fileName = ".";

if (args.length > 0)
    fileName = args[0];

File file = new File(fileName);

if (!file.exists())
    System.out.println(fileName + " doesn't exists");
else if (file.isDirectory()) {
    System.out.println(fileName + " is a directory\n"
        + "It's canonical path is " + file.getCanonicalPath()
        + "\nIt contains " + file.list().length + " files");
    // file.list() returns an array of names of the directory contents
}
else if (file.isFile()) {
    System.out.println(fileName + " is a file with length "
        + file.length() + " bytes\n"
        + "It's canonical path is " + file.getCanonicalPath()
        + "\nIt was last modified " + file.lastModified()
        + " milliseconds since 00:00:00 GMT, January 1, 1970");
}
```

Closing Streams

Creating a stream allocates a system resource (a file).

```
DataInputStream in = new DataInputStream(  
    new FileInputStream("test.bin"));
```

Failing to release the resource may cause problems.

```
//in.close();
```

E.g. a file cannot be deleted if it is allocated by a stream, so nothing will be printed by

```
File testFile = new File("test.bin");  
if (testFile.delete()) System.out.println("File deleted");
```

Exceptions

Forgetting to close a stream may cause **exceptions**, i.e. something *unexpected*.

The type of exceptions thrown by our program is declared by:

```
public static void main(String[]  
    args) throws IOException { ... }
```

E.g. failing to close a stream may cause a file not to be deleted (and another file not to be renamed) and `EOFException` may occur because trying to read past the end of the file.

An `EOFException` is a subtype of `IOException`.

```

public static void main(String[] args) throws IOException
{
    // IOExceptions must be caught
    DataInputStream input = null;
    if (args.length != 1) {
        System.out.println("Usage: java BinaryInput filename");
        System.exit(1);
    }
    try { input = new DataInputStream( new FileInputStream(args[0]) ); }
    catch (IOException e) {
        // exceptions from the try-block is caught and a message printed
        System.out.println("Could not open " + args[0]);
        System.exit(1);
    }
    int count = 0;
    try {
        while (true) {
            int myData = input.readInt();
            count++;
            System.out.print(myData + " ");
            if (count % 4 == 0)
                System.out.println();
        }
        // the loop terminates when input.readInt() throws and EOFException,
        the exception is caught by the catch-block
    }
    catch (EOFException e) { }
    if (count % 4 != 0) System.out.println();
}

```

```

public static int readBinaryInput(String filename, int howMany) throws IOException {
    // the method reads howMany integers or until EOF from filename, returns numbers of
    // integers read
    DataInputStream input = null;
    try { input = new DataInputStream(new FileInputStream(filename)); }
    catch (IOException e) {
        System.out.println("Could not open " +filename);
        throw e; // exception is thrown to the calling environment, and method stops
    }
    int count = 0;
    try {
        while (count < howMany) {
            int myData = input.readInt();
            count++;
            System.out.print(myData + " ");
            if (count % 4 == 0) System.out.println();
        } // the loop terminates normally or as the result of an EOFException
    }
    catch (EOFException e) {
        // just catch the exception and discard it
    }
    finally {
        // belongs to try-catch above. Executed no matter how the try block exits, even if
        // due to some uncaught IOException
        if (count % 4 != 0)
            System.out.println();
        if (input != null)
            input.close();
    }
    return count;
    // returns count, if loops terminate normally or if EOFException is thrown in the
    // second try-block
}

```

Writing Text Files

Text files (files of characters) are often more convenient to work with.

```
public static void main(String[] args)
    throws IOException {
    PrintWriter out = new PrintWriter(
        new FileWriter("test.txt"));
    out.println("Hello, trying to write text to a file");
    out.close();
}
```

Instances of `FileWriter` writes streams of characters to a file (a byte stream).

Instances of `PrintWriter` prints primitive types, strings and objects to a character output stream.

Reading Text Files

