1 Jak vzniká program

Na začátku potřebujeme nějaký editor čistého textu[1], kompiler[2] a Java Runtime Environment (JRE). JRE je většinou již obsažen v systému, místo editoru a kompileru je možné použít nějaké IDE, třeba Netbeans. Cesta k výslednému programu vypadá následovně:

1. Pomocí editoru napíšeme zdrojový kód.
2. Ten kompiler přeloží do bytekódu, který je nezávislý na použité platformě.
3. Při spuštění programu JRE přeloží bytekód do strojového kódu[3], kterému počítač rozumí, ale je platformově závislý[4].

2 První program

2.1 Instalace potřebných programů

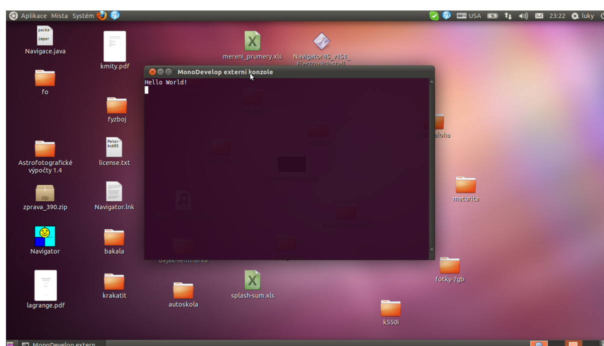
JRE již pravděpodobně máte nainstalováno, je ale většinou nutné doinstalovat JDK. Verzi pro Windows najdeme na stránkách Oracle, verzi pro Linux tamtéž, nebo instalujeme z repozitářů. Svůj oblíbený textový editor předpokládám každý nainstalovaný má ;-)

2.2 „Hello world!“

Nyní skutečně začneme programovat. Otevřete svůj oblíbený editor a do něj napište následující zdrojový kód:

```
public class AhojSvete
{
  public static void main(String[] args)
  {
    System.out.println("Ahoj světe!");
  }
}
```

Kód uložte do souboru s názvem `AhojSvete.java`, **nijak jinak se nesmí jmenovat**. Zapněte terminál (příkazovou řádku, Powershell, ...) a pomocí příkazu `cd` se dostaňte do adresáře s uloženým souborem. Zadejte příkaz `javac AhojSvete.java`, čímž jste provedli překlad a vytvořili soubor `AhojSvete.class`, který obsahuje náš program v bytekódu. Nyní program spustíte pomocí příkazu `java AhojSvete` (bez přípony). Mělo by se vám objevit něco podobného.



Obrázek 1:

[1] Například Poznámkový blok nebo PSPad, rozhodně ne Word!
[2] Přesněji celý Java Development Kit (JDK).
[3] Tomu se říká Just-in-time kompilace.
[4] Tzn. každá kombinace OS a procesoru má vlastní strojový kód.

Popíšeme si, co program dělá. Klíčová slova `public class` necháme na později, zatím budeme vědět pouze to, že tam musí být. `public static void main (String[] args)` je definice hlavní metody. Ta pokaždé musí vypadat přesně takto, včetně názvu `main`. V těle této metody (mezi závorkami { a }) začíná vykonávání jednotlivých příkazů programu.

Počítač tedy narazí na příkaz `System.out.println("Ahoj světe");`, který neříká nic jiného, než „vypiš na standardní výstup Ahoj světe! a odřádkuj“.

3 Základní stavební kameny jazyka Java

3.1 Příkazy

Programy nejsou nic jiného, než sekvence příkazů. Ty samozřejmě mohou být různě složité, od „vezmi do ruky lžíci“ až po „připrav večeři o deseti chodech“, přičemž často se ty složité skládají z těch jednoduchých. Platí jednoduché pravidlo, že **každý příkaz, který jehož součástí není tělo (závorky { a }), musí být ukončen středníkem!** Je to velmi častá chyba, kterou občas z nepozornosti udělají i zkušení programátoři.
Štábní kultura:

Je dobrým zvykem psát jeden příkaz vždy na jednu řádku.

3.2 Bloky

Blok je množina příkazů ohraničená složenými závorkami { a }. Za blokem se nemusí psát středník, za určitých okolností dokonce nesmí. Blok většinou představuje tělo nějakého složitějšího příkazu, např.: tělo metody `main()`.
Štábní kultura:

Příkazy vnořené do bloku se odsazují o jeden tabulátor oproti úrovni „mateřského“ příkazu. Složené závorky ohraničující blok jsou na samostatné řádce na stejné úrovni, jako příkaz, ke kterému patří.

3.3 Komentáře

Komentáře jsou značky sloužící pouze programátorovi ke zpřehlednění kódu. Při překladu jsou automaticky odstraněny. Komentář může být jednořádkový nebo víceřádkový

3.3.1 Jednořádkový komentář

Začíná dvěma lomítky a pokračuje až do konce řádky.

```
//toto je jednořádkový komentář
```

3.3.2 Víceřádkový komentář

Začíná lomítkem následovaným hvězdičkou a končí hvězdičkou následovanou lomítkem.

```
/*
toto je
víceřádkový
komentář
*/
```

V Javě nejsou povoleny tzv. vnořené komentáře.

```
/*
součást komentáře
  /* stále součást komentáře */
tato část už není součást komentáře
*/
```

4 Primitivní datové typy

4.1 Proměnné

Proměnné nám slouží k ukládání mezivýsledků práce programu. Dělíme je na několik podtypů.

Celočíselný typ

Jak z názvu vyplývá, do proměnných tohoto typu ukládáme celá čísla. Podle délky čísla používáme různé proměnné.

klíčové slovo	délka v paměti	rozsah hodnot
byte	1 byte	-128 až 127
short	2 byte	-32768 až 32767
int	4 byte	-2147483648 až 2147483647
long	8 byte	-9223372036854775808 až 9223372036854775807

Nejpoužívanější z těchto typů je typ integer (`int`), i když správně bychom měli používat typ, který je na naše použití nejen dostatečně dlouhý, ale také není dlouhý až zbytečně moc, abychom neplýtvali přidělenou pamětí.
Pozn.: V Javě neexistují tzv. neznaménkové typy.

