Vstupy a výstupy v Javě

Table of Contents

Koncepce vstupně/výstupních operací v Javě

- V/V operace jsou založeny na *vstupně/výstupních proudech* (streams).
- Tím pádem je možno značnou část logiky programu psát nezávisle na tom, o který *konkrétní typ* V/V zařízení jde.
- Současně s tím jsou díky tomu V/V operace plně *platformově nezávislé*.

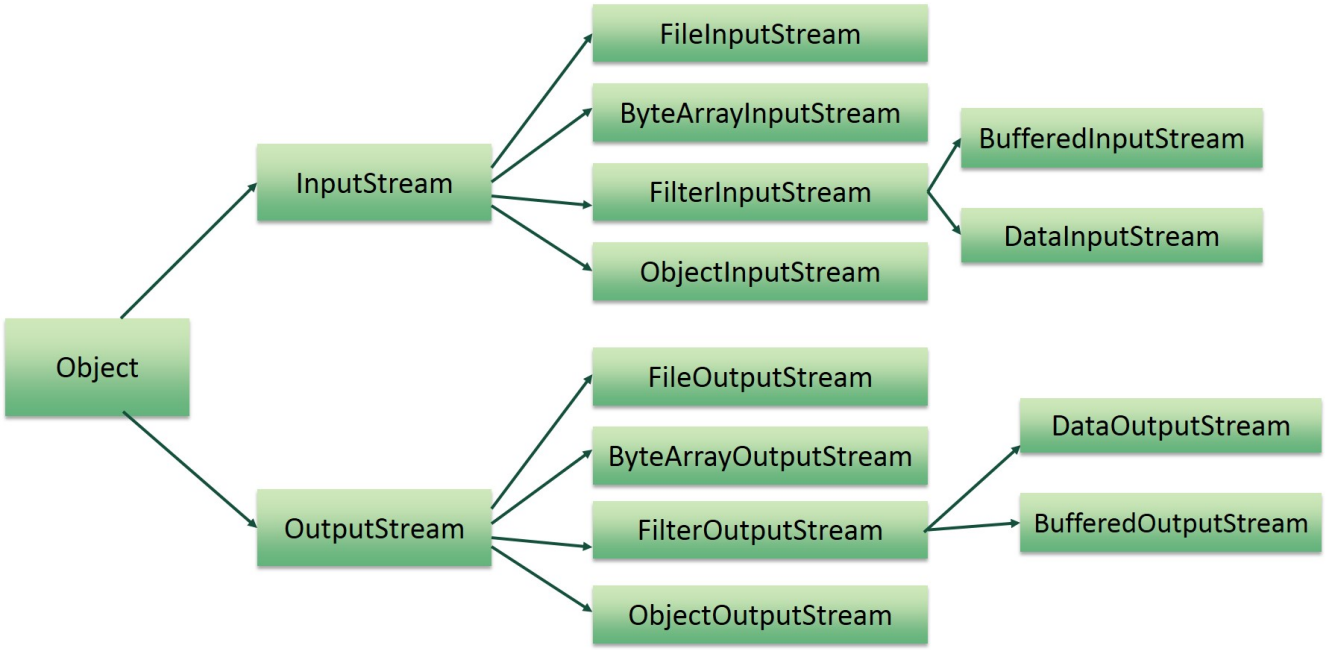
Table 1. Vstupně/výstupní proudy

typ dat	vstupní	výstupní
binární	*InputStream*	*OutputStream*

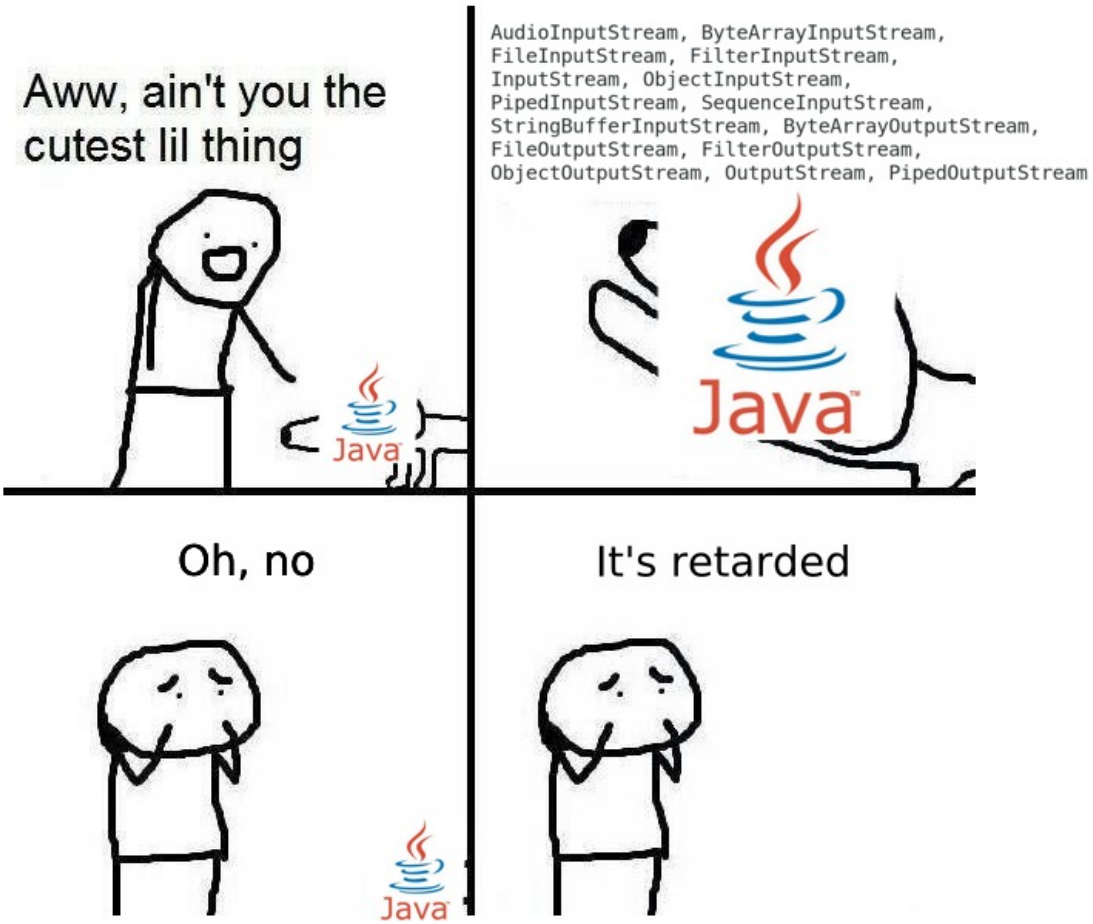
znakové	*Reader*	*Writer*

Vstupy a výstupy v Javě

Zdroj: http://www.tutorialspoint.com/java/



Je toho příliš mnoho?



API proudů

- Téměř vše ze vstupních/výstupních tříd a rozhraní je v balíku *java.io*.
- Počínaje Java 1.4 se rozvíjí alternativní balík *java.nio (New I/O)*, zde se ale budeme věnovat klasickým I/O z balíku *java.io*.
- Blíže viz dokumentace API balíků java.io, java.nio.

Skládání vstupně/výstupních proudů

Proudy jsou koncipovány jako "stavebnice" — lze je spojovat za sebe a tím přidávat vlastnosti:

```
// casual input stream
InputStream is = System.in;
// bis enhances stream with buffering option
BufferedInputStream bis = new BufferedInputStream(is);
```

Neplést si **streamy** (proudy dat) s **lambda streamy**!

Návrhový vzor *Decorator*

- Jedná se o použití návrhové vzoru Decorator:
  - "Nízkoúrovňové" třídy (např. `FileReader`) odpovídají třídě `ConcreteComponent` ze vzoru a poskytují základní funkcionalitu.
