7 Výjimky a práce se soubory

1 Výjimky

Při programování máme často tendenci spoléhat se na to, že všechno je tak, jak má být, a vše probíhá podle našich ideálních představ. Tento přístup nám ale může připravit nejedno nemilé překvapení, jak si ukážeme na příkladu funkce, která vezme textový řetězec reprezentující číslo v binární podobě a převede jej na příslušnou hodnotu typu `int`.

Připomeňme si, jak se tento převod dá provést. Předpokládejme, že máme číslo x zapsané v binární soustavě s pomocí jedniček a nul, např. $x = (01011001)_2$. Označme si jednotlivé číslice zleva doprava $d_n, d_{n-1}, \ldots, d_0$. Hodnotu x pak můžeme vyjádřit jako součet

$$x = d_n \cdot 2^n + d_{n-1} \cdot 2^{n-1} + \cdots + d_1 \cdot 2^1 + d_0 \cdot 2^0.$$

Pro náš příklad tedy platí:

$$x = 0 \cdot 2^7 + 1 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0.$$

Po zjednodušení dostáváme $x = 2^6 + 2^4 + 2^3 + 2^0$, tj. $x = 89$. Z následující tabulky si můžeme udělat intuitivní představu o získání hodnoty čísla zapsaného v binární soustavě.

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	1	0	1	1	0	0	1
d_7	d_6	d_5	d_4	d_3	d_2	d_1	d_0

Jinými slovy, pro každou číslici na pozici i čísla zapsaného v binární soustavě, číslováno od nuly a zprava doleva, spočítáme hodnotu 2^i. Pokud je číslice na dané pozici rovna 1, zahrneme hodnotu 2^i do výsledného součtu. Naše funkce pro převod čísel zapsaných v binární soustavě by pak mohla vypadat třeba následovně:

```
1 public static int binaryToInt(String number) {
2   int result = 0;
3   int digitValue = 1;
4
5   for (int i = number.length() - 1; i >= 0; i--) {
6     int digit = number.charAt(i);
```

```
    if (digit == '1') result += digitValue;
    digitValue *= 2;
  }
  return result;
}
```

Při letmém vyzkoušení metoda funguje dobře.

```
binaryToInt("001")       // ==> 1
binaryToInt("011")       // ==> 3
binaryToInt("01011001")  // ==> 89
binaryToInt("0101010")   // ==> 42
```

Zkuste se zamyslet, co se stane, když se v řetězci místo nul a jedniček objeví i nějaká dvojka nebo dokonce trojka. Co se stane, když vstup bude příliš velký, např. jednička následovaná třicetidvěma nulami? Metoda v obou případech vrátí výsledek. Nedá se však říct, že tento výsledek bude správný. To otevírá otázku, jak se vypořádat s chybami tohoto druhu.

1.1 Výjimky a jejich vyvolání

V reálných programech je prakticky nemožné vyhnout se nestandardním situacím, které mají dopad na korektní chování programu. Problematických míst je celá řada, například uživatel zadá na vstup hodnotu, která neodpovídá očekávanému vstupu (viz příklad výše), program se pokusí pracovat se souborem, ke kterému nemá oprávnění, nebo se může přerušit síťové spojení. Pro situace jako jsou tyto nabízí jazyk Java důmyslný mechanismus, který umožňuje identifikovat problematické situace a následně, na vhodném místě, na ně reagovat. Tento mechanismus se označuje jako systém *výjimek*.

Poznámka 1 *Pro lepší představu, jak funguje systém výjimek, si opět vypomůžeme analogií z reálného světa. Dejme tomu, že dělník v továrně narazí na problém, který není schopen sám vyřešit, např. pokazí se mu stroj, dostane vadné součástky. Tento dělník o tomto problému vyhotoví zprávu (záznam) a předá ji svému nadřízenému, například mistrovi. Ten podle typu problému buď problém vyřeší (nechá opravit stroj, objedná jiné součástky), nebo, pokud je problém nad jeho možnosti, předá zprávu od dělníka svému nadřízenému, vedoucímu výroby, aby to vyřešil. Pokud vedoucí výroby dokáže problém vyřešit, tak jej vyřeší, jinak zprávu od dělníka posouvá dál svému nadřízenému. Takto by se dalo pokračovat dál v hierarchii továrny, než by se našel někdo, kdo problém vyřeší. Systém výjimek funguje v podobném duchu, jen s tím rozdílem, že zpráva o problému je běžný objekt a místo vztahu nadřízený a podřízený zaměstnanec máme vztah volající metoda a volaná metoda. Jinými slovy, pokud ve volané metodě vznikne problém, který není schopna vyřešit, vrátí volající metodě informaci o problému, a ta jej může buď vyřešit, nebo předat dál.*

