Základy Java

Přehled Javy

Programovací jazyk Java byl vytvořen u Sun Microsystems pod taktovkou Jamese Goslinga roku 1995 jako hlavní komponenta platformy Java (Java 1.0 [J2SE])

S rozvojem Javy a její vzrůstající popularitě přišly různé konfigurace, které vyhovují různým typům platforem. Např. J2EE pro podnikové aplikace, J2ME pro mobilní aplikace.

Sun Microsystems přejmenoval nové verze J2 na Java SE, Java EE a Java ME. Java přichází se zárukou **Jednou Napiš, Kdekoliv Spusť**.

Java je:

- **Objektově orientovaná**: V Javě je cokoliv objektem. Java může být snadno rozšířená díky svému objektovému založení.
- **Nezávislá na platformě:** Když kompilujete Javu, tak na rozdíl od ostatních programovacích jazyků včetně C a C++, není kód kompilován na specifický stroj, ale raději do **bytecodu** nezávislého na platformě. Tento bytecode se šíří pomocí internetu a poté je interpretován virtuálním strojem (JVM) na platformě, kde je spuštěn.
- **Jednoduchá:** Java byla navržena tak, aby bylo jednoduché se s ní naučit. Pokud rozuímte základním konceptům OOP, pak se Javu naučíte velice snadno.

- **Bezpečná:** S bezpečnostními prvky Javy můžete vyvíjet systémy bez virů a bez vměšování se. Autentifikační techniky jsou založeny na šifrování veřejným klíčem.
- **Nezávislá na platformě**: Kompilátor Javy generuje objektový soubor nezávislý na platformě, který po zkompilování funguje na řadě procesorů, samozřejmě v přítomnosti Java runtime systému.
- **Přenosná:** Díky své nezávislosti na architektuře a díky absenci aspektů specifikace závislých na implementaci je Java přenosná. Kompilátor Javy je napsán v ANSI C.

- **Robustní:** Java se snaží odstranit situace náchylné k chybám zaměřením především na kontrolování chyb za kompilace a za běhu.
- **Vícevláknová:** S vlastností vícevláknovosti je v Javě možné psát programy, které dokáží dělat mnoho úkolů najednou. Tato fundamentální vlastnost umožňuje programátorům psát hladce běžící interaktivní aplikace.
- **Interpretovaná:** Bytecode Javy je překládán za běhu do strojových instrukcí a nikde se nezapamatovává. Vývoj je rychlejší a analytičtější, protože překlad a linkování jsou inkrementální a jednoduché procesy.

- **Výkonná:** S použitím Just-In-Time kompilátorů dosahu Java vysokých výkonů.
- **Distribuovaná:** Java je navržená pro šířitelné prostředí internetu.
- **Dynamická**: Java je považovaná za více dynamickou než C nebo C++, protože byla navržena tak, aby se přizpůsobila vyvíjejícímu se prostředí. Programy v Javě mohou přechovávat značné množství run-time informací, které mohou být použity k ověření a rozhodování o přístupu objektů za běhu.

Poznámky k jazyku JAVA

JAVA jako jazyk, navazující na C++

Ve druhé polovině devadesátých let se ukázala pouhá podpora objektového přístupu jako nepostačující, vzniká jazyk Java. Na rozdíl od C++, nepřejímá z C bezmyšlenkovitě vše a některé prvky dokonce záměrně nepodporuje, například ukazatele, nicméně opět staví na syntaxi jazyka C.

Všeobecně

Velmi rozšířené, zajímavé a vcelku kvalitní je prostředí NetBeans.

Jeho výhody jsou tři:

- Je dobře podporované firmou Oracle
- Má kvalitní *free* verzi
- Je použitelné pro řadu jazyků. Vedle C++ to jsou JAVA, PHP, Python a mnoho dalších

Základní elementy v Javě

Když uvažujeme nad programem v Javě, můžeme jej definovat jako soubor objektů, které vzájemně komunikují voláním svých metod. Pojďme se v rychlosti podívat na to, co znamená třída, objekt, metody, proměnné a instance.

Objekt – Objekty reálného světa mají stavy a chování. Příklad: Pes má stavy – barva, jméno, rasa, a rovněž chování – vrtění chvostem, štěkání, žraní. Objekt je instncí třídy.

Třída – třída může být definována jako šablona, vzor, který popisuje stavy a chování objektu.

Metody – Metoda je v podstatě chování. Třída může obsahovat mnoho metod. Metody obsahují logiku, manipulují s daty a vykonávají veškeré akce. Jedná se o obdobu funkcí z klasických programovacích jazyků.

Proměnné instancí – Každý objekt má unikátní sadu instančních proměnných. Stav objektu je vytvořen hodnotami přiřazenými k této instanci.

Základní syntaxe

Základní syntaxi si můžeme ukázat na jednoduchém příkladu:

```
public class MyFirstJavaProgram {

   /* This is my first java program.
    * This will print 'Hello World'
      as the output */

    public static void main(String []args){
       // prints Hello World
       System.out.println("Hello World");
}
}
```

Zapamatujte si

Při programování v Javě je velice důležité pamatovat na tyto věci:

Citlivost na malá/velká písmena - Java rozlišuje velikost písmen, což znamená, že *Hello* a *hello* jsou v Javě dvě různé věci.

Jména tříd – pro všechny názvy tříd musí být použity názvy s **velkým** počátečním písmenem.

Příklad: MojePrvniTrida

Názvy metod – veškeré metody musí začínat **malým** písmenem. Pokud se jedná o víceslovný název, každé vnitřní slovo by mělo začínat velkým písmenem.

Příklad: public void nazevMojiMetody

Název programu – název programu se musí přesně shodovat s názvem třídy (nezapomeňte, že Java je citlivá na velikost písmen!) s příponou ".java". **Pokud se neshoduje název programu s názvem třídy, pak se váš program nezkompiluje**.

Příklad: Předpokládejme třídu MujPrvniProgramVJave. Pak by měl být program uložen pod názvem 'MujPrvniProgramVJave.java'.

public static void main(String args[]) – běh programu v Javě vždycky začíná od metody *main()*, která je povinnou součástí programu v Javě.

Identifikátory v Javě

Všechny komponenty v Javě musejí mít název. Názvy používané pro třídy, proměnné a metody se nazývají **identifikátory**.

Zapamatujte si

V Javě je potřeba pamatovat na některé věci kolem identifikátorů:

Identifikátor musí začínat písmenem (**A** až **Z** nebo **a** až **z**), znakem dolaru($) nebo podtržítkem (_). V zásadě by šlo používat i písmena s diakritikou, ale nedělejte to.

Po prvním písmenu mohou být identifikátory libovolnou kombinací písmen. Doporučuje se používat “velbloudí zápis” jako například *TotoJeMojePrvniTrida*.

Klíčové slovo nemůže být použito jako identifikátor.

Nejdůležitější je, že identifikátory jsou citlivé na velikost písmen

Příklady správných identifikátorů jsou: *vek, $plat, _hodnota, __1_hodnota*

Příklady nesprávných identifikátorů: *123abc, -plat*

Klíčová slova v Javě

V Javě existují následující klíčová slova. Tato slova se nesmějí použít jako identifikátory.

abstract	assert	boolean	break
byte	case	catch	char
class	const	continue	default
do	double	else	enum
extends	final	finally	float
for	goto	if	implements
import	instanceof	int	interface
long	native	new	package
private	protected	public	return

short	static	strictfp	super
switch	synchronized	this	throw
throws	transient	try	void
volatile	while		

Typy v Javě

Pojem typu

V paměti počítače jsou všechny údaje vyjádřeny pomocí nul a jedniček:

```
0110 1100
```

Tento údaj může znamenat velmi mnoho věcí. Může to být instrukce programového kódu. Může to být číslo. Může to být znak textu. Může to být pixel obrázku. Těch možností je velice moc. Přitom s čísly se pracuje úplně jinak než s texty. A s těmi zase jinak, než s obrázky. Proto počítač potřebuje přesně vědět, co ten údaj znamená.

Řekneme mu to pomocí **typů**. Jestliže ten údaj znamená číslo, pak říkáme, že je typu číslo. Pokud znamená znak, říkáme, že je typu znak. A tak podobně.

Dva druhy typů

Programovací jazyk Java je staticky typovaný. To znamená, že proměnné musejí být nejdříve nadeklarovány, než se dají použít, a že jejich typ se během výpočtu nemůže měnit.

Hned na začátku je potřeba říci, že typy můžeme rozdělit do dvou kategorií: na typy **primitivní** a na typy **referenční**.

Primitivní datové typy

Příklad primitivního datového typu ukazuje obrázek.

Programovací jazyk Java je staticky typovaný. To znamená, že proměnné musejí být nejdříve nadeklarovány, než se dají použít,

a že jejich typ se během výpočtu nemůže měnit. Deklarace proměnné probíhá následujícím způsobem:

```
int price = 123456;
```

Touto deklarací říkáte programu, že existuje proměnná s názvem "price", obsahuje celé číslo a má počáteční hodnotu "123456". Datový typ tedy určuje hodnoty, které může proměnná obsahovat, plus operace, které s ní lze provádět. Vedle *int*, programovací jazyk Java podporuje ještě sedm dalších primitivních datových typů. Primitivní typ je předdefinován jazykem a jeho pojmenování je vyhrazené klíčové slovo. Primitivní hodnoty nesdílejí stav s jinými primitivními hodnotami.

Osm primitivních datových typů podporovaných programovacím jazykem Java:

typ	popis	velikost	min. hodnota	max. hodnota
byte	celé číslo	8 bitů	-128	+127
short	celé číslo	16 bitů	-32768	+32767
int	celé číslo	32 bitů	-2147483648	+2147483647
long	celé číslo	64 bitů	-922337203…	+922337203…
float	reálné číslo	32 bitů	-3.40282e+38	+3.40282e+38
double	reálné číslo	64 bitů	-1.797…e+308	+1.797…e+308
char	znak UNICODE	16 bitů	/u0000	/uFFFF

boolean	logic.hodnota	1 bit	-	-

• **byte**: datový typ byte je 8-bitové celé číslo ve dvojkovém doplňku. Má minimální hodnotu -128 a maximální hodnotu 127 (včetně). Byte může být užitečný pro úsporu paměti ve velkých polích, kde skutečně záleží na úspoře paměti. Dá se ho také použít místo *int*, kde jeho limity mohou pomoci zdokumentovat kód; Skutečnost, že rozsah proměnné je omezen, může sloužit jako forma dokumentace.

• **short**: typ dat **short** je 16-bitové celé číslo ve dvojkovém doplňku. Má minimální hodnotu -32 768 a maximální hodnotu

32 767 (včetně). Stejně jako v případě **byte**, může být užitečný pro úsporu paměti ve velkých polích.

• **int**: výchozí datový typ int je 32-bitové celé číslo ve dvojkovém doplňku. Má minimální hodnotu -2^{31} a maximální hodnotu $2^{31}-1$. V Java SE 8 a novějších můžete použít datový typ int také k reprezentaci 32-bitového celého čísla bez znaménka, které má minimální hodnotu 0 a maximální hodnotu $2^{32}-1$. Dělá se to pomocí třídy Integer, do které byly přidány statické metody, jako je compareUnsigned, divideUnsigned. právě na podporu aritmetických operací s celými čísly bez znaménka.

• **long**: datový long typ je 64-bitové celé číslo ve dvojkovém doplňku. Má minimální hodnotu -2^{63} a maximální hodnotu $2^{63}-1$.

V Java SE 8 a novějších můžete použít datový typ long také k reprezentaci 64-bitového celého čísla bez znaménka, které má minimální hodnotu 0 a maximální hodnotu 2^{64}-1. Dělá se to pomocí třídy Long, do které byly přidány statické metody, jako je compareUnsigned, divideUnsigned. právě na podporu aritmetických operací s dlouhými celými čísly bez znaménka.

• **float**: typ float dat s jednoduchou přesností je 32-bitové číslo s plovoucí desetinnou čárkou, vyjádřené podle normy IEEE 754. Jeho rozsah hodnot je v tabulce. Tento typ dat nesmí být používán pro přesné hodnoty, jako jsou měny. Místo toho použijte třídu java.math.BigDecimal.

• **double**: typ dat double s dvojitou přesností je 64-bitové číslo s plovoucí desetinnou čárkou, vyjádřené podle normy IEEE 754. Jeho rozsah hodnot je v tabulce. Jedná se o typickou volbu pro desetinná čísla. Tento typ dat nesmí být používán pro přesné hodnoty, jako jsou měny. Místo toho použijte třídu java.math.BigDecimal.

• **boolean**: datový typ boolean má pouze dvě možné hodnoty: true a false. Tento typ dat se použije pro jednoduché příznaky, například v podmínkách. Tento typ dat představuje jeden bit informace, ale její "velikost" není přesně definována.

• **char**: typ dat je char reprezentuje jeden 16-bitový Unicode znak. To má minimální hodnotu "\ u0000" (nebo 0) a maximální hodnotu " \ uFFFF" (nebo 65 535 včetně).

Vedle osmi primitivních datových typů uvedených výše, programovací jazyk Java také poskytuje speciální podporu znakových řetězců pomocí třídy java.lang.String. Když zapíšete znakový řetězec v uvozovkách, automaticky se vytvoří nový objekt String; například

```
String s = "Toto je řetězec";
```

Objekty String jsou neměnné, což znamená, že po vytvoření, **jejich hodnoty nemohou být změněny**. Technicky vzato, třída String není primitivní datový typ, ale s ohledem na zvláštní

podporu, která mu byla předložena jazykem, budete s ním pracovat, jako by to primitivní datový typ byl.

Literály

Proměnným primitivního typu je možné přiřadit hodnoty pomocí literálů:

```
boolean vysledek = true;
char capitalC = "C";
byte b = 100;
short s = 10000;
int i = 100000;
```

Literál je typu long, pokud končí písmenem L nebo l; jinak je typu int. Doporučuje se použít velké písmeno L, protože malé písmeno l je těžké odlišit od číslice 1.

