

Výjimky a soubory

Jiří Vokřínek

Katedra počítačů

Fakulta elektrotechnická

České vysoké učení technické v Praze

Přednáška 4

B0B36PJV – Programování v JAVA

Část 1 – Výjimky

Výjimky

Část 2 – Soubory

Soubory

Práce se soubory

Čtení a zápis souboru v Javě

Binární soubory

Textové soubory

Část I

Výjimky

Výjimky

Výjimky (Exceptions)

- Představují mechanismus ošetření chybových (výjimečných) stavů
- Mechanismus výjimek umožňuje metodu rozdělit na hlavní (standardní) část a řešení nestandardní situace

Umožňuje zpřehlednit kód metod

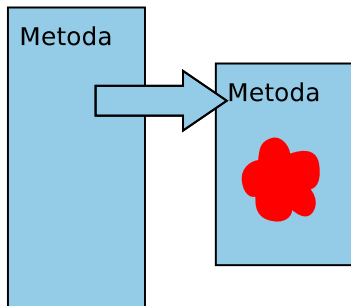
- Chyba nemusí znamenat ukončení programu
 - Chybu je možné ošetřit, zotavit běh programu a pokračovat ve vykonávání dalšího kódu

<http://docs.oracle.com/javase/tutorial/essential/exceptions/index.html>

Výjimka nikoliv vyjímka – výjimka označuje název děje nebo výsledku děje, je to podstatné jméno odvozené od slovesa.

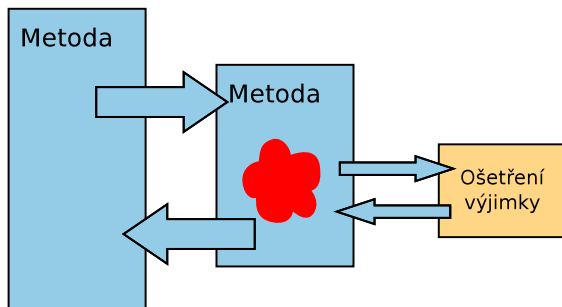
Nestandardní situace

- Vznik nestandardní situace může ukončit program



Princip ošetření výjimky

- Ošetřením výjimky program může pokračovat ve své „standardní“ činnosti



Výjimky (Exceptions)

- Mechanismus výjimek umožňuje přenést řízení z místa, kde výjimka vznikla do místa, kde bude zpracována
 - Oddělení *výkonné* části od části *chybu řešící*
- Posloupnost příkazů, ve které může vzniknout výjimka, uzavíráme do bloku klíčovým slovem **try**
- Příslušnou výjimku pak „zachytáváme“ prostřednictvím **catch**
- Metodu můžeme deklarovat jako metodu, která může vyvolat výjimku – klíčovým slovem **throws**
- Java ošetření některých výjimečných situací vynucuje
 - **Reakce na očekávané chyby se vynucuje** na úrovni překladač
- Při vzniku výjimky je automaticky vytvořen **objekt**, který nese informaci o vzniklé výjimce (**Throwable**)

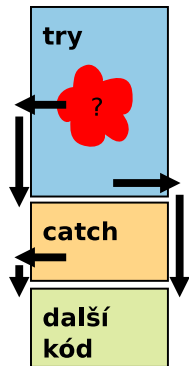
<http://docs.oracle.com/javase/8/docs/api/java/lang/Throwable.html>

Základní ošetření části kódu, kde může vzniknout výjimka

try – catch

- Volání příkazů/metod výkonné části dáváme do bloků příkazu **try**
- V případě vyvolání výjimky se řízení předá konstrukci ošetření výjimky **catch**
- Při předání vyvolání výjimky se ostatní příkazy v bloku **try** nevolají

```
try {  
    //prikazy kde muze vzniknout vyjimka  
} catch (Exception e) {  
    //osetreni vyjimky  
}  
// prikazy
```



Mechanismus šíření výjimek v Javě

Při vzniku výjimky hledá JVM odpovídající řešení, které je schopné výjimku ošetřit (převzít řízení):

- Pokud vzniká výjimka v bloku **try** hledá se odpovídající klauzule **catch** v tomto příkazu
- Pokud výjimka vznikne mimo příkaz **try**, předá se řízení do místa volání metody a pokračuje se podle předchozího bodu
- Pokud konstrukce pro ošetření výjimky v těle metody není, skončí funkce nestandardně a výjimka se šíří na dynamicky nadřazenou úroveň
- Není-li výjimka ošetřena ani ve funkci **main**, vypíše se a program skončí
- Pro rozlišení různých typů výjimek jsou v Javě zavedeny třídy. Výjimky jsou instancemi těchto tříd.