Výjimka je objekt, který je potomkem třídy `Exception`, jenž identifikuje, kde v kódu došlo k chybě a o jaký typ chyby se jedná. Například, chtěli-li bychom identifikovat problém s velkými čísly v našem příkladu,[1] vytvořili bychom si třídu:

[1] Datový typ `int` je schopen pojmout pouze celočíselné hodnoty z intervalu -2^{31} do $2^{31}-1$.

```
public class IntegerOverflowException extends Exception { }
```

Pokud metoda narazí na problém, se kterým se není schopna vypořádat, vytvoří objekt výjimky a ten předá volající metodě pomocí operace `throw`. To se označuje jako *vyvolání výjimky*. V našem případě by vyvolání výjimky mohlo vypadat následovně.

```
if (number.length() > 31) throw new IntegerOverflowException();
```

Vyvolání výjimky způsobí okamžité ukončení metody a běh programu je přesunut do volající metody, která se musí se vzniklou výjimkou vypořádat. V tomto případě nedochází k předání návratové hodnoty z volané metody. Aby s touto evantualitou mohla volající metoda počítat, musí mít metoda, která chce použít výjimky, ve své deklaraci uvedeno, jakou výjimkou může skončit.[2] Slouží k tomu klíčové slovo `throws` za nímž následuje výčet typů výjimek. Například:

```
public static int binaryToInt(String number) throws IntegerOverflowException {
  // ...
}
```

Jelikož je výjimka běžný objekt, můžeme do něj umístit další informace, které umožní lépe identifikovat problém. V našem příkladu bychom mohli chtít identifikovat neplatné číslice. Vytvoříme si proto třídu:

```
1  public class InvalidDigitException extends Exception {
2
3    private final char invalidDigit;
4
5    public InvalidDigitException(char invalidDigit) {
6      this.invalidDigit = invalidDigit;
7    }
8
9    public char getInvalidDigit() {
10     return invalidDigit;
11   }
12 }
```

Při vyvolání výjimky uvedeme, který znak dělal problém.

```
if ((digit != '0') && (digit != '1')) throw new InvalidDigitException(digit);
```

[2] Toto pravidlo však pro některé druhy výjímek neplatí, např. pro výjimky dědící ze třídy `RuntimeException`, které slouží k indikaci chyby v programu. Typickým příkladem je výjimka `NullPointerException`, která je vyvolána, pokud dojde k zavolání metody nad hodnotou `null`.

1.2 Ošetření výjimek

Když si naši metodu `binaryToInt` rozšířenou o výjimky vyzkoušíme a dáme ji zavolat z jiné metody, bude překladač trvat na tom, abychom doplnili informaci o tom, že i metoda, která `binaryToInt` volá, může skončit stejnými výjimkami.

```
1 private static void testConversion() throws IntegerOverflowException,
  ↪  InvalidDigitException {
2   System.out.println(binaryToInt("01011001"));
3   System.out.println(binaryToInt("10201"));
4   System.out.println(binaryToInt("001"));
5 }
```

Tento požadavek dává smysl. Není-li možné korektně dokončit jednu z operací v těle metody (viz metoda `testConversion()`), není možné dokončit ani celou metodu. V našem případě volání metody `binaryToInt` na řádku 3 skončí výjimkou a metoda `testConversion` dál nepokračuje.

Podobné to bude i v případě metody `main` volající metodu `testConversion()`, opět bude překladač trvat na tom, abychom uvedli, že metoda může skončit s výjimkami.

```
public static void main(String[] args) throws IntegerOverflowException,
 ↪  InvalidDigitException {
  testConversion();
}
```

V tomto případě, pokud dojde k vyvolání výjimky, bude to mít fatální důsledky. Vyvolání výjimky povede k okamžitému ukončení programu, jelikož jsme nikde neuvedli, co se má stát, když k výjimce dojde.