Reálný typ

Tento typ bývá často označován jako typ s plovoucí desetinnou čárkou (floating-point). To je způsobeno jeho reprezentací v paměti, kde je ukládán jako $X,XXX \cdot\ 10^n$, je tedy zafixován počet platných číslic a hýbe se exponent. Existují dva typy lišící se přesností.

klíčové slovo	délka v paměti	rozsah hodnot
float	4 byte	$\pm 1,5 \cdot 10^{-45}$ až $\pm 3,4 \cdot 10^{38}$ s 8 platnými místy
double	8 byte	$\pm 5,0 \cdot 10^{-324}$ až $\pm 1,7 \cdot 10^{308}$ s 16 nebo 17 platnými místy

Znakový typ

Typ character se používá k ukládání jednoho znaku. Používá se například při sekvenčním čtení znak po znaku.

klíčové slovo	délka v paměti	rozsah hodnot
char	2 byte	libovolný Unicode znak

Logický typ

Typ boolean se používá při vyhodnocování logických operací.

klíčové slovo	délka v paměti	rozsah hodnot
boolean	1 byte	`true` (pravda) nebo `false` (nepravda)

4.1.1 Použití proměnných

Každou proměnnou musíme před jejím prvním použitím definovat. Definice znamená, že určíme její jednoznačné jméno a typ a počítač v závislosti na potřebném prostoru v paměti (podle použitého datového typu) jí potřebnou paměť přiřadí.

Pojmenování proměnných: Jméno proměnné nesmí obsahovat speciální ani bílé znaky s výjimkou podtržítka. Zároveň nesmí být shodné s žádným z klíčových slov jazyka Java. Může obsahovat číslice, malá a velká písmena bez diakritiky a podtržítko, číslice ale nesmí být na začátku jména. Pozor na to, že Java rozlišuje velká a malá písmena. **Tento odstavec platí o jménech v jazyce Java obecně!**
Štábní kultura:
Je doporučeno začínat název proměnné malým písmenem. Pokud se jméno skládá z více slov, každé slovo, kromě prvního, by mělo začínat velkým písmenem. Podle názvu by mělo být poznat, k čemu proměnná slouží.

```java
class Promenne
{
  public static void main (String[] args)
  {
    int promenna1; //definice promenne typu integer s nazvem promenna1
    promenna1 = 10; //prirazeni hodnoty promenne promenna1
    double nejakaJinaPromenna = 2.0; //definice promenne typu double a zaroven prirazeni jeji
        hodnoty
    System.out.println(promenna1); //vypise 10 a odradkuje
    System.out.println(nejakaJinaPromenna); //vypise 2.00000 a odradkuje
  }
}
```

Hodnotu proměnné lze za běhu programu měnit (dalším přiřazením), přičemž lze i přiřazovat hodnotu jedné proměnné do jiné, pokud jsou tyto proměnné stejného typu. Pokud nejsou, lze to provést s jistými omezeními také (více v kapitole Přetypování).

Pokud přiřazujeme z jedné proměnné do druhé, a poté znovu změníme hodnotu první proměnné, hodnota druhé proměnné se nám tím nijak nezmění.

```java
int a, b; //definice promennych a, b
a = 0; //a = 0, hodnota b je nedefinovana
b = a; //a = 0, b = 0
a = 2; //a = 2, b = 0
```

Proměnná je viditelná pouze v bloku, ve kterém je definována a v jeho podblocích.

```java
{
int a;
a=2;
}
a=3; //chyba
```

4.2 Konstanty

Konstanty jsou z pohledu programátora proměnné, jejichž hodnotu nemůže po prvním přiřazení měnit. Z pohledu počítače je ale rozdíl mezi proměnnými a konstantami daleko propastnější, protože konstanta, narozdíl od proměnné, nemá vyhrazené místo v paměti, ale při překladu se všechny její výskyty v kódu nahradí její hodnotou. Výhody jsou zřejmé:

- Nepřepíšete se a nezměníte náhodou hodnotu něčeho, co byste měnit neměli.
- Pokud chcete změnit parametry programu, stačí přepsat hodnotu konstanty jen jednou v kódu.
- Víte, že v půlce řádky 1052 značí číslo 120 šířku zobrazeného okna.
- Nezabíráte zbytečně místo v paměti.

Dokud píšte prográmky o deseti nebo dvaceti řádcích, mohou vám tyto argumenty připadat směšné. Pokud ale píšete program o tisíci řádcích, pak jste rádi hlavně za první tři.

Konstanty mají stejné typy jako proměnné, jejich zápis se liší modifikátorem `final`, který při definici píšete před typ konstanty. Zároveň musíte konstantě při definici přiřadit její hodnotu.

```java
final int PrvniKonstanta = 10;
final String VarovnaHlaska = "Program vykonal neplatnou operaci a musel být ukončen.";
int promenna = PrvniKonstanta; //hodnotu honstanty samozrejme muzete predat nejake promenne
System.out.println(VarovnaHlaska); //nebo predat jako parametr metody.
```

Nepojmenované konstanty

Za konstanty jsou také považovány

- řetězcové konstanty (uzavřené do uvozovek) – "Ahoj světe"
- literály (znakové konstanty uzavřené do apostrofů) – 'a'
- čísla
- klíčová slova `true` a `false`

Číselné zápisy

Číselnou konstantu můžeme zapsat nekolika způsoby:

- **Dekadický zápis** – klasické celé nebo desetinné číslo, používá se desetinná tečka, např.: 10, 3.141592, ...
- **Zápis s exponentem** – např.: 6.43E-12 nebo 6.43e-12

Pokud je ale v programu používáme vícekrát ve stejném významu, doporučuje se je přiřadit do pojmenované konstanty a nadále používat v programu tu. V opačném případě budete osočeni z toho, že používáte tzv. „magická čísla“, co to obnáší, poznáte, až budete chtít změnit výšku okna a budete zkoumat, ve kterém z těch 500 výskytů čísla 768 ve vašem zdrojovém kódu je toto číslo použito jako výška okna v pixelech a ve kterém je to naprosto nesouvisející konstanta mající jen čirou náhodou stejnou hodnotu.

4.3 Přetypování

Přetypování je operace, která se provádí, pokud chceme přiřadit hodnotu proměnné, či konstanty jednoho typu, do proměnné druhého typu. Existuje implicitní a explicitní přetypování.

