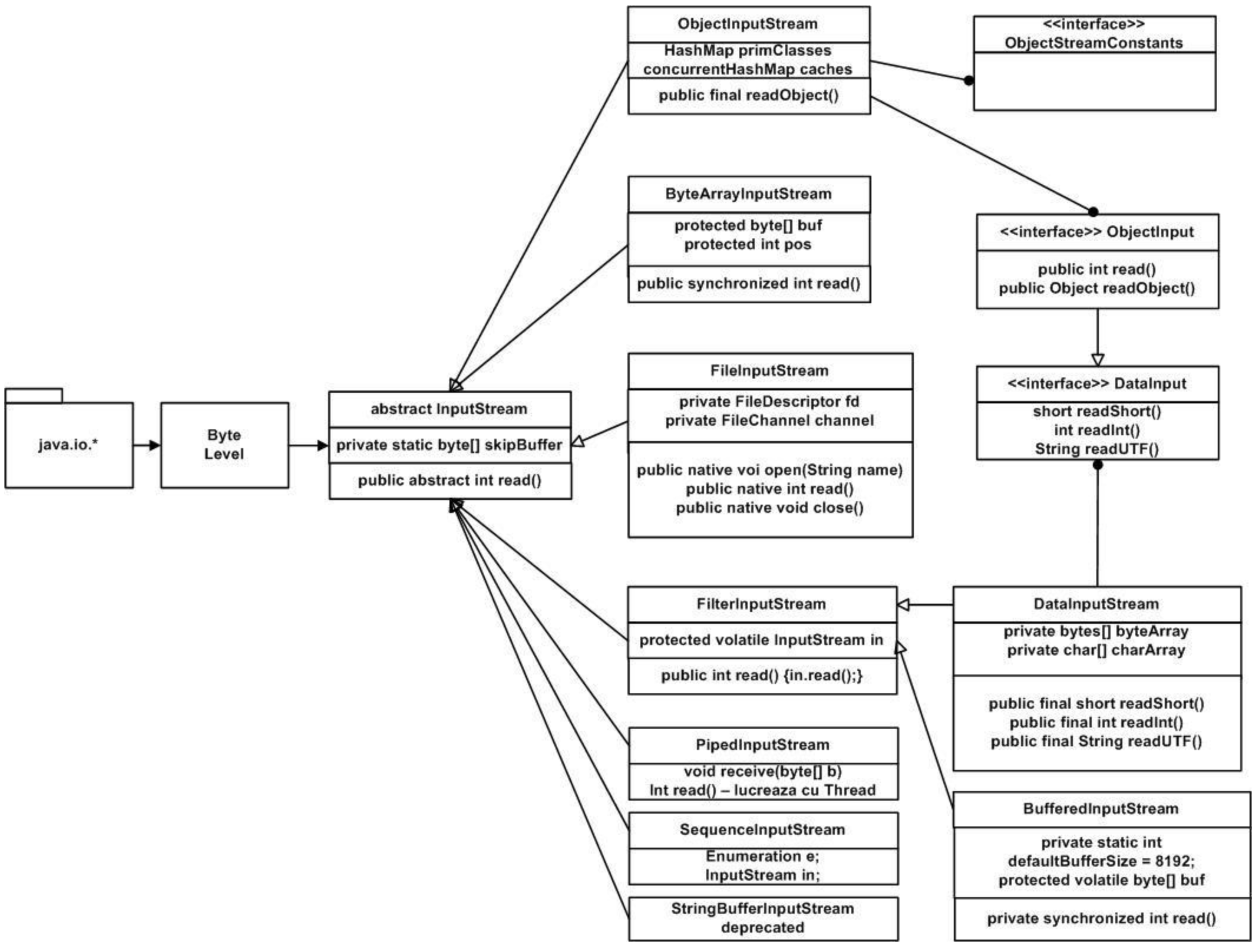