Pro ošetření výjimek má jazyk Java konstrukci `try-catch`. Za klíčovým slovem `try` následuje blok kódu, a pokud v tomto bloku kódu dojde k výjimce, je provádění kódu přerušeno a začne se provádět kód v části `catch`. Vyzkoušejte si.

```
try {
  System.out.println(binaryToInt("01011001"));
  System.out.println(binaryToInt("10201"));
  System.out.println(binaryToInt("001"));
} catch (Exception ex) {
  System.out.println("Doslo k vyjimce: " + ex);
}
```

Za klíčovým slovem `catch` se uvádí typ výjimky, kterou chceme ošetřit, a název proměnné, do které bude uložen objekt výjimky. To nám umožňuje rozdělit si výjimky podle typů a odpovídajícím způsobem na ně reagovat.

```
1 try {
2   /* ... */
3 } catch (InvalidDigitException e) {
4   System.out.println("Neplatna cislice: " + e.getInvalidDigit());
5 } catch (IntegerOverflowException e) {
6   System.out.println("Cislo je prilis velke");
7   e.printStackTrace();
8 }
```

Všimněme si, že u výjimky typu `InvalidDigitException` jsme využili informaci o neplatné číslici a zobrazili ji na výstupu. U výjimky typu `IntegerOverflowException` jsme na řádku 7 zavolali její metodu `printStackTrace` (zděděnou z třídy `Exception`), která vypíše, na kterém řádku došlo k výjimce a posloupnost volání metod, jak se k této výjimce dospělo.

Pokud nepotřebujeme rozlišovat mezi jednotlivými typy výjimek, můžeme sloučit jejich zpracování do jednoho bloku `catch` následovně:

```
1 try {
2   /* ... */
3 } catch (InvalidDigitException | IntegerOverflowException e) {
4   e.printStackTrace();
5 }
```

Konstrukci `try-catch` lze ještě obohatit o blok označený `finally`, který se provede vždy bez ohledu na to, jestli došlo k výjimce, nebo ne. Toho se využívá často při práci se soubory, jak uvidíte v následujícím textu.

2 Balíčky

Předtím, než se dostaneme k práci se soubory, uděláme odbočku k organizaci tříd. Není neobvyklé, že projekt v Javě obsahuje tisíce nebo desetisíce tříd. Samotný projekt může mít desítky nebo stovky tříd, ale obvykle jsou do projektu připojeny i další knihovny, díky kterým počet tříd, ze kterých se aplikace skládá, naroste.[3] Při takto velkých počtech tříd už je potřeba mít nástroj, který nám umožní třídy organizovat a řešit situace, kdy máme dvě třídy stejného jména. Například existuje rozhraní `List` reprezentující obecný seznam a třída `List` reprezentující komponentu grafického uživatelského rozhraní.

Tímto nástrojem jsou balíčky. Balíčky z pohledu souborového systému odpovídají jednotlivým adresářům a z pohledu kódu určují přesné jméno třídy. Ukažme si to prakticky.

Uvažujme, že máme třídu `Foo` uloženou v souboru `src/cz/upol/jj1/lecture07/Foo.java`. Z pohledu jazyka Java se tato třída bude nacházet v balíčku `cz.upol.jj1.lecture07`. Což je nutné v kódu deklarovat pomocí klíčového slova `package` na začátku souboru se zdrojovým kódem.

[3] Jen samotná standardní knihovna Javy 21 má přes 15 tisíc tříd.

```java
package cz.upol.jj1.lecture07;

public class Foo {
// ...
}
```

Poznámka 2 *Pokud nebude souhlasit adresář, ve kterém je soubor se zdrojovým kódem uložen, a název balíčku uvedený na začátku zdrojového kódu, oznámí to překladač jako chybu. Protože jsou třídy v balíčku uloženy jako soubory, máme garantované, že každá třída je unikátně identifikovaná svým jménem.*[4]

Název třídy a balíček, ve kterém se tato třída nachazí, jednoznačně identifikují třídu. V našem případě by jednoznačným identifikátorem třídy bylo `cz.upol.jj1.lecture07.Foo` a vytvoření instance třídy bychom provedli operátorem `new` následovně.

```java
cz.upol.jj1.lecture07.Foo foo = new cz.upol.jj1.lecture07.Foo();
```

Poznámka 3 *Název balíčku volíme ideálně tak, aby byl „celosvětově" unikátní, aby nehrozilo, že když přidáme do projektu nějakou knihovnu, že bude obsahovat stejně pojmenované třídy a balíčky. Obvykle se proto jako název balíčku volí doména organizace,*[5] *kde projekt vznikl, případně název projektu, pokud je dostatečně unikátní.*

Používat pro práci s třídami jejich plná jména by bylo obtěžující. Proto balíčky představují, tzv. *jmenné prostory*, kde každá třída má své jednoznačně přirazené jméno a třídy v rámci jednoho balíčku (*jmeného prostoru*) mohou používat názvy ostatních tříd, aniž by bylo potřeba uvádět jejich plné jméno. Jinými slovy u názvů tříd v rámci jednoho balíčku není potřeba uvádět název balíčku.