4.3.1 Implicitní přetypování

Pokud dosazujeme do nějakého datového typu hodnotu jiného datového typu a existuje nějaký rozumný převod (např. z `int` na `float`, ale už ne z `int` na `byte`), tak se typová konverze provede automaticky (implicitně).

```
int a = 10;
double b;
b = a; //zde se provede implicitní typová konverze z int na double
```

Implicitní typová konverze ale proběhne pouze tehdy, pokud díky ní neztratíme žádnou informaci, tedy pokud převádíme na širší datový typ.

```
double a = 3.141592;
int b;
b = a; //zde překladač ohlásí chybu (do int neuložíme desetinnou část).
```

4.3.2 Explicitní přetypování

Pokud potřebujeme přeci jen provést typovou konverzi, i když implicitní konverze není možná, či potřebujeme konverzi vynutit, použijeme explicitní přetypování. Samozřejmě zase můžeme převádět jen „podobné“ datové typy.

Explicitní přetypování se provádí pomocí operátoru přetypování `(datový typ) proměnná`.

```
double a = 3.141592;
int b;
b = (int) a; //b = 3
```

Je jedno velmi časté použití explicitního přetypování, a to u dělení. Operátor dělení totiž vrátí výsledek takového typu, jako je nejširší datový typ, který do operace vstupuje. Pokud tedy dělíme dvě celá čísla, na výstupu zase dostaneme výsledek ořízlý na celé číslo. Pokud ale při dělení jeden z operandů přetypujeme na desetinný datový typ, dostaneme na výstupu desetinné číslo.

```
int a = 5;
int b = 10;
double c = a / b //c bude v tomto případě 0 (celá část 0.5)

c = ((float) a) / b //c = 0.5
```

5 Operátory

Operátory, jako je operátor sčítání, jistě znáte z matematiky, v programování je jejich použití podobné. Operátory dělíme na tři podtypy, v závislosti na tom, nad kolika operandy pracují:

- Unární operátory – jeden operand
- Binární operátory – dva operandy
- Ternární operátor – tři operandy

5.1 Aritmetické operátory

název	**typ**	**operátor**	**popis**	**příklad**
kladná hodnota	unární	$+$	zdůraznění, že je dané číslo kladné	$+10$
záporná hodnota	unární	$-$	opačné číslo k operandu	-10 nebo $-a$
předcházející dekrement	unární	$--$	odečte od operandu jedna **před provedením příkazu**	$--a$
předcházející inkrement	unární	$++$	přičte k operandu jedna **před provedením příkazu**	$++a$
následný dekrement	unární	$--$	odečte od operandu jedna **po provedení příkazu**	$a++$
následný inkrement	unární	$++$	přičte k operandu jedna **po provedení příkazu**	$a--$
sčítání	binární	$+$	sečte dvě čísla	$a+b$
odčítání	binární	$-$	odečte dvě čísla	$a-b$
násobení	binární	$*$	vynásobí dvě čísla	$a*b$
dělení	binární	$/$	vydělí dvě čísla	a/b
zbytek po dělení (modulo)	binární	$\%$	vrátí zbytek po celočíselném dělení	$a\%b$

5.2 Přiřazovací operátory

Používají se pro přiřazení hodnoty proměnné.

název	**typ**	**operátor**	**popis**	**příklad**
přiřazení	binární	$=$	přiřadí hodnotu proměnné	$a=10$
přičtení	binární	$+=$	$a=a+b$	$a+=b$
odečtení	binární	$-=$	$a=a-b$	$a-=b$
vynásobení	binární	$*=$	$a=a*b$	$a*=b$
vydělení	binární	$/=$	$a=a/b$	$a/=b$
přiřazení modula	binární	$\%=$	$a=a\%b$	$a\%=b$

5.3 Relační a logické operátory

Tyto operátory používáme k vyhodnocování logických výrazů (většinou porovnávání proměnných), jejich výstupem je buď `true` nebo `false`.

název	typ	operátor	popis	příklad
menší než	binární	$<$	zjistí, jestli je a menší než b	$a < b$
větší než	binární	$>$	zjistí, jestli je a větší než b	$a > b$
menší nebo rovno	binární	$<=$	zjistí, jestli je a menší než b nebo rovno s b	$a <= b$
větší nebo rovno	binární	$>=$	zjistí, jestli je a větší než b nebo rovno s b	$a >= b$
je rovno	binární	$==$	zjistí, jestli je a rovno s b	$a == b$
není rovno	binární	$!=$	zjistí, jestli je a různé od b	$a! = b$
a zároveň	binární	&&	konjunkce logických výrazů	$(a == b)\&\&(a! = b)$
nebo	binární	$\vert\vert$	disjunkce logických výrazů	$(a == b)\vert\vert(a! = b)$
negace	unární	!	negace logického výrazu	$!(a < b)$

5.4 Bitové operátory

název	typ	operátor	popis	příklad
bitový posun doprava	binární	$>>$	posune bity v proměnné o n míst doprava	$a >> 2$
neznaménkový bitový posun doprava	binární	$>>>$	posune bity v proměnné o n míst doprava	$a >>> 2$
bitový posun doleva	binární	$<<$	posune bity v proměnné o n míst doprava	$a << 3$
bitové OR	binární	$\vert$	operace OR po bitech	$a \vert b$
bitové AND	binární	&	operace AND po bitech	$a\&b$
bitové XOR	binární	^	operace exkluzivní OR po bitech	a^b
bitová negace	unární	~	negace po bitech (kde je 1 bude 0 a obráceně)	~a

5.5 Ostatní operátory

název	typ	operátor	popis	příklad
přetypování	unární	(`typ`)	změní datový typ	$(int)a$
ověření kompatibility	binární	`instanceof`	zjistí kompatibilitu s objektem	a `instanceof String`
ternární operátor	ternární	`?:`	pokud je a pravda, je výsledkem b, jinak c	$d = a?b : c$

Štábní kultura:

Pokud si nejsme jistí pořadím, v jakém budou vykonávány operace ve výrazu složeném z více operátorů, vždy závorkujeme. Lze použít neomezený počet do sebe vnořených kulatých závorek.