Hodnoty typů byte, short, int a long mohou být vytvořeny z literálů int. Hodnoty typu long, které překračují rozsah int, mohou být vytvořeny z dlouhých literálů. Celočíselné literály mohou být vyjádřeny v těchto číselných soustavách:

- Desetinná: základ 10; čísla se skládají z číslic 0 až 9
- Hexadecimální: základ 16, číslia se skládají z číslic 0 až 9 a písmen A až F. Označuje jej **prefix 0x**.
- Binární: základ 2, číslia se skládají z číslic 0 a 1 (v Java SE 7 a novější). Označuje jej **prefix 0b**.

```
// Číslo 26 v desítkové soustavě
int decVal = 26;
```

```
// Číslo 26, v šestnáctkové soustavě
int hexVal = 0x1A;

// Číslo 26, v binární
int binVal = 0b11010;
```

Literál s pohyblivou řádovou čárkou je typu float, pokud končí písmenem F nebo f; jinak je typu double a může případně skončit s písmenem D nebo d.

Typy s pohyblivou řádovou čárkou (float a double) lze také vyjádřit pomocí E nebo E (pro vědecké notaci), F nebo f (32-bit float) a D nebo d (64-bit double, což je výchozí):

```
double d1 = 123.4;
```

```
// Stejná hodnota jako d1, ale ve vědecké
notaci
double d2 = 1.234e2;
float f1 = 123.4f;
```

Literály typů char a String mohou obsahovat znaky Unicode (UTF-16). Pokud to váš editor a souborový systém dovolí, můžete použít tyto znaky přímo v kódu. Pokud ne, můžete použít "Unicode escape" jako "\ u0108" (velké C s háčkem), nebo například "S \ u00ED Se \ u00F1or" (Si Senor ve španělštině).

Vždy používejte ‘apostrofy’ pro char a "uvozovky" pro řetězce.

Escape sekvence Unicode může být použita i jinde v programu (například v názvech polí), a to nejen v char nebo v řetězcích.

Programovací jazyk Java podporuje také několik speciálních escape sekvencí pro řetězcové literály: **\b** (backspace), **\t** (tab) **\n** (posun o řádek), **\f** (form feed) **\r** (návrat vozíku), **\"**(uvozovky), **\'**(apostrof), a **** (zpětné lomítko).

K dispozici je také speciální literál *null*. Ten je často používán v programech jako marker pro indikaci, že nějaký objekt není k dispozici.

V Java SE 7 a vyšších, se mezi číslicemi může objevit libovolný počet znaků podtržení (_). Tato funkce umožňuje zlepšit čitelnost kódu.

Například, pokud váš kód obsahuje čísla s mnoha číslicemi, můžete pomocí podtržítka oddělit číslice ve skupinách po třech podobně, jako by se používala mezera nebo jiný oddělovač.

Následující příklad ukazuje další způsoby, jak můžete použít podtržítko v číselných literálech:

```
long creditCardNumber =
1234_5678_9012_3456L;
long socialSecurityNumber = 999_99_9999L;
float pi = 3.14_15F;
long hexBytes = 0xFF_EC_DE_5E;
long hexWords = 0xCAFE_BABE;
long maxLong = 0x7fff_ffff_ffff_ffffL;
byte nibbles = 0b0010_0101;
```

```
long bajty =
0b11010010_01101001_10010100_10010010;
```

Referenční datové typy

Všechny datové typy, které nejsou primitivní, jsou referenční. Příkladem referenčního datového typu jsou všechny třídy, pole a další typy.

Příklad referenčního datového typu ukazuje obrázek.

Referenční datové typy jsou v paměti uloženy na dvou místech. Když takový typ deklarujete, kompilátor v paměti vytvoří místo a přiřadí mu jméno. Říkáme, že jsme **deklarovali** proměnnou. Zajímavé je, že délka tohoto místa vůbec nezáleží na tom, kolik

bytů zabírá referenční typ; místo má takovou délku, aby v něm mohla být adresa do paměti.

Ve druhém kroku, který se nazývá inicializace, vytvoří se v paměti místo pro proměnnou. Toto místo má velikost podle potřeby, aby se do něj vešla všechna data patřící k proměnné. Adresa tohoto místa (čili reference) se vloží do pojmenovaného umístění, které vzniklo při deklaraci.

Obě operace, deklaraci a inicializaci, lze spojit do jednoho příkazu, jak to ukazuje obrázek.

Tento způsob práce s referenčními typy má výhody i nevýhody:

Výhodu je, že velikost dat se může dynamicky měnit za běhu programu, podle potřeby.

Nevýhodou je, že **každý referenční typ musíme před použitím inicializovat**. Nestačí jej deklarovat například takto

```
int [] totoJeSpatne;
```

protože tím se ještě žádné místo pro data nevytvořilo a `totoJeSpatne` je prázdné. Jakýkoliv pokus o práci s `totoJeSpatne` bez následné inicializace skončí chybou.

Modifikátory v Javě

Stejně jako u jiných jazyků, je možné modifikovat třídy, metody, atd. použitím **modifikátorů**. Rozeznáváme dva typy modifikátorů:

- **Modifikátory přístupu:** default, public, protected, private
- **Ostatní modifikátory:** final, abstract, strictfp

Proměnné v Javě

V Javě se setkáváme s následujícími druhy proměnných:

- **Proměnné instancí** (ne-statické proměnné). Instanční proměnné jsou definovány a platí jen uvnitř své instance. Ve

třídě nejsou přístupné. Z toho plyne, že instanční proměnné stejného jména mohou mít v různých instancích různé hodnoty.

- Proměnné třídy (**statické proměnné**) jsou definovány ve třídě a jejich modifikátor je **static**. Existuje jen jediná kopie takové proměnné. Tato kopie se sdílí mezi všemi instancemi. To znamená, že pokud ji jedna instance změní, tak se změna promítne do všech ostatních instancí.
- **Lokální proměnné** jsou proměnné, které se vytvářejí jen dočasně, pro potřeby nějakého příkazu nebo bloku příkazů. Příklad: `for (int i=0; i<7; i++)…`

Komentáře v Javě

Java podporuje jednořádkové a víceřádkové komentáře velice podobně jako C a C++. Veškeré znaky uvnitř komentáře jsou kompilátorem Javy ignorovány.

```
public class MyFirstJavaProgram{

   /* This is my first java program.
      This will not print 'Hello World'
   as the output.
      This is an example of multi-line
   comment. */

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

```
        // This is a single line comment
    /* This is a single line comment.*/
        System.out.println("Hello World");
    }
}
```

Vynechávání řádků

Řádce obsahující jenom bílé znaky (mezera, tabulátor, CR/LF), případně komentář, se říká blank line, a Java ji ignoruje.

Pole v Javě

Pole jsou objekty, které uchovávají řadu proměnných stejného typu. Pole samo o sobě je objekt na *heapu* (hromadě). V následujích kapitolách si řekneme, jak pole deklarovat, vytvořit a inicializovat.

Enum v Javě

Enumy (správně česky: výčtové typy) byly zavedeny v Javě 5.0. **Enumy** omezují proměnnou tak, aby mohla obsahovat jenom několik málo předem definovaných hodnot.

Použitím enumu je možné snížit počet chyb ve vašem kódu.

Když například uvažujeme o aplikaci pro obchod s džusy, tak by bylo možné zredukovat velikost sklenice na malou, střední a velkou. Tím se zajistí, že program nemůže pracovat s jinými hodnotami než malá-střední-velká. A navíc, je to mnohem názornější, než používat číselné konstanty (jako třeba malá=0, střední=1 apod.)

Příklad:

```
class FreshJuice {

// zde si pripravim vyctovy
// typ FreshJuiceSize
enum FreshJuiceSize{SMALL,MEDIUM,LARGE}
```

```
// vytvorim promennou "size"
// toho typu
FreshJuiceSize size;
}

public class TestClass{

public static void main(String []args){
System.out.println("Starting:");

// vytvorim juice = instance
// tridy FreshJuice
FreshJuice juice = new FreshJuice();

// trida FreshJuice obsahuje
```

```
// promennou “size”
// a take konstantu MEDIUM
juice.size =
FreshJuice.FreshJuiceSize.MEDIUM;

System.out.println("Size: " +
juice.size);
System.out.println("Finished OK.");
}
}
```

Výše popsaný příklad bude mít následující výsledek:

```
   Size: MEDIUM
```

Poznámka: enumy mohou být deklarovány samy za sebe nebo uvnitř třídy.Uvnitř v enumech lze rovněž definovat metody, proměnné, konstruktory.

Základní operátory v Javě

Java poskytuje bohatou paletu operátorů pro manipulaci s proměnnými. Veškeré operace v Javě můžeme rozdělit do následujících skupin:

- Aritmetické operátory
- Relační operátory
- Bitové operátory
- Logické operátory
- Operátory přiřazení
- Různé operátory

Aritmetické operátory

Aritmetické operátory se používají v matematických výrazech stejným způsobem jako v matematice. Následující tabulka je výpisem aritmetických operátorů:

Předpokládejme, že celočíselná proměnná A má hodnotu 10 a B 20.

Operator	Popis	Příklad
+	Součet – sčítá hodnoty na obou stranách operátoru	A + B dává 30
-	Odečítání – odečítá pravý operand od levého	A - B dává -10

*	Násobení – násobí hodnoty na obou stranách operátoru	A * B dává 200
/	Dělení –dělí operand vlevo operandem vpravo	B / A dává 2
%	Modulo – dělí operand nalevo operandem vpravo a vrací zbytek	B % A dává 0
++	Inkrementace – zvýší hodnotu operandu o 1	B++ dává 21
--	Dekrementace – sníží hodnotu operandu o 1	B-- dává 19

Relační operátory:

Jazyk Java podporuje následující relační oerátory.

Předpokládejme, že je v proměnné A hodnota 10 a v B 20:

Opera-tor	Popis	Příklad
==	Kontroluje, zda jsou si hodnoty na obou stranách operátoru rovny nebo ne, pokud ano, pak se výraz vyhodnotí jako true.	(A == B) není pravda.
!=	Kontroluje, zda jsou si hodnoty na obou stranách operátoru rovny nebo ne, pokud ne, pak se výraz vyhodnotí jako true	(A != B) je pravda

>	Kontroluje, zda je hodnota levého operandu větší než hodnota pravého operandu, pokud ano, pak se výraz vyhodnotí jako true.	(A > B) není pravda
<	Kontroluje, zda je hodnota levého operandu menší než hodnota pravého, pokud ano, pak se výraz vyhodnotí jako true.	(A < B) je pravda
>=	Kontroluje, zda je hodnota levého operandu větší nebo rovna hodnotě pravého operandu, pokud ano, pak se výraz vyhodnotí jako true.	(A >= B) není pravda
<=	Kontroluje, zda je hodnota levého operandu menší nebo rovna hodnotě	(A <= B) je pravda

pravého, pokud ano, pak se výraz vyhodnotí jako true.

Pozor na to, že rovnost testujeme dvěma == proto, aby se to nepletlo s běžným přiřazením do proměnné, které se dělá jen jedním = . Je to velmi častá chyba.

Pokud chceme nějaký výraz znegovat, napíšeme ho do závorky a před něj vykřičník.

Když budeme chtít vykonat více než jen jeden příkaz, musíme příkazy vložit do bloku ze složených závorek.

Pozor!

Stringy porovnáváme pomocí metody equals, nikoli pomocí operátoru == jako u čísel !

Je to dáno tím, že String je referenční datový typ. Podmínka `"Text1" == "Text2"` je špatně, musíme psát `"Text1".equals("Text2")`.

Bitové operátory

Java definuje několik bitových operátorů, které lze aplikovat na číselné typy long, int, short, char a byte.

Bitový operátor pracuje s bity a provádí operace z bitu na bit.

Předpokládejme, že pokud a = 60 a b = 13, pak budou ve dvojkové soustavě vypadat takto:

