5. Řetězce (třída String)

Pro práci s řetězci (tj. s posloupností znaků) se v jazyce Java používá třída **String**. Třída `String` slouží k ukládání konstantních řetězců, jejichž hodnota se během činnosti programu nezmění (jedná se o příklad read-only třídy) – toto však neznamená, že stejný identifikátor nemůže v průběhu programu ukazovat na různé hodnoty (různé instance). Instanci třídy `String` lze vytvořit třemi způsoby:

- explicitně následující definicí (není vhodné tento způsob vůbec používat, neboť druhá varianta je kratší, rychlejší a přehlednější):
  `String text = new String("abcd");`
- implicitně, kdy překladač automaticky doplní potřebný kód pro vytvoření instance typu String:
  `String text = "abcd";`
- implicitně, kdy se nedefinuje ani identifikátor (odpovídá konstantě primitivních datových typů):
  `"abcd";`

Pro práci s řetězci jsou definovány operátory + a += pro spojení dvou řetězců (instancí třídy `String`) do nové instance třídy `String` (do nového řetězce)[11]. Při použití operátoru += (např. `retezec1 += retezec2`) se odkaz na novou instanci přiřadí k identifikátoru uvedenému na levé straně výrazu. Spojování řetězců si ukážeme v následujícím příkladu:

```
int cislo = 10;
String retezec1 = "Výsledek je";
System.out.println(retezec1 + " " + cislo + " korun");
```

Příklad vypíše následující řádek:

```
Výsledek je 10 korun
```

Pokud se k řetězci připojuje operátorem + proměnná primitivní typu (`int`, `long` a další), překladač automaticky zajistí její převod na typ `String` (viz předchozí příklad s metodou `println()`). Pokud se k řetězci připojuje instance objektu, překladač automaticky doplní volání metody `toString()`, která převede objekt na řetězec. Pozor na posloupnost operací viz následující příklady:

```
System.out.println ("Výsledek je " + 5 + 7);
```

Tento řádek vypíše následující text:

```
Výsledek je 57
```

Pokud chceme čísla sečíst, musíme sčítání uzavřít do kulatých závorek.

```
System.out.println ("Výsledek je " + (5 + 7));
```

A tento řádek kódu vypíše toto:

```
Výsledek je 12
```

5.1. Porovnávání řetězců

Při práci s řetězci v programu je třeba si uvědomit, že řetězce nejsou primitivní datový typ, ale instance třídy `String`, tj. referenční datové typy. Pro porovnání obsahu dvou řetězců se používá metoda `equals()`. Syntaxe porovnání řetězců `retezec1` a `retezec2` je následující:

```
retezec1.equals(retezec2)
```

a výsledek je typu `boolean`.
Od verze 1.4 Java optimalizuje ukládání řetězců v paměti – cílem je ukládat stejné řetězce v paměti pouze jednou. Při následující deklaraci se v paměti vytvoří pouze jedna instance, na kterou budou odkazovat oba identifikátory.

[11] Jedná se o jediné případy přetížení (overloading) operátorů v Javě.

```
    String retezec1 = "textik";
    String retezec2 = "textik";
```

Proto i při porovnání pomocí operátoru == je výsledkem hodnota `true`, i když to na první pohled odporuje tomu, co jsme si říkali o relačních operátorech pro referenční typy (při porovnání dvou proměnných referenčního typu pomocí operátoru == je výsledkem hodnota `true`, pokud obě proměnné odkazují na stejnou instanci).
Optimalizace ukládání řetězců probíhá v době překladu a při zavádění tříd do paměti. Pokud se však řetězec vytvoří až za běhu, tak se neoptimalizuje uložení a porovnání pomocí operátoru == proto nefunguje. V následujícím kódu metoda `println()` vypíše hodnotu `false`:

```
    String retezec1 = "text";
    System.out.println( (retezec1 + "ik") == "textik"));
```

Doporučujeme vždy používat pro porovnání metodu `equals()` – je pouze nepatrně pomalejší, ale garantuje porovnávání dle obsahu za všech situací.