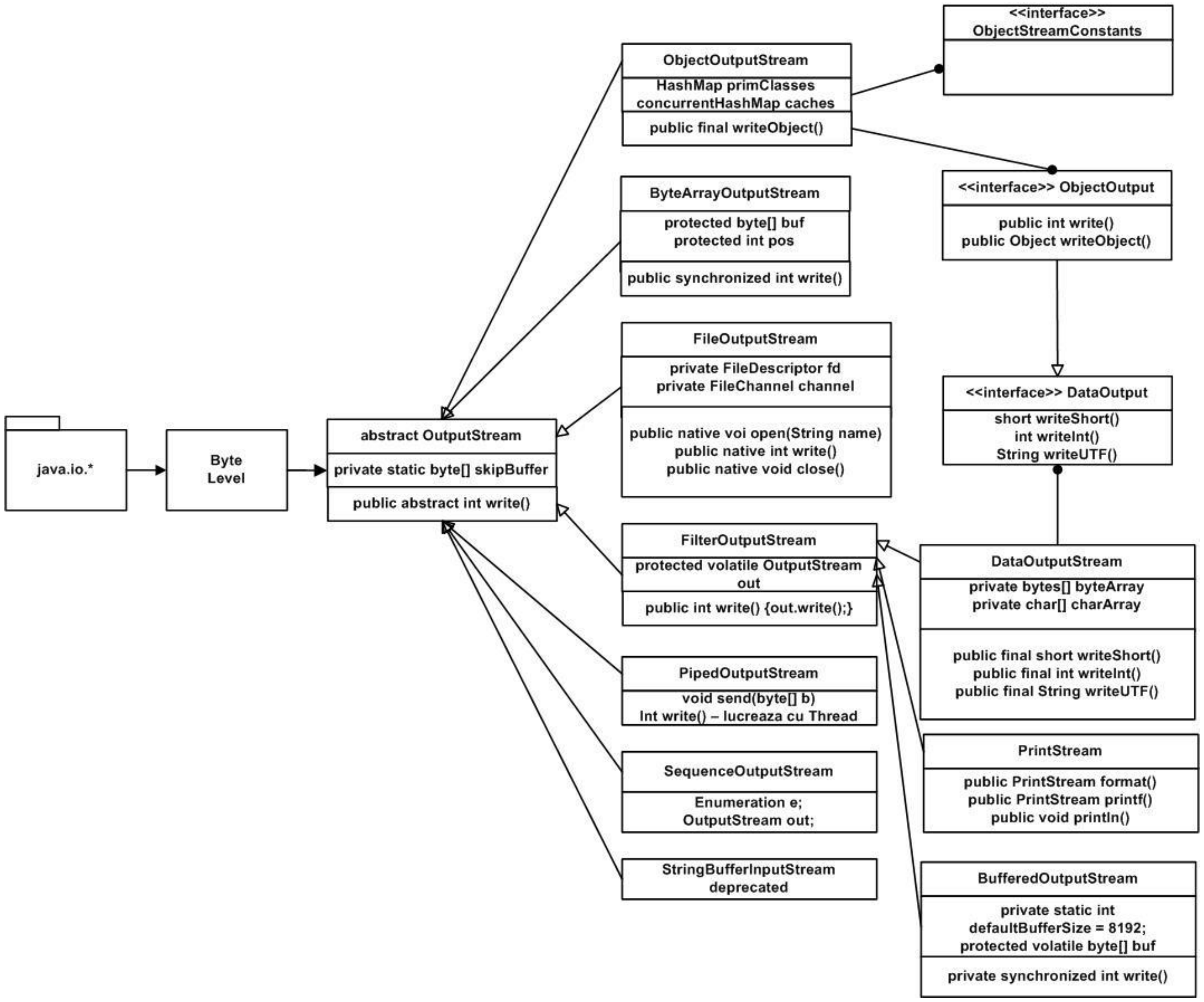
Java Basic IO