6 Podmíněné příkazy

6.1 Konstrukce if-else

Tato konstrukce slouží k testování podmínek. Je na první pohled velmi podobná ternárnímu operátoru. Její syntaxe je

```
if(podmínka){
//blok příkazů, pokud podmínka == true
}
else{
//blok příkazů, pokud podmínka == false
}
```

kde `podmínka` je libovolný výraz typu `boolean`. Pokud je část `else` prázdná, můžeme využít pouze části `if`. Případně můžeme podmínky řetězit.

```
if(podmínka1){
//blok příkazů, pokud podmínka1 == true
}
else if(podmínka2){
//blok příkazů, pokud podmínka1 == false && podmínka2 == true
}
else{
```

```
//blok příkazů, pokud žádná z podmínek není splněna.
}
```

Rozdíl oproti ternárnímu operátoru je ten, že ternární operátor je výraz, vrací tedy hodnotu, kterou můžeme uložit do proměnné. Oproti tomu příkaz `if` vykonává jednotlivé příkazy v jeho těle. Následující příklad je situace, kdy můžeme použít obojí, ale ternární operátor má kratší zápis (a jeho vykonání je rychlejší). Pokud si ale nebudeme ternární operátor pamatovat, nic se neděje, lze vždy nahradit konstrukcí `if-else`. Obrácené pravidlo ale neplatí.

```
int a = 1;
int b = 2;
int c;
// test na dělení nulou
if(b != 0){
  c = a / b;
}
else{
  c = 0; //řekněme, že nám to tak vyhovuje
}

//to samé ternárním operátorem
c = (b != 0)? (a / b) : 0;
```

6.2 Konstrukce switch

Tato konstrukce se používá, pokud chceme vykonat určité příkazy v závislosti na tom, čemu se rovná hodnota testované proměnné. Ta může být pouze celočíselným nebo výčtovým datovým typem nebo typem `String`. Syntaxe je

```
switch(promenna){
case hodnota1:
  //blok prikazu, ktere se maji provest, pokud promenna == hodnota1
  break;
case hodnota2:
  //blok prikazu, ktere se maji provest, pokud promenna == hodnota2
  break;
default:
  //blok prikazu, ktere se maji provest, pokud promenna se nerovna zadne z hodnot
  break;
}
```

Za každým blokem příkazů musí být příkaz `break`, který zajistí, že se příkaz `switch` ukončí, a tudíž se nevykoná větev `default` v případě, že se podmínka splní u nějaké předchozí větve. Větev nemusí obsahovat `break`, pokud je daná větev prázdná, tedy neobsahuje žádné příkazy.

```
switch(promenna){
case hodnota1:
case hodnota2:
  //blok prikazu, ktere se maji provest, pokud promenna je rovna hodnota1 nebo hodnota2
  break;
default:
  //blok prikazu, ktere se maji provest, pokud promenna se nerovna zadne z hodnot
  break;
}
```

7 Cykly

Cykly se používají, pokud chceme nějakou část kódu opakovat několikrát za sebou. Existují tři typy cyklů, které se liší použitím.

7.1 Cyklus for

Cyklus `for` nám slouží, pokud dopředu víme, kolikrát cyklus proběhne. Syntaxe je následující.

```
for(inicializace; podmínka; inkrement){
//blok příkazů v cyklu
}
```

Prvně se vykoná příkaz, který je na místě označen jako inicializace. Poté se ověří podmínka a pokud platí, vykoná se blok příkazů a následně příkaz v místě označeném jako inkrement. Poté se znovu ověří podmínka, vykonají příkazy, vykoná inkrement, a tak stále dokola, dokud podmínka platí.

Běžně se cyklus `for` řídí pomocí tzv. iterační proměnné. To je proměnná, kterou běžně definujeme v místě inicializace, podmínka je typu iterPromenna < PocetPruchodu a inkrement je nejčastěji iterPromenna++. Následující příklad desetkrát vypíše Ahoj světe!.

```
for(int i = 0; i < 10; i++){
  System.out.println("Ahoj světe!");
}
```

7.2 Cyklus while

Tento cyklus používáme, pokud dopředu nevíme, kolikrát nám cyklus proběhne. Typicky je to v případě, když podmínka pro průběh cyklu závisí na příkazech vykonávaných uvnitř cyklu. Syntaxe je

```
while(podmínka){
//blok příkazů v cyklu
}
```

Cyklus jednoduše probíhá, dokud je podmínka splněna. Příklad, který bude číst vstup od uživatele, dokud uživatel nenapíše "q".

```
char s;
while(s != 'q'){
  s = (char) System.in.read();
}
```

Jen pro doplnění, cyklus `for` lze napsat pomocí cyklu `while` jako

```
inicializace;
while(podmínka){
  //blok příkazů v cyklu
  inkrement;
}
```

7.3 Cyklus do-while

Cyklus `do-while` se používá v případech, kdy bychom použili cyklus `while`, který má alespoň jednou proběhnout, i když podmínka není splněna už na začátku. Syntaxe je

```
do{
  //blok příkazů v cyklu
}while (podmínka);
```

Za podmínkou musí být středník. Rozdíl oproti cyklu `while` je ten, že cyklus `while` nemusí proběhnout ani jednou, protože podmínka se testuje před průchodem cyklu, zatímco u cyklu `do-while` se podmínka testuje až po proběhnutí, tento cyklus tedy proběhne minimálně jednou.

Cyklus `do-while` lze napsat pomocí cyklu `while` takto

```
//blok příkazů v cyklu
while (podmínka){
  //blok příkazů v cyklu
}
```

Pokud je v cyklu jeden příkaz, je to více méně jedno. Pokud tam ale bude příkazů deset nebo dvacet, pak je jistě přehlednější napsat cyklus `do-while`.

7.4 Příkazy break a continue

Příkaz `break` funguje stejně, jako u příkazu `switch`, tedy ukončí běh cyklu a běh programu pokračuje za cyklem.

Příkaz `continue` ukončí pouze aktuální iteraci cyklu a pokračuje další.

