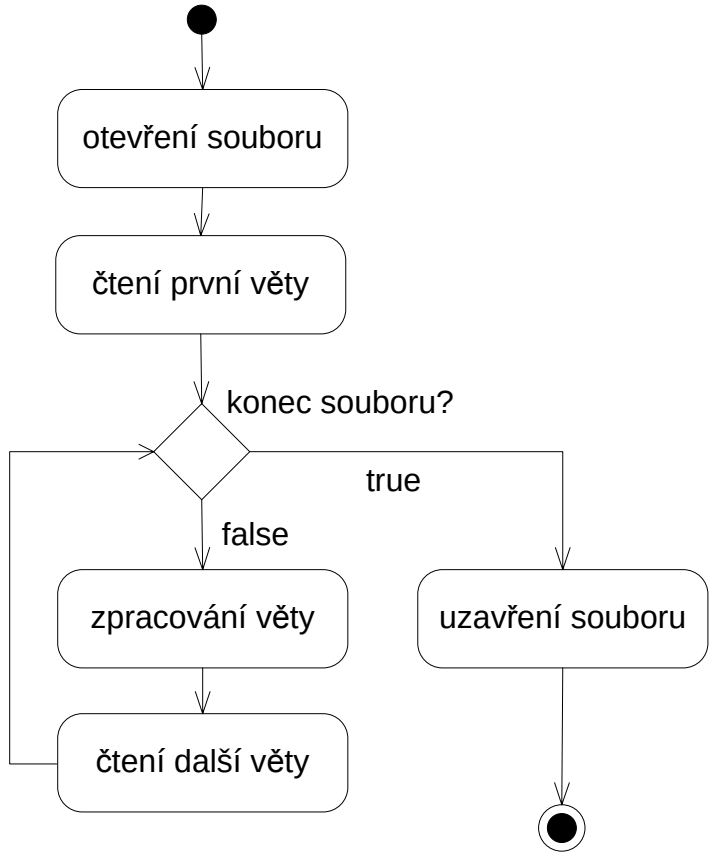
13.Vstupy a výstupy

13.1. Základní principy práce se soubory

Pro používání souborů v programu je potřeba zvládnout minimálně následující tři skupiny operací:

- čtení z textového souboru,
- zápis do textového souboru,
- operace na adresářové struktuře – nalezení souboru v adresáři, zjištění údajů o souboru, přejmenování souboru, výmaz souboru, založení adresáře, zrušení adresáře.

Průběh čtení z textového souboru má ve většině programovacích jazyků strukturu zobrazenou na obrázku13.1.



Obrázek 13.1 Průběh čtení z textového souboru

Tato struktura zajišťuje přečtení a zpracování všech vět v souboru i ošetření situace, kdy v souboru není ani jedna věta. Ostatní případné chyby (neexistence souboru, chyby na disku, atd.) je nutné ošetřovat jinak – většinou se používají výjimky, některé chybové stavy lze testovat předem (např. existenci souboru).

Pro zápis do textového souboru je struktura programu méně formalizovaná. Před psaním do souboru je potřeba soubor otevřít. Obvykle je možné upřesnit, zda se vytváří nový soubor, přepisuje stávající soubor či zda se bude zapisovat na konec existujícího souboru. V průběhu aplikace je poté možné zapisovat do souboru jednotlivé řádky. Nesmí se zapomenout na uzavření souboru – pokud se soubor explicitně neuzavře, obvykle chybí část textu ve vytvořeném souboru. Ošetřování chyb je podobné, jako při čtení souboru.

Operace nad adresářovou strukturou jsou v jednotlivých jazycích implementovány různě, v Javě je většina těchto operací v samostatné třídě `File`.