1. Lucru cu **Interfete**, **Fluxuri de intrare/iesire la nivel de octet & caracter (char=2 bytes)**

2. Ierarhia de clase bazata pe **fluxuri de I/O la nivel de octet**

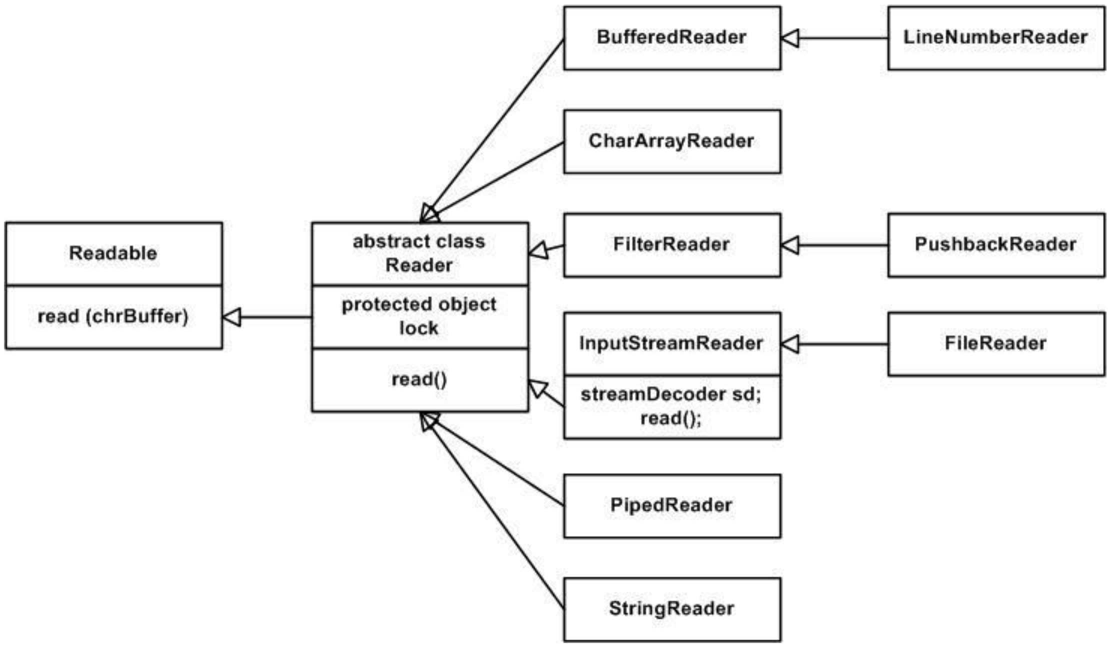


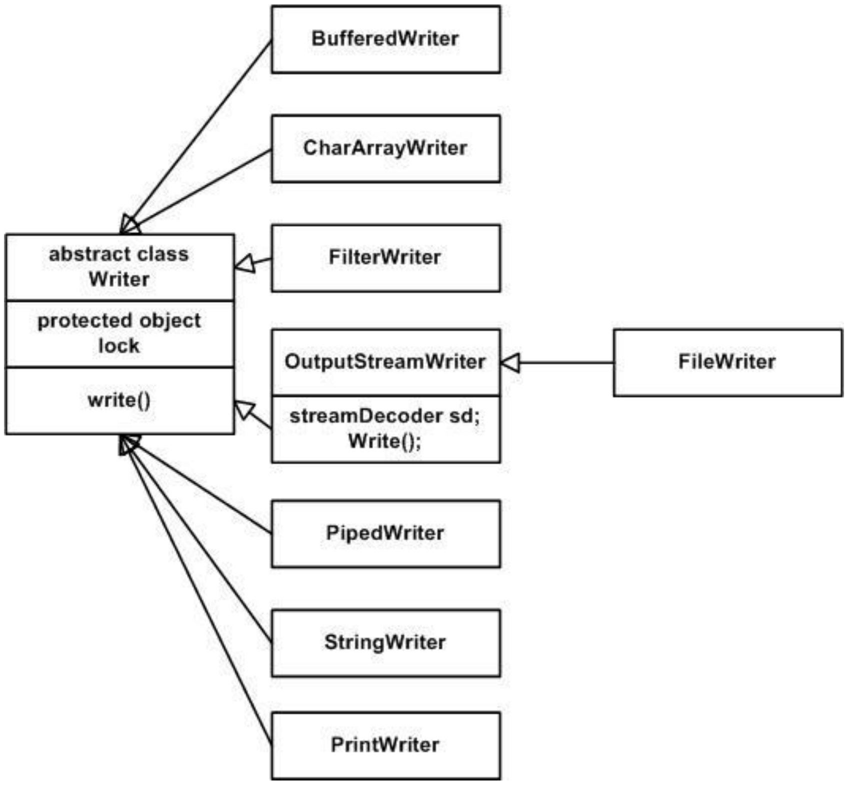
ObjectInputStream
HashMap primClasses
concurrentHashMap caches
public final readObject()
<<interface>> ObjectStreamConstants
ByteArrayInputStream
protected byte[] buf
protected int pos
public synchronized int read()
<<interface>> ObjectInput
public int read()
public Object readObject()
FileInputStream
private FileDescriptor fd
private FileChannel channel
public native voi open(String name)
public native int read()
public native void close()
<<interface>> DataInput
short readShort()
int readInt()
String readUTF()
java.io.*
Byte Level
abstract InputStream
private static byte[] skipBuffer
public abstract int read()
FilterInputStream
protected volatile InputStream in
public int read() {in.read();}
DataInputStream
private bytes[] byteArray
private char[] charArray
public final short readShort()
public final int readInt()
public final String readUTF()
PipedInputStream
void receive(byte[] b)
Int read() – lucreaza cu Thread
SequenceInputStream
Enumeration e;
InputStream in;
StringBufferInputStream
deprecated
BufferedInputStream
private static int
defaultBufferSize = 8192;
protected volatile byte[] buf
private synchronized int read()