5.2. Další operace s řetězci

Pro převody primitivních datových typů na řetězce je ve třídě `String` definována metoda třídy `valueOf()`. Například:

```
String retezec1 = String.valueOf(3.7629);
```

uloží do objektu `retezec1` hodnotu 3.7629 jako řetězec[12]. Stejného výsledku lze dosáhnout i následujícím výrazem

```
String retezec1 = "" + 3.7629;
```

kdy se využije vlastnosti automatické konverze proměnných na řetězec. Tato druhá varianta je však pomalejší, neboť zde se vedle konverze vytvoří objekt typu `String` s prázdným řetězcem a oba řetězce se spojí do další instance třídy `String`. Metoda `String.valueOf` s parametrem typu objekt vrací na výstupu výsledek metody `toString()` příslušného objektu.
Třída `String` poskytuje velké množství dalších metod pro práci s řetězci. Mezi základní patří následující:

metoda	**popis**
int **length**()	vrací délku uloženého řetězce
boolean **equals**(String str)	porovnání obsahu dvou instancí třídy `String`
boolean **equalsIgnoreCase**(String str)	porovnání obsahu dvou instancí třídy `String` s tím, že se nerozlišují malá/velká písmena
boolean **endsWith**(String koncovka)	zjišťuje, zda uložený řetězec končí zadanou koncovkou
boolean **startsWith**(String str)	zjišťuje, zda uložený řetězec začíná uvedeným parametrem
String **toLowerCase**()	vrací instanci třídy `String` se všemi znaky převedenými na malá písmena
String **toUpperCase**()	vrací instancí třídy `String` se všemi znaky převedenými na velká písmena

[12] Při prohlížení tohoto výrazu mohou u některých čtenářů vzniknout pochybnosti, proč zde není operátor `new` pro vytvoření nového objektu. Metoda `valueOf()` sama uvnitř vytvoří novou instanci třídy `String` a na výstupu vrací pouze odkaz na tuto instanci, který se přiřadí k příslušnému identifikátoru.

metoda	popis
String **substring**(int beginIndex)	vrací instanci třídy `String` obsahující část řetězce začínající na zadaném indexu
String **substring**(int beginIndex, int endIndex)	vrací instanci třídy `String` obsahující část řetězce začínající na zadaném indexu a končící druhým indexem
int **indexOf**(String str)	vrací pozici v rámci uloženého řetězce, na které začíná řetězec uvedený jako parametr; pokud neobsahuje zadaný řetězec, vrací hodnotu –1
char[] **toCharArray**()	převede uložený řetězec do pole znaků
static String **valueOf**(Object o)	vrátí textovou reprezentaci objektu – pokud je parametrem hodnota `null`, vrátí řetězec „`null`“, jinak vrátí výsledek metody `o.toString()`

Tabulka 5.1 Přehled nejpoužívanějších metod třídy String

5.3. Speciální (escape) znaky v řetězcích

Při psaní textových řetězců i znakových konstant **ve zdrojovém kódu** programu lze používat speciální (escape) znaky uvozené zpětným lomítkem – viz tabulka 5.2.

escape znak	popis
\t	tabulátor
\n	nový řádek, používá se při výstupu do konzole či do souboru
\"	uvozovky
\’	apostrof
\\	zpětné lomítko

Tabulka 5.2 Přehled escape znaků používaných třídou String

Do řetězců lze vkládat znaky z tabulky Unicode[13] pomocí zápisu \u*xxxx*, kde na místě xxxx jsou uvedeny hexadecimální znaky. Např. zápis \u20ac odpovídá znaku € (euro).
Dále lze vkládat znaky pomocí oktálových čísel (prvních 256 znaků z tabulky Unicode \u0000 až \u00ff) – za zpětným lomítkem se uvedou oktalová čísla 0 – 377.