```
  a = 0011 1100
  b = 0000 1101
-----------------
a&b = 0000 1100
a|b = 0011 1101
a^b = 0011 0001
~a  = 1100 0011
```

Následující tabulka vypisuje bitové operátory.

Předpokládáme, že A = 60 a B = 13:

Oper	Popis	Příklad

a-tor		
&	Binární AND kopíruje do výsledku bit, pokud existuje u obou operandů	(A & B) dává 12, což je 0000 1100
|	Binární OR kopíruje bit, pokud existuje alespoň v jednom operandu	(A | B) dává 61, což je 0011 1101
^	Binární XOR kopíruje bit, pokud se nastaven v jednom operandu, ale ne v obou	(A ^ B) dává 49, což je 0011 0001
~	Binární jednotkový doplněk je unární	(~A) dává -61, což je 1100 0011 ve dvojkovém doplňku, díky

	operátor a má za následek bitů z 0 na 1 a naopak	binárnímu číslu se znaménkem
<<	Binární posun doleva. Hodnoty operandu nalevo se posunou doleva o počet bitů specifikovaných v pravém operandu.	A << 2 dává 240, což je 1111 0000
>>	Binární posun doprava. Hodnoty operandu nalevo se posunou doprava o počet bitů specifikovaných v	A >> 2 Dává 15, což je 1111

	pravém operandu.	
>>>	Posun vpravo bez vyplňování. Hodnota operandu vlevo je posunuta doprava o počet bitů specifikovaných operandem napravo a posunuté hodnoty jsou zaplněny nulami	A >>>2 dává 15, což je 0000 1111

Logické operátory:

Následující tabulka vypisuje logické operátory.

Předpokládejme booleovské hodnoty, A je true a B je false.

Operator		
&&	Logické AND. Pokud jsou oba operandy nenulové, pak je podmínka pravda.	(A && B) je nepravda.
||	Logické OR. Pokud je alespoň jeden operand nenulový, pak je výraz vyhodnotcen jako pravdivý.	(A || B) je pravda.
!	Logická negace. Použit k převrácení logického stavu svého operandu. Pokud je podmínka pravdivá, logická negace z ní udělá nepravdivou	!(A && B) je pravda.

Operátory přiřazení

Jazyk Java podporuje následující operátory přiřazení:

Opera-tor	Popis	Příklad
=	Jednoduchý operátor přiřazení. Přiřazuje hodnoty z pravé strany operandů do operandu nalevo.	C = A + B přiřadí hodnotu A + B do C
+=	Součet A přiřazení. Přičte pravý operand k levému a přiřadí výsledek levému operandu	C += A je ekvivalentní s C = C + A
-=	Rozdíl A přiřazení, odečítá pravý operand	C -= A je

	od levého a přiřazuje výsledek levému operandu	ekvivalentní s C = C - A
***=**	Násobení A přiřazení, vynásobí pravý operand levým a přiřadí hodnotu levému operandu	C *= A je ekvivalentní s C = C * A
/=	Rozdíl A přiřazení, dělí operand vlevo operandem vpravo a přiřazuje výsledek levému operandu	C /= A je ekvivalentní s C = C / A
%=	Modulo A přiřazení, bere modulo za použití dvou operandů a přiřazuje výsledek levému operandu	C %= A je ekvivalentní s C = C % A
<<=	Posun doleva A přiřazení	C <<= 2 je ekvivalentní s C = C << 2

>>=	Posun doprava A přiřazení	C >>= 2 je ekvivalentní s C = C >> 2
&=	Bitové AND přiřazení	C &= 2 je ekvivalentní s C = C & 2
^=	bitové XOR(exkluzivní OR) a přiřazení	C ^= 2 je ekvivalentní s C = C ^ 2
|=	bitové OR (inkluzivní) a přiřazení	C |= 2 je ekvivalentní s C = C | 2

Různé operátory

Java podporuje ještě pár dalších operátorů.

Podmínečný operátor (? :)

Podmínečný operátor je rovněž znám jako ternární operátor.

Tento operátor se skládá ze tří operandů a používá se k vyhodnocení Booleovských výrazů.

```
proměnná x = (výraz) ? hodnota pokud true : hodnota pokud false
```

Následuje příklad:

```
public class Test {
```

```
    public static void main(String args[]){
        int a , b;
        a = 10;
        b = (a == 1) ? 20: 30;
        System.out.println("Hodnota b je : "
       +  b );

        b = (a == 10) ? 20: 30;
        System.out.println("Hodnota b je : "
       + b );
    }
}
```

Tento kód by měl následující výstup:

```
Hodnota b je : 30
Hodnota b je : 20
```

Operátor instanceof

Tento operátor se používá pouze pro referenční proměnné. Operátor kontroluje, zda je objekt určitého typu (typu třídy nebo rozhraní)

```
( Referenční proměnná ) instanceof  (typ třídy/rozhraní)
```

Pokud objekt referencovaný proměnnou nalevo od operátoru úspěšně projde testem JE-TO? pro typ třídy/rozhraní na pravé straně, pak bude výsledek pravdivý.

```
public class Test {

   public static void main(String args[]){
      String name = "James";
```

```
    // následující kód vrátí true,
    // protože name je typu String
        boolean result = name
       instanceof String;
        System.out.println( result );
    }
}
```

Toto bude výstup:

```
true
```

Tento operátor vrátí true, i pokud je porovnávaný objekt **kompatibilní vzhledem k přiřazení** s objektem napravo.

Následuje ještě jeden příklad:

```
class Vehicle {}
```

```
public class Car extends Vehicle {
   public static void main(String args[]){
      Vehicle a = new Car();
      boolean result =  a instanceof Car;
      System.out.println( result );
   }
}
```

Toto by nám vrátilo na výstupu:

```
true
```

Priorita operátorů v Javě

Priorita operátorů určuje pořadí výpočtu členů ve výrazu. Určité operátory mají přednost před ostatními, například násobení má přednost před sčítáním:

Například x = 7 + 3 * 2 přiřadí do x hodnotu 13, ne 20, protože operátor * má přednost před +, takže se nejdřív vynásobí 3*2 a pak se přičte 7.

Operátor s nejvyšší předností objevíte na úplném začátku tabulky, operátor s nejnižší předností na konci. Ve výrazu budou vyhodnoceny výrazy s vyšší prioritou před těmi s nižší.

Kategorie	Operator	Asociativita
Přípona	() [] . (operátor tečka)	Zleva doprava
Unární	++ - - ! ~	**Zprava doleva**
Násobící	* / %	Zleva doprava

Sčítací	+ -	Zleva doprava
Posun	>> >>> <<	Zleva doprava
Relační	> >= < <=	Zleva doprava
Rovnost	== !=	Zleva doprava
Bitové A	&	Zleva doprava
Bitové XOR	^	Zleva doprava
Bitové OR	|	Zleva doprava

Logické A	&&	Zleva doprava
Logické NEBO	||	Zleva doprava
Podmínka	?:	**Zprava doleva**
Přiřazení	= += -= *= /= %=>> = <<= &= ^= |=	**Zprava doleva**
Čárka	,	Zleva doprava

Příklady použití operátorů v Javě

Následující jednoduchý program nám demonstruje aritmetické operace. Zkopírujte si následující kód, ulože do souboru Test.java a zkompilujte a spusťte jej.

```
public class Test {

  public static void main(String args[]) {
     int a = 10;
     int b = 20;
     int c = 25;
     int d = 25;
```

```
        System.out.println("a + b = "
        + (a + b) );
        System.out.println("a - b = "
        + (a - b) );
        System.out.println("a * b = "
        + (a * b) );
        System.out.println("b / a = "
        + (b / a) );
        System.out.println("b % a = "
        + (b % a) );
        System.out.println("c % a = "
        + (c % a) );
        System.out.println("a++   = "
        +  (a++) );
        System.out.println("b--   = "
```

```
        +  (a--) );
        // Check the difference in d++ and ++d
        System.out.println("d++   = "
        +  (d++) );
        System.out.println("++d   = "
        +  (++d) );
    }
}
```

Což by vyprodukovalo následující výstup:

```
a + b = 30
a - b = -10
a * b = 200
b / a = 2
b % a = 0
c % a = 5
```

```
a++   = 10
b--   = 11
d++   = 25
++d   = 27
```

Příklad relačních operátorů

Následující jednoduchý program demonstruje relační operátory.

Uložte si jej, zkompilujte a spusťte.

```
public class Test {

  public static void main(String args[]) {
     int a = 10;
     int b = 20;
     System.out.println("a == b = "
      + (a == b) );
```

```
        System.out.println("a != b = "
        + (a != b) );
        System.out.println("a > b = "
        + (a > b) );
        System.out.println("a < b = "
        + (a < b) );
        System.out.println("b >= a = "
        + (b >= a) );
        System.out.println("b <= a = "
        + (b <= a) );
    }
}
```

Měli byste vidět takovýto výsledek:

```
a == b = false
a != b = true
```

```
a > b = false
a < b = true
b >= a = true
b <= a = false
```

Příklad bitových operátorů

Následující jednoduchý program demonstruje bitové operátory.

Uložte si jej, zkompilujte a spusťte.

```
public class Test {

  public static void main(String args[]) {
     int a = 60;/* 60 = 0011 1100 */
     int b = 13;/* 13 = 0000 1101 */
     int c = 0;
```

```
c = a & b;         /* 12 = 0000 1100 */
System.out.println("a & b = " + c );

c = a | b;         /* 61 = 0011 1101 */
System.out.println("a | b = " + c );

c = a ^ b;         /* 49 = 0011 0001 */
System.out.println("a ^ b = " + c );

c = ~a;           /*-61 = 1100 0011 */
System.out.println("~a = " + c );

c = a << 2;      /* 240 = 1111 0000 */
System.out.println("a << 2 = " + c );
```

```
        c = a >> 2;      /* 215 = 1111 */
        System.out.println("a >> 2  = " + c );

        c = a >>> 2;      /* 215 = 0000 1111 */
        System.out.println("a >>> 2 = " + c );
    }
}
```

Měli byste vidět takovýto výsledek:

```
a & b = 12
a | b = 61
a ^ b = 49
~a = -61
a << 2 = 240
a >> 15
```

```
a >>> 15
```

Příklad logických operátorů

Podmínky je možné skládat, a to pomocí dvou základních operátorů:

Operátor	Význam
&&	a současně
||	nebo

Priorita se dá upravit pomocí závorek, jako na následujícím příkladu:

```
Scanner sc = new Scanner(System.in,
   "Windows-1250");
```

```
System.out.println("Zadejte celé číslo
   v rozmezí 10-20 nebo 30-40":");
int a = Integer.parseInt(sc.nextLine());
if (((a >= 10) && (a <= 20))
   || ((a >=30) && (a <= 40)))
   System.out.println(
      "Zadal jsi správně");
else
         System.out.println(
      "Zadal jsi špatně");
```

Následující jednoduchý program rovněž demonstruje logické operátory. Uložte jej do souboru *Test.java*, zkompilujte a spusťte.

```
public class Test {
```

```
  public static void main(String args[]) {
     boolean a = true;
     boolean b = false;

     System.out.println("a && b = "
     + (a&&b));

     System.out.println("a || b = "
     + (a||b) );

     System.out.println("!(a && b) = "
     + !(a && b));
  }
}
```

Uvidíte následující výstup:

```
a && b = false
a || b = true
!(a && b) = true
```

Příklad operátorů přiřazení

Následující jednoduchý program demonstruje operátory přiřazení. Uložte si jej, zkompilujte, spusťte.

```
public class Test {

  public static void main(String args[]) {
     int a = 10;
     int b = 20;
```

```
        int c = 0;

        c = a + b;
        System.out.println("c = a + b = " + c
);

        c += a ;
        System.out.println("c += a  = " + c );

        c -= a ;
        System.out.println("c -= a = " + c );

        c *= a ;
        System.out.println("c *= a = " + c );
```

```
a = 10;
c = 15;
c /= a ;
System.out.println("c /= a = " + c );

a = 10;
c = 15;
c %= a ;
System.out.println("c %= a  = " + c );

c <<= 2 ;
System.out.println("c <<= 2 = " + c );

c >>= 2 ;
System.out.println("c >>= 2 = " + c );
```

```
        c >>= 2 ;
        System.out.println("c >>= a = "+ c );

        c &= a ;
        System.out.println("c &= 2 = "+ c );

        c ^= a ;
        System.out.println("c ^= a = "+ c );

        c |= a ;
        System.out.println("c |= a = "+ c );
    }
}
```

Uvidíte následující výstup:

```
c = a + b = 30
c += a  = 40
c -= a = 30
c *= a = 300
c /= a = 1
c %= a  = 5
c <<= 2 = 20
c >>= 2 = 5
c >>= 2 = 1
c &= a  = 0
c ^= a   = 10
c |= a   = 10
```

Rozhodování v Javě

V Javě jsou dva typy výrzů rozhodování.

- výrazy **if**
- výrazy **switch**

Příkaz if

Příkaz *if* se skládá z booleovského výrazu, který je následován jedním nebo více příkazy.

Syntaxe:

Syntaxe příkazu *if* je následující:

```
if(booleovský výraz)
{
   //Příkazy se provedou,
   //pokud se Booleovský výraz vyhodnotí
jako true.
```

```
}
```

Pokud se Booleovský výraz vyhodnotí jako true, pak se vykoná blok kódu uvnitř if výrazu. Pokud ne, bude vykonána první sada kódu na konci výrazu (za uzavírající složenou závorkou).

Příklad:

```
public class Test {

   public static void main(String args[]){
      int x = 10;

      if( x < 20 ){
         System.out.print
      ("This is if statement");
      }
```

```
    }
}
```

Toto bude mít následující výstup:

```
This is if statement
```

Příkaz if...else

Příkaz *if* může být následován nepovinným výrazem *else*, který se spustí v případě, že výraz vyhodnotí jako false.

Nesprávné řešení

Ukažme si nejprve příklad špatného řešení. Mějme spočítat odmocninu. V následujícím příkladu si nezapomeňte naimportovat *java.util.Scanner*, aby program znal třídu Scanner!

Také si naimportujte třídu Math, která obsahuje matematické funkce včetně druhé odmocniny, *sqrt()*.

