Object Oriented Programming

Java

Part 11: Input/Output

Introduction

- It is common to separate the data entering and leaving the environment to the program in binary information and textual information.
- **Binary information**: constituted by zeros and ones which codify any data type and need to be interpreted by the application.
- **Textual information**: constituted by characters and digits which in Java correspond to the **primitive type `char` (UTF-16)**.

`java.io` package (1)

- All classes manipulating files are contained in `java.io`.
- Classes that require access to file must contain:

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

`java.io` package (2)

- The `java.io` package is divided in 5 parts:
  1. **Byte stream**:
     - input stream.
     - output stream.
  2. **Character stream**:
     - readers.
     - writers.
  3. **Data stream**, which deal with primitive data types (`int`,…) and strings (`String`).
  4. **Object stream**, which transcribe objects to `Bytes` and vice-versa.
  5. **Class `File`**, which deals with file attributes.

Pre-defined streams

- J2SE provides the final static class `System`. This class contains three static fields:
  - **`InputStream in`**: input stream of the program (usually the keyboard)
  - **`PrintStream out`**: output stream of the program (usually the monitor)
  - **`PrintStream err`**: error stream (usually coincides with `System.out`)

Binary streams

- **`InputStream`**: classe abstracta que lê fluxos binários de informação.
- **`OutputStream`**: classe abstracta que escreve fluxos binários de informação.
- **`PrintStream`**: subclasse concreta de `OutputStream` que escreve fluxos não necessariamente binários de informação.

`InputStream` class (1)

- Some methods of the `InputStream` class:

`void close()`

Closes this input stream and releases any system resources associated with the stream.

`abstract int read()`

Reads the next byte of data from the input stream. The value byte is returned as an `int` in the range 0 to 255. If no byte is available the value -1 is returned.

`int read(byte[] b)`

Reads bytes from the input stream and stores them into the buffer array b. The number of bytes actually read is returned as an integer.

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

Reads up to `len` bytes of data from the input stream into an array of bytes. An attempt is made to read as many as `len` bytes, but a smaller number may be read. The number of bytes read is returned as an integer.

`InputStream` class (2)

`long skip(long n)`

Skips over and discards `n` bytes of data from this input stream. The skip method may end up skipping over some smaller number of bytes, possibly 0 (e.g. EOF). The actual number of bytes skipped is returned. If `n` is negative, no bytes are skipped.

`boolean markSupported()`

Tests if this input stream supports the mark and reset methods.

`void mark(int readLimit)`

Marks the current position in this input stream. A subsequent call to the reset method repositions this stream at the last marked position so that subsequent reads re-read the same bytes. The `readlimit` arguments tells this input stream to allow that many bytes to be read before the mark position gets invalidated.

`void reset()`

Repositions this stream to the position at the time the mark method was last called on this input stream.

`OutputStream` class

- Some methods of `OutputStream`:

`close()`
Closes this output stream and releases any system resources associated with this stream.

`void flush()`
Flushes this output stream and forces any buffered output bytes to be written out.

`void write(byte[] b)`
Writes `b.length` bytes from the specified byte array to this output stream.

`void write(byte[] b, int off, int len)`
Writes `len` bytes from the specified byte array starting at offset `off` to this output stream.

`PrintStream` class

- Some methods of `PrintStream`:

```
void print(boolean b)
void print(char c)
void print(char[] c)
void print(double d)
void print(float f)
void print(int i)
void print(long l)
void print(Object obj)
void print(String s)
```

Prints the value of each type.

```
void println()
```

Prints and terminates the line.

Binary streams for files

- **`File`**: class that abstracts the representation of a file and directory paths.
- **`FileDescriptor`**: final class that represents an open file.
- **`FileInputStream`**: subclass of `InputStream` that reads binary streams from files (`File` or `FileDescriptor`).
- **`FileOutputStream`**: subclass of `OutputStream` that writes binary streams to files (`File` or `FileDescriptor`).

`File` class

- Some constructors of the `File` class:

`File(String pathname)`
Creates a new `File` instance with this `pathname` string. The `pathname` may be relative or absolute. If it a relative pathname, then the current directory of the used is taken into consideration.