Pokud chceme používat třídu z jiného balíčku, musíme buď uvést její plné jméno, tj. včetně názvu balíčku, nebo třídu musíme *naimportovat*. To uděláme tak, že na začátek souboru uvedeme za klíčovým slovem `import` název třídy. Následně bude třída k dispozici jen pod svým názvem, jako by byla součástí aktuálního balíčku.

```java
package com.acme.project;

import cz.upol.jj1.lecture08.Foo;

public class Bar { /* ... */ }
```

Alternativně můžeme za název balíčku uvést `.*`, čímž se importují všechny veřejné třídy z daného balíčku. Avšak mnohem běžnější je vyjmenovat jen ty třídy, které používáme.

Poznámka 4 *Pro snazší práci jsou implicitně importované všechny třídy z balíčku* `java.lang`. *Nemusíme proto psát, např.* `java.lang.Math`, *ale jen* `Math`.

[4] Nemůžeme mít dva soubory stejného jména.

[5] Zapsaná v opačném pořadí.

Poznámka 5 *Pokud u třídy neuvedeme* `public class Foo`*, ale jen* `class Foo`*, bude dostupná jen pro třídy z daného balíčku. Podobně, pokud u metody nebo atributu neuvedeme klíčové slovo určující viditelnost metody, resp. atributu, bude daná metoda či atribut přístupná všem třídám z daného balíčku.*

Poznámka 6 *Pokud soubor se zdrojovým kódem umístíme do kořenového adresáře, není potřeba uvádět na začátku souboru klíčové slovo* `package`*. V takovém případě je třída umístěna v tzv. výchozím balíčku (default package). Není to však dobrá praxe a třídy bychom měli vždy mít umístěné (organizované) v balíčcích se souvisejícími třídami.*

3 Práce se soubory

Třídy pro práci se soubory najdeme v balíčku `java.io` a dají se rozdělit do dvou kategorií, (i) třídy pro práci s binárními daty a (ii) třídy pro práci s textovými daty. Záměrně je zde použito slovo data, protože tyto třídy se dají použít nejen pro práci se soubory, ale používají se například i pro výstup na konzoli, komunikaci po síti nebo pro práci s daty v paměti.

3.1 Práce s textovými soubory

Pro práci s textovými soubory máme k dispozici dvě abstraktní třídy `Writer` a `Reader`, kde třída `Writer` nabízí metody `write` pro zápis jednotlivých znaků a případně řetězců a třída `Reader` má metody `read`, kterými je možné číst jednotlivé znaky, případně načíst pole znaků.

Třídy `Reader` a `Writer` jsou abstraktní a neříkají odkud a jak se mají data číst nebo zapisovat. Pokud chceme pracovat se soubory máme implementace těchto tříd `FileReader` a `FileWriter`, kterým předáme soubor (cestu k souboru)[6] a pomocí metod `read` a `write` můžeme číst a zapisovat do souboru.

Třídy `Reader` a `Writer` poskytují velmi omezené rozhraní, proto standardní knihovna Javy obsahuje třídy, které přidávávají další metody pro pohodlnější práci s daty. Za zmínku stojí třídy `BufferedReader` a `BufferedWriter`, které data, se kterými se pracuje, udržují v bufferu, čímž snižují počet operací s daným souborem. Třída `BufferedReader` navíc nabízí užitečnou metodu `readLine()`, která umožňuje číst jednotlivé řadky souboru.

Pro pohodlnější zápis textových dat máme k dispozici třídu `PrintWriter`, která obsahuje metody `print` určené pro pohodlnější zápis základních datových typů.