Základní dělení nestandardních situací (výjimek)

1. **RuntimeException** – situace, na které bychom měli reagovat, můžeme reagovat a dokážeme reagovat
 - Situace, kterým se můžeme vyvarovat programově např. kontrolou mezi pole nebo null hodnoty
 - Indexování mimo rozsah pole, dělení nulou, ...
ArrayIndexOutOfBoundsException, **ArithmeticException**, **NullPointerException**, ...

<http://docs.oracle.com/javase/8/docs/api/java/lang/RuntimeException.html>

2. **Exception** – situace, na které musíme reagovat
 - Java vynucuje ošetření nestandardní situace
 - Například **IOException**, **FileNotFoundException**
3. **Error** – situace, na které obecně reagovat nemůžeme – závažné chyby
 - Chyba v JVM, HW chyba: **VirtualMachineError**, **OutOfMemoryError**, **IOError**, **UnknownError**, ...

<http://docs.oracle.com/javase/8/docs/api/java/lang/Error.html>

Příklad – RuntimeException 1/3

Při spuštění sice získáme informaci o chybě, ale bez zdrojového kódu nevíme přesně co a proč program předčasně končilo

- `java DemoException` → **NullPointerException**
- `java DemoException 1` → **ArrayIndexOutOfBoundsException**
- `java DemoException 1 a` → **NumberFormatException**
- `java DemoException 1 1` – program vypíše hodnotu 1

```
public class DemoException {  
    public int parse(String[] args) {  
        return Integer.parseInt(args[1]);  
    }  
  
    public static void main(String[] args) {  
        DemoException demo = new DemoException();  
        int value = demo.parse(args.length == 0 ? null : args);  
        System.out.println("2nd argument: " + value);  
    } }  
}
```

lec04/DemoException

Příklad – RuntimeException 2/3

■ Explicitní kontrola parametru

```
public class DemoExceptionTest {
    public int parse(String[] args) {
        int ret = -1;
        if (args != null && args.length > 1) {
            ret = Integer.parseInt(args[1]);
        } else {
            throw new RuntimeException("Input argument not set");
        }
        return ret;
    }

    public static void main(String[] args) {
        DemoExceptionTest demo = new DemoExceptionTest();
        int value = demo.parse(args);
        System.out.println("2nd argument: " + value);
    } }
```

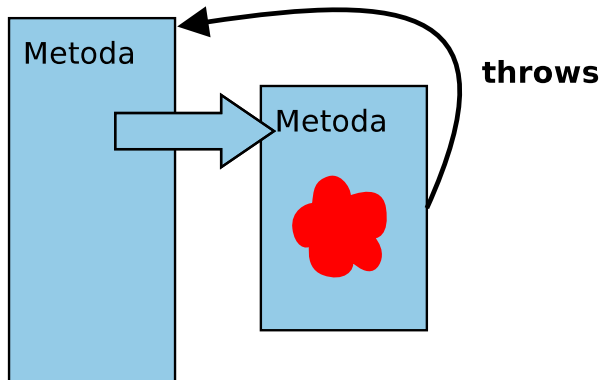
lec04/DemoExceptionTest

■ Neřeší však **NumberFormatException**

Předání ošetření výjimky (**Exception**) výš

- Ošetření výjimky lze předat nadřazené metodě deklarací **throws**

```
public void readData(void) throws IOException {  
    ...  
}
```



Příklad – RuntimeException 3/3

- Výjimku `NumberFormatException` odchytíme a „nahradíme“ upřesňující zprávou
- Výjimku propagujeme výše prostřednictvím `throw`

```
public int parse(String[] args) {  
    try {  
        if (args != null && args.length > 1) {  
            return Integer.parseInt(args[1]);  
        } else {  
            throw new RuntimeException("Input argument not set");  
        }  
    } catch (NumberFormatException e) {  
        throw new NumberFormatException("2nd argument must be  
int");  
    }  
}
```

`lec04/DemoExceptionTestThrows`

Způsoby ošetření

- Zachytíme a kompletně ošetříme
- Zachytíme, částečně ošetříme a dále předáme výše

Např. Interně v rámci knihovny logujeme výjimku

- Ošetření předáme výše, výjimku nelze nebo ji nechceme ošetřit
- *Bez ošetření výjimky* – špatně
 - Aspoň výpis na standardní chybový výstup

```
} catch (Exception e) {  
    e.printStackTrace();  
}
```

- Případně logovat (např. do souboru) v případě grafické aplikace nebo uživatelského prostředí

system logger, log4j, ...