```
import java.lang.Math;
Scanner sc = new Scanner(System.in,
      "Windows-1250");
System.out.println("Zadej nějaké číslo, "+
   "ze kterého spočítám odmocninu:");
int a = Integer.parseInt(sc.nextLine());
if (a > 0){
         System.out.println(
      "Zadal jsi číslo větší než 0, "
      +" tedy ho lze odmocnit");
         double o = Math.sqrt(a);
         System.out.println("Odmocnina z
čísla "
```

```
    + a + " je " + o);
}
System.out.println("Děkuji za zadání");
```

Snadno nahlédneme, že algoritmus je nesprávný: pro správnou funkci odmocniny bychom měli ošetřit všechny varianty, třeba takhle:

```
if (a > 0){
        System.out.println("Zadej nějaké "+
   "číslo, ze kterého spočítám "+
   "odmocninu:");
        double o = Math.sqrt(a);
        System.out.println("Odmocnina "+
   "z čísla " + a + " je " + o);
}
```

```
if (a < 0){
        System.out.println("Odmocnina ze
záporného "+
   "čísla neexistuje!");
   }
System.out.println("Děkuji za zadání");
```

Kód můžeme výrazně zjednodušit pomocí klíčového slova **else**, které vykoná následující příkaz nebo blok příkazů v případě, že je podmínka nepravdivá. Kód je mnohem přehlednější a nemusíme vymýšlet opačnou podmínku, což by v případě složené podmínky mohlo být někdy i dosti obtížné.

Syntaxe:

Syntaxe příkazu if...else je:

```
if(booleovský výraz){
   //Vykoná se, pokud vyjde podmínka
   // jako true
}else{
   //Vyhodnotí se, pokud
   // vyjde podmínka jako false
}
```

V případě více příkazů by byl za *else* opět blok ze složených závorek **{ }**.

Příklad:

```
public class Test {

   public static void main(String args[]){
      int x = 30;
```

```
        if( x < 20 ){
              System.out.print("Toto je IF");
        }else{
           System.out.print("Toto je ELSE");
        }
    }
}
```

Toto by mělo následující výstup:

```
Toto je ELSE
```

Příkaz s *else* se také využívá v případě, kdy potřebujeme v příkazu manipulovat s proměnnou z podmínky a nemůžeme se na ni tedy ptát potom znovu.

Program si sám pamatuje, že se podmínka nesplnila a přejde do sekce *else*. Ukažme si to na příkladu: Mějme číslo a, kde bude hodnota 0 nebo 1 a po nás se bude chtít, abychom hodnotu prohodili (pokud tam je 0, dáme tam 1, pokud 1, dáme tam 0).

Schválně si to zkuste.

Naivně bychom mohli kód napsat takto:

```
int a = 0; // do a si přiřadíme na začátku
0
if (a == 0) // pokud a==0, dáme a=1
        a = 1;
if (a == 1) // pokud a==1, dáme a=0
        a = 0;
System.out.println(a);
```

ale to samozřejmě nefunguje.

Na začátku máme v **a** nulu, první podmínka se jistě splní a dosadí do **a** jedničku.

Jenže ale tím se splní i ta druhá.

Když podmínky otočíme, budeme mít ten samý problém s jedničkou.

Raději použijeme **else**:

```
int a = 0; // do a si přiřadíme 0

if (a == 0) // pokud a==0, přiřadíme 1
```

```
        a = 1;
else // pokud a==1, přiřadíme 0
        a = 0;

System.out.println(a);
```

Příkaz if...else if...else

Příkaz *if* může být následován volitelným příkazem *else if...else*, který je velice užitečný k testování složitých podmínek.

Když používáme příkazy *if, else if, else*, tak musíme pamatovat na pár věcí.

- If může mít nula nebo jeden else, a to musí přijít po jakýcholiv příkazech else if.
- If může mít nula až mnoho příkazů *else if*, a ty musejí přijít před posledním *else*.
- Jakmile uspěje nějaký else if, nebude už vyhodnocován žádný další *else if* ani *else*.

Synaxe:

Synaxe *if...else* je následující:

```
if(Booleovský výraz1){
   //Provede se, pokud je
   /Booleovský výraz1 true
}else if(Booleovský výraz2){
```

```
    // Provede se, pokud je
    //Booleovský výraz2 true
}else if(){
    // Provede se, pokud je
    //Booleovský výraz3 true
}else {
    //Provede se, pokud žádná
    //z výše vypsaných podmínek není true
}
```

Příklad:

```
public class Test {

    public static void main(String args[]){
        int x = 30;
```

```
     if( x == 10 ){
        System.out.print(
    "
    Hodnota X je 10");
     }else if( x == 20 ){
        System.out.print(
    "
    Hodnota X je 20");
     }else if( x == 30 ){
        System.out.print("
    Hodnota X je 30");
     }else{
              System.out.print("Toto je
ELSE");
     }
```

```
    }
}
```

Toto by mělo následující výstup:

```
Hodnota X je 30
```

Vnořený příkaz if...else

Je vždy legální vnořovat příkazy *if-else*, což znamená, že můžete použít jeden příkaz *if else* uvnitř jiného *if* nebo *else if* příkazu.

Syntaxe:

Syntaxe pro vnořený *if...else* je následující:

```
if(Booleovský výraz1){
   // Provede se, pokud je Booleovský
výraz1 true
```

```
   if(Booleovský výraz2){
      // Provede se, pokud je Booleovský
výraz2 true
   }
}
```

Příkaz *else if...else* lze vnořovat podobně jako příkaz *if*.

Příklad:

```
public class Test {

   public static void main(String args[]){
      int x = 30;
      int y = 10;

      if( x == 30 ){
```

```
            if( y == 10 ){
                System.out.print("X = 30 and Y
= 10");
             }
          }
      }
}
```

Toto by mělo následující výsledek:

```
X = 30 and Y = 10
```

Příkaz switch

Switch je konstrukce, převzatá z jazyka C (jako většina gramatiky Javy). Umožňuje nám zjednodušit zápis více podmínek pod sebou.

Příkaz *switch* umoňuje, aby byla otestována rovnost oproti seznamu podmínek. Každá hodnota se nazývá case, a podímnka příkazu switch je testována pro každou z nich.

Syntaxe:

Synaxe příkazu switch je následující:

```
switch(expression){
    case value :
       //Statements
       break; //optional
    case value :
       //Statements
       break; //optional
    //You can have any number
```

```
    // of case statements.
    default : //Optional
       //Statements
}
```

K příkazu *switch* se vážou následující pravidla:

- Proměnná použitá v příkazu *switch* může být jenom byte, short, int, nebo char
- Můžete mít libovolný počet výrazů *case* uvnitř příkazu *switch*. Každý *case* je následován hodnotou, se kterou se porovnává, a dvojtečkou.
- Hodnota pro *case* musí být stejného datového typu jako proměnná v příkazu *switch*, a musí to být konstanta nebo literál.

- Pokud se hodnota podmínky pro *switch* rovná hodnotě v case, budou se vykonávat příkazy za *case* až do doby, dokud program nenarazí na příkaz *break*.
- Když je dosaženo výrazu *break*, bude *switch* ukončen, a program přeskočí na řádku následující po příkazu *switch*.

Ne každý case potřebuje break. Pokud program na žádný break nenarazí, *pokračuje prováděním* následujících příkazů case tak dlouho, dokud nebude dosaženo příkazu break anebo dokud se neukončí blok *switch*.

Příkaz *switch* může mít nepovinný defaultní případ, který se musí objevit až na konci příkazu *switch*. Defaultní případ může

být použit pro vykonávání úkolu, pokud není ani jeden case pravdivý.

Příklad:

```
public class Test {

   public static void main(String args[]){
      //char grade = args[0].charAt(0);
      char grade = 'C';

      switch(grade)
      {
         case 'A' :

System.out.println("Vynikající!");
```

```
            break;
         case 'B' :
         case 'C' :
            System.out.println("Dost
dobře");
            break;
         case 'D' :

System.out.println("Dostatečné");
         case 'F' :
            System.out.println(
          "Zkus to znovu");
            break;
         default :
            System.out.println(
```

```
        "Nevalidní známka");
      }
      System.out.println(
        "Tvoje hodnocení je " + grade);
  }
}
```

Zkomilujte a spusťte si program nahoře za použití různých argumentů v příkazové řádce. Dostanete následující výsledek:

```
Dost dobře
Tvoje hodnocení je C
```

Příklad:

Ukážme si to ještě na dalším příkladu. Představme si kalkulačku, která načetla 2 čísla a vypočítá všechny 4 operace. Nyní si ale budeme chtít zvolit, kterou operaci chceme:

```
System.out.println("Vítejte v kalkulačce");
System.out.println("Zadejte první číslo:");
float a = Float.parseFloat(sc.nextLine());
System.out.print("Zadejte druhé číslo:");
float b = Float.parseFloat(sc.nextLine());
System.out.println("Zvolte si operaci:");
System.out.println("1 - sčítání");
System.out.println("2 - odčítání");
System.out.println("3 - násobení");
System.out.println("4 - dělení");
```

```
int volba =
Integer.parseInt(sc.nextLine());
float vysledek = 0;
switch (volba)
{
        case 1: vysledek = a + b;
        break;
        case 2: vysledek = a - b;
        break;
        case 3: vysledek = a * b;
        break;
        case 4: vysledek = a / b;
        break;
}
if ((volba > 0) && (volba < 5))
```

```
        System.out.printf("Výsledek: %f",
    vysledek);
else
        System.out.println("Neplatná
volba");
```

Všimněte si, že jsme proměnnou *výsledek* deklarovali jen na začátku, jen tak do ní můžeme potom přiřazovat. Kdybychom ji deklarovali u každého přiřazení, Java by kód nezkompilovala a vyhodila chybu redeklarace proměnné!

Důležité je také přiřadit výsledku nějakou výchozí hodnotu, zde nula, jinak by vznikla chyba - snažíme vypsat proměnnou, která nebyla jednoznačně inicializována.

Tento program by v tomto případě fungoval stejně i bez těch else, ale nač se dále ptát, když již máme výsledek.

Cykly – for, while a do...while v Javě

Může nastat situace, kdy budeme potřebovat spustit blok kódu několikrát po sobě, čemuž se často říká cyklus.

Jak již slovo cyklus napoví, že se něco bude opakovat. Když chceme v programu něco udělat 100x, jistě nebudeme psát pod sebe 100x ten samý kód, ale vložíme ho do cyklu.

Java má tři velice flexibilní mechanismy smyček. Můžete použít jednu z následujících tří cyklů:

- Cyklus **while**
- Cyklus **do...while**
- Cyklus **for**
- Java 5 zavedla *pokročilý cyklus for*. Je používán především pro práci s polem.

Cyklus while

Cyklus while je řídicí struktura, která umožňuje zopakovat úkol n-krát po sobě.

Syntaxe:

Syntaxe cyklu while je následující:

```
while(Booleovský výraz)
{
    //Příkazy
}
```

Pomocí vývojového diagramu bychom mohli cyklus while zobrazit takto.

Při spuštění budou vykonány příkazy uvnitř smyčky, pokud je *Booleovský výraz* pravdivý. Bude tomu tak dlouho, dokud je podmínka pravdivá.

Klíčovou vlastností smyčky *while* je, že nemusí proběhnout ani jednou. Pokud je výraz otestován a vyjde jako false, pak se tělo smyčky přeskočí na první příkaz za smyčkou.

Příklad:

```
public class Test {

   public static void main(String args[]) {
      int x = 10;

      while( x < 20 ) {
         System.out.print(" : " + x );
         x++;
         System.out.print("\n");
      }
```

```
    }
}
```

Toto by mělo následující výstup:

```
hodnota x : 10
hodnota x : 11
hodnota x : 12
hodnota x : 13
hodnota x : 14
hodnota x : 15
hodnota x : 16
hodnota x : 17
hodnota x : 18
hodnota x : 19
```

Jiný příklad

Náš příklad s kalkulačkou můžeme vylepšit tak, že výpočet se bude opakovat, dokud nezadáme “konec“.

Začátek upravíme takto:

```
Scanner sc = new Scanner(System.in,
      "Windows-1250");
System.out.println("Vítejte v kalkulačce");
String pokracovat = "ano";
while (pokracovat.equals("ano"))
{
….. tady bude původní kód kalkulačky …..
```

a konec změníme takto:

```
System.out.println("Přejete si zadat další
příklad? [ano/ne] ");
```

```
        pokracovat = sc.nextLine();
}  // konec cyklu while

System.out.println("Děkuji za použití "+
    "kalkulačky. ");
```

Cyklus do...while

Cyklus do...while je podobná smyčce while, s tím rozdílem, že cyklus do...while zaručeně proběhne alespoň jednou.

Syntaxe:

Syntaxe smyčky do...while je:

```
do
```

```
{
   //Příkazy
}while(Booleovský výraz);
```

Pomocí vývojového diagramu bychom mohli cyklus do..while zobrazit takto.

Povšimněte si, že se Booleovský výraz objeví na konci smyčky, takže se příkazy uvnitř smyčky poprvé vykonají ještě dříve, než je vyhodnocena podmínka.

Pokud je Booleovská podmínka vyhodnocena jako true, pak program skočí zpět na do, a příkazy ve smyčce se vykonají znovu. Tento proces se opakuje tak dlouho, dokud nevyjde Booleovská podmínka jako false.

Dobrá rada

Dobrá rada: slovo **while** vždycky pište na stejný řádek jako uzavírací svorku **}**. Kdybyste ho psali na samostatný řádek, tak byste se po čase mohli divit, proč tam je cyklus while bez těla s příkazy.

Příklad:

```
public class Test {

   public static void main(String args[]){
      int x = 10;
```

```
        do{
            System.out.print(" : " + x );
            x++;
            System.out.print("\n");
        }while( x < 20 );
    }
}
```

Toto bude mít následující výstup:

```
 : 10
 : 11
 : 12
 : 13
 : 14
 : 15
 : 16
```

```
: 17
: 18
: 19
```

Smyčka *for*

Smyčka for je řídicí struktura pro opakování. Umožňuje efektivní zápis smyčky, která musí být vyhodnocena n-krát.

Smyčka for je užitečná, pokud víte, kolikrát se bude úkol opakovat.

Synaxe:

Syntaxe smyčky for je:

```
for(inicializace; Booleovský výraz;
aktualizace)
```

```
{
    //Příkazy
}
```

Zde je vývojový diagram smyčky for:

Inicializační krok je spuštěn jako první, a jen jednou. Tento krok vám umožňuje deklarovat a inicializovat jakékoliv kontrolní proměnné cyklu. Není nutné zadat žádný příkaz, ale středník na konci je povinný.

Pak je vyhodnocena Booleovská podmínka. Pokud je pravdivá, je provedeno tělo cyklu. Pokud je nepravdivá, pak se tělo cyklu nespustí a program skočí na první příkaz za cyklem for.

Poté, co je vykonáno tělo cyklu for, skočí program zpět na příkaz aktualizace. Tento příkaz nám umožňuje aktualizovat jakékoliv řídicí proměnné cyklu. Tento příkaz může být vynechán, avšak středník za Booleovským výrazem je povinný.

Booleovský výraz je nyní opět vyhodnocen. Pokud je pravdivý, pak se cyklus spustí a proces se opakuje (tělo cyklu, aktualizace, pak Booleovský výraz). Poté, co Booleovský výraz vyjde jako false, cyklus skončí.

Příklad:

```
public class Test {

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

```
        for(int x = 10; x < 20; x = x+1) {
            System.out.print("hodnota x : " + x );
            System.out.print("\n");
        }
    }
}
```

Toto by mělo následující výstup:

```
: 10
: 11
: 12
: 13
: 14
: 15
: 16
```

```
: 17
: 18
: 19
```

Cyklus proběhne 10x, zpočátku je v proměnné x desítka, cyklus vypíše hodnotu a zvýší proměnnou x o jedna. Poté běží stejně s dalšími čísly.