  - "Obalující" třídy, jejichž konstruktor vyžaduje již existující proud (např. `BuferedReader`) jsou dekorátory, které se "předsadí" před původní objekt a poskytují dodatečné metody. Na pozadí přitom komunikují s původním objektem.
  - Klientský kód může komunikovat jak s dekorátorem, tak přímo s původním objektem.

Stručné shrnutí

Closable	bytes closable	characters closable	lines closable
Input	InputStream	InputStreamReader	BufferedReader
Output	OutputStream	OutputStreamWriter	BufferedWriter

Základem znakových vstupních proudů je abstraktní třída *Reader* (pro výstupní *Writer*).

Konverze binárního proudu na znakový

- Ze vstupního binárního proudu *InputStream* (čili každého) je možné vytvořit znakový *Reader*.
- Ale pozor. Jedná se dvě různé hierarchie. Nelze tedy například vytvořit binární proud a konvertovat ho na buffered reader:

```
FileInputStream is = new FileInputStream("file.txt");
BufferedReader reader = new BufferedReader(is); // Syntax error - incompatible type of
is
```

Konverze binárního proudu na znakový (pokr.)

Musí se použít k tomu určená třída `InputStreamReader` (obdobně pro výstupní proudy `OutputStreamWriter`).

```
// binary input stream
InputStream is = ...
// character stream, decoding uses standard charset
Reader reader = new InputStreamReader(is);
// charsets are defined in java.nio package
Charset charset = java.nio.Charset.forName("ISO-8859-2");
// character stream, decoding uses ISO-8859-2 charset
```

```
Reader reader2 = new InputStreamReader(is, charset);
```

Na zjištění, jestli je možné ze čtenáře číst, se používa metoda `reader.ready()`.

Podporované názvy znakových sad naleznete na webu IANA Charsets.