- Some methods of the `File` class:

`boolean exists()`	Tests whether the file/directory exists.
`boolean canRead()`	Tests whether the application can read the file.
`boolean canWrite()`	Tests whether the application can write the file.
`boolean delete()`	Deletes the file.
`long length()`	Returns the length of the file.
`String getName()`	Returns the name of the file/directory.

`FileDescriptor` class

- A `FileDescriptor` object represents a value dependent of the operating system that describes an open file.
- Applications do not build objects of type `FileDescriptor`, those should be obtained through the `getFD` method of `FileInputStream` and `FileOutputStream`.

Classe `FileInputStream`

- Constructors of the `FileInputStream` class:

```
FileInputStream(File file)
FileInputStream(FileDescriptor fdObj)
FileInputStream(String name)
```

Builds a new `FileInputStream` for the given file.

- Besides the usual `close`, `read`, `skip`, etc, methods it also contains the method:

```
FileDescriptor getFD()
```

Returns the `FileDescriptor` object that represents the connection to the actual file in the file system being used by this `FileInputStream`.

`FileOutputStream` class

- Constructors of the `FileOutputStream` class:

```
FileOutputStream(File file)
FileOutputStream(File file, boolean append)
FileOutputStream(FileDescriptor fdObj)
FileOutputStream(String name)
FileOutputStream(String name, boolean append)
```

Build a `FileOutputStream` for the given file. The constructors that receive a `boolean append` write in the end of the file if `append` is `true` and write in the beginning of the file if `append` is `false`.

- Beside the usual `close`, `write`, etc, methods it also contains:

```
FileDescriptor getFD()
```

Returns the `FileDescriptor` object that represents the connection to the actual file in the file system being used by this `FileOutputStream`.

Examples of binary streams

```
import java.io.*;
// ...
int nbytes=0;
FileInputStream f = new FileInputStream(“test”);
while(f.read() != -1)
    nBytes++;
System.out.println(“Number = ” + nBytes);
f.close()
// ...
```

```
import java.io.*;
// ...
FileOutputStream f = new FileOutputStream(“test”);
f.write(65); f.write(66); f.write(67);
f.close()
// ...
```

Character streams

- `Reader`: abstract class that read char streams.
- `Writer`: abstract class that write char streams.
- `InputStreamReader`: subclass of `Reader` that is a bridge from byte streams to character streams. It reads bytes and decodes them into characters using a specified charset.
- `OutputStreamWriter`: subclass of `Writer` that is a bridge from character streams to byte streams. Characters written to it are encoded into bytes using a specified charset.

Standard charsets

- Any implementation of Java supports the following charsets:
  - **US-ASCII**
    Seven-bit ASCII, a.k.a. ISO646-US, a.k.a. the Basic Latin block of the Unicode character set.
  - **ISO-8859-1**
    ISO Latin Alphabet No. 1, a.k.a. ISO-LATIN-1.
  - **UTF-8**
    Eight-bit UCS Transformation Format.
  - **UTF-16BE**
    Sixteen-bit UCS Transformation Format, big-endian byte order.
  - **UTF-16LE**
    Sixteen-bit UCS Transformation Format, little-endian byte order.
  - **UTF-16**
    Sixteen-bit UCS Transformation Format, byte order identified by an optional byte-order mark.

`InputStreamReader` class (1)

- Some constructors of `InputStreamReader`:

```
InputStreamReader(InputStream in)
```
Creates an `InputStreamReader` that uses the default character encoding (usually the one of the operating system).

```
InputStreamReader(InputStream in, String charsetName)
```
Creates an `InputStreamReader` with the character encoding received as parameter.

- The `InputStreamReader` class is a wrapper of an `InputStream` object.

`InputStreamReader` class (2)

- Some methods of `InputStreamReader`:

`void close()`
Closes this input stream reader and releases any system resources associated with this stream.

`String getEncoding()`
Returns the character encoding of this input stream reader.

`int read()`
Reads the next char of data from the input stream reader. The value byte is returned as an `int` in the range 0 to 255. If no byte is available the value -1 is returned.

`int read(char[] cbuf, int off, int len)`
Reads up to `len` bytes of data from the input stream into an array of bytes. An attempt is made to read as many as `len` bytes, but a smaller number may be read. The number of bytes read is returned as an integer.