Jak se s těmito třídami pracuje, si budeme demonstrovat na příkladu jednoduché databáze zaměstnanců, kde u každého jednoho zaměstnance budeme mít tři atributy jméno, věk a plat.

```
1 public record Employee(String name, int age, double salary) { }
```

A tato data budeme mít uložena v poli:

```
1 Employee[] employees = {
2   new Employee("Alice", 20, 25000),
```

[6] Volitelně i znakovou sadu, která se má použít pro uložení/čtení znaků z/do souboru.

```
3   new Employee("Bob", 42, 145000),
4   new Employee("Chuck", 37, 40230.56) };
```

3.1.1 Zápis do textového souboru

Ukažme si pro začátek zápis tohoto pole do souboru.

```
1  FileWriter writer = null;
2  PrintWriter pwr = null;
3  try {
4     writer = new FileWriter("/tmp/employees.csv");
5     pwr = new PrintWriter(writer);
6
7     for (Employee emp : employees) {
8       pwr.print(emp.name());
9       pwr.print(",");
10      pwr.print(emp.age());
11      pwr.print(",");
12      pwr.print(emp.salary());
13      pwr.println();
14    }
15 } catch (IOException e){
16   System.out.println("Unable to write a file.");
17   e.printStackTrace();
18 } finally {
19   try {
20     if (pwr != null) pwr.close();
21     if (writer != null) writer.close();
22   } catch (IOException e) {
23     System.out.println("Unable to close a file.");
24     e.printStackTrace();
25   }
26 }
```

Nejdříve si vytvoříme třídy typu `Writer`, které se starají o zápis do souboru (řádky 4 a 5), třída `FileWriter` nám umožní zapisovat do zadaného souboru a třída `PrintWriter` nám poskytne přívětivější rozhraní pro zápis dat, které využíváme na řádcích 7 až 14.

Jednotlivé operace při práci se souborem mohou selhat, tj. skončit výjimkou. Například pokud nelze zapsat do souboru z důvodů nedostatku oprávnění, místa v paměti, neplatné cesty apod. Proto jsou všechny operace souborem obaleny blokem `try-catch`. Pokud taková situace nastane, je proveden `catch` blok na řádcích 16 a 17.

Po skončení práce se souborem, je nutné jej uzavřít, ať už metoda probíhá standardně nebo došlo k vyjímce. Proto v bloku `finally` voláme metodu `close`. Avšak tato metoda může skončit výjimkou, a proto ji opět balíme do bloku `try-catch`.

Protože práce se soubory a podobnými zdroji,[7] kde si musíme hlídat jejich uzavření, je v takovéto podobě nepříliš komfortní, existuje varianta bloku `try`, která se označuje jako *try-with-resources*. Tato varianta bloku `try` se postará o inicilizaci i automatické uzavření zdrojů po skončení provádění bloku `try`. Od běžného bloku `try-catch` se liší v tom, že za `try` provedeme nastavení proměnných (objektů), které chceme po skončení bloku `try` uzavřít, a na konci bloku dojde k jejich automatickému uzavření, aniž bychom museli definovat blok `finally`.

Použití by v našem případě vypadalo následovně:

```
1 try (FileWriter writer = new FileWriter("/tmp/employees.csv");
2    PrintWriter pwr = new PrintWriter(writer)) {
3
4    for (Employee emp : employees) {
5      // ...
6    }
7 } catch (IOException e){
8   System.out.println("Unable to write a file.");
9   e.printStackTrace();
10 }
```

Jednotlivé objekty, které chceme uzavřít definujeme na řádcích 1 a 2, a abychom je takto mohli použít, musí implementovat rozhraní `AutoClosable`, což je v případě tříd `FileWriter` a `PrintWriter` splněno.

Tímto se nám podařilo kód zestručnit a zpřehlednit, ale z pohledu návrhu to není stále ideální řešení, protože fakticky dělá dvě věci, (i) stará se o práci se souborem (otevření a zavření), (ii) zapisuje data do souboru. Je dobré tyto dvě aktivity oddělit, například proto, abychom mohli změnit způsob zápisu dat nebo abychom mohli změnit, kam se data zapisují.

Proto si vyčleníme zápis do samostatné metody, například následovně:

```
1 private static void writeEmployees(Writer writer, Employee[] employees) {
2   try (PrintWriter pwr = new PrintWriter(writer)) {
3      for (Employee emp : employees) {
4        pwr.print(emp.name());
5        // ...
6        pwr.println();
7      }
8   }
9 }
```

[7] Např. síťová spojení, databázová spojení.