5.4. Formátování řetězců

Od verze 5 Java podporuje formátování řetězců podobné funkcím `sprintf()` a `printf()` v jazyce C. Ve třídě `String` je k dispozici statická metoda `format()`, která se obvykle používá pro formátování řetězců. Toto formátování se dále používá v metodě `printf()` ve třídách `PrintWriter` a `PrintStream`.
V těchto metodách se jako první parametr zadává předpis pro formátování výstupu. Poté následují parametry, které se doplní do příslušné části předpisu. Vlastní formátování zajišťuje třída `Formatter`, u které jsou podrobně popsána pravidla pro formátování.
Při volání metod používajících formátování lze jako první parametr uvést odkaz na příslušné národní prostředí (`Locale`), které ovlivňuje např. formátování data a času či znaky použité na místě desetinné čárky.
Předpis pro formátování obsahuje texty/znaky, které mají být součástí každého výstupu a formáty pro jednotlivé parametry uvozené znakem procento (%). Následují příklady použití:

[13] Od verze 5.0 Java podporuje rozšířené znaky Unicode – znaky, které jsou mimo základní rozsah 2 bytů. Podrobnosti o jejich používání najdete v dokumentaci Javy u firmy Sun.

```
String vystup = String.format("strana: %d/%d",
                              strana, pocetStran);
System.out.printf("úhly – alfa: %f6.4, beta: %f6.4, gama: %f6.4%n",
                            alfa, beta, gama);
System.out.printf("%-30s %2d %f4.2%n", prijmeni, semestr, prumer);
```

V prvním příkladu se na místo prvního řetězce *%d* doplní obsah proměnné *strana*, místo druhého řetězce *%d* se doplní obsah proměnné *pocetStran*.
Obecná specifikace formátu vypadá následovně:

```
%[argument_index$][příznaky][šířka][.přesnost]konverze
```

Z jednotlivých částí je nutný pouze úvodní znak procenta a určení konverze parametru. Číslo *argument_index* odkazuje na pořadí parametrů (počítají se od 1) a používá se při formátování data. Šířka udává minimální počet výstupních znaků pro formátování výstupu. Pokud se parametr do uvedeného počtu znaků nevejde, tak se vypisuje ve skutečné velikosti. Pokud je počet znaků delší než vlastní výstup, tak se výstup zarovná vpravo (pokud není uveden příznak pro zarovnání vlevo). Přesnost omezuje počet výstupních znaků, přesný význam závisí na konkrétní konverzi. U desetinných čísel znamená počet číslic za desetinnou tečkou (zaokrouhluje se), u formátu *%s* určuje maximální počet znaků. Pro ostatní konverze nemá přesnost smysl.
V následující tabulce jsou uvedeny možné konverze. Pokud se místo malého písmene označujícího formát použije velké písmeno, tak se výstup na konci převede na velká písmena.

konverze	typ parametru	popis
'b', 'B'	libovolný	Pokud je parametr typu *boolean* či *Boolean*, tak vloží *true* či *false*. Pokud je parametr jiného typu, výsledkem je *true*. Pokud je parametrem hodnota *null*, vloží se *false*.
'h', 'H'	libovolný	Pokud je parametrem hodnota *null*, vloží se *null*. Jinak se vloží výsledek metody *Integer.toHexString(arg.hashCode())*.
's', 'S'	libovolný	Pokud je parametrem hodnota *null*, vloží se *null*. Pokud parametr implementuje rozhraní *Formattable*, vloží se výsledek metody *arg.formatTo()*. Jinak se vloží výsledek metody *arg.toString()*.
'c', 'C'	znak	Vloží se znak v Unicode.
'd'	celé číslo	Vloží se celé číslo v dekadickém tvaru.
'o'	celé číslo	Celé číslo se vloží v oktalové notaci.
'x', 'X'	celé číslo	Vloží se celé číslo v hexadecimální notaci.
'e', 'E'	desetinné číslo	Vloží se desetinné místo ve vědecké notaci.
'f'	desetinné číslo	Vloží se desetinné číslo.
'g', 'G'	desetinné číslo	V závislosti na velikosti čísla a požadované přesnosti se vloží desetinné číslo v normální či ve vědecké notaci.
'a', 'A'	desetinné číslo	Vloží se číslo v hexadecimálním tvaru.
't', 'T'	datum/čas	Prefix pro konverzi data a času – následuje upřesňující znak, bližší popis viz dokumentace třídy *Formatter*.
'%'		Vloží se znak '%'.
'n'		Vloží se správný oddělovač řádků pro konkrétní platformu.