Jakmile je v 3 dvacítka, již nesouhlasí podmínka `x < 20` a cyklus skončí.

O vynechávání složených závorek platí to samé, co u podmínek. V tomto případě tam nemusí být, protože cyklus spouští pouze jediný příkaz.

Vložené cykly *for*

Nic nám nebrání, vložit dva nebo více cyklů do sebe:

```
System.out.println("Malá násobilka "+
   "pomocí dvou vložených cyklů:");
for (int j = 1; j <= 10; j++)
{
        for (int i = 1; i <= 10; i++)

System.out.printf("%d*%d=%d",
          i, j, i*j);
        System.out.println();  //odřádkovat
}
```

Výstupem bude tabulka malé násobilky.

Jiný příklad *for*

Napíšeme program, který bude umět vypočítat libovolnou (celou) mocninu libovolného čísla.

Přitom a^n spočítáme tak, že n-1 krát vynásobíme číslo **a** číslem **a**. Výsledek si samozřejmě musíme ukládat do proměnné.

```
Scanner sc = new Scanner(System.in,
"Windows-1250");

System.out.println("Mocninátor");
System.out.println("==========");
System.out.println("Zadejte základ mocniny:
");
int a = Integer.parseInt(sc.nextLine());
```

```
System.out.println("Zadejte exponent: ");
int n = Integer.parseInt(sc.nextLine());

int vysledek = a;
for (int i = 0; i < (n - 1); i++)
        vysledek = vysledek * a;

System.out.printf("Výsledek: %d",
vysledek);
System.out.println("Děkuji za použití
mocninátoru");
```

Odstrašující příklad

// tento kód je špatně

```
for (int i = 1; i <= 10; i++)
```

```
            i = 1;
```

Vidíme, že program se zasekl. Cyklus stále inkrementuje proměnnou i, ale ta se vždy sníží na 1. Nikdy tedy nedosáhne hodnoty > 10, cyklus nikdy neskončí. Program zastavíme tlačítkem Stop u okna konzole

Rozšířený cyklus *for* v Javě

Rozšířená smyčka for byla zavedena v Javě 5. Je používaná především pro pole.

Syntaxe:

Syntaxe rozšíření for smyčky:

```
for(deklarace : výraz)
```

```
{
    //Příkazy
}
```

Deklarace: Nově deklarovaná bloková proměnná, která je typově kompatibilní s prvky pole, ke kterému přistupujete. Proměnná bude dostupná uvnitř bloku *for* a její hodnota bude stejná jako stávající prvek pole.

Výraz: Toto vyhodnotí pole, přes které iterujeme. Výraz může být proměnná pole nebo volání metody, která vrací pole.

Příklad:

```
public class Test {
   public static void main(String args[]){
      int [] cisla = {10,20,30,40,50};
```

```
        for(int x : cisla ){
            System.out.print( x );
            System.out.print(",");
        }
        System.out.print("\n");
        String [] jmena =
       {"James", "Larry", "Tom", "Lacy"};
        for( String jmeno : jmena ) {
            System.out.print( jmeno );
            System.out.print(",");
        }
    }
}
```

Toto by mělo následující výstup:

```
10,20,30,40,50,
```

```
James,Larry,Tom,Lacy,
```

Klíčové slovo *break*

Klíčové slovo *break* se používá k přerušení cyklu. Klíčové slovo *break* musí být použito uvnitř libovolného cyklu nebo v příkazu *switch*.

Klíčové slovo *break* zastaví provádění nejvnitřnější smyčky a začne vyhodnocovat další řádku kódu za blokem.

Syntaxe:

Syntaxe *break* je jediný příkaz uvniř libovolné smyčky.

```
break;
```

Příklad:

```
public class Test {

   public static void main(String args[]) {
      int [] cisla = {10, 20, 30, 40, 50};

      for(int x : cisla ) {
         if( x == 30 ) {
        break;
         }
         System.out.print( x );
         System.out.print("\n");
      }
   }
}
```

Toto by mělo následující výstup:

```
10
20
```

Klíčové slovo *continue*

Klíčové slovo *continue* může být použito v libovlné kontrolní smyčkové struktuře. Způsobuje, že smyčka okamžitě skočí na další iteraci smyčky.

Ve smyčce for způsobuje *continue* okamžité přeskočení programu na výrok aktualizace.

Ve smyčkách while a do...while program okamžitě přeskočí na Booleovský výraz.

Syntaxe:

Syntaxe *continue* je jediný příkaz uvnitř libovolné smyčky:

```
continue;
```

Příklad:

```
public class Test {

   public static void main(String args[]) {
      int [] cisla = {10, 20, 30, 40, 50};

      for(int x : cisla ) {
         if( x == 30 ) {
         continue;
         }
         System.out.print( x );
```

```
            System.out.print("\n");
        }
    }
}
```

Toto by mělo následující výstup:

```
10
20
40
50
```

Třída Numbers

Když pracujeme s čísly, obvykle používáme **primitivní datové typy** jako byte, int, long, double atd.

Příklad:

```
int i = 5000;
float gpa = 13.65;
byte mask = 0xaf;
```

Nicméně při vývoji se občas dostaneme do situace, kdy potřebujeme použít objekty namísto primitivních datových typů. Kvůli tomu nám Java poskytuje pro každý primitivní datový typ **obalovací třídy**.

Veškeré obalovací třídy (Integer, Byte, Double, Float, Short) jsou podtřídami abstraktní třídy Number.

O toto zaobalení se stará kompilátor, procesu se říká zabalení. Takže je-li vyžadován objekt a je použito primitivního typu,

kompilátor **zabalí** primitivní typ do své obalové třídy. Podobně kompilátor **rozbalí** objekt do primitivního typu. **Number** je část balíku *java.lang*.

Zde je příklad zabalení a rozbalení:

```
public class Test{

   public static void main(String args[]){
   // zabalí int do objektu Integer
        Integer x = 5;

   // rozbalí Integer do int
        x =  x + 10;
   System.out.println(x);
    }
```

```
}
```

Toto by mělo následující výstup:

```
15
```

Když je do x přiřazena celočíselná hodnota, kompilátor zabalí celé číslo, protože x je celočíselný objekt. Později je x rozbaleno tak, aby mohlo být sečteno jako celé číslo.

Metody Number:

Zde je seznam metod instance, které jsou všechny podtřídami, které implementuje třída Number:

SN	Medody s popisem
1	**xxxValue()** Konvertuje hodnotu *this* objektu třídy Number na xxx datový

	typ a vrací jej.
2	**compareTo()** Porovnává this objekt třídyNumber k parametru.
3	**equals()** Rozhoduje, zda je this objekt třídy Number rovno parametru.
4	**valueOf()** Vrací objekt Integer s hodnoutou specifikovanou v primitivním typu.
5	**toString()** Vrací objekt typu String reprezentující hodnotu specifikovaného int nebo Integer.
6	**parseInt()** Tato metoda se používá ke získání primitivního datového

	typu určitého Stringu.
7	**abs()** Navrací absolutní hodnotu parametru.
8	**ceil()** Vrací nejmenší celé číslo, které větší nebo rovno parametru. Návratový typ je double.
9	**floor()** Vrací největší celé číslo, které je menší nebo rovno parametru. Navrací double.
10	**rint()** Vrací celé číslo, které je hodnotě v parametru nejblíž. Navrací double.
11	**round()** Navrací nejbližší long nebo int, jak je indikováno v

	návratovém typu hodnoty, k parametru.
12	**min()** Navrací menší ze dvou parametrů.
13	**max()** Navrací větší ze dnou parametrů.
14	**exp()** Navrací základ přirozených logaritmů, e, umocněné na hodnotu v parametru.
15	**log()** Navrací přirozený logaritmus parametru.
16	**pow()** Navrací hodnotu prvního parametru umocněnou druhým parametrem.
17	**sqrt()**

	Navrací odmocninu parametru.
18	**sin()** Navrací sinus specifikované hodnoty typu double.
19	**cos()** Navrací cosinus specifikovné hodnoty typu double.
20	**tan()** Navrací tangens specifikované hodnoty typu double.
21	**asin()** Navrací arcussinus specifikované hodnoty typu double
22	**acos()** Navrací arcuscosinus specifikované hodnoty typu double
23	**atan()** Navrací arcustangens specifikované hodnoty typu double
24	**atan2()**

	Konvertuje kartézské souřadnice (x, y) na polární (r, theta) a vrací theta
25	**toDegrees()** Konvertuje parametr na stupně
26	**toRadians()** Konvertuje parametr na radiany
27	**random()** Navrací náhodné číslo.

Třída Character

Když běžně pracujeme s čísly, používáme datových typů char.

Příklad:

```
char ch = 'a';

// Unikódový znak pro řeckou omegu
char uniChar = '\u039A';

// pole písmen
char[] charArray ={'a', 'b', 'c', 'd',
'e'};
```

Nicméně ve vývoji se dostáváme do situací, kdy potřebujeme objekty namísto primitivních datových typů. Abychom toho

dosáhli, poskytuje nám Java pro primitivní datový typ char obalující třídu **Character**.

Třída character nám nabízí řadu užitečných metod třídy (např. static) pro manipulaci s písmeny. Můžte vytvořit objekt Character za použití konstruktoru Character:

```
Character ch = new Character('a');
```

Kompilátor Javy za vás za určitých podmínek objekt Character vytvoří. Například pokud budete předávat char do metody, která očekává objekt, pak za vás kompilátor automaticky zkonvertuje char na Character. Této vlastnosti se říká autoboxing a unboxing v případě, že se jedná o konverzi druhým směrem.

Příklad:

```
// Zde je ‘a’ primitivního typu char
// zabaleno do objektu ch typu Character
Character ch = 'a';


// Zde je primitivní 'x' zabaleno
// kvůli metodě test a návratová hodnota
// zase je zabalena do c
char c = test('x');
```

Escape sekvence:

Písmeno za zpětným lomítkem (\) je *escape sekvence* a pro kompilátor má speciální význam.

Písmeno nová řádka (\n) jsme už mnohokrát použili v příkazech System.out.println(), abychom se po vypsání stringu dostali na novou řádku.

Následující tabulka nám ukazuje escape sekvence jazyku Java:

Escape Sekvence	Popis
\t	Vložení tabelátoru na toto místo.
\b	Vložení backspace na toto místo.
\n	Vložení nové řádky na toto místo.
\r	Vložení návratu vozíku na toto místo.
\f	Vložení form feed do textu na toto místo.
\'	Vložení apostrofu na toto místo.
\"	Vložení uvozovky na toto místo.
****	Vložení zpětného lomítka na toto místo.

Když je v příkazu print dosaženo escapovací sekvence, interpretuje ji kompilátor.

Příklad:

Pokud chcete umístit uvozovky mezi uvozovky, musíte použít escapovací sekvence \" na vnitřních uvozovkách.

```
public class Test {

   public static void main(String args[]) {
      System.out.println(
   "Řekla mi \"ahoj\".");
   }
```

```
}
```

Toto by mělo následující výstup:

```
Řekla mi „ahoj“.
```

Metody znaků

Zde je seznam některých důležitých metod instancí, které využívají podtřídy třídy Character:

SN	Metody s popisem
1	**isLetter()** Rozhoduje, zda je specifikovaná hodnota písmeno.
2	**isDigit()** Rozhoduje, zda j specifikovaná hodnota číslice.
3	**isWhitespace()** Rozhoduje, zda je specifikovaná hodonota bílým znakem.

4	**isUpperCase()** Rozhoduje, zda je specifikovaná hodnota velkým písmem.
5	**isLowerCase()** Rozhoduje, zda je specifikovaná hodnota malým písmem.
6	**toUpperCase()** Navrací specifikovanou hodnotu velkým písmem.
7	**toLowerCase()** Navrací specifikovanou hodnotu malým písmem.
8	**toString()** Navrací objekt String reprezentující specifikovanou hodnotu, tj. Řetězec o jednom znaku.

Třída řetězec

Řetězce, které se při programování v Javě vyskytují velice často, jsou posloupností znaků. V programovacím jazyku Java jsou řetězce **objekty**.

Platforma Java poskytuje třídu String k vytvoření a práci s řetězci.

Vytváření řetězců:

Nejpřímější cesta k vytvoření řetězce je zápisem:

```
String greeting = "Hello world!";
```

Kdykoliv kompilátor ve vašem kódu narazí na řetězcový literál, vytvoří objekt String, v tomto případě s hodnotoou “Hello word!”.

Stejně jako jiné objekty tvoříte i řetězce za použití klíčového slova new a konstruktoru. Třída Sting má jedenáct konstruktorů, které vám umožňují zadat počáteční hodnotu z různých zdrojů, např. pole znaků.

```
public class StringDemo{

   public static void main(String args[]){
      char[] helloArray =
     { 'h', 'e', 'l', 'l', 'o', '.'};
      String helloString =
```

```
        new String(helloArray);
          System.out.println( helloString );
    }
}
```

Toto by mělo následující výstup:

```
hello.
```

Poznámka: *Třída String je neměnná, takže jednou vytvořený řetězec není možné dále měnit. Pokud je nutné dělat řadu změn v řetězcích či znacích, měli byste tak učinit přes třídy String Buffer a String Builder.*

Délka řetězce:

Metody používané ke získání informací o objektu jsou známy jako metody přístupu. Jedna z metod přístupu, které můžete

použít, je metoda *length()*, která navrací počet písmen v objektu string.