`OutputStreamWriter` class (1)

- Some constructors of `OutputStreamWriter`:

`OutputStreamWriter(OutputStream in)`
Creates an `OutputStreamWriter` that uses the default character encoding (usually the one of the operating system).
`OutputStreamWriter(OutputStream in, String charsetName)`
Creates an `OutputStreamWriter` with the character encoding received as parameter.

- The `OutputStreamWriter` class is a wrapper of an `OutputStream` object.

`OutputStreamWriter` class (2)

- Some methods of `OutputStreamWriter`:

`void close()`
Closes this output stream writer and releases any system resources associated with this stream.

`void flush()`
Flushes this output stream and forces any buffered output bytes to be written out.

`String getEncoding()`
Returns the character encoding of this input stream reader.

`void write(char[] cbuf, int off, int len)`
Writes `len` characters from the specified char array starting at offset `off` to this output stream.

`void write(String str, int off, int len)`
Writes `len` characters from the specified String starting at position `off` to this output stream.

Character streams for files

- **`FileReader`**: subclass of `InputStreamReader` for reading character files.
- **`FileWriter`**: subclass of `OutputStreamWriter` for writing character files.

`FileReader` class

- Constructors of `FileReader`:

```
FileReader(File file)
FileReader(FileDescriptor fdObj)
FileReader(String name)
```

Creates a `FileReader` for the file received as parameter.

`FileWriter` class

- Constructors of the `FileWriter` class:

```
FileWriter(File file)
FileWriter(File file, boolean append)
FileWriter(FileDescriptor fdObj)
FileWriter(String name)
FileWriter(String name, boolean append)
```

Creates a `FileWriter` for the file received as parameter. The constructors that receive a `boolean append` write in the end of the file if `append` is `true` and write in the beginning of the file if `append` is `false`.

Examples of character streams (1)

```
import java.io.*;
// ...
int c, ones=0;
FileReader f = new FileReader(“test”);
while(c = f.read() != -1)
    if ((char)c == ‘1’)
        ones++;
System.out.println(“Occurred ” + ones + “ones!”);
f.close()
// ...
```