13.2. Vstupy a výstupy v Javě

Pro práci se vstupy a výstupy nám Java poskytuje celou řadu tříd a jejich metod. Základní třídy jsou uloženy v balíčku `java.io`, další lze nalézt jinde[29]. Koncepce vstupu a výstupu je založena na mechanizmu tzv. vstupních a výstupních proudů (**stream**) a jejich obalování dalšími třídami (filtry) pro přidání další funkčnosti[30].
Třídy pro práci se soubory lze rozdělit do následujících skupin:

	abstraktní třída (předek)	**třídy pracující s konkrétními typy úložišť dat**	**filtry**	**poznámky**
čtení po bytech	InputStream	FileInputStream, PipedInputStream, ByteArrayInputStream, …	BufferedInputStream, DataInputStream, ObjectInputStream, GZIPInputStream, DigestInputStream, CipherInputStream, AudioInputStream, …	filtr `InputStreamReader` převádí instanci potomka třídy `InputStream` na `Reader`
čtení po znacích	Reader	FileReader, PipedReader, ByteArrayReader, …	BufferedReader, LineNumberReader, …	
zápis po bytech	OutputStream	FileOutputStream, PipedOutputStream, ByteArrayOutputStream, …	PrintStream, BufferedOutputStream, ObjectOutputStream, DataOutputStream, GZIPOutputStream, DigestOutputStream, CipherOutputStream, …	filtr `OutputStreamWriter` převádí instanci potomka třídy `OutputStream` na `Writer`
zápis po znacích	Writer	FileWriter, PipedWriter, ByteArrayWriter, …	PrintWriter, BufferedWriter, ...	

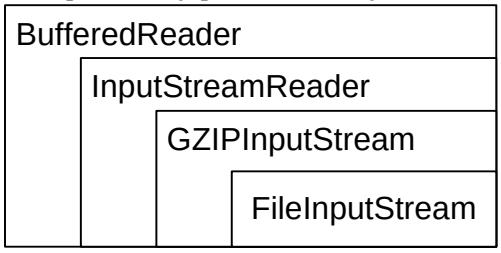
Tabulka 13.1 Rozdělení tříd pro práci s proudy do základních skupin

Rozlišení tříd v závislosti na tom, zda pracují s byty či se znaky vychází z významu textových souborů a z používání 16–bitového kódování znaků v Javě. Pokud se mají číst či zapisovat řetězce (instance třídy `String`), měli by se používat potomci tříd `Reader` či `Writer`. V každé skupině je abstraktní třída, která je předkem ostatních a která definuje základní operace dostupné ve všech potomcích. Vstup/výstup je vždy vázán na konkrétní úložiště, ze kterého se mají údaje číst či kam se mají zapisovat. Nejčastěji se používají soubory, lze používat i rouru (`Pipe`), vyhrazenou část paměti (`ByteArray`) i další. Pro čtení ze sítě (ukládání na síť) nejsou k dispozici veřejné třídy, ale např. třída `URL` poskytuje metody pro získání konkrétní instance pro čtení ze sítě (zápis na síť) – ukázka použití sítě jako úložiště je na straně 132.
Ve většině případů nám nepostačuje funkčnost základní třídy a chceme ji doplnit o další – buffrování proudů z důvodu výkonnosti, práci s celými řádky, podpora binárních dat, komprimace dat, šifrování dat, atd. Filtry jsou potomky příslušné abstraktní třídy, a tudíž se dají vzájemně zapouzdřovat do sebe v rámci příslušné skupiny.

---

[29] Od verze 1.4 nabízí Java další třídy pro práci se souboru v balíčku `java.nio` – cílem těchto tříd je vyšší výkonnost v oblastech síťové komunikace, použití regulárních výrazů při čtení ze souborů, využití bufferů, podpora znakových sad.
[30] Třídy pro vstup/výstup jsou typickou ukázkou použití návrhového vzoru decorator.

Zvláštní postavení mezi filtry mají třídy `InputStreamReader` a `OutputStreamReader`, které slouží pro převod ze čtení/zápisu po bytech do čtení/zápisu po znacích. Následující obrázek ukazuje zapouzdření tříd v případě, kdy se mají přečíst řádky z textového souboru na disku, který je zkomprimovaný pomocí metody GZIP:



Obrázek 13.2 Zapouzdření tříd při čtení komprimovaného souboru

Filtr pro dekomprimaci souboru je pouze pro čtení po bytech, z toho důvodu se použije pro otevření souboru třída `FileInputStream`, která se zabalí filtrem pro dekomprimaci (`GZIPInputStream`) a dále se převede na `Reader` pomocí třídy `InputStreamReader`. Výsledek se zapouzdří do třídy `BufferedReader`, která poskytuje metodu pro čtení po řádcích. Se třídami pro vstup/výstup jsou spojené kontrolované výjimky. Nejčastěji se odchytávají výjimky **FileNotFoundException** a obecná výjimka **IOException**.