Poté, co byly spuštěny následující řádky kódu, byla hodnota len rovna 17:

```
public class StringDemo {

   public static void main(String args[]) {
      String palindrome =
     "Dot saw I was Tod";
      int len = palindrome.length();
      System.out.println
     ("String Length is : " + len);
   }
}
```

Toto by mělo následující výstup:

```
String Length is : 17
```

Zřetězení (konkatenace) řetězců

Třída String v sobě zahrnuje metodu pro zřetězení dvou řetězců:

```
string1.concat(string2);
```

Tato metoda navrací nový řetězec, který má hodnotu *string1* a na konci připojenou hodnotu *string2*. Metodu *concat()* můžete rovněž použít s literálem, tak jako v následujcím příkladu:

```
"My name is ".concat("Zara");
```

Řetězce se běžněji spojují pomocí operátoru **+**, např:

```
"Hello," + " world" + "!"
```

což má následující výstup:

```
"Hello, world!"
```

Pojďme se podívat na následující příklad:

```
public class StringDemo {

    public static void main(String args[]) {
        String string1 = "má malý ";
        System.out.println("Kobyla "
    + string1 + "bok.");
    }
}
```

Toto by mělo následující výstup

```
Kobyla má malý bok.
```

Vytváření formátovacích řetězců

Můžete použít metody *printf()* a *format()* k výstupu formátovaných čísel. Třída String má ekvivalentní metodu *format()*, která navrací objekt String namísto objektu PrintStream.

Použitím statické metody pro String *format()* vytvoříte formátovací řetězec, který můžete opakovaně používat, narozdíl od příkazu *print()*, který je jednorázový. Například namísto:

```
System.out.printf(
  "The value of the float variable is " +
  "%f, while the value of the integer " +
  "variable is %d, and the string " +
```

```
  "is %s", floatVar, intVar, stringVar);
```

můžete napsat:

```
String fs;
fs = String.format(
  "The value of the float variable is " +
  "%f, while the value of the integer " +
  "variable is %d, and the string " +
  "is %s", floatVar, intVar, stringVar);
System.out.println(fs);
System.out.println(fs);
System.out.println(fs);
```

Speciální znaky a scapování

Vidíme, že řetězec může obsahovat speciální znaky, které jsou předsazené zpětným lomítkem "\". Je to zejména znak \n, který kdekoli v textu způsobí odřádkování a poté \t, kde se jedná o tabulátor. Pojďme si to vyzkoušet:

```
System.out.println("První řádka\nDruhá řádka");
```

Znak "\" označuje nějakou speciální sekvenci znaků v řetězci a je dále využíván např. k psaní unicode znaku jako "\uxxxx", kde xxxx je kód znaku.

Problém může nastat ve chvíli, kdy chceme napsat samotné "\", musíme ho tzv. odescapovat:

```
System.out.println("Toto je zpětné lomítko: \\");
```

Stejným způsobem můžeme odescapovat např. uvozovku tak, aby ji Java nechápala jako konec řetězce:

```
System.out.println("Toto je uvozovka: \"");
```

Vstupy z konzole a polí v okenních aplikacích se samozřejmě escapují samy, aby uživatel nemohl zadat \n a podobně. V kódu to má programátor povoleno a musí na to myslet.

Metody řetězců

Zde je seznam metod podporovaných třídou String

SN	Metody s popisem
1	**char charAt(int index)** Navrací znak se specifikovaným indexem
2	**int compareTo(Object o)** Porovnává tento String s jiným Objektem
3	**int compareTo(String anotherString)** Lexikograficky porovnává dva řetězce
4	**int compareToIgnoreCase(String str)** Lexikograficky porovnává dva řetězce, ignoruje rozdíly ve velikosti písmen
5	**String concat(String str)** Zřetězuje specifikovaný řetězec na konec tohoto řetězce.
6	**boolean contentEquals(StringBuffer sb)** Navrací true jedině a právě tehdy, když tento String

	reprezentuje stejnou řadu znaků jako specifikovaný StringBuffer.
7	**static String copyValueOf(char[] data)** Navrací řetězec, který reprezentuje řadu znaků ve specifikovaném poli.
8	**static String copyValueOf(char[] data, int offset, int count)** Navrací řetězec, který reprezentuje řadu znaků ve specifikovaném poli, count znaků od pozice offset.
9	**boolean endsWith(String suffix)** Testuje, zda je řetězec zakončen specifikovanou koncovkou.
10	**boolean equals(Object anObject)** Porovnává tento řetězec se specifikovaným objektem.
11	**boolean equalsIgnoreCase(String anotherString)** Porovnává tento řetězec s jiným řetězcem, ignoruje rozdíly

	ve velikosti písmen.
12	**byte[] getBytes()** Zakóduje tento String do sekvence bytů za použití defaultní znakové řady platformy, a uchovává výsledek v novém bytovém poli.
13	**byte[] getBytes(String charsetName)** Zakóduje tento String do sekvence bytů za použití pojmenované znakové řady platformy, a uchovává výsledek v novém bytovém poli.
14	**void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)** Kopíruje znaky z tohoto řetězce do cílového pole znaků.
15	**int hashCode()** Navrací hašovací kód tohoto řetězce.

16	**int indexOf(int ch)** Navrací v rámci tohoto řetězce index, kde bylo poprvé dosaženo specifikovaného znaku.
17	**int indexOf(int ch, int fromIndex)** Navrací v rámci tohoto řetězce index, kde bylo poprvé dosaženo specifikovaného znaku, hledat začíná na specifikovaném indexu.
18	**int indexOf(String str)** Navrací index v rámci tohoto stringu, pod kterým byla poprvé nalezena specifikovaná část řetězce.
19	**int indexOf(String str, int fromIndex)** Navrací index v rámci tohoto stringu, pod kterým byla poprvé nalezena specifikovaná část řetězce , hledat začíná na specifikovaném indexu.

20	**String intern()** Navrací kanonickou reprezentaci objektu řetězec.
21	**int lastIndexOf(int ch)** Navrací index v rámci řetězce, u nějž byl poslední výskyt specifikovaného znaku.
22	**int lastIndexOf(int ch, int fromIndex)** Navrací index v rámci řetězce, u nějž byl poslední výskyt specifikovaného znaku, prohledává odzadu a začíná u specifikovaného indexu.
23	**int lastIndexOf(String str)** Navrací index v rámci řetězce, u nějž byl zaznamenaný poslední výskyt podřetězce.
24	**int lastIndexOf(String str, int fromIndex)** Navrací index v rámci řetězce s posledním výskytem

	specifikovaného podřetězce, hledá odzadu a začíná na specifikovaném indexu.
25	**int length()** Navrací délku tohoto řetězce.
26	**boolean matches(String regex)** Rozhoduje, zda řetězec odpovídá danému regulárnímu výrazu.
27	**boolean regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len)** Testuje, zda jsou dvě oblasti řetězců shodné.
28	**boolean regionMatches(int toffset, String other, int ooffset, int len)** Testuje, zda jsou dvě oblasti řetězců shodné.
29	**String replace(char oldChar, char newChar)**

	Navrací nový řetězec vzniklý náhradou všech výskytů řetězce staryRetezec za řetězec novyRetezec.
30	**String replaceAll(String regex, String replacement** Nahrazuje každý podřetězec tohoto řetězce, který odpovídá danému regulárnímu výrazu, danou náhradou.
31	**String replaceFirst(String regex, String replacement)** Nahrazuje každý podřetězec tohoto řetězce, který odpovídá danému regulárnímu výrazu, danou náhradou.
32	**String[] split(String regex)** Rozdělí řetězec okolo shody s daným regulárním výrazem.
33	**String[] split(String regex, int limit)** Rozdělí řetězec okolo shody s daným regulárním výrazem.
34	**boolean startsWith(String prefix)** Testuje, zda začíná tento řetězec danou předponou.

35	**boolean startsWith(String prefix, int toffset)** Testuje, zda začíná tento řetězec danou předponou, hledat začíná od indexu.
36	**CharSequence subSequence(int beginIndex, int endIndex)** Vrací novou řadou znaků, která je podmnožinou této řady.
37	**String substring(int beginIndex)** Navrací nový řetězec, který je podřetězcem tohoto řetězce.
38	**String substring(int beginIndex, int endIndex)** Navrací nový řetězec, který je podřetězcem tohoto řetězce.
39	**char[] toCharArray()** Konvertuje tento řetězec na nové pole znaků.
40	**String toLowerCase()** Konvertuje všechny znaky v tomto řetězci na malá písmena za použití pravidel defaultní lokalizace.

41	**String toLowerCase(Locale locale)** Konvertuje všechny znaky v tomto řetězci na malá písmena za použití pravidel specifikované lokalizace.
42	**String toString()** Tento objekt (který již je typu string!) je sám navrácen.
43	**String toUpperCase()** Konvertuje všchny znaky tohoto řetězce na velká písmena za použití pravidel defaultní lokalizace.
44	**String toUpperCase(Locale locale)** Konvertuje všchny znaky tohoto řetězce na velká písmena za použití pravidel specifikované lokalizace.
45	**String trim()** Navrací kopii řetězce s ořezanými bílými znaky vepředu i vzadu.

46 **static String valueOf(primitive data type x)**
Navrací řetězcovou reprezentaci předaného parametru datového typu.

Třídy String Buffer a String Builder

Třídy **StringBuffer** a **StringBuilder** se používají, když je nezbytné udělat hodně změn na písmenech řetězce.

Narozdíl od objektů String, mohou být objekty typu *StringBuffer* a *StringBuilder* modifikovány pořád dokolečka, aniž by za sebou zanechávaly spoustu nepoužitých objektů.

Třída *StringBuilder* byla představena v Javě 5. Hlavním rozdílem mezi *StringBuffer* a *StringBuilder* je ten, že metody *StringBuilderu* nejsou bezpečné pro vícevláknové procesy (=nejsou synchronizovány).

Je doporučováno používat **StringBuilder** všude, kde je to jenom možné, protože je rychlejší než StringBuffer. Nicméně, pokud je nezbytná bezpečnost vláken, je naší nejlepší volobou StringBuffer.

Příklad:

```
public class Test{

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

```
        StringBuffer sBuffer =
       new StringBuffer(" test");
        sBuffer.append(" String Buffer");
        System.ou.println(sBuffer);
    }
}
```

Toto by mělo následující výstup:

```
test String Buffer
```

Metody třídy StringBuffer

Zde je seznam důležitých metod podporovaných třídou StringBuffer:

SN	Metody s popisem
1	**public StringBuffer append(String s)**

	Updatuje hodnotu objektu, který metodu vyvolal.
2	**public StringBuffer reverse()** Tato metoda obrací hodnotud objektu StringBuffer, který tuto metodu zavolal.
3	**public delete(int start, int end)** Maže řetězec od indexu start po index end.
4	**public insert(int offset, int i)** Tato metoda vkládá řetězec na pozici udanou offsetem.
5	**replace(int start, int end, String str)** Tato metoda nahrazuje znaky v podřetězci tohoto StringBufferu znaky specifikovaného Stringu.

Zde je seznam dalších metod (s výjimkou metod set), které jsou velice podobné třídě String:

SN	Metody s popisem
1	**int capacity()** Navrací současnou kapacitu vyrovnávací paměti řetězce.
2	**char charAt(int index)** Vrací ten znak z posloupnosti znaků umístěné v bufferu, který leží na zadaném indexu..
3	**void ensureCapacity(int minimumCapacity)** Stará se o to, aby byla kapacita vyrovnávací paměti alespoň tak velká jako specifikované minimum.
4	**void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)**

	Znaky jsou kopírovány z vyrovnávací paměti řetězce do cílového pole znaků dst.
5	**int indexOf(String str)** Navrací index v rámci řetězce, na němž byl poprvé zaznamenán specifikovaný podřetězec.
6	**int indexOf(String str, int fromIndex)** Navrací index v rámci tohoto řetězce, na němž byl poprvé zaznamenán specifikovaný podřetězec, hledání začíná u specifikovaného indexu.
7	**int lastIndexOf(String str)** Navrací index tohoto řetězce s posledním výskytem specifikovaného podřetězce.
8	**int lastIndexOf(String str, int fromIndex)** Navrací index v rámci tohoto řetězce s posledním výskytem

	specifikovaného podřetězce.
9	**int length()** Navrací délku (počet znaků) vyrovnávací paměti tohoto řetězce.
10	**void setCharAt(int index, char ch)** Znak na specifikovaném indexu vyrovnávvací paměti tohoto řetězce je nastaven na ch.
11	**void setLength(int newLength)** Nastaví délku vyrovnávací paměti tohoto řetězce.
12	**CharSequence subSequence(int start, int end)** Navrací novou posloupnost znaků, která je podmnožinou této posloupnosti.
13	**String substring(int start)** Navrací nový řetězec, který obsahuje posloupnost znaků

	momentálně uložených ve vyrovnávací paměti řetězce. Tento podřetězec začíná na specifikovaném indexu a pokračuje až do konce vyrovnávací paměti řetězce.
14	**String substring(int start, int end)** Navrací nový řetězec, který obsahuje podposloupnost znaků, které jsou momentálně obsaženy ve vyrovnávací paměti řetězce.
15	**String toString()** Konvertuje na řetězec reprezentující data ve vyrovnávací paměti řetězce.

Pole

Všeobecně

Pokud potřebujeme uchovávat větší množství proměnných stejného typu, tento problém nám řeší pole.

Můžeme si ho představit jako řadu přihrádek, kde v každé máme uložený jeden prvek. Přihrádky jsou očíslované tzv. indexy, první má index 0.

Programovací jazyky se velmi liší v tom, jak s polem pracují.

V některých jazycích (zejména starších, kompilovaných) nebylo možné za běhu programu vytvořit pole s dynamickou velikostí (např. mu změnit velikost podle nějaké proměnné). Pole se muselo deklarovat s konstantní velikostí přímo ve zdrojovém kódu.

Toto se obcházelo tzv. pointery, specifickými datovými strukturami, což často vedlo k chybám při manuální správě paměti a k nestabilitě programu (např. v C++).

Naopak, některé interpretované jazyky umožňují nejen deklarovat pole s libovolnou velikostí, ale dokonce tuto velikost na již existujícím poli měnit (např. PHP).

My víme, že Java je virtuální stroj, tedy cosi mezi kompilerem a interpretem.

Proto můžeme pole založit s velikostí, kterou dynamicky zadáme až za běhu programu, **ale velikost existujícího pole modifikovat nemůžeme**.

Pole je jednoduché. Rychle se s ním pracuje, protože prvky jsou v paměti jednoduše uloženy za sebou, zabírají všechny stejně místa a rychle se k nim přistupuje.

Mnoho vnitřních funkčností v Javě proto nějak pracuje s polem nebo pole vrací. Je to klíčová struktura.

Java nám poskytuje datovou strukturu **array**, která uchovává sbírku prvků stejného typu řazených po sobě. Pole se používá k uchovávání sbírky dat, ale bývá užitečnější chápat je jako sbírku proměnných stejného typu.

Namísto deklarace jednotlivých proměnných, např. *cislo0*, *cislo1*, ..., až *cislo99*, můžete deklarovat jednu proměnnou typu pole, např. *cisla*, a použít k reprezentaci jednotlivých proměnných zápisu *cisla[0]*, *cisla[1]* ... až *cisla[99]*.

Deklarace proměnných typu pole

Abychom mohli v programu použít pole, musíme deklarovat proměnnou tak, aby se odkazovala na pole, a musíme specifikovat typ pole na něž může proměnná odkazovat.