A kód zjednodušíme:

```
try (FileWriter writer = new FileWriter("/tmp/employees.csv")) {
  writeEmployees(writer, employees);
} catch (IOException e){
  System.out.println("Unable to write a file.");
  e.printStackTrace();
}
```

Pokud bychom uvažovali pouze práci se soubory, mohl by se tento krok zdát zbytečným. Avšak uvážíme-li, že data nemusíme zapisovat jen do souboru, je tento krok velice praktický.

Existuje například třída `StringWriter`, která je potomkem třídy `Writer` a která uchovává zapsaný text v paměti ve formě řětězce, který můžeme následně získat.

Můžeme udělat například:

```
StringWriter swr = new StringWriter();
writeEmployees(swr, employees);
System.out.println(swr.toString());
```

Toto řešení má řadu praktických aplikací, zejména se uplatňuje při testování.

3.1.2 Čtení ze souboru

Při práci s textovými soubory většinou zpracováváme postupně jednotlivé řádky. Proto se způsob čtení dat nebude příliš odlišovat od zápisu. Místo třídy `PrintWriter` použijeme třídu `BufferedReader`, abychom mohli číst jednotlivé řádky a následně použijeme funkce pro práci s řetězci k získání konkretních hodnot. Pro úplnost dodejme, že metoda `BufferedReader.readLine()` vrací jednotlivé řádky souboru, a pokud narazí nakonec vrací `null`.

Kód pro zpracování dat by tedy mohl vypadat následovně:

```
try (Reader reader = new FileReader("/tmp/employees.csv");
     BufferedReader brd = new BufferedReader(reader)) {
  String line;
  do {
     line = brd.readLine();
     if (line != null) {
        String[] parts = line.split(",");
        String name = parts[0];
        int age = Integer.parseInt(parts[1]);
        double salary = Double.parseDouble(parts[2]);
```

```
11
12          Employee employee = new Employee(name, age, salary);
13          System.out.println(employee); // nebo jina vhodna operace
14        }
15    } while (line != null);
16 } catch (IOException e) {
17     System.out.println("Unable to read a file.");
18     e.printStackTrace();
19 }
```

Poznamenejme, že při zpracování jednotlivých řádků souboru (na řádcích 7 až 10) může dojít k několika výjimkám, které nejsou podchyceny, například pokud řádek neobsahuje minimálně dva znaky `','` nebo pokud pro věk a plat nejsou v souboru uvedeny odpovídající číselné hodnoty.

Podobně jako v případě zápisu dat, nemusíme číst data pouze ze souboru. Můžeme data například načítat ze zadeného řetězce, k tomu slouží třída `StringReader`, a prakticky si to můžete vyzkoušet nahrazením prvního řádku ve výše uvedeném kódu následujícím řádkem:

```
1 try (Reader reader = new StringReader("David,60,12345.80\nEman,45,2345.2");
```

3.2 Práce s binárními soubory

Práce s textovými soubory je z uživatelského pohledu velice přívětivá, protože pro manipulaci s daty nám stačí běžný textový editor a při programování si vystačíme s funkcemi pro práci s řetězci. Na druhou stranu tyto funkce patří mezi výpočetně náročnější, uložení dat v textovém formátu nebývá příliš úsporné a je relativně snadné vytvořit nevalidní soubor.[8] Proto se v praxi vedle textových souborů používají soubory binární, kde místo s jednotlivými znaky pracujeme s jednotlivými byty.

Práce s binárními soubory je v mnoha ohledech podobná práci s textovými soubory. Máme dvě abstraktní třídy `OutputStream` a `InputStream`, které umožňují zapisovat a číst jednotlivé byty (případně sekvence bytů). Tyto třídy mají potomky `FileOutputStream` a `FileInputStream`, které zapisují a čtou data do/ze souboru. Abychom nemuseli pracovat přímo s jednotlivými byty, máme k dispozici třídy `DataOutputStream` a `DataInputStream`, které poskytují metody pro pohodlný zápis běžných datových typů.

3.2.1 Zápis do souboru

Jak jsme uvedli, práce s binárními soubory je velmi podobná práci s textovými soubory, proto i kód bude velmi podobný. Vytvoříme si proto nejdříve metodu, která zapíše záznamy do objektu typu `OutputStream`.