13.3. Vstupní proudy

Pro vstupy slouží proudy založené na třídách **InputStream** a **Reader**. V následujících tabulkách je uveden přehled tříd z balíčku **java.io**, které se týkají čtení ze vstupních proudů:

třída	použití
FileInputStream	čtení ze souboru, parametrem konstruktoru je `String` se jménem souboru nebo objekt typu `File`
PipedInputStream	čtení z roury (do které zapisuje `PipedOutputStream`)
SequenceInputStream	vytvoří jeden vstupní proud ze dvou vstupních proudů, které jsou parametrem konstruktoru
ByteArrayInputStream	čtení z pole bytů v paměti, které je parametrem konstruktoru

Tabulka 13.2 Třídy z balíčku java.io pro vstup po bytech z jednotlivých uložišť

třída	použití
BufferedInputStream	vytváří buffer pro čtení, čímž je toto čtení efektivnější
DataInputStream	čte data z binárního souboru (který je ve formátu přenositelném mezi různými platformami), soubor lze vytvářet pomocí třídy `DataOutputStream`
PushbackInputStream	umožňuje vrátit část přečtených bytů zpět do vstupního proudu

Tabulka 13.3 Třídy (filtry) z balíčku java.io přidávající funkčnost pro čtení po bytech

Pro čtení po znacích je deklarována abstraktní třída `Reader` a její potomci. Měly by se používat vždy, kdy se čte text, neboť v této třídě je garantována správná obsluha znakových sad a převod textu do vnitřního kódování Javy (do znakové sady `Unicode`). V tabulce13.4. je uveden přehled tříd pro vytvoření konkrétních potomků třídy `Reader` a jejich srovnání s potomky třídy `InputStream`:

třída	**použití**	**odpovídající InputStream**
InputStreamReader	převádí `InputStream` na `Reader`	-
FileReader	čtení ze souboru, parametrem konstruktoru je `String` se jménem souboru nebo objekt typu `File`	FileInputStream
PipedReader	čtení z roury (z objektu, do kterého zapisuje `PipedWriter`)	PipedInputStream
CharArrayReader	čtení z pole znaků v paměti, které je parametrem konstruktoru	ByteArrayInputStream
StringReader	převede `String` na `Reader`	StringBufferInputStream

Tabulka 13.4 Třídy pro čtení po znacích z jednotlivých úložišť a jejich obdoba pro čtení po bytech

třída	**použití**	**odpovídající InputStream**
BufferedReader	vytváří buffer pro čtení, současně poskytuje metodu `readLine()` pro čtení po řádcích	BufferedInputStream
LineNumberReader	přidává metodu pro číslování čtených textových řádků	LineNumberInputStream
PushbackReader	umožňuje vrátit část přečtených znaků zpět do vstupního proudu	PushBackInputStream

Tabulka 13.5 Třídy (filtry) pro rozšíření funkčnosti potomků třídy Reader

13.3.1. Čtení z textového souboru

Pokud chceme číst po řádcích textový soubor uložený na disku, je nutné si vytvořit a zapouzdřit vhodný vstupní proud. Prvním krokem je vytvoření instance třídy `FileReader` – pro čtení souboru A.TXT by vytvoření instance vypadalo následovně:

```
FileReader vstupZn = new FileReader ("A.TXT");
```

Instance třídy `FileReader` podporuje čtení po znacích pomocí metody **read()**

```
int znak = vstupZn.read();
```

Ukázka čtení ze souboru po znacích[31] je uvedena dále v této kapitole na straně 133. Protože chceme číst po řádcích, je potřeba zabalit instanci třídy `FileReader` do filtru **BufferedReader**, který poskytuje metodu **readLine()** pro čtení jednotlivých řádků:

```
BufferedReader vstupRad = new BufferedReader (vstupZn);
String radek = vstupRad.readLine();   // čtení první řádky
```