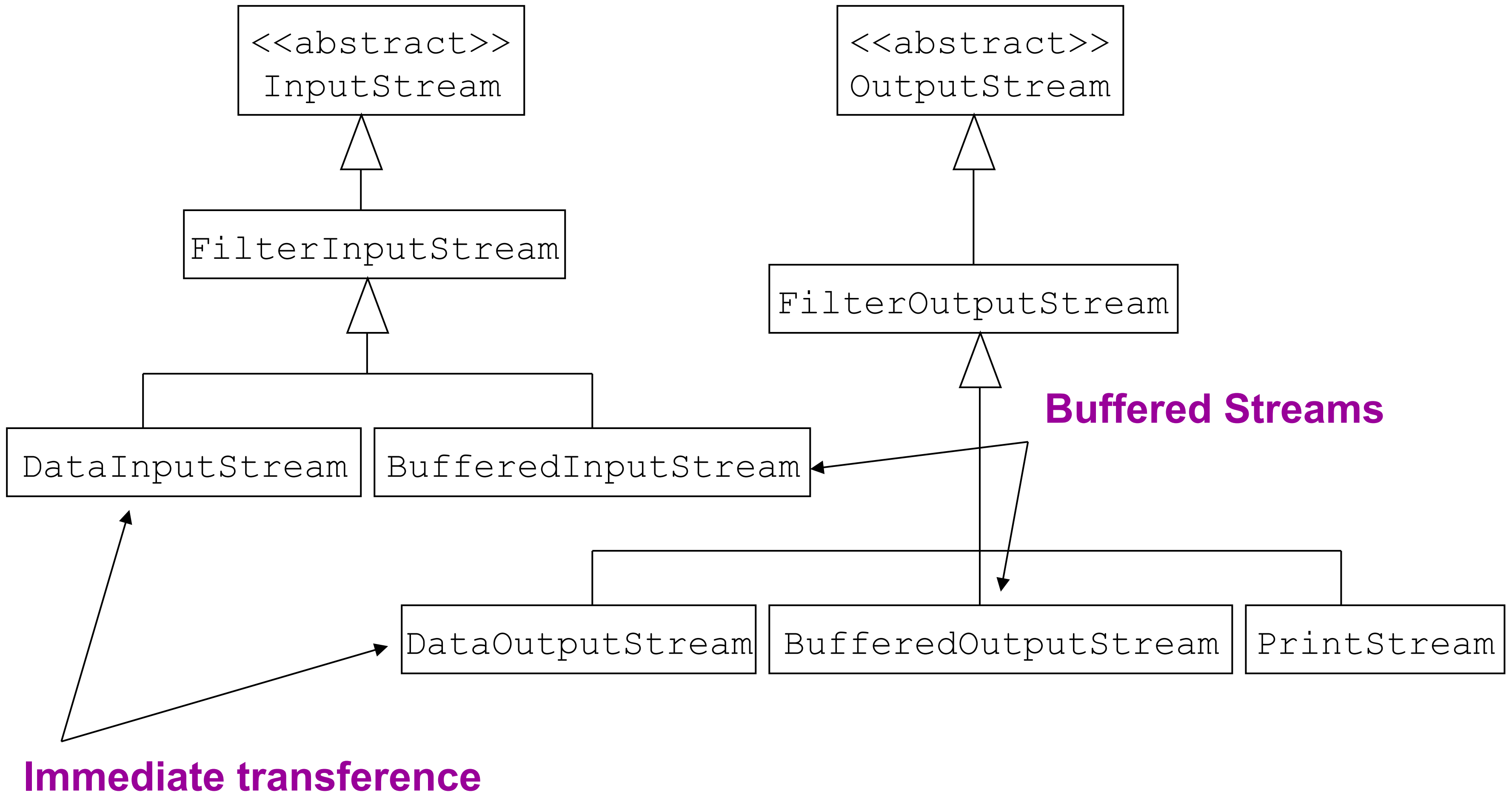
Examples of character streams (2)

```
class ReadAnInt {
     public static void main(String[] args) throws IOException {
          int ch, value=0;
          Reader in = args.length==0 ?
               new InputStreamReader(System.in) :
               new FileReader(args[0]);
          for( ;(ch=in.read()!=-1; )
               value=10*value+ch-(int)'0';
          System.out.println(”Int read " + value);
     }
}
```

Buffered streams (1)

- Usually, input and output streams transfer data immediately.
- In order to make the reading/writing operations more efficient, one should use a buffer.

Buffered streams (2)



Buffered streams (3)

- **`FilterInputStream`**: subclass of `InputStream` that contains some other input stream, which it uses as its basic source of data, possibly transforming the data along the way or providing additional functionality.
- **`FilterOutputStream`**: subclass of `OutputStream` that simply overrides all its methods with versions that pass all requests to the underlying output stream.
- **`FilterReader`**: abstract subclass of `Reader` for reading filtered character streams.
- **`FilterWriter`**: asbtract subclass of `Writer` for writing filtered character streams.

Buffered streams (4)

- Usually, subclasses of `FilterInputStream`, `FilterOutputStream`, `FilterReader` and `FilterWriter` may further override some of the methods of these superclasses as well as provide additional methods and fields.

Buffered streams (5)

```
public class UppercaseConverter extends FilterReader {
    public UppercaseConverter(Reader in) {
        super(in);
    }
    public int read()
    throws IOException {
        int c = super.read();
        return (c==-1 ? c : Character.toUpperCase((char)c);
    }
    public int read(char[] buf, int offset, int count)
    throws IOException {
        int n = super.read(buf,offset,count);
        int ultimo = offset+n;
        for (int i=offset;i<ultimo;i++)
            buf[i] = Character.toUpperCase(buf[i]);
        return n;
    }
}
```

Buffered streams (6)

```
public static void main(String[] args) throws IOException {
     FileReader src = new FileReader(args[0]);
     FilterReader fr = new UppercaseConverter(src);
     int c;
     while ((c=f.read())!=-1)
          System.out.print((char)c);
     System.out.println();
}
```

Buffered streams (7)

- **`BufferedInputStream`**: subclass of `FilterInputStream` adds the ability to buffer the input and to support the `mark` and `reset` methods.
- **`BufferedOutputStream`**: subclass of `FilterOutputStream` that implements a buffered output stream.
- **`BufferedReader`**: subclass of `Reader` with buffered input and `mark` and `reset` methods.
- **`BufferedWriter`**: subclass of `Writer` with buffered output.