[8] Často se objevují další problémy, například, máme-li soubor, kde jsou hodnoty odděleny čárkami, ovevírá se otázka, jak reprezentovat text s čárkou. Řešení podobných okrajových případů často v konečném důsledku vede k vytvoření komplikovaného programu nebo formátu dat.

```java
private static void writeEmployees(OutputStream output, Employee[] employees) throws
  ↪  IOException {
  try (DataOutputStream dos = new DataOutputStream(output)) {
     for (Employee emp : employees) {
       dos.writeUTF(emp.name());
       dos.writeInt(emp.age());
       dos.writeDouble(emp.salary());
     }
  }
}
```

Všimněme si, že třída `DataOutputStream` se stará o pohodlný zápis hodnot podobně jako třída `PrintWriter` a že nám v binárním formátu odpadají oddělovače hodnot, protože každá zapsaná hodnota má přesně určený binární tvar. Dále si všimněme bloku `try`(-with-resource), kterému chybí část `catch`. To je úplně v pořádku, protože v této metodě nevíme, jak zareagovat na výjimky, proto jejich zpracování delegujeme do volající metody. Blok `try` v tomto případě slouží k uzavření nepoužívaných zdrojů.

Pokud budeme chtít data zapsat do souboru, vytvoříme objekt typu `FileOutputStream` a předáme jej výše popsané metodě, například následovně:

```java
try (OutputStream output = new FileOutputStream("/tmp/employees.dat")) {
  writeEmployees(output, employees);
} catch (IOException e){
  System.out.println("Unable to write a file.");
  e.printStackTrace();
}
```

Pokud se budeme chtít na obsah souboru podívat, je dobré použít vhodný nástroj, který umí zobrazit binární soubory v hexadecimální podobě. Buď se jedná o samostatné aplikace nebo takové nástroje bývají součástí některých textových editorů nebo souborových manažerů. V unixových operačních systémech pak existují nástroje pro příkazovou řádku, např. `hexdump`, který námi vytvořený soubor zobrazí následovně:

```
00000000  00 05 41 6c 69 63 65 00  00 00 14 40 d8 6a 00 00  |..Alice....@.j..|
00000010  00 00 00 00 03 42 6f 62  00 00 00 2a 41 01 b3 40  |.....Bob...*A..@|
00000020  00 00 00 00 00 05 43 68  75 63 6b 00 00 00 25 40  |......Chuck...%@|
00000030  e3 a4 d1 eb 85 1e b8                              |.......|
```

Podobně jako nám třída `StringWriter` umožnila zapsat výstup do textového řětězce, máme k dispozici třídu `ByteArrayOutputStream`, díky niž se data zapíší do pole bytů, se kterým můžeme dál pracovat, jak ukazuje následující kód.

```java
ByteArrayOutputStream bos = new ByteArrayOutputStream();
```

```
writeEmployees(bos, employees);
System.out.println(Arrays.toString(bos.toByteArray()));
```

3.2.2 Čtení ze souboru

Při čtení ze souboru postupujeme analogicky jako u zápisu.

```
try (InputStream input = new FileInputStream("/tmp/employees.dat");
     DataInputStream dis = new DataInputStream(input)) {
  while (dis.available() > 0) {
    String name = dis.readUTF();
    int age = dis.readInt();
    double salary = dis.readDouble();

    Employee employee = new Employee(name, age, salary);
    System.out.println(employee); // nebo jina vhodna operace
  }
} catch (IOException e){
  System.out.println("Unable to write a file.");
  e.printStackTrace();
}
```

Ve smyčce (řádky 3 až 10), za podmínky, že jsou k dispozici data, čteme jednotlivé záznamy. Všimněme si, že čteme přímo hodnoty proměnných a nemusíme provádět konverze řetězců na číselné hodnoty.

Analogicky, jako jsme četli hodnoty z řetězce pomocí třídy `StringReader`, můžeme číst data z pole bytů pomocí třídy `ByteArrayInputStream`.

3.3 Práce se soubory s objekty

V příkladech, které jsme si ukázali, jsme se vždy museli postarat o uspořádání dat v souboru a jejich vhodnou reprezentaci. Pokud nám nezáleží na konkrétním formátu dat, můžeme tuto činnost delegovat na objekty standardní knihovny jazyka Java. K dispozici máme dvě třídy `ObjectOutputStream` a `ObjectInputStream`, kde první mimo jiné nabízí metodu `writeObject`, která zapíše obecný objekt do zadaného `OutputStreamu`, a druhá třída nabízí metodu `readObject`, která takto zapsaný objekt přečte.