```
public static void main(String[] args)
    throws IOException {
    BufferedReader in = new BufferedReader(
        new FileReader("test.txt"));
    System.out.println(in.readLine());
    in.close();
}
```

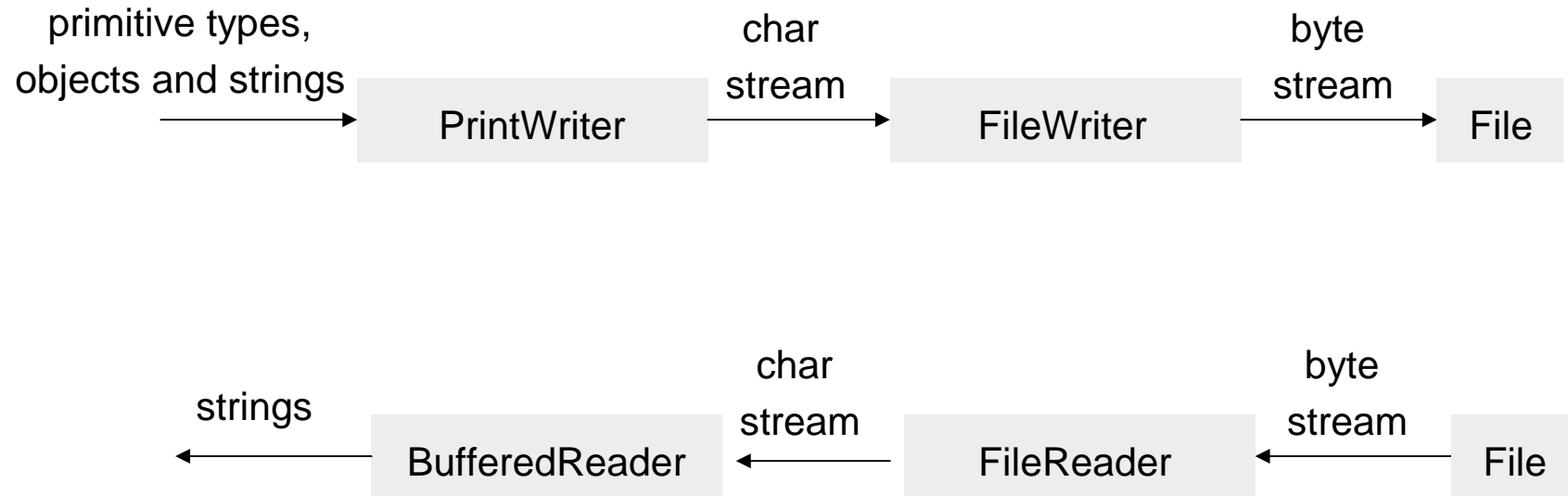
Instances of `FileReader` read a stream of characters from a file (a byte stream).

Instances of `BufferedReader` read text from a character input stream, buffering characters so as to provide for the efficient reading.

Readin one character at the time

```
public static void main(String[] args)
    throws IOException {
    BufferedReader in = new BufferedReader(
        new FileReader("test.txt"));
    int c = in.read();
    // read next character (as an int), returns -1 if
    // at EOF
    while (c != -1) {
        System.out.print((char)c);
        c = in.read();
    }
    in.close();
}
```

Reading and Writing Text Files



Parsing Text Files

If one number on each line and we know how many lines then:

```
PrintWriter out = new PrintWriter(new FileWriter("test.txt"));  
out.println(1234);  
out.println(1234.5678);  
out.println(5678.9999);  
out.close();
```

```
BufferedReader in = new BufferedReader(  
    new FileReader("test.txt"));  
System.out.println(Double.parseDouble(in.readLine().trim()));  
System.out.println(Double.parseDouble(in.readLine().trim()));  
System.out.println(Double.parseDouble(in.readLine().trim()));  
in.close();
```

If one number on each line but we don't know how many lines then

```
PrintWriter out = new PrintWriter(new FileWriter("test.txt"));
double d = 1000*Math.random();
do {
    out.println(d);
    d = 1000*Math.random();
} while (d < 900);
out.close();
```

```
BufferedReader in = new BufferedReader(
    new FileReader("test.txt"));
String line = in.readLine();
while (line != null) {
    System.out.println(Double.parseDouble(line.trim()));
    line = in.readLine();
    // when at EOF readLine returns null
}
in.close();
```

If there are several numbers on each line it gets complicated (see e.g. app. C.1 about `tio.ReadInput`)

```
PrintWriter out = new PrintWriter(  
                                new FileWriter("test.txt"));  
double d = 1000*Math.random();  
do {  
    out.print(d); out.print(" ");  
    //writes numbers separeted by blank  
    d = 1000*Math.random();  
} while (d < 900);  
out.close();
```

```
ReadInput in = new ReadInput("test.txt");  
while (in.hasMoreElements())  
    //determines if there are non-white space left  
    System.out.println(in.readDouble());
```

If one forgets to close the `PrintWriter` stream before reading from `test.txt` the file is empty and nothing is read.