Příklad explicitní deklarace propagace výjimky - 1/2

- Hodnota 2. argumentu je pro nás klíčová, proto použijeme výjimku **Exception**, která vyžaduje ošetření
- Výjimku předáváme výš deklarací **throws**

```
public int parse(String[] args) throws Exception {
    try {
        if (args != null && args.length > 1) {
            return Integer.parseInt(args[1]);
        } else {
            throw new Exception("Input argument not set");
        }
    } catch (NumberFormatException e) {
        throw new Exception("2nd input argument must be
integer");
    }
}
```

Příklad explicitní deklarace propagace výjimky - 2/2

- Kompilace třídy však selže, neboť je nutné výjimku explicitně ošetřit

```
DemoExceptionTestThrow.java:18: error: unreported
exception Exception; must be caught or declared to be
    thrown
        int value = demo.parse(args)
```

- Proto musí být volání v bloku `try`

```
try {
    int value = demo.parse(args);
    System.out.println("2nd argument: " + value);
} catch (Exception e) {
    System.out.println("Error: " + e.getMessage());
}
    lec04/DemoExceptionTestThrow
```

- Nebo `main` musí deklarovat propagaci výjimky výš

```
public static void main(String[] args) throws Exception {
    lec04/DemoExceptionTestThrowMain
```

V tomto případě je použití výjimky `Exception` nevhodné.

Příklad explicitní deklarace propagace výjimky - 2/2

- Kompilace třídy však selže, neboť je nutné výjimku explicitně ošetřit

```
DemoExceptionTestThrow.java:18: error: unreported
    exception Exception; must be caught or declared to be
        thrown
        int value = demo.parse(args)
```

- Proto musí být volání v bloku **try**

```
try {
    int value = demo.parse(args);
    System.out.println("2nd argument: " + value);
} catch (Exception e) {
    System.out.println("Error: " + e.getMessage());
}
    lec04/DemoExceptionTestThrow
```

- Nebo **main** musí deklarovat propagaci výjimky výš

```
public static void main(String[] args) throws Exception {
    lec04/DemoExceptionTestThrowMain
```

*V tomto případě je použití výjimky **Exception** nevhodné.*

Příklad explicitní deklarace propagace výjimky - 2/2

- Kompilace třídy však selže, neboť je nutné výjimku explicitně ošetřit

```
DemoExceptionTestThrow.java:18: error: unreported
    exception Exception; must be caught or declared to be
        thrown
        int value = demo.parse(args)
```

- Proto musí být volání v bloku **try**

```
try {
    int value = demo.parse(args);
    System.out.println("2nd argument: " + value);
} catch (Exception e) {
    System.out.println("Error: " + e.getMessage());
}
lec04/DemoExceptionTestThrow
```

- Nebo **main** musí deklarovat propagaci výjimky výš

```
public static void main(String[] args) throws Exception {
lec04/DemoExceptionTestThrowMain
```

*V tomto případě je použití výjimky **Exception** nevhodné.*

Příklad explicitní deklarace propagace výjimky - 2/2

- Kompilace třídy však selže, neboť je nutné výjimku explicitně ošetřit

```
DemoExceptionTestThrow.java:18: error: unreported
    exception Exception; must be caught or declared to be
        thrown
        int value = demo.parse(args)
```

- Proto musí být volání v bloku **try**

```
try {
    int value = demo.parse(args);
    System.out.println("2nd argument: " + value);
} catch (Exception e) {
    System.out.println("Error: " + e.getMessage());
}
    lec04/DemoExceptionTestThrow
```

- Nebo **main** musí deklarovat propagaci výjimky výš

```
public static void main(String[] args) throws Exception {
    lec04/DemoExceptionTestThrowMain
```

*V tomto případě je použití výjimky **Exception** nevhodné.*

Kdy předávat výjimku výš?

- Pokud je to možné, výjimečnou situaci řešíme co nejbližší místa jejího vzniku
- Výjimkám typu **RuntimeException** můžeme předcházet
NullPointerException, ArrayIndexOutOfBoundsException typicky indikují opomenutí.
- Předávání výjimek **throws** se snažíme vyhnout
Zejména na „uživatelskou“ úroveň.
- Výjimky typu **Exception** předáme výš pouze pokud nemá cenu výjimku ošetřovat, např. požadovanou hodnotu potřebujeme a bez ní nemá další činnost programu smysl
- Java při překladu kontroluje kritické části, které vyžadují ošetření nebo deklaraci předání výjimky výš

Kontrolované a nekontrolované výjimky

- **Kontrolované** výjimky musí být explicitně deklarovány v hlavičce metody
 - Jedná se o výjimky třídy **Exception**
 - Označující se také jako **synchronní výjimky**
- **Nekontrolované** výjimky se mohou šířit z většiny metod, a proto by jejich deklarování obtěžovalo
 - Jedná se o **asynchronní výjimky**
 - Rozlišujeme na výjimky, které
 - běžný uživatel není schopen ošetřit (**Error**)
 - chyby, které ošetřujeme podle potřeby; podtřídy třídy **RuntimeException**.