Tabulka 5.3 Přehled možných konverzí používaných v metodách format() a printf()

Při specifikaci formátu lze použít příznaky uvedené v následující tabulce.

příznak	**význam**
'-'	pokud bude výstup kratší, než uvedená délka, zarovná se doleva
'0'	u čísel budou uvedeny úvodní nuly
'+'	u čísel bude vždy uvedeno znaménko
','	u čísel budou použity oddělovače řádů dle národního prostředí (Locale)

Tabulka 5.4 Přehled specifikací formátu

Při nesprávně zapsaném formátu či při neodpovídajícím datovém typu vznikají výjimky, které by měl programátor odchytávat.

5.5. Regulární výrazy

Regulární výrazy umožňují zjistit, zda zadaný řetězec či jeho část odpovídá zadanému vzoru. Dále je možné nahradit část řetězce jiným a dle vzoru rozdělit řetězec na jednotlivé části. Regulární výrazy jsou součástí Javy od verze 1.4, kdy byl doplněn balíček `java.util.regex` a třída `String` byla rozšířena o několik metod pracujících s regulárními výrazy – viz následující tabulka.

hlavička metody třídy String	**popis metody**
public boolean **matches**(String regex)	zjišťuje, zda celý uložený řetězec v instanci odpovídá celému vzoru
public String **replaceFirst**(String regex, String replacement)	metoda nahradí první výskyt odpovídající zadanému vzoru druhým řetězcem
public String **replaceAll**(String regex, String replacement)	metoda nahradí všechny výskyty odpovídající zadanému vzoru druhým řetězcem
public String[] **split**(String regex)	metoda rozdělí řetězec na základě zadaného vzoru na jednotlivé části a vrátí jako pole řetězců
public String[] **split**(String regex, int limit)	metoda rozdělí řetězec na základě zadaného vzoru na jednotlivé části a vrátí jako pole řetězců; proti předchozí variantě je omezen maximální počet částí, na které se řetězec rozdělí

Tabulka 5.5 Metody třídy String, které pracují s regulárními výrazy

V případě vícenásobného použití stejného regulárního výrazu (např. v cyklu) je efektivnější použít třídy `Pattern` a `Matcher`, které obsahují i další metody rozšiřující možnosti regulárních výrazů. Vzory jsou základní prvky pro vytváření regulárních výrazů – vzorem se popisuje, jak má přípustný řetězec vypadat. Nejjednodušším vzorem je řetězec, který chceme vyhledat. Následuje přehled základních pravidel pro vytváření vzorů, další možnosti jsou uvedeny v dokumentaci.

speciální symbol v regulárním výrazu	**popis**
. (tečka)	libovolný znak
+ (plus)	minimálně jedno opakování předchozího znaku/výrazu
* (hvězdička)	0 až nekonečno opakování předchozího znaku/výrazu
[ ] (hranaté závorky)	množina, tj. libovolný znak z množiny znaků uvnitř závorek

speciální symbol v regulárním výrazu	**popis**
\ (zpětné lomítko)	potlačuje speciální význam následujícího znaku či uvozuje speciální skupinu znaků (pokud je za lomítkem písmeno či číslice)
\\	zpětné lomítko
\b	hranice slova
\d	číslice
\s	„netisknutelné“ znaky – mezera, tabulátor, konec řádku
X|Y	buď znak X nebo znak Y
(X)	označení skupiny