Návrhový vzor *Bridge*

- `InputStreamReader` (i `OutputStreamWriter`) implementuje návrhový vzor Bridge:
- Hierarchie tříd `InputStream` představuje *implementaci* čtení binárních dat ze vstupních proudů.
- `InputStreamReader` pak převádí tato binární data do *abstrakce* textových dat.

Konverze znakového proudu na "buffered"

```
InputStreamReader isr = new InputStreamReader(is);
// takes another Reader and makes it bufferable
BufferedReader br = new BufferedReader(isr);
// BufferedReader supports read by line
String firstLine = br.readLine();
String secondLine = br.readLine();
```

Znakové výstupní proudy

- Jedná se o protějšky k vstupním proudům, názvy jsou konstruovány analogicky (např. *FileReader* → *FileWriter*).
- Místo generických metod *read* mají *write(...)*.

```
OutputStream os = System.out;
os.write("Hello World!");
// we have to use generic newline separator
os.write(System.lineSeparator());

// bw has special method for that
BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(os));
bw.newline();
```

Zavírání proudů a souborů

- **Soubory** zavíráme vždy.
- **Proudy** nezavíráme.
- Když zavřeme `System.out`, metoda `println` pak přestane vypisovat text.

Povinné zavírání proudů

- Při otevření souboru (a konverzi na proud) se musíme postarat o dodatečné uzavření souboru.
- Před *Java 7* se to muselo dělat blokem *finally*:

```
public String readFirstLine(String path) throws IOException {
  BufferedReader br = new BufferedReader(new FileReader(path));
  try {
     return br.readLine();
  } finally {
     if (br != null) br.close();
  }
}
```

Nově sa dá použít tzv. ***try-with-resources***:

```
public String readFirstLine(String path) throws IOException {
  try (BufferedReader br = new BufferedReader(new FileReader(path))) {
    return br.readLine();
  }
}
```

Více proudů

Pomocí *try-with-resources* lze ošetřit i více proudů současně — zavřou se pak všechny.

```
try (
     ZipFile zf = new ZipFile(zipFileName);
     BufferedWriter writer = new BufferedWriter(outputFilePath, charset)
    ) {
    ...
}
```

Obecně lze do hlavičky *try-with-resources* dát nejen proud, ale cokoli, co implementuje *java.io.Closeable*.

Replit.com demo k vstupům a výstupům

- https://Replit.com/@tpitner/PB162-Java-Lecture-11-input-and-output

Výpis textu *PrintStream* a *PrintWriter*

PrintStream

je typ proudu standardního výstupu *System.out* (a chybového *System.err*).

- Vytváří se z binárního proudu, lze jím přenášet i binární data.
- Většina operací nevyhazuje výjimky, čímž uspoří neustálé hlídání (try-catch).
- Na chybu se lze zeptat pomocí *checkError()*.

PrintWriter

pro znaková data

- Vytváří se ze znakového proudu, lze specifikovat kódování.

Příklad s nastavením kódování:

```
PrintWriter writer = new PrintWriter(new OutputStreamWriter(output, "UTF-8"));
```

Načítání vstupů (například z klávesnice)

- Třída `java.io.Scanner` - pro čtení z obecného proudu (ale i *stdin*)
- Nebo třída `java.io.Console` - přímo vždy z konzoly

Čtení z konzoly je typické pro aplikace spouštěné z příkazové řádky a není tudíž vždy možné - např. když spouštíme na serveru, v cloudu…

Replit.com demo ke třídě `Scanner`

- https://replit.com/@tpitner/PB162-Java-Lecture-11-scanner

Replit.com demo ke třídě `Console`

- https://replit.com/@tpitner/PV168-Java-Seminar-Console

Serializace objektů I

Postupy ukládání a rekonstrukce objektů:

serializace

postup, jak z objektu vytvořit sekvenci bajtů perzistentně uložitelnou na paměťové médium (disk) a později restaurovatelnou do podoby výchozího javového objektu.

deserializace

je právě zpětná rekonstrukce objektu

Aby objekt bylo možno serializovat, musí implementovat **prázdné** rozhraní *java.io.Serializable*.

Serializace objektů II

- Proměnné objektu, které nemají být serializovány, musí být označeny modifikátorem, klíčovým slovem, *transient*.
- Pokud požadujeme "speciální chování" při (de)serializaci, musí objekt definovat metody:

```
private void writeObject(ObjectOutputStream stream) throws IOException

private void readObject(ObjectInputStream stream) throws IOException,
ClassNotFoundException
```

ObjectInputStream je proud na čtení serializovaných objektů.

Návrhový vzor *Memento*

- (De)serializace souvisí návrhovým vzorem Memento:
  - Vzor umožňuje odložit si aktuální stav objektu a později ho obnovit.
  - (De)serializace v Javě tak může sloužit ke snadné implementaci tohoto vzoru.

Odkazy

Java Oracle Tutorial — essential Java I/O