Třída **Error**

- Představuje závažné chyby na úrovni virtuálního stroje (JVM)
- Nejsme schopni je opravit
- Třída **Error** je nadtřída všech výjimek, které převážně vznikají v důsledku sw nebo hw chyb výpočetního systému, které většinou nelze v aplikaci smysluplně ošetřit

Třída `RuntimeException`

- Představuje třídu chyb, kterou lze úspěšně ošetřit
- Je třeba je očekávat—jsou to **asynchronní výjimky**
- Nemusíme na ně reagovat a můžeme je propagovat výše
 - Překladač ošetření této výjimky nevyžaduje
- Reagujeme na ně dle našeho odhadu jejich výskytu
 - Pokud špatně odhadneme a nastane chyba, JVM indikuje místo vzniku chyby a my můžeme ošetření výjimky, nebo ošetření vzniku výjimky implementovat

Zpravidla situace, která „nikdy nenastane“ se jednou stane. Otázkou tak spíše je, jak často to se to stane při běžném použití programu.
- Prakticky není možné (vhodné) ošetřit všechny výjimky **RuntimeException**, protože to zpravidla vede na nepřehledný kód

Vytvoření vlastní výjimky

- Pro rozlišení případných výjimečných stavů můžeme vytvořit své vlastní výjimky
- Buď odvozením od třídy **Exception** – kontrolované (synchronní) výjimky
- Nebo odvozením od třídy **RuntimeException** – asynchronní

Příklad vlastní výjimky – RuntimeException

- Vlastní výjimku **MyRunTimeException** vytvoříme odvozením od třídy **RuntimeException**
- Výjimku **MyRunTimeException** není nutné ošetřovat

```
class MyRuntimeException extends RuntimeException {  
    public MyRuntimeException(String str) {  
        super(str);  
    }  
}  
  
void demo1() {  
    throw new MyRuntimeException("Demo MyRuntimeException");  
}
```

lec04/MyExceptions

Vytvoření vlastní výjimky – Exception

- Vlastní výjimku **MyException** vytvoříme odvozením od třídy **Exception**
- Výjimku **MyException** je nutné ošetřovat, proto metodu **demo2** deklarujeme s **throws**

```
class MyException extends Exception {  
    public MyException(String str) {  
        super(str);  
    }  
}  
  
void demo2() throws MyException {  
    throw new MyException("Demo MyException");  
}
```

lec04/MyExceptions

Ošetřování různých výjimek

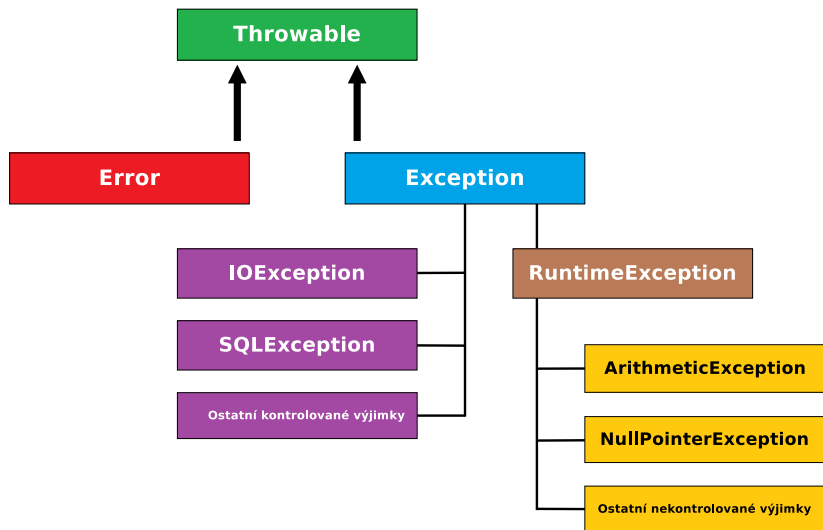
- Příslušná sekce **catch** ošetřuje kompatibilní výjimky
- Můžeme proto na různé chyby reagovat různě

```
public static void main(String[] args) {
    MyExceptions demo = new MyExceptions();
    try {
        if (args.length > 0) {
            demo.demo1();
        } else {
            demo.demo2();
        }
    } catch (MyRuntimeException e) {
        System.out.println("MyRuntimeException:" + e.
            getMessage());
    } catch (MyException e) {
        System.out.println("MyException:" + e.getMessage());
    }
}
```

lec04/MyExceptions

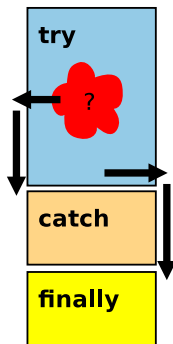
- Při ošetřování výjimek můžeme uplatnit dědické vztahy a hierarchii tříd výjimek