```
dataType[] arrayRefVar;
// preferovaný způsob.
```

nebo

```
dataType arrayRefVar[];
// funguje, ale není to preferovaný způsob.
```

Poznámka: *Styl dataType[] arrayRefVar je preferovaný. typ dataType arrayRefVar[] pochází z C/C++, a byl přejat, aby vyhovoval programátorům C/C++.*

Příklad:

Následující úryvky kódu jsou příklady této syntaxe:

```
double[] myList;
// preferovaný způsob.
```

nebo

```
double myList[];
// funguje, ale není to preferovaný způsob.
```

Tvoření polí

Pole můžete vytvořit za použití operátru new s následující syntaxí:

```
arrayRefVar = new datovyTyp [velikostPole]
```

Zmíněný příklad udělá dvě věci:

- Vytvoří pole za použití *new datovyTyp[velikostPole]*
- Přiřadí odkaz na nově vytvořené pole do proměnné *arrayRefVar*.

Deklarování proměnné pole, vytvoření pole a přiřazení reference na pole do proměnné může být zkombinováno do jednoho příkazu, jak vidíme níže:

```
dataType[] arrayRefVar = new
dataType[arraySize];
```

Alternativně můžete vytvářet pole následujícím způsobem:

```
dataType[] arrayRefVar = {value0, value1,
..., valuek};
```

K prvkům pole se přistupuje přes **index**. Indexy pole jsou založeny na 0, to znamená, že začínají od 0 a končí u **arrayRefVar.length-1**.

Příklad:

Následující příkaz deklaruje proměnnou pole, myList, vytváří pole o 10 prvcích typu double, a přiřazuje referenci k myList.

```
double[] myList = new double[10];
```

Obrázek reprezentuje pole *myList*. Zde *myList* obsahuje deset hodnot double a indexy jsou od 0 do 9.

Zpracování polí

K prvkům pole přistupujeme přes hranatou závorku, takže budeme-li chtít na první index (tedy index 0) uložit číslo 1, zapíšeme to takto:

```
int[] pole = new int[10];
pole [0] = 1;
```

Při zpracovávání prvků pole často používáme cykly, buď *for* nebo *foreach*, protože všechny prvky v poli jsou stejného typu a velikost pole je známá.

Příklad:

Zde je úplný příklad, který ukazuje, jak vytvořit, inicializovat a zpracovat pole.

```
public class TestArray {

   public static void main(String[] args) {
      double[] myList = {1.9,2.9,3.4,3.5};

      // Vypis vsechny prvky pole
      for (int i=0;i<myList.length;i++) {
         System.out.println(myList[i]
     + " ");
      }
```

```
        // Secist vsechny prvky
        double total = 0;
        for (int i=0;i<myList.length;i++) {
       total += myList[i];
        }
        System.out.println("Soucet je "
       + total);

        // Finding the largest element
        double max = myList[0];
        for (int i=1;i<myList.length;i++) {
           if (myList[i]>max) max=myList[i];
        }
        System.out.println("Max is " + max);
    }
```

```
}
```

Toto by mělo následující výstup:

```
1.9
2.9
3.4
3.5
Total is 11.7
Max is 3.5
```

Ruční naplnění pole

Pole můžeme naplnit i „ručně“ a nemusíme přitom zapisovat index po indexu. Použijeme k tomu složených závorek a prvky oddělujeme čárkou:

```
String[] simpsonovi = {"Homer", "Marge",
"Bart", "Lisa", "Meggie"};
```

Podobně lze “ručně” plnit i číselná a jiná pole.

Cykly foreach

JDK 1.5 zavedl nový cyklus známý jako cyklus *foreach* nebo pokročilá smyčka *for*, který umožňuje postupně projít celé pole bez použití indexové proměnné.

Příklad:

Následující kód zobrazuje veškeré prvky pole myList:

```
public class TestArray {

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

```
        double[] myList = {1.9,2.9,3.4,3.5};

        // Vypiš všechny prvky v poli
        for (double element: myList) {
           System.out.println(element);
        }
    }
}
```

Toto by vyprodukovalo následující výstup:

```
1.9
2.9
3.4
3.5
```

Předávání polí metodám

Metodám můžete předávat pole úplně stejně, jako jim předáváte hodnoty primitivních typů.

```
public static void printArray(int[] array)
{
  for (int i = 0; i < array.length; i++) {
    System.out.print(array[i] + " ");
  }
}
```

Metodu můžete také spustit předáním pole. Například následující příkaz zavolá metodu printArray, aby zobrazila 3. 1, 2, 6, 4 a 2.

```
printArray(new int[]{3, 1, 2, 6, 4, 2});
```

Vracení pole z metody

Metoda může rovněž navrátit pole. Například metoda níže vrací pole, které vzniklo obrácením pole na vstupu:

```
public static int[] reverse(int[] list) {
  int[] result = new int[list.length];

  for (int i=0,j=result.length-1;
       i<list.length;i++,j--) {
    result[j] = list[i];
  }
  return result;
```

```
}
```

Třída Arrays

Podobně jako primitivní typy, také pole mají svou obalovací třídu **Arrays**. Třída *java.util.Arrays* obsahuje nejrůznější statické metody pro řazení a prohledávání polí, porovnávání polí a dosazování prvků do polí. Tyto metody jsou přetěžované pro všechny primitivní typy.

SN	Metody s popisem
1	**public static int binarySearch(Object[] a, Object key)** Prohledává specifikované pole Objektu Object (Byte, Int , double, atd.) na specifikovanou hodnotu za použití

	binárního prohledávacího algoritmu. Pole musí být před zavoláním této metody seřazeno. Je navracen index hledaného klíče, pokud ho seznam obsahuje.
2	**public static boolean equals(long[] a, long[] a2)** Navrací true, pokud jsou obě specifikovaná long pole shodná. Dvě pole jsou považována za shodná, pokud obě obsahují stejný počet prvků a veškeré korespondující páry prvků v obou polích jsou shodné. Navrací true, pokud jsou pole shodná. Stejnou metodu by šlo použít pro veškeré primitivní datové typy (Byte, short, Int, atd.).
3	**public static void fill(int[] a, int val)** Přiřadí specifikovanou hodnotu int ke každému prvku z pole intů. Stejnou metodu by šlo použít pro veškeré ostatní primitivní datové typy (Byte, short, Int, atd.)

4	**public static void sort(Object[] a)** Vzestupně řadí specifikované pole pole přirozeného řazení svých prvků. Stejnou metodu by šlo použít pro všechny ostatní datové typy (Byte, short, Int, atd.)

Příklad

Umožníme uživateli zadat jméno **x** a poté zkontrolujeme, zda patří do rodiny Simpsonových.

```
Scanner sc = new Scanner(System.in,
   "Windows-1250");
```

```
String[] simpsonovi =
   {"Homer", "Marge", "Bart", "Lisa",
"Meggie"};

System.out.println(
   "Zadej jméno
    z rodiny Simpsonů): ");
String x = sc.nextLine();

Arrays.sort(simpsonovi);

int pozice = Arrays.binarySearch(
   simpsonovi, x);

if (pozice >= 0)
```

```
        System.out.println(
    "Je to Simpson!");
else
        System.out.println(
    "Není to Simpson!");
```

Pole s proměnlivou délkou

Říkali jsme si, že délku pole můžeme definovat i za běhu programu, pojďme si to zkusit na příkladu, který bude počítat průměr z proměnlivého počtu čísel.

Rozumí se tím samozřejmě, že sice můžeme zadat libovolný (kladný) počet čísel, ale jakmile jsme ho jednou zadali, tak už ho nemůžeme změnit.

Příklad

```
Scanner sc = new Scanner(System.in,
   "Windows-1250");

System.out.println("
   Ahoj, spočítám ti průměr známek.
   Kolik známek zadáš? ");

int pocet =
   Integer.parseInt
      (sc.nextLine());
```

```
int[]cisla
   = new int[pocet];

for (int i=0;i<pocet; i++){
    System.out.printf("Zadejte %d. číslo: ",
    i + 1);
         cisla [i] =
Integer.parseInt(sc.nextLine());
}

// spočítání průměru
int soucet = 0;
for (int i: cisla)
         soucet += i;
```

```
float prumer = soucet / (float)
cisla.length;

System.out.printf("Průměr tvých známek je:
%f", prumer);
```

Tento příklad by šel samozřejmě napsat i bez použití pole, ale co kdybychom chtěli spočítat např. medián? Nebo např. vypsat zadaná čísla pozpátku? To už by bez pole nešlo.

Takhle máme k dispozici v poli původní hodnoty a můžeme s nimi neomezeně a jednoduše pracovat.

Pozor!

U výpočtu průměru si všimněte, že při dělení je před jedním operandem napsáno **(float)**.

Tím říkáme, že chceme dělit neceločíselně.

Jinak matematické operace probíhají tak, jak to odpovídá typům čísel. Takže dělení celých čísel 3 / 2 dostaneme výsledek 1, zatímco při při 3 / 2.0F dostaneme výsledek 1.5. Zde je princip stejný.

Jednoduché příklady

Ukážeme si několik jsenoduchých příkladů na procvičení stringů a polí.

Příklad – Substring

Vrátí podřetězec od dané pozice do konce řetězce. Můžeme zadat druhý parametr, kterým je délka podřetězce.

```
System.out.println("Kdo se směje naposled, ten je admin.".substring(13, 21));
```

Výstup:

```
naposled
```

Příklad – CompareTo

Umožňuje porovnat dva řetězce podle abecedy. Vrací -1, pokud je řetězec abecedně před řetězcem v parametru, 0 pokud jsou stejné a +1 pokud je abecedně za řetězcem v parametru:

```
System.out.println("akát".compareTo("blýska vice"));
```

Výstup:

```
-1
```

Příklad – Kódování znaků, ASCII tabulka

Kódování znaků

V éře operačního systému MS DOS prakticky nebyla jiná možnost, jak zaznamenávat text, než v kódování podle ASCII tabulky.

Jednotlivé znaky byly uloženy jako čísla typu byte, tedy s rozsahem hodnot od 0 do 255.

ASCII tabulka měla také 255 znaků a každému ASCII kódu (číselnému kódu) přiřazovala jeden znak.

Hlavní výhoda byla v tom, že znaky jsou uloženy v tabulce za sebou, podle abecedy. Např. na pozici 97 nalezneme "a", 98 "b" a podobně. Podobně je to s čísly.

Diakritické znaky tam jsou bohužel rozházené.

Hlavní nevýhoda je, že do tabulky se jednoduše nevešly všechny znaky všech národních abeced.

Nyní se proto používá unicode (UTF8) kódování, kde jsou znaky reprezentovány trochu jiným způsobem.

V Javě ale máme stále možnost pracovat s ASCII hodnotami jednotlivých znaků.

Číselná hodnota znaků

Zkusíme si nyní převést znak do jeho ASCII hodnoty a naopak podle ASCII hodnoty daný znak vytvořit.

Potřebujeme k tomu tzv. **přetypování**.

Přetypování např. **(int)c** znamená, že kompilátor má na **c**, přestože to je znak, pohlížet jako by to bylo číslo.

Podobně, **(char)i** je znak.

```
char c;
// znak
```

```
int i;
// ordinální (ASCII)
// hodnota znaku

// převedeme znak
// na jeho ASCII hodnotu
c = 'a';
i = (int)c;
System.out.printf(
"Znak %c jsme převedli na ASCII hodnotu
%d\n", c, i);

// Převedeme ASCII hodnotu
// na znak
i = 98;
```

```
c = (char)i;
System.out.printf(
"ASCII hodnotu %d jsme převedli na znak
%c", i, c);
```

Příklad - Cézarova šifra

Vytvoříme si jednoduchý program pro na šifrování textu. Pokud jste někdy slyšeli o Cézarově šifře, bude to přesně to, co si zde naprogramujeme.