8 Pole

Pole je uspořádaná n-tice prvků daného datového typu. Pole se deklaruje takto

```
int[] nasePole = new int[20];
```

Tímto jsme vytvořili pole typu `int` o dvaceti prvcích. K jednotlivým prvkům přistupujeme pomocí indexů, které číslujeme od nuly. U pole o N prvcích má poslední prvek index N-1. K prvkům se přistupuje tak, že do hranatých závorek napíšeme index prvku, ke kterému chceme přistoupit.

```
nasePole[0] = 4; //první prvek pole nasePole má hodnotu 4
nasePole[1] = 8;
System.out.println(nasePole[0]);
```

Pokud chceme pole naplnit hned při vytvoření, lze to provést pomocí složených závorek.

```
char[] poleZnaku = new char[] {'a', 'c', 'x'}; //pole znaku o trech prvcich
//nebo
char[] poleZnaku2 = {'a', 'b', 'c'};
```

Délku pole můžeme zjistit pomocí vlastnosti `length`.

```
float[] pole3 = new float[50];
System.out.println(pole3.length); //vypise 50
```

8.1 Vícerozměrná pole

Pole, se kterými jsme zatím pracovali, byla tzv. jednorozměrná. Pokud bychom chtěli např. implementovat v našem programu matice, potřebovali bychom dvourozměrné pole. Práce s nimi je analogická.

```
int[][] nasePole = new int[20][30]; //pole o 20 radcich a 30 sloupcich
nasePole[0][3] = 10; //prvek na prvnim radku ve ctvrtem sloupci ma hodnotu 10
int[][] pole2 = {{1,2,3}, {4,5,6}}; //naplneni pri deklaraci
```

Protože dvourozměrné pole je vlastně pole polí, vlastností `length` získáme počet řádků. Pokud chceme získat počet sloupců, musíme se dotázat na délku některého prvku vnějšího pole.

```
int[][] nasePole = new int[20][30]; //pole o 20 radcich a 30 sloupcich
Console.WriteLine(nasePole.length); //vypise 20
Console.WriteLine(nasePole[0].length); //vypise 30
```

Pole, která jsme vytvářeli, byla pravoúhlá, každý řádek měl stejný počet sloupců. Protože se ale jedná o pole polí, můžeme vytvořit i pole zubatá.

```
int[][] pole3 = new int[2][]; //dvourozmerne pole o dvou radcich
pole3[0] = new int[5]; //prvni radek ma 5 sloupcu
pole3[1] = new int[10]; //druhy radek ma 10 sloupcu
pole3[0][4] = 8; //prvek na prvnim radku v patem sloupci ma hodnotu 8
```

9 Statické metody

Zatím jsme vše psali do metody `main`. Pokud ovšem budeme chtít provádět stejnou rutinu v kódu vícekrát, je dobré si na to napsat metodu a tu už pak jen volat. Tento přístup jsme již mnohokrát použili, například při volání metody `println`.

Zatím se budeme zabývat tvorbou pouze statických metod. K jejich volání totiž nepotřebujeme vytvářet instanci třídy, do níž jsou začleněny.

9.1 Definice metody

Statickou metodu definujeme takto.

```
static navratovy_typ NazevMetody(parametry_oddelene_carkou)
{
  //telo metody
}
```

Metody se podobají funkcím[5] v matematice. Předáme jim nějaké parametry, a v závislosti na nich nám funkce něco vrátí. Příkladem může být $x = sin(y)$. V programování je zápis obdobný, jen parametry i návratová hodnota mohou být libovolného datového typu. Parametrů může být libovolný počet, i nulový, v tom případě píšeme prázdné kulaté závorky (příkladem je metoda `println()`). Návratová hodnota může být buď libovolný datový typ, nebo typ `void`. Metoda typu `void` nevrací žádnou hodnotu, příkladem je metoda `println()`.

Pro příklad napíšeme metodu, která sečte dvě celá čísla. Definice metod píšeme vždy do těla třídy (`class`).

```
static int Secti(int a, int b)
{
  a += b;
  return a;
}
```

[5] V procedurálních jazycích se jim dokonce funkce říká.

Všimněte si hlavně řádku s `return`. Tento příkaz ukončí provádění metody a použije hodnotu proměnné `a` jako návratovou hodnotu metody. Každá metoda, která není typu `void` musí vracet hodnotu daného typu, a to v každém případě. Pokud totiž metoda obsahuje podmíněné výrazy, občas se zapomíná na specifikaci návratové hodnoty ve všech větvích podmíněného příkazu, pokud tímto příkazem metoda končí.

Tuto metodu zavoláme (např. z těla metody `Main`) takto.

```
int cislo1 = 2;
int cislo2 = 5;
int soucet = Secti(cislo1, cislo2);
```

Parametry se metodám předávají tzv. hodnotou. To znamená, že se vezme hodnota proměnné (či konstanty), kterou používáme při volání a předá se parametrům metody. Původní proměnná se tedy nezmění. Ve výše uvedeném příkladu to znamená, že `cislo1` bude mít i po zavolání metody hodnotu 2, ačkoliv jsme v metodě změnili hodnotu proměnné `a` na 7.

Výjimku z tohoto pravidla tvoří tzv. referenční datové typy, které se vždy předávají referencí (odkazem). Mezi tyto typy řadíme pole a objekty.

```
class Cviceni
{
  public static void main (String[] args)
  {
    int[] i = new int[50];
    i[1] = 40;
    neco(i);
    System.out.println(i[1]); //vypise 10
  }

  static void neco (int[] a)
  {
    a[1] = 10;
  }
}
```

9.2 Přetížení metod

Přetížení znamená, že definujeme více metod o shodných názvech, ale rozdílném typu, počtu nebo pořadí parametrů. Návratový typ těchto metod může být stejný, nebo se může lišit. Typickým příkladem je metoda `println()`. Tato metoda je 10x přetížena, neboť na vstupu očekává `int`, `long`, `float`, `double`, `boolean`, `char`, `char[]`, `String`, `Object` nebo nic.

9.3 Rekurze

V Javě je povoleno, aby metoda obsahovala volání sebe sama. Takto lze implementovat např. výpočet faktoriálu jako

```
static int Faktorial(int a)
{
return (a==0)?1:a*Faktorial(a-1);
}
```

nicméně často je použití rekurze špatná volba, neboť volání metody něco stojí[6] a navíc je nutné někde v paměti[7] uchovávat data o dosud neukončených metodách, může se tedy stát, že zásobník přeteče. V tomto případě lze faktoriál napsat lépe pomocí cyklu.