Struktura a hierarchie výjimek



Blok `finally`

- Při běhu programu může být nutné vykonat konkrétní akce bez ohledu na vyvolání výjimky
- Typickým příkladem je uvolnění alokovaných zdrojů, např. souborů
- Příkazy, které se mají vždy provést před opuštěním funkce je možné zapsat do bloku `finally`
- Příkazy v bloku `finally` se provedou i když blok příkazu v `try` obsahuje `return` a k vyvolání výjimečné situace nedojde



<http://docs.oracle.com/javase/tutorial/essential/exceptions/finally.html>

Příklad – try – catch – finally – 1/2

```
public class BlockFinally {  
    void causeRuntimeException() {  
        throw new RuntimeException("RuntimeException");  
    }  
    void causeException() throws MyException {  
        throw new MyException("Exception");  
    }  
    void start(int v) {  
        ...  
    }  
    public static void main(String[] args) {  
        BlockFinally demo = new BlockFinally();  
        demo.start(args.length > 0 ? Integer.parseInt(args  
[0]) : 1);  
    }  
}
```

lec04/BlockFinally

Příklad – try – catch – finally – 2/2

```
void start(int v) {  
    try {  
        if (v == 0) {  
            System.out.println("v:0 call runtime");  
            causeRuntimeException();  
        } else if (v == 1) {  
            System.out.println("v:1 call exception");  
            causeException();  
        } else if (v == 2) {  
            System.out.println("v:2 call return");  
            return;  
        }  
    } catch (MyException e) {  
        System.out.println("start handle Exception");  
    } finally {  
        System.out.println("Leave start!");  
    }  
}
```

- Vyzkoušejte pro různá volání: java BlockFinally 0; java BlockFinally 1; java BlockFinally 2

[lec04/BlockFinally](#)

Výjimky a uvolnění zdrojů – 1/2

- Kromě explicitního uvolnění zdrojů v sekci **finally** je možné využít také konstrukce **try-with-resources** příkazu **try**
- Při volání **finally**

```
void writeInt(String filename, int w) throws
    IOException {
    FileWriter fw = null;
    try {
        fw = new FileWriter(filename);
        fw.write(w);
    } finally {
        if (fw != null) {
            fw.close();
        }
    }
}
```

totiž může dojít k výjimce při zavírání souboru a tím potlačení výjimky vyvolané při čtení ze souboru.

Výjimky a uvolnění zdrojů 2/2

- Proto je výhodnější přímo využít konstrukce **try-with-resources** příkazu **try**

```
void writeInt(String filename, int w) throws
    IOException {
    try (FileWriter fw = new FileWriter(filename)) {
        fw.write(w);
    }
}
```

- **try-with-resources** lze použít pro libovolný objekt, který implementuje **java.lang.AutoCloseable**

<http://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html>

Část II

Soubory

Soubory a organizace dat v souborovém systému

- Soubor je množina údajů uložená ve vnější paměti počítače

Obvykle na pevném disku

- Typické operace pro soubor jsou:

1. Otevření souboru
2. Čtení dat
3. Zápis dat
4. Zavření souboru

- Přístup k datům (údajům) v souboru může být

- Sekvenční (postupný)

Postupné čtení nebo zápis dat do souboru

- Náhodný (adresovatelný)

Umožňuje adresovat libovolné místo v souboru podobně jako při přístupu do pole

- Způsob přístup k údajům v souboru není zakódován v souboru, ale je dán programem

Podobně také případ, zdali soubor „chápeme“ jako textový nebo binární.

Adresa (cesta) k souboru

- Soubory jsou uloženy v souborovém systému
- Soubory organizujeme do složek (adresářů), které tvoří hierarchii adresářů a souborů tvořící stromovou strukturu

Lze sice vytvořit i cykly, zpravidla je to však speciální případ.

- Souborový systém představuje „adresovatelný“ prostor, kde
- ke každému souboru existuje „adresa“ identifikující v jakém adresáři (složce) se soubor nachází

- Adresa je složena ze jmen jednotlivých adresářů oddělených znakem /

Podobně jako URL

např. `/usr/local/bin/netbeans-8.0` – představuje cestu k

- `netbeans-8.0` – soubor pro spuštění programu Netbeans
- `bin` – adresář v adresáři `local`
- `local` – adresáři v adresáři `usr`
- `/` – kořenový adresář

Umístění souboru tak můžeme jednoznačně určit

Umístění souboru – absolutní a relativní cesta

- Adresa absolutního umístění souboru v systému souborů začíná kořenovým adresářem /
- Cesta k souboru může být také relativní vzhledem k nějakému pracovnímu (např. projektovému) adresáři
- Speciální význam mají adresáře
 - .. – odkazuje do adresáře o úroveň výše
 - . – je aktuální adresář
- Příklady
 - `/usr/local/bin/netbeans`
 - Relativní cesta vzhledem k `/usr/local/tmp` je `../bin/netbeans`
 - Relativní cesta vzhledem k `/usr/local/bin` je
 - `netbeans`
 - `./netbeans`