Pokud metoda `readLine()` při čtení řádku zjistí, že je konec souboru, vrátí hodnotu `null`. Pokud není konec souboru, vrátí metoda `readLine()` instanci třídy `String` obsahující přečtený řádek. V našem případě se výsledek metody `readLine()` ukládá do proměnné `radek`. Tím máme

[31] Metoda `read()` vrací přečtený znak, při zjištění konce souboru vrátí hodnotu −1. To je důvod, proč metoda `read()` vrací hodnoty typu `int` a ne `char` – při přečtení znaku vrací kladné číslo či nulu, které lze převést na typ `char`, záporné hodnotě −1 žádný znak neodpovídá.

k dispozici základní kameny pro vytvoření cyklu *while* pro čtení řádek s testem na konec souboru. Pro uzavření souboru je nutné zavolat metodu **close()**[32].

```
while (radek != null) {
     … zpracování řádky …
     radek = vstupRad.readLine();
}
vstupRad.close();
```

Základní cyklus pro čtení souboru se v Javě občas zapisuje zkráceně (otvírání a zavírání souboru a obsluha výjimek zůstávají stejné). Je to ukázáno v následujícím příkladu, ve kterém se přečtený řádek vypíše na standardní výstup.

```
String radek;
while ((radek = vstup.readLine()) != null) {
     System.out.println (radek);
}
vstup.close();
```

Chyby při čtení souboru se odchytávají pomocí výjimek **FileNotFoundException** a **IOException**. Tyto výjimky je nutné odchytit, neboť patří mezi kontrolované výjimky (potomci třídy *Exception*, ale ne *RunTimeException* – viz kapitola věnovaná výjimkám). Výjimka *FileNotFoundException* upozorňuje na nejčastější chybu při čtení souboru – neexistenci vstupního souboru. Výjimka *FileNotFoundException* je potomkem třídy *IOException* a proto musí být uvedena před výjimkou *IOException* (při odchytávání výjimek se jde od konkrétních k obecným). Výjimka *IOException* vznikne při obecné chybě vstupu/výstupu. Lze ji použít i pro odchycení dalších chyb vstupu/výstupu, neboť je předkem ostatních výjimek vstupu/výstupu (ztrácejí se přitom ale informace týkající se konkrétního typu výjimky). Bez ošetření výjimky *IOException* není tento program přeložitelný.
Následující kód ukazuje, jak přečíst po řádcích textový soubor A.TXT a vypsat ho na konzolu.

```
     try {
          BufferedReader vstup = new BufferedReader
                                    (new FileReader ("A.TXT"));
          String radek;
          radek = vstup.readLine();
          while (radek != null) {
               System.out.println (radek);
               radek = vstup.readLine();
          }
          vstup.close();
     }
     catch (FileNotFoundException e) {
          System.out.println ("Soubor A.TXT neexistuje");
     }
     catch (IOException e){
          System.out.println ("Chyba na vstupu souboru A.TXT");
     }
```

Zkuste si na papíře simulovat průběh obou variant algoritmu pro čtení z textového souboru na souboru se třemi řádky a na prázdném souboru.

13.3.2. Čtení z konzole

Pro čtení z konzole lze použít systémovou proměnnou **System.in**, což je instance třídy *InputStream* a tudíž je možno číst pouze po bytech. Tento standardní vstup otevírá JVM vždy při

[32] Metoda *close()* je k dispozici i ve třídě *FileReader* při čtení po znacích.

své inicializaci (stejně jako proměnnou `System.out` pro standardní výstup). Pro přečtení řádky z konzole je tedy nutné tento proud obalit potřebnými filtry. Nejprve se převede vstup ze čtení po bytech na čtení po znacích zabalením do instance třídy **InputStreamReader**:

```
InputStreamReader ctiZnak = new InputStreamReader(System.in);
```