```
PrintWriter out = new PrintWriter(  
    new FileWriter("test.txt"));  
double d = 1000*Math.random();  
do {  
    out.print(d); out.print(" ");  
    //writes numbers separeted by blank  
    d = 1000*Math.random();  
} while (d < 900);  
//out.close();  
  
ReadInput in = new ReadInput("test.txt");  
while (in.hasMoreElements())  
    //determines if there are non-white space left  
    System.out.println(in.readDouble());
```

Encrypting Text Files

Encrypting ***clear text*** is to make ordinary text look apparently meaningless.

An encryption algorithm is called a ***cipher***.

The ***Caesar cipher*** with ***key*** n replaces each letter in the clear text by a letter n places later in the alphabet.

E.g. “Also You my son Brutus” becomes “Bmtp zpv nz tpo Csvvvt” if the key is 1 (assuming blanks are not encrypted).

Considering only upper case letters ('A', ..., 'Z')
the Caesar cipher may be implemented by

```
static char caesarCipher(char c, int key) {
    return (char)((c - 'A' + key) % 26 + 'A');
}

public static void main(String[] args) {
    System.out.println("Type an upper case string and a key");
    String clearText = Console.in.readWord();
    int key = Console.in.readInt();

    char[] c = clearText.toCharArray();
    for (int i = 0; i < c.length; i++)
        c[i] = caesarCipher(c[i],key);

    String cipherText = new String(c);
    System.out.println("The Caesar encoding of " +
        clearText + " is " + cipherText);
}
```

Encrypting Text Files

The Caesar cipher is easy to **break**, e.g. try all possible keys from 1 to 25.

The ***Vignere cipher*** is a generalization of the Caesar cipher using ***a series of keys***.

Encoding “Also you my son Brutus” with the key series 1 2 3 gives:

Clear text: Also you my son Brutus

Offset: 1231 231 2 3 123 123123

Encoded text: Bnvp aro ob tqq Ctyuxv

Notice that different occurrences of a letter may be mapped differently.

Considering only 'A', ..., 'Z' the Vignere cipher may be implemented by

```
static void vignereCipher(char[] c, int[] key) {  
    for (int i = 0; i < c.length; i++)  
        c[i] = caesarCipher(c[i],key[i%key.length]);  
}
```

```
public static void main(String[] args) {  
    System.out.println("Type an upper case string,  
        the number of keys, and the keys");  
    String clearText = Console.in.readword();  
    int n = Console.in.readInt();  
    int[] key = new int[n];  
    for (int i = 0; i < n; i++)  
        key[i] = Console.in.readInt();  
    char[] c = clearText.toCharArray();  
    vignereCipher(c,key);  
    ...  
}
```

Encrypting Text Files

The longer the ***Vignere key*** the better the cipher and the harder it is to break.

If the key is as long as the clear text we have the ***Vernam cipher***.

If the key is only used once, ***a one-time pad***, the cipher is provably unbreakable.

Next we show how to use a random number simulator to generate a one-time pad.

```

final static boolean ENCODE = true;
final static boolean DECODE = false;

static char caesarCipher(char c, int key, boolean mode) {
    // the mode decides whether we encrypt or decrypt
    if (mode == ENCODE)
        return (char)((c - 'A' + key) % 26 + 'A');
    else // mode == DECODE
        return (char)((c - 'A' + 26 - key) % 26 + 'A');
    // If c is encoded by key k then decryption the code of c
    // by key k gives c again
}

static void vernamCipher(char[] c, int seed, boolean mode) {
    Random r = new Random(seed);
    // Instances of Random generates a sequence of pseudo random
    // numbers. Instances instantiated with the same key
    // generates the same sequence
    for (int i = 0; i < c.length; i++) {
        int key = r.nextInt(26);
        // a random number between in [0,...,25]
        c[i] = caesarCipher(c[i],key,mode);
    }
}

```

```

public static void main(String[] args) {
    System.out.println("Type an upper case string and a
    seed");
    String clearText = Console.in.readword();
    int seed = Console.in.readInt();

    char[] c = clearText.toCharArray();
    vernamCipher(c,seed,ENCODE);
    // encoding the typed string

    String cipherText = new String(c);
    System.out.println("The Vernam encoding of " +
    clearText + " is " + cipherText);

    vernamCipher(c,seed,DECODE);
    // decoding the encoded string, using the same seed for
    // decoding as for encoding

    String decodedText = new String(c);
    System.out.println("The Vernam decoding of " +
    cipherText + " is " + decodedText);
}

```

The End