Buffered streams (8)

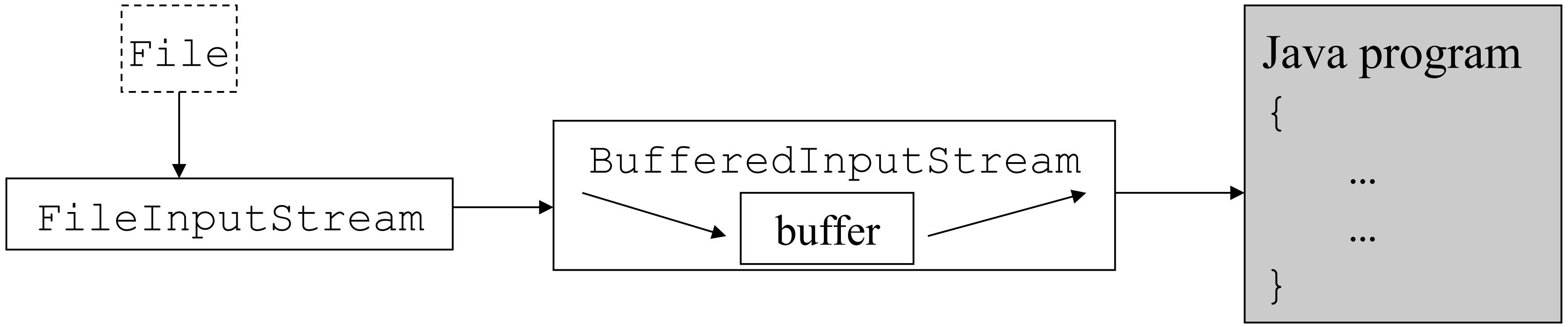
- Constructors of `BufferedInputStream` class:

```
BufferedInputStream(InputStream in)
BufferedInputStream(InputStream in, int size)
```

Creates a `BufferedInputStream` wrapping the input stream received as parameter. The `size` in the second constructor is related to the size of the buffer.

Buffered streams (9)

- **When a `read` is initially called in a `BufferedInputStream`, a `read` is invoked in the underlying input stream, filling the buffer (if there are sufficient data).**
- **New calls to the `read` method return data from the buffer. When all the buffer has been read, a `read` is invoked again in the underlying input stream.**
- **This process continues until the input is completely read.**



Buffered streams (10)

- Constructors of `BufferedOutputStream` class:

```
BufferedOutputStream(OutputStream out)
BufferedOutputStream(OutputStream out, int size)
```

Creates a `BufferedOutputStream` wrapping the output stream received as parameter. The `size` in the second constructor is related to the size of the buffer.

Buffered streams (11)

- Similarly to the `read` method:
  - **The write method of the `BufferedOutputStream` stores data from successive writes in the stream's buffer, flushing the buffer to the underlying output stream only when the buffer is full.**
  - **This buffer may contain data from several `write` methods invoked over a `BufferedInputStream`, so that several writes in the `BufferedInputStream` correspond to only one `write` in the underlying output stream.**

Buffered streams (12)

- The write is done via array of bytes:

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

- Before closing a buffered output stream, it is necessary to invoke `flush`, so that all bytes stored in the stream's buffer are flushed:

```
void flush()
```

- After the flush a close is needed:

```
void close()
```

Buffered streams (13)

- Constructors of `BufferedReader`:

```
BufferedReader(Reader in)
BufferedInputStream(Reader in, int size)
```

Creates a `BufferedReader` wrapping the reader received as parameter. The `size` in the second constructor is related to the size of the buffer.

- Constructors of `BufferedWriter`:

```
BufferedWriter(Writer out)
BufferedWriter(Writer out, int size)
```

Creates a `BufferedWriter` wrapping the writer received as parameter. The `size` in the second constructor is related to the size of the buffer.