[6]Říká se tomu režie volání metody.
[7]Na tzv. zásobníku, jehož velikost je omezena.

```
static int Faktorial(int a)
{
int fakt = 1;
while(a > 0){
fakt *= a;
a--;
}
return fakt;
}
```

9.4 Zastínění nelokálních proměnných lokálními

Pokud je definována nějaká proměnná na úrovni třídy a poté proměnná stejného jména v metodě, bude v metodě tímto jménem přístupná proměnná definovaná v metodě.

```
class Cviceni
{
  static int prom = 1;
  public static void main (String[] args)
  {
    System.out.println(prom); //vypise 1
    int prom = 2;
    System.out.println(prom); //vypise 2
    System.out.println(Cviceni.prom); //vypise 1
  }
}
```

10 Terminálový vstup a výstup

10.1 Formátovaný terminálový výstup

Nejjednodušší možností interakce programu s uživatelem je terminálový vstup a výstup. S terminálovým výstupem jsme se již setkali, slouží k němu metody `print()` a `println()` třídy `java.io.PrintStream`, ke kterým přistupujeme skrze veřejný statický člen `out` třídy `System`.

10.2 Terminálový vstup

Pro neformátovaný terminálový vstup lze použít metodu `read()` objektu `System.in`. Ta vrací `int`, který reprezentuje jeden znak. Metoda může vyhodit výjimku `java.io.IOException`, kterou je nutné náležitě ošetřit, viz dále.

```
import java.io.*;

class Cviceni
{
public static void main (String[] args) throws IOException
{
char s;
while(s != 'q'){
s = (char) System.in.read();
System.out.println(s);
}
}
}
```

Pro formátovaný (nejen terminálový) vstup je vhodné použít např. třídu `java.util.Scanner`.

```java
java.util.Scanner s = new java.util.Scanner(System.in);
int i = s.nextInt();
```

10.3 Vstupní parametry programu

Pokud nechceme s programem interagovat v průběhu, ale stačí nám to při jeho spuštění, existuje jednodušší způsob. Mějme program `Cviceni.class`, který tentokrát spustíme pomocí příkazu `java Cviceni 2 ahoj 5b`. Řetězce `"2"`, `"ahoj"` a `"5b"` se nazývají vstupními parametry programu a najdeme je v poli `args`, které je parametrem metody `main()`.

```java
class Cviceni
{
/* Vypise:
2
ahoj
5b
*/
public static void main (String[] args) throws IOException
{
for(int i=0; i<args.length; i++){
System.out.println(args[i]);
}
}
}
```

11 Objektově orientované programování

Java je do velké míry založena na principu objektově orientovaného programování (OOP). To obecně spočívá v myšlence, že celý program je tvořen objekty majícími vnitřní stav, které si navzájem posílají zprávy. V Javě je tato myšlenka realizována pomocí tříd.

11.1 Třída a objekt

Třída je objekt[8], který obsahuje datové složky (vnitřní stav) a metody. Metody i datové složky mohou být statické a nestatické. Se statickými jsme se již setkali a patří dané třídě. Nestatické (instanční) datové složky a metody nepatří dané třídě, ale tzv. objektu (instanci třídy). Třídu můžeme brát jako šablonu pro vytváření objektů[9]. Objekty pak vytváříme pomocí klíčového slova `new`.

```java
class Cviceni
{
  static int staticka = 0;
  public int instancni = 1;
  public void setInstancni(int i){
    this.instancni = i; //staci instancni = i;
  }

  public static void main (String[] args) throws IOException
  {
    Cviceni.staticka = 1; //staci staticka = 1;
    Cviceni cv1 = new Cviceni(); //zalozeni instance tridy Cviceni
    Cviceni cv2 = new Cviceni(); //zalozeni druheho objektu
    cv1.setInstancni(2);
    System.out.println(cv1.instancni); //vypise 2
    System.out.println(cv2.instancni); //vypise 1
```

[8] Ne ve smyslu OOP.

[9] Ostatně třída je uživatelsky definovaný datový typ.

```
    }
}
```

11.2 Zapouzdření a modifikátory přístupu

Zapouzdření je, společně s dědičností a polymorfismem, jedním ze tří pilířů OOP. Říká, že k datovým složkám objektů se nemá přistupovat zvnějšku přímo, ale pouze skrz metody daného objektu. V Javě k implementaci tohoto omezení slouží modifikátory přístupu `public`, `protected` a `private`, které píšeme do definice proměnné nebo metody zpravidla jako první klíčové slovo.

`public` znamená, že je proměnná či metoda dostupná odkudkoliv, `protected` pouze z dané třídy, dceřiné třídy, či daného balíku a `private` pouze z dané třídy. Pokud nic neuvedeme, chová se složka jako `protected`, až na to, že takové složky nelze použít v dceřiné třídě nacházející se v jiném balíku. Je dobrou praxí modifikátor vždy explicitně vyplnit.

11.3 Veřejné a neveřejné třídy

Modifikátor `public` můžeme využít i v definici třídy. Pak je tato třída viditelná i z jiných balíků. Na druhou stranu v jednom souboru může být právě jedna veřejná třída a soubor se pak musí jmenovat `*JmenoTridy*.java`. Třída s metodou `main()` musí být vždy označena jako `public`.

11.4 Vznik a zánik objektu

11.4.1 Konstruktor

Při vzniku objektu se volá tzv. konstruktor, je definován jako metoda jmenující se stejně jako třída, bez definice návratové hodnoty. Konstruktor může být (stejně jako metoda) přetížen. Pokud není žádný konstruktor definován programátorem, automaticky se vytvoří tzv. implicitní konstruktor, který je bezparametrický a má prázdné tělo.

```
class Cviceni
{
private i;

public Cviceni(int i){
  this.i = i;
}

public static void main (String[] args)
{
Cviceni cv1 = new Cviceni(2);
}
}
```

11.4.2 Metoda finalize()

Zánik objektu řeší Java automaticky, ve chvíli, kdy už není potřebován, pomocí tzv. garbage collectoru. To je proces běžící v rámci JVM, který uvolňuje již nepotřebnou paměť. Občas se nám ale hodí před zánikem objektu po sobě uklidit, např. zavřít soubor, který objekt využíval. K tomu slouží metoda s hlavičkou `protected void finalize() throws Throwable`, kterou GC sám zavolá před zrušením objektu. Většinou není potřeba ji implementovat. Pokud ji implementujeme, měl by posledním příkazem těla být `super.finalize()`.