Typy souborů

Podle způsobu kódování informace v souboru rozlišujeme:

■ Textové soubory

- Přímě čitelné a jednoduše editovatelné

Běžným textovým editorem

■ Binární soubory

- Zpravidla potřebujeme specializovaný program pro čtení, zápis a modifikaci souboru
- Přístup k souboru tak spíše realizujeme prostřednictvím programového rozhraní

V obou případech je pro výměnu souboru a jejich použitelnost v jiných programech klíčový konkrétní způsob organizace údajů a informací uložených v souborech

Používání standardních formátů a to jak textových (např. XML, HTML, JSON, CSV), tak binárních (např. HDF).

Textové soubory

- Textový soubor je posloupnost znaků členěná na řádky

Zpravidla členěná na řádky. Není to nutné, ale zvyšuje čitelnost a usnadňuje zpracování souboru (po řádcích).
- EOL (End of Line) – znak konce řádku
- EOL je platformově závislý
 - CR – Carriage Return – Macintosh – `"\r"` – 0x0d
 - LF – Line Feed – Unix – `"\n"` – 0x0a
 - CR/LF – MS-DOS, Windows – `"\r\n"` – 0x0d 0x0a
- Každý znak je reprezentován jedním bajtem, případně 2 nebo více bajty

Viz znakové sady a kódování

Binární soubory

- Binární soubor je posloupnost bajtů
- Informace v binárním souboru je kódována vnitřním kódem počítače
- Do binárního souboru mohou být zapsány
 - bajt (byte)
 - jednoduché proměnné
 - pole
 - data celých objektů

V Javě lze využít tzv. [serializace](#)

Informace o typu souboru ani o způsobu kódování informací v něm uložených není v souboru obsažena. Správnou interpretaci přečteného souboru musí zajistit uživatelský program.

Binární soubory

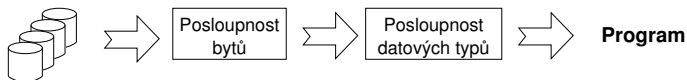
- Binární soubor je posloupnost bajtů
- Informace v binárním souboru je kódována vnitřním kódem počítače
- Do binárního souboru mohou být zapsány
 - bajt (byte)
 - jednoduché proměnné
 - pole
 - data celých objektů

V Javě lze využít tzv. [serializace](#)

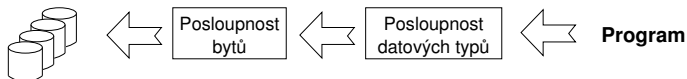
Informace o typu souboru ani o způsobu kódování informací v něm uložených není v souboru obsažena. Správnou interpretaci přečteného souboru musí zajistit uživatelský program.

Přístup k souborům

- Přenos informace (dat) z/do souboru lze rozdělit do několika vrstev
- Vrstva může poskytovat různý pohled na obsah souboru
- V základním pohledu je každý soubor **posloupnost bytů**
- Čtení ze souboru



- Zápis do souboru



Výhoda vrstveného přístupu je v možnosti jednoduše přidávat nové způsoby zpracování dat.

- K datům v souboru můžeme přistupovat dvěma základními způsoby: **sekvenčně** a **přímým** (náhodným) přístupem

Sekvenční přístup

- Při sekvenčním přístupu jsou jednotlivé byty načítány postupně
- Během načítání bytů mohou být data postupně interpretovaná
 - Např. po přečtení 4 bytů je možné interpretovat takovou posloupnost jako celé číslo typu `int`.*
- Na aktuální pozici v souboru ukazuje tzv. **kurzor**
- Každé další čtení ze souboru vrací příslušný počet přečtených bytů a o stejný počet bytů je kurzor posunut
- Při načítání se lze “vracet” pouze na začátek, nelze se vrátit např. o několik bytů zpět
- Při zápisu jsou postupně ukládány další byty na konec souboru
 - Při otevření souboru rozlišujeme kromě otevření pro čtení také otevření pro zápis nebo přidávání (`append`).*
- Sekvenční přístup načítání / zápisu je možné použít i pro jiné vstupy/výstupy než soubory uložené na disku
 - Např. Zpracování dat po sériovém portu, Ethernet nebo obecně data z Internetu*

Přímý přístup

- Při práci se soubory v přímém přístupu je možné zapisovat / číst na libovolné místo v souboru
- Práce se souborem se tak podobá přístupu k položkám v poli
- **Kurzor** lze libovolně nastavovat v rozsahu velikosti souboru (v bytech)
- Soubor musí být k dispozici