Buffered streams (14)

- Some methods of `BufferedWriter` class:

`void newLine()`
Writes a new line that depends on the system (not necessarily `'\n'`).

Examples of buffered streams (1)

```
new BufferedInputStream(new FileInputStream("foo.in"));
```

```
new BufferedReader(new FileReader("foo.in"));
```

```
new BufferedOutputStream(new FileOutputStream("foo.out"));
```

```
new BufferedWriter(new FileWriter("foo.out"));
```

Examples of buffered streams (2)

```
import java.io.*;
public class Teste {
     public static void main(String[] args){
          BufferedOutputStream idOut;
          try {
               idOut = new BufferedOutputStream(
                    new FileOutputStream(args[0]));
               for(char c='a';c<='z';c++) {
                    idOut.write((byte)c);
                    idOut.write((byte)'\n');
               }
               idOut.flush();
               idOut.close();
          } catch(IOException e) {
             System.out.println(e.getMessage());
          }
     }
}
```

Examples of buffered streams (3)

```
BufferedReader stdin = new BufferedReader(
     new InputStreamReader(System.in));
try {
     String input = stdin.readLine();
     System.out.println(”Line: " + input);
} catch(IOException e) {
     System.err.println(“Error in reading”);
}
```

Data streams (1)

- **`DataInput`**: interface that provides reading bytes from a binary stream and reconstructing from them data in any of the Java primitive types and strings.
- **`DataOutput`**: interface that provides converting data from any of the Java primitive types and strings to a series of bytes and writing these bytes to a binary stream.
- **`DataInputStream`**: implementation of `DataInput`.
- **`DataOutputStream`**: implementation of `DataOutput`.
- **`RandomAccessFile`**: class that implements both `DataInput` e `DataOutput`, which support both reading and writing to a random access file.

Data streams (2)

- Read/write methods:

Type	Write	Read
`boolean`	`void writeBoolean(boolean)`	`boolean readBoolean()`
`char`	`void writeChar(char)`	`char readChar()`
`byte`	`void writeByte(byte)`	`byte readByte()`
`short`	`void writeShort(short)`	`short readShort()`
`int`	`void writeInt(int)`	`int readInt()`
`long`	`void writeLong(long)`	`long readLong()`
`float`	`void writeFloat(float)`	`float readFloat()`
`double`	`void writeDouble(double)`	`double readDouble()`
`String`	`void writeUTF(String)`	`String readUTF()`

Data stream example

```
import java.io.*;
public class Teste {
     public static void main(String[] args){
          DataOutputStream idOut;
          try {
               idOut = new DataOutputStream(
                    new FileOutputStream(args[0]));
               for(int k=0;k<10;k++)
                    idOut.writeUTF("Linha " + k + "\n");
               idOut.close();
          } catch(IOException e) {
               System.out.println(e.getMessage());
          }
     }
}
```

Random access file

- There are two standard ways for accessing a file:
  - **Sequential access** is limited by a sequential access order, from the beginning to the end of the file.
  - **Random access** allows arbitrary forward and backward access in the file.

RandomAccessFile class (1)

- **RandomAccessFile** implements both DataInput and DataOutput interfaces.
- Constructors of the RandonAccessFile class:

```
RandomAccessFile(File file, String mode)
RandomAccessFile(String name, String mode)
```

Creates a random access file stream to read from, and optionally to write to, the file specified by the File/String argument. The mode argument specifies the access mode in which the file is to be opened. The permitted values are: "r" (for reading) or "rw" (reading and writing), among others

`RandomAccessFile` class (2)

- Some reading methods:

```
int read()
int read(byte[])
int read(byte[], int offset, int length)
boolean readBoolean()
byte readByte()
char readChar()
double readDouble()
float readFloat()
int readInt()
long readLong()
short readShort()
String readUTF()
```

`RandomAccessFile` class (3)

- Some writing methods:

```
void write(byte[])
void write(byte[], int offset, int length)
void writeBoolean(boolean b)
void writeByte(int v)
void writeBytes(String s)
void writeChar(int v)
void writeChars(String s)
void writeDouble(double v)
void writeFloat(float v)
void writeInt(int v)
void writeLong(long v)
void writeShort(int v)
void writeUTF(String str)
```

`RandomAccessFile` class (4)

- Some positioning method:

`void seek(long pos)`
Sets the file-pointer at `pos`, measured from the beginning of this file, at which the next read or write occurs.