11.4.3 Statický inicializační blok

Statický inicializační blok je obdoba konstruktoru pro inicializaci statických složek třídy. Je volán pouze jednou a to před prvním použitím dané třídy (volání statické metody, vytvoření objektu třídy, ...). Syntaxe je následující.

```
class Cviceni
{
static int i;

//inicializacni blok
static{
i = 2;
}
}
```

11.5 Dědičnost

Dědičnost je druhým pilířem OOP. Spočívá v myšlence, že lze objekty kategorizovat, protože mají společné vlastnosti a lze jim nalézt společného předka. Například Člověk i Pes[10] jsou potomky předka Savec, a ten je zase potomkem předka Živočich, atd. Všichni živočichové pak umí (mají metody) žrát(), vylučovat() a pohybovatSe()[11]. Savec a jeho potomci pak dále umí kojit()[12] a Pes umí narozdíl od Člověka štěkat() a má Ocas (datovou složku).

Syntakticky pak dědění docílíme pomocí klíčového slova `extends`. Potomek pak bude obsahovat všechny `public` a `protected` složky předka, které může překrýt (předefinovat). Při překrytí můžeme změnit modifikátory přístupu, ovšem vždy pouze na volnější, než měl předek. Stejně jako pro přístup k složkám daného objektu a konstruktoru daného objektu slouží `this`, pro přístup k složkám předka a konstruktoru předka slouží `super`.

```
class Ctyruhelnik {
protected double a,b,c,d; //delky stran

public Ctyruhelnik(double a, double b, double c, double d){
  this.a = a;
  this.b = b;
  this.c = c;
  this.d = d;
}

public double getObvod(){
  return a+b+c+d;
}
}

class Ctverec extends Ctyruhelnik {
public Ctverec(double a){
  super(a,a,a,a); //volani konstruktoru predka, MUSI to byt prvni prikaz, protoze predek ma
      pouze konstruktor s parametry
}
}
```

Každá třída může mít v Javě pouze jednoho přímého předka. Vícenásobnou dědičnost nahrazuje koncept rozhraní. Třídy, které zdánlivě od nikoho nedědí, jsou potomky třídy `java.lang.Object`, která je společným prapředkem všech tříd.

11.5.1 Test JE-MÁ

Je důležité zmínit, že dědičnost není všelékem, často je vhodnější použít kompozici, tedy objekt, který chceme s naším objektem svázat, bude jeho datovou složkou. Pro rozlišení vhodnosti použití dědičnosti a kompozice lze

[10]Představujme si je jako třídy.
[11]Čímž se liší třeba od rostlin.
[12]Jen samice, takže náš model není dokonalý, ale pro představu stačí.

použít testu je-má. Pokud lze říci, že naše třída JE speciálním případem jiné třídy (předka) je vhodné použít dědičnost. Pokud naše třída (resp. objekt) MÁ jiné objekty jako součást, je vhodné použít kompozici. Např.: Plachetnice JE Loď, ale Plachetnice MÁ Plachty.

11.5.2 Abstraktní a finální třída

Je možné, že některý předek bude natolik abstraktní, že nebude logické tvořit jeho instanci. Příkladem může být `PlanimetrickyObjekt` jako předek třídy `Ctyruhelnik`. Libovolný planimetrický objekt jistě má obvod[13], ale nejde rozumně spočítat, nevíme, kolik má stran, ... Pak takovou třídu označíme v hlavičce klíčovým slovem `abstract`. Od takové třídy nelze vytvářet instance, ale lze od ní dědit. Klíčové slovo lze použít i na metody, ty pak nemají tělo, ale pouze hlavičku. Třída, která obsahuje alespoň jednu abstraktní metodu musí být také abstraktní.

Naopak některá třída může být natolik specializovaná, že už nechceme, aby od ní někdo dědil. Proto ji označíme slovem `final`. Toto klíčové slovo lze použít i v definici jednotlivých metod, takové pak nepůjde překrýt v potomkovi.

11.6 Polymorfismus

Třetím pilířem OOP je polymorfismus, který říká, že potomek může vždy zastoupit předka. To dá rozum, neboť potomek dědí od předka všechny jeho veřejně dostupné metody a datové složky. Použití je mnoho, ukažme si příklad, přičemž předpokládejme, že máme dostupné metody `Ctyruhelnik` a `Ctverec`, jak je uvedeno výše.

```
public class Cviceni
{
static void vypisObvod(Ctyruhelnik c){
  System.out.println("Obvod je: "+c.getObvod());
}

public static void main (String[] args)
{
  Ctyruhelnik c = new Ctverec(2.0);
  vypisObvod(c);
}
}
```

Bez polymorfismu, bychom museli metodu `vypisObvod()` přetížit pro `Ctverec` i `Ctyruhelnik`, ačkoliv by tělo metody bylo v obou případech stejné.

Je dobré poznamenat, že pro proměnnou `c` z příkladu, bychom nemohli volat hypotetické metody definované pouze pro `Ctverec`, protože proměnná je typu `Ctyruhelnik` a ten dané metody neobsahuje.

11.6.1 Pozdní a časná vazba

Řekněme, že by byla metoda `getObvod()` ve třídě `Ctverec` překryta. Která z metod by se pak zavolala v metodě `vypisObvod`, ta z Čtyřúhelníku, nebo ta ze Čtverce? Odpověď není obecně v konceptu OOP jednoznačná a jde o rozdíl mezi časnou a pozdní vazbou.

Časná vazba znamená, že se volaná metoda, nebo datová složka, ke které se přistupuje, řídí dle typu proměnné, v tomto případě by se tedy zavolala metoda ze třídy `Ctyruhelnik`. Časná vazba se tomu říká proto, protože lze určit volanou metodu už v době překladu.