Vhodné pro soubory, které jsou přístupné na disku, a které lze "celé" kdykoliv načíst do paměti.
- Vhodné pokud známe vnitřní strukturu souboru a můžeme se přímo odkazovat na příslušné místo pro aktualizaci nebo načtení příslušné datové položky

<https://docs.oracle.com/javase/tutorial/essential/io/rafs.html>

Soubory a proudy

- Java rozlišuje soubory („*files*”) a proudy („*streams*”)
 - **Soubor** je množina údajů uložená ve vnější paměti počítače
 - **Proud** je přístup (nástroj) k přenosu informací z/do souboru, ale také z/do libovolného jiného média, které je schopné generovat nebo pojmout data jako posloupnost bytů

sítě, sériová linka, paměť, jiný program, atd.
- Informace může mít tvar znaků, bytů, skupin bytů, objektů, ...
- Přenos informace se děje ve více vrstvách v proudech (**streams**)
 1. **Otevření** přenosového proudu pro **byty** nebo **znaky**
 2. **Otevření** přenosového proudu pro **datové typy Javy**
 3. **Filtrace** dat podle dalších požadavků, např. bufferování, řádkování, atd.

Proudy v Javě (Standardní třídy)

- **Bytové – `FileInputStream` / `FileOutputStream`**
 - **`DataOutputStream`** – přenos primitivních datových typů
 - **`ObjectOutputStream`** – přenos objektů
 - **`BufferedOutputStream`** – bufferování
- **Znakové – `FileReader` / `FileWriter`**
 - **`BufferedReader`** – bufferování
 - **`StreamTokenizer`** – tokenizace

<https://docs.oracle.com/javase/8/docs/api/java/io/StreamTokenizer.html>

- **`RandomAccessFile`** – práce se soubory s náhodným přístupem
- **`File`** – zpracování souborů/adresářů: test existence, oddělovač adresářů/souborů, vytvoření, mazání, atd.

Využívá služeb operačního systému

- V Javě jsou příslušné třídy definovány v balíku **`java.io`** případně **`java.nio`**

Příklad – Soubor jako posloupnost bytů

Vytvoření kopie vstupního souboru

- Vstupní soubor postupně načítáme byte po byte a ukládáme do výstupního souboru

```
public void demoStreamCopy(String inputFile,
    String outputFile) throws IOException {
    FileInputStream in = new FileInputStream(inputFile);
    FileOutputStream out = new FileOutputStream(
        outputFile);
    int b = in.read(); // read byte of data
    while (b != -1) {
        out.write(b);
        b = in.read();
    }
    out.close();
    in.close();
}
```

DemoFileStream.java

Příklad ošetření chybových stavů

```
try {
    demo.demoStreamCopy(args[0], args[1]);
} catch (FileNotFoundException e) {
    System.err.println("File not found");
} catch (IOException e) {
    System.err.println("Error occurred during copying");
    e.printStackTrace();
}
```

■ Příklad spuštění programu

```
java DemoCopyException in2.txt out.txt
File not found
```

```
java DemoCopyException in.txt out2.txt
Error occurred during copying
java.io.IOException: Stream Closed
    at java.io.FileOutputStream.write(Native Method)
    at java.io.FileOutputStream.write(FileOutputStream.java:295)
    at DemoCopyException.demoStreamCopy(DemoCopyException.java:16)
    at DemoCopyException.main(DemoCopyException.java:24)
```

Proč jsou chyby různé?

Příklad – DemoCopyException

```
public void demoStreamCopy(String inputFile, String
    outputFile) throws IOException {
    FileInputStream in = new FileInputStream(inputFile);
    FileOutputStream out = new FileOutputStream(outputFile);
    if (outputFile.equalsIgnoreCase("out2.txt")) {
        out.close();
    }
    int b = in.read(); // read byte of data
    while (b != -1) {
        out.write(b);
        b = in.read();
    }
}
```

DemoCopyException.java

Soubor jako posloupnost primitivních typů – Zápis

- Pro zápis hodnoty základního datového typu jako posloupnost bytů přidáme další vrstvu **DataOutputStream**

```
String fname = args.length > 0 ? args[0] : "out.bin";
```

```
DataOutputStream out = new DataOutputStream(  
    new FileOutputStream(fname));
```

```
for (int i = 0; i < 10; ++i) {  
    double d = (Math.random() % 100) / 10.0;  
    out.writeInt(i);  
    out.writeDouble(d);  
    System.out.println("Write " + i + " " + d);  
}
```

DemoFilePrimitiveTypesWrite.java

Soubor jako posloupnost primitivních typů – Čtení

```
String fname = args.length > 0 ? args[0] : "out.bin";
DataInputStream in = new DataInputStream(
    new FileInputStream(fname));

for (int i = 0; i < 10; ++i) {
    int v = in.readInt();
    double d = in.readDouble();
    System.out.println("Read " + v + " " + d);
}
```

DemoFilePrimitiveTypesRead.java

Co se stane když zaměníme pořadí načítání `readInt` a `readDouble`?