`long length()`
Returns the length of the file.

`long setLength(long newLength)`
Sets the length of this file.

`int skipBytes(int n)`
Attempts to skip over `n` bytes of input discarding the skipped bytes. The actual number of bytes skipped is returned.

`RandomAccessFile` class (5)

```
File data = new File("..","info");
RandomAcessFile f = new RandomAcessFile(data, "rw");
for(int i=65;i<91;i++)
     f.write(i) // write the alphabet
byte[] ler = new byte[10];
f.seek(4); // jumps for the 5th byte
f.read(ler,0,5); // read the first 5 positions
for(int=0;i<5;i++)
     System.out.println((char)ler[i] + ":");
f.seek(3); // jumps for the 4th byte
System.out.println((char)f.read());
```

In the terminal is printed `E:F:G:H:I:D`

Object streams

- `ObjectInputStream`: subclass of `InputStream` for reading serialized objects.
- `ObjectOutputStream`: subclass of `OutputStream` to serialize objects.
- `Serializable`: interface that must be implemented by classes that require serialization.
- **The `ObjectInputStream` and `ObjectOutputStream` classes, when used jointly with `FileInputStream` and `FileOutputStream`, respectively, allow to store the objects in an application in a persistent way.**

`ObjectInputStream` class (1)

- Some constructores of `ObjectInputStream`:

`ObjectInputStream(InputStream in)`
Creates an `ObjectInputStream` wrapping the input stream argument.

- Some methods of `ObjectInputStream`:

`boolean defaultReadObject()`
Reads the non-static and non-transient fields of the current class from this stream.
`Object readObject()`
Reads the object from the input stream.

- **Classes that require atypical serialization should implement:**
  - **`private void readObject(ObjectInputStream stream) throws IOException, ClassNotFoundException;`**

`ObjectInputStream` class (2)

- Other reading methods:

```
int read()
int read(byte[], int offset, int length)
boolean readBoolean()
byte readByte()
char readChar()
double readDouble()
float readFloat()
int readInt()
long readLong()
short readShort()
String readUTF()
```

`ObjectOutputStream` class (1)

- Some constructors of `ObjectOutputStream`:

```
ObjectOutputStream(OutputStream out)
```
Creates an `ObjectOutputStream` wrapping the outpup stream argument.

- Some methods of `ObjectInputStream`:

```
protected boolean defaultWriteObject()
```
Writes the non-static and non-transient fields of the current class to this stream.

```
void writeObject(Object obj)
```
Writes the object the output stream.

- **Classes that require atypical serialization should implement:**

  -
    ```
    private void writeObject(ObjectOutputStream stream)
    throws IOException, ClassNotFoundException;
    ```

`ObjectOutputStream` class (2)

- Other writing methods:

```
void write(byte[])
void write(byte[], int offset, int length)
void writeBoolean(boolean b)
void writeByte(int v)
void writeBytes(String s)
void writeChar(int v)
void writeChars(String s)
void writeDouble(double v)
void writeFloat(float v)
void writeInt(int v)
void writeLong(long v)
void writeShort(int v)
void writeUTF(String str)
```

Examples of object streams (1)

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