Při použití pozdní vazby se metoda určuje dle skutečného typu objektu, v tomto případě by se tedy použila metoda ze třídy `Ctverec`. Časná vazba se tomu říká proto, protože lze určit metodu obecně až v době běhu programu.[14]

Java většinou[15] používá pozdní vazbu, v našem příkladu tedy bude zavolána metoda ze třídy `Ctverec`.

[13] Ok, je to jen příklad, nebuďte hnidopiši.

[14] Do proměnné `c` bychom mohli v metodě `main()` přiřadit objekt typu čtverec nebo čtyřúhelník např. na základě uživatelova vstupu.

[15] Výjimkou jsou např. finální třídy, které nemohou být děděny, a tedy je volaná metoda jednoznačná už v době překladu.

11.7 Rozhraní

Rozhraní se velmi podobají čistě abstraktním třídám. Mohou obsahovat pouze deklarace (bez těla) veřejných metod a veřejné konstanty. Neboť je vše v rozhraní veřejné, nemusí být uvedeno klíčové slovo `public`.

Typické použití rozhraní je, pokud chceme garantovat, že několik tříd implementuje danou metodu, ale je nevhodné použít dědičnost. Rozhraní pak můžeme použít jako formální parametr, či typ proměnné, jako v příkladu ohledně polymorfismu. Jako skutečný parametr pak můžeme použít libovolný objekt, který rozhraní implementuje.

Rozhraní se definuje podobně jako třída, pouze se `class` nahradí za `interface`. To že třída implementuje rozhraní definujeme pomocí slova `implements`, za kterým následuje seznam rozhraní oddělený čárkami.

```
interface Printable {
void print();
}

class Ctverec extends Ctyruhelnik implements Printable {
public Ctverec(double a){
  super(a,a,a,a);
}
public void print(){
  System.out.println("Ahoj, jsem ctverec o strane delky "+this.a);
}
}
```

12 Výjimky

Občas se stane, že při běhu programu nastane nestandardní situace, např. při pokusu o otevření souboru pro čtení se zjistí, že daný soubor neexistuje. Na nastalou situaci je vhodné reagovat (situaci ošetřit). Často se pak stane, že v místě vzniku mimořádné situace nemáme všechny prostředky pro její ošetření, je tedy nutné mít mechanismus schopný zprávu o mimořádné události doručit do vhodnějšího místa programu. Velmi vhodným mechanismem se pak ukazuje být mechanismus výjimek.

Výjimka je instance třídy, která má za předka třídu `java.lang.Exception`. Výjimku vyvoláme pomocí klíčového slova `throw`. Po vyvolání výjimky se přeruší klasický tok programu a výjimka začne „probublávat“ vzhůru skrz hierarchii volaných metod, dokud není zachycena a ošetřena, nebo nevybublá ven i z metody `main()`. V takovém případě nastane pád programu s chybovou hláškou informující o neošetřené výjimce.

Pro zachycení výjimky se používá konstrukt `try-catch-finally`. V bloku `try` jsou všechny příkazy, při jejichž vykonávání může nastat výjimka, kterou chceme ošetřit. Při vzniku výjimky se, v souladu s výše uvedeným, příkazy nacházející za místem vzniku v bloku `try` již neprovedou. V blocích `catch` jsou pak pravidla pro ošetření výjimky. Bloků může být více za sebou, každý ošetřující jiný typ výjimky, přičemž použit bude první vyhovující blok ve směru odshora dolů, i v případě, že typu výjimky vyhovuje více bloků `catch`. Blok `finally` není povinný, příkazy v něm uvedené se provedou po konci provedení bloků `try` a `catch` a provedou se vždy, nezávisle na tom, jestli výjimka nastala, nebo ne. Sem je vhodné umístit uvolnění zdrojů např. zavření souboru, pokud jsme s ním v bloku `try` pracovali.

Pokud v nějaké metodě může vzniknout neošetřená výjimka, a není to potomek třídy `RuntimeException`, ale běžná výjimka, musí být daná výjimka uvedena v hlavičce metody za klíčovým slovem `throws`. Pokud je takových výjimek více, oddělíme je čárkou.

```
class Cviceni
{
static int sectiKladna(int a, int b) throws InvalidOperationException{
  if(a<=0 || b<=0){
    throw new InvalidOperationException("Cisla musi byt kladna.");
  }
  return a+b;
}

public static void main (String[] args)
```

```
    {
12    try{
      int c = sectiKladna(-1,2);
14    System.out.println(c); //neprovede se
      }
16    catch(InvalidOperationException e){
        System.out.println(e.getMessage);
18    }
      catch(Exception e){ //neprovede se
20      System.out.println("Nastala neznáma chyba."); e.printStackTrace();
      }
22    finally{
        System.out.println("Toto se vypise vzdy.");
24    }
    }
26  }

28  class InvalidOperationException extends Exception{
    public InvalidOperationException(String message){
30    super(message);
    }
32  }
```

13 Balíčky

Asi jste si již všimli, že některé třídy byly uvedeny s podivně dlouhým názvem, např. `java.lang.Object`. Má se to tak, že jde o třídu `Object`, z balíčku `java.lang`. Balíčky jsou způsob, jak zajistit, aby ve velkých projektech byly názvy tříd nekonfliktní, tj. aby nebylo více tříd stejného jména. Pokud se nějaká třída nachází v jiném balíčku, než náš kód, musíme buď použít její celé jméno, nebo na začátku souboru použít příkaz `import`. Jeho použití bylo předvedeno v příkladu s terminálovým vstupem, kdy jsme příkazem `import java.io.*` importovali všechny třídy balíčku `java.io` a tedy jsme se mohli na výjimku `java.io.IOException` odkázat pouze krátkým jménem. K importu nemusíme používat jen hvězdičkovou notaci, ale můžeme specifikovat pouze třídu, kterou chceme importovat.

Existují dva speciální balíčky, a to `java.lang`, který je importován automaticky a implicitní balíček, do kterého patří všechny třídy, které nejsou součástí jiného balíčku.

13.1 Vlastní balíčky

Vlastní balíček vytvoříme tak, že na začátek souboru, který chceme do balíčku přidat, napíšeme příkaz `package nazevBalicku;`. Hierarchie balíčků se pak musí odrážet v adresářové struktuře, například soubor `Pokus.class` balíčku `cz.fykos.timkol` musí být v adresáři `cz/fykos/timkol`.