Soubor primitivních typů a objektů

- Uvedenými metodami lze zapisovat a číst pouze tzv. **serializovatelné objekty**, mezi které patří
 - Primitivní datové typy
 - Řetězce a pole primitivních typů
 - Složitější objekty, pokud implementují rozhraní **Serializable**
- Rozhraní **Serializable** nepředefisuje žádnou metodu, je značkou, že objekt chceme serializovat

Pro vytvoření příslušné implementace pro převod hodnot do/z posloupnosti bytů.
- Pro serializaci musí být každá datová položka serializovatelná
- nebo označena, že nebude serializována klíčovým slovem **transient**

<https://docs.oracle.com/javase/8/docs/platform/serialization/spec/serialTOC.html>

Informativní

Příklad serializace 1/3

```
import java.io.Serializable;  
  
public class Customer implements Serializable {  
  
    private String name;  
    private String surname;  
    private int age;  
  
    public Customer(String name, String surname, int age) {  
        this.name = name;  
        this.surname = surname;  
        this.age = age;  
    }  
}
```

Customer.java

Příklad serializace 2/3

```
void write(Customer customer, String fname) throws
    IOException {
    try (ObjectOutputStream out = new
        ObjectOutputStream(new FileOutputStream(fname))) {
        out.writeObject(customer);
    }
}
```

```
Customer read(String fname) throws IOException,
    ClassNotFoundException {
    ObjectInputStream in = new ObjectInputStream(new
        FileInputStream(fname));
    return (Customer) in.readObject();
}
```

DemoObjectSerialization.java

Příklad serializace 3/3

```
Customer customer = new Customer("AAA", "BBB", 47);
System.out.println("Customer: " + customer);
write(customer, fname);
customer = new Customer("ZZZ", "WWW", 17);
System.out.println("Customer: " + customer);
customer = read(fname);
System.out.println("Customer: " + customer);
```

■ Příklad výstupu

```
Customer: AAA BBB age: 47
```

```
Customer: ZZZ WWW age: 17
```

```
Customer: AAA BBB age: 47
```

Soubory s náhodným přístupem 1/2

- Třída **RandomAccessFile** pro zápis/čtení do/z libovolného místa v souboru

```
public void write(String fname, int n) throws
    IOException {

    RandomAccessFile out =
        new RandomAccessFile(fname, "rw");

    for (int i = 0; i < n; ++i) {
        out.writeInt(i);
        System.out.println("write: " + i);
    }
    out.close();
}
```

Soubory s náhodným přístupem 2/2

- Pro přístup na konkrétní položku je nutné určit „adresu“ položky v souboru jako pozici v počtu bytů od začátku souboru

```
final int SIZE = Integer.SIZE / 8;
```

```
RandomAccessFile in =  
    new RandomAccessFile(fname, "r");
```

```
for (int i = 0; i < 5; ++i) {  
    in.seek(i * 2 * SIZE);  
    int v = in.readInt();  
    System.out.println("read: " + v);  
}
```

DemoRandomAccess.java

Textově orientované soubory

- Při čtení a zápisu je nutné zajistit konverzi znaků

Kódování

- Příklad zápisu s využitím třídy `PrintWriter`

```
public void write(String fname) throws IOException {
    String months[] = {"jan", "feb", "mar", "apr", "may",
        "jun", "jul", "aug", "sep", "oct", "nov", "dec"};
};

PrintWriter out = new PrintWriter(fname, "UTF-8");
for (int i = 0; i < months.length; ++i) {
    out.println(months[i]);
}
out.close();
}
```

Příklad čtení textového souboru

- Pro čtení můžeme využít třídy **Scanner** podobně jako při čtení ze standardního vstupu

```
public void start() throws IOException {
    String fname = "text_file.txt";

    write(fname);

    FileInputStream in = new FileInputStream(fname);
    Scanner scan = new Scanner(in);
    while (scan.hasNext()) {
        String str = scan.next();
        System.out.println("Read: " + str);
    }
    in.close();
}
```

DemoTextFile.java

Shrnutí přednášky

Diskutovaná témata

- Výjimky
- Soubory a přístup k souborům
- Typy souborů (textový a binární)
- Práce se soubory v Javě
 - Binární soubory
 - Textové soubory
- Příště: GUI v Javě

Diskutovaná témata

- Výjimky
- Soubory a přístup k souborům
- Typy souborů (textový a binární)
- Práce se soubory v Javě
 - Binární soubory
 - Textové soubory
- Příště: GUI v Javě