```
FileInputStream fis = new FileInputStream("foo.out");
ObjectInputStream ois = new ObjectInputStream(fis);
int i = ois.readInt();
String hoje = (String) ois.readObject();
Date data = (Date) ois.readObject();
ois.close();
fis.close();
```

Examples of object streams (2)

- Serialization of an `HashMap` object in a file for future use:

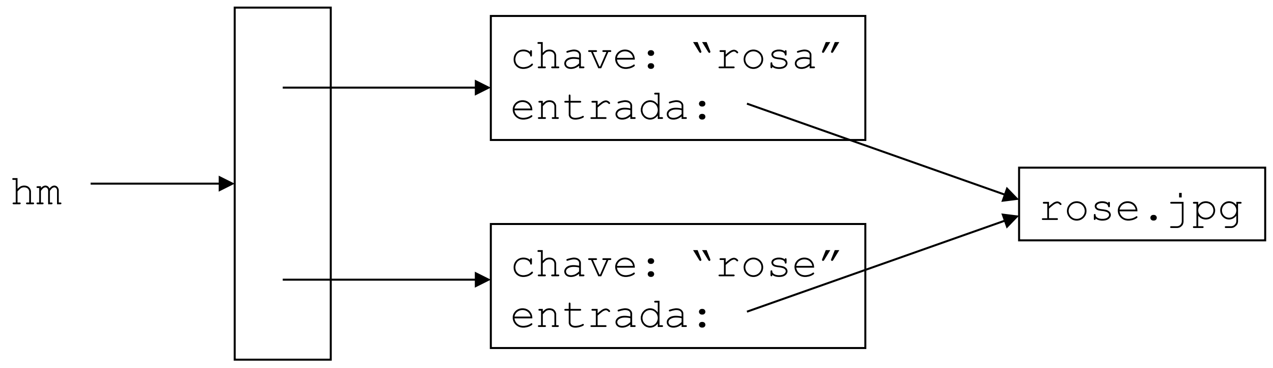
```
FileOutputStream fos = new FileOutputStream(“hm.out”);
ObjectOutputStream oos = new ObjectOutputStream(fos);
HashMap<?,?> hm = getHashMap();
oos.writeObject(hm);
```

- A new copy of the object may be loaded from the previous serialization:

```
FileInputStream fis = new FileInputStream(“hm.out”);
ObjectInputStream ois = new ObjectInputStream(fis);
HashMap<?,?> hm = (HashMap<?,?>) ois.readObject();
```

Serialization with shared references (1)

- The serialization preserves the integrity of the graph of objects in memory:



- When the `HashMap` is loaded again into memory from the previous serialization, there will be two references to the same object `rose.jpp`, and not two references to two different copies of `rose.jpg`.

Serialization with shared references (2)

- When this is not desired one can use the following methods:

`void writeUnshared(Object obj)`

Writes an unshared object to the `ObjectOutputStream` . This method is identical to `writeObject`, except that it always writes the given object as a new, unique object in the stream (as opposed to a back-reference pointing to a previously serialized instance).

`Object readUnshared()`

Reads an unshared object from the `ObjectInputStream`. If `readUnshared` is called to deserialize a back-reference (the stream representation of an object which has been written previously to the stream), an `ObjectStreamException` will be thrown.

`Serializable` interface (1)

- An object can be written in an object stream only if its class implement the **`Serializable`** interface.
- If an attribute value is not intended to be serialized, the `transient` modifier should be used (in this case the null iterator is sent to the object stream).
- The `Serializable` interface has no methods or attributes.

`Serializable` interface (2)

```
public class Nome implements java.io.Serializable {
    private String name;
    private long id;
    private transient boolean hashComputed = false;
    private transient int hash;
    private static long nextID = 0;
    public Name(String name) {
        this.name = name;
        id = nextId++;
    }
    public int hashCode() {
        if (!hashComputed) {
            hash = name.hashCode();
            hashComputed = true;
        }
        return hash;
    }
    //overriding equals and other methods…
}
```

Serializable interface (3)

```
public class Name implements java.io.Serializable {
    private String name;
    private long id;
    private transient int hash;
    private static long proxID = 0;
    public Nome(String name) {
        this.name = name;
        id = nextId++;
        hash = name.hashCode();
    }
    private void writeObject(ObjectOutputStream out)
    throws IOException {
        out.writeUTF(nome);
        out.writeLong(id);
    }
    // ... continues in the next slide
```

Serializable interface (4)

```java
    private void readObject(ObjectInputStream in)
    throws IOException, ClassNotFoubdException {
        nome = in.readUTF();
        id = in.readLong();
        hash = name.hashCode();
    }
    public int hashCode() {
        return hash;
    }
    //overriding equals and other methods…
}
```

`Serializable` interface (5)

```
private void writeObject(ObjectOutputStream out)
throws IOException {
     out.defaultWriteObject();
}
```

```
private void readObject(ObjectInputStream in)
throws IOException, ClassNotFoubdException {
     in.defaultReadObject();
     hash = name.hashCode();
}
```