Šifrování textu spočívá v posouvání znaku v abecedě o určitý, pevně stanovený počet znaků. Například slovo "ahoj" se s posunem textu o 1 přeloží jako "bipk". Posun umožníme uživateli vybrat.

Budeme potřebovat proměnné pro původní text, zašifrovanou zprávu a pro posun. Dále cyklus projíždějící jednotlivé znaky a výpis zašifrované zprávy.

Zprávu si necháme zapsanou napevno v kódu, abychom ji nemuseli při každém spuštění programu psát. Kdo chce, může si text zprávy pokaždé načíst pomocí **sc.nextLine()**.

Šifra nepočítá s diakritikou, mezerami ani s interpunkčními znaménky.

Diakritiku zakážeme a budeme předpokládat, že ji uživatel nebude zadávat. Ideálně bychom měli diakritiku před šifrováním odstranit, stejně tak cokoli kromě písmen.

Kostra programu bude vypadat následovně:

```
// inicializace proměnných
String s =
"cernediryjsoutamkdebuhdelilnulou
";
System.out.printf("Původní zpráva: %s\n",
s);
String zprava = "";
int posun = 1;

// cyklus projíždějící jednotlivé znaky

for (char c : s.toCharArray())
{
   // zpracování
```

```
}

// výpis

System.out.printf("Zašifrovaná zpráva
: %s\n", zprava
);
```

Nyní se přesuneme dovnitř cyklu. Převedeme znak c na ASCII hodnotu (neboli ordinální hodnotu), tuto hodnotu zvýšíme o posun a převedeme zpět na znak. Tento znak nakonec připojíme k výsledné zprávě.

```
int i = (int)c;
i += posun;
char znak = (char)i;
zprava += znak;
```

Výsledný program má vadu na kráse. Zkusme si zadat vyšší posun nebo napsat slovo "zebra". Vidíme, že znaky mohou po "z" přetéct do ASCII hodnot dalších znaků.

Uzavřeme znaky do kruhu tak, aby posun plynule po "z" přešel opět k "a" a dále. Postačí nám k tomu operátor **%** (modulo), zbytek po celočíselném dělení.

Kód bude nyní vypadat takto:

```
int i = (int)c;
i = (posun+i) mod 26;
```

```
char znak = (char)i;
zprava += znak;
```

Příklad - Split, Morseova abeceda

Pojďme si opět připravit strukturu programu. Budeme potřebovat 2 řetězce se zprávou, jeden se zprávou v Morseově abecedě, druhý zatím prázdný, do kterého budeme ukládat výsledek našeho snažení. Dále budeme jako v případě samohlásek potřebovat nějaký vzor písmen. K písmenům samozřejmě vzor jejich znaku v Morseovce. Písmena můžeme opět uložit do pouhého Stringu, protože mají jen jeden znak. Znaky Morseovy abecedy mají již znaků více, ty musíme dát do

pole. Struktura našeho programu by nyní mohla vypadat následovně:

```
// řetězec, který chceme dekódovat
String s = ".. ... .-.. .- -. -.. ... ---
..-. -";
System.out.printf("Původní zpráva: %s\n",
s);
// řetězec s dekódovanou zprávou
String zprava = "";

// vzorová pole
String abecedniZnaky =
"abcdefghijklmnopqrstuvwxyz";
String[]morseovyZnaky = {".-", "-...", "-.-
.", "-..", ".", "..-.", "--.", "....",
```

```
".", ".---", "-.-", ".-..", "--", "-.", "-
--", ".--.", "--.-", ".-.", "...", "-",
"..-",
"...-", ".--", "-..-", "-.--", "--.."};
```

Můžete si potom přidat další znaky jako čísla a interpunkční znaménka, my je zde vynecháme. Nyní si řetězec s rozbijeme metodou **split()** na pole řetězců, obsahujících jednotlivé znaky Morseovky. Splitovat budeme podle znaku mezery. Pole následně proiterujeme cyklem **foreach**:

```
// rozbití řetězce na znaky Morseovky
String[]znaky = s.split(" ");

// iterace znaků Morseovky
for (String morseuvZnak : znaky)
```

```
{


}
```

Ideálně bychom se měli nějak vypořádat s případy, kde uživatel zadá např. více mezer mezi znaky (to uživatelé rádi dělají). *Split()* poté vytvoří o jeden řetězec v poli více, který bude prázdný. Ten bychom měli poté v cyklu detekovat a ignorovat, my se s tím v tutoriálu nebudeme zabývat.

V cyklu se pokusíme najít aktuálně čtený znak Morseovky v poli *morseovyZnaky*. Bude nás zajímat jeho index, protože když se podíváme na ten samý index v poli *abecedniZnaky*, bude tam odpovídající písmeno. To je samozřejmě z toho důvodu, že obě

pole mají v sobě stejné znaky řazeny dle abecedy. Umístěme do těla cyklu následující kód:

```
char abecedniZnak = '?';

int index = -1;
for (int i = 0; i < morseovyZnaky.length;
i++)
{
        if (morseovyZnaky
[i].equals(morseuvZnak))
                index = i;
}

if (index >= 0) // znak nalezen
```

```
        abecedniZnak =
abecedniZnaky.charAt(index);
zprava += abecedniZnak;
```

Kód nejprve do abecedního znaku uloží '?', protože se může stát, že znak v naší sadě nemáme. Následně se pokusíme zjistit jeho index. Pole v Javě bohužel nemá metodu *indexOf()* a zatím nechci zabíhat do složitějších datových struktur. Napíšeme si tedy vyhledání Stringu v poli sami, bude to docela jednoduché.

Nejprve nastavíme index na -1, protože nevíme, jestli pole daný String (Morseův znak) obsahuje. Následně pole projedeme cyklem a budeme kontrolovat jednotlivé Stringy s naším stringem (znakem). Již víme, že musíme k porovnání dvou

řetězců použít metodu equals. Pokud řetězce souhlasí, uložíme si aktuální index.

Pokud jsme znak našli (index >= 0), dosadíme do abecedniZnak znak z abecedních znaků pod tímto indexem. Nakonec znak připojíme ke zprávě. Operátor += nahrazuje zprava = zprava + abecedniZnak.

Závěrem samozřejmě zprávu vypíšeme:

```
System.out.printf("Dekódovaná zpráva:
%s\n", zprava);
```

Třída Math

Základní matematické funkce jsou v Javě obsaženy v třídě Math. Naimportujeme ji příkazem

```
import java.lang.Math;
```

Třída nám poskytuje dvě základní konstanty: **PI** a **E**. PI je pochopitělně číslo π (3.1415...) a E je Eulerovo číslo, tedy základ přirozeného logaritmu (2.7182...). Asi je jasné, jak se s třídou pracuje, ale pro jistotu si na ukázku konstanty vypišme do konzole:

```
System.out.printf("Pí: %f \ne: %f",
Math.PI, Math.E);
```

Vidíme, že vše voláme na třídě Math. Na kódu není nic moc zajímavého kromě toho, že jsme v textovém řetězci použili speciální znak \n, který způsobí odřádkování.

min, max

Obě funkce berou jako parametr dvě čísla libovolného datového typu. Funkce min() vrátí to menší, funkce max() to větší z nich.

round, ceil, floor

Všechny tři funkce se týkají zaokrouhlování. Round() bere jako parametr desetinné číslo a vrací zaokrouhlené číslo typu double tak, jak to známe ze školy (od 0.5 nahoru, jinak dolů). Ceil() zaokrouhlí vždy nahoru a floor() vždy dolů.

Round() budeme jistě potřebovat často, další funkce jsem prakticky často použil např. při zjišťování počtu stránek při výpisu komentářů v knize návštěv. Když máme 33 příspěvků a na stránce jich je vypsáno 10, budou tedy zabírat 3.3 stránek. Výsledek musíme zaokrouhlit nahoru, protože v reálu stránky budou samozřejmě 4.

abs a signum

Obě metody berou jako parametr číslo libovolného typu; abs() vrátí jeho absolutní hodnotu a signum() vrátí podle znaménka -1, 0 nebo 1 (pro záporné číslo, nulu a kladné číslo).

sin, cos, tan

Klasické goniometrické funkce, jako parametr berou úhel typu double, který považují v radiánech, nikoli ve stupních. Pro konverzi stupňů na radiány stupně vynásobíme * (Math.PI/180). Výstupem je opět double.

acos, asin, atan

Opět klasické cyklometrické funkce (arkus funkce), které podle hodnoty goniometrické funkce vrátí daný úhel. Parametrem je hodnota v double, výstupem úhel v radiánech (také double). Pokud si přejeme mít úhel ve stupních, vydělíme radiány / (180 / Math.PI).

pow a sqrt

Pow() bere dva parametry typu double, první je základ mocniny a druhý exponent. Pokud bychom tedy chtěli spočíst např. 2^3, kód by byl následující:

```
System.out.println(Math.pow(2, 3));
```

Sqrt je zkratka ze square root a vrátí tedy druhou odmocninu z daného čísla typu double. Obě funkce vrací výsledek jako double.

exp, log, log10

Exp vrací Eulerovo číslo, umocněné na daný exponent. Log vrací přirozený logaritmus daného čísla. Log10 vrací potom dekadický logaritmus daného čísla.

V seznamu metod nápadně chybí libovolná odmocnina. My ji však dokážeme spočítat i na základě funkcí, které Math poskytuje.

Víme, že platí: 3. odm. z 8 = 8^(1/3). Můžeme tedy napsat:

```
System.out.println(Math.pow(8, (1.0/3.0));
```

POZOR! Je velmi důležité, abychom při dělení napsali alespoň jedno číslo s desetinnou tečkou, jinak bude Java předpokládat celočíselné dělení a výsledkem by v tomto případě bylo 80 = 1.

Rekurze

Co je to rekurze?

Metoda, která volá sama sebe, se nazývá rekurzivní. Takže, když metoda volá sama sebe, tomu se říká rekurze. Klíčovou složkou rekurzivní metody je příkaz, který provádí volání sebe sama.

K čemu je rekurze dobrá?

Rekurze se dá jednoduše použít při počítání faktoriálu. Pokud nevíte co je to faktoriál, tak si to teď vysvětlíme. Faktoriál z čísla N je součin všech celých čísel mezi 1čkou a N. Faktoriál z čísla 5 je 1 * 2 * 3 * 4 * 5 = 120. Takže faktoriál z čísla 5 je 120.

Bez rekurze byste následující příklad řešili asi cyklem. Podívejme se nejdříve na to, jak se to dá řešit bez rekurze:

hlavní třída Program.java

```
public class Program {
    public static void main(String args[]){
        // vytvořím nový objekt třídy
Faktorial
        Faktorial f = new Faktorial ();
        // vypíšu faktoriál z čísla 5
        System.out.println(f. faktorial
(5));
    }
}
```

Třída Faktorial.java

```
public class Faktorial {
    // Vytvořím veřejnou metodu, která
vrací Integer.
    public int faktorial (int n){
        // N je číslo, z kterého chceme
dostat faktoriál.
        int i, vysledek = 1;
        // součin všech čísel mezi 1 a N
        for(i = 1; i <= n;i++)
            vysledek = vysledek * i;
        // vrátím faktoriál z čísla N
        return vysledek;
    }
}
```

Jak jste si asi všimli, veřejná metoda faktoriál ve třídě Faktorial nám vrátila součin všech čísel od 1 do 5 (tedy N). Teď se podíváme na stejný příklad s rekurzí.

Hlavní třída Program.java, je stejná jako ta předchozí.

```
public class Program {

    public static void main(String args[]){
        Faktorial f = new Faktorial();
        System.out.println(f.faktorial(5));
    }
}
```

Třída Faktorial.java, v které už je použita rekurze.

```
public class Faktorial {
```

```
    public int faktorial(int n){
        int vysledek;
        if(n == 1)
            return 1;
        vysledek = faktorial(n-1) * n;
        return vysledek;
    }
}
```

Jak vidíme, metoda volá sama sebe `vysledek = faktorial(n-1) * n;` a tudíž můžeme s klidem říct, že je rekurzivní. Je o dost jednodušší, než kdybychom použili cyklus. Při zavolání metody faktorial s argumentem 1 vrátí metoda jedničku. V opačném případě vrátí součin `faktorial(n-1)`

– `n`. Pro vyhodnocení tohoto výrazu, se zavolá metoda *faktorial()* s hodnotou n-1. Tento proces se opakuje, dokud se n nerovná 1 a volání metody se nezačnou vracet. Například při volání výpočtu faktoriálu z čísla 2 způsobí první volání metody *faktorial()* provedení druhého volání s argumentem 1. Toto volání vrátí hodnotu 1, která se následně vynásobí číslem 2. Odpověď je tedy 2.

Fibonacciho posloupnost

Další ukázkový příklad rekurze je Fibonacciho posloupnost. Jeho složitost je O(2n).

```
public class Program {

    public static void main(String args[]){
        Fibonacci fib = new Fibonacci();

System.out.println(fib.fibonaci(5));
    }
}
```

Fibonacci.java

```
public class Fibonacci {

    public int fibonaci(int n)
    {
        if (n == 1) return 1;
```

```
        else return fibonaci(n-1) +
fibonaci(n-2);
    }
}
```

Program funguje, ale už z principu je velmi pomalý.

Mimochodem, existuje ještě jiné řešení, které má složitost O(n).

```
public class Fibonacci {
    private List<Integer> list = new
ArrayList<Integer>();
    public int fibonaci(int n)
    {   list.add(n);
        if (n == 1) return 1;
        if(!list.contains(n-1)) return
fibonaci(n - 1);
```

```
        if(!list.contains(n-2)) return
fibonaci(n - 2);
        return 0;
    }
}
```

V podstatě šlo o to, vytvořit seznam a ukládat do něj mezivýsledky. Ale protože ještě neznáme konstrukci

```
List<Integer>
```

tak se tímto řešením zatím nebudeme zabývat, podrobnosti nalezneme až v kapitole o kolekcích.

Obsah