Pro čtení po řádcích tento vstup zabalíme ještě do instance třídy **BufferedReader** jako při čtení ze souboru. Standardní vstup se zavírá automaticky při skončení programu, není tedy nutné použít metodu `close()`. Stejně jako při čtení ze souboru musíme ošetřit výjimky **IOException**. Následující příklad přečte jednu řádku z konzole:

```
    System.out.print("Zadej text: ");
    try {
         BufferedReader cti =new BufferedReader
                              (new InputStreamReader(System.in));
         String radek = cti.readLine();
         System.out.println (radek);
    }
    catch (IOException e) {
         System.out.println("chyba vstupu");
    }
```

13.4. Výstupní proudy

Obdobně jako u vstupu lze třídy pro výstup rozdělit do čtyř skupin: třídy pro vytvoření výstupního proudu pro zápis po bytech (**OutputStream**), třídy pro rozšíření funkčnosti výstupního proudu, třídy pro vytvoření výstupu po znacích (**Writer**) a třídy pro rozšíření funkčnosti při výstupu po znacích.

třída	**použití**
FileOutputStream	zápis do souboru, parametrem konstruktoru je `String` se jménem souboru nebo objekt typu `File`, při použití druhého parametru typu `boolean` lze přidávat na konec existujícího souboru,
PipedOutputStream	zápis do roury (ze kterého čte `PipedInputStream`)
ByteArrayOutputStream	zápis do pole bytů v paměti, které je parametrem konstruktoru

Tabulka 13.6 Třídy z balíčku java.io pro zápis do jednotlivých uložišť po bytech

třída	**použití**
BufferedOutputStream	vytváří buffer pro efektivnější zápis
DataOutputStream	do výstupního proudu zapisuje proměnné a objekty v binárním formátu přenositelném mezi platformami, vytvořené soubory lze číst přes `DataInputStream` či v jiných programovacích jazycích
PrintStream	vypisuje textovou reprezentaci proměnných a objektů pomocí metod `print()`, `println()` a `printf()`

Tabulka 13.7 Třídy (filtry) z balíčku java.io pro přidání funkčnosti při zápisu po bytech

třída	použití	odpovídající OutputStream
OutputStreamWriter	převádí `OutputStream` na `Writer`	-
FileWriter	zápis do souboru, parametrem konstruktoru je `String` se jménem souboru nebo objekt typu `File`	FileOutputStream
PipedWriter	zápis do roury (do objektu, ze kterého čte `PipedReader`)	PipedOutputStream
StringWriter	zápis do bufferu, který může být převeden do objektu `String` či `StringBuffer`	-
CharArrayWriter	zápis do pole znaků v paměti	ByteArrayOutputStream

Tabulka 13.8 Třídy z balíčku java.io pro zápis do jednotlivých uložišť po znacích

třída	použití	odpovídající OutputStream
BufferedWriter	vytváří buffer pro efektivnější zápis	BufferedOutputStream
PrintWriter	vypisuje textovou reprezentaci proměnných a objektů pomocí metod `print()`, `println()` a `printf()`	PrintStream

Tabulka 13.9 Třídy (filtry) z balíčků java.io pro rozšíření funkčnosti při zápisu po znacích

13.4.1. Zápis do textového souboru

Pro zápis do textového souboru je potřeba vytvořit instanci potomka třídy `Writer`, který umí zapisovat do souboru na disku – vytvořit instanci třídy **FileWriter**, kde parametrem konstruktoru bude `String` se jménem souboru[33]. Pokud soubor na disku neexistuje, bude vytvořen nový, pokud existuje, bude přepsán. Jestliže chceme do již existujícího souboru připisovat na konec, musíme použít konstruktor třídy `FileWriter` se dvěma parametry. Druhým parametrem je logická hodnota, která určuje, zda budeme zapisovat za konec souboru (`true`) nebo původní soubor přepisovat (`false`). Pro zápis po znacích do souboru A.TXT bude řádek kódu s vytvořením instance třídy `FileWriter` vypadat takto:

```
FileWriter vystupZn = new FileWriter ("A.TXT");
```

Takto připravený výstup nám umožní zápis po znacích. Výhodnější je však zapisovat po řádcích, proto použijeme ještě filtr **PrintWriter**, který podporuje převody proměnných a objektů do textového tvaru (např. číslo typu `int` převede do textové reprezentace) a který má metodu **println()** pro zápis celého řádku.

```
PrintWriter vystup = new PrintWriter (vystupZn);
```

Po skončení zápisu nesmíme zapomenout na metodu **close()** pro zavření souboru, jinak může dojít ke ztrátě části dat. Jako u každé práce s vstupy a výstupy je nutné ošetřit výjimky.
Následující příklad ukazuje zapsání deseti řádek do textového souboru na disk po řádcích (metodou `println()`).