java.io.*
Byte Level
abstract OutputStream
private static byte[] skipBuffer
public abstract int write()
ObjectOutputStream
HashMap primClasses
concurrentHashMap caches
public final writeObject()
ByteArrayOutputStream
protected byte[] buf
protected int pos
public synchronized int write()
FileOutputStream
private FileDescriptor fd
private FileChannel channel
public native voi open(String name)
public native int write()
public native void close()
FilterOutputStream
protected volatile OutputStream out
public int write() {out.write();}
PipedOutputStream
void send(byte[] b)
Int write() – lucreaza cu Thread
SequenceOutputStream
Enumeration e;
OutputStream out;
StringBufferInputStream deprecated
<<interface>> ObjectStreamConstants
<<interface>> ObjectOutput
public int write()
public Object writeObject()
<<interface>> DataOutput
short writeShort()
int writeInt()
String writeUTF()
DataOutputStream
private bytes[] byteArray
private char[] charArray
public final short writeShort()
public final int writeInt()
public final String writeUTF()
PrintStream
public PrintStream format()
public PrintStream printf()
public void println()
BufferedOutputStream
private static int defaultBufferSize = 8192;
protected volatile byte[] buf
private synchronized int write()

3. Ierarhia de clase bazata pe **fluxuri de I/O la nivel de caracter**

InputStreamReader / OutputStreamReader ⬄ puntea de legatura intre fluxul de octeti si cel de caractere.





4. Buffered Streams
    a. //character stream + buffered streams (flush - property)
    b. //Buffered: BufferedInputStream / BufferedOutputStream <=> 4 byte level
    c. //   BufferedReader   / BufferedWriter <=> 4 character level

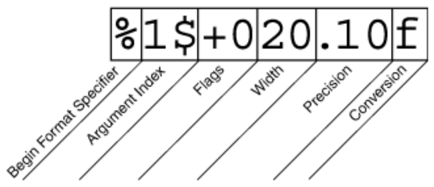
5. Scanning (din JDK 6 apare clasa ‘Scanner’)

```
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();
            }
        }
    }
}
```

6. Formatare textului (System.out.format – din JDK 5):

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

The additional elements are all optional. The following figure shows how the longer specifier breaks down into elements.



Elements of a Format Specifier.

The elements must appear in the order shown. Working from the right, the optional elements are:

- **Precision**. For floating point values, this is the mathematical precision of the formatted value. For `s` and other general conversions, this is the maximum width of the formatted value; the value is right-truncated if necessary.
- **Width**. The minimum width of the formatted value; the value is padded if necessary. By default the value is left-padded with blanks.
- **Flags** specify additional formatting options. In the `Format` example, the `+` flag specifies that the number should always be formatted with a sign, and the `0` flag specifies that `0` is the padding character. Other flags include `-` (pad on the right) and `,` (format number with locale-specific thousands separators). Note that some flags cannot be used with certain other flags or with certain conversions.
- The **Argument Index** allows you to explicitly match a designated argument. You can also specify `<` to match the same argument as the previous specifier. Thus the example could have said:

```
System.out.format("%f, %<+020.10f %n", Math.PI);
```

7. Din JDK 6 apare clasa ‘Console’:

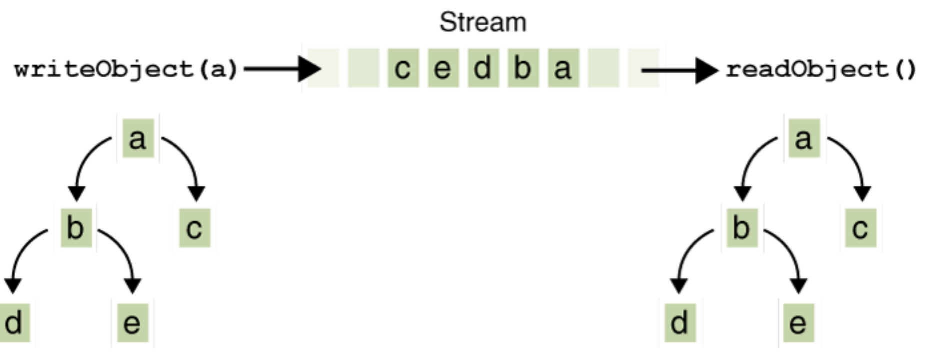
```
import java.io.Console;
import java.util.Arrays;
import java.io.IOException;

public class Password {

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

        Console c = System.console();
        if (c == null) {
            System.err.println("No console.");
            System.exit(1);
        }

        String login = c.readLine("Enter your login: ");
        char [] oldPassword = c.readPassword("Enter your old password: ");

        if (verify(login, oldPassword)) {
            boolean noMatch;
            do {
                char [] newPassword1 =
                    c.readPassword("Enter your new password: ");
                char [] newPassword2 =
                    c.readPassword("Enter new password again: ");
                noMatch = ! Arrays.equals(newPassword1, newPassword2);
                if (noMatch) {
                    c.format("Passwords don't match. Try again.%n");
                } else {
```

```
                change(login, newPassword1);
                c.format("Password for %s changed.%n", login);
            }
            Arrays.fill(newPassword1, ' ');
            Arrays.fill(newPassword2, ' ');
        } while (noMatch);
    }

    Arrays.fill(oldPassword, ' ');

  }

  //Dummy verify method.
  static boolean verify(String login, char[] password) {
    return true;
  }

  //Dummy change method.
  static void change(String login, char[] password) {}
}
```