Tabulka 5.6 Přehled nejčastěji používaných speciálních symbolů pro regulární výrazy

Příklady

V příkladech jsou vzory zadány jako řetězce ve zdrojovém kódu programu – v této poměrně obvyklé situaci se uplatňuje zpětné lomítko též jako speciální znak pro řetězcovou konstantu (viz tabulka 5.7) a proto je potřeba pro vložení zpětného lomítka vzoru uvést dvě zpětná lomítka.

příklad regulárního výrazu a popis
`retezec.matches("ahoj");` `True`, pokud řetězec obsahuje pouze "ahoj" – vhodnější je použít metodu `equals()`, neboť je rychlejší.
`retezec.matches(".*\\bahoj\\b.*");` `True`, pokud řetězec slovo ahoj (jsou zde uvedeny hranice slova).
`retezec.matches("\\s*");` `True`, pokud řetězec je „prázdný“, tj. má nulovou délku, či obsahuje pouze mezery nebo tabulátory.
`retezec.matches(".*\\s+");` `True`, pokud na konci řetězce je minimálně jedna mezera a tabulátor.
`retezec.matches("[a-wyz].*");` `True`, pokud řetězec začíná písmenem `a` až `z` s výjimkou písmene `x`.
`retezec.matches(".*\\\\.*");` Zjišťuje se, zda řetězec obsahuje aspoň jedno zpětné lomítko.
`retezec.matches(".*\\s(try)|(catch)\\s.*");` `True`, pokud řetězec obsahuje slovo `try` či `catch` (nebo oboje) oddělené od okolí mezerou či tabulátorem.
`radek = radek.replaceFirst("^\\s+","");` Zrušení mezer na začátku řádku.
`radek = radek.replaceFirst("\\s+$","");` Zrušení mezer na konci řádku.
`radek = radek.replaceAll("\\s+","");` Zrušení všech mezer v řádku.

příklad regulárního výrazu a popis

`String radek = "xabcd01:1:IN:Student Testovací";`
`String [] polozky = radek.split(":");`
Řádek se rozdělí na jednotlivé části, oddělovačem je znak dvojtečky. Vznikne pole se čtyřmi prvky.

`String radek = "xabcd01  1  IN  Student Testovací";`
`String [] polozky = radek.split("\\s+",4);`
Řádek se rozdělí na jednotlivé části s tím, že oddělovačem je posloupnost mezer a tabulátorů. Výstup je omezen na maximálně čtyři položky, které i v našem případě vzniknou.

Tabulka 5.7 Příklady použití regulárních výrazů

V následující ukázce se zpracovávají řádky z textového souboru o evidovaných počítačích. Řádek vstupního souboru vypadá následovně (IP adresa, jméno, umístění):

```
146.102.33.1:s019h01:019sb
```

Program ze vstupního souboru vybere řádky z místnosti 019sb (na začátku se provádějí kontroly vstupního řádku), jeho rozdělení na části pomocí metody `split()` a nahrazení části textu jiným (v našem případě se nahradí prázdným řetězcem, tj. v podstatě se začátek řetězce zruší). Pro výše uvedený ukázkový řádek bude výsledkem hodnota "33.1".

```
String radek = vstup.readLine();
while (radek != null) {
     if (radek.matches("#.*")) {   // komentar
          next;
     }
     if (radek.matches("[ \t]*")) { // prázdný řádek ci
                                    //pouze mezery a tabulátory
          next;
     }
     if (radek.matches("[0-9\\.]+:.+:.+")) {    // řádek je O.K.
          if (radek.matches(".*:019sb")) { //z místnosti 019sb?
               String [] prvky = radek.split(":");     // rozdělení řádku
                                                       // na části dle :
               String ipAdresa=prvky[0].replaceFirst("146.102.","");
                                                       // cast IP adresy
               ...
          }
     }
     radek = vstup.readLine();
}
```

Čtení ze souboru je podrobně popsáno v kapitole věnované vstupně/výstupním operacím.