[33] Parametrem konstruktoru třídy `FileWriter` může být též instance třídy `File` popisující soubor.

```
    try {
         PrintWriter vystup = new PrintWriter
                                  (new FileWriter("A.TXT"));
         for (int i=0; i<10 ; i++)
              vystup.println("řádek "+i);
         vystup.close();
    }
    catch (IOException e) {
         System.out.println("Chyba při zápisu");
    }
```

Výstup na konzolu (System.out) nemusíme nijak "zabalit", protože autoři Javy použili pro statickou proměnnou **System.out** typ *`PrintStream`*, tj. třídu která umí zapisovat celé řádky (tj. má k dispozici metodu *`println()`*).

13.5. Další třídy a metody pro práci s proudy

Základy mechanismu práce se vstupními a výstupními proudy jsou definovány v balíčku *`java.io`*. Existují další třídy a metody pro vytváření vstupních proudů (např. pro čtení ze sítě, pro čtení BLOB z databází) i další třídy pro přidání funkčnosti k proudům (např. komprimace či šifrování).

13.5.1. Čtení ze sítě

Nejdříve si ukážeme, jak číst soubor ze sítě. Základem je **třída URL**, v rámci které se zadává adresa souboru, ke kterému chceme přistupovat. Nejjednodušší je zadat textovou adresu jako parametr konstruktoru:

```
    URL mojeURL = new URL("http://www.vse.cz/index.html");
```

Pokud zadáme špatný parametr, vyvolá konstruktor kontrolovanou výjimku **MalformedURLException**. Instance třídy URL může vytvořit vstupní proud následujícím způsobem:

```
    InputStream is = mojeURL.openStream();
```

S takto vytvořeným proudem můžeme pracovat jako s kterýmkoliv jiným proudem. Následující příklad vypíše soubor na URL *`http://www.vse.cz/index.html`* na standardní výstup (porovnejte s prvním příkladem v této kapitole na straně 129):

```
    try {
         URL mojeURL = new URL("http://www.vse.cz/index.html”);
         InputStream is = mojeURL.openStream();
         BufferedReader vstup = new BufferedReader
                              (new InputStreamReader (is));
         String radek = vstup.readLine();
         while (radek != null) {
              System.out.println (radek);
              radek = vstup.readLine();
         }
         vstup.close();
    }
    catch (MalformedURLException e) {
         System.out.println ("Chybne URL: "+mojeURL);
    }
    catch (IOException e){
         System.out.println ("Chyba na vstupu");
         e.printStackTrace();
    }
```

13.5.2. Komprimace a dekomprimace souborů

Třídy a metody pro komprimaci a dekomprimaci souborů typu ZIP a GZIP jsou v balíčku **java.util.zip**. Následující příklad zkomprimuje vstupní soubor, jehož jméno je zadáno jako parametr na příkazové řádce do výstupního zkomprimovaného souboru *test.gz*. Všimněte si, jak je uděláno zapouzdření potřebných filtrů při vytváření výstupního souboru.

```
import java.io.*;
import java.util.zip.*;
public class GZIPCompress {
     public static void main (String [] args){
          if (args.length == 0) {
               System.out.println("Nutno zadat jmeno souboru");
               System.exit(1);
          }
          try {
               BufferedReader vstup =
                    new BufferedReader (
                         new FileReader(args[0]));
               BufferedOutputStream vystup =
                    new BufferedOutputStream (
                         new GZIPOutputStream (
                              new FileOutputStream("test.gz")));
               int znak;
               znak = vstup.read();
               while (znak != -1) {
                    vystup.write(znak);
                    znak = vstup.read();
               }
               vstup.close();
               vystup.close();
          }
          catch (FileNotFoundException e) {
               System.out.println("Soubor "+args[0]+" nelze otevrit");
          }
          catch (IOException e){
               System.out.println ("Chyba na vstupu/vystupu");
               e.printStackTrace();
          }
     }
}
```

13.6. Třída File

Třída **File** slouží pro manipulaci se soubory a adresáři. Instance třídy *File* může odkazovat na adresář nebo soubor. Při vytváření instance není nutné, aby soubor (případně adresář) fyzicky existoval na disku – třída poskytuje metody pro vytváření souborů a adresářů, pro testování existence. Pro oddělování adresářů v popisu cesty používáme lomítko /, ve Windows lze použít i zpětné lomítko (ve zdrojovém textu programu se musí uvést dvě zpětná lomítka, neboť zpětné lomítko má i speciální význam v řetězcích). Lze použít i statickou proměnnou *File.separator*, která obsahuje oddělovač v závislosti na operačním systému. Instanci třídy *File* je možné vytvořit pomocí tří různých konstruktorů:

- *File(String jmeno);*
- *File(String cesta, String jmeno);*
- *File(File adresar, String jmeno);*

příklad použití	**význam**
File mujSoubor = new File ("a.txt")	instance *mujSoubor* nastavena na soubor *a.txt* v aktuálním adresáři
File mojeDopisy = new File("C:"+File.separator+ "dopisy")	instance *mojeDopisy* nastavena na adresář *dopisy* na disku C (popř. na soubor, neboť z kódu není poznat, zda se jedná o adresář či o soubor)
File mujDopis = new File("C:/dopisy/dopis1.txt")	instance *mujDopis* nastavena na soubor *dopis1.txt* v adresáři *dopisy* na disku C
File mujDopis = new File("C:\\dopisy","dopis1.txt")	instance *mujDopis* nastavena na soubor *dopis1.txt* v adresáři *dopisy* na disku C
File dopis = new File(mojeDopisy,"dopis1.txt")	instance *mujDopis* nastavena na soubor *dopis1.txt* v adresáři *dopisy* na disku C, použili jsme instanci *mojeDopisy* vytvořenou dříve

Tabulka 13.10 Použití konstruktorů třídy File

Pokud chceme ověřit fyzickou existenci souboru nebo adresáře, použijeme u instance třídy *File* metodu **exists()**. Třída dále obsahuje metody **isFile()** a **isDirectory()**, které zjistí, zda daná instance třídy *File* je soubor či adresář (musí fyzicky existovat na disku, není nutné předtím volat test *exists()*). Všechny tyto tři metody vracejí hodnotu typu boolean.
Pro vytvoření adresáře slouží metoda **mkdir()**, pro vytvoření souboru metoda **createNewFile().** Zjistit velikost existujícího souboru nebo adresáře můžeme pomocí metody **length()**. Soubor nebo adresář je možno smazat metodou **delete()** nebo přejmenovat pomocí metody **renameTo()**. Pro výpis adresáře slouží metoda **list()**, která vrací pole řetězců se jmény souborů a adresářů.
Následující program vypíše obsah adresáře *dokumenty* na disku C.

```
    File adresar = new File ("C:/dokumenty");
    String [] obsah = adresar.list();
    for (int i = 0;i < obsah.length;i++)
         System.out.println(obsah[i]);
    }
```

Nyní program upravíme tak, aby vypsal pouze adresáře obsažené v adresáři *dokumenty*.

```
    File adresar = new File ("C:/dokumenty");
    String [] obsah = adresar.list();
    for (int i = 0;i < obsah.length;i++){
         File prvek = new File (adresar,obsah[i]);
         if (prvek.isDirectory()) {
              System.out.println(obsah[i]);
    }}
```

Pokud chceme použít pro výpis **masku** (tj. vypsat pouze některé soubory či adresáře na základě nějaké podmínky), lze v metodě *list()* uvést jako parametr instanci třídy, která implementuje rozhraní **FilenameFilter.** Pro využití této možnosti musíme napsat třídu, která bude implementovat toto rozhraní, a bude obsahovat metodu **boolean accept (File dir, String name)**. Tato metoda se poté bude volat pro každý nalezený soubor v adresáři a měla by vracet hodnotu *true* pro každý soubor, který se má vypisovat.