8. DataInputStream si DataOutputStream – sunt implementari i/O pentru interfetele DataInput si DataOutput, pentru scrierea in flux de octeti a tipurilor de date fundamentale. Problema apare cand se doreste scrierea ‘float a=0.1’. Se recomanda utilizarea clasei BigDecimal dar pentru aceasta este nevoie de scrierea unui obiect in fluxul de octeti.

9. Object Streams & Serialization

This is demonstrated in the following figure, where `writeObject` is invoked to write a single object named **a**. This object contains references to objects **b** and **c**, while **b** contains references to **d** and **e**. Invoking `writeobject(a)` writes not just **a**, but all the objects necessary to reconstitute **a**, so the other four objects in this web are written also. When **a** is read back by `readObject`, the other four objects are read back as well, and all the original object references are preserved.



I/O of multiple referred-to objects

You might wonder what happens if two objects on the same stream both contain references to a single object. Will they both refer to a single object when they're read back? The answer is **"yes."** A stream can only contain one copy of an object, though it can contain any number of references to it. Thus if you explicitly write an object to a stream twice, you're really writing only the reference twice. For example, if the following code writes an object `ob` twice to a stream:

```
Object ob = new Object();
out.writeObject(ob);
out.writeObject(ob);
```

Each **`writeObject`** has to be matched by a **readObject**, so the code that reads the stream back will look something like this:

```
Object ob1 = in.readObject();
Object ob2 = in.readObject();
```

This results in two variables, **`ob1`** and **`ob2`**, that are references to a single object.

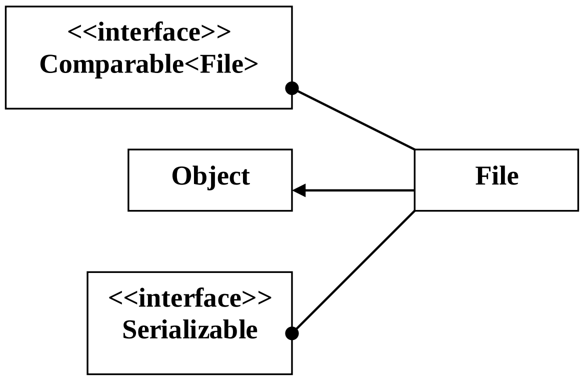
However, if a single object is written to two different streams, it is effectively duplicated — a single program reading both streams back will see **two distinct objects**.

- La salvare obiectului se serializeaza doar signature clase (tipul membrilor + signatura metodelor) drept dovada ca in Deserializare1.java este aceeasi metoda "afiseaza()" cu aceeasi signatura dar cu alt corp

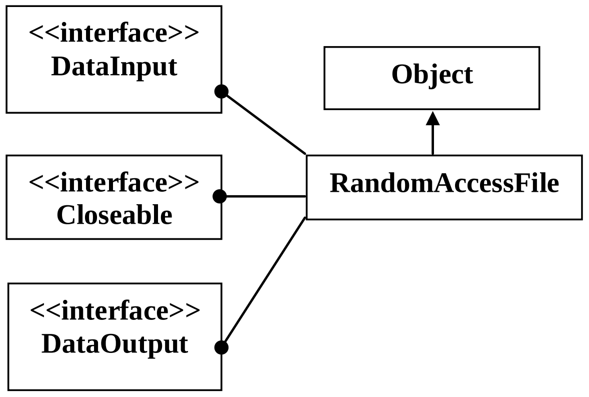
- Daca se adauga un camp nou in ObiectSimplu atunci clasele nu mai sunt compatibile la serializare/deserializare

-

10. RandomAccessFile si File



<<interface>> Comparable<File>
Object
File
<<interface>> Serializable



<<interface>> DataInput
Object
<<interface>> Closeable
RandomAccessFile
<<interface>> DataOutput