
Kapitola 1. Přednáška 11 - Použití javadoc a jar. Vstupy a výstupy

Obsah

Dokumentace a distribuce aplikací	1
Dokumentace javových programů	2
Typy komentářů	2
Kde uvádíme dokumentační komentáře	3
Generování dokumentace	3
Značky javadoc	3
Příklad zdrojového textu se značkami javadoc	4
Spouštění javadoc	4
Příklady	5
Distribuce aplikací	5
Spuštění jar	5
Volby jar	6
jar - příklad	6
Rozšíření .jar archivů	8
Tvorba spustitelných archivů	8
Vytvoření spustitelného archivu - příklad	9
Spuštění archivu - příklad	9
Další příklad spuštění jar	9
Vstupy a výstupy v Javě	10
Koncepce vstupně/výstupních operací v Javě	10
Práce se soubory	10
Třída File	11
Třída File (2)	12
Třída File (3)	12
Práce s adresáři	14
Práce s binárními proudy	14
Vstupní binární proudy	14
Důležité neabstraktní třídy odvozené od InputStream	14
Další vstupní proudy	15
Práce se znakovými proudy	15
Výstupní proudy	16
Konverze: znakové <-> binární proudy	16
Serializace objektů	17
Odkazy	17

Dokumentace a distribuce aplikací

- Dokumentace javových programů, dokumentace API
- Typy komentářů - dokumentační komentáře
- Generování dokumentace
- Značky javadoc
- Distribuční archívy .jar
- Vytvoření archívu, metainformace
- Spustitelné archívy

Dokumentace javových programů

Základním a standardním prostředkem je tzv. *dokumentace API*

- Dokumentace je *naprosto nezbytnou součástí* javových programů.
- Rozlišujeme dokumentaci např. instalační, systémovou, uživatelskou, programátorskou...

Zde se budeme věnovat především dokumentaci programátorské, určené těm, kdo budou náš kód využívat ve svých programech, rozšiřovat jej, udržovat jej. Programátorské dokumentaci se říká *dokumentace API* (apidoc, apidocs).

Při jejím psaní dodržujeme tato pravidla:

- Dokumentujeme především *veřejné* (public) a *chráněné* (protected) prvky (metody, proměnné). Ostatní dle potřeby.
- Dokumentaci píšeme *přímo do zdrojového kódu* programu *ve speciálních dokumentačních komentářích* vpisovaných *před příslušné prvky* (metody, proměnné).
- Dovnitř metod píšeme jen *pomocné komentáře* pro programátory (nebo pro nás samotné).

Typy komentářů

Podobně jako např. v C/C++:

řádkové od značky // do konce řádku, nepromítnou se do dokumentace API

blokové začínají /* pak je text komentáře, končí */ na libovolném počtu řádků

dokumentační od značky /** po značku */ může být opět na libovolném počtu řádků

Každý další řádek může začínat mezerami či *, hvězdička se v komentáři neprojeví.

Kde uvádíme dokumentační komentáře

Dokumentační komentáře uvádíme:

- Před *hlavičkou třídy* - pak komentuje třídu jako celek.
- Před *hlavičkou metody nebo proměnné* - pak komentuje příslušnou metodu nebo proměnnou.
- Celý balík (package) je možné komentovat *speciálním samostatným HTML souborem* package-summary.html uloženým v adresáři balíku.

Generování dokumentace

Dokumentace má standardně podobu HTML stránek (s rámy i bez)

Dokumentace je generována nástrojem `javadoc` z

1. dokumentačních komentářů
2. i ze samotného zdrojového textu

Lze tedy (základním způsobem) dokumentovat i program bez vložených komentářů!

Chování `javadoc` můžeme změnit

1. volbami (options) při spuštění,
2. použitím jiného tzv. `docletu`, což je třída implementující potřebné metody pro generování komentářů.

Princip generování ze zdrojových textů pomocí speciálních `docletů` se dnes používá i po jiné než dokumentační účely - např. pro generátory zdrojových kódu aplikací EJB apod.