Pokud chceme takto objekty ukládat a načítat, musí jejich třída implementovat rozhraní `java.io.Serializable`, například následovně:

```
public record Employee(String name, int age, double salary) implements Serializable { }
```

Toto rozhraní nemá žádnou metodu a slouží pouze k signalizaci, že je možné objekt převést do binární podoby.

Použití tříd `ObjectOutputStream` a `ObjectInputStream` se nevymyká principům, které jsme již viděli. Například zápis je realizováno následovně:

```
try (OutputStream output = new FileOutputStream("/tmp/employees.obj");
     ObjectOutputStream oos = new ObjectOutputStream(output)) {
  oos.writeObject(employees);
}
```

A čtení probíhá v podobném duchu:

```
try (InputStream input = new FileInputStream("/tmp/employees.obj");
     ObjectInputStream ois = new ObjectInputStream(input)) {
  Employee[] employees = (Employee[]) ois.readObject();
  System.out.println(Arrays.toString(employees)); // nebo jina vhodna operace
}
```

Všimněme si, že pro uložení/načtení celého pole hodnot nám stačí zavolání jedné metody. Je to však vykoupeno komplikovanějším formátem dat, který je prakticky použitelný jen v rámci ekosystému jazyka Java. Pro představu, soubor s uloženými daty vypadá následovně:

```
00000000  ac ed 00 05 75 72 00 15  5b 4c 6c 65 63 74 75 72  |....ur..[Llectur|
00000010  65 30 37 2e 45 6d 70 6c  6f 79 65 65 3b e4 6e 77  |e07.Employee;.nw|
00000020  91 d2 00 6c 1e 02 00 00  78 70 00 00 00 03 73 72  |...l....xp....sr|
00000030  00 12 6c 65 63 74 75 72  65 30 37 2e 45 6d 70 6c  |..lecture07.Empl|
00000040  6f 79 65 65 00 00 00 00  00 00 00 00 02 00 03 49  |oyee...........I|
00000050  00 03 61 67 65 44 00 06  73 61 6c 61 72 79 4c 00  |..ageD..salaryL.|
00000060  04 6e 61 6d 65 74 00 12  4c 6a 61 76 61 2f 6c 61  |.namet..Ljava/la|
00000070  6e 67 2f 53 74 72 69 6e  67 3b 78 70 00 00 00 14  |ng/String;xp....|
00000080  40 d8 6a 00 00 00 00 00  74 00 05 41 6c 69 63 65  |@.j.....t..Alice|
00000090  73 71 00 7e 00 02 00 00  00 2a 41 01 b3 40 00 00  |sq.~.....*A..@..|
000000a0  00 00 74 00 03 42 6f 62  73 71 00 7e 00 02 00 00  |..t..Bobsq.~....|
000000b0  00 25 40 e3 a4 d1 eb 85  1e b8 74 00 05 43 68 75  |.%@.......t..Chu|
000000c0  63 6b                                            |ck|
```

3.4 Textové nebo binární data

V temto semináři jsme si vždy demonstrovali práci buď s textovými, nebo binárními daty. Tato dichotomie je z části falešná. Přirozeně na textová data můžeme nahlížet taky jako na sekvenci bytů. Proto máme v Javě třídy `InputStreamReader` a `OutputStreamWriter`, které umožňují z objektu typu `InputStream` vytvořit objekt typu `Reader` a z objektu typu `OutputStream` vytvořit objekt typu `Writer`. Z toho důvodu se v Javě na řadě míst, kde se pracuje se vstupy a výstupy, objevují jen obecné abstraktní třídy `InputStream` a `OutputStream`, a potřebujeme-li pracovat s textovými daty, není problém vytvořit si potřebné třídy typu `Reader` a `Writer`.

Poznámka 7 *V tomto semináři jsem si nastínili základní práci se soubory, kterou lze použít ve spoustě běžných situací. Vedle toho standardní knihovna obsahuje balíček* `java.nio` *s třídami, které umožňují neblokující přístup k datům, což se hodí například při komunikaci po síti. Tento způsob práce s daty jde však nad rámec úvodního kurzu jazyka Java, proto se mu nebudeme podrobněji věnovat.*

Poznámka 8 *Vedle představených tříd obsahují balíčky* `java.io` *a* `java.nio` *další třídy. Mimo jiné tam lze nalézt třídy pro multiplatformní práci se soubory a souborovými systémy. Protože práce s těmito třídami odpovídá běžné práci s objekty, je na p.t. čtenářích, aby obsah těchto balíčku a tříd prozkoumali formou samostudia.*