5.6. Třídy StringBuffer a StringBuilder

Ke třídě `String` existuje alternativní třída `StringBuffer`, která na rozdíl od třídy `String` není read-only – při přidávání/rušení řetězců se nevytváří nové instance, čímž lze dosáhnout větší efektivity programu. Od Javy 5.0 existuje ještě třída `StringBuilder`, která má stejné metody jako `StringBuffer`. Hlavní rozdíl je v tom, že třídy `StringBuilder` není synchronizovaná – tj. nelze ji používat ve vláknech. Pokud je použita v jednom vlákně, tak je rychlejší než `StringBuffer`.

Použití třídy `StringBuffer` má význam v situaci, kdy potřebujeme spojit více řetězců, vkládat řetězce dovnitř existujícího řetězce či střídavě vkládat/rušit části řetězce. V následující tabulce je přehled vybraných metod tříd `StringBuffer` a `StringBuilder`.

metoda	**popis**
int **length**()	vrací délku uloženého řetězce
StringBuffer **append**(String str)	na konec uloženého řetězce vloží obsah instance třídy `String`
StringBuffer **append**(Object o)	k uloženému řetězci přidá textovou reprezentaci objektu (výsledek operace `String.valueOf(o)`)
StringBuffer **insert**(int pozice, String str)	na zadanou pozici vloží řetězec
StringBuffer **insert**(int pozice, Object o)	na zadanou pozici vloží textovou reprezentaci objektu (výsledek operace `String.valueOf(o)`)
StringBuffer **delete**(int zacatek, int konec)	smaže znaky z pozice `zacatek` po pozici `konec`
String **toString**()	vrátí obsah jako instanci třídy `String`
String **substring**(int beginIndex)	vrací instanci třídy `String` obsahující část řetězce začínající na zadaném indexu
String **substring**(int beginIndex, int endIndex)	vrací instanci třídy `String` obsahující část řetězce začínající na zadaném indexu a končící druhým indexem

Tabulka 5.8 Přehled vybraných metod třídy StringBuffer, třída StringBuilder má stejné metody

V následujícím příkladu se prochází v cyklu mapa s druhy zvířat a jejich počty. Pro každý druh se vytvoří řetězec s popisem.

```
     for (String klic: mapa.keySet()) {
          StringBuffer sb = new StringBuffer();
          sb.append("zvire ");
          sb.append(klic);
          sb.append(", pocet kusu ");
          sb.append(mapa.get(klic)).toString();
          String radek = sb.toString();
     }
```

Tento kód je asi o třetinu rychlejší (ale méně přehledný), než následující kód pracující s instancemi třídy `String`.

```
     for (String klic: mapa.keySet()) {
          String radek = "zvire "+klic+", pocet kusu "+mapa.get(klic);
     }
```

Třídy `StringBuffer` a `StringBuilder` jsou navrženy tak, že po zavolání metod `append()`, `insert()` a `delete()` vrací odkaz na sebe sama. To umožňuje přehlednější zápis řetězením těchto příkazů, viz následující kód. Tento kód je nejrychlejší – používá se instance třídy `StringBuilder`, využívá se pouze jedna instance této třídy a na začátku je nastavena předpokládaná velikost řetězce.

```
StringBuilder sb = new StringBuilder(60);
for (String klic: mapa.keySet()) {
     sb.delete(0, sb.length());
     sb.append("zvire ").append(klic);
     sb.append(", pocet kusu ").append(mapa.get(klic));
     String radek = sb.toString();
}
```