Značky javadoc

`javadoc` můžeme podrobněji instruovat pomocí značek vkládaných do dokumentačních komentářů, např.:

@author	specifikuje autora API/programu
@version	označuje verzi API, např. "1.0"
@deprecated	informuje, že prvek je zavrhováný

@exception	popisuje informace o výjimce, kterou metoda propouští ("vyhazuje")
@param	popisuje jeden parametr metody
@since	uvedeme, od kdy (od které verze pg.) je věc podporována/přítomna
@see	uvedeme odkaz, kam je také doporučeno nahlédnout (související věci)

Příklad zdrojového textu se značkami javadoc

Zdrojový text třídy Window:

```
/**
 * Klasse, die ein Fenster auf dem Bildschirm repräsentiert
 * Konstruktor zum Beispiel:
 * <pre>
 * Window win = new Window(parent);
 * win.show();
 * </pre>
 *
 * @see awt.BaseWindow
 * @see awt.Button
 * @version 1.2 31 Jan 1995
 * @author Bozo the Clown
 */
class Window extends BaseWindow {
    ...
}
```

Příklad dokumentačního komentáře k proměnné:

```
/**
 * enthält die aktuelle Anzahl der Elemente.
 * muss positiv und kleiner oder gleich der Kapazität sein
 */
protected int count;
```

Tyto a další příklady a odkazy lze vidět v původním materiálu JavaStyleGuide des IGE [<http://www.iam.unibe.ch/~scg/Resources/PSE/2001/WWW/projektHandbuch/codeInspections/JavaStyleGuide.html>], odkud byly ukázky převzaty.

Spouštění javadoc

- javadoc [options] [packagenames] [sourcefiles] [classnames]
[@files]

- možné volby:
 - `-help, -verbose`
 - `-public, -protected, -package, -private` - specifikuje, které prvky mají být v dokumentaci zahrnuty (implicitně: `-protected`)
 - `-d destinationdirectory` - kam se má dok. uložit
 - `-doctitle title` - titul celé dokumentace

Příklady

Zdroják s dokumentačními komentáři - Komentáře
[<http://www.fi.muni.cz/~tomp/java/ucebnice/javasrc/svet/Clovek.java>]

Ukázkové spuštění

```
javadoc
```

```
javadoc -classpath . -d apidocs svet
```

vytvoří dokumentaci tříd z balíku `svet` do adresáře `apidocs`

Distribuce aplikací

Distribucí nemyslíme použití nástroje typu "InstallShield"..., ale spíše něčeho podobného `tar/ZIPu`

- Java na sbalení množiny souborů zdrojových i přeložených (`.class`) nabízí nástroj `jar`.
- Sbalením vznikne soubor (archív) `.jar` formátově podobný `ZIPu` (obvykle *je to ZIP formát*), ale nemusí být komprimován.
- Kromě souborů obsahuje i metainformace (tzv. *MANIFEST*)
- Součástí archívu nejsou jen `.class` soubory, ale i další zdroje, např. *obrázky, soubory s národními variantami řetězců, zdrojové texty programu, dokumentace...*

Spuštění jar

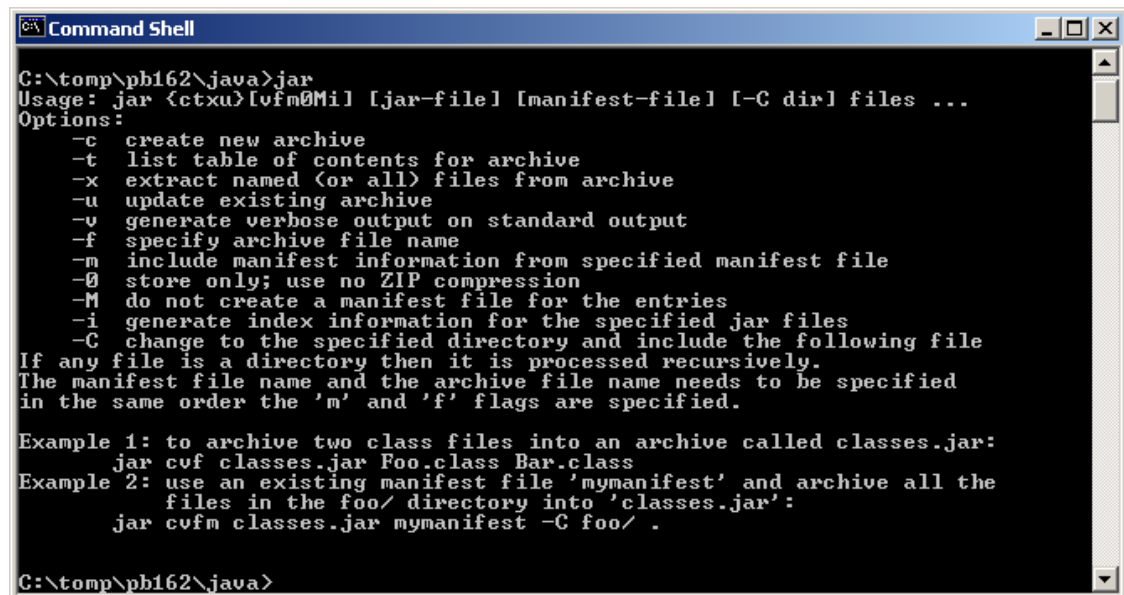
- `jar {ctxu} [vfmOM] [jar-file] [manifest-file] [-C dir] files`
- - `c` - vytvoří archív
 - `t` - vypíše obsah archívu

- x - extrahuje archiv
- u - aktualizuje obsah archívu
- volby:
 - v - verbose
 - 0 - soubory nekomprimuje
 - f - pracuje se se souborem, ne se "stdio"
 - m - přibálí metainformace z manifest-file
- parametr files uvádí, které soubory se sbalí - i nejavové (např. typicky dokumentace API - HTML, datové soubory)

Volby jar

Volby JAR lze vypsat i spuštěním jar bez parametrů:

Obrázek 1.1. Volby nástroje JAR



```
C:\tomp\pb162\java>jar
Usage: jar <ctxu>[vfm0Ml] [jar-file] [manifest-file] [-C dir] files ...
Options:
  -c    create new archive
  -t    list table of contents for archive
  -x    extract named (or all) files from archive
  -u    update existing archive
  -v    generate verbose output on standard output
  -f    specify archive file name
  -m    include manifest information from specified manifest file
  -0    store only; use no ZIP compression
  -M    do not create a manifest file for the entries
  -i    generate index information for the specified jar files
  -C    change to the specified directory and include the following file
If any file is a directory then it is processed recursively.
The manifest file name and the archive file name needs to be specified
in the same order the 'm' and 'f' flags are specified.

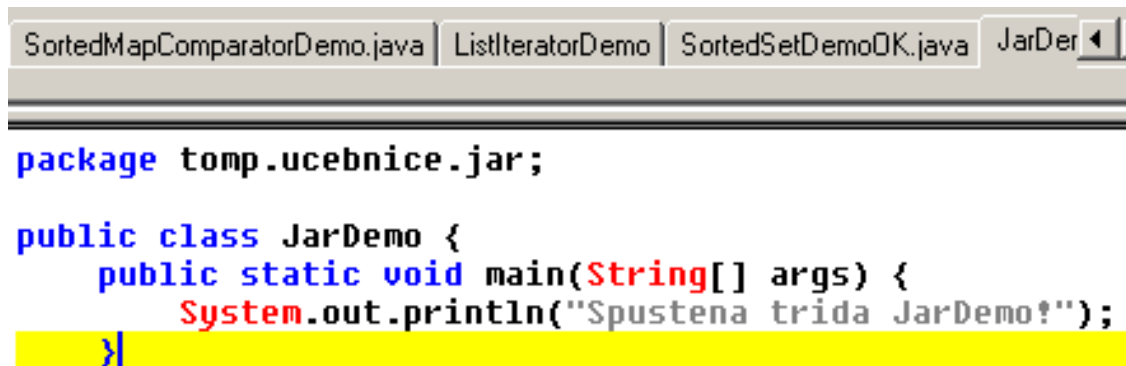
Example 1: to archive two class files into an archive called classes.jar:
jar cvf classes.jar Foo.class Bar.class
Example 2: use an existing manifest file 'mymanifest' and archive all the
files in the foo/ directory into 'classes.jar':
jar cvfm classes.jar mymanifest -C foo/ .

C:\tomp\pb162\java>
```

jar - příklad

Vezměme následující zdrojový text třídy JarDemo v balíku tomp.ucebnice.jar, tj. v adresáři
c:\tomp\pb162\java\tomp\ucebnice\jar:

Obrázek 1.2. Třída JarDemo

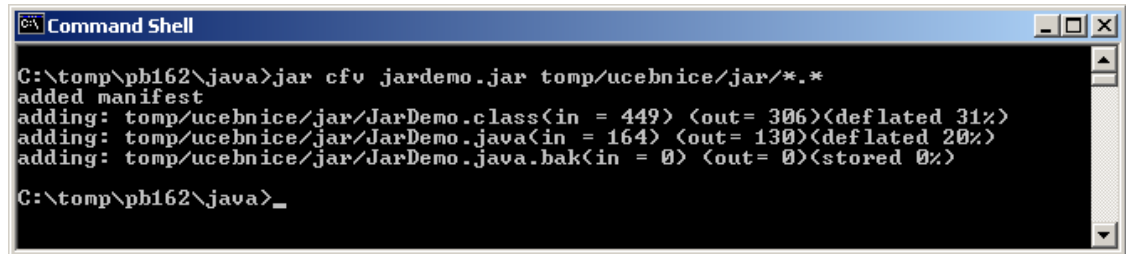


The screenshot shows a Java IDE with a tab bar at the top containing four files: SortedMapComparatorDemo.java, ListIteratorDemo, SortedSetDemoOK.java, and JarDer. The JarDer tab is selected. The code in the editor is as follows:

```
package tomp.ucebnice.jar;  
  
public class JarDemo {  
    public static void main(String[] args) {  
        System.out.println("Spustena trida JarDemo!");  
    }  
}
```

Vytvoříme archiv se všemi soubory z podadresáře tomp/ucebnice/jar (s volbou c - create, v - verbose, f - do souboru):

Obrázek 1.3. Vytvoření archivu se všemi soubory z podadresáře tomp/ucebnice/jar



The screenshot shows a Command Shell window with the following text:

```
C:\tomp\pb162\java>jar cfv jardemo.jar tomp/ucebnice/jar/*.  
added manifest  
adding: tomp/ucebnice/jar/JarDemo.class(in = 449) (out= 306)(deflated 31%)  
adding: tomp/ucebnice/jar/JarDemo.java(in = 164) (out= 130)(deflated 20%)  
adding: tomp/ucebnice/jar/JarDemo.java.bak(in = 0) (out= 0)(stored 0%)  
C:\tomp\pb162\java>_
```

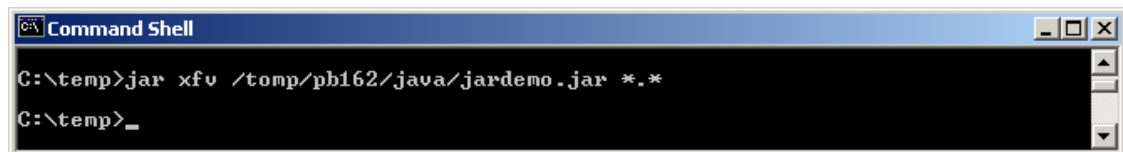
Vzniklý .jar soubor lze prohlédnout/rozbalit také běžným nástrojem typu unzip, gunzip, WinZip, PowerArchiver nebo souborovým managerem typu Servant Salamander...

Obrázek 1.4. .jar archiv v okně PowerArchiveru



Tento archiv rozbalíme v adresáři /temp následujícím způsobem:

Obrázek 1.5. Vybalení všech souborů z archívu



Rozšíření .jar archívů

Formáty vycházející z JAR:

- pro webové aplikace - .war
- pro enterprise (EJB) aplikace - .ear

liši se podrobnějším předepsáním adresářové struktury a dalšími povinnými metainformacemi

Tvorba spustitelných archívů

Vytvoříme jar s manifestem obsahujícím tento řádek:

```
Main-Class: NázevSpouštěnéTřídy
```

poté zadáme:

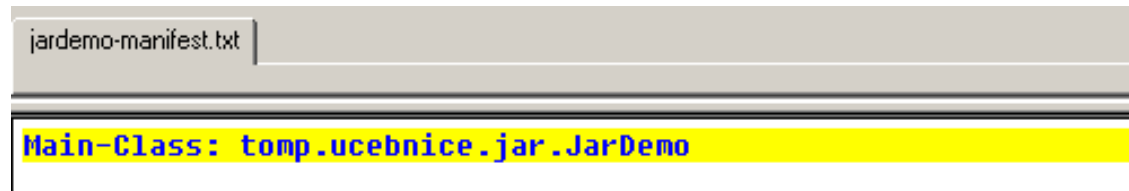
```
java -jar NázevBalíku.jar
```

a spustí se metoda `main` třídy `NázevSpouštěnéTřídy`.

Vytvoření spustitelného archívu - příklad

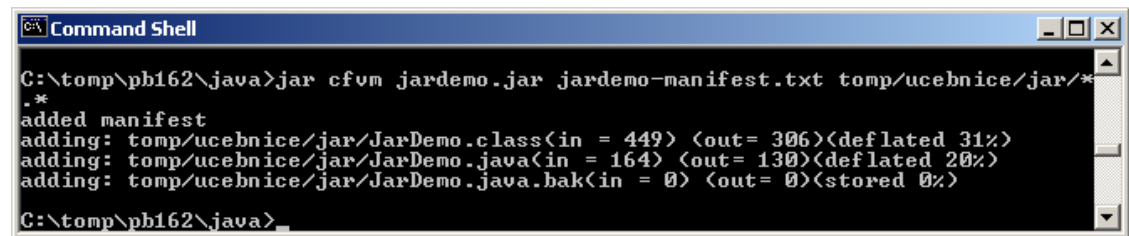
Nejprve vytvoříme soubor manifestu. Příklad jeho obsahu:

Obrázek 1.6. Soubor manifestu



Následně zabalíme archív s manifestem:

Obrázek 1.7. Zabalení archívu s manifestem



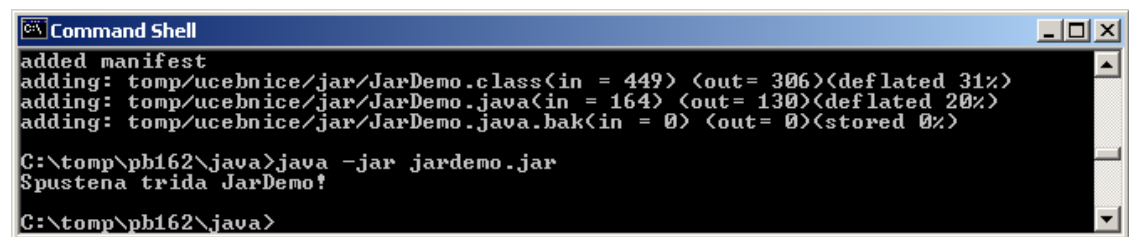
Spuštění archívu - příklad

Spuštění aplikace zabalené ve spustitelném archívu je snadné:

```
java -jar jardemo.jar
```

a spustí se metoda main třídy tomp.ucebnice.jar.JarDemo:

Obrázek 1.8. Spuštění aplikace z archívu



Další příklad spuštění jar

- `jar tfv svet.jar | more`
- vypíše po obrazovkách obsah (listing) archívu `svet.jar`

Vstupy a výstupy v Javě

- Koncepce I/O proudů v Javě, skládání (obalování vlastnostmi)
- Práce se soubory a adresáři, třída `File`
- Binární proudy, třídy `InputStream`, `OutputStream`
- Znakové proudy, třídy `Reader`, `Writer`
- Serializace objektů

Koncepce vstupně/výstupních operací v Javě

založeny na v/v proudech

plně **platformově nezávislé**

V/V proudy jsou

- **znakové** (`Reader/Writer`) nebo
- **binární** (`Stream`)

koncipovány jako "stavebnice" - lze vkládat do sebe a tím přidávat vlastnosti, např.

```
is = new InputStream(...);  
bis = new BufferedInputStream(is);
```

Téměř vše ze vstupních/výstupních tříd a rozhraní je v balíku `java.io`.

počínaje J2SDK1.4 se rozvíjí alternativní balík - `java.nio` (*New I/O*)

Blíže	viz	dokumentace	API	balíků	<code>java.io</code> <code>java.nio</code>
[http://java.sun.com/j2se/1.4.2/docs/api/java/io/package-summary.html], [http://java.sun.com/j2se/1.4.2/docs/api/java/nio/package-summary.html].					

Práce se soubory

vše je opět v balíku `java.io`

základem je třída `java.io.File` - nositel jména souboru, jakási "brána" k fyzickým souborům na disku.

používá se jak pro soubory, tak adresáře, linky i soubory identifikované UNC jmény (`\\počítač\adresář...`)

opět plně platformově nezávislé

na odstínění odlišnosti jednotlivých systémů souborů lze použít vlastností (uvádíme jejich hodnoty pro JVM pod systémem MS Windows):

- `File.separatorChar` \ - jako `char`
- `File.separator` \ - jako `String`
- `File.pathSeparatorChar` ; - jako `char`
- `File.pathSeparator` ; - jako `String`
- `System.getProperty("user.dir")` - adresář uživatele, pod jehož UID je proces JVM spuštěn

Třída `File`

Vytvoření konstruktorem - máme několik možností:

<code>new File(String filename)</code>	vytvoří v aktuálním adresáři soubor s názvem <i>filename</i>
<code>new File(File baseDir, String filename)</code>	vytvoří v adresáři <i>baseDir</i> soubor s názvem <i>filename</i>
<code>new File(String baseDirName, String filename)</code>	vytvoří v adresáři <i>se jménem baseDirName</i> soubor s názvem <i>filename</i>
<code>new File(URL url)</code>	vytvoří soubor se (souborovým - file:) URL <i>url</i>

Testy existence a povahy souboru:

<code>boolean exists()</code>	test na existenci souboru (nebo adresáře)
<code>boolean isFile()</code>	test, zda jde o soubor (tj. ne adresář)

() test, zda jde o adresář

Test práv ke čtení/zápisu:

boolean **canRead()** test, zda lze soubor číst
boolean **canWrite()** test, zda lze do souboru zapisovat

Třída **File** (2)

Vytvoření souboru nebo adresáře:

boolean **createNewFile()** (pro soubor) vrací `true`, když se podaří *soubor* vytvořit
boolean **mkdir()** (pro adresář) vrací `true`, když se podaří adresář vytvořit
boolean **mkdirs()** navíc si dotvoří i příp. neexistující adresáře na cestě

Vytvoření dočasného (temporary) souboru:

static File **createTempFile(String prefix, String suffix)** vytvoří dočasný soubor ve standardním pro to určeném adresáři (např. `c:\temp`) s uvedeným prefixem a sufixem názvu
static File **createTempFile(String prefix, String suffix, File directory)** dtto, ale vytvoří dočasný soubor v adr. `directory`
Zrušení:
boolean **delete()** zrušení souboru nebo adresáře

Přejmenování (ne přesun mezi adresáři!):

boolean **renameTo(File src, File dest)** přejmenuje soubor nebo adresář

Třída **File** (3)

Další vlastnosti:

long **length()** délka (velikost) souboru v bajtech

() čas poslední modifikace v ms od začátku éry - podobně jako systémový čas vrácený `System.currentTimeMillis()`.

String jen jméno souboru (tj. poslední část cesty)

(
String **getName()** celá cesta k souboru i se jménem

getPath() absolutní cesta k souboru i se jménem

String **getAbsolutePath()**

() adresář, v němž je soubor nebo adresář obsažen

Blíže viz dokumentace API třídy File [<http://java.sun.com/j2se/1.4.2/docs/api/java/io/File.html>].

Práce s adresáři

Klíčem je opět třída `File` - použitelná i pro adresáře

Jak např. získat (filtrovaný) seznam souborů v adresáři?

pomocí metody `File[] listFiles(FileFilter ff)` nebo podobně

`File[] listFiles(FileNameFilter fnf)`:

`FileFilter` je rozhraní s jedinou metodou `boolean accept(File pathname)`, obdobně `FileNameFilter`, viz Popis API `java.io.FileNameFilter` [<http://java.sun.com/j2se/1.4/docs/api/java/io/FileNameFilter.html>]

Práce s binárními proudy

Vstupní jsou odvozeny od abstraktní třídy `InputStream`

Výstupní jsou odvozeny od abstraktní třídy `OutputStream`

Vstupní binární proudy

Uvedené metody, kromě `abstract byte read()`, nemusejí být nutně v neabstraktní podtřídě překryty.

<code>void close()</code>	uzavře proud a uvolní příslušné zdroje (systémové "file handles" apod.)
<code>void mark(int readlimit)</code>	poznačí si aktuální pozici (později se lze vrátit zpět pomocí <code>reset()</code>)...
<code>boolean markSupported()</code>	...ale jen když platí tohle
<code>abstract int read()</code>	přečte bajt (0-255 pokud OK; jinak -1, když už není možné přečíst)
<code>int read(byte[] b)</code>	přečte pole bajtů
<code>int read(byte[] b, int off, int len)</code>	přečte pole bajtů se specifikací délky a pozice plnění pole b
<code>void reset()</code>	vrátí se ke značce nastavené metodou <code>mark(int)</code>
<code>long skip(long n)</code>	přeskočí zadaný počte bajtů

Důležité neabstraktní třídy odvozené od `InputStream`

`java.io.FilterInputStream` - je bázeová třída k odvozování všech vstupních proudů přidávajících vlastnost/schopnost filtrovat poskytnutý vstupní proud.

Příklady filtrů (ne všechny jsou v `java.io`):

BufferedInputStream	proud s vyrovnávací pamětí (je možno specifikovat její optimální velikost)
java.util.zip.CheckedInputStream	proud s kontrolním součtem (např. CRC32)
javax.crypto.CipherInputStream	proud dešifrující data ze vstupu
DataInputStream	má metody pro čtení hodnot primitivních typů, např. <code>readFloat()</code>
java.security.DigestInputStream	počítá současně i haš (digest) čtených dat, použitý algoritmus lze nastavit
java.util.zip.InflaterInputStream	dekomprimuje (např. GZIPem) zabalený vstupní proud (má ještě specializované podtřídy)
LineNumberInputStream	doplňuje informaci o tom, ze kterého řádku vstupu čteme (zavrhovaná - <i>deprecated</i> - třída)
ProgressMonitorInputStream	přidává schopnost informovat o průběhu čtení z proudu
PushbackInputStream	do proudu lze data vracet zpět

Další vstupní proudy

Příklad rekonstrukce objektů ze souborů

```
FileInputStream istream = new FileInputStream("t.tmp");
ObjectInputStream p = new ObjectInputStream(istream);
int i = p.readInt();
String today = (String)p.readObject();
Date date = (Date)p.readObject();
istream.close();
```

javax.sound.sampled.AudioInputStream	vstupní proud zvukových dat
ByteArrayInputStream	proud dat čtených z pole bajtů
PipedInputStream	roura napojená na "protilehlý" <code>PipedOutputStream</code>
SequenceInputStream	proud vzniklý spojením více podřízených proudů do jednoho virtuálního
ObjectInputStream	proud na čtení serializovaných objektů

Práce se znakovými proudy

základem je abstraktní třída `Reader`, konkrétními implementacemi jsou:

- `BufferedReader`, `CharArrayReader`, `InputStreamReader`, `PipedReader`, `StringReader`
- `LineNumberReader`, `FileReader`, `PushbackReader`

Výstupní proudy

nebudeme důkladně probírat všechny typy

principy:

- jedná se o protějšky k vstupním proudům, názvy jsou konstruovány analogicky (např. `FileReader` -> `FileWriter`)
- místo generických metod `read` mají `write(...)`

Příklady:

`PrintStream` poskytuje metody pro pohodlný zápis hodnot primitivních typů a řetězců - příkladem jsou `System.out` a `System.err`

`PrintWriter` poskytuje metody pro pohodlný zápis hodnot primitivních typů a řetězců

Konverze: znakové <-> binární proudy

Ze vstupního binárního proudu `InputStream` (čili každého) je možné vytvořit znakový `Reader` pomocí

```
// nejprve binární vstupní proud - toho kódování znaků nezajímá
InputStream is = ...
```

```
// znakový proud isr
// použije pro dekódování standardní znakovou sadu
Reader isr = new InputStreamReader(is);
```

```
// sady jsou definovány v balíku java.nio
Charset chrs = java.nio.Charset.forName("ISO-8859-2");
```

```
// znakový proud isr2
// použije pro dekódování jinou znakovou sadu
Reader isr2 = new InputStreamReader(is, chrs);
```

Podporované názvy znakových sad naleznete na webu IANA Charsets

[<http://www.iana.org/assignments/character-sets>].

Obdobně pro výstupní proudy - lze vytvořit Writer z OutputStream.

Serializace objektů

- nebudeme podrobně studovat, zatím stačí vědět, že:
 - **serializace objektů** je postup, jak z objektu vytvořit sekvenci bajtů persistentně 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`
- proměnné objektu, které nemají být serializovány, musí být označeny modifikátorem - klíčovým slovem - `transient`
- pokud požaduje "speciální chování" při de/serializaci, musí objekt definovat metody
 - `private void readObject(java.io.ObjectInputStream stream) throws IOException, ClassNotFoundException`
 - `private void writeObject(java.io.ObjectOutputStream stream) throws IOException`
- metody:
 - `DataOutputStream.writeObject(Object o)`

Odkazy

Tutoriály k Java I/O: kapitola z Sun Java Tutorial [<http://java.sun.com/docs/books/tutorial/essential/io/>]

Demo programy na serializaci (z učebnice): Serializace objektů
[<http://www.fi.muni.cz/~tomp/java/ucebnice/javasrc/tomp/ucebnice/serializace/>]