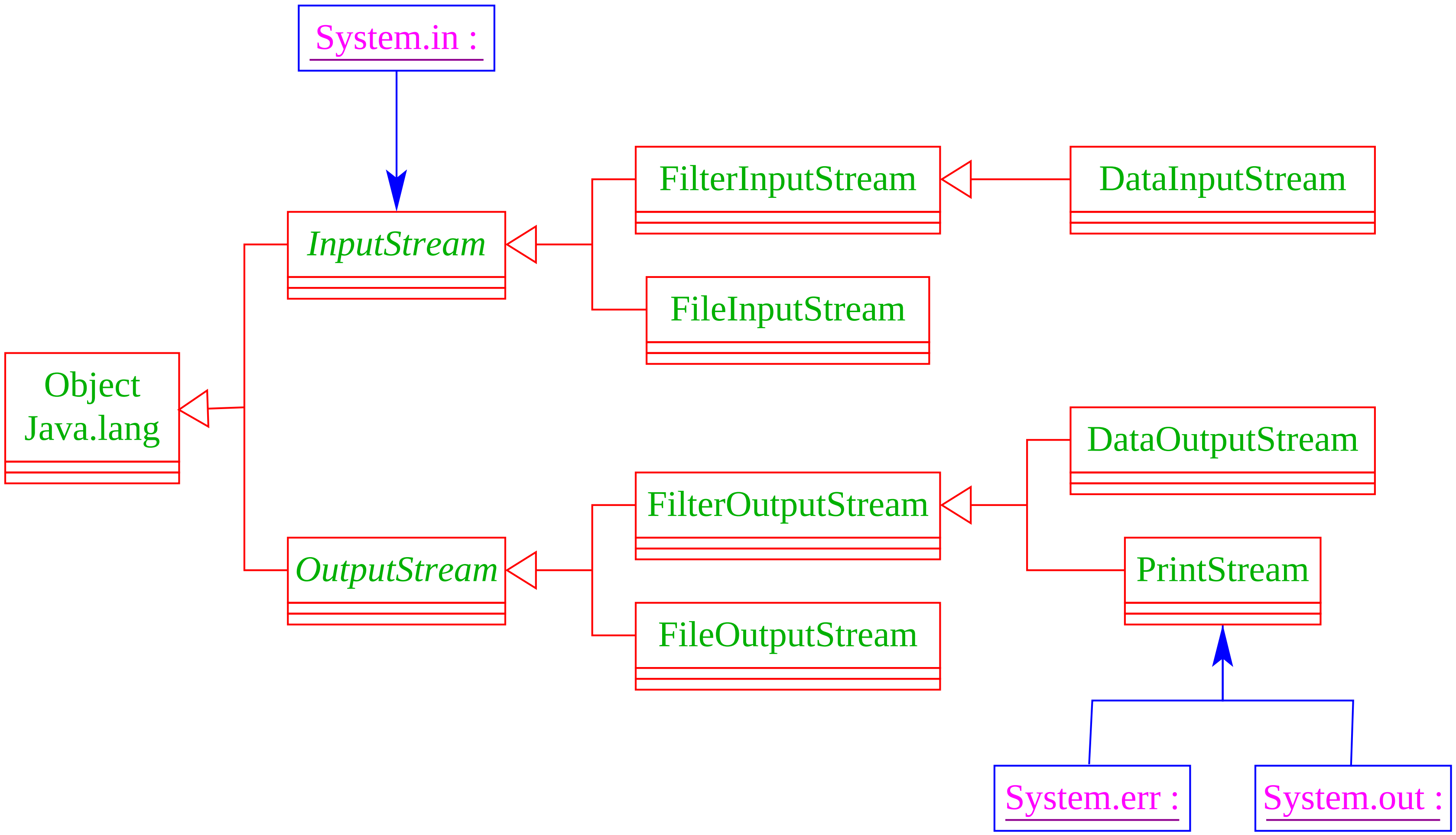
Problem 2:

Wie schreibt bzw. wie liest man die beiden unterschiedlichen Darstellungen?

Dazu stellt Java im Paket `java.io` eine Vielzahl von Klassen zur Verfügung ...

15.1 Byteweise Ein- und Ausgabe

Zuerst eine (unvollständige :-) Übersicht ...


System.in :
InputStream
FilterInputStream
DataInputStream
FileInputStream
Object
Java.lang
DataOutputStream
FilterOutputStream
PrintStream
OutputStream
FileOutputStream
System.err :
System.out :

- Die grundlegende Klasse für `byte`-Eingabe heißt `InputStream`.
- Diese Klasse ist abstrakt.
- Trotzdem ist `System.in` ein Objekt dieser Klasse :-)

Nützliche Operationen:

- `public int available()` :
  gibt die Anzahl der vorhandenen Bytes an;
- `public int read() throws IOException` :
  liest ein Byte vom Input als `int`-Wert; ist das Ende des Stroms erreicht, wird -1 geliefert;
- `void close() throws IOException` : schließt den Eingabe-Strom.

Achtung:

- `System.in.available()` liefert stets `0`.
- Außer `System.in` gibt es keine Objekte der Klasse `InputStream`
- ... außer natürlich Objekte von Unterklassen.

Konstruktoren von Unterklassen:

- `public FileInputStream(String path) throws IOException`
  öffnet die Datei `path`;
- `public DataInputStream(InputStream in)`
  liefert für einen InputStream `in` einen `DataInputStream`.

Beispiel:

```
import java.io.*;
public class FileCopy {
public static void main(String[] args) throws IOException {
   FileInputStream file = new FileInputStream(args[0]);
   int t;
   while(-1 != (t = file.read()))
      System.out.print((char)t);
   System.out.print("\n");
   } // end of main
} // end of FileCopy
```

- Das Programm interpretiert das erste Argument in der Kommando-Zeile als Zugriffspfad auf eine Datei.
- Sukzessive werden Bytes gelesen und als `char`-Werte interpretiert wieder ausgegeben.
- Das Programm terminiert, sobald das Ende der Datei erreicht ist.

Achtung:

- `char`-Werte sind intern 16 Bit lang ...
- Ein Latin-1-Text wird aus dem Input-File auf die Ausgabe geschrieben, weil ein Byte/Latin-1-Zeichen `xxxx xxxx`
  - intern als `0000 0000 xxxx xxxx` abgespeichert und dann
  - extern als `xxxx xxxx` ausgegeben wird :-)

Erweiterung der Funktionalität:

- In der Klasse `DataInputStream` gibt es spezielle Lese-Methoden für jeden Basis-Typ.

Unter anderem gibt es:

- `public byte readByte() throws IOException;`
- `public char readChar() throws IOException;`
- `public int readInt() throws IOException;`
- `public double readDouble() throws IOException.`

Beispiel:

```
import java.io.*;
public class FileCopy {
public static void main(String[] args) throws IOException {
   FileInputStream file = new FileInputStream(args[0]);
   DataInputStream data = new DataInputStream(file);
   int n = file.available(); char x;
   for(int i=0; i<n/2; ++i) {
      x = data.readChar();
      System.out.print(x);
      }
   System.out.print("\n");
   } // end of main
} // end of erroneous FileCopy
```

... führt i.a. zur Ausgabe: ?????????

Der Grund:

- `readChar()` liest nicht ein Latin-1-Zeichen (i.e. 1 Byte), sondern die 16-Bit-Repräsentation eines Unicode-Zeichens ein.
- Das Unicode-Zeichen, das zwei Latin-1-Zeichen hintereinander entspricht, ist (i.a.) auf unseren Bildschirmen nicht darstellbar. Deshalb die Fragezeichen ...

- Analoge Klassen stehen für die Ausgabe zur Verfügung.
- Die grundlegende Klasse für `byte`-Ausgabe heißt `OutputStream`.
- Auch `OutputStream` ist abstrakt :-)

Nützliche Operationen:

- `public void write(int b) throws IOException` : schreibt das unterste Byte von `b` in die Ausgabe;
- `void flush() throws IOException` : falls die Ausgabe gepuffert wurde, soll sie nun ausgegeben werden;
- `void close() throws IOException` : schließt den Ausgabe-Strom.

- Weil `OutputStream` abstrakt ist, gibt es keine Objekte der Klasse `OutputStream`, nur Objekte von Unterklassen.

Konstruktoren von Unterklassen:

- `public FileOutputStream(String path) throws IOException;`
- `public FileOutputStream(String path, boolean append) throws IOException;`
- `public DataOutputStream(OutputStream out);`
- `public PrintStream(OutputStream out)` — der Rückwärts-Kompatibilität wegen, d.h. um Ausgabe auf `System.out` und `System.err` zu machen ...

Beispiel:

```
import java.io.*;
public class File2FileCopy {
public static void main(String[] args) throws IOException {
   FileInputStream fileIn = new FileInputStream(args[0]);
   FileOutputStream fileOut = new FileOutputStream(args[1]);
   int n = fileIn.available();
   for(int i=0; i<n; ++i)
      fileOut.write(fileIn.read());
   fileIn.close(); fileOut.close();
   System.out.print("\t\tDone!!!\n");
   } // end of main
} // end of File2FileCopy
```

- Das Programm interpretiert die 1. und 2. Kommando-Zeilen-Argumente als Zugriffspfade auf eine Ein- bzw. Ausgabe-Datei.
- Die Anzahl der in der Eingabe enthaltenen Bytes wird bestimmt.
- Dann werden sukzessive die Bytes gelesen und in die Ausgabe-Datei geschrieben.

Erweiterung der Funktionalität:

Die Klasse `DataOutputStream` bietet spezielle Schreib-Methoden für verschiedene Basis-Typen an.

Beispielsweise gibt es:

- `void writeByte(int x) throws IOException;`
- `void writeChar(int x) throws IOException;`
- `void writeInt(int x) throws IOException;`
- `void writeDouble(double x) throws IOException.`

Beachte:

- `writeChar()` schreibt genau die Repräsentation eines Zeichens, die von `readChar()` verstanden wird, d.h. 2 Byte.

Beispiel:

```
import java.io.*;
public class Numbers {
public static void main(String[] args) throws IOException {
   FileOutputStream file = new FileOutputStream(args[0]);
   DataOutputStream data = new DataOutputStream(file);
   int n = Integer.parseInt(args[1]);
   for(int i=0; i<n; ++i)
      data.writeInt(i);
   data.close();
   System.out.print("\t\tDone!\n");
   } // end of main
} // end of Numbers
```

- Das Programm entnimmt der Kommando-Zeile den Datei-Pfad sowie eine Zahl `n`.
- Es wird ein `DataOutputStream` für die Datei eröffnet.
- In die Datei werden die Zahlen 0,...,`n-1` binär geschrieben.
- Das sind also 4`n` Bytes.

Achtung:

- In der Klasse `System` sind die zwei vordefinierten Ausgabe-Ströme `out` und `err` enthalten.
- `out` repräsentiert die Ausgabe auf dem Terminal.
- `err` repräsentiert die Fehler-Ausgabe für ein Programm (i.a. ebenfalls auf dem Terminal).
- Diese sollen jedes Objekt als Text auszugeben können.
- Dazu dient die Klasse `PrintStream`.

- Die `public`-Objekt-Methoden `print()` und `println()` gibt es für jeden möglichen Argument-Typ.
- Für (Programmierer-definierte) Klassen wird dabei auf die `toString()`-Methode zurückgegriffen.
- `println()` unterscheidet sich von `print`, indem nach Ausgabe des Arguments eine neue Zeile begonnen wird.

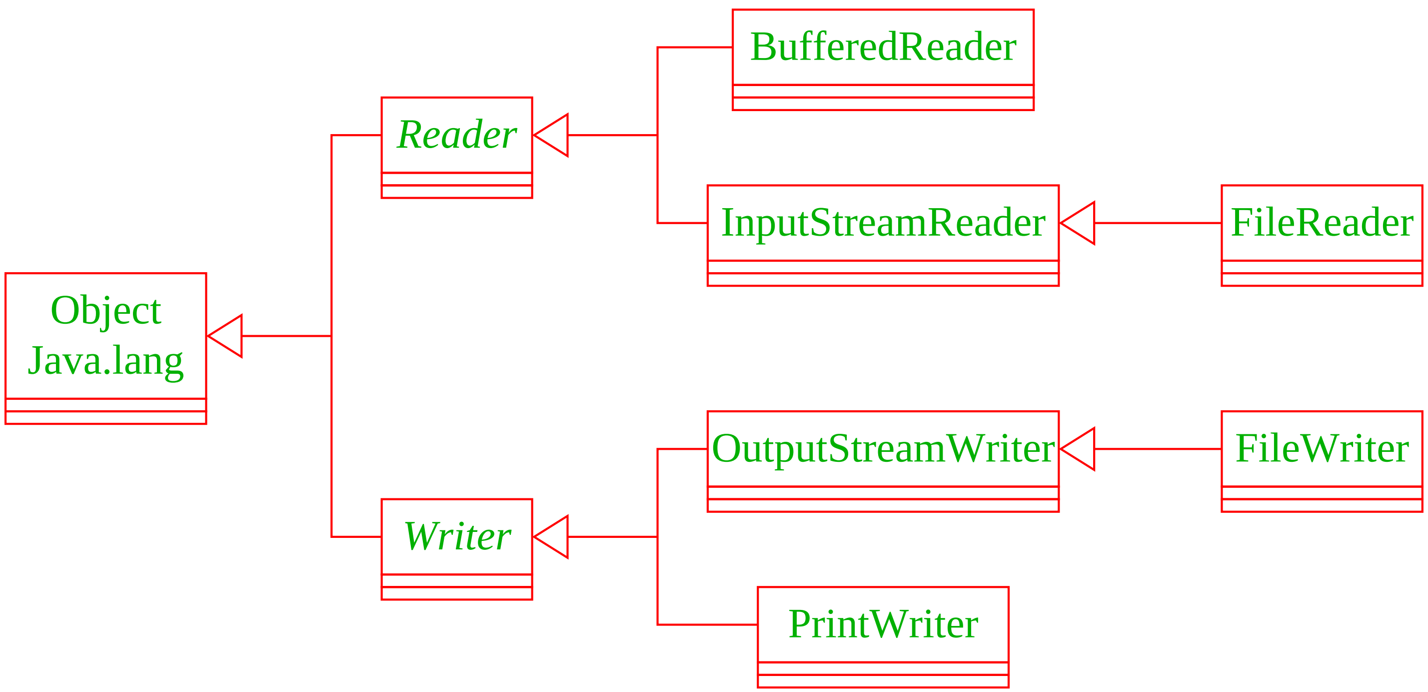
Achtung:

- `PrintStream` benutzt (neuerdings) die Standard-Codierung des Systems, d.h. bei uns Latin-1.
- Um aus einem (Unicode-)String eine Folge von sichtbaren (Latin-1-)Zeichen zu gewinnen, wird bei jedem Zeichen, das kein Latin-1-Zeichen darstellt, ein '?' gedruckt ...

$\Longrightarrow$ für echte (Unicode-) Text-Manipulation schlecht geeignet ...

15.2 Textuelle Ein- und Ausgabe

Wieder erstmal eine Übersicht über hier nützliche Klassen ...



- Die Klasse `Reader` ist abstrakt.

Wichtige Operationen:

- `public boolean ready() throws IOException :`
  liefert `true`, sofern ein Zeichen gelesen werden kann;
- `public int read() throws IOException :`
  liest ein (Unicode-)Zeichen vom Input als `int`-Wert; ist das Ende des Stroms erreicht, wird -1 geliefert;
- `void close() throws IOException :` schließt den Eingabe-Strom.

- Ein `InputStreamReader` ist ein spezieller Textleser für Eingabe-Ströme.
- Ein `FileReader` gestattet, aus einer Datei Text zu lesen.
- Ein `BufferedReader` ist ein `Reader`, der Text zeilenweise lesen kann, d.h. eine zusätzliche Methode `public String readLine() throws IOException;` zur Verfügung stellt.

Konstruktoren:

- `public InputStreamReader(InputStream in);`
- `public InputStreamReader(InputStream in, String encoding) throws IOException;`
- `public FileReader(String path) throws IOException;`
- `public BufferedReader(Reader in);`

Beispiel:

```
import java.io.*;
public class CountLines {
public static void main(String[] args) throws IOException {
   FileReader file = new FileReader(args[0]);
   BufferedReader buff = new BufferedReader(file);
   int n=0; while(null != buff.readLine()) n++;
   buff.close();
   System.out.print("Number of Lines:\t\t"+ n);
   } // end of main
} // end of CountLines
```

- Die Objekt-Methode `readLine()` liefert `null`, wenn beim Lesen das Ende der Datei erreicht wurde.
- Das Programm zählt die Anzahl der Zeilen einer Datei :-)

- Wieder stehen analoge Klassen für die Ausgabe zur Verfügung.
- Die grundlegende Klasse für textuelle Ausgabe heißt `Writer`.

Nützliche Operationen:

- `public void write(` *type* `x) throws IOException` :
  gibt es für die Typen `int` (dann wird ein einzelnes Zeichen geschrieben), `char[]` sowie `String`.
- `void flush() throws IOException` :
  falls die Ausgabe gepuffert wurde, soll sie nun tatsächlich ausgegeben werden;
- `void close() throws IOException` : schließt den Ausgabe-Strom.

- Weil `Writer` abstrakt ist, gibt keine Objekte der Klasse `Writer`, nur Objekte von Unterklassen.

Konstruktoren:

- `public OutputStreamWriter(OutputStream str);`
- `public OutputStreamWriter(OutputStream str, String encoding);`
- `public FileWriter(String path) throws IOException;`
- `public FileWriter(String path, boolean append) throws IOException;`
- `public PrintWriter(OutputStream out);`
- `public PrintWriter(Writer out);`

Beispiel:

```
import java.io.*;
public class Text2TextCopy {
public static void main(String[] args) throws IOException {
   FileReader fileIn = new FileReader(args[0]);
   FileWriter fileOut = new FileWriter(args[1]);
   int c = fileIn.read();
   for(; c!=-1; c=fileIn.read())
      fileOut.write(c);
   fileIn.close(); fileOut.close();
   System.out.print("\t\tDone!!!\n");
   } // end of main
} // end of Text2TextCopy
```

Wichtig:

- Ohne einen Aufruf von `flush()` oder `close()` kann es passieren, dass das Programm beendet wird, bevor die Ausgabe in die Datei geschrieben wurde :-(
- Zum Vergleich: in der Klasse `OutputStream` wird `flush()` automatisch nach jedem Zeichen aufgerufen, das ein Zeilenende markiert.

Bleibt, die zusätzlichen Objekt-Methoden für einen `PrintWriter` aufzulisten...

- Analog zum `PrintStream` sind dies
  `public void print(` *type* `x);` und
  `public void println(` *type* `x);`

- ... die es gestatten, Werte jeglichen Typs (aber nun evt. in geeigneter Codierung) auszudrucken.

Text-Ein- und Ausgabe arbeitet mit (Folgen von) Zeichen.

Nicht schlecht, wenn man dafür die Klasse `String` etwas